UNIVERSITÀ DEGLI STUDI DI FIRENZE
Scuola di Scienze Matematiche Fisiche e Naturali
Corso di Laurea Magistrale in Informatica

B-SPLINE METHODS FOR THE DESIGN OF SMOOTH SPATIAL PATHS WITH OBSTACLE AVOIDANCE

METODI B-SPLINE PER IL DISEGNO DI PERCORSI REGOLARI IN AMBIENTI TRIDIMENSIONALI CONTENENTI OSTACOLI

Tesi di Laurea Magistrale in Informatica

Relatore: *Alessandra Sestini* Correlatore: *Carlotta Giannelli*

Candidato: STEFANO MARTINA

Luglio 2016, Anno accademico *2015/16*

Stefano Martina (`stefano.martina@stud.unifi.it`): *B-Spline methods for the design of smooth spatial paths with obstacle avoidance,* Corso di Laurea Magistrale in Informatica . © Copyright 2016 Stefano Martina - this work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License .

ABSTRACT

Path planning problem consists in finding an interpolating curve between two points in a scene with obstacles. It has significant applications in robotics and scientific visualization. It is important to find a curve with certain qualities of smoothing, thus we focus on the curve fairing. Furthermore, for the representation, we use B-spline curves that are an affirmed standard in Computer-Aided Design (CAD) and Computer-Aided Geometric Design (CAGD).

We design different algorithms to solve the problem and we present their complexity analysis. The resulting work is highly interdisciplinary: we address different approaches, analytical and stochastic.

We realize an application in Python using Visualization Tool Kit (VTK) for the visualization that implements the presented algorithms. Finally, we systematically test the application with different scenarios.

Dedicated to a future of elevation for the human condition.

I hear and I forget.
I see and I remember.
I do and I understand.

— Confucius

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my coordinator Prof. Alessandra Sestini and my advisor Prof. Carlotta Giannelli for their patience and the priceless revision work. I want to thank them also for introducing me to this research area and for their guidance through the development of this project.

I also would like to thank all the professors for contributing, with their passion, to make me a better person and for instilling in me the passion for knowledge. Among them, a special mention goes to Prof. Pierluigi Crescenzi for the advice that he gave me for complexity analysis, Prof. Maria Cecilia Verri for the advice on the algorithms, Prof. Gregorio Landi and Prof. Franco Bagnoli for introducing me to stochastic methods and Prof. Enrico Vicario for the many opportunities.

I would like to thank all my friends for being life companions and for all the adventures that we had and we will continue to have.

A special loving thank goes to my parents Mauro and Rosa, and to my brother Simone, for patiently contributing to sustain me all this years, for tangible and intangible needs.

Finally, last but not least, I want to thank Federica, the woman of my life, for making the world a worthier place to live in and for the unconditional love that she gives to me everyday (and also for helping me to make this text more readable).

Stefano Martina

CONTENTS

1

INTRODUCTION

The design of *motion planning* strategies plays a fundamental role in different applications, from robotics to scientific visualization. *Path planning* problem is more specific, it consists in identifying paths that do not intersect any obstacle.

In this project we are interested in generating smooth paths. Smoothness is a desirable property that is frequently presented in literature for the planar case. Consider, for instance, the papers [28], [18], [27] and [16]. The first considers an interesting combined approach to the problem: analytical to find a smooth curve, and stochastic to locate a desired cusp on it. On the second, they concentrate in finding a curvature bounded path starting from a Voronoi Diagram (VD) constructed accordingly to the environment. The third work focuses on the process of transforming an existing polyline path in a smooth curve. In the last, they used Pythagorean Hodograph (PH) curves that have interesting features for the Computer-Aided Manufacturing (CAM) field [13].

In regards of spatial path planning, the smoothing problem is less covered in literature. For instance, in [20] it is not clear how a smooth path is obtained from the initial polynomial chain. In [41] the smoothness is considered, but the method is not optimal because it consists in alternating smoothing and obstacle-checking phases until an admissible solution is obtained. Other works, like [2] and [25], use stochastic methods to achieve smoothness.

B-spline curves are a reference standard in Computer-Aided Design (CAD) and Computer-Aided Geometric Design (CAGD) [21][22][12][15]. Thus, we decide to develop a 3D path planning application using this kind of curves, as [41] do. However, as earlier mentioned, [41] uses a *try and check* approach to the curve smoothing, we present a method that finds a smooth curve on the first attempt instead.

The considered topic is highly interdisciplinary. In fact we integrate in this project an extended set of competencies acquired during the courses.

We apply notions of *linear algebra* for the collision checks; *numerical analysis* for the curves design; *computational geometry*, *graph theory*, *probability* and *algorithm theory* for the algorithms design; and, finally, *theoretical computer science* for cost analysis.

We focus on finding a trade-off between having a short curve, a smooth curve, and keeping the time complexity low. Different solutions are explored, with different qualitative effects on the curve.

Regarding the scene representation, different kinds of polyhedral obstacle are considered.

A framework in Python is developed using Visualization Tool Kit (VTK) for the graphic output. We use a roadmap method based on Voronoi Diagrams (VDs) to create a graph (details in Section 5.1.1) that is the base structure for the project. Using such structure, three different solutions are presented.

1. The first method benefits from the Convex Hull Property (CHP) of B-spline curves (Section 3.2.1). A transformation is applied on the graph such that every path in it can be used as a control polygon for an obstacle-free curve (Section 5.1.2). Therefore, the algorithm selects the shortest path in the transformed graph and builds the curve on it (Section 5.2.1).

2. The second method still benefits from the CHP, but it picks the shortest path directly in the base graph. If violations of the CHP emerge in it, then rectification measures are taken (Section 5.2.2).

3. The third method uses a probabilistic approach. Starting from the shortest path in the original graph, it performs a simulated annealing optimization (Section 3.4.3) that converges in a state where we have an optimal trade-off between having a short curve, and low curvature and torsion peaks (Section 5.6).

This document consists of three parts. The first (Part I) is dedicated to the state of the art: we provide a survey of different topics and algorithms related to *motion planning*.

The second part (Part II) is committed to describing all the different parts of the algorithm. In detail:

- Chapter 3 gives to the reader all the necessary notions to understand the rest of the chapter;

- Chapter 4 describes how the environment and the resulting curve are represented;

- Finally in Chapter 5 we describe how to obtain the basic structures (Section 5.1), how to avoid the obstacles using the three methods described before (Section 5.2), and how to improve the obtained curve simplifying the control polygon (Section 5.5), increasing the curve degree (Section 5.3) and changing the B-spline knot vector (Section 5.4).

The third part (Part III) describes the instruments used to implement the algorithms (Chapter 6) and presents a series of tests with different scenes and configurations (Chapter 7) with their conclusions (Chapter 8).

In conclusion, Appendix B contains all the source code of the application.

Part I

STATE OF THE ART

2

MOTION PLANNING

The problem of *motion planning* consists in determining a set of low level tasks, given an high level goal to be fulfilled [7]. For instance, a classic motion planning problem is the *piano movers'* problem that involves the motion of a free flying rigid body in the 3-dimensional space from a start to a goal configuration by applying translations and rotations and by avoiding collisions with a set of obstacles [7][26]. Motion planning finds applications in different areas, like robotics, Unmanned Aerial Vehicles (UAVs) [17] and autonomous vehicles [31]. These are the most famous applications but it finds utilization also in other less common areas like motion of digital actors or molecule design [7].

Initially, the term motion planning referred only to the translations and rotations of objects, ignoring the dynamics of them, but lately research in this field started considering also the physical constraints of the object to be moved [26]. Usually, the term *trajectory planning* is used to refer to the problem of taking the path produced by a motion planning algorithm and determine the time law for moving a robot on it by respecting its mechanical constraints [26].

An important concept for motion planning problems is the *state space*, that can have different dimensions, one for each degree of freedom of the object to move. It can be a discrete or a continuous space [26]. We can call the space state $\mathbb{S}$ and, considering that there are obstacles or constraints on the scene, we can call $\mathbb{S}_{\mathsf{free}} \subseteq \mathbb{S}$ the portion of the state space such that all its configurations are admissible. On this space state we have special states $\mathbf{s} \in \mathbb{S}$ and $\boldsymbol{e} \in \mathbb{S}$ for the desired *start* and *end* configurations, respectively.

Another concept is the geometric design of the scene and the actor. The obstacles can be represented as convex polygons/polyhedrons, or also as more complex shapes [26].

Furthermore, it is important to define the possible admissible transformations of the body, if it is possible only to translate and rotate it or if its motion is composed of rigid kinematic chains or trees or if it is even possible to have not rigid transformations (flexible materials) [26].

2.1 PROBLEM TYPES

Many different problems related to motion planning have been introduced in literature. In this section we present a short survey of the most relevant problems in order of increasing complexity. Refer to [17] for details.

POINT VEHICLE The body of the object to be moved is represented as a point in the space. Thus the state space $\mathbb{S}$ consists in the euclidean space $\mathbb{E}^2$ if we consider land vehicles or $\mathbb{E}^3$ if we consider aerial vehicles.

POINT VEHICLE WITH DIFFERENTIAL CONSTRAINTS This problem extends the point vehicle's problem by adding the constraints of the physical dynamic. For instance, constraints on acceleration, velocity, curvature, etc... when we want to model a real vehicle (whose shape is still approximated with a point).

JOGGER'S PROBLEM This kind of problems concerns the dynamic of a jogger that has a limited field of view. Consequently, in this case, we do not have a complete view of the scene and the path is updated as soon as the knowledge of the scene increases.

BUG'S PROBLEM This problem is an extreme case of the jogger's problem with a null field of view. Thus the scene updating can be done only when an obstacle is touched.

WEIGHTED REGIONS' PROBLEM This problem considers some regions of the space as more desirable than others, rather than contemplate completely obstructive obstacles. For instance, this is the case of finding a path in an off-road environment where the vehicle can move faster on certain terrains and slower over different configurations.

MOVER'S PROBLEM The vehicle is modeled as a rigid body, thus, we need to add the dimensions for the spatial rotation of the body to the state space.

GENERAL VEHICLE WITH DIFFERENTIAL CONSTRAINTS This problem combines the *mover's problem* and the *point vehicle with differential constraints* by adding to the mover's problem the physical constraints on the motion dynamic.

TIME VARYING ENVIRONMENTS These problems regards moving obstacles.

MULTIPLE MOVERS This problem considers more than one vehicle. We need to manage different paths and the problem of avoiding possible collisions between different movers. As a matter of fact, we have to avoid collisions between the paths followed by different movers only if the collision point is reached by the movers simultaneously.

2.2 ALGORITHM TYPES

We can divide the algorithms for motion planning in different types taking into account the specific problem they resolve. The algorithms belonging to a certain type can be further divided in different categories. For more details on the different algorithms see [17] and [7].

2.2.1 *Roadmap methods*

This kind of algorithms reduces the problem of motion planning to graph search algorithms. The state space is approximated with a certain graph in order to find a solution in terms of a polygonal chain.

2.2.1.1 *Visibility graph*

The visibility graph is one of the most known roadmap methods. The nodes of the graph correspond to the vertices of each polygonal obstacles in the considered scenario. The edges of the graph correspond to linear segments between pair of vertices that do not intersect any obstacle. The Dijkstra's algorithm is then usually considered to compute the *shortest path* between two vertices of the graph [10]. Note that the shortest path associated to the visibility graph in a planar configuration is the absolute shortest path from the start to the goal position with respect to the considered scenario, see e.g., [8]. While this method finds the optimal solution (with respect to a *distance* criterion) in the planar case, it does not properly scale in a 3-dimensional setting.

2.2.1.2 *Edge sample visibility graph*

The edge sample visibility graph is an extension of the visibility graph method to the 3-dimensional case. The main idea consists in distributing a discrete set of points along the edges of the obstacles by considering a certain density. The visibility graph and the related shortest path of this configuration are then computed, but the corresponding solution is not as optimal as in the planar case.

2.2.1.3 *Voronoi roadmap*

This method builds a graph that is kept equidistant to the obstacles, using VDs as base method for constructing it. We discuss VDs in detail in Section 3.3 and Voronoi roadmap method in Section 5.1.

2.2.1.4 *Silhouette method*

This method was developed by Canny [6]. It is not useful for practical uses but just for proving algorithmic bounds because it is proven to be complete in any dimension. It works sweeping the space with a line (plane in 3-dimensional space) perpendicular to the segment between $\mathbf{s}$ and $\mathbf{e}$ and building the shape of the obstacles when the sweeping line intersects them.

2.2.2 *Cell decomposition*

This method decomposes $\mathcal{S}_{\mathrm{free}}$ in smaller convex polygons - i.e. trapezoids cylinders or balls - that are connected by a graph, then searches a solution in such graph. A cell decomposition method can be exact or approximate, the former kind operates occupying all $\mathcal{S}_{\mathrm{free}}$ with the graph structure, the latter one can occupy also portions of $\mathcal{S} \setminus \mathcal{S}_{\mathrm{free}}$ or all $\mathcal{S}$. Then the various polygons are labelled as obstacle-empty, inside obstacle or partially occupied by obstacles.

2.2.3 *Potential field methods*

This kind of methods operates assigning a potential field on every region of the space, the lowest potential is assigned to the goal point $\mathbf{e}$ and a high potential value is assigned to the obstacles. Then the path is calculated as a trajectory of a particle that reacts to those potentials, it is repelled by the obstacles and attracted by the end point.

2.2.4 *Probabilistic approaches*

This kind of methods uses probabilistic techniques for exploring the space of solutions and finding a good approximation of the optimal solution. In our project we provide also a mixed roadmap-probabilistic method, see Section 3.4 and Section 5.6 for further details.

2.2.5 *Rapidly-expanding Random Tree (RRT)*

This method operates by doing a stochastic search, starting from the reference frame of the object to be moved and expanding a tree through the random sampling of the state space.

2.2.6 *Decoupled trajectory planning*

This kind of algorithms operates in a two-step way. First a discrete path through the state space is found, then the path is modified to adapt it to the dynamics constraints - i.e. the trajectory is constructed.

2.2.7 *Mathematical programming*

This method manages the trajectory planning problem as a numerical optimization problem, using methods like nonlinear programming to find the optimal solution.

2.3 PATH PLANNING

In our project we concentrate on a subset of the motion planning problem, the *path planning* problem that consists [7] in determining a parametric curve

$$\mathbf{C} \; : \; [a, b] \subset \mathbb{R} \; \rightarrow \; \mathbb{S}$$

such that $\mathbf{C}(a) = \mathbf{s}$ coincides with the desired starting configuration, $\mathbf{C}(b) = \mathbf{e}$ the desired end configuration and the image of $\mathbf{C}$ is a subset of $\mathbb{S}_{\mathsf{free}}$, in other words

$$\mathbf{C}(u) \in \mathbb{S}_{\mathsf{free}} \quad \forall u \in [a, b].$$

In principle the space of the states $\mathbb{S}$ can be of any dimension, for instance if we focus on the piano movers' problem the state is composed by 3 dimensions for the position and other 3 dimensions for the rotation of the object [26]. Also the curve $\mathbf{C}$ can be parameterized in any way.

In this project we concentrate on the problem of path planning where the state space is $\mathbb{S} = \mathbb{E}^3$ and the curve is parameterized in $[0, 1]$. Thus we find a curve from one point $\mathbf{s} \in \mathbb{E}^3$ to another point $\mathbf{e} \in \mathbb{E}^3$ avoiding obstacles. The object that we move is considered just as a point.

Part II

PROJECT

3

PREREQUISITES

3.1 SPLINES AND B-SPLINES

A *spline* is a piecewise polynomial function with prescribed regularity on its domain.

More formally we define a spline [9][12][35][34]

$$s : [a, b] \subset \mathbb{R} \to \mathbb{R}$$

as follows. We have a partition of that interval defined by the *breakpoints*

$$\tau = \{\tau_0, \dots, \tau_\ell\}$$

such that $a = \tau_0 < \tau_1 < \cdots < \tau_{\ell-1} < \tau_\ell = b$. Such breakpoints define ℓ intervals

$$I_i = \begin{cases} [\tau_i, \tau_{i+1}) & \text{if } i = 0, \dots, \ell - 2 \\ [\tau_i, \tau_{i+1}] & \text{if } i = \ell - 1. \end{cases}$$

It is possible to define the following spaces:

PIECEWISE POLYNOMIAL FUNCTIONS SPACE $P_{m,\tau}$ is the space of the functions that are polynomials of maximum degree m in each interval I_i of the partition, formally:

$$P_{m,\tau} = \{f : [a, b] \to \mathbb{R} \mid \exists p_0 \dots p_{\ell-1} \in \Pi_m \text{ such that } f(t) = p(t),\ \forall t \in I_i,\ i = 0 \dots \ell - 1\}$$

where Π_m is the space of the polynomials of degree $\leqslant m$. The dimension of $P_{m,\tau}$ is

$$\dim(P_{m,\tau}) = \ell(m + 1)$$

because the dimension of Π_m is $m + 1$.

CLASSIC SPLINE SPACE $S_{m,\tau}$ is the space of the piecewise polynomial functions of degree m that have continuity C^{m-1} in the junctions of the intervals, formally:

$$S_{m,\tau} = P_{m,\tau} \cap C^{m-1}[a,b].$$

The dimension of this space is

$$\ell(m+1) - (\ell-1)\cdot m \;=\; \ell + m. \tag{1}$$

GENERALIZED SPLINE SPACE $S_{m,\tau,M}$ is the space of piecewise polynomial functions of degree m with a prescribed regularity at each breakpoint ranging from -1 to $m-1$. The regularity is prescribed by the multiplicity vector

$$M = \{m_1, \dots, m_{\ell-1}\}, \quad m_i \in \mathbb{N}, \quad 1 \leqslant m_i \leqslant m+1$$

as follows,

$$\begin{aligned} S_{m,\tau,M} = \{&f : [a,b] \to \mathbb{R} \mid \exists p_0 \dots p_{\ell-1} \in \Pi_m \text{ such that} \\ &f(t) = p(t),\ \forall t \in I_i,\ i = 0 \dots \ell-1 \text{ and} \\ &\qquad p_{i-1}^{(j)}(\tau_i) = p_i^{(j)}(\tau_i),\ j = 0, \dots, m - m_i,\ i = 1, \dots, \ell-1\}. \end{aligned}$$

The dimension of the space is equal to

$$\dim(S_{m,\tau,M}) = \ell(m+1) - \sum_{i=1}^{\ell-1}(m - m_i + 1) = m + \mu + 1 \qquad (\mu = \sum_{i=1}^{\ell-1} m_i)$$

and is true that

$$\Pi_m \subseteq S_{m,\tau} \subseteq S_{m,\tau,M} \subseteq P_{m,\tau},$$

in particular:

- if $m_i = 1$ for all $i = 1, \dots, \ell-1$, then $S_{m,\tau,M} = S_{m,\tau}$;
- if $m_i = m+1$ for all $i = 1, \dots, \ell-1$, then $S_{m,\tau,M} = P_{m,\tau}$.

3.1.1 *Truncated-powers basis for classic splines*

A truncated power $(t-\tau_i)_+^m$ is defined by

$$(t-\tau_i)_+^m = \begin{cases} 0, & \text{if} \quad t \leqslant \tau_i \\ (t-\tau_i)^m, & \text{otherwise.} \end{cases}$$

It is possible to demonstrate that the functions

$$g_i(t) = (t - \tau_i)_+^m) \;\in S_{m,\tau}, \quad i = 1, \dots, \ell - 1$$

are linearly independents, and that the set

$$1, t, t^2, \dots, t^m, (t - \tau_1)_+^m, \dots, (t - \tau_{\ell-1})_+^m$$

forms a basis for the classic spline space [9]. Then a generic element $s \in S_{m,\tau}$ can be expressed as follows,

$$s(t) = \sum_{i=0}^{m} c_i t^i + \sum_{j=1}^{\ell-1} d_j (t - \tau_j)_+^m \qquad \begin{array}{l} c_i \in \mathbb{R},\ i = 0, \dots, m \\ d_j \in \mathbb{R},\ j = 1, \dots, \ell - 1. \end{array} \tag{2}$$

3.1.2 *B-splines basis for classic splines*

B-splines are a specific basis which can be alternatively used to represent any generalized spline [9][12][35][34]. In this paragraph, however, we consider only their definition to generate the classic spline space $S_{m,\tau}$. Furthermore in some textbooks, for notational convenience, the *order*$=$ $m+1$ is considered.

For defining the B-splines [9] we need to extend the partition vector $\tau = \{\tau_0, \cdots, \tau_\ell\}$ with m knots to the left and m to the right, thus we define a new vector, usually called *extended knot* vector,

$$T = \{t_0, \dots, t_{m-1}, t_m, \dots, t_{n+1}, t_{n+2}, \dots, t_{n+m+1}\}$$

such that

$$t_0 \leqslant \cdots \leqslant t_{m-1} \leqslant \overset{\equiv \tau_0 \equiv a}{t_m} < \cdots < \overset{\equiv \tau_\ell \equiv b}{t_{n+1}} \leqslant t_{n+2} \leqslant \cdots \leqslant t_{n+m+1}.$$

Since τ has $\ell + 1$ elements, we can calculate the value of

$$n = \ell + m - 1.$$

Thus the dimension of $S_{m,\tau}$, for Eq. (1), is

$$\dim(S_{m,\tau}) = \ell + m = n + 1$$

The $n+1$ basis $N_{i,m+1}(t)$ of the B-splines of degree m are defined, for $i = 0, \dots, n$, by the recursive formula:

$$N_{i,1}(t) = \begin{cases} 1, & \text{if} \quad t_i \leqslant t < t_{i+1} \\ 0, & \text{otherwise} \end{cases} \qquad\qquad i = 0, \dots, n+m$$

$$N_{i,r}(t) = \omega_{i,r-1}(t) \cdot N_{i,r-1}(t) \;+\; (1 - \omega_{i+1,r-1}(t)) \cdot N_{i+1,r-1}(t)$$

$$\text{i=0,\dots,n+m+1-3, r=2,\dots,m+1}$$

where

$$\omega_{i,r}(t) = \begin{cases} \frac{t-t_i}{t_{i+r}-t_i}, & \text{if } t_i \neq t_{i+r} \\ 0, & \text{otherwise.} \end{cases}$$

Then any function $s \in S_{m,\tau}$ can be expressed also as a linear combination of B-splines,

$$s(t) = \sum_{i=0}^{n} \nu_i N_{i,m+1}(t) \qquad , \nu_i \in \mathbb{R}, i = 0, \dots, n. \tag{3}$$

3.1.3 *Spline curves*

A *spline curve* in the affine space $\mathbb{E}^d$ is the image of a parametric vector function $\mathbf{S} : [a, b] \to \mathbb{E}^d$ whose components are all splines belonging to a fixed spline space $S_{m,\tau}$, for $d = 3$

$$\mathbf{S}(u) = \begin{bmatrix} x(u) \\ y(u) \\ z(u) \end{bmatrix}. \tag{4}$$

$\mathbf{S}(u)$ can be written as follows in the truncated-powers basis Eq. (2) replacing the coefficients c_i and d_i with points

$$\begin{aligned} \mathbf{S}(u) = \sum_{i=0}^{m} \mathbf{c}_i \cdot t^i + \sum_{j=1}^{\ell-1} \mathbf{d}_j \cdot (u - \tau_j)_+^m \qquad & , \mathbf{c}_i \in \mathbb{E}^d,\ \mathbf{d}_j \in \mathbb{E}^d \\ & i = 0, \dots, m;\ j = 0, \dots, \ell - 1. \end{aligned} \tag{5}$$

This representation is not practical because there isn't an intuitive correlation between the points $\mathbf{c}_i$, $\mathbf{d}_j$ and the curve itself. Moreover the determination of an interpolant to assess argued points in $\mathbb{E}^d$ is not a well conditioned problem if this form is adopted [9]. To overcome those drawbacks the *B-splines basis* is adopted (Section 3.1.2).

We can apply control vertices to a spline expressed with the B-spline basis as in Eq. (3) replacing the coefficients ν_i with points, in this case $\mathbf{S}(u)$ is represented as follows

$$\mathbf{S}(u) = \sum_{i=0}^{n} \boldsymbol{\nu}_i \cdot N_{i,m+1}(u) \qquad , \boldsymbol{\nu}_i \in \mathbb{E}^d,\ i = 0, \dots, n. \tag{6}$$

The representation of Eq. (6) is more convenient than the previous one (Eq. (5)) because the curve $\mathbf{S}(u)$ roughly follows the shape given by the points v_i. Those points are called *control vertices* because they are used to control the curve shape. Conformally the polygon they define is called *control polygon*.

3.2 B-SPLINES CURVES PROPERTIES

In this section we describe some properties of B-spline curves that we use for the development of the project.

3.2.1 *Convex Hull Property (CHP)*

The Convex Hull Property (CHP) states that a B-spline curve $\mathbf{S}(u)$ of order m, defined by the control polygon $v_0, v_1, \ldots, v_n$, is contained inside the union of the convex hulls composed of $m+1$ vertices of the control polygon [12]. If we call $\mathbf{Conv}(w_0, w_1, \ldots, w_j)$ the convex hull of the vertices $w_0, w_1, \ldots, w_j$ then we have

$$\begin{aligned} C_0 &= \mathbf{Conv}(v_0, v_1, \ldots, v_m) \\ C_1 &= \mathbf{Conv}(v_1, v_2, \ldots, v_{m+1}) \\ &\cdots \\ C_{n-m} &= \mathbf{Conv}(v_{n-m}, v_{n-m+1}, \ldots, v_n) \end{aligned}$$

and the area where $\mathbf{S}$ is contained is

$$C = C_0 \cup C_1 \cup \cdots \cup C_{n-m}$$

or in other words must be true

$$\mathbf{S}(u) \cap C = \mathbf{S}(u) \qquad \forall u \in [a, b]$$

whatever is the partition vector.

In Fig. 1 an example of control polygon is visible, together with the region C (in cyan) where an associated quadratic B-spline curve is located.

Note that the CHP holds also in 3-dimensional space - i.e. a quadratic B-spline in 3-dimensional space is contained inside a flat surface composed by the union of triangles. From degree 3 the area where $\mathbf{S}$ is contained is not plane anymore because it is composed by the union of solid polyhedrons.

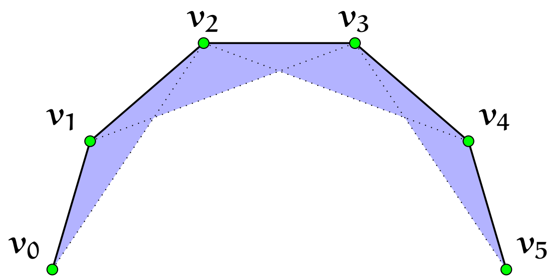

Figure 1.: Convex hull containing B-spline of degree 2

3.2.2 *Aligned vertices*

Because control polygons with sequences of aligned vertices have been adopted in some parts of this project, in this section their specific effect on the curve shape is analyzed.

We can have the following situations:

m ALIGNED CONTROL VERTICES If m control vertices $v_i, \ldots, v_{i+m-1}$ of the control polygon are on the same line then the curve $\mathbf{S}$ touches the segment joining those vertices.

$m+1$ ALIGNED CONTROL VERTICES If $m+1$ control vertices $v_i, \ldots, v_{i+m}$ of the control polygon are on the same line then a polynomial arc of the curve $\mathbf{S}$ lays on the segment joining those vertices.

3.2.3 *Smoothness*

A function is said smooth of class C^d if it is possible to calculate the d-th derivative of it and if such derivative is continue. A function f that is not continue is said to be of class C^{-1}, a function that is continue until derivative d is said to be of class C^d, a function that is always continue for every derivative is said to be of class C^∞.

A B-spline curve of degree m with n control vertices is composed by $n-m$ polynomial segments, one for each interval

$$[t_i, t_{i+1}] \qquad i = m, \ldots, n+1;$$

this means that $\mathbf{S}(u)$ is C^∞ for

$$u \in (t_i, t_{i+1}) \qquad i = m, \ldots, n+1.$$

Note that, if we use generalized B-spline curves, an interval $[t_i, t_{i+1}]$ can also degenerate in just a point if we have a knot multiplicity > 1, in such case there isn't a polynomial segment. On every breakpoint τ_i with $i = 1, \dots, n+m$ we have that the curve has smoothness[1] C^{m-1}.

In our project we don't use generalized B-spline curves, thus a curve of degree m has global smoothness

$$C^{m-1}.$$

3.2.4 *End point interpolation*

In general a B-spline curve with control vertices

$$\boldsymbol{v}_0, \dots, \boldsymbol{v}_n$$

and extended knot vector

$$T = \{t_0, \dots, t_{m-1}, t_m, \dots, t_{n+1}, t_{n+2}, \dots, t_{n+m+1}\}$$

does not necessarily interpolate any control vertex v_i, neither the first nor the last one. But we are interested in using B-spline for representing paths from one point to another. Hence, it should be a nice feature to have that the curve defined in the domain $[a, b]$ is shaped such that

$$\begin{cases} \mathbf{S}(u) = \boldsymbol{v}_0 & \text{for } u = a \\ \mathbf{S}(u) = \boldsymbol{v}_n & \text{for } u = b. \end{cases} \tag{7}$$

We can obtain [9] the conditions of Eq. (7) if we impose

$$t_0 = \dots = t_{m-1} = \overset{\equiv a}{t_m} < \dots < \overset{\equiv b}{t_{n+1}} = t_{n+2} = \dots = t_{n+m+1}$$

on the extended partition vector T. In other words

$$T = \{\overbrace{a, \dots, a}^{m}, t_{m+1}, \dots, t_n, \overbrace{b, \dots, b}^{m}\}.$$

3.2.5 *Curvature and torsion*

Since we are interested in comparing different curves, we need to recognize if a certain curve is a *good* or a *bad* one. One factor that characterizes

1 If we use generalized B-spline curves, it has smoothness C^{m-r} where r is the multiplicity of the knot [12].

a certain curve can be its smoothness (Section 3.2.3) - i.e. a C^3 curve is better than a C^2 curve - but this isn't enough for comparing curves. Usually *curvature* and *torsion* are used for this purpose [11][35]. Both are scalar quantities defined on sufficiently smooth parametric curves for each value of the parameter, and they do not depend on the selected parametrization.

For a generic parametric curve[2] $\mathbf{S}(u)$ defined for $u \in [a, b]$ given the notation $\wedge$ for the vector product and for Eq. (4)

$$\dot{\mathbf{S}}(u) = \frac{d}{du}\mathbf{S}(u) = \begin{bmatrix} \frac{d}{du}x(u) \\ \frac{d}{du}y(u) \\ \frac{d}{du}z(u) \end{bmatrix},$$

we define the curvature $\kappa(u)$ and, in points with non vanishing curvature, the torsion $\tau(u)$ as

$$= \begin{cases} \kappa(u) = \dfrac{\left\|\dot{\mathbf{S}}(u) \wedge \ddot{\mathbf{S}}(u)\right\|_2}{\left\|\dot{\mathbf{S}}(u)\right\|_2^{\,3}} & (8) \\[2ex] \tau(u) = \dfrac{\det\left[\dot{\mathbf{S}}(u), \ddot{\mathbf{S}}(u), \dddot{\mathbf{S}}(u)\right]}{\left\|\dot{\mathbf{S}}(u) \wedge \ddot{\mathbf{S}}(u)\right\|_2} = \dfrac{\left(\dot{\mathbf{S}}(u) \wedge \ddot{\mathbf{S}}(u)\right) \cdot \dddot{\mathbf{S}}(u)}{\left\|\dot{\mathbf{S}}(u) \wedge \ddot{\mathbf{S}}(u)\right\|_2}. & (9) \end{cases}$$

Equation (8) and Eq. (9) describe completely the behavior of $\mathbf{S}(u)$ locally for each value of u. Curvature and torsion have also a geometric interpretation: for each value $\tilde{u}$ of the parameter u, the inverse $\frac{1}{\kappa(\tilde{u})}$ of the curvature is the radius of curvature of $\mathbf{S}$ at $\mathbf{S}(\tilde{u})$ - i.e. the radius of the osculating circle tangent in that point. $\tau(\tilde{u})$ indicates (if $\kappa(\tilde{u}) \neq 0$) how sharply the plane where the curve lies is rotating.

The value of $\kappa(u)$ can be only non negative, while $\tau(u)$ is a signed quantity.

Two curves of same smoothness can be compared using the plots of curvature and torsion, in general curves that have lower peaks of $\kappa(u)$ and $\tau(u)$ are better than curves with higher peaks.

2 Thus also a B-spline curve.

3.2.6 *Arc length*

Sometimes we are interested in evaluating the length of a generic parametric curve[3] $\mathbf{S}(u)$ defined for $u \in [a, b]$. We can obtain such length, called *arc length*, calculating the integral

$$\int_a^b \left\| \dot{\mathbf{S}}(u) \right\|_2 du.$$

We can approximate this value using a discrete tabulation of the curve $\mathbf{S}(u)$ and an integrating method like the *trapezoidal rule* [32][38].

3.3 VORONOI DIAGRAMS

In this section we introduce Voronoi Diagrams (VDs), an important structure used in the project. VDs [8] provide a method to create a partition of the space using distances from a set of input points called *sites*. Formally we have a set

$$S = \{\mathbf{s}_0, \mathbf{s}_1, \dots, \mathbf{s}_n\} \subset \mathbb{E}^d$$

of n sites in the euclidean space of dimension d, and we build a set of n Voronoi *cells*[4]

$$Vor(S) = \{V(\mathbf{s}_0), \dots, V(\mathbf{s}_n)\} \subset 2^{\mathbb{E}^d}$$

such that

$$V(\mathbf{s}_i) = \{\mathbf{p} \in \mathbb{E}^d \ : \ \|\mathbf{p} - \mathbf{s}_i\|_2 < \left\|\mathbf{p} - \mathbf{s}_j\right\|_2 \ \forall \mathbf{s}_j \neq \mathbf{s}_i\}$$

is the set of the points in $\mathbb{E}^d$ closer to $\mathbf{s}_i$ than to any other site.

Figure 2 is an example of the VD built on some random sites, the dashed lines in the figure are edges that go to infinite.

The most important algorithm for calculating VDs is the *Fortune's sweeping line* algorithm that builds the diagram in $\mathcal{O}(n \log n)$ and it is optimal. The algorithm involves building $Vor(S)$ incrementally while sweeping the space, see Fig. 3. Every time that the sweeping line finds a site the algorithm creates a parabola using the site as focus and the sweeping line as directrix. Such parabolas, or better the arcs between each intersection of them, constitute the *beach line*. A parabola disappears from the scene

3 See Footnote 2.

4 $2^{\mathbb{E}^d}$ is the power set of $\mathbb{E}^d$, the set of all the subsets of $\mathbb{E}^d$.

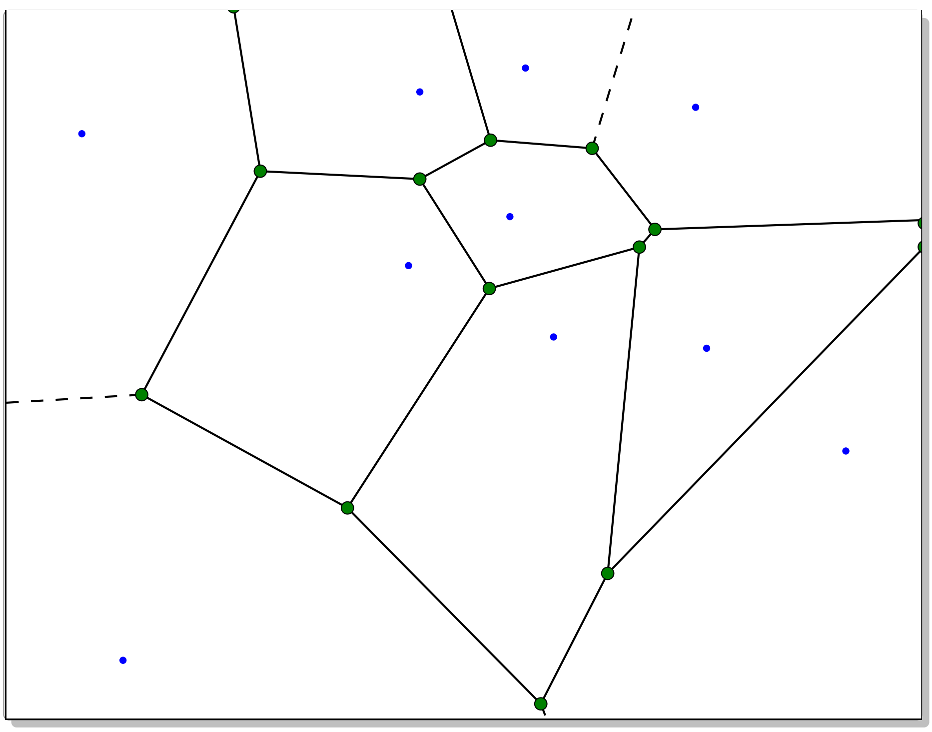

Figure 2.: Example of a VD, dashed lines are infinite edges.

when the associated arc vanishes. The evolution of the intersection points on the beach line constitutes the edges of the VD, and each point where an arc of the beach line disappears constitutes a vertex of the VD. Refer to [8] and [14] for details about the Fortune's algorithm.

One property of VDs is that $V(\mathbf{s}_i)$ can be a closed or an open area - i.e. the edges of the cells can be infinite - it is important to keep this in mind if we want to interpret $\mathrm{Vor}(S)$ as a graph. In that case the graph will have edges that go to infinite. We call such graph $G(\mathrm{Vor}(S))$.

Another property is that, if we have $d+1$ sites $\mathbf{s}'_0, \ldots, \mathbf{s}'_d$ that lay on the surface of a $(d-1)$-sphere[5] that does not have any other site on the interior, then the center point of the $(d-1)$-sphere is the vertex shared only between the $d+1$ cells $V(\mathbf{s}'_0), \ldots, V(\mathbf{s}'_d)$ [8]. This is not true for less than $d+1$ sites on a $(d-1)$-sphere because they are not enough to define it univocally, but is possible to have $n > d+1$ sites on a $(d-1)$-sphere. In that case, the center of the $(d-1)$-sphere is the shared vertex of the cells corresponding to the n sites. This is important to reason about the

5 A circumference in 2-dimensional space, a sphere in 3-dimensional space, an hyper-sphere in n-dimensional space with $n \geqslant 3$.

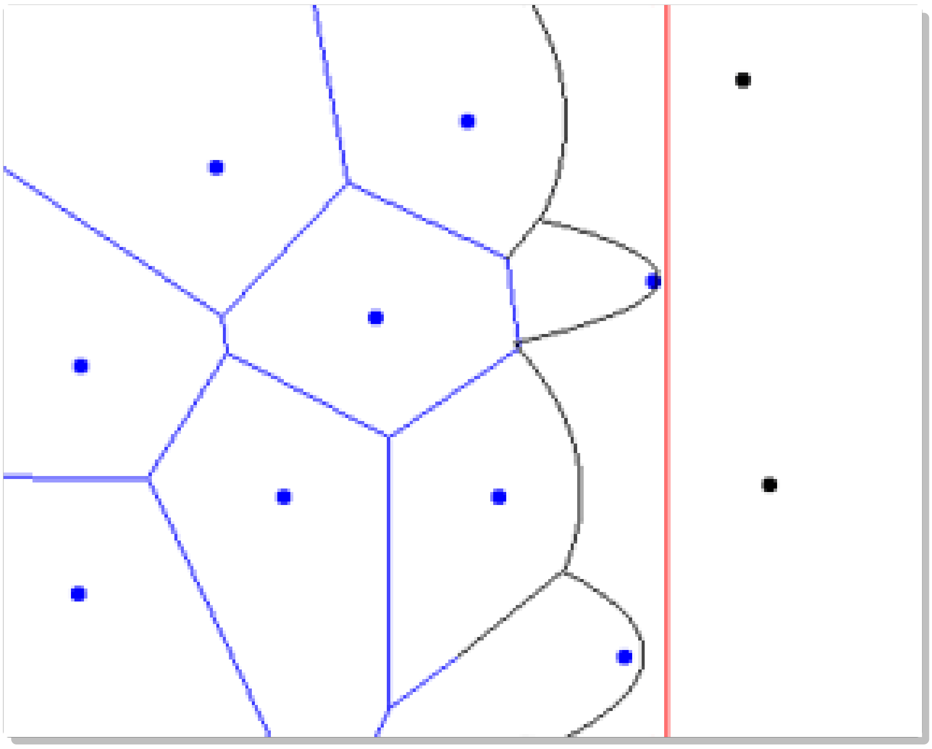

Figure 3.: Fortune's algorithm execution

topography of $G(Vor(S))$ because if we allow more than $d+1$ sites on a $(d-1)$-sphere then the maximum degree $\Delta(G(Vor(S)))$ of the graph can be arbitrarily big (up to the number of vertices). However, the fact that we work with coordinates in $\mathbb{E}^d$ legitimizes the restriction[6] of not allowing more than $d+1$ sites on an $(d-1)$-sphere, limiting $\Delta(G(Vor(S)))$ to $d+1$.

3.4 STATISTICAL METHODS

In this section we briefly introduce the Monte Carlo Method (MCM) [29][37][3] and the Simulated Annealing (SA) [23][19], two statistical methods to calculate unknown quantities and to find functions minima. Additionally, we introduce Lagrangian Relaxation (LR) [40], a method to transform a constrained optimization problem in an unconstrained one by increasing the state space dimension.

3.4.1 *Notes on probabilities*

3.4.1.1 *PE*

For a random variable X normally distributed with

- mean μ;
- variance σ;

for

$$r = 0.6745\sigma$$

we have that

$$\mathbf{P}\left(|X-\mu| < r\right) = \mathbf{P}\left(|X-\mu| > r\right) = 0.5.$$

Thus values of X that deviate from μ less or more than r have the same probability, and r identifies the most PE in a normal distribution.

6 We can also relax this restriction and in case create multiple nodes connected by zero-distance edges on the graph.

3.4.1.2 *Probability central limit theorem (PCLT)*

Consider N independent and *identically-distributed* random variables $X_1, X_2, \ldots, X_N$, with same mean and same variance

$$\begin{aligned}\mathbf{E}[X_1] = \mathbf{E}[X_2] = \cdots = \mathbf{E}[X_N] &= m \\ \mathbf{Var}(X_1) = \mathbf{Var}(X_2) = \cdots = \mathbf{Var}(X_N) &= b^2.\end{aligned}$$

Consider the sum of those random variables:

$$Y = X_1 + X_2 + \cdots + X_N;$$

we have that

$$\begin{aligned}\mathbf{E}[Y] &= \mathbf{E}[X_1 + X_2 + \cdots + X_N] = Nm \\ \mathbf{Var}(Y) &= \mathbf{Var}(X_1 + X_2 + \cdots + X_N) = Nb^2.\end{aligned}$$

Consider a normally distributed random variable Z with parameters:

$$\begin{aligned}\mu &= Nm \\ \sigma &= b\sqrt{N}\end{aligned}$$

with Probability Density Function (PDF) $p_Z(x)$.

The *PCLT* affirms that, for N big enough, and for every interval (x_1, x_2), applies:

$$\mathbf{P}(x_1 < Y < x_2) \approx \int_{x_1}^{x_2} p_Z(x)dx. \tag{10}$$

Thus, the sum of an elevate number of identically-distributed random variables is a random variable with normal distribution with mean Nm and variance Nb^2, even if $X_1, X_2, \ldots, X_N$ aren't normally distributed.

3.4.2 *Monte Carlo Method (MCM)*

If we suppose to calculate an unknown quantity m, we need to find a random variable X such that:

$$\mathbf{E}[X] = m.$$

If we have such distribution with variance:

$$\mathbf{Var}(X) = b^2$$

it is possible to formalize the following passages.

Consider N random variables $X_1, X_2, \ldots, X_N$ that have distribution identical to the distribution of X. For the PCLT Eq. (10) we have that, for N big enough

$$Y = X_1 + X_2 + \cdots + X_N$$

is normally distributed with parameters

$$\begin{aligned} \mu &= Nm \\ \sigma &= b\sqrt{N}. \end{aligned}$$

For the *three sigma rule* [33] we have that:

$$\mathbf{P}\left(\mu - 3\sigma < Y < \mu + 3\sigma\right) \approx 0.997$$

that is

$$\mathbf{P}\left(Nm - 3b\sqrt{N} < Y < Nm + 3b\sqrt{N}\right) \approx 0.997$$

dividing by N

$$\mathbf{P}\left(m - \frac{3b}{\sqrt{N}} < \frac{Y}{N} < m + \frac{3b}{\sqrt{N}}\right) \approx 0.997$$

that is

$$\mathbf{P}\left(\left|\frac{Y}{N} - m\right| < \frac{3b}{\sqrt{N}}\right) \approx 0.997$$

results in

$$\mathbf{P}\left(\left|\frac{1}{N}\sum_{i=1}^{N} X_i - m\right| < \frac{3b}{\sqrt{N}}\right) \approx 0.997. \tag{11}$$

Equation (11) asserts that, if we extract a sample for each random variable X_i, the arithmetic mean of those values is approximately equal to m. Moreover, the error of such approximation is equal to $3b/\sqrt{N}$, that tend to 0 increasing N. It is also possible to further reduce the uncertainty $(1 - 0.997 = 0.003)$ by increasing the number k of sigma used for the approximation and evaluating the error $kb/\sqrt{N}$.

In practice, since the random variables X_i have the same distribution of X, it is sufficient to extract N samples from X to reach the same conclusions.

The Monte Carlo Method (MCM) is constituted by the following procedure, to be adapted according to the problems:

1. find the distribution X having desired quantity m as mean value and b^2 as variance;
2. extract N samples from X, with N big enough to have an error as small as desired;
3. the arithmetic mean of those N samples is the approximation of the desired value m.

Essentially, we transforme the problem from *calculating* m to *finding the distribution* X, or anyway the N samples distributed accordingly to X.

If we want to characterize more in detail the error committed taking N samples, we can use to PE. If we set $k = 0.6745$ then we have that

$$\mathbf{P}\left(\left|\frac{1}{N}\sum_{i=1}^{N} X_i - m\right| < \frac{0.6745 \cdot b}{\sqrt{N}}\right) \approx 0.5$$

and so

$$r_N = \frac{0.6745 \cdot b}{\sqrt{N}}$$

indicates how much the value $\frac{1}{N}\sum_{i=1}^{N} X_i$ deviates from the desired value m. Such value characterize the absolute error

$$\left|\frac{1}{N}\sum_{i=1}^{N} X_i - m\right|$$

committed taking N samples.

MCM is useful to simulate events that have an high degree of uncertainty in the inputs or an high degree of liberty in the state: for instance, numerically integrate a function with many dimensions or Simulated Annealing (SA) (Section 3.4.3).

3.4.3 *Simulated Annealing (SA)*

The SA is a method used to find the global maximum or minimum of a function. It is inspired by a method used in metallurgy that consists in heating and then cooling slowly a material to increase the size of the crystals and improving the chemico-physical properties. The function that must be optimized can be defined in a multiple-dimensional space.

3.4.3.1 *Statistical thermodynamic*

To describe the basic principles of statistical thermodynamic we consider the following example. In a one-dimensional lattice every point is a particle with a value of spin that can be *up* or *down*. If the lattice has N points then the system can be in 2^N different configurations, where each one of those configurations corresponds to a value of energy, for instance:

$$E = B(n_+ - n_-)$$

where B is some constant, n_+ is the number of particles with spin *up* and n_- is the number of particles with spin *down*.

The probability $B(\sigma)$ of finding the system in a certain configuration σ is given by the distribution of *Boltzmann-Gibbs*:

$$P(\sigma) = Ce^{-E_\sigma/T} \tag{12}$$

where E_σ is the energy of the configuration, T is the temperature[7] and C is a normalization constant.

The average energy of the system is then:

$$\begin{aligned} \bar{E} &= \frac{\sum_\sigma E_\sigma P(\sigma)}{\sum_\sigma P(\sigma)} \\ &= \frac{\sum_\sigma E_\sigma e^{-E_\sigma/T}}{\sum_\sigma e^{-E_\sigma/T}}. \end{aligned}$$

The computation of the value of $\bar{E}$ can be difficult with an high number of states, but it is possible to create a MCM simulating the random fluctuation between the states such that the distribution given by Eq. (12) is respected. Starting from an arbitrary initial configuration, after a certain number of *Monte Carlo trials*, the method converges to the equilibrium status $\bar{E}$ and it continues to fluctuate around it. SA is a method of this kind.

3.4.3.2 *Simulated Annealing (SA) algorithm*

SA operates on a system starting from a certain initial state s_0, then it executes a series of iterations where a neighbour of the state is evaluated and, with a certain distribution of probability, the system is moved in the new state or not.

7 The real Boltzmann-Gibbs distribution is $P(\sigma) = Ce^{-E_\sigma/kT}$ where k is the *Boltzmann constant* and T is the thermodynamic temperature, but for the example the temperature is a parameter not correlated to the physical world, thus it is possible to ignore k.

A possible algorithm for a SA method is Algorithm 1. s_0 is the initial state; $\mathtt{temp}$ is the function that assigns a temperature based on the current iteration number such that for low k the returned temperature is high and for high k the returned temperature is low; $\mathtt{neighbour}$ is the function that returns a random neighbour of the current state; $\mathtt{uniform}$ returns an uniformly-randomly chosen number in $[0, 1]$; P_a is the distribution of accepting probability that depends on the energy of the current state, on the energy of the neighbour, and on the current temperature. In case of acceptance, the neighbour becomes the current state and the process continues.

Algorithm 1 Simulated Annealing (SA)

```
function ANNEAL(s0)
    s ← s0
    for k ← 0, kMax do
        T ← temp(k / kMax)
        sNew ← neighbour(s)
        if uniform(0, 1) < Pa(E(s), E(sNew), T) then
            s ← sNew
        end if
    end for
    return s
end function
```

The relation with the statistical thermodynamic is that P_a is chosen such that Eq. (12) holds[8], moreover $\mathtt{temp}$ returns decreasing values of temperature with the succession of iterations. This explains the comparison with the metallurgy annealing.

Initially P_a was chosen such that

$$P_a(E(s), E(sNew), T) = \begin{cases} 1, & \text{if } E(sNew) < E(s) \\ e^{-(E(sNew)-E(s))/T}, & \text{otherwise} \end{cases}$$

but this isn't strictly necessary to develop a SA method.

8 A similar distribution is enough.

3.4.4 *Lagrangian Relaxation (LR)*

A general constrained discrete optimization problem can be expressed in the form:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & g(x) = 0 \end{aligned} \tag{13}$$

where $x \in X$ is the state of the system in a discrete space X, $f(x)$ is the function to minimize, and $g(x) = 0$ is the constraint. The functions can also be in a multidimensional discrete space, in that case the x is a vector $\mathbf{x} = (x_1, \ldots, x_n)$ of variables.

To solve this class of problems a *Lagrange relaxation* method can be used [4]: it expands the variable space X by a *Lagrange multiplier* space Λ, equal in dimension to the number of constraints - one in the Problem 13.

The *generalized discrete Lagrangian function*, corresponding to the Problem 13, is:

$$L_d(x, \lambda) = f(x) + \lambda H(g(x)) \tag{14}$$

where λ is a variable in Λ; if the dimension of Λ is more than one λ, it must be transposed in Eq. (14). $H(x)$ is a non negative function with the property that $H(0) = 0$ and aimed to transform g in a non negative function. For instance, it can be $H(g(x)) = |g(x)|$ or $H(g(x)) = g^2(x)$.

Under the previous assumptions, the set of *local minima* in Problem 13 - that respect the constraints - coincides with the set of *discrete saddle point* in the augmented space. A point (x^*, λ^*) is a discrete saddle point if:

$$L_d(x^*, \lambda) \leqslant L_d(x^*, \lambda^*) \leqslant L_d(x, \lambda^*)$$

for all $x \in \mathcal{N}(x^*)$ and for all $\lambda \in \Lambda$, where $\mathcal{N}(x^*)$ is the set of all x^*'s neighbours.

To solve the optimization Problem 13 it is necessary to calculate, among the saddle points, the global minimum for f. We can use an optimization method, like SA, that descends in X and ascends in Λ.

3.5 INTERSECTIONS IN SPACE

We work in a spatial environment with polyhedral obstacles. Thus, in order to define admissible paths, first of all we need routines performing the following three basic geometric tasks:

1. establish if a point is in or out of a convex polyhedron;
2. check if a segment intersects a triangle;
3. establish whether two triangles intersect.

In the project we need to consider three kinds of collision detection methods in 3-dimension euclidean space.

3.5.1 *Point inside convex polyhedron in 3D space*

To test if a point $\mathbf{p}$ is inside a convex polyhedron V with vertices $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$, we use a method that rely on convex hulls [8][36].

Algorithm 2 Check if point $\mathbf{p}$ is inside convex polyhedron V

```
1:  function isPointInPolyhedron(p, V)
2:      inside ← True
3:      C ← convexHullVertices([p, v1, v2, ..., vn])
4:      for all c ∈ C do
5:          if c = p then
6:              inside ← False
7:              break
8:          end if
9:      end for
10:     return inside
11: end function
```

Algorithm 2 performs the first task. It first computes the vertices of the convex hull of all the vertices of V plus the point $\mathbf{p}$ and then checks if $\mathbf{p}$ is one of them or not. If $\mathbf{p}$ is on the convex hull that means that $\mathbf{p}$ is external[9] to V because we have extended the convex hull formed by the vertices of V. Otherwise this means that $\mathbf{p}$ is inside V.

The cost of this algorithm is

$$\mathcal{O}(n \log n)$$

where n is the number of vertices of V, because the cost to construct the convex hull is [8] $\mathcal{O}(n \log n)$, and then we have another negligible term $\mathcal{O}(n)$ for the cycle on Line 4.

9 Or $\mathbf{p}$ coincides with a vertex of V.

3.5.2 *Segment-triangle in 3D space*

We need to deal with the intersection between a segment $S = \overline{\mathbf{a}_2\mathbf{b}_2}$ and a triangle $T = \triangle \mathbf{a}_1\mathbf{b}_1\mathbf{c}_1$. S and T can be in one of the following cases also summarized in Table 1:

case 1 S and T do not intersect and the plane containing T is not in the sheaf of planes generated by the line containing S;

case 2 S and T do not intersect and the plane containing T is in the sheaf of planes generated by the line containing S;

case 3 S and T intersect only at one point and the plane containing T is not in the sheaf of planes generated by the line containing S;

case 4 S and T intersect in one or infinite points and the plane containing T is in the sheaf of planes generated by the line containing S.

The discriminating factors among the cases are two: the presence of intersection and coplanarity. Table 1.

	not coplanar	coplanar
not intersect	**case 1**	**case 2**
intersect	**case 3**	**case 4**

Table 1.: Relations between S and T

In Fig. 4 a **case 3** situation is shown where there is intersection in only one point $\mathbf{x}$. To establish whether S and T intersect, we need to solve four equations in four unknowns [36] where we look for a point $\mathbf{x}$ being a convex linear combination of $\mathbf{a}_2$ and $\mathbf{b}_2$ and at the same time a convex linear combination of $\mathbf{a}_1$, $\mathbf{b}_1$ and $\mathbf{c}_1$. In other words, when there is a collision, then there is a solution for the unknowns α, β, γ, δ, ζ of the system

$$\begin{cases} \alpha\mathbf{a}_2 + \beta\mathbf{b}_2 = \gamma\mathbf{a}_1 + \delta\mathbf{b}_1 + \zeta\mathbf{c}_1 \\ \alpha + \beta = 1 \\ \gamma + \delta + \zeta = 1 \end{cases} \tag{15}$$

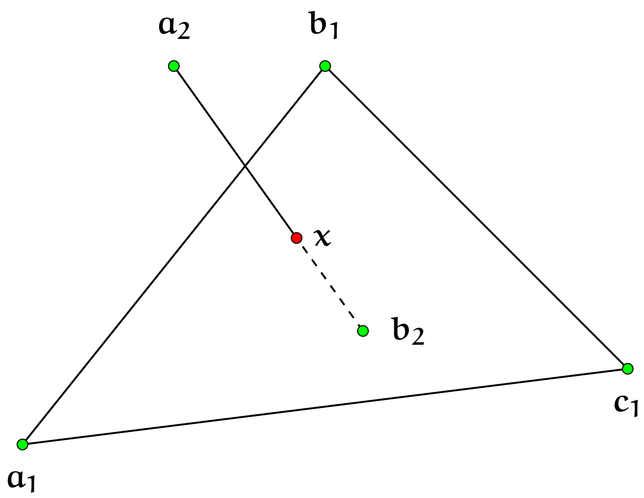

Figure 4.: Example intersection between a segment $\overline{\mathbf{a}_2\mathbf{b}_2}$ and a triangle $\triangle \mathbf{a}_1\mathbf{b}_1\mathbf{c}_1$.

with the further conditions

$$\begin{cases} \alpha \geqslant 0 \\ \beta \geqslant 0 \\ \gamma \geqslant 0 \\ \delta \geqslant 0 \\ \zeta \geqslant 0. \end{cases} \tag{16}$$

Note that the first equation of System 15 has vectorial coefficients $\mathbf{a}_2$, $\mathbf{b}_2$, $\mathbf{a}_1$, $\mathbf{b}_1$, $\mathbf{c}_1$, thus we have a system with five unknowns in five equations. If System 15 has just one solution then we are in **case 1** when System 16 is fulfilled or in **case 3** when it is not. If it has infinite solutions then we are on **case 2** or **case 4**, depending again on the fulfillment of System 16. Finally, if it has no solution, then S or T are degenerated.

We are interested in finding only **case 3** collisions because, for simplicity, we consider the special case of a segment that lays on the surface of a triangle as nonintersecting with it, and, for coherence, we restrict the conditions of System 16 to

$$\begin{cases} \alpha > 0 \\ \beta > 0 \\ \gamma > 0 \\ \delta > 0 \\ \zeta > 0. \end{cases} \tag{17}$$

System 15 can be simplified in the three equations

$$\begin{cases} \alpha \mathbf{a}_2 + (1-\alpha)\mathbf{b}_2 = \gamma \mathbf{a}_1 + \delta \mathbf{b}_1 + (1-(\gamma+\delta))\mathbf{c}_1 \end{cases} \tag{18}$$

in the unknowns α, γ and δ with the relative conditions

$$\begin{cases} \alpha > 0 \\ \alpha < 1 \\ \gamma > 0 \\ \delta > 0 \\ \gamma + \delta < 1. \end{cases} \tag{19}$$

Algorithm 3 Find intersection between segment S and triangle T

```
function INTERSECT(S, T)
    intersect ← False
    coordinates ← ∅
    if (α, γ, δ) ← solve(System 18) then
        if satisfy(System 19) then
            intersect ← True
            coordinates ← (γ, δ, 1 − (γ + δ))
        end if
    end if
    return (intersect, coordinates)
end function
```

Thus, Algorithm 3 which performs Task 2 essentially consists in solving System 18 with the parameters $\mathbf{a}_2$, $\mathbf{b}_2$, $\mathbf{a}_1$, $\mathbf{b}_1$ and $\mathbf{c}_1$ from S and T; and then in checking if the solution is admissible. The condition of Line 4 is $\mathtt{True}$ if System 18 has solution and if that is unique.

We also have the positive secondary effect that from the solution (α, γ, δ) of System 18 we can extract the barycentric coordinates $(\gamma, \delta, 1 - (\gamma + \delta))$ of the intersection point $\mathbf{x}$ on the system of the vertices $\mathbf{a}_1$, $\mathbf{b}_1$, $\mathbf{c}_1$ of T.

3.5.3 *Triangle-triangle in 3D space*

We are interested in detecting collisions between two triangles $T_1 = \triangle \mathbf{a}_1\mathbf{b}_1\mathbf{c}_1$ and $T_2 = \triangle \mathbf{a}_2\mathbf{b}_2\mathbf{c}_2$ in 3-dimensional space. First of all consider the coplanarity relation between the two triangles, we have the cases:

case 1 T_1 and T_2 are contained by the same plane;

case 2 T_1 and T_2 are contained by different planes.

To simplify the problem we decide - similarly to the case of intersection between segment and triangle - that when we are on **case 1** we consider T_1 and T_2 not intersecting in any case, even if from a geometrical point of view they share points. After this premise we can assert that the possible relation between T_1 and T_2 can be exclusively one of the following types [36]:

type 0 T_1 and T_2 do not intersect;

type 1 two edges of T_1 intersect the plane section delimited by T_2, or vice versa;

type 2 one edge of T_1 intersects the plane section delimited by T_2 and one edge of T_2 intersects the plane section delimited by T_1.

On Fig. 5 and Fig. 6 we can see two examples of **type 1** and **type 2**, respectively. To establish if T_1 and T_2 intersect we need to check if every edge of T_1 intersects T_2 and if every edge of T_2 intersects T_1. If we find at least one edge that intersects with one triangle, then T_1 and T_2 intersect. Algorithm 4 executes such check, the function `intersect` on Line 3 and Line 8 is the intersection check between a segment and a triangle done by Algorithm 3.

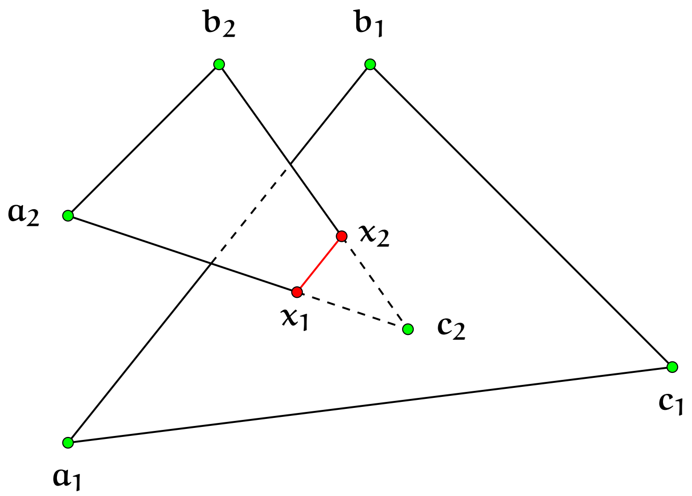

Figure 5.: Example of **type 1** intersection between a triangle $T_1 = \triangle \mathbf{a_1 b_1 c_1}$ and another triangle $T_2 = \triangle \mathbf{a_2 b_2 c_2}$.

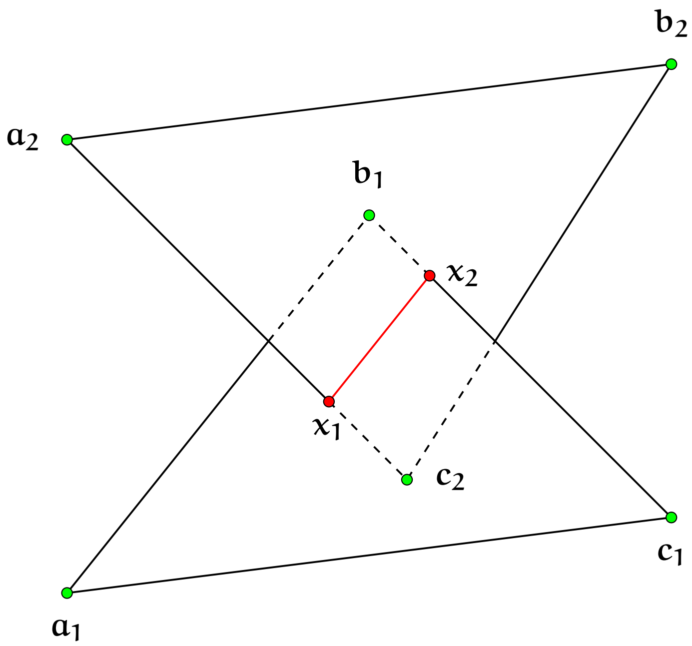

Figure 6.: Example of **type 2** intersection between a triangle $T_1 = \triangle \mathbf{a_1 b_1 c_1}$ and another triangle $T_2 = \triangle \mathbf{a_2 b_2 c_2}$.

Algorithm 4 Find intersection between triangle T_1 and triangle T_2

function INTERSECT($T_1 = (\mathbf{a}_1, \mathbf{b}_1, \mathbf{c}_1),\ T_2 = (\mathbf{a}_2, \mathbf{b}_2, \mathbf{c}_2)$)
    for all $S \in \{\overline{\mathbf{a}_1\mathbf{b}_1},\ \overline{\mathbf{b}_1\mathbf{c}_1},\ \overline{\mathbf{c}_1\mathbf{a}_1}\}$ **do**
        if $\mathrm{intersect}(S, T_2)$ **then**
            return True
        end if
    end for
    for all $S \in \{\overline{\mathbf{a}_2\mathbf{b}_2},\ \overline{\mathbf{b}_2\mathbf{c}_2},\ \overline{\mathbf{c}_2\mathbf{a}_2}\}$ **do**
        if $\mathrm{intersect}(S, T_1)$ **then**
            return True
        end if
    end for
    return False
end function

4

SCENE REPRESENTATION

The problem of scene description basically consists in fixing a representation of the obstacles and of the path, besides establishing the structures adopted for their storage.

4.1 BASIC ELEMENTS AND PATH

First of all, since we are interested in spatial path planning, all the point coordinates are in $\mathbb{E}^3$. Furthermore, we concentrate on B-splines because we want a standard representation for the output of the algorithm, the path between a start point s and an end point e. B-spline curves are the standard adopted in CAD and CAGD systems [21][22].

The structures that uniquely identify a B-spline curve are three: its degree m, the associated control polygon and the extended knot vector.

Regarding the degree of the curve, we let the users choose among quadratics ($m = 2$), cubics ($m = 3$) and quartics ($m = 4$). The users choose also the starting and ending points, s and e respectively, associated to the parameter values $t_0 = \cdots = t_m$ and $t_{n+1} = \cdots = t_{n+m+1}$.

The number of vertices and the other vertices themselves come from the algorithm, and they depend on the position of s and e and on the obstacles, see Section 5.1 for details.

The knots are generated automatically using one of the two methods described in Section 5.4.

Thus, for the curve we memorize only the control vertices P and the degree m. As usual, in any computer graphic system, when we want its plotting, we tabulate $\mathbf{S}$ for a certain number[1] of values of t and then we draw the polygonal chain that connects them.

1 Enough for having a smooth look.

4.2 BASIC OBSTACLE REPRESENTATION

Besides the curve, in the scene we need to represent the obstacles. We call Obs the set of all obstacles in scene. We choose to represent each obstacle $\mathrm{Ob} \in \mathrm{Obs}$ as a set of triangular faces called Obstacle Triangular Faces (OTFs), each one containing three vertices. To summarize, we have

$$\begin{aligned}
\mathrm{Obs} &= \{\mathrm{Ob}_0, \dots, \mathrm{Ob}_{\#\mathrm{Obs}}\} \\
\mathrm{Ob}_i &= \{\mathrm{Otf}_{i,0}, \dots, \mathrm{Otf}_{i,\#\mathrm{Otf}_i}\} && i = 0, \dots, \#\mathrm{Obs} \\
\mathrm{Otf}_{i,j} &= \{\mathbf{p}_{i,j,0}, \mathbf{p}_{i,j,1}, \mathbf{p}_{i,j,2}\} && i = 0, \dots, \#\mathrm{Obs}; \quad j = 0, \dots, \#\mathrm{Otf}_i
\end{aligned}$$

where $\#\mathrm{Obs}$ is the number of obstacles in the scene and $\#\mathrm{Otf}_i$ is the number of OTFs in obstacle Ob_i.

We choose this specific configuration because this way all the intersections that can occur are between triangle and triangle or triangle and segment and they can be easily calculated. This implies that, if an obstacle is a polyhedron more complex than just a tetrahedron, its faces must to be preliminarily triangulated.

We provide the methods explained in Section 4.3 to abstract the creation of OTFs.

Using this solution, we can potentially insert open polyhedrons[2] or intersecting shapes in the scene, as we do not have any restriction on the position of the points $\mathbf{p}_{i,j,k}$.

4.3 COMPLEX OBSTACLES

In order to simplify the scene construction, we create four methods to easily build obstacles:

- one for tetrahedrons;
- one for parallelepipeds[3];
- a more general one for convex hulls;
- a special method for a bucket-shaped obstacle that we use in the tests.

Algorithm 5 takes the four vertices of a tetrahedron and adds to Obs a new obstacle that have all the faces of the unique tetrahedron that can be built with the four points.

2 For instance a tetrahedron without one face
3 Aligned with the axis.

Algorithm 5 Abstract construction of tetrahedron

procedure BUILDTETRAHEDRON($Obs, \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}$)
    $Obs \leftarrow Obs \cup \{\ \{\mathbf{a}, \mathbf{b}, \mathbf{c}\},\ \{\mathbf{a}, \mathbf{b}, \mathbf{d}\},\ \{\mathbf{b}, \mathbf{c}, \mathbf{d}\},\ \{\mathbf{c}, \mathbf{a}, \mathbf{d}\}\ \}$
end procedure

Algorithm 6 Abstract construction of convex hull polyhedron

procedure BUILDCONVEXHULLPOLYHEDRON($Obs, \mathbf{p}_0, \dots, \mathbf{p}_n$)
    $Ob \leftarrow \emptyset$
    $facets \leftarrow convexHull(\{\mathbf{p}_0, \dots, \mathbf{p}_n\})$
    for all $f \in facets$ **do**
        $simplices \leftarrow triangularize(f)$
        for all $\{\mathbf{s}_0, \mathbf{s}_1, \mathbf{s}_2\} \in simplices$ **do**
            $Ob \leftarrow Ob \cup \{\mathbf{s}_0, \mathbf{s}_1, \mathbf{s}_2\}$
        end for
    end for
    $Obs \leftarrow Obs \cup Ob$
end procedure

Algorithm 6 is more complex, first we need to build the convex hull of the input points (see [8] and [32] for details on the convex hull algorithm), then we obtain a set of $facets$ that have to be triangulated (see [8] and [32] for details on the triangularization algorithms). Finally we add each triangle as a new OTF of the obstacle.

4.4 BOUNDING BOX

We also give to the user the possibility of adding a bounding box around the scene. It is built as an obstacle, using OTFs, in fact we provide a method that takes two points $\mathbf{a}$ and $\mathbf{b}$ and builds the parallelpiped having those points as extremes and with all the faces triangularized like in Fig. 7.

In regards to the intersections, the OTFs of the bounding box are considered exactly like the OTFs of the obstacles throughout the whole project. The only differences are that the bounding box is not visible when the scene is plotted and a point inside the bounding box is not considered to be inside the obstacle.

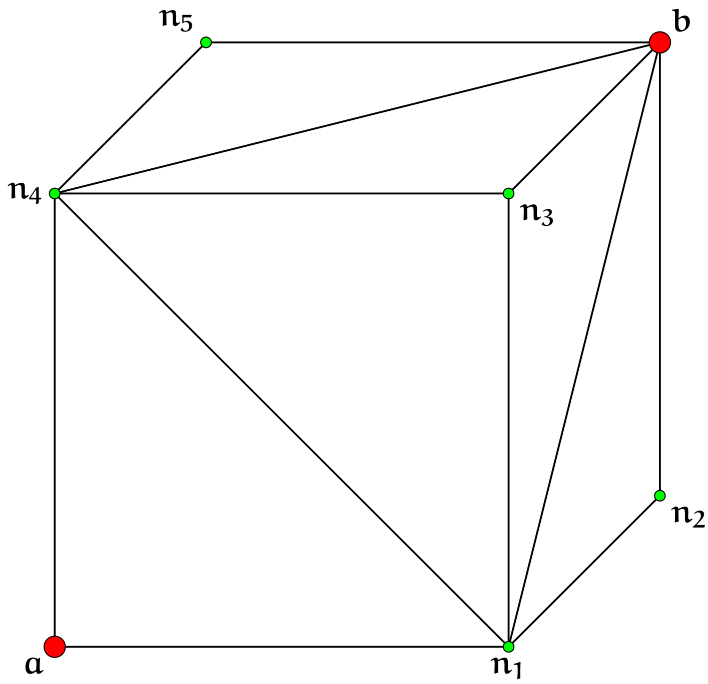

Figure 7.: Bounding box with extremes $\mathfrak{a}$ and $\mathfrak{b}$.

5

ALGORITHMS

In this chapter we analyze step-by-step the algorithms that implement the different parts of the program. We do this with the help of the test scene in Fig. 8.

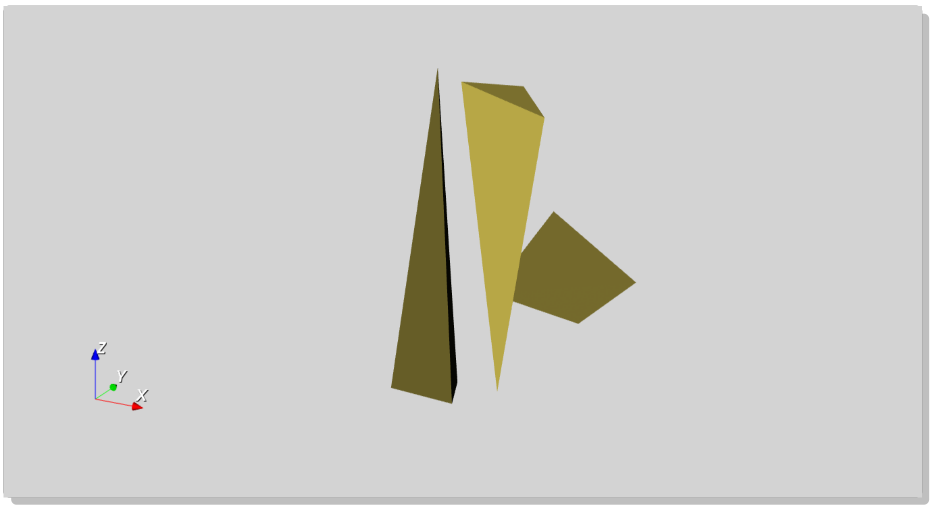

Figure 8.: Initial scene.

The general idea is to use an open B-spline curve of a certain degree interpolating the chosen starting and ending points, whose control polygon is a suitable modification of a polygonal chain extracted from a graph obtained with a VD method. In Section 5.1 we explain in detail how to build such polygonal chain. The chain, before being used as a control polygon for the B-spline, is refined and adjusted - as explained in detail in Section 5.2 and Section 5.3 - in order to ensure that the associated B-spline curve has no obstacle collision. Furthermore, in Section 5.4 we implement a method for an optional adaptive arrangement of the breakpoints of the B-spline. Finally, Section 5.5 is devoted to an optional post-processing of the path.

5.1 POLYGONAL CHAIN

In the first phase, the purpose is to extract a suitable polygonal chain from the scene, such that the extremes coincide with the start point s and the end point e. In particular, we are interested in short length chains. We calculate the shortest path in a graph that is obtained by using an adaptation to three dimensions of a well known bidimensional method [5][18][39] that use VDs as base.

We choose a Voronoi method because it builds a structure roughly equidistant from obstacles,resulting in a low probability of collisions between the curve and the obstacles.

5.1.1 *Base Graph*

First we start distributing points on the OTFs and on an invisible bounding box, as in Fig. 9. The sites are distributed using a recursive method, for

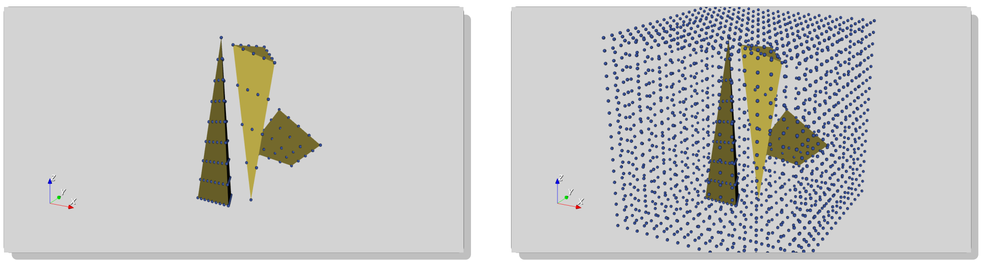

Figure 9.: Scene with Voronoi sites (distributed only on the obstacles surfaces on the left, and on obstacles and bounding box on the right).

each triangle of the scene we add three points - one for each vertex, if not already added before - and then we calculate the area of the triangle. If the area is bigger than a threshold, we decompose the triangle in four triangles adding three more vertices on the midpoints of the edges of the original triangle as in Fig. 10. We repeat the process recursively for each new triangle.

We construct the VD using the Fortune's algorithm [14] on those points as input sites, and we build a graph

$$G = (V, E)$$

using the vertices of the Voronoi cells as graph nodes in V, and the edges of the cells[1] as graph edges in E. Furthermore, we make G denser by

1 Rejecting potential infinite edges.

Figure 10.: Decomposition of an OTF.

adding all the diagonals as edges for every cell's face, in other words we connect every vertex to every other vertex of a face.

Subsequently, we prune such graph deleting every edge that intersects an OTF using the methods explained in Section 3.5. The edge-pruning process considers a margin around the OTFs during the collision checks.

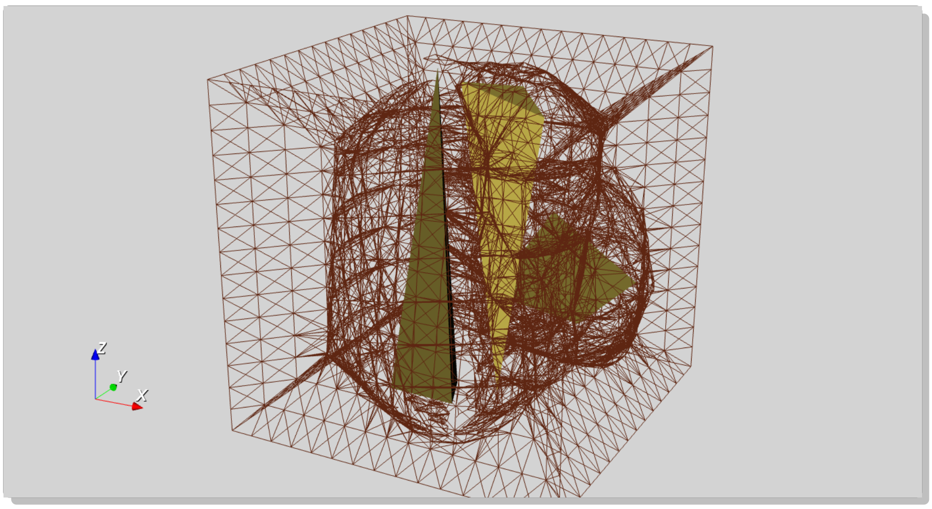

Figure 11.: Scene with pruned graph.

The result, visible in Fig. 11, is a graph that embraces the obstacles like a cobweb where the possible paths are roughly equidistant from the obstacles.

As visible in Fig. 12, in the bidimensional scenario the equivalent method implies distributing the sites (the blue dots) in the edges of the polygonal obstacles and then pruning the graph when an edge of the graph intersects an edge of the obstacle. The result is a sparse graph composed of chains around the obstacles (the green dots).

We decide to extend the method in 3 dimensions distributing points in the whole OTF surface. An alternative to this would be distributing points only along the edges of the obstacles.

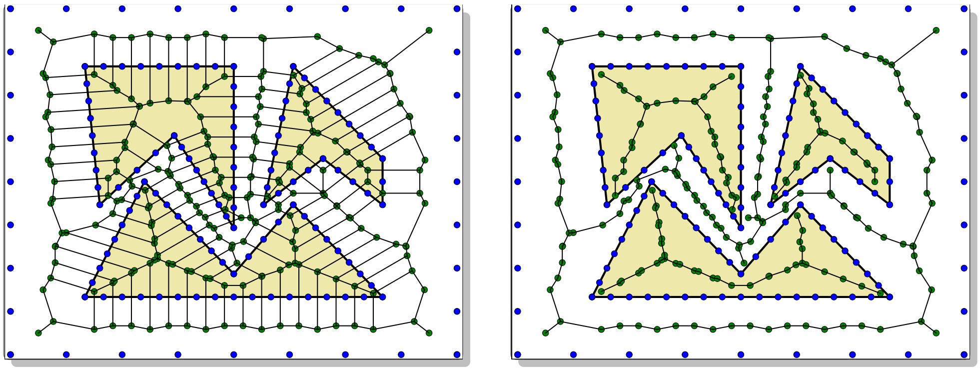

Figure 12.: Voronoi graph in 2D before (left) and after (right) pruning.

We attach the desired start and end points $\mathbf{s}$ and $\mathbf{e}$ on the obtained graph G and we can obtain a path between the two points using an algorithm like Dijkstra [10][24]. To attach $\mathbf{s}$ and $\mathbf{e}$ we finds the vertex $\mathbf{v_n} \in V_{vis} \subseteq V$ such that $\mathtt{dist}(\mathbf{s}, \mathbf{v_n}) \leqslant \mathtt{dist}(\mathbf{s}, \mathbf{v_i}), \forall \mathbf{v_i} \in V_{vis}$, where

$$V_{vis} = \{\mathbf{v} \in V \ : \ \overline{\mathbf{s}\mathbf{v_i}} \text{ do not intersects any obstacle}\},$$

then adds $\mathbf{s}$ to V and the edge $(\mathbf{s}, \mathbf{v_n})$ to E. Similarly for $\mathbf{e}$.

Before using that path as a control polygon, we need to take into account the degree of the B-spline and the position of the obstacles, the details are in Section 5.2 and Section 5.3.

5.1.1.1 *Complexity considerations*

Fortune's algorithm runs in time $\mathcal{O}(|I| \log |I|)$ [8], where I is the set of input sites. If we impose a maximum area A for the obstacles [2] then $|I| = \mathcal{O}(|O|)$ where O is the set of obstacles, because in the worst case we have that $|I| = C \cdot A \cdot |O|$ for some constant C that depends on the chosen density of sites per area.

In conclusion, the time cost for the creation of the graph is

$$\mathcal{O}(|O| \log |O|) \tag{20}$$

and the number of the vertices in the graph is

$$|V| = \mathcal{O}(|I|) = \mathcal{O}(|O|) \tag{21}$$

2 Inserting the obstacles in a progressive order, the area of the i-th obstacle cannot be a function $f(i)$ of the number of the obstacles.

because the number of vertices in the resulting graph has the same order of magnitude of the number of input sites.

If we formulate the hypothesis of having maximum degree k in G - i.e. each vertex in V is connected to other k vertices at most - then we have that

$$|E| = \mathcal{O}(k|V|) = \mathcal{O}(k|O|). \tag{22}$$

In the worst case $k = |V|$ and $|E| = \mathcal{O}(|V|^2)$ but for VDs in plane there is a property that if we have n input sites that lay on a circumference, without any other site inside the circumference, then the center of the circumference is a vertex shared by n cells (Section 3.3 for details). The same property holds in the 3D case with respect to spheres.

We can make the assumption that no more than three sites can lay on a circumference, hence, no vertex can have more than three neighbours, or the same with four vertices in sphere. This assumption is plausible because we use floating point numbers for the coordinates of the vertices of the obstacles and it is unlikely that more than four points lay on a sphere.

Moreover, the average numbers of faces in a VD's cell and, consequently, vertices in a face are bounded by a constant [30]. Thus, we can make the assumption that we do not increase the maximum graph degree by more than a constant when we make the graph denser by adding the faces' diagonals.

With the previous two assumptions k is a constant, and Eq. (22) becomes

$$|E| = \mathcal{O}(|V|) = \mathcal{O}(|O|).$$

To prune the graph of every edge that intersects obstacles, we need to solve a system of three unknowns in three equations for every edge and every OTF[3], so we have a cost of

$$\mathcal{O}(|E| \cdot |O|) = \mathcal{O}(k|O|^2) \tag{23}$$

and, if we make the assumption of k constant, it becomes

$$\mathcal{O}(|O|^2).$$

3 See Section 3.5.2.

5.1.2 *Graph's transformation*

Before calculating the shortest path on the chosen graph with Dijkstra [10][24], we transform it in a graph containing all the triples of three adjacent vertices in the original graph. This because we want to filter the triples for collisions as described in Section 5.2.1. We call the transformed graph

$$G_t = (V_t, E_t)$$

where we have triples of vertices of G in V_t.

The original graph G is not directed and it is weighted with the distance from vertex to vertex, whereas the transformed graph G_t is directed and weighted. If in G the nodes $\mathbf{a}$ and $\mathbf{b}$ are neighbouring, and $\mathbf{b}$ and $\mathbf{c}$ are neighbouring, then G_t has the two nodes $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ and $(\mathbf{c}, \mathbf{b}, \mathbf{a})$. In G_t a node $(\mathbf{a}_1, \mathbf{b}_1, \mathbf{c}_1)$ is a predecessor of $(\mathbf{a}_2, \mathbf{b}_2, \mathbf{c}_2)$ if $\mathbf{b}_1 = \mathbf{a}_2$ and $\mathbf{c}_1 = \mathbf{b}_2$, and the weight of the arc from $(\mathbf{a}_1, \mathbf{b}_1, \mathbf{c}_1)$ to $(\mathbf{a}_2, \mathbf{b}_2, \mathbf{c}_2)$ in G_t is equal to the weight of the arc from $\mathbf{a}_1$ to $\mathbf{b}_1 (= \mathbf{a}_2)$ in G.

The steps necessary to create G_t are summarized in Algorithm 7. The input G is the base graph that has vertices V and edges E, $N_G(\mathbf{a})$ is the set of neighbours in G of the vertex $\mathbf{a}$, and the output is G_t.

The transformation of the graph is useful only for the obstacle avoidance algorithm of Section 5.2.1, theoretically it is possible to bypass such transformation for the algorithm described in Section 5.2.2.

5.1.2.1 *Complexity considerations*

If we suppose a maximum degree k for each vertex in the graph G - i.e. each vertex in V can have k edges insisting on it at most, then the number of vertices in the transformed graph G_t is

$$|V_t| \leqslant |V| \cdot k \cdot (k-1) = \mathcal{O}(k^2|V|) \tag{24}$$

because for each vertex $\boldsymbol{v}$ in G we need to consider all the neighbours of $\boldsymbol{v}$ and the neighbours of the neighbours of $\boldsymbol{v}$ (excluded $\boldsymbol{v}$).

For how we define the triples neighbour rule in G_t we have that each triple is a predecessor of $k-1$ other triples at most. For instance, $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ in V_t is the predecessor of all the triples $(\mathbf{b}, \mathbf{c}, *)$ where $*$ can be one of the k neighbours of $\mathbf{c}$ in V excluded $\mathbf{b}$. Thus, the number of edges in G_t is

$$|E_t| \leqslant |V_t| \cdot (k-1) = \mathcal{O}(k|V_t|) = \mathcal{O}(k^3|V|). \tag{25}$$

Algorithm 7 Create triples graph G_t

```
function createTriplesGraph(G)
    V_t ← E_t ← ∅
    for all (a, b) ∈ E do
        leftOut ← leftIn ← rightOut ← rightIn ← ∅
        for all v ∈ N_G(a) \ {b} do
            leftOut ← leftOut ∪ {(v, a, b)}
            leftIn ← leftIn ∪ {(b, a, v)}
            V_t ← V_t ∪ {(v, a, b), (b, a, v)}
        end for
        for all v ∈ N_G(b) \ {a} do
            rightOut ← rightOut ∪ {(v, b, a)}
            rightIn ← rightIn ∪ {(a, b, v)}
            V_t ← V_t ∪ {(v, b, a), (a, b, v)}
        end for
        for all o ∈ leftOut do
            for all i ∈ rightIn do
                E_t ← E_t ∪ (o, i)
            end for
        end for
        for all o ∈ rightOut do
            for all i ∈ leftIn do
                E_t ← E_t ∪ (o, i)
            end for
        end for
    end for
    G_t ← (V_t, E_t)
    return G_t
end function
```

Furthermore, the time cost for the creation of G_t is

$$\mathcal{O}(k^2|E|) = \mathcal{O}(k^3|O|) \tag{26}$$

because Algorithm 7 scans all the edges e on Line 3 for creating the transformed graph and for each iteration the biggest cost is due to the two *for* on Line 15 and Line 20.

5.2 OBSTACLE AVOIDANCE

Before using the polynomial chain extracted as explained in Section 5.1 as a control polygon for the B-spline, we need to discuss a problem: every possible path in the graph G is free from collisions by construction - in fact we prune the graph of every edge that intersects an obstacle - but this does not guarantee that the associated curve will not cross any obstacle. This concept is exemplified in Fig. 13.

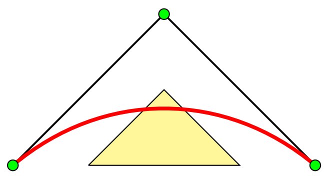

Figure 13.: B-spline that intersects an obstacle in the plane.

In this chapter we formulate the hypothesis of using quadratic B-splines[4], in Section 5.3 we explain how it is possible to use curves with a higher degree. With this assumption, we can exploit the CHPs explained in Section 3.2 and assert that the resulting curve is contained inside the union of all the triangles of three consecutive control vertices of the control polygon. Using that property we can solve the problem of the collision, maintaining all the triangles associated to the control polygon free from collision with OTFs. Note that the CHP of quadratic B-splines is also valid in space, hence, the convex hull is still composed of triangles, like the faces of the obstacles. This simplifies all the checks for collisions because they are all between triangles in space and we can use the methods described in Section 3.5.

We design two different algorithms to approach the collision problem. The first solution, described in Section 5.2.1, implements a modified version of Dijkstra's algorithm that finds the shortest path from start to

4 B-spline curves with degree 2.

end in the graph such that all the triangles formed by three consecutive points in the path are free from collisions. The second solution, described in Section 5.2.2, uses the classical Dijkstra's algorithm to find the shortest path from s to e in the graph G, checking later for collisions in the triangles formed of three consecutive points in such path. When a collision is found we add vertices to the path to manage that.

5.2.1 *First solution: Dijkstra's algorithm in* G_t

The first solution of the problem exploits the graph G_t obtained as explained in Section 5.1.2. Before applying Dijkstra's algorithm to G_t all the triples are filtered checking if the triangle composed of the vertices of the triple intersects an OTF. If a triple intersects an obstacle then it is removed from the graph so that a path cannot pass from such vertices in that order.

Note that if a triple $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ is removed from V_t - and consequently also the triple $(\mathbf{c}, \mathbf{b}, \mathbf{a})$ - this does not necessarily exclude the three vertices $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ from being part of the final polynomial chain. For instance, in Fig. 14 we have a graph G with vertices $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{f}$ and an obstacle that intersects triples on the transformed graph[5] G_t. The triple $(\mathbf{a}, \mathbf{b}, \mathbf{c})$ and $(\mathbf{c}, \mathbf{b}, \mathbf{a})$ are removed from G_t because the corresponding triangle intersects the obstacle, and the path $\mathbf{d} \to \mathbf{a} \to \mathbf{b} \to \mathbf{c} \to \mathbf{e}$ cannot be admissible. This doesn't preclude the nodes $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ to be part of the final admissible path $\mathbf{d} \to \mathbf{a} \to \mathbf{b} \to \mathbf{e} \to \mathbf{c} \to \mathbf{f}$.

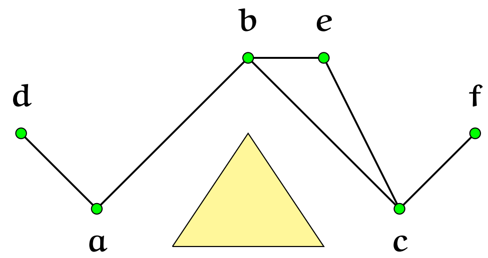

Figure 14.: Example of triples.

On the cleaned transformed graph it is possible to find the shortest path

$$P_t = (\mathbf{a}_0, \mathbf{b}_0, \mathbf{c}_0), (\mathbf{a}_1, \mathbf{b}_1, \mathbf{c}_1), \ldots, (\mathbf{a}_i, \mathbf{b}_i, \mathbf{c}_i), \ldots, (\mathbf{a}_n, \mathbf{b}_n, \mathbf{c}_n)$$

5 In the plane, this graph cannot be obtained using the procedure based on VDs explained in Section 3.3, but a similar situation is plausible considering Voronoi cells in space.

using an algorithm like Dijkstra. Then the shortest path P in G is constructed by taking the central vertex b_i of every triple (a_i, b_i, c_i) of P_t, plus the extremes a_0 and c_n of the first and last triple, obtaining

$$P = a_0, b_0, b_1, \dots, b_i, \dots, b_{n-1}, b_n, c_n.$$

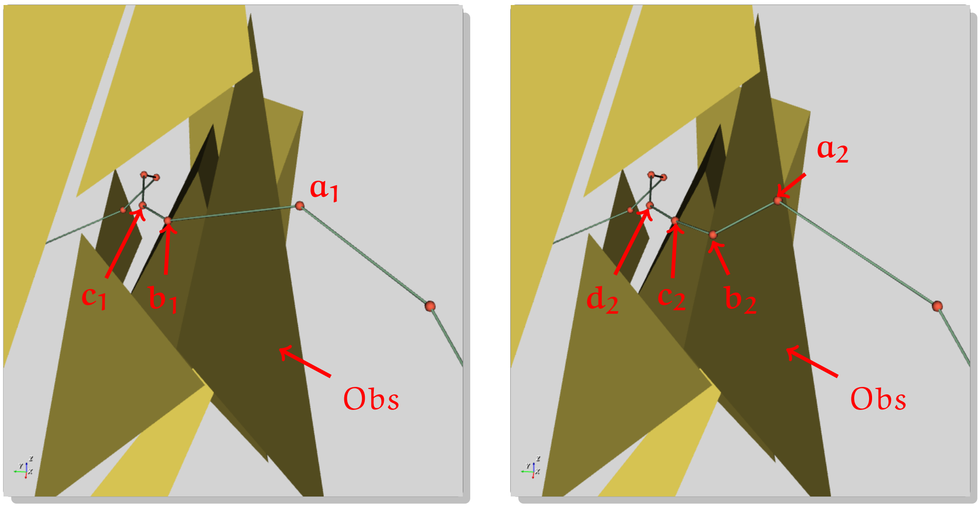

Figure 15.: Effects of application of solution one.

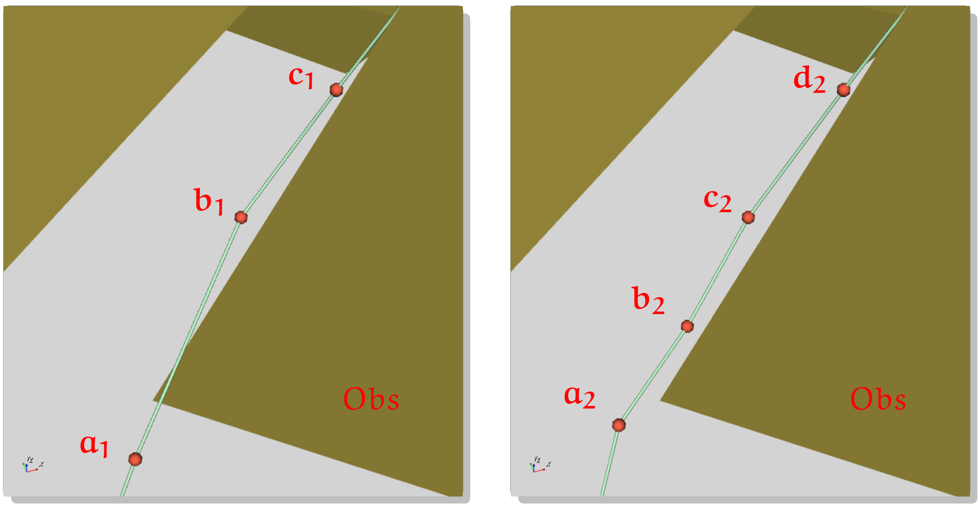

Figure 16.: Effects of application of solution one, other viewpoint.

In Fig. 15 and Fig. 16 the effect of the application of the first solution is shown. The triangle formed by the vertices a_1, b_1, c_1 in the left picture of

Fig. 15 is colliding with the obstacle Obs in the back. In the right picture there is the path $\mathbf{a}_2, \mathbf{b}_2, \mathbf{c}_2, \mathbf{d}_2$ obtained applying the solution - in this case no triangles in the path collide with obstacles. In Fig. 16 another point of view of pictures in Fig. 15 is visible.

5.2.1.1 *Complexity considerations*

For each triple and each OTF we need to solve three 3×3 linear systems for the collision check[6], hence, in total the cost is

$$\mathcal{O}(|V_t| \cdot |O|)$$

and for Eq. (21) and Eq. (24) this is equal to

$$\mathcal{O}(|O|^2 k^2). \tag{27}$$

The cost of applying Dijkstra's algorithm[7] in G_t is [1][26]

$$\begin{aligned} \mathcal{O}(|E_t| + |V_t| \log |V_t|) &= \mathcal{O}(k^3|V| + k^2|V| \log(k^2|V|)) \\ &= \mathcal{O}(k^3|O| + k^2|O| \log(k^2|O|)). \end{aligned} \tag{28}$$

Such cost has two special cases:

- if G is a *clique* - i.e. each node in V is connected to every other node [1] - then $k = |V| - 1$ and the cost is

  $$\mathcal{O}(|V|^4);$$

- if k is constant - i.e. doesn't grow with $|V|$ - the cost is

  $$\mathcal{O}(|V| \log |V|).$$

The latter case is the more plausible if we assume the hypothesis that no more than four input sites in space can be on the same sphere, in fact in that case every Voronoi cell cannot have a vertex with more than four edges connected to it (see Section 3.3 for details).

If we sum all the costs we obtain:

$$\mathcal{O}(k^2|O|^2 + k^3|O|) \tag{29}$$

6 See Section 3.5.3.
7 In the worst case where no triples are removed in the cleaning phase.

where all the other terms are absorbed in those two. If we have k constant, as we said before, then we have an overall cost of

$$\mathcal{O}(|O|^2) \tag{30}$$

that originates from the collision-check controls.

We can improve this result if we divide the algorithm in two parts:

1. first we can construct the graph with cost $\mathcal{O}(|O|^2)$;
2. then we can use the same graph in different situations[8] with cost $\mathcal{O}(|O| \log |O|)$, only for the routing.

Description	Cost	Reference
Creation of G	$\mathcal{O}(\|O\| \log \|O\|)$	Eq. (20)
Pruning of G	$\mathcal{O}(k\|O\|^2)$	Eq. (23)
Creation of G_t	$\mathcal{O}(k^3\|O\|)$	Eq. (26)
Pruning of G_t	$\mathcal{O}(\|O\|^2 k^2)$	Eq. (27)
Routing in G_t	$\mathcal{O}(k^3\|O\| + k^2\|O\| \log(k^2\|O\|)$	Eq. (28)
Total	$\mathcal{O}(k^2\|O\|^2 + k^3\|O\|)$	Eq. (29)
Total (k constant)	$\mathcal{O}(\|O\|^2)$	Eq. (30)

Table 2.: Summary of the costs for solution one

On Table 2 we summarize all the terms that contributes to the total costs, and the total cost itself.

5.2.2 *Second solution: Dijkstra's algorithm in* G

The First solution is interesting from an algorithmic point of view, but it is not very practical. It ignores all the triples that intersect an obstacle, thus possible paths in G are lost.

We develop a solution that uses another approach: obtain the shortest path from the Voronoi's graph G directly using Dijkstra's algorithm, without removing any triple. On this path - that we call P - we check every triple of consecutive vertices, and if it collides with an OTF then we take countermeasures (see Section 3.5.3 for the procedure implemented

8 With specific starting and ending points.

to identify collisions between two triangles). For instance, if the path is composed from the vertices

$$P = (\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_n)$$

then we check every one of the triangles

$$\begin{aligned} T_0 &= \triangle \mathbf{v}_0 \mathbf{v}_1 \mathbf{v}_2 \\ T_1 &= \triangle \mathbf{v}_1 \mathbf{v}_2 \mathbf{v}_3 \\ &\dots \\ T_i &= \triangle \mathbf{v}_i \mathbf{v}_{i+1} \mathbf{v}_{i+2} \\ &\dots \\ T_{n-3} &= \triangle \mathbf{v}_{n-3} \mathbf{v}_{n-2} \mathbf{v}_{n-1} \\ T_{n-2} &= \triangle \mathbf{v}_{n-2} \mathbf{v}_{n-1} \mathbf{v}_n \end{aligned}$$

for intersections with OTFs. $\triangle \mathbf{v}_i \mathbf{v}_j \mathbf{v}_k$ denotes the triangle having points $\mathbf{v}_i$, $\mathbf{v}_j$ and $\mathbf{v}_k$ as vertices.

Consider that G is pruned from all the edges that intersect any obstacle, thus none of the edges of the triangles T_i can intersect an OTF. The only possibility is that edges[9] of OTF intersect a triangle T_i. Hence for each T_i we have a (possibly empty) set of points of intersection between it and the edges of each OTF - we call that set O.

In Fig. 17 we have an example of the triangle

$$T_i = \triangle \mathbf{v}_i \mathbf{v}_{i+1} \mathbf{v}_{i+2}$$

that is intersected by obstacles in the points

$$O = \{\mathbf{o}_1, \mathbf{o}_2, \mathbf{o}_3\}.$$

Each one of the points in O is expressed in barycentric coordinates of the vertices $\mathbf{v}_i$, $\mathbf{v}_{i+1}$ and $\mathbf{v}_{i+2}$ of the triangle:

$$\begin{aligned} \mathbf{o}_1 &= \alpha_1 \mathbf{v}_i + \beta_1 \mathbf{v}_{i+1} + \gamma_1 \mathbf{v}_{i+2} \\ \mathbf{o}_2 &= \alpha_2 \mathbf{v}_i + \beta_2 \mathbf{v}_{i+1} + \gamma_2 \mathbf{v}_{i+2} \\ \mathbf{o}_3 &= \alpha_3 \mathbf{v}_i + \beta_3 \mathbf{v}_{i+1} + \gamma_3 \mathbf{v}_{i+2} \end{aligned}$$

where $\alpha_i + \beta_i + \gamma_i = 1$ for $i = 1, 2, 3$.

We want to avoid collisions adding vertices in the control polygon, such that consecutive triangles are free from obstacles. We obtain this by adding two new control vertices:

9 If we ignore special cases, two edges for each OTF at most.

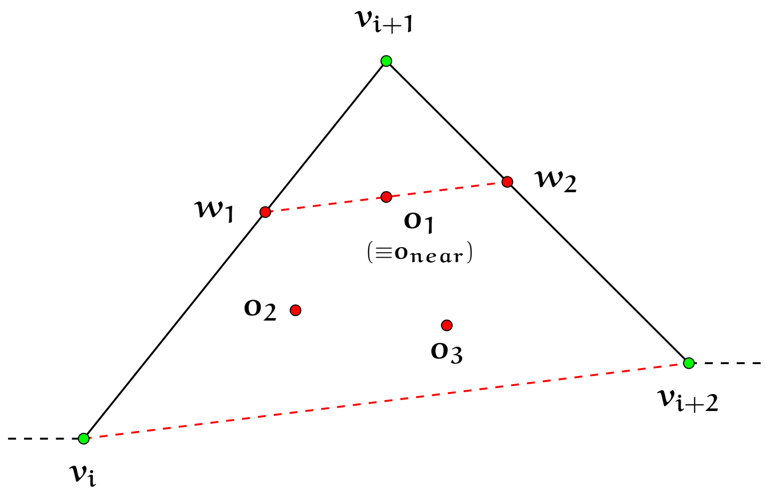

Figure 17.: $T_i (= \triangle v_i v_{i+1} v_{i+2})$ and the points o_1, o_2, o_3 of intersection between it and the edges of some OTFs.

- w_1 between v_i and v_{i+1};
- w_2 between v_{i+1} and v_{i+2}.

We add those points in a way that makes the segment $\overline{w_1 w_2}$ parallel to the segment $\overline{v_i v_{i+2}}$ and $\overline{w_1 w_2}$ passing just above the obstacle point o_{near} that is the nearest to v_{i+1} (o_1 in Fig. 17). The degenerate triangles $\triangle v_i w_1 v_{i+1}$ and $\triangle v_{i+1} w_2 v_{i+2}$, and the not degenerate triangle $\triangle w_1 v_{i+1} w_2$ replace the original triangle T_i. They are built in a way that do not make them collide with obstacles.

When we check for collisions between a segment and a triangle, we resolve a system of three unknowns in three equations and we extract the barycentric coordinates of the point of collision from the solutions. When we have all the coordinates of the points in O, we can obtain o_{near} by picking the one with the biggest β and then, using the corresponding β_{near}, we can obtain

$$
\begin{aligned}
W_1 &= \beta_{near} v_{i+1} + (1 - \beta_{near}) v_i \\
W_2 &= \beta_{near} v_{i+1} + (1 - \beta_{near}) v_{i+2}.
\end{aligned}
$$

In Fig. 18 and Fig. 19 we can see the effects of the application of this solution to a piece of the curve. The original pieces of control polygon are on the left pictures; the triangle composed of those vertices collide with the obstacle on the back. The two new vertices w_1 and w_2 are added to avoid the collision.

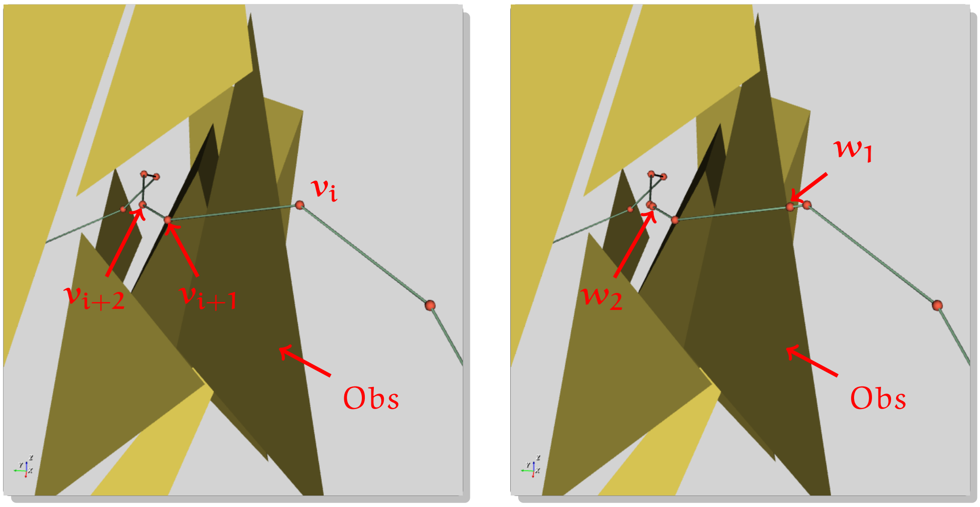

Figure 18.: Effects of application of solution two.

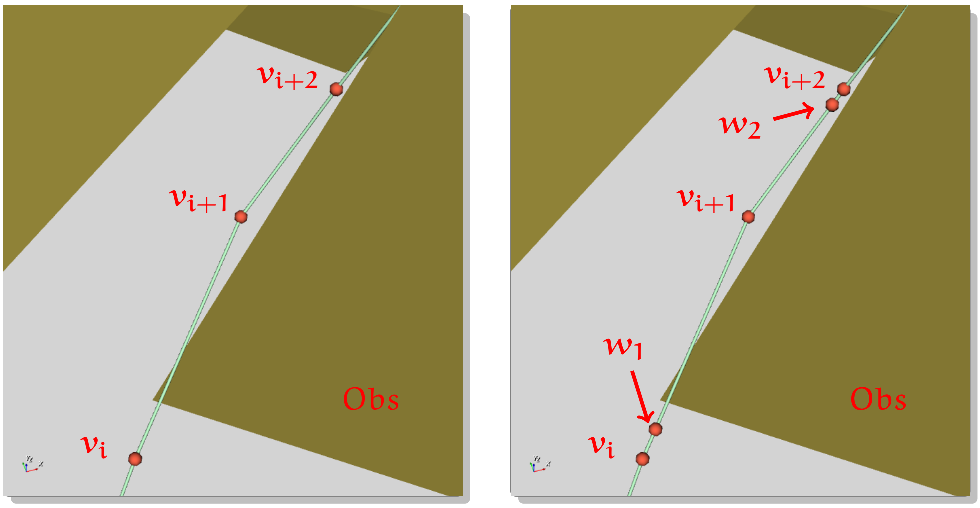

Figure 19.: Effects of application of solution two, other viewpoint.

5.2.2.1 *Complexity considerations*

For this solution we still have the costs of Eq. (20) and Eq. (23) for the creation and pruning of the graph G. In addition, we need to apply Dijkstra's algorithm in G to obtain P with a cost [1][26]

$$\mathcal{O}(|E| + |V| \log |V|).$$

For Eq. (21) and Eq. (22) this cost is equal to

$$\mathcal{O}(k|O| + |O| \log |O|) \tag{31}$$

and if we make the assumption of k constant we have a cost

$$\mathcal{O}(|O| \log |O|).$$

We need to consider every face of obstacles in O for every triangle in P to check and remove the collisions in the path. The cost to do this is[10] $\mathcal{O}(|P| \cdot |O|)$ where $|P|$ means the number of vertices in P. In the worst case $|P| = \mathcal{O}(|V|) = \mathcal{O}(|O|)$, hence we have a cost

$$\mathcal{O}(|P| \cdot |O|) = \mathcal{O}(|O|^2). \tag{32}$$

Summing up all the costs, we have

$$\mathcal{O}(k|O|^2) \tag{33}$$

and, if we consider k constant,

$$\mathcal{O}(|O|^2). \tag{34}$$

On Table 3 we summarize all the terms that contribute to the total cost, and the total cost itself.

The cost is comparable with the one of the first solution. Furthermore, in this case we can divide the algorithm in two parts:

1. first we can construct G with cost $\mathcal{O}(|O|^2)$;
2. then we can use it for different situations with cost $\mathcal{O}(|O| \log |O| + |P| \cdot |O|)$.

10 If #OTFs $= \mathcal{O}(|O|)$ - i.e. the number of OTFs does not grow faster than the number of obstacles.

Description	Cost	Reference
Creation of G	$\mathcal{O}(\|O\| \log \|O\|)$	Eq. (20)
Pruning of G	$\mathcal{O}(k\|O\|^2)$	Eq. (23)
Routing in G	$\mathcal{O}(k\|O\| + \|O\| \log \|O\|)$	Eq. (31)
Clean path	$\mathcal{O}(\|P\| \cdot \|O\|) = \mathcal{O}(\|O\|^2)$	Eq. (32)
Total	$\mathcal{O}(k\|O\|^2)$	Eq. (33)
Total (k costant)	$\mathcal{O}(\|O\|^2)$	Eq. (34)

Table 3.: Summary of the costs for solution two.

5.3 DEGREE INCREASE

We have hitherto assumed we are dealing only with quadratic B-splines - i.e. of degree 2 - because, for the CHP (Section 3.2.1), we need to check intersections only between two triangles (one belonging to P and the other to OTFs). If we want to use higher degree curves, we can modify the previous algorithms to deal with polyhedral convex hulls, but this implies an increase in complexity.

We are interested in increasing the degree to achieve smooth curves with continue curvature and torsion. We adopt a compromise: we adapt the path obtained from the previous algorithms adding vertices and forcing the curve to remain in the same convex hull. However, this approach have the disadvantage that we cannot achieve a good torsion[11] because the curve changes plane in an inflection point of the curvature.

We modify

$$P = (v_0, \dots, v_n)$$

adding a certain number of aligned new vertices $(w_0, w_1, \dots)$ between each pair (v_i, v_{i+1}) of vertices in P for $i = 0, \dots, n-1$. The number of w_j between each pair (v_i, v_{i+1}) depends on the desired grade of the curve. In fact we need $m-2$ new vertices between each (v_i, v_{i+1}) for B-spline curves of degree m. Thus the final modified path for a B-spline curve of degree $m \geqslant 3$ is

$$\tilde{P} = (v_0, w_0, \dots, w_{m-3}, v_1, \dots, v_i, w_{i(m-2)}, \dots, w_{(i+1)(m-2)-1}, v_{i+1}, \dots, v_n).$$

This strategy is used in this project only to lift the degree from 2 to 3 or 4.

11 We can improve this with the post process.

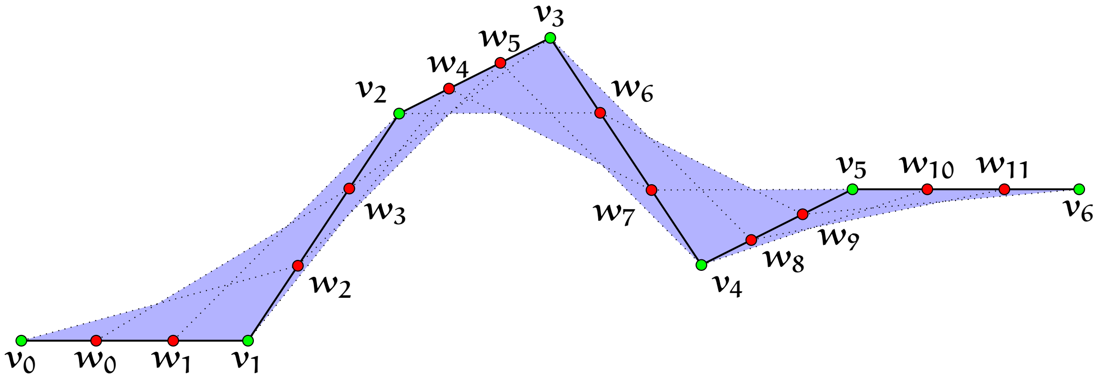

Figure 20.: Increase the degree m from 2 to 4.

An example of path

$$P = (v_0, v_1, v_2, v_3, v_4, v_5, v_6)$$

is visible in Fig. 20. We have the vertices of P in green, the added vertices in red and the cyan area is the convex hull of the final curve.

We want to adapt P to quartic B-spline curves, hence we need to add two new vertices between each pair of vertices (v_i, v_{i+1}) for $i = 0, \ldots, 6$. Those new vertices are

$$(w_0, w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8, w_9, w_{10}, w_{11}).$$

Note that, with this algorithm, when we increase the degree from 2 to $m \geqslant 3$ we have that the convex hull containing a B-spline curve of degree m in $\tilde{P}$ is a subset of the convex hull containing a B-spline curve of degree 2 in P. This happens because the polyhedrons of consecutive $m+1$ vertices in $\tilde{P}$ collapse in triangles contained inside the triangles of consecutive vertices in P. For instance, in Fig. 20 the convex hull of the first 5 vertices v_0, w_0, w_1, v_1, w_2 of $\tilde{P}$ coincides with the triangle $\triangle v_0 v_1 w_2$ that is contained inside the triangle $\triangle v_0 v_1 v_2$ of the first 3 vertices of P.

One effect of the application of this method is that a curve of degree m in $\tilde{P}$ touches every segment of the original control polygon P. This is because adding $m-2$ aligned vertices between each pair (v_i, v_{i+1}) will result in m aligned vertices on each original segment (Section 3.2.2).

5.4 KNOTS SELECTION

In the previous sections we never discuss the criterion adopted to determine the extended knot vector T

$$T = \{t_0, \dots, t_{m-1}, t_m, \dots, t_{n+1}, t_{n+2}, \dots, t_{n+m+1}\}$$

associated to the B-spline curve.

In this section we discuss two methods implemented to establish T.

First of all, we want the curve to interpolate the chosen start and end points that correspond to the extremes $\mathbf{v}_0$ and $\mathbf{v}_n$ of the extracted path P. We see in Section 3.2.4 that we can achieve such interpolation if we impose

$$\begin{aligned} &t_0 = t_1 = \cdots = t_m = a \\ &t_{n+1} = t_{n+2} = \cdots = t_{n+m+1} = b \end{aligned} \tag{35}$$

where a and b are the extremes of the parametric domain of the curve.

The constraint of Eq. (35) is a mandatory choice, thus we cannot change it. Regarding the parametric domain, we choose it to be $[0, 1]$ because changing the extremes do not change the behavior of the curve, only changing the ratios of the distances between the knots is effective [12]. We still need to chose how to select the inner $n - m$ knots $t_{m+1}, \dots, t_n$, and we develop two different ways to do this:

method 1 Use a uniform partition where $t_i - t_{i-1} = c$ for $i = m + 1, \dots, n + 1$ for c constant;

method 2 Use an adaptive partition, where we try to create dense knots in correspondence of points on the curve where we have dense control vertices.

method 1 is the easiest way to choose a knot vector and it is a common first choice in textbooks [12][11], but it has the disadvantage of ignoring the geometry of the curve [12]. The steps to accomplish **method 1** are quite straightforward: we need to pick the nodes

$$\frac{i}{n - m + 1}$$

for $i = 1, \dots, n - m$. Thus, we concentrate on **method 2**.

We start from the idea that if we have a control polygon with uniformly-spaced vertices - i.e. $\|\mathbf{v}_1 - \mathbf{v}_0\|_2 = \|\mathbf{v}_2 - \mathbf{v}_1\|_2 = \cdots = \|\mathbf{v}_n - \mathbf{v}_{n-1}\|_2$ - then

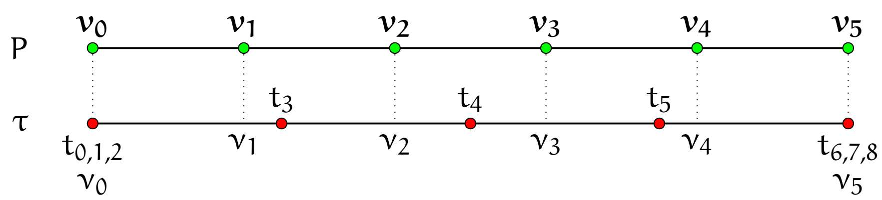

Figure 21.: Optimal case for a quadratic curve (we want uniform partition).

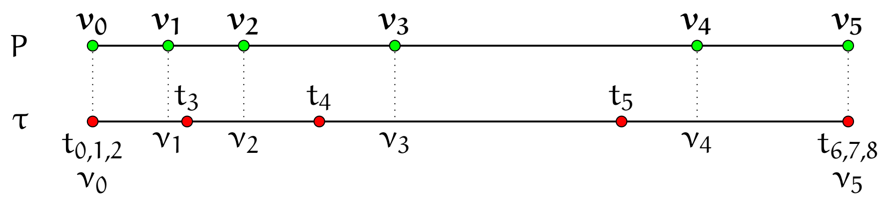

Figure 22.: General case for a quadratic (same distances between t_i and enclosing v_j, v_{j+1} as Fig. 21).

we agree on a uniform partition of the knots ($t_{m+1} - t_m = t_{m+2} - t_{m+1} = \cdots = t_{n+1} - t_n$). In Fig. 21 there is an example of a quadratic B-spline curve with uniformly-spaced control polygon. The above segment is a *rectified* visualization of the control polygon with six control vertices $\mathbf{v}_0, \ldots, \mathbf{v}_5$. The segment below represents the partition of the domain from a (on the left) to b (on the right), with the projections $v_0, \ldots, v_5$ of the control vertices, scaled in length to the parametric domain axis[12], and the knots $t_0, \ldots, t_8$ on it.

Starting from this situation, if we have a generic control polygon with segments of different length, as in Fig. 22, then we want each t_i to keep the same distance, in ratio, between the surrounding v_j and v_{j+1}, respect to the optimal case. For instance, in Fig. 21 $\frac{t_3 - v_1}{v_2 - v_1} = \frac{1}{4}$ and $\frac{v_2 - t_3}{v_2 - v_1} = \frac{3}{4}$, this means that in Fig. 22 the same values must be preserved.

The problem now is how to calculate the values of t_i in the general case. We consider only the inner part τ of the partition vector, included the extremes

$$\tau_i = t_{i+m} \qquad i = 0, \ldots, n - m + 1$$

where $\tau_0 = a = 0$ and $\tau_{n-m+1} = b = 1$. In Fig. 21 and Fig. 22 $\tau = (t_2, t_3, t_4, t_5, t_6)$. Now we calculate the positions of all τ_i respect to the

12 $\mathbf{v}_0$ is projected to a, $\mathbf{v}_5$ is projected to b, and the ratios between the distances between vertices are preserved.

ν_j in the optimal case. We can achieve that using as unit the uniform distance $\nu_j - \nu_{j-1}$ to calculate the positions of τ_i. We specifically calculate

$$\tau_i^\nu = \frac{n}{n-m+1} \cdot i \qquad i = 0, \dots, n-m+1 \tag{36}$$

obtaining the numbers τ_i^ν whose integer part $\lfloor \tau_i^\nu \rfloor$ represents the index j of the ν_j that is to the left of τ_i, and the decimal part $(\tau_i^\nu - \lfloor \tau_i^\nu \rfloor)$ represents the distance from it: $\frac{\tau_i - \nu_j}{\nu_{j+1} - \nu_j}$.

Now we calculate the projections ν_i in the *generic* case. We start calculating the incremental distances between the vertices

$$\begin{cases} d_0 = 0 \\ d_i = d_{i-1} + \|\boldsymbol{v}_i - \boldsymbol{v}_{i-1}\|_2 \qquad i = 1, \dots, n \end{cases}$$

and, remembering that the parametric domain is $[0, 1]$, we have

$$\nu_i = \frac{d_i}{d_n} \qquad i = 0, \dots, n. \tag{37}$$

Using the positions in Eq. (36) on the projection in Eq. (37), we obtain the values

$$\tau_i = \nu_{\lfloor \tau_i^\nu \rfloor} + (\tau_i^\nu - \lfloor \tau_i^\nu \rfloor)(\nu_{\lfloor \tau_i^\nu \rfloor + 1} - \nu_{\lfloor \tau_i^\nu \rfloor}) \qquad i = 0, \dots, n-m+1.$$

Finally, adding the duplicated knots, we obtain

$$t_i = \tau_{\min(n-m+1,\ \max(0,\ i-m))} \qquad i = 0, \dots, n+m+1.$$

5.5 POST PROCESSING

The purpose of the post processing phase is to try to simplify the path $P = (\boldsymbol{v}_0, \dots, \boldsymbol{v}_n)$ obtained in the previous phase removing useless vertices, in order to achieve a smoother path.

To obtain this, we realize Algorithm 8 that iterates through all the vertices, except the extremes, and checks if each $\boldsymbol{v}_i$ can be removed without consequences. With consequences we mean that removing $\boldsymbol{v}_i$ would cause a triangle in P to intersect one of the OTFs.

To clarify the concept, consider the simplification in 2-dimensional space in Fig. 23. The path to process is

$$P = (\dots, \boldsymbol{v}_{i-2}, \boldsymbol{v}_{i-1}, \boldsymbol{v}_i, \boldsymbol{v}_{i+1}, \boldsymbol{v}_{i+2}, \dots)$$

Algorithm 8 Post processing algorithm on path P.

```
procedure POSTPROCESS(P)
    for i ← 1, n − 1 do
        if i = 1 or not intersectOTF(△v_{i−2} v_{i−1} v_{i+1}) then
            if i = n − 1 or not intersectOTF(△v_{i−1} v_{i+1} v_{i+2}) then
                P ← P \ {v_i}
            end if
        end if
    end for
end procedure
```

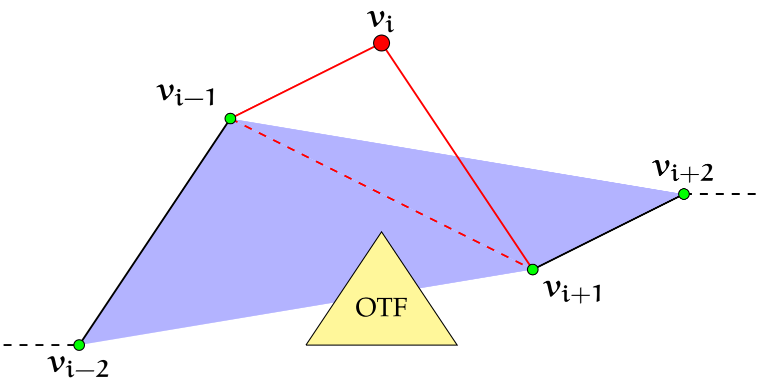

Figure 23.: Example of post process check that removes v_i.

and we are considering removing v_i obtaining a modified path

$$\tilde{P} = (\ldots, v_{i-2}, v_{i-1}, v_{i+1}, v_{i+2}, \ldots).$$

Before doing, this we need to check if any triangle in $\tilde{P}$ intersects any OTF. In detail, we need to check only the two triangles $\triangle v_{i-2}v_{i-1}v_{i+1}$ and $\triangle v_{i-1}v_{i+1}v_{i+2}$ because the other triangles in $\tilde{P}$ are already present in P. For instance the obstacle in the figure do not intersects any of the triangles in P, but it intersects $\triangle v_{i-2}v_{i-1}v_{i+1}$ in $\tilde{P}$.

5.5.1 *Complexity considerations*

We need to check if two triangles intersect with an OTF for every vertex of P, hence we have a complexity of

$$\mathcal{O}(|P| \cdot |O|) = \mathcal{O}(|O|^2)$$

where O is the set of obstacles and $|P|$ is the number of vertices in P.

5.6 THIRD SOLUTION: SIMULATED ANNEALING

The solutions in Section 5.2.1 and Section 5.2.2 have two problems in common:

- both reject configurations in a prudent way considering only the control polygon;
- and both do not optimize neither length nor other quantities.

These solutions have also the following benefits:

- they produce paths that are obstacle-free from construction;
- the application of the post-processing often produces a reduction in the curve length.

In this section, we describe a third approach based on probabilistic computation.

We can consider the problem of finding the shortest path as a constrained optimization problem, in which a certain configuration of the control vertices (and consequently the B-spline) is the state of the system, and we aim to minimize both the length of the control polygon (and therefore the B-spline[13]) and the peak in curvature and torsion of the B-spline, under the constraint that the B-spline must not intersect the obstacles. We are interested in optimizing the length of the curve and the maximum peaks of both curvature and torsion, because we want a path that is short but also fair.

5.6.1 *Lagrangian Relaxation (LR) applied to the project*

We can apply the concept explained in Section 3.4.4 to the project.

The variable space X is composed of all the possible configurations of the path, or, in other words,it is defined by all the possible values of the vector $P = (v_1, \dots, v_n)$ of all n ordered vertices $v_i = (x_i, y_i, z_i)$ of the path. The Problem 13 can be formulated as follows:

$$\begin{aligned} \underset{P}{\text{minimize}} \quad & \alpha \cdot \mathrm{maxCurv}(P) + \beta \cdot \mathrm{maxTors}(P) + \gamma \cdot \mathrm{normLen}(P) \\ \text{subject to} \quad & \left| \mathrm{bspline}(P) \cap \bigcup_{i \in I} \mathrm{obstacle}_i \right| = 0, \end{aligned}$$

where $\mathrm{maxCurv}(P)$ is the curvature peak of the B-spline constructed using P as control polygon, $\mathrm{maxTors}(P)$ is the absolute value of the torsion peak and $\mathrm{normLen}(P)$ is the length of the control polygon P normalized as a percentage of the length of the initial status[14] . α, β and γ are fixed coefficients used to give different weights to the curvature peak, torsion peak and length during the optimization process. The normalization of length is necessary to decouple the weight of the length from the length of the path.

Curvature and torsion are obtained in a discrete form. The B-spline curve is tabulated in a number of points that depends on the length of P by a multiplied constant, then for each point the curvature and torsion values are calculated.

Regarding the constraint, $\mathrm{bspline}(P)$ is the set of points of the *B-spline*, using P as control polygon, and $\mathrm{obstacle}_i$ is the area of the i^{th} of m obstacles, and $I = \{1, \dots, m\}$.

---

13 We give to the users also the possibility of selecting the arc length as the quantity to minimize.

14 If the user chooses to minimize the arc length, then $\mathrm{normLen}(P)$ becomes the length of the B-spline curve.

Thus, we need to build the Lagrangian function corresponding to Eq. (14). The constraint function is not negative and is calculated as the ratio

$$\mathrm{constraint}(P) = \frac{|\{\mathbf{p} \in \mathrm{spline}(P) \ : \exists i \text{ s.t. } \mathbf{p} \in \mathrm{obstacle}_i\}|}{|\{\mathbf{p} \in \mathrm{spline}(P)\}|}. \quad (38)$$

The points $\mathbf{p}$ of the spline are calculated in a discrete form, like curvature and torsion. Thus, the constraint depends on the tabulation of the curve and it is also possible to have borderline cases where the constraint does not reflect the real situation[15].

The function in Eq. (38) is not negative, thus the Lagrangian function, corresponding to Eq. (14), is

$$L_d(P, \lambda) = \mathrm{gain}(P) + \lambda \cdot \mathrm{constraint}(P) \quad (39)$$

where, for convenience,

$$\mathrm{gain}(P) = \alpha \cdot \mathrm{maxCurv}(P) + \beta \cdot \mathrm{maxTors}(P) + \gamma \cdot \mathrm{normLen}(P). \quad (40)$$

5.6.2 *Annealing phase*

The purpose of the simulated annealing phase is to find the minimum saddle point in the curve represented by the Eq. (39).

The Algorithm 9 is the annealing process, and its input is the initial status of the system $\mathbf{x}$ - i.e. the initial configuration of the control polygon. It operates this way:

1. λ and the temperature are initialized on Line 2 and Line 3 respectively;
2. the *while* on Line 4 is the main loop and the terminating condition is given by a minimum temperature or a minimum variation of energy between two iterations;
3. the *for* at Line 5 repeats the annealing move for a certain number of trials, on each iteration the algorithm probabilistically tries to make a move of the state of the system;
    - first, on Line 6, it chooses between moving in the Lagrangian space or in the space of the path;

15 For instance, if we have very thin obstacles, a curve can pass through them having only few points (or even none) inside.

Algorithm 9 Annealing

1: **procedure** ANNEALING(x)
2: $\lambda \leftarrow \mathit{initialLambda}$
3: $T \leftarrow \mathit{initialTemperature}$
4: **while** not $\mathit{terminationCondition}()$ **do**
5: **for all** number of trials **do**
6: $\mathit{changeLambda} \leftarrow \mathit{True}$ with $\mathit{changeLambdaProb}$
7: **if** $\mathit{changeLambda}$ **then**
8: $\lambda' \leftarrow \mathit{neighbour}(\lambda)$
9: $\lambda \leftarrow \lambda'$ with probability $e^{-([\mathit{energy}(x,\lambda)-\mathit{energy}(x,\lambda')]^{+}/T)}$
10: **else**
11: $x' \leftarrow \mathit{neighbour}(x)$
12: $x \leftarrow x'$ with probability $e^{-([\mathit{energy}(x',\lambda)-\mathit{energy}(x,\lambda)]^{+}/T)}$
13: **end if**
14: **end for**
15: $T \leftarrow T \cdot \mathit{warmingRatio}$
16: **end while**
17: **end procedure**

- after that, based on the previous choice, the algorithm probabilistically tries to move the system in a neighbouring state: in the Lagrangian space at Line 8 or in the path space at Line 11;

4. finally, at the end of every trial set, at Line 15, the temperature T is cooled by a certain factor.

The termination condition in Line 4 is triggered by a minimum variation of energy $\Delta\mathit{energy}$ between two consecutive iterations of the cycle. The termination is also triggered when a minimum temperature is reached, this happens to impose a limit on the number of cycles.

The choice of the neighbour is probabilistic. If the energy increases in the Lagrangian space or decreases in the path space, then the probability of choosing the new state is 1. If the energy decreases in the Lagrangian space or increases in the path space, then the new state is accepted with a probability that is:

$$\exp\left(-\frac{\Delta\mathit{energy}}{T}\right).$$

The $\mathit{neighbour}$ function depends on the input:

- a neighbour of λ is a value that is equal to λ plus a uniform perturbation in range $[-\mathit{maxLambdaPert}, \mathit{maxLambdaPert}]$;

- a neighbour of the path is obtained by randomly picking one of the vertices v_i (except the extremes v_0 and v_n), then uniformly choosing a direction and a distance in a specific range and, finally, moving v_i by the chosen values.

The $energy$ function is equivalent to L_d in the Eq. (39):

$$energy(x, \lambda) = gain(P) + \lambda \cdot constraint(P). \tag{41}$$

The annealing process finds a saddle point by probabilistically increasing the energy when λ is moved, and decreasing the energy when the points are moved.

5.6.2.1 *Complexity considerations*

For this solution, we still have the costs of Eq. (20) and Eq. (23) for the creation and pruning of the graph G. In addition, we need to apply Dijkstra's algorithm in G to obtain the initial path P with the cost of Eq. (31).

Regarding the annealing phase, for each *step* (an iteration of Line 4 in Algorithm 9) we have a fixed number of *trials* (the iterations of Line 5). For each trial, we need to calculate the value of the energy of Eq. (41) that is the sum of the gain and the constraint.

For the gain of Eq. (40) we need to calculate the values of curvature and torsion for every tabulated point. Furthermore, there is also a cost[16] of $\mathcal{O}(|P|)$ to calculate the length of the control polygon. Thus, the cost for calculating the gain is

$$\mathcal{O}(|Sp| + |P|)$$

where Sp is the set of the tabulated points of the curve. The number of points in Sp depends on the length of the control polygon $len(P)$. Thus, we have a cost of $\mathcal{O}(len(P) + |P|)$, but in the worst case $|P| = \mathcal{O}(|V|) = \mathcal{O}(|O|)$, thus the cost is

$$\mathcal{O}(len(P) + |P|) = \mathcal{O}(len(P) + |O|). \tag{42}$$

In regards to the constraint of Eq. (38), we need to calculate if every point of the curve is inside an obstacle. This means a cost of

$$\mathcal{O}(len(P)|O|). \tag{43}$$

16 Only if the users do not choose to minimize the arc length.

Hence, the total cost for the calculation of the annealing phase is

$$\mathcal{O}(\#\mathtt{steps} \cdot \#\mathtt{trials} \cdot (\mathtt{len}(P)|O|)),$$

but the number of steps and trials are bounded by constants[17]. Thus, the cost becomes

$$\mathcal{O}(\mathtt{len}(P)|O|). \tag{44}$$

The total cost for the solution is

$$\mathcal{O}(k|O|^2 + \mathtt{len}(P)|O|). \tag{45}$$

Similarly to the previous solutions, if we have that k is a constant then the total cost becomes

$$\mathcal{O}(|O|^2 + \mathtt{len}(P)|O|). \tag{46}$$

Description	Cost	Reference
Creation of G	$\mathcal{O}(\lvert O\rvert \log \lvert O\rvert)$	Eq. (20)
Pruning of G	$\mathcal{O}(k\lvert O\rvert^2)$	Eq. (23)
Routing in G	$\mathcal{O}(k\lvert O\rvert + \lvert O\rvert \log \lvert O\rvert)$	Eq. (31)
Gain	$\mathcal{O}(\mathtt{len}(P) + \lvert P\rvert) = \mathcal{O}(\mathtt{len}(P) + \lvert O\rvert)$	Eq. (42)
Constraint	$\mathcal{O}(\mathtt{len}(P)\lvert O\rvert)$	Eq. (43)
Annealing	$\mathcal{O}(\mathtt{len}(P)\lvert O\rvert)$	Eq. (44)
Total	$\mathcal{O}(k\lvert O\rvert^2 + \mathtt{len}(P)\lvert O\rvert)$	Eq. (45)
Total (k costant)	$\mathcal{O}(\lvert O\rvert^2 + \mathtt{len}(P)\lvert O\rvert)$	Eq. (46)

Table 4.: Summary of the costs for solution three

In Table 4 we summarize all the costs. It is difficult to quantitatively compare the cost of this solution with the previous ones. This is due to the presence of the factor $\mathtt{len}(P)$ that depends on the geometry of the scene. however, we can affirm that this solution is more complex than the previous two by a term $\mathcal{O}(\mathtt{len}(P)|O|)$.

Furthermore, in this solution we can divide the algorithm in two parts:

1. first we can construct G with cost $\mathcal{O}(|O|^2)$;
2. then we can use it for different scenarios with cost $|O| \log |O| + \mathtt{len}(P)|O|$.

17 Although such constants can be very high.

Part III

EVALUATION

6

CODE STRUCTURE

We designed the code with an Object Oriented Programming (OOP) methodology in Python 3 (`https://www.python.org/`). A versatile language with a strong appeal on scientific community, easy to learn and with an increasing active community of developers behind. We relied on SciPy (`https://www.scipy.org/`) and NumPy (`http://www.numpy.org/`) libraries for taking care of different numerical methods. Furthermore we used NetworkX library (`http://networkx.github.io/`) to represent graphs and to route in them. Regarding the graphic output we used VTK (`http://www.vtk.org/`) bindings in Python.

The main class is `Voronizator`, it maintains the status of the scene and provides all the methods for the interface with users.

- `addBoundingBox` is used for adding a bounding box at specified coordinates to the scene.
- `addPolyhedron` is used for adding a new obstacle to the scene, it required a `Polyhedron` object as argument.
- `setPolyhedronsSites` add the sites for the VD to the scene.
- The method `makeVoroGraph` is used for creating the graphs G and G_t using the algorithms described in Section 5.1.1 and Section 5.1.2.
- `setAdaptivePartition` and `setBsplineDegree` are used for selecting between uniform or adaptive knot partition, and for choosing the desired degree of the curve.
- `extractXmlTree` and `importXmlTree` are used for saving and restoring the scene in XML format.
- All the other methods `plot*` are used for drawing the different elements of the scene. They require a plotter as argument.

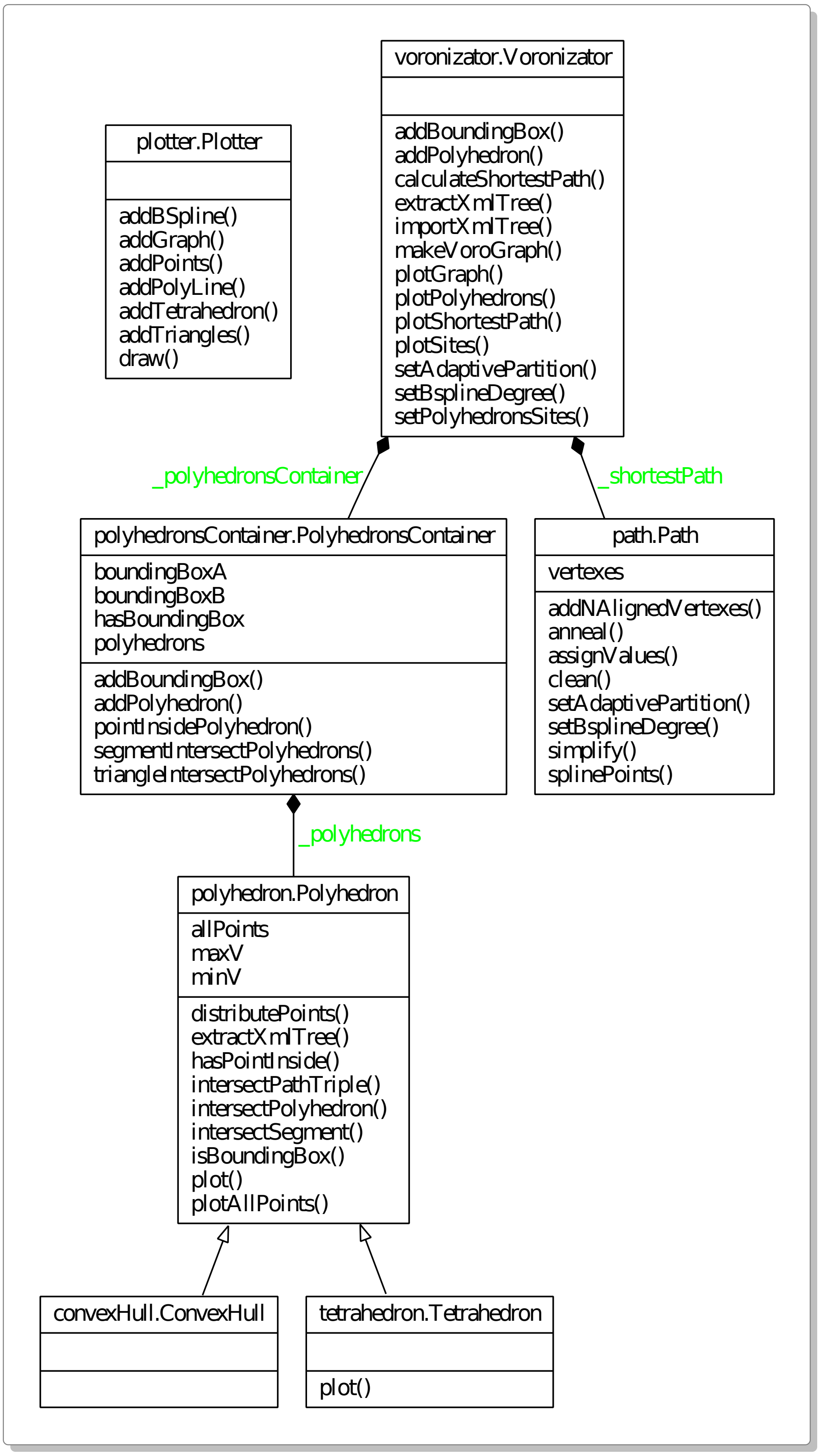

Figure 24.: Excerpt UML of the project

The class `Plotter` provides the interface for drawing all the necessary elements using VTK.

The class `Polyhedron` represents a single obstacle (that can also be one of the two subclasses `ConvexHull` and `Tetrahedron`). it provides all the necessary methods for performing geometry checks of point inclusion and intersection with segments, triangles, and other obstacles.

The class `Path` represents the control polygon of the curve. It provides the methods `addNAlignedVertexes` and `simplify` that perform respectively the degree increase (Section 5.3) and the post processing (Section 5.5). The method `clean` is necessary for the second solution described in Section 5.2.2. Furthermore this class provides also the functionality for optimizing the curve using the SA (Section 5.6) with the method `anneal`.

Furthermore we provide scripts for the creation of random scenes and for the execution of the different methods.

7

TESTING

We execute the tests summarized in Tables 7, 8, 9 and10 for evaluating the algorithms of the project. The focus of the testing phase is to assess the validity of the different algorithm, thus we present a detailed series of tests trying to cover all the functionalities.

Each table presents the following fields.

- *Scene* specifies the considered scene among those listed in Tables 7, 8, 9 and10;
- $\mathbf{s} \to \mathbf{e}$ indicates the starting and ending points
- *Deg.* is the degree of the B-spline curve;
- *Met.* is the method used, where
  - Method A is Dijkstra in G';
  - Method B is Dijkstra in G;
  - Method C is Simulated Annealing;
- *P. p.* indicates if the post processing is used (✓) or not (✗);
- *Part.* indicates if the uniform knot partition (**U**) or the adaptive one (**A**) is used;
- *Config.* is used only for method C and indicates the used annealing configuration among those listed in Table 6.

Table 5 gives some details about the scenes. The fields are the followings.

- *Scene* specifies the name of the scene
- *B.b. A* and *B.b. B* are the extremes of the bounding box.

- *Obs. shapes* indicates the shape of the obstacles in the scene[1];
- # *obs.* is the number of obstacles in the scene.
- *Max. empty area* is the maximum empty area for the distribution of the Voronoi sites on the OTFs of the obstacles (see Section 5.1.1);
- *Figure* is the reference to the figure of the scene that contemplates also the graph G.

Table 6 indicates the configurations for the SA phase. The fields are the followings:

- *Config.* specifies the name of the configuration set;
- T_0 is the initial *temperature*;
- *Trials* is the number of trials for each annealing cycle;
- *warm.* is the warming ratio of temperature between two consecutive cycles;
- *min* T is the minimum temperature at which the process terminates;
- *min* ΔE is the minimum difference of energy between two consecutive cycles at which the process terminates;
- λ *pert* is the maximum perturbation of λ in every move;
- V *pert fact* is the maximum perturbation of a path vertex in every move, expressed in fraction of the control polygon length;
- λ_0 is the initial value of λ;
- λP is the probability of changing λ instead the path in each move;
- *Len type* indicates if it is considered the control polygon (poly) or the arc (arc) length as optimizing quantity;
- *Ratios* is a triple of *weights* that indicates the importance, during the optimization, of curvature, torsion and length respectively;

All the results of the tests are visible in the figures presented in Appendix A. The used visualization for the tests with scene 2 is different from the others to enhance the visualization of the curve. Only the edges of the obstacles are drawn.

1 Scene 3 has only one bucket-shaped obstacle with center in $[0.5, 0.5, 0.5]$, with width 0.2, height 0.4 and thickness 0.02.

Scene	B.b. A	B.b. B	Obs. shape	# obs.	Max. empty area	Figure
1	$[-0.1, -0.1, -0.1]$	$[1.1, 1.1, 1.1]$	Tetrahedrons	10	0.1	Fig. 25
1b	$[-0.1, -0.1, -0.1]$	$[1.1, 1.1, 1.1]$	Tetrahedrons	10	0.01	Fig. 26
2	$[-0.1, -0.1, -0.1]$	$[1.1, 1.1, 1.1]$	Tetrahedrons	100	0.1	Fig. 27
3	$[0, 0, 0]$	$[1, 1, 1]$	Polyhedron	1	0.1	Fig. 28

Table 5.: Testing scenes.

Config.	T_0	Trials	warm.	min T	min ΔE	λ pert	V pert fact	λ_0	λP	Len type	Ratios
1	10	10	0.7	$1e-7$	$1e-6$	1000	10	0	$5e-2$	arc	$[0.1, 0.1, 0.8]$
2	10	10	0.7	$1e-7$	$1e-6$	1000	10	0	$5e-2$	poly	$[0.1, 0.1, 0.8]$
2b	10	100	0.7	$1e-7$	$1e-6$	1000	100	0	$5e-2$	poly	$[0.1, 0.1, 0.8]$
3	10	10	0.7	$1e-7$	$1e-6$	1000	10	0	$5e-2$	arc	$[0.3, 0.3, 0.4]$
3b	10	10	0.9	$1e-5$	$1e-6$	1000	100	0	$5e-2$	arc	$[0.3, 0.3, 0.4]$

Table 6.: Annealing configurations.

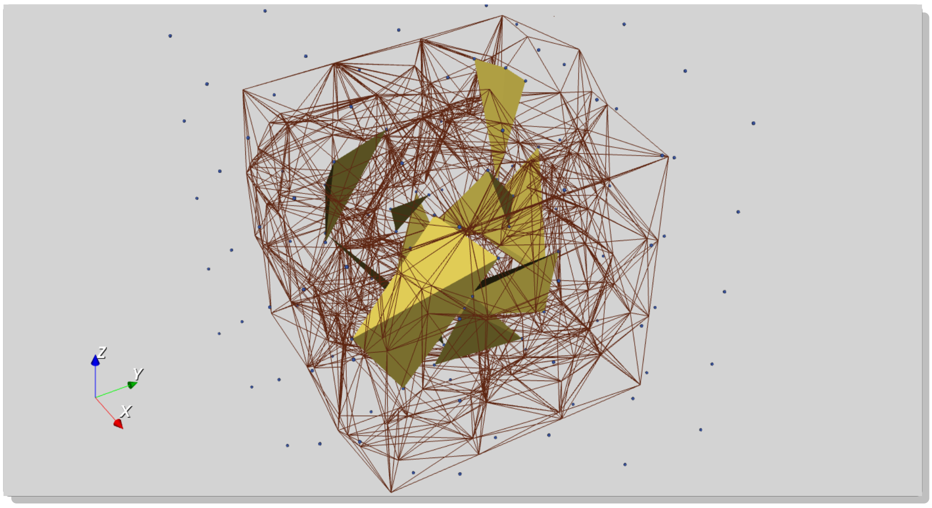

Figure 25.: Scene 1.

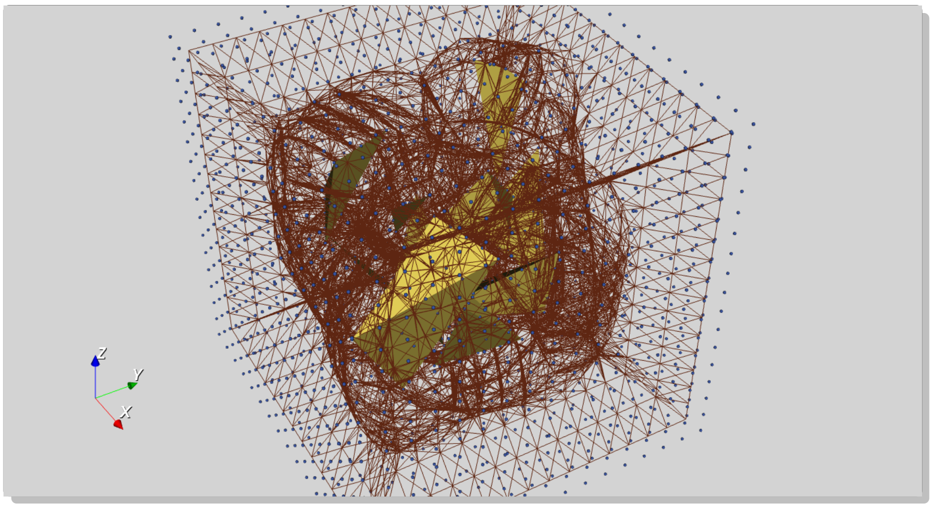

Figure 26.: Scene 1b.

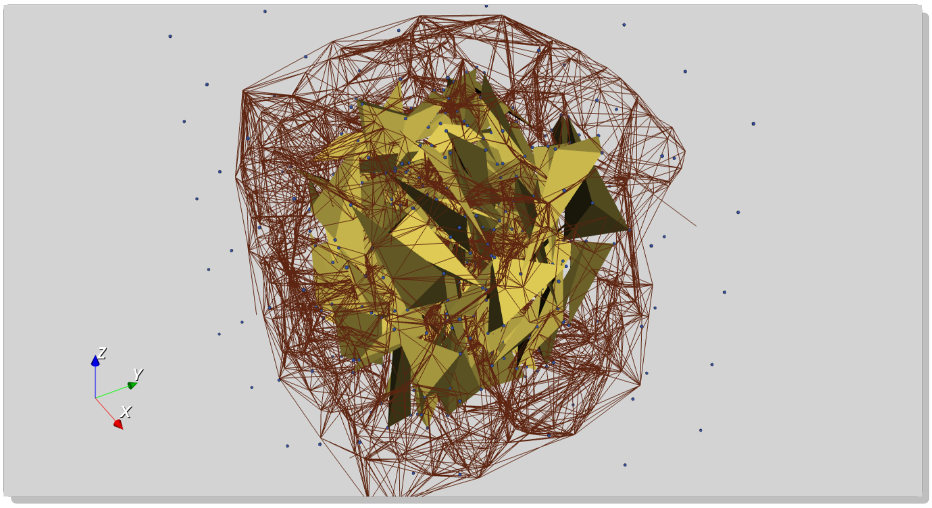

Figure 27.: Scene 2.

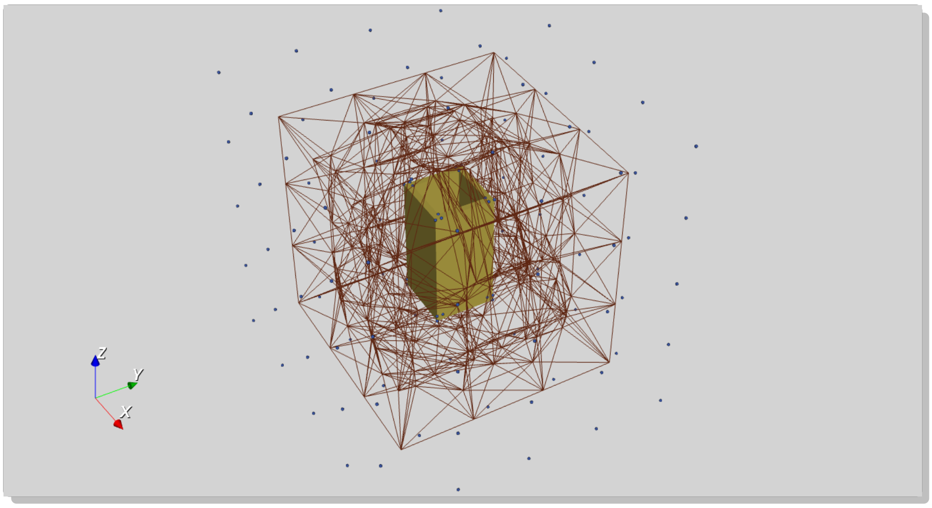

Figure 28.: Scene 3.

#	Scene	$s \to e$	Deg.	Met.	P. p.	Part.	Config.	figure
1	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	A	✗	**U**	-	??
2	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	A	✓	**U**	-	??
3	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	A	✗	**A**	-	??
4	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	A	✓	**A**	-	??
5	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	A	✗	**U**	-	??
6	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	A	✓	**U**	-	??
7	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	A	✗	**A**	-	??
8	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	A	✓	**A**	-	??
9	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	A	✗	**U**	-	??
10	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	A	✓	**U**	-	??
11	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	A	✗	**A**	-	??
12	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	A	✓	**A**	-	??
13	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	B	✗	**U**	-	??
14	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	B	✓	**U**	-	??
15	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	B	✗	**A**	-	??
16	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	B	✓	**A**	-	??
17	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	B	✗	**U**	-	??
18	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	B	✓	**U**	-	??
19	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	B	✗	**A**	-	??
20	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	B	✓	**A**	-	??
21	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	B	✗	**U**	-	??
22	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	B	✓	**U**	-	??
23	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	B	✗	**A**	-	??
24	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	B	✓	**A**	-	??

Table 7.: Summary of the tests.

#	Scene	$s \to e$	Deg.	Met.	P. p.	Part.	Config.	figure
25	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	B	✗	**U**	-	??
26	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	B	✓	**U**	-	??
27	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	B	✗	**A**	-	??
28	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	B	✓	**A**	-	??
29	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	B	✗	**U**	-	??
30	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	B	✓	**U**	-	??
31	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	B	✗	**A**	-	??
32	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	B	✓	**A**	-	??
33	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	B	✗	**U**	-	??
34	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	B	✓	**U**	-	??
35	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	B	✗	**A**	-	??
36	1b	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	B	✓	**A**	-	??
37	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	2	B	✗	**U**	-	??
38	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	2	B	✓	**U**	-	??
39	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	2	B	✗	**A**	-	??
40	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	2	B	✓	**A**	-	??
41	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	3	B	✗	**U**	-	??
42	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	3	B	✓	**U**	-	??
43	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	3	B	✗	**A**	-	??
44	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	3	B	✓	**A**	-	??
45	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	4	B	✗	**U**	-	??
46	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	4	B	✓	**U**	-	??
47	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	4	B	✗	**A**	-	??
48	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	4	B	✓	**A**	-	??

Table 8.: Summary of the tests (continue).

#	Scene	$s \to e$	Deg.	Met.	P. p.	Part.	Config.	figure
49	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	A	✗	**U**	-	??
50	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	A	✓	**U**	-	??
51	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	A	✗	**A**	-	??
52	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	A	✓	**A**	-	??
53	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	A	✗	**U**	-	??
54	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	A	✓	**U**	-	??
55	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	A	✗	**A**	-	??
56	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	A	✓	**A**	-	??
57	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	A	✗	**U**	-	??
58	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	A	✓	**U**	-	??
59	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	A	✗	**A**	-	??
60	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	A	✓	**A**	-	??
61	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	B	✗	**U**	-	??
62	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	B	✓	**U**	-	??
63	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	B	✗	**A**	-	??
64	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	B	✓	**A**	-	??
65	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	B	✗	**U**	-	??
66	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	B	✓	**U**	-	??
67	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	B	✗	**A**	-	??
68	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	B	✓	**A**	-	??
69	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	B	✗	**U**	-	??
70	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	B	✓	**U**	-	??
71	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	B	✗	**A**	-	??
72	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	B	✓	**A**	-	??

Table 9.: Summary of the tests (continue).

#	Scene	$\mathbf{s} \to \boldsymbol{e}$	Deg.	Met.	P. p.	Part.	Config.	figure
73	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	C	-	**U**	1	**??**
74	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	C	-	**U**	1	**??**
75	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	C	-	**U**	1	**??**
76	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	C	-	**U**	2	**??**
77	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	C	-	**U**	2	**??**
78	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	C	-	**U**	2	**??**
79	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	2	C	-	**U**	3	**??**
80	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	3	C	-	**U**	3	**??**
81	1	[0.2,0.2,0.2]→[0.9,0.9,0.9]	4	C	-	**U**	3	**??**
82	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	2	C	-	**U**	1	**??**
83	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	3	C	-	**U**	1	**??**
84	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	4	C	-	**U**	1	**??**
85	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	2	C	-	**U**	2	**??**
86	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	3	C	-	**U**	2	**??**
87	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	4	C	-	**U**	2	**??**
88	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	2	C	-	**U**	3	**??**
89	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	3	C	-	**U**	3	**??**
90	2	[0.5,0.5,0.5]→[0.5,0.5,0.95]	4	C	-	**U**	3	**??**
91	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	C	-	**U**	1	**??**
92	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	C	-	**U**	1	**??**
93	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	C	-	**U**	1	**??**
94	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	C	-	**U**	2b	**??**
95	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	C	-	**U**	2b	**??**
96	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	C	-	**U**	2b	**??**
97	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	2	C	-	**U**	3b	**??**
98	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	3	C	-	**U**	3b	**??**
99	3	[0.5,0.5,0.4]→[0.5,0.5,0.2]	4	C	-	**U**	3b	**??**

Table 10.: Summary of the tests (continue).

8

CONCLUSIONS

In this chapter we describe the evidence that emerges from the tests. Furthermore, we discuss possible future improvements of the project.

8.1 TESTS ANALYSIS

We present three scenes to do the tests. Scene 1 consists in 10 obstacles randomly disposed, scene 1b is the same scene with a more dense graph, scene 2 has 100 obstacles and scene 3 has only one bucket-shaped obstacle.

We set the starting and ending points to be at the extremes of the bounding box for scene 1 and scene 1b- i.e. the purpose of the tests in these scenes is to cross the area with the obstacles. For scene 2, we set the starting point in the centre of the crowded area and the aim is to manage to exit from the area. For scene 3, we set the starting point inside the bucket and we want to arrive under it.

Starting with the test for methods A and B, we manage to test all the possible configurations of scenes, degree, method, post-processing and knot partition. For method C, we tested 5 different parameter sets for the Simulated Annealing (SA).

Regarding the performances, the fastest method is B- to have an idea of the temporal scales, consider that, on a quad core Intel i5-2430M CPU at 2.40GHz with 8 Gb of RAM, an execution in scene 1 with post processing and adaptive knot partitions takes:

- 76 seconds for method A;
- 11 seconds for method B.

Method B is faster than method A, but we have to take into consideration that method B is more refined than method A because the latter needs to rebuild G_t when connecting the start and ending points.

It is difficult to compare method C to the previous two because of the different parameter sets, however, an execution of it in scene 1 with configuration 1 takes 168 seconds.

First of all, we notice that the application of the adaptive partition results in a deterioration of the curvature plots - i.e. an increase in magnitude of the curvature peaks - for the experiments with scenes 1, 1b and 2. However the experiments with scene 3 result in an improvement. Thus, this method is not always reliable in terms of the curve fairing.

We notice that the curvature presents peaks near the start and/or the end on some tests. See for instance tests 1, 3, 5. The cause of this is the attachment method of start and ending points. In fact they are attached to the nearest vertex of G, but it can be also too close and in a bad direction, adding a deleterious hook to the control polygon.

The post-processing fulfills the purpose of simplifying the path. Consider, for instance, test 33 (**??**) where the curvature plot has different peaks. After the application of post-processing, we obtain test 34 (**??**) where the curvature peaks are mitigated.

The degree increase algorithm is improving the curvature: the plots are continuous for degree 3 and continuous and smooth for degree 4. Unfortunately, it is not reliable for the torsion (not shown in the tests) because, by adding aligned vertices, we force plane changes on zero-curvature points, where the torsion is not defined.

Method C produces high quality curves (see tests from 74 to 99) with low peaks of curvature and torsion. Moreover, this solution is not conditioned by the problem in degree increase mentioned before, the plots of the torsion are good.

An disadvantage of this solution is the discretization of collision check. Thus, depending on the parameters, it is possible that the path intersects an obstacle without noticing it. Other disadvantages are the slower execution time and the difficulty in finding the right values of the annealing parameters. In fact, wrong values of warming, or an insufficient number of trials can *freeze* the system in a not optimal status. Furthermore, it is necessary to adapt the parameters to different problems. For instance, the configuration 2 and 3 are not fitted for tests from 94 to 99: using those settings is not enough to let the system converge in an admissible state.

Regarding the complexity, we have that the highest cost which is $\mathcal{O}(|O| \log |O|)$ in the number of obstacles $|O|$, comes from the creation of the scene. A run on the scene have complexity

- $\mathcal{O}(|O| \log |O|)$ for method A;

- $\mathcal{O}(|O| \log |O| + |P||O|)$ for method B, where $|P|$ is the number of vertices in the control polygon;
- $\mathcal{O}(|O| \log |O| + \mathit{len}(P)|O|)$ for method C, where $\mathit{len}(P)$ is the length of the control polygon.

8.2 FUTURE IMPROVEMENTS

The present work contemplates many possible improvements. One of them is to provide a better method to attach the start and ending points. For instance one possibility is to connect them to all the visible vertices of G.

Another possible improvement is to design another algorithm for the adaptive knot partition. The current one does not improve enough the fairness of the curve.

Considering the different benefits and drawbacks of the implemented solutions, would be interesting to further elaborate the idea of a mixed approach to the problem: analytical and stochastic.

An new interesting solution might be implementing a stochastic optimization on the path obtained from solution 1 or solution 2, that is obstacle-free guaranteed. This hypothetical stochastic optimization must avoid states that violates the Convex Hull Property (CHP), and it can work directly on the state space without the Lagrangian relaxation. In fact, in that scenario the initial status is already obstacle-free. Furthermore, we believe that the optimization process do not need to explore too much the state space trespassing obstacle zones.

We believe that the described process can be very effective in improving curvature, torsion and length of the path. It can obtain curves with the quality of the implemented third solution and without the disadvantages of it: the slow computation and the possible collision errors caused by the discretization of the inclusion checks.

Another improvement could be studying different basic structures besides the graph extract from Voronoi Diagram (VD). For instance, Rapidly-expanding Random Tree (RRT).

Part IV

APPENDICES

A

TESTS RESULTS

To save space, the tests are not present in this version. Please refer to `https://github.com/trianam/dissertation/raw/master/dissertation.pdf` for the full version.

B

SOURCE CODE

B.1 CLASSES

B.1.1 *voronizator.py*

```
import numpy as np
import numpy.linalg
import scipy as sp
import scipy.spatial
import networkx as nx
import numpy.linalg
import polyhedron
import polyhedronsContainer
import path
import uuid
import xml.etree.cElementTree as ET

class Voronizator:
    _pruningMargin = 0.3

    def __init__(self, sites=np.array([]), bsplineDegree=4, adaptivePartition=False):
        self._shortestPath = path.Path(bsplineDegree, adaptivePartition)
        self._sites = sites
        self._graph = nx.Graph()
        self._tGraph = nx.DiGraph()
        self._startTriplet = None
        self._endTriplet = None
        self._polyhedronsContainer = polyhedronsContainer.PolyhedronsContainer()
        self._pathStart = np.array([])
        self._pathEnd = np.array([])
        self._startId = uuid.uuid4()
        self._endId = uuid.uuid4()
        self._bsplineDegree = bsplineDegree

    def setBsplineDegree(self, bsplineDegree):
        self._bsplineDegree = bsplineDegree
        self._shortestPath.setBsplineDegree(bsplineDegree)

    def setAdaptivePartition(self, adaptivePartition):
        self._shortestPath.setAdaptivePartition(adaptivePartition)

    def setCustomSites(self, sites):
        self._sites = sites

    def setRandomSites(self, number, seed=None):
        if seed != None:
            np.random.seed(0)
        self._sites = sp.rand(number,3)

    def addPolyhedron(self, polyhedron):
        self._polyhedronsContainer.addPolyhedron(polyhedron)

    def addBoundingBox(self, a, b, maxEmptyArea=1, invisible=True, verbose=False):
        if verbose:
            print('Add bounding box', flush=True)

        self._polyhedronsContainer.addBoundingBox(a,b,maxEmptyArea, invisible)
```

```python

    def setPolyhedronsSites(self, verbose=False):
        if verbose:
            print('Set sites for Voronoi', flush=True)

        sites = []
        for polyhedron in self._polyhedronsContainer.polyhedrons:
            sites.extend(polyhedron.allPoints)

        self._sites = np.array(sites)

    def makeVoroGraph(self, prune=True, verbose=False, debug=False):
        if verbose:
            print('Calculate Voronoi cells', flush=True)
        ids = {}
        vor = sp.spatial.Voronoi(self._sites)

        if verbose:
            print('Make pruned Graph from cell edges ', end='', flush=True)
            printDotBunch = 0
        vorVer = vor.vertices
        for ridge in vor.ridge_vertices:
            if verbose:
                if printDotBunch == 0:
                    print('.', end='', flush=True)
                printDotBunch = (printDotBunch+1)%10

            for i in range(1, len(ridge)):
                for j in range(i):
                    if (ridge[i] != -1) and (ridge[j] != -1):
                        a = vorVer[ridge[i]]
                        b = vorVer[ridge[j]]
                        if (not prune) or (not self._polyhedronsContainer.segmentIntersectPolyhedrons(a,b, intersectionMargin = self._pruningMargin)):
                            if tuple(a) in ids.keys():
                                idA = ids[tuple(a)]
                            else:
                                idA = uuid.uuid4()
                                self._graph.add_node(idA, coord=a)
                                ids[tuple(a)] = idA

                            if tuple(b) in ids.keys():
                                idB = ids[tuple(b)]
                            else:
                                idB = uuid.uuid4()
                                self._graph.add_node(idB, coord=b)
                                ids[tuple(b)] = idB

                            self._graph.add_edge(idA, idB, weight=np.linalg.norm(a-b))

        if verbose:
            print('', flush=True)

        self._createTripleGraph(verbose, debug)

    def calculateShortestPath(self, start, end, attachMode='near', prune=True, useMethod='cleanPath', postSimplify=True, verbose=False, debug=False):
        """
        useMethod: cleanPath; trijkstra; annealing; none
        """
        if useMethod == 'trijkstra' or useMethod == 'cleanPath' or useMethod == 'annealing' or useMethod == 'none':
            if verbose:
                print('Attach start and end points', flush=True)
            if attachMode=='near':
                self._attachToGraphNear(start, end, prune)
            elif attachMode=='all':
                self._attachToGraphAll(start, end, prune)
            else:
                self._attachToGraphNear(start, end, prune)

            self._attachSpecialStartEndTriples(verbose)

            self._pathStart = start
            self._pathEnd = end

            if useMethod == 'trijkstra':
                self._removeCollidingTriples(verbose, debug)

            triPath = self._dijkstra(verbose, debug)
            path = self._extractPath(triPath, verbose)
            self._shortestPath.assignValues(path, self._polyhedronsContainer)

```

```
            if useMethod == 'cleanPath':
                self._shortestPath.clean(verbose, debug)
            elif useMethod == 'annealing':
                self._shortestPath.anneal(verbose)

            #print(self._bsplineDegree)
            if useMethod != 'annealing' and useMethod != 'none':
                if self._bsplineDegree == 3:
                    self._shortestPath.addNAlignedVertexes(1, verbose, debug)
                if self._bsplineDegree == 4:
                    self._shortestPath.addNAlignedVertexes(2, verbose, debug)

            if postSimplify:
                self._shortestPath.simplify(verbose, debug)


    def plotSites(self, plotter, verbose=False):
        if verbose:
            print('Plot Sites', end='', flush=True)

        if self._sites.size > 0:
            plotter.addPoints(self._sites, plotter.COLOR_SITES, thick=True)

    def plotPolyhedrons(self, plotter, verbose=False):
        if verbose:
            print('Plot Polyhedrons', end='', flush=True)

        for poly in self._polyhedronsContainer.polyhedrons:
            poly.plot(plotter)
            if verbose:
                print('.', end='', flush=True)

        if verbose:
            print('', flush=True)

    def plotShortestPath(self, plotter, verbose=False):
        if verbose:
            print('Plot shortest path', flush=True)

        if self._shortestPath.vertexes.size > 0:
            if self._polyhedronsContainer.hasBoundingBox:
                splineThickness = np.linalg.norm(np.array(self._polyhedronsContainer.boundingBoxB) - np.array(self.
                    ↪ _polyhedronsContainer.boundingBoxA)) / 1000.
                pointThickness = splineThickness * 2.
                lineThickness = splineThickness / 2.

                plotter.addPolyLine(self._shortestPath.vertexes, plotter.COLOR_CONTROL_POLIG, thick=True, thickness=
                    ↪ lineThickness)
                plotter.addPoints(self._shortestPath.vertexes, plotter.COLOR_CONTROL_POINTS, thick=True, thickness=
                    ↪ pointThickness)
                plotter.addBSpline(self._shortestPath, self._bsplineDegree, plotter.COLOR_PATH, thick=True, thickness=
                    ↪ splineThickness)

            else:
                plotter.addPolyLine(self._shortestPath.vertexes, plotter.COLOR_CONTROL_POLIG, thick=True)
                plotter.addPoints(self._shortestPath.vertexes, plotter.COLOR_CONTROL_POINTS, thick=True)
                plotter.addBSpline(self._shortestPath, self._bsplineDegree, plotter.COLOR_PATH, thick=True)

    def plotGraph(self, plotter, verbose=False):
        if verbose:
            print('Plot graph edges', flush=True)

        plotter.addGraph(self._graph, plotter.COLOR_GRAPH)

    def plotGraphNodes(self, plotter, verbose=False):
        if verbose:
            print('Plot graph nodes', flush=True)

        plotter.addGraphNodes(self._graph, plotter.COLOR_GRAPH)

    def extractXmlTree(self, root):
        if self._polyhedronsContainer.hasBoundingBox:
            xmlBoundingBox = ET.SubElement(root, 'boundingBox')
            ET.SubElement(xmlBoundingBox, 'a', x=str(self._polyhedronsContainer.boundingBoxA[0]), y=str(self.
                ↪ _polyhedronsContainer.boundingBoxA[1]), z=str(self._polyhedronsContainer.boundingBoxA[2]))
            ET.SubElement(xmlBoundingBox, 'b', x=str(self._polyhedronsContainer.boundingBoxB[0]), y=str(self.
                ↪ _polyhedronsContainer.boundingBoxB[1]), z=str(self._polyhedronsContainer.boundingBoxB[2]))

        xmlPolyhedrons = ET.SubElement(root, 'polyhedrons')
        for polyhedron in self._polyhedronsContainer.polyhedrons:
            xmlPolyhedron = polyhedron.extractXmlTree(xmlPolyhedrons)

```

```python
    def importXmlTree(self, root, maxEmptyArea):
        xmlBoundingBox = root.find('boundingBox')
        if xmlBoundingBox:
            xmlA = xmlBoundingBox.find('a')
            xmlB = xmlBoundingBox.find('b')

            self._polyhedronsContainer.hasBoundingBox = True
            self._polyhedronsContainer.boundingBoxA = [float(xmlA.attrib['x']), float(xmlA.attrib['y']), float(xmlA.attrib['z'])]
            self._polyhedronsContainer.boundingBoxB = [float(xmlB.attrib['x']), float(xmlB.attrib['y']), float(xmlB.attrib['z'])]

        xmlPolyhedrons = root.find('polyhedrons')
        if xmlPolyhedrons:
            for xmlPolyhedron in xmlPolyhedrons.iter('polyhedron'):
                invisible = False
                if 'invisible' in xmlPolyhedron.attrib.keys():
                    invisible = bool(eval(xmlPolyhedron.attrib['invisible']))

                boundingBox = False
                if 'boundingBox' in xmlPolyhedron.attrib.keys():
                    boundingBox = bool(eval(xmlPolyhedron.attrib['boundingBox']))

                faces = []
                for xmlFace in xmlPolyhedron.iter('face'):
                    vertexes = []
                    for xmlVertex in xmlFace.iter('vertex'):
                        vertexes.append([float(xmlVertex.attrib['x']), float(xmlVertex.attrib['y']), float(xmlVertex.attrib['z'])])

                    faces.append(vertexes)

                newPolyhedron = polyhedron.Polyhedron(faces=np.array(faces), invisible=invisible, maxEmptyArea=maxEmptyArea, boundingBox=boundingBox)
                self._polyhedronsContainer.addPolyhedron(newPolyhedron)

    def _attachToGraphNear(self, start, end, prune):
        firstS = True
        firstE = True
        minAttachS = None
        minAttachE = None
        minDistS = 0.
        minDistE = 0.
        for node,nodeAttr in self._graph.node.items():
            if (not prune) or (not self._polyhedronsContainer.segmentIntersectPolyhedrons(start,nodeAttr['coord'], intersectionMargin= self._pruningMargin)):
                if firstS:
                    minAttachS = node
                    minDistS = np.linalg.norm(start - nodeAttr['coord'])
                    firstS = False
                else:
                    currDist = np.linalg.norm(start - nodeAttr['coord'])
                    if currDist < minDistS:
                        minAttachS = node
                        minDistS = currDist

            if (not prune) or (not self._polyhedronsContainer.segmentIntersectPolyhedrons(end, nodeAttr['coord'], intersectionMargin= self._pruningMargin)):
                if firstE:
                    minAttachE = node
                    minDistE = np.linalg.norm(end - nodeAttr['coord'])
                    firstE = False
                else:
                    currDist = np.linalg.norm(end - nodeAttr['coord'])
                    if currDist < minDistE:
                        minAttachE = node
                        minDistE = currDist

        if minAttachS != None:
            self._addNodeToTGraph(self._startId, start, minAttachS, minDistS, rightDirection=True)
        if minAttachE != None:
            self._addNodeToTGraph(self._endId, end, minAttachE, minDistE, rightDirection=False)

    def _attachToGraphAll(self, start, end, prune):
        for node,nodeAttr in self._graph.node.items():
            if (not prune) or (not self._polyhedronsContainer.segmentIntersectPolyhedrons(start, nodeAttr['coord'], intersectionMargin= self._pruningMargin)):
                self._addNodeToTGraph(self._startId, start, node, np.linalg.norm(start - nodeAttr['coord']), rightDirection=True)
            if (not prune) or (not self._polyhedronsContainer.segmentIntersectPolyhedrons(end, nodeAttr['coord'], intersectionMargin= self._pruningMargin)):
                self._addNodeToTGraph(self._endId, end, node, np.linalg.norm(end - nodeAttr['coord']), rightDirection=False)
```

```python

    def _addNodeToTGraph(self, newId, coord, attachId, dist, rightDirection):
        self._graph.add_node(newId, coord=coord)
        self._graph.add_edge(newId, attachId, weight=dist)
        for otherId in filter(lambda node: node != newId, self._graph.neighbors(attachId)):
            newTriplet = uuid.uuid4()
            if rightDirection:
                self._tGraph.add_node(newTriplet, triplet=[newId,attachId,otherId])
                self._tGraph.add_edges_from([(newTriplet, otherTriplet, {'weight':dist}) for otherTriplet in self._tGraph.nodes() if self._tGraph.node[otherTriplet]['triplet'][0] == attachId  and self._tGraph.node[otherTriplet]['triplet'][1] == otherId])

            else:
                self._tGraph.add_node(newTriplet, triplet=[otherId,attachId,newId])
                self._tGraph.add_edges_from([(otherTriplet, newTriplet, {'weight':dist}) for otherTriplet in self._tGraph.nodes() if self._tGraph.node[otherTriplet]['triplet'][1] == otherId  and self._tGraph.node[otherTriplet]['triplet'][2] == attachId])

    def _attachSpecialStartEndTriples(self, verbose):
        #attach special starting and ending triplet
        if verbose:
            print('Create starting and ending triplets', flush=True)

        self._startTriplet = uuid.uuid4()
        self._endTriplet = uuid.uuid4()
        self._tGraph.add_node(self._startTriplet, triplet = [self._startId,self._startId,self._startId], hit = False)
        self._tGraph.add_node(self._endTriplet, triplet = [self._endId,self._endId,self._endId], hit = False)
        self._tGraph.add_edges_from([(self._startTriplet, n, {'weight':0.}) for n in self._tGraph.nodes() if self._tGraph.node[n]['triplet'][0] == self._startId])
        self._tGraph.add_edges_from([(n, self._endTriplet, {'weight':0.}) for n in self._tGraph.nodes() if self._tGraph.node[n]['triplet'][2] == self._endId])

    def _createTripleGraph(self, verbose, debug):
        #create triplets

        if debug:
            triplets_file = open("triplets.txt","w")

        if verbose:
            print('Create triplets ', end='', flush=True)
            printDotBunch = 0

        tripletIdList = {}
        def getUniqueId(triplet):
            if tuple(triplet) in tripletIdList.keys():
                tripletId = tripletIdList[tuple(triplet)]
            else:
                tripletId = uuid.uuid4()
                tripletIdList[tuple(triplet)] = tripletId
                self._tGraph.add_node(tripletId, triplet = triplet)
            return tripletId

        for edge in self._graph.edges():
            if verbose:
                if printDotBunch == 0:
                    print('.', end='', flush=True)
                printDotBunch = (printDotBunch+1)%10


            tripletsSxOutgoing = []
            tripletsSxIngoing = []
            tripletsDxOutgoing = []
            tripletsDxIngoing = []

            for nodeSx in filter(lambda node: node != edge[1], self._graph.neighbors(edge[0])):
                tripletId = getUniqueId([nodeSx,edge[0],edge[1]])
                tripletsSxOutgoing.append(tripletId)
                if debug:
                    triplets_file.write('SxO: {}\n'.format(self._tGraph.node[tripletId]['triplet']))

                tripletId = getUniqueId([edge[1],edge[0],nodeSx])
                tripletsSxIngoing.append(tripletId)
                if debug:
                    triplets_file.write('SxI: {}\n'.format(self._tGraph.node[tripletId]['triplet']))

            for nodeDx in filter(lambda node: node != edge[0], self._graph.neighbors(edge[1])):
                tripletId = getUniqueId([nodeDx,edge[1],edge[0]])
                tripletsDxOutgoing.append(tripletId)
                if debug:
                    triplets_file.write('DxO: {}\n'.format(self._tGraph.node[tripletId]['triplet']))

                tripletId = getUniqueId([edge[0],edge[1],nodeDx])
```

```
                tripletsDxIngoing.append(tripletId)
                if debug:
                    triplets_file.write('DxI: {}\n'.format(self._tGraph.node[tripletId]['triplet']))

            for tripletSx in tripletsSxOutgoing:
                for tripletDx in tripletsDxIngoing:
                    self._tGraph.add_edge(tripletSx, tripletDx, {'weight':self._graph.edge[self._tGraph.node[tripletSx]['triplet'][0]][self._tGraph.node[tripletDx]['triplet'][0]]['weight']})

            for tripletDx in tripletsDxOutgoing:
                for tripletSx in tripletsSxIngoing:
                    self._tGraph.add_edge(tripletDx, tripletSx, {'weight':self._graph.edge[self._tGraph.node[tripletDx]['triplet'][0]][self._tGraph.node[tripletSx]['triplet'][0]]['weight']})

        if verbose:
            print('', flush=True)

        if debug:
            triplets_file.close()


    def _dijkstra(self, verbose, debug):
        try:
            if verbose:
                print('Dijkstra algorithm', flush=True)

            length,triPath=nx.bidirectional_dijkstra(self._tGraph, self._startTriplet, self._endTriplet)


        except (nx.NetworkXNoPath, nx.NetworkXError):
            print('ERROR: Impossible to find a path')
            triPath = []

        return triPath

    def _extractPath(self, triPath, verbose):
        if verbose:
            print('Extract path', flush=True)

        path = []
        for t in triPath:
            path.append(self._graph.node[self._tGraph.node[t]['triplet'][1]]['coord'])
        return np.array(path)


    def _removeCollidingTriples(self, verbose, debug):
        if verbose:
            print('Remove colliding triples', flush=True)
            printDotBunch = 0

        toRemove = []
        for triple in self._tGraph:
            if verbose:
                if printDotBunch == 0:
                    print('.', end='', flush=True)
                printDotBunch = (printDotBunch+1)%10

            a = self._graph.node[self._tGraph.node[triple]['triplet'][0]]['coord']
            b = self._graph.node[self._tGraph.node[triple]['triplet'][1]]['coord']
            c = self._graph.node[self._tGraph.node[triple]['triplet'][2]]['coord']
            intersect,intersectRes = self._polyhedronsContainer.triangleIntersectPolyhedrons(a, b, c)
            if intersect:
                toRemove.append(triple)

        if verbose:
            print ("", flush=True)

        for triple in toRemove:
            self._tGraph.remove_node(triple)
```

B.1.2 *path.py*

```
import random
import math
import numpy as np
import scipy as sp
```

```python
import scipy.interpolate

class Path:
    _initialTemperature = 10#1000
    _trials = 10#100
    _warmingRatio = 0.9#0.9
    _minTemperature=0.00001#0.00000001
    _minDeltaEnergy=0.000001
    _maxVlambdaPert = 1000.
    _maxVertexPertFactor = 100.
    _initialVlambda = 0.
    _changeVlambdaProbability = 0.05
    #====1
    #_useArcLen = True
    #_ratioCurvTorsLen = [0.1, 0.1, 0.8]
    #====2
    _useArcLen = False
    _ratioCurvTorsLen = [0.1, 0.1, 0.8]
    #====3
    #_useArcLen = True
    #_ratioCurvTorsLen = [0.3, 0.3, 0.4]

    def __init__(self, bsplineDegree, adaptivePartition):
        self._bsplineDegree = bsplineDegree
        self._adaptivePartition = adaptivePartition
        self._vertexes = np.array([])
        self._dimC = 0
        self._polyhedronsContainer = []
        self._vlambda = self._initialVlambda


    @property
    def vertexes(self):
        return self._vertexes

    def assignValues(self, path, polyhedronsContainer):
        self._vertexes = path
        self._dimC = self._vertexes.shape[1]

        self._polyhedronsContainer = polyhedronsContainer
        tau, u, spline, splineD1, splineD2, splineD3, curv, tors, arcLength, polLength = self._splinePoints(self._vertexes)
        self._maxVertexPert = polLength / self._maxVertexPertFactor


    def setBsplineDegree(self, bsplineDegree):
        self._bsplineDegree = bsplineDegree

    def setAdaptivePartition(self, adaptivePartition):
        self._adaptivePartition = adaptivePartition

    def clean(self, verbose, debug):
        if verbose:
            print('Clean path (avoid obstacles)', flush=True)

        newPath = []
        if len(self._vertexes) > 0:
            a = self._vertexes[0]
            newPath.append(self._vertexes[0])

        for i in range(1, len(self._vertexes)-1):
            v = self._vertexes[i]
            b = self._vertexes[i+1]

            intersect,intersectRes = self._polyhedronsContainer.triangleIntersectPolyhedrons(a, v, b)
            if intersect:
                alpha = intersectRes[1]

                a1 = (1.-alpha)*a + alpha*v
                b1 = alpha*v + (1.-alpha)*b

                newPath.append(a1)
                newPath.append(v)
                newPath.append(b1)

                a = b1
            else:
                newPath.append(v)

                a = v

        if len(self._vertexes) > 0:
            newPath.append(self._vertexes[len(self._vertexes)-1])
```

```
        self._vertexes = np.array(newPath)


    def anneal(self, verbose):
        if verbose:
            print('Anneal path', flush=True)

        tau, u, self._spline, splineD1, splineD2, splineD3, curv, tors, arcLength, polLength = self._splinePoints(self._vertexes)
        self._currentEnergy, self._maxCurvatureLength, self._currentConstraints = self._initializePathEnergy(self._vertexes, self._spline, splineD1, splineD2, self._vlambda)


        temperature = self._initialTemperature
        while True:
            initialEnergy = self._currentEnergy
            numMovedLambda = 0
            numMovedVertex = 0
            for i in range(self._trials):
                movedLambda,movedVertex = self._tryMove(temperature)
                if movedLambda:
                    numMovedLambda += 1
                if movedVertex:
                    numMovedVertex += 1
            deltaEnergy = abs(initialEnergy - self._currentEnergy)
            temperature = temperature * self._warmingRatio
            if verbose:
                print("T:{}; E:{}; DE:{}; L:{}; C:{}; ML:{}; MV:{}".format(temperature, self._currentEnergy, deltaEnergy, self._vlambda, self._currentConstraints, numMovedLambda, numMovedVertex), flush=True)
                #print(self._vertexes)

            if (temperature < self._minTemperature) or (numMovedVertex > 0 and (deltaEnergy < self._minDeltaEnergy) and self._currentConstraints == 0.):
                break


    def simplify(self, verbose, debug):
        if verbose:
            print('Simplify path (remove useless triples)', flush=True)
        if self._bsplineDegree == 2:
            self._simplify2()
        elif self._bsplineDegree == 3:
            self._simplify3()
        elif self._bsplineDegree == 4:
            self._simplify4()

    def _simplify2(self):
        simplifiedPath = []
        if len(self._vertexes) > 0:
            a = self._vertexes[0]
            simplifiedPath.append(self._vertexes[0])
        first = True
        for i in range(1,len(self._vertexes)-1):
            v = self._vertexes[i]
            b = self._vertexes[i+1]
            keepV = False

            intersectCurr,nihil = self._polyhedronsContainer.triangleIntersectPolyhedrons(a, v, b)

            if not intersectCurr:
                if first:
                    intersectPrec = False
                else:
                    #a1 = self._vertexes[i-2]
                    intersectPrec,nihil = self._polyhedronsContainer.triangleIntersectPolyhedrons(a1, a, b)

                if i == len(self._vertexes)-2:
                    intersectSucc = False
                else:
                    b1 = self._vertexes[i+2]
                    intersectSucc,nihil = self._polyhedronsContainer.triangleIntersectPolyhedrons(a, b, b1)

                if intersectPrec or intersectSucc:
                    keepV = True

            else:
                keepV = True

            if keepV:
                first = False
                simplifiedPath.append(v)
```

```
                a1 = a
                a = v

        if len(self._vertexes) > 0:
            simplifiedPath.append(self._vertexes[len(self._vertexes)-1])

        self._vertexes = np.array(simplifiedPath)

    def _simplify3(self):
        simp = list(self._vertexes)
        toEval = 0
        while toEval < len(simp)-2:
            toEval += 1
            if toEval >= 3:
                if toEval < len(simp)-1:
                    if self._polyhedronsContainer.convexHullIntersectsPolyhedrons([simp[toEval-3], simp[toEval-2], simp[toEval-1], simp[toEval+1]]):
                        continue

            if toEval >= 2:
                if toEval < len(simp)-2:
                    if self._polyhedronsContainer.convexHullIntersectsPolyhedrons([simp[toEval-2], simp[toEval-1], simp[toEval+1], simp[toEval+2]]):
                        continue

            if toEval < len(simp)-3:
                if self._polyhedronsContainer.convexHullIntersectsPolyhedrons([simp[toEval-1], simp[toEval+1], simp[toEval+2], simp[toEval+3]]):
                        continue

            del simp[toEval]
            toEval -= 1

        self._vertexes = np.array(simp)

    def _simplify4(self):
        simp = list(self._vertexes)
        toEval = 0
        while toEval < len(simp)-2:
            toEval += 1
            if toEval >= 4:
                if toEval < len(simp)-1:
                    if self._polyhedronsContainer.convexHullIntersectsPolyhedrons([simp[toEval-4], simp[toEval-3], simp[toEval-2], simp[toEval-1], simp[toEval+1]]):
                        continue

            if toEval >= 3:
                if toEval < len(simp)-2:
                    if self._polyhedronsContainer.convexHullIntersectsPolyhedrons([simp[toEval-3], simp[toEval-2], simp[toEval-1], simp[toEval+1], simp[toEval+2]]):
                        continue

            if toEval >= 2:
                if toEval < len(simp)-3:
                    if self._polyhedronsContainer.convexHullIntersectsPolyhedrons([simp[toEval-2], simp[toEval-1], simp[toEval+1], simp[toEval+2], simp[toEval+3]]):
                        continue

            if toEval < len(simp)-4:
                if self._polyhedronsContainer.convexHullIntersectsPolyhedrons([simp[toEval-1], simp[toEval+1], simp[toEval+2], simp[toEval+3], simp[toEval+4]]):
                        continue

            del(simp[toEval])
            toEval -= 1

        self._vertexes = np.array(simp)

    def addNAlignedVertexes(self, numVertexes, verbose, debug):
        if verbose:
            print('Increase degree', flush=True)

        newPath = []
        for i in range(1, len(self._vertexes)):
            a = self._vertexes[i-1]
            b = self._vertexes[i]
            newPath.append(a)

            if numVertexes == 1:
                n = 0.5 * a + 0.5 * b
                newPath.append(n)

```

```
            elif numVertexes == 2:
                n1 = 0.33 * b + 0.67 * a
                n2 = 0.33 * a + 0.67 * b
                newPath.append(n1)
                newPath.append(n2)

        if len(self._vertexes) > 0:
            newPath.append(self._vertexes[len(self._vertexes)-1])

        self._vertexes = np.array(newPath)

    def splinePoints(self):
        return self._splinePoints(self._vertexes)

    def _splinePoints(self, vertexes):

        x = vertexes[:,0]
        y = vertexes[:,1]
        z = vertexes[:,2]

        polLen = self._calculatePolyLength(vertexes)

        tau,t = self._createKnotPartition(vertexes)

        #[knots, coeff, degree]
        tck = [t,[x,y,z], self._bsplineDegree]

        u=np.linspace(0,1,(max(polLen*5,1000)),endpoint=True)

        out = sp.interpolate.splev(u, tck)
        outD1 = sp.interpolate.splev(u, tck, 1)
        outD2 = sp.interpolate.splev(u, tck, 2)

        spline = np.stack(out).T
        splineD1 = np.stack(outD1).T
        splineD2 = np.stack(outD2).T

        if self._bsplineDegree >= 3:
            outD3 = sp.interpolate.splev(u, tck, 3)
            splineD3 = np.stack(outD3).T
        else:
            splineD3 = None

        curv = []
        tors = []
        arcLength = 0.
        for i in range(len(u)):
            d1Xd2 = np.cross(splineD1[i], splineD2[i])
            Nd1Xd2 = np.linalg.norm(d1Xd2)
            Nd1 = np.linalg.norm(splineD1[i])

            currCurv = 0.
            if Nd1 >0.: #>= 1.:
                currCurv = Nd1Xd2 / math.pow(Nd1,3)

            currTors = 0.
            if self._bsplineDegree >= 3 and Nd1Xd2 > 0.: #>= 1.:
                try:
                    currTors = np.dot(d1Xd2, splineD3[i]) / math.pow(Nd1Xd2, 2)
                except RuntimeWarning:
                    currTors = 0.

            curv.append(currCurv)
            tors.append(currTors)

            if i >= 1:
                dMin = min(prevNd1, Nd1)
                dMax = max(prevNd1, Nd1)
                arcLength += (u[i]-u[i-1]) * (dMin + ((dMax-dMin) / 2.))

            prevNd1 = Nd1

        return (tau, u, spline, splineD1, splineD2, splineD3, curv, tors, arcLength, polLen)

    def _createKnotPartition(self, controlPolygon):
        nv = len(controlPolygon)
        nn = nv - self._bsplineDegree + 1

        if not self._adaptivePartition:
            T = np.linspace(0,1,nv-self._bsplineDegree+1,endpoint=True)
        else:
            d = [0]
```

```
            for j in range(1, nv):
                d.append(d[j-1] + np.linalg.norm(controlPolygon[j] - controlPolygon[j-1]))
            t = []
            for i in range(nn-1):
                a = i * (nv-1) / (nn-1)
                ai = math.floor(a)
                ad = a - ai
                p = ad * controlPolygon[ai+1] + (1-ad) * controlPolygon[ai]
                l = d[ai] + np.linalg.norm(p - controlPolygon[ai])
                t.append(l / d[nv-1])

            t.append(1.)

            T = np.array(t)

        Text = np.append([0]*self._bsplineDegree, T)
        Text = np.append(Text, [1]*self._bsplineDegree)

        return (T,Text)

    def _initializePathEnergy(self, vertexes, spline, splineD1, splineD2, vlambda):
        tau, u, spline, splineD1, splineD2, splineD3, curv, tors, arcLength, polLength = self._splinePoints(vertexes)
        if self._useArcLen:
            length = arcLength
        else:
            length = polLength

        self._initialLength = length
        maxCurvatureLength = self._calculateMaxCurvatureLength(length, curv, tors)
        constraints = self._calculateConstraints(spline)
        energy = maxCurvatureLength + vlambda * constraints

        return (energy, maxCurvatureLength, constraints)

    def _tryMove(self, temperature):
        """
        Move the path or lambda multipiers in a neighbouring state,
        with a certain acceptance probability.
        Pick a random vertex (except extremes), and move
        it in a random direction (with a maximum perturbance).
        Use a lagrangian relaxation because we need to evaluate
        min(measure(path)) given the constraint that all quadrilaters
        formed by 4 consecutive points in the path must be collision
        free; where measure(path) is, depending of the choose method,
        the length of the path or the mean
        of the supplementary angles of each pair of edges of the path.
        If neighbourMode=0 then move the node uniformly, if
        neighbourMode=1 then move the node with gaussian probabilities
        with mean in the perpendicular direction respect to the
        previous-next nodes axis.
        """

        movedLambda = False
        movedVertex = False
        moveVlambda = random.random() < self._changeVlambdaProbability
        if moveVlambda:
            newVlambda = self._vlambda
            newVlambda = newVlambda + (random.uniform(-1.,1.) * self._maxVlambdaPert)

            newEnergy = self._calculatePathEnergyLambda(newVlambda)

            #attention, different formula from below
            if (newEnergy > self._currentEnergy) or (math.exp(-(self._currentEnergy-newEnergy)/temperature) >= random.random()):
                self._vlambda = newVlambda
                self._currentEnergy = newEnergy
                movedLambda = True

        else:
            newVertexes = np.copy(self._vertexes)
            movedV = random.randint(1,len(self._vertexes) - 2) #don't change extremes

            moveC = random.randint(0,self._dimC - 1)
            newVertexes[movedV][moveC] = newVertexes[movedV][moveC] + (random.uniform(-1.,1.) * self._maxVertexPert)

            newEnergy,newMaxCurvatureLength,newConstraints = self._calculatePathEnergyVertex(newVertexes)

            #attention, different formula from above
            if (newEnergy < self._currentEnergy) or (math.exp(-(newEnergy-self._currentEnergy)/temperature) >= random.random()):
                self._vertexes = newVertexes
                self._currentEnergy = newEnergy
```

```
                self._currentMaxCurvatureLength = newMaxCurvatureLength
                self._currentConstraints = newConstraints
                movedVertex = True

        return (movedLambda, movedVertex)

    def _calculatePathEnergyLambda(self, vlambda):
        """
        calculate the energy when lambda is moved.
        """
        return (self._currentEnergy - (self._vlambda * self._currentConstraints) + (vlambda * self._currentConstraints))

    def _calculatePathEnergyVertex(self, vertexes):
        """
        calculate the energy when a vertex is moved and returns it.
        """
        tau, u, spline, splineD1, splineD2, splineD3, curv, tors, arcLength, polLength = self._splinePoints(vertexes)
        if self._useArcLen:
            length = arcLength
        else:
            length = polLength

        constraints = self._calculateConstraints(spline)#this is bottleneck
        maxCurvatureLength = self._calculateMaxCurvatureLength(length, curv, tors)

        energy = maxCurvatureLength + self._vlambda * constraints

        return (energy, maxCurvatureLength, constraints)

    def _calculatePolyLength(self, vertexes):
        length = 0.
        for i in range(1, len(vertexes)):
            length += sp.spatial.distance.euclidean(vertexes[i-1], vertexes[i])
            #length += np.linalg.norm(np.subtract(vertexes[i], vertexes[i-1]))
        return length

    def _calculateMaxCurvatureLength(self, length, curv, tors):
        normLength = length/self._initialLength * 100 #for making the ratio indipendent of the initial length

        maxCurvature = 0.
        maxTorsion = 0.
        for i in range(0, len(curv)):
            currCurv = curv[i]
            currTors = abs(tors[i])
            if currCurv > maxCurvature:
                maxCurvature = currCurv
            if currTors > maxTorsion:
                maxTorsion = currTors

        return self._ratioCurvTorsLen[0]*maxCurvature + self._ratioCurvTorsLen[1]*maxTorsion + self._ratioCurvTorsLen[2]*
             ↪ normLength

    def _calculateConstraints(self, spline):
        """
        calculate the constraints function. Is the ratio of the points
        of the calculated spline that are inside obstacles respect the
        total number of points of the spline.
        """
        pointsInside = 0
        for p in spline:
            if self._polyhedronsContainer.pointInsidePolyhedron(p):
                pointsInside = pointsInside + 1

        constraints = pointsInside / len(spline)

        return constraints
```

B.1.3 *plotter.py*

```
import numpy as np
import scipy as sp
import scipy.interpolate
import scipy.spatial
import pickle
import vtk
import vtk.util.colors
```

```python
import math
import warnings
warnings.filterwarnings("error")

class Plotter:
    COLOR_BG = vtk.util.colors.light_grey
    COLOR_BG_PLOT = vtk.util.colors.ghost_white
    #vtk.util.colors.ivory
    COLOR_OBSTACLE = vtk.util.colors.banana
    COLOR_SITES = vtk.util.colors.cobalt
    COLOR_PATH = vtk.util.colors.brick
    COLOR_CONTROL_POINTS = vtk.util.colors.tomato
    COLOR_CONTROL_POLIG = vtk.util.colors.mint
    COLOR_GRAPH = vtk.util.colors.sepia
    COLOR_PLOT_CURV = vtk.util.colors.blue
    COLOR_PLOT_TORS = vtk.util.colors.red
    COLOR_LABELS = vtk.util.colors.blue
    COLOR_LENGTH = vtk.util.colors.red

    _DEFAULT_LINE_THICKNESS = 0.0005
    _DEFAULT_POINT_THICKNESS = 0.002
    _DEFAULT_BSPLINE_THICKNESS = 0.001

    class KeyPressInteractorStyle(vtk.vtkInteractorStyleUnicam):
        _screenshotFile = "/tmp/screenshot.png"
        _cameraFile = "/tmp/cameraData.dat"
        _cameraFile2 = "/tmp/cameraData2.dat"
        def __init__(self, parent=None):
            self.AddObserver("KeyPressEvent", self._keyPressEvent)
            self.AddObserver("RightButtonPressEvent", self._mousePressEvent)
            #super(KeyPressInteractorStyle, self).__init__()

        def SetCamera(self, camera):
            self._camera = camera

        def SetRenderer(self, renderer):
            self._renderer = renderer

        def SetRenderWindow(self, renderWindow):
            self._renderWindow = renderWindow

        def _keyPressEvent(self, obj, event):
            if obj.GetInteractor().GetKeySym() == "l":
                print("Scene screenshot in "+self._screenshotFile)
                w2if = vtk.vtkWindowToImageFilter()
                w2if.SetInput(self._renderWindow)
                w2if.Update()

                writer = vtk.vtkPNGWriter()
                writer.SetFileName(self._screenshotFile)
                writer.SetInputData(w2if.GetOutput())
                writer.Write()

            elif obj.GetInteractor().GetKeySym() == "c":
                print("Save camera data in "+self._cameraFile)
                record = {}
                record['position'] = self._camera.GetPosition()
                record['focalPoint'] = self._camera.GetFocalPoint()
                record['viewAngle'] = self._camera.GetViewAngle()
                record['viewUp'] = self._camera.GetViewUp()
                record['clippingRange'] = self._camera.GetClippingRange()

                with open(self._cameraFile, 'wb') as f:
                    pickle.dump(record, f)

            elif obj.GetInteractor().GetKeySym() == "v":
                print("Restore camera data from "+self._cameraFile)

                with open(self._cameraFile, 'rb') as f:
                    record = pickle.load(f)

                    self._camera.SetPosition(record['position'])
                    self._camera.SetFocalPoint(record['focalPoint'])
                    self._camera.SetViewAngle(record['viewAngle'])
                    self._camera.SetViewUp(record['viewUp'])
                    self._camera.SetClippingRange(record['clippingRange'])

                    self._renderWindow.Render()

            elif obj.GetInteractor().GetKeySym() == "b":
                print("Restore camera data from "+self._cameraFile2)

```

```
                with open(self._cameraFile2, 'rb') as f:
                    record = pickle.load(f)

                    self._camera.SetPosition(record['position'])
                    self._camera.SetFocalPoint(record['focalPoint'])
                    self._camera.SetViewAngle(record['viewAngle'])
                    self._camera.SetViewUp(record['viewUp'])
                    self._camera.SetClippingRange(record['clippingRange'])

                    self._renderWindow.Render()


            self.OnKeyPress()

        def _mousePressEvent(self, obj, event):
            clickPos = obj.GetInteractor().GetEventPosition()
            picker =vtk.vtkPropPicker()
            picker.Pick(clickPos[0], clickPos[1], 0, self._renderer)
            pos = picker.GetPickPosition()
            print(pos)


    class KeyPressContextInteractorStyle(vtk.vtkContextInteractorStyle):
        _screenshotFile = "/tmp/screenshot.png"
        def __init__(self, parent=None):
            self.AddObserver("KeyPressEvent",self._keyPressEvent)

        def SetRenderWindow(self, renderWindow):
            self._renderWindow = renderWindow

        def _keyPressEvent(self, obj, event):
            if obj.GetInteractor().GetKeySym() == "l":
                print("Plot screenshot in "+self._screenshotFile)
                w2if = vtk.vtkWindowToImageFilter()
                w2if.SetInput(self._renderWindow)
                w2if.Update()

                writer = vtk.vtkPNGWriter()
                writer.SetFileName(self._screenshotFile)
                writer.SetInputData(w2if.GetOutput())
                writer.Write()



    def __init__(self):
        self._rendererScene = vtk.vtkRenderer()
        self._rendererScene.SetBackground(self.COLOR_BG)

        self._renderWindowScene = vtk.vtkRenderWindow()
        self._renderWindowScene.AddRenderer(self._rendererScene)
        self._renderWindowInteractor = vtk.vtkRenderWindowInteractor()
        self._renderWindowInteractor.SetRenderWindow(self._renderWindowScene)
        #self._interactorStyle = vtk.vtkInteractorStyleUnicam()
        self._interactorStyle = self.KeyPressInteractorStyle()
        self._interactorStyle.SetCamera(self._rendererScene.GetActiveCamera())
        self._interactorStyle.SetRenderer(self._rendererScene)
        self._interactorStyle.SetRenderWindow(self._renderWindowScene)

        self._contextViewPlotCurv = vtk.vtkContextView()
        self._contextViewPlotCurv.GetRenderer().SetBackground(self.COLOR_BG_PLOT)

        self._contextInteractorStyleCurv = self.KeyPressContextInteractorStyle()
        self._contextInteractorStyleCurv.SetRenderWindow(self._contextViewPlotCurv.GetRenderWindow())

        self._chartXYCurv = vtk.vtkChartXY()
        self._contextViewPlotCurv.GetScene().AddItem(self._chartXYCurv)
        self._chartXYCurv.SetShowLegend(True)
        self._chartXYCurv.GetAxis(vtk.vtkAxis.LEFT).SetTitle("")
        self._chartXYCurv.GetAxis(vtk.vtkAxis.BOTTOM).SetTitle("")

        self._contextViewPlotTors = vtk.vtkContextView()
        self._contextViewPlotTors.GetRenderer().SetBackground(self.COLOR_BG_PLOT)

        self._contextInteractorStyleTors = self.KeyPressContextInteractorStyle()
        self._contextInteractorStyleTors.SetRenderWindow(self._contextViewPlotTors.GetRenderWindow())

        self._chartXYTors = vtk.vtkChartXY()
        self._contextViewPlotTors.GetScene().AddItem(self._chartXYTors)
        self._chartXYTors.SetShowLegend(True)
        self._chartXYTors.GetAxis(vtk.vtkAxis.LEFT).SetTitle("")
        self._chartXYTors.GetAxis(vtk.vtkAxis.BOTTOM).SetTitle("")

```

```
        self._textActor = vtk.vtkTextActor()
        self._textActor.GetTextProperty().SetColor(self.COLOR_LENGTH)

        self._addedBSpline = False

    def draw(self):
        self._renderWindowInteractor.Initialize()
        self._renderWindowInteractor.SetInteractorStyle(self._interactorStyle)

        axes = vtk.vtkAxesActor()
        widget = vtk.vtkOrientationMarkerWidget()
        widget.SetOutlineColor(0.9300, 0.5700, 0.1300)
        widget.SetOrientationMarker(axes)
        widget.SetInteractor(self._renderWindowInteractor)
        #widget.SetViewport(0.0, 0.0, 0.1, 0.1)
        widget.SetViewport(0.0, 0.0, 0.2, 0.4)
        widget.SetEnabled(True)
        widget.InteractiveOn()

        textWidget = vtk.vtkTextWidget()

        textRepresentation = vtk.vtkTextRepresentation()
        textRepresentation.GetPositionCoordinate().SetValue(.0,.0 )
        textRepresentation.GetPosition2Coordinate().SetValue(.3,.04 )
        textWidget.SetRepresentation(textRepresentation)

        textWidget.SetInteractor(self._renderWindowInteractor)
        textWidget.SetTextActor(self._textActor)
        textWidget.SelectableOff()
        textWidget.On()

        self._rendererScene.ResetCamera()
        camPos = self._rendererScene.GetActiveCamera().GetPosition()
        self._rendererScene.GetActiveCamera().SetPosition((camPos[2],camPos[1],camPos[0]))
        self._rendererScene.GetActiveCamera().SetViewUp((0.0,0.0,1.0))
        self._rendererScene.GetActiveCamera().Zoom(1.4)

        self._renderWindowScene.Render()

        if self._addedBSpline:
            self._contextViewPlotCurv.GetRenderWindow().SetMultiSamples(0)
            self._contextViewPlotCurv.GetInteractor().Initialize()
            self._contextViewPlotCurv.GetInteractor().SetInteractorStyle(self._contextInteractorStyleCurv)
            #self._contextViewPlotCurv.GetInteractor().Start()

            self._contextViewPlotTors.GetRenderWindow().SetMultiSamples(0)
            self._contextViewPlotTors.GetInteractor().Initialize()
            self._contextViewPlotTors.GetInteractor().SetInteractorStyle(self._contextInteractorStyleTors)
            self._contextViewPlotTors.GetInteractor().Start()
        else:
            self._renderWindowInteractor.Start()


    def addTetrahedron(self, vertexes, color):
        vtkPoints = vtk.vtkPoints()
        vtkPoints.InsertNextPoint(vertexes[0][0], vertexes[0][1], vertexes[0][2])
        vtkPoints.InsertNextPoint(vertexes[1][0], vertexes[1][1], vertexes[1][2])
        vtkPoints.InsertNextPoint(vertexes[2][0], vertexes[2][1], vertexes[2][2])
        vtkPoints.InsertNextPoint(vertexes[3][0], vertexes[3][1], vertexes[3][2])

        unstructuredGrid = vtk.vtkUnstructuredGrid()
        unstructuredGrid.SetPoints(vtkPoints)
        unstructuredGrid.InsertNextCell(vtk.VTK_TETRA, 4, range(4))

        mapper = vtk.vtkDataSetMapper()
        mapper.SetInputData(unstructuredGrid)

        actor = vtk.vtkActor()
        actor.SetMapper(mapper)
        actor.GetProperty().SetColor(color)

        self._rendererScene.AddActor(actor)

    def addTriangles(self, triangles, color):
        vtkPoints = vtk.vtkPoints()
        idPoint = 0
        allIdsTriangle = []

        for triangle in triangles:
            idsTriangle = []

            for point in triangle:
```

```
                vtkPoints.InsertNextPoint(point[0], point[1], point[2])
                idsTriangle.append(idPoint)
                idPoint += 1

            allIdsTriangle.append(idsTriangle)

        unstructuredGrid = vtk.vtkUnstructuredGrid()
        unstructuredGrid.SetPoints(vtkPoints)
        for idsTriangle in allIdsTriangle:
            unstructuredGrid.InsertNextCell(vtk.VTK_TRIANGLE, 3, idsTriangle)

        mapper = vtk.vtkDataSetMapper()
        mapper.SetInputData(unstructuredGrid)

        actor = vtk.vtkActor()
        actor.SetMapper(mapper)
        actor.GetProperty().SetColor(color)

        self._rendererScene.AddActor(actor)

    def addPolyLine(self, points, color, thick=False, thickness=_DEFAULT_LINE_THICKNESS):
        vtkPoints = vtk.vtkPoints()
        for point in points:
            vtkPoints.InsertNextPoint(point[0], point[1], point[2])

        if thick:
            cellArray = vtk.vtkCellArray()
            cellArray.InsertNextCell(len(points))
            for i in range(len(points)):
                cellArray.InsertCellPoint(i)

            polyData = vtk.vtkPolyData()
            polyData.SetPoints(vtkPoints)
            polyData.SetLines(cellArray)

            tubeFilter = vtk.vtkTubeFilter()
            tubeFilter.SetNumberOfSides(8)
            tubeFilter.SetInputData(polyData)
            tubeFilter.SetRadius(thickness)
            tubeFilter.Update()

            mapper = vtk.vtkPolyDataMapper()
            mapper.SetInputConnection(tubeFilter.GetOutputPort())

        else:
            unstructuredGrid = vtk.vtkUnstructuredGrid()
            unstructuredGrid.SetPoints(vtkPoints)
            for i in range(1, len(points)):
                unstructuredGrid.InsertNextCell(vtk.VTK_LINE, 2, [i-1, i])

            mapper = vtk.vtkDataSetMapper()
            mapper.SetInputData(unstructuredGrid)

        actor = vtk.vtkActor()
        actor.SetMapper(mapper)
        actor.GetProperty().SetColor(color)

        self._rendererScene.AddActor(actor)

    def addPoints(self, points, color, thick=False, thickness=_DEFAULT_POINT_THICKNESS):
        vtkPoints = vtk.vtkPoints()
        for point in points:
            vtkPoints.InsertNextPoint(point[0], point[1], point[2])

        pointsPolyData = vtk.vtkPolyData()
        pointsPolyData.SetPoints(vtkPoints)

        if thick:
            sphereSource = vtk.vtkSphereSource()
            sphereSource.SetRadius(thickness)

            glyph3D = vtk.vtkGlyph3D()
            glyph3D.SetSourceConnection(sphereSource.GetOutputPort())
            glyph3D.SetInputData(pointsPolyData)
            glyph3D.Update()

            mapper = vtk.vtkPolyDataMapper()
            mapper.SetInputConnection(glyph3D.GetOutputPort())
        else:
            vertexFilter = vtk.vtkVertexGlyphFilter()
            vertexFilter.SetInputData(pointsPolyData)
            vertexFilter.Update()
```

```

            mapper = vtk.vtkPolyDataMapper()
            mapper.SetInputData(vertexFilter.GetOutput())

        actor = vtk.vtkActor()
        actor.SetMapper(mapper)
        actor.GetProperty().SetColor(color)

        self._rendererScene.AddActor(actor)

    def addBSpline(self, path, degree, color, thick=False, thickness=_DEFAULT_BSPLINE_THICKNESS):
        self._addedBSpline = True

        tau, u, spline, splineD1, splineD2, splineD3, curv, tors, arcLength, polLength = path.splinePoints()

        self._textActor.SetInput("Length: "+str(arcLength))

        numIntervals = len(tau)-1

        curvPlotActor = vtk.vtkXYPlotActor()
        curvPlotActor.SetTitle("Curvature")
        curvPlotActor.SetXTitle("")
        curvPlotActor.SetYTitle("")
        curvPlotActor.SetXValuesToIndex()

        torsPlotActor = vtk.vtkXYPlotActor()
        torsPlotActor.SetTitle("Torsion")
        torsPlotActor.SetXTitle("")
        torsPlotActor.SetYTitle("")
        torsPlotActor.SetXValuesToIndex()

        uArrays = []
        curvArrays = []
        torsArrays = []
        for i in range(numIntervals):
            uArrays.append(vtk.vtkDoubleArray())
            uArrays[i].SetName("t")

            curvArrays.append(vtk.vtkDoubleArray())
            curvArrays[i].SetName("Curvature")

            torsArrays.append(vtk.vtkDoubleArray())
            torsArrays[i].SetName("Torsion")

        curvTorsArray = vtk.vtkDoubleArray()

        #minCurv = minTors = minNd1Xd2 = float("inf")
        #maxCurv = maxTors = float("-inf")

        for i in range(len(u)):
            for j in range(numIntervals):
                if u[i] >= tau[j] and u[i] < tau[j+1]:
                    break

            uArrays[j].InsertNextValue(u[i])
            curvArrays[j].InsertNextValue(curv[i])
            torsArrays[j].InsertNextValue(tors[i])

            curvTorsArray.InsertNextValue(curv[i])# + abs(tors[i]))

        #print("minCurv: {:e}; maxCurv: {:e}; minTors: {:e}; maxTors: {:e}; minNd1Xd2: {:e}".format(minCurv, maxCurv, minTors, maxTors, minNd1Xd2))

        for inter in range(numIntervals):
            plotTable = vtk.vtkTable()
            plotTable.AddColumn(uArrays[inter])
            plotTable.AddColumn(curvArrays[inter])
            plotTable.AddColumn(torsArrays[inter])

            points = self._chartXYCurv.AddPlot(vtk.vtkChart.LINE)
            points.SetInputData(plotTable, 0, 1)
            points.SetColor(self.COLOR_PLOT_CURV[0], self.COLOR_PLOT_CURV[1], self.COLOR_PLOT_CURV[2])
            points.SetWidth(1.0)
            if inter > 0:
                points.SetLegendVisibility(False)

            points = self._chartXYTors.AddPlot(vtk.vtkChart.LINE)
            points.SetInputData(plotTable, 0, 2)
            points.SetColor(self.COLOR_PLOT_TORS[0], self.COLOR_PLOT_TORS[1], self.COLOR_PLOT_TORS[2])
            points.SetWidth(1.0)
            if inter > 0:
                points.SetLegendVisibility(False)
```

```
        vtkPoints = vtk.vtkPoints()
        for point in spline:
            vtkPoints.InsertNextPoint(point[0], point[1], point[2])

        polyDataLabelP = vtk.vtkPolyData()
        polyDataLabelP.SetPoints(vtkPoints)

        labels = vtk.vtkStringArray()
        labels.SetNumberOfValues(len(spline))
        labels.SetName("labels")
        for i in range(len(spline)):
            if i == 0:
                labels.SetValue(i, "S")
            elif i == len(spline)-1:
                labels.SetValue(i, "E")
            else:
                labels.SetValue(i, "")

        polyDataLabelP.GetPointData().AddArray(labels)

        sizes = vtk.vtkIntArray()
        sizes.SetNumberOfValues(len(spline))
        sizes.SetName("sizes")
        for i in range(len(spline)):
            if i == 0 or i == len(spline)-1:
                sizes.SetValue(i, 10)
            else:
                sizes.SetValue(i,1)

        polyDataLabelP.GetPointData().AddArray(sizes)

        pointMapper = vtk.vtkPolyDataMapper()
        pointMapper.SetInputData(polyDataLabelP)

        pointActor = vtk.vtkActor()
        pointActor.SetMapper(pointMapper)

        pointSetToLabelHierarchyFilter = vtk.vtkPointSetToLabelHierarchy()
        pointSetToLabelHierarchyFilter.SetInputData(polyDataLabelP)
        pointSetToLabelHierarchyFilter.SetLabelArrayName("labels")
        pointSetToLabelHierarchyFilter.SetPriorityArrayName("sizes")
        pointSetToLabelHierarchyFilter.GetTextProperty().SetColor(self.COLOR_LABELS)
        pointSetToLabelHierarchyFilter.GetTextProperty().SetFontSize(15)
        pointSetToLabelHierarchyFilter.GetTextProperty().SetBold(True)
        pointSetToLabelHierarchyFilter.Update()

        labelMapper = vtk.vtkLabelPlacementMapper()
        labelMapper.SetInputConnection(pointSetToLabelHierarchyFilter.GetOutputPort())
        labelActor = vtk.vtkActor2D()
        labelActor.SetMapper(labelMapper)

        self._rendererScene.AddActor(labelActor)

        if thick:
            cellArray = vtk.vtkCellArray()
            cellArray.InsertNextCell(len(spline))
            for i in range(len(spline)):
                cellArray.InsertCellPoint(i)

            polyData = vtk.vtkPolyData()
            polyData.SetPoints(vtkPoints)
            polyData.SetLines(cellArray)

            polyData.GetPointData().SetScalars(curvTorsArray)

            tubeFilter = vtk.vtkTubeFilter()
            tubeFilter.SetNumberOfSides(8)
            tubeFilter.SetInputData(polyData)
            tubeFilter.SetRadius(thickness)
            tubeFilter.Update()

            mapper = vtk.vtkPolyDataMapper()
            mapper.SetInputConnection(tubeFilter.GetOutputPort())

        else:
            unstructuredGrid = vtk.vtkUnstructuredGrid()
            unstructuredGrid.SetPoints(vtkPoints)
            for i in range(1, len(spline)):
                unstructuredGrid.InsertNextCell(vtk.VTK_LINE, 2, [i-1, i])
```

```
            unstructuredGrid.GetPointData().SetScalars(curvArray)

            mapper = vtk.vtkDataSetMapper()
            mapper.SetInputData(unstructuredGrid)

        actor = vtk.vtkActor()
        actor.SetMapper(mapper)
        actor.GetProperty().SetColor(color)

        self._rendererScene.AddActor(actor)


        #self.addPolyLine(list(zip(out[0], out[1], out[2])), color, thick, thickness)

    def addBSplineDEPRECATED(self, controlPolygon, degree, color, thick=False, thickness=_DEFAULT_BSPLINE_THICKNESS):
        x = controlPolygon[:,0]
        y = controlPolygon[:,1]
        z = controlPolygon[:,2]

        polLen = 0.
        for i in range(1, len(controlPolygon)):
            polLen += sp.spatial.distance.euclidean(controlPolygon[i-1], controlPolygon[i])

        t = range(len(controlPolygon))
        ipl_t = np.linspace(0.0, len(controlPolygon) - 1, max(polLen*100,100))

        x_tup = sp.interpolate.splrep(t, x, k = degree)
        y_tup = sp.interpolate.splrep(t, y, k = degree)
        z_tup = sp.interpolate.splrep(t, z, k = degree)

        x_list = list(x_tup)
        xl = x.tolist()
        x_list[1] = xl + [0.0, 0.0, 0.0, 0.0]

        y_list = list(y_tup)
        yl = y.tolist()
        y_list[1] = yl + [0.0, 0.0, 0.0, 0.0]

        z_list = list(z_tup)
        zl = z.tolist()
        z_list[1] = zl + [0.0, 0.0, 0.0, 0.0]

        x_i = sp.interpolate.splev(ipl_t, x_list)
        y_i = sp.interpolate.splev(ipl_t, y_list)
        z_i = sp.interpolate.splev(ipl_t, z_list)

        self.addPolyLine(list(zip(x_i, y_i, z_i)), color, thick, thickness)

    def addGraph(self, graph, color):
        vtkPoints = vtk.vtkPoints()
        vtkId = 0
        graph2VtkId = {}

        for node in graph.nodes():
            vtkPoints.InsertNextPoint(graph.node[node]['coord'][0], graph.node[node]['coord'][1], graph.node[node]['coord'][2])
            graph2VtkId[node] = vtkId
            vtkId += 1

        unstructuredGrid = vtk.vtkUnstructuredGrid()
        unstructuredGrid.SetPoints(vtkPoints)

        for edge in graph.edges():
            unstructuredGrid.InsertNextCell(vtk.VTK_LINE, 2, [graph2VtkId[edge[0]], graph2VtkId[edge[1]]])

        mapper = vtk.vtkDataSetMapper()
        mapper.SetInputData(unstructuredGrid)

        actor = vtk.vtkActor()
        actor.SetMapper(mapper)
        actor.GetProperty().SetColor(color)

        self._rendererScene.AddActor(actor)

    def addGraphNodes(self, graph, color):
        nodes = []

        for node in graph.nodes():
            nodes.append((graph.node[node]['coord'][0], graph.node[node]['coord'][1], graph.node[node]['coord'][2]))

        self.addPoints(nodes, color, thick=True)
```

B.1.4 *polyhedronsContainer.py*

```
import numpy as np
import scipy as sp
import scipy.spatial
import polyhedron
import parallelepiped

class PolyhedronsContainer:
    def __init__(self):
        self._polyhedrons = []
        self._hasBoundingBox = False
        self._boundingBoxA = None
        self._boundingBoxB = None

    @property
    def polyhedrons(self):
        return self._polyhedrons

    @property
    def hasBoundingBox(self):
        return self._hasBoundingBox

    @hasBoundingBox.setter
    def hasBoundingBox(self, value):
        self._hasBoundingBox = value

    @property
    def boundingBoxA(self):
        return self._boundingBoxA

    @boundingBoxA.setter
    def boundingBoxA(self, value):
        self._boundingBoxA = value

    @property
    def boundingBoxB(self):
        return self._boundingBoxB

    @boundingBoxB.setter
    def boundingBoxB(self, value):
        self._boundingBoxB = value

    def addPolyhedron(self, polyhedron):
        self._polyhedrons.append(polyhedron)

    def addBoundingBox(self, a, b, maxEmptyArea, invisible):
        self._hasBoundingBox = True
        self._boundingBoxA = a
        self._boundingBoxB = b

        self.addPolyhedron(parallelepiped.Parallelepiped(a=a, b=b, invisible=invisible, maxEmptyArea=maxEmptyArea,
              ↪ boundingBox=True))

    def pointInsidePolyhedron(self, p):
        inside = False
        if self._hasBoundingBox:
            if (p<self._boundingBoxA).any() or (p>self._boundingBoxB).any():
                inside = True

        if not inside:
            for polyhedron in self._polyhedrons:
                if (not polyhedron.isBoundingBox()) and polyhedron.hasPointInside(p):
                    inside = True
                    break

        return inside


    def segmentIntersectPolyhedrons(self, a, b, intersectionMargin = 0.):
        intersect = False
        if self._hasBoundingBox:
            if((a<self._boundingBoxA).any() or (a>self._boundingBoxB).any() or (b<self._boundingBoxA).any() or (b>self.
                  ↪ _boundingBoxB).any()):
                intersect = True

        if not intersect:
            minS = np.array([min(a[0],b[0]),min(a[1],b[1]),min(a[2],b[2])])
            maxS = np.array([max(a[0],b[0]),max(a[1],b[1]),max(a[2],b[2])])

```

```python
            for polyhedron in self._polyhedrons:
                if polyhedron.intersectSegment(a,b,minS,maxS, intersectionMargin=intersectionMargin)[0]:
                    intersect = True
                    break

        return intersect

    def triangleIntersectPolyhedrons(self, a, b, c):
        triangle = polyhedron.Polyhedron(faces=np.array([[a,b,c]]), distributePoints = False)
        intersect = False
        result = np.array([])
        for currPolyhedron in self._polyhedrons:
            currIntersect,currResult = currPolyhedron.intersectPathTriple(triangle)
            if currIntersect and (not intersect or (currResult[1] > result[1])):
                intersect = True
                result = currResult

        return (intersect, result)

    def convexHullIntersectsPolyhedrons(self, vertexes):
        convHull = sp.spatial.ConvexHull(vertexes, qhull_options="QJ Pp")
        for simplex in convHull.simplices:
            if self.triangleIntersectPolyhedrons(convHull.points[simplex[0]], convHull.points[simplex[1]], convHull.points[simplex[2]])[0]:
                return True

        return False
```

B.1.5 *polyhedron.py*

```python
import numpy as np
import scipy as sp
import scipy.spatial
import math
import xml.etree.cElementTree as ET

class Polyhedron:
    def __init__(self, faces, invisible=False, distributePoints=True, maxEmptyArea=0.1, boundingBox=False):
        """
        can be composed only by combined triangles
        faces -> an np.array of triangular faces
        if invisible=True when plot will be called it will be useless
        """
        self._faces = faces
        self._invisible = invisible
        self._boundingBox = boundingBox

        self._minV = np.array([float('inf'),float('inf'),float('inf')])
        self._maxV = np.array([float('-inf'),float('-inf'),float('-inf')])
        for face in self._faces:
            for vertex in face:
                for i in range(len(vertex)):
                    if vertex[i] < self._minV[i]:
                        self._minV[i] = vertex[i]

                for i in range(len(vertex)):
                    if vertex[i] > self._maxV[i]:
                        self._maxV[i] = vertex[i]

        if distributePoints:
            self.distributePoints(maxEmptyArea)
        else:
            self._allPoints = np.array([])

    @property
    def allPoints(self):
        return self._allPoints

    @property
    def minV(self):
        return self._minV

    @property
    def maxV(self):
        return self._maxV

```

```python
    def isBoundingBox(self):
        return self._boundingBox

    def _area(self, triangle):
        a = np.linalg.norm(triangle[1]-triangle[0])
        b = np.linalg.norm(triangle[2]-triangle[1])
        c = np.linalg.norm(triangle[0]-triangle[2])
        s =  (a+b+c) / 2.
        return math.sqrt(s * (s-a) * (s-b) *(s-c))

    _comb2 = lambda self,a,b: 0.5*a + 0.5*b
    #_comb3 = lambda self,a,b,c: 0.33*a + 0.33*b + 0.33*c

    def distributePoints(self, maxEmptyArea):
        allPoints = []
        triangles = []

        for face in self._faces:
            triangles.append(face)

        while triangles:
            triangle = triangles.pop(0)
            a = triangle[0]
            b = triangle[1]
            c = triangle[2]
            if not any((a == x).all() for x in allPoints):
                allPoints.append(a)
            if not any((b == x).all() for x in allPoints):
                allPoints.append(b)
            if not any((c == x).all() for x in allPoints):
                allPoints.append(c)
            if (self._area(triangle) > maxEmptyArea):
                ab = self._comb2(a,b)
                bc = self._comb2(b,c)
                ca = self._comb2(c,a)
                #abc = self._comb3(a,b,c)

                triangles.append(np.array([a,ab,ca]))
                triangles.append(np.array([ab,b,bc]))
                triangles.append(np.array([bc,c,ca]))
                triangles.append(np.array([ab,bc,ca]))
                #triangles.append(np.array([a,ab,abc]))
                #triangles.append(np.array([ab,b,abc]))
                #triangles.append(np.array([b,bc,abc]))
                #triangles.append(np.array([bc,c,abc]))
                #triangles.append(np.array([c,ca,abc]))
                #triangles.append(np.array([ca,a,abc]))

        self._allPoints = np.array(allPoints)

    def hasPointInside(self, p):
        """
        check if a point is inside the convex hull of obstacle vertexes
        """
        outside = True
        if (p>self._minV).all() and (p<self._maxV).all():
            vertexes = [p]
            for triangle in self._faces:
                vertexes.append(triangle[0])
                vertexes.append(triangle[1])
                vertexes.append(triangle[2])

            chull = sp.spatial.ConvexHull(np.array(vertexes))
            outside = False
            for vertex in chull.vertices:
                if (p == chull.points[vertex]).all():
                    outside = True
                    break
            # for simplex in chull.simplices:
            #     if (p == chull.points[simplex[0]]).all() or (p == chull.points[simplex[1]]).all() or (p == chull.points[
                ↪ simplex[2]]).all():
            #         outside = True
            #         break

        return not outside

    def intersectSegment(self, a, b, minS=None, maxS=None, intersectionMargin=0.):
        if minS is None or maxS is None:
            minS = np.array([min(a[0],b[0]),min(a[1],b[1]),min(a[2],b[2])])
            maxS = np.array([max(a[0],b[0]),max(a[1],b[1]),max(a[2],b[2])])

        if not ((self._minV > maxS).any() or (self._maxV < minS).any()):
```

```
            for triangle in self._faces:
                #solve {
                #        a+k(b-a) = v*triangle[0] + w*triangle[1] + s*triangle[2]
                #        v+w+s = 1
                #      }
                # for variables k, v, w, s

                #simplified in
                #      a+k(b-a) = (1-w-s)*triangle[0] + w*triangle[1] + s*triangle[2]
                # for variables k, w, s

                diffba = b-a
                difft0t1 = triangle[0] - triangle[1]
                difft0t2 = triangle[0] - triangle[2]
                difft0a = triangle[0] - a

                A = np.array([
                    [diffba[0], difft0t1[0], difft0t2[0]],
                    [diffba[1], difft0t1[1], difft0t2[1]],
                    [diffba[2], difft0t1[2], difft0t2[2]]])
                B = np.array([difft0a[0], difft0a[1], difft0a[2]])

                try:
                    x = np.linalg.solve(A,B)
                    # check (with margins) if
                    #          0 < k < 1,
                    #          w > 0
                    #          s > 0
                    #          w+s < 1
                    if (x[0] >= 0. - intersectionMargin) and (x[0] <= 1. + intersectionMargin) and (x[1] >= 0. - intersectionMargin) and (x[2] >= 0. - intersectionMargin) and (x[1]+x[2] <= 1. + intersectionMargin):
                        return (True, x)
                except np.linalg.linalg.LinAlgError:
                    pass

        return (False,np.array([]))

    def intersectPolyhedron(self, polyhedron):
        """alert, not case of one polyhedron inside other"""
        if not ((self._minV > polyhedron.maxV).any() or (self._maxV < polyhedron.minV).any()):
            for otherFace in polyhedron._faces:
                for myFace in self._faces:
                    if (
                            self.intersectSegment(otherFace[0],otherFace[1])[0] or
                            self.intersectSegment(otherFace[1],otherFace[2])[0] or
                            self.intersectSegment(otherFace[2],otherFace[0])[0] or
                            polyhedron.intersectSegment(myFace[0], myFace[1])[0] or
                            polyhedron.intersectSegment(myFace[1], myFace[2])[0] or
                            polyhedron.intersectSegment(myFace[2], myFace[0])[0]):
                        return True
        return False

    def intersectPathTriple(self, triple):
        """alert, not case of one polyhedron inside other, and only
        check if the segments of self intersect the triple."""
        intersect = False
        result = np.array([])

        if not ((self._minV > triple.maxV).any() or (self._maxV < triple.minV).any()):
            for myFace in self._faces:
                intersect1, result1 = triple.intersectSegment(myFace[0], myFace[1])
                intersect2, result2 = triple.intersectSegment(myFace[1], myFace[2])
                intersect3, result3 = triple.intersectSegment(myFace[2], myFace[0])
                if intersect1:
                    intersect = True
                    result = result1
                if intersect2 and (not intersect or (result2[1] > result[1])):
                    intersect = True
                    result = result2
                if intersect3 and (not intersect or (result3[1] > result[1])):
                    intersect = True
                    result = result3

        return intersect,result


    def plotAllPoints(self, plotter):
        if self._allPoints.size > 0:
            plotter.addPoints(self._allPoints, plotter.COLOR_SITES)

    def plot(self, plotter):
```

```
        if self._invisible == False:
            plotter.addTriangles(self._faces, plotter.COLOR_OBSTACLE)

    def extractXmlTree(self, root):
        xmlPolyhedron = ET.SubElement(root, 'polyhedron', invisible=str(self._invisible), boundingBox=str(self._boundingBox))
        for face in self._faces:
            xmlFace = ET.SubElement(xmlPolyhedron, 'face')
            for vertex in face:
                xmlVertex = ET.SubElement(xmlFace, 'vertex', x=str(vertex[0]), y=str(vertex[1]), z=str(vertex[2]))
```

B.1.6 *compositePolyhedron.py*

```
import numpy as np
import polyhedron

class CompositePolyhedron(polyhedron.Polyhedron):
    def __init__(self, components):
        self._components = components

        self._boundingBox = False

        self._minV = np.array([float('inf'),float('inf'),float('inf')])
        self._maxV = np.array([float('-inf'),float('-inf'),float('-inf')])

        for component in self._components:
            for i in range(3):
                if component.minV[i] < self._minV[i]:
                    self._minV[i] = component.minV[i]

                if component.maxV[i] > self._maxV[i]:
                    self._maxV[i] = component.maxV[i]

    @property
    def allPoints(self):
        allPoints = []
        for component in self._components:
            allPoints.extend(list(component.allPoints))

        return np.array(allPoints)

    @property
    def minV(self):
        return self._minV

    @property
    def maxV(self):
        return self._maxV


    def distributePoints(self, maxEmptyArea):
        for component in self._components:
            component.distributePoints(maxEmptyArea)

    def hasPointInside(self, p):
        hasPI = False
        for component in self._components:
            if component.hasPointInside(p):
                hasPI = True
                break

        return hasPI

    def intersectSegment(self, a, b, minS=None, maxS=None, intersectionMargin=0.):
        intersect = (False,np.array([]))
        for component in self._components:
            current = component.intersectSegment(a,b,minS,maxS,intersectionMargin)
            if current[0]:
                intersect = current
                break

        return intersect

    def intersectPolyhedron(self, polyhedron):
        intersect = False
        for component in self._components:
            if component.intersectPolyhedron(polyhedron):
```

```
                intersect = True
                break

        return intersect

    def intersectPathTriple(self, triple):
        intersect = (False, np.array([]))
        for component in self._components:
            current = component.intersectPathTriple(triple)
            if current[0]:
                intersect = current
                break

        return intersect

    def plotAllPoints(self, plotter):
        for component in self._components:
            component.plotAllPoints(plotter)

    def plot(self, plotter):
        for component in self._components:
            component.plot(plotter)

    def extractXmlTree(self, root):
        for component in self._components:
            component.extractXmlTree(root)
```

B.1.7 *tetrahedron.py*

```
import numpy as np
import polyhedron

class Tetrahedron(polyhedron.Polyhedron):
    def __init__(self, a, b, c, d, invisible=False, distributePoints=True, maxEmptyArea=0.1):
        super(Tetrahedron, self).__init__(np.array([[a,b,c],[a,b,d],[b,c,d],[c,a,d]]), invisible, distributePoints,
             ↪ maxEmptyArea)

        self._vertexes = [a,b,c,d]

    def plot(self, plotter):
        if self._invisible == False:
            plotter.addTetrahedron(self._vertexes, plotter.COLOR_OBSTACLE)
```

B.1.8 *parallelepiped.py*

```
import numpy as np
import polyhedron

class Parallelepiped(polyhedron.Polyhedron):
    def __init__(self, a, b, invisible=False, distributePoints=True, maxEmptyArea=0.1, boundingBox=False):

        c = [a[0], b[1], a[2]]
        d = [b[0], a[1], a[2]]
        e = [a[0], a[1], b[2]]
        f = [b[0], b[1], a[2]]
        g = [b[0], a[1], b[2]]
        h = [a[0], b[1], b[2]]

        super(Parallelepiped, self).__init__(faces=np.array([
            [a,g,e],[a,d,g],[d,f,g],[f,b,g],[f,b,h],[f,h,c],
            [h,a,e],[h,c,a],[e,h,g],[h,b,g],[a,d,f],[a,f,c]
            ]), invisible=invisible, distributePoints=distributePoints, maxEmptyArea=maxEmptyArea, boundingBox=boundingBox)
```

B.1.9 *convexHull.py*

```python
import numpy as np
import scipy as sp
import scipy.spatial
import polyhedron

class ConvexHull(polyhedron.Polyhedron):
    def __init__(self, points, invisible=False, distributePoints=True, maxEmptyArea=0.1):
        convHull = sp.spatial.ConvexHull(points)
        faces = []
        for simplex in convHull.simplices:
            faces.append([convHull.points[simplex[0]], convHull.points[simplex[1]], convHull.points[simplex[2]]])

        super(ConvexHull, self).__init__(np.array(faces), invisible, distributePoints, maxEmptyArea)
```

B.1.10 *bucket.py*

```python
import numpy as np
import compositePolyhedron
import parallelepiped

class Bucket(compositePolyhedron.CompositePolyhedron):
    def __init__(self, center, width, height, thickness, invisible=False, distributePoints=True, maxEmptyArea=0.1,
        ↪ boundingBox=False):
        c = center
        l = width
        h = height
        d = thickness
        parallelepipeds = []

        parallelepipeds.append(parallelepiped.Parallelepiped(
            np.array([c[0]-(l/2), c[1]-(l/2), c[2]-(h/2)]),\
            np.array([c[0]+(l/2), c[1]+(l/2), c[2]-(h/2)+d]), invisible, distributePoints, maxEmptyArea, boundingBox))

        parallelepipeds.append(parallelepiped.Parallelepiped(
            np.array([c[0]-(l/2), c[1]+(l/2)-d, c[2]-(h/2)+d]),\
            np.array([c[0]+(l/2), c[1]+(l/2), c[2]+(h/2)]), invisible, distributePoints, maxEmptyArea, boundingBox))

        parallelepipeds.append(parallelepiped.Parallelepiped(
            np.array([c[0]-(l/2), c[1]-(l/2), c[2]-(h/2)+d]),\
            np.array([c[0]+(l/2), c[1]-(l/2)+d, c[2]+(h/2)]), invisible, distributePoints, maxEmptyArea, boundingBox))

        parallelepipeds.append(parallelepiped.Parallelepiped(
            np.array([c[0]-(l/2), c[1]-(l/2)+d, c[2]-(h/2)+d]),\
            np.array([c[0]-(l/2)+d, c[1]+(l/2)-d, c[2]+(h/2)]), invisible, distributePoints, maxEmptyArea, boundingBox))

        parallelepipeds.append(parallelepiped.Parallelepiped(
            np.array([c[0]+(l/2)-d, c[1]-(l/2)+d, c[2]-(h/2)+d]),\
            np.array([c[0]+(l/2), c[1]+(l/2)-d, c[2]+(h/2)]), invisible, distributePoints, maxEmptyArea, boundingBox))

        super(Bucket, self).__init__(parallelepipeds)
```

B.2 SCRIPTS

B.2.1 *makeRandomScene.py*

```python
#!/bin/python

import sys
import numpy as np
import random
import math
import pickle
import voronizator
import tetrahedron

if len(sys.argv) >= 14 and len(sys.argv) <= 15:
    i = 1
```

```python
    minX = float(sys.argv[i])
    i += 1
    minY = float(sys.argv[i])
    i += 1
    minZ = float(sys.argv[i])
    i += 1
    maxX = float(sys.argv[i])
    i += 1
    maxY = float(sys.argv[i])
    i += 1
    maxZ = float(sys.argv[i])
    i += 1
    bbMargin = float(sys.argv[i])
    i += 1
    fixedRadius = bool(eval(sys.argv[i]))
    i += 1
    if fixedRadius:
        radius = float(sys.argv[i])
        i += 1
    else:
        minRadius = float(sys.argv[i])
        i += 1
        maxRadius = float(sys.argv[i])
        i += 1
    avoidCollisions = bool(eval(sys.argv[i]))
    i += 1
    numObstacles = int(sys.argv[i])
    i += 1
    maxEmptyArea = float(sys.argv[i])
    i += 1
    fileName = sys.argv[i]

else:
    minX = float(input('Insert min scene X: '))
    minY = float(input('Insert min scene Y: '))
    minZ = float(input('Insert min scene Z: '))
    maxX = float(input('Insert max scene X: '))
    maxY = float(input('Insert max scene Y: '))
    maxZ = float(input('Insert max scene Z: '))
    bbMargin = float(input('Insert bounding box margin: '))
    fixedRadius = bool(eval(input('Do you want fixed obstacle radius? (True/False): ')))
    if fixedRadius:
        radius = float(input('Insert obstacle radius: '))
    else:
        minRadius = float(input('Insert min obstacle radius: '))
        maxRadius = float(input('Insert max obstacle radius: '))
    avoidCollisions = bool(eval(input('Do you want to avoid collisions between obstacles? (True/False): ')))
    numObstacles = int(input('Insert obstacles number: '))
    maxEmptyArea = float(input('Insert max empty area (for points distribution in obstacles): '))

    fileName = input('Insert file name: ')

voronoi = voronizator.Voronizator()
obstacles = []

for ob in range(numObstacles):
    print('Creating obstacle {} '.format(ob+1), end='', flush=True)
    ok = False
    while not ok:
        print('.', end='', flush=True)
        if not fixedRadius:
            radius = random.uniform(minRadius,maxRadius)
        center = np.array([random.uniform(minX+radius,maxX-radius), random.uniform(minY+radius,maxY-radius), random.uniform(minZ+radius,maxZ-radius)])
        points = []
        for pt in range(4):
            elev = random.uniform(-math.pi/2., math.pi/2.)
            azim = random.uniform(0., 2.*math.pi)
            points[:0] = [center+np.array([
                radius*math.cos(elev)*math.cos(azim),
                radius*math.cos(elev)*math.sin(azim),
                radius*math.sin(elev)])]

        newObstacle = tetrahedron.Tetrahedron(a = points[0], b = points[1], c = points[2], d = points[3], distributePoints = True, maxEmptyArea = maxEmptyArea)

        ok = True
        if avoidCollisions:
            for obstacle in obstacles:
                if newObstacle.intersectPolyhedron(obstacle):
                    ok = False
                    break
```

```python

        if ok:
            voronoi.addPolyhedron(newObstacle)
            if avoidCollisions:
                obstacles[:0] = [newObstacle]

    print(' done', flush=True)

voronoi.addBoundingBox([minX-bbMargin, minY-bbMargin, minZ-bbMargin], [maxX+bbMargin, maxY+bbMargin, maxZ+bbMargin], maxEmptyArea, verbose=True)

voronoi.setPolyhedronsSites(verbose=True)
voronoi.makeVoroGraph(verbose=True)

print('Write file', flush=True)
record = {}
record['voronoi'] = voronoi

with open(fileName, 'wb') as f:
    pickle.dump(record, f)
```

B.2.2 *makeBucketScene.py*

```python
#!/bin/python

import sys
import numpy as np
import random
import math
import pickle
import voronizator
import bucket

if len(sys.argv) == 9:
    i = 1
    minPoint = np.array(tuple(eval(sys.argv[i])),dtype=float)
    i += 1
    maxPoint = np.array(tuple(eval(sys.argv[i])),dtype=float)
    i += 1
    center = np.array(tuple(eval(sys.argv[i])),dtype=float)
    i += 1
    width = float(sys.argv[i])
    i += 1
    height = float(sys.argv[i])
    i += 1
    thickness = float(sys.argv[i])
    i += 1
    maxEmptyArea = float(sys.argv[i])
    i += 1
    fileName = sys.argv[i]

else:
    minPoint = np.array(tuple(eval(input('Insert min point (x,y,z): '))),dtype=float)
    maxPoint = np.array(tuple(eval(input('Insert max point (x,y,z): '))),dtype=float)
    center = np.array(tuple(eval(input('Insert bucket center point (x,y,z): '))),dtype=float)
    width = float(input('Insert bucket width: '))
    height = float(input('Insert bucket height: '))
    thickness = float(input('Insert bucket thickness: '))
    maxEmptyArea = float(input('Insert max empty area (for points distribution in obstacles): '))
    fileName = input('Insert file name: ')

voronoi = voronizator.Voronizator()

print('Create bucket', flush=True)
voronoi.addPolyhedron(bucket.Bucket(center, width, height, thickness, distributePoints=True, maxEmptyArea=maxEmptyArea))
voronoi.addBoundingBox(minPoint, maxPoint, maxEmptyArea, verbose=True)
voronoi.setPolyhedronsSites(verbose=True)
voronoi.makeVoroGraph(verbose=True)

print('Write file', flush=True)
record = {}
record['voronoi'] = voronoi

with open(fileName, 'wb') as f:
    pickle.dump(record, f)
```

B.2.3 *plotScene.py*

```python
#!/bin/python

import sys
import pickle
import plotter

if len(sys.argv) >= 2:
    if len(sys.argv) == 5:
        i = 2
        plotSites = bool(eval(sys.argv[i]))
        i += 1
        plotGraph = bool(eval(sys.argv[i]))
        i += 1
        plotGraphNodes = bool(eval(sys.argv[i]))
    else:
        plotSites = bool(eval(input('Do you want to plot Voronoi sites? (True/False): ')))
        plotGraph = bool(eval(input('Do you want to plot graph edges? (True/False): ')))
        plotGraphNodes = bool(eval(input('Do you want to plot graph nodes? (True/False): ')))


    print('Load file', flush=True)
    with open(sys.argv[1], 'rb') as f:
        record = pickle.load(f)

    voronoi = record['voronoi']

    print('Build renderer, window and interactor', flush=True)
    plt = plotter.Plotter()

    voronoi.plotPolyhedrons(plt, verbose = True)
    if plotSites:
        voronoi.plotSites(plt, verbose = True)
    if plotGraph:
        voronoi.plotGraph(plt, verbose = True)
    if plotGraphNodes:
        voronoi.plotGraphNodes(plt, verbose = True)

    print('Render', flush=True)
    plt.draw()

else:
    print('use: {} sceneFile [plotSites plotGraph]'.format(sys.argv[0]))
```

B.2.4 *executeInScene.py*

```python
#!/bin/python

import sys
import numpy as np
import pickle
import plotter

if len(sys.argv) >= 2:
    if len(sys.argv) == 8:
        i = 2
        startPoint = np.array(tuple(eval(sys.argv[i])),dtype=float)
        i += 1
        endPoint = np.array(tuple(eval(sys.argv[i])),dtype=float)
        i += 1
        bsplineDegree = int(sys.argv[i])
        i += 1
        useMethod = str(sys.argv[i])
        i += 1
        postSimplify = bool(eval(sys.argv[i]))
        i += 1
        adaptivePartition = bool(eval(sys.argv[i]))
    else:
        startPoint = np.array(tuple(eval(input('Insert start point (x,y,z): '))),dtype=float)
        endPoint = np.array(tuple(eval(input('Insert end point (x,y,z): '))),dtype=float)
        bsplineDegree = int(input('Insert B-spline degree (2/3/4): '))
        useMethod = str(input('Wich method you want to use? (none/trijkstra/cleanPath/annealing): '))
        postSimplify = bool(eval(input('Do you want post processing? (True/False): ')))
```

```
        adaptivePartition = bool(eval(input('Do you want adaptive partition? (True/False): ')))

    print('Load file', flush=True)
    with open(sys.argv[1], 'rb') as f:
        record = pickle.load(f)

    voronoi = record['voronoi']
    voronoi.setBsplineDegree(bsplineDegree)
    voronoi.setAdaptivePartition(adaptivePartition)

    voronoi.calculateShortestPath(startPoint, endPoint, 'near', useMethod=useMethod, postSimplify=postSimplify, verbose=True,
        ↪ debug=False)

    print('Build renderer, window and interactor', flush=True)
    plt = plotter.Plotter()

    #voronoi.plotSites(plt, verbose = True)
    voronoi.plotPolyhedrons(plt, verbose = True)
    #voronoi.plotGraph(plt, verbose = True)
    voronoi.plotShortestPath(plt, verbose = True)

    print('Render', flush=True)
    plt.draw()

else:
    print('use: {} sceneFile [startPoint endPoint degree(2,4) useMethod postProcessing adaptivePartition]'.format(sys.argv
        ↪ [0]))
```

B.2.5 *scene2coord.py*

```
#!/bin/python

import sys
import pickle
import xml.etree.cElementTree as ET

if len(sys.argv) == 3:

    print('Load file', flush=True)
    with open(sys.argv[1], 'rb') as f:
        record = pickle.load(f)

    voronoi = record['voronoi']

    print('Create XML', flush=True)
    xmlRoot = ET.Element('scene')
    voronoi.extractXmlTree(xmlRoot)
    xmlTree = ET.ElementTree(xmlRoot)

    print('Write file', flush=True)
    xmlTree.write(sys.argv[2])

else:
    print('use: {} sceneFile coordinateFile'.format(sys.argv[0]))
```

B.2.6 *coord2scene.py*

```
#!/bin/python

import sys
import pickle
import xml.etree.cElementTree as ET
import voronizator

if len(sys.argv) == 4:
    xmlFileName = sys.argv[1]
    sceneFileName = sys.argv[2]
    maxEmptyArea = float(sys.argv[3])

    xmlRoot = ET.parse(xmlFileName).getroot()
```

```

    voronoi = voronizator.Voronizator()

    print('Import XML', flush=True)
    voronoi.importXmlTree(xmlRoot, maxEmptyArea)

    print('Set sites and make graph', flush=True)
    voronoi.setPolyhedronsSites(verbose=True)
    voronoi.makeVoroGraph(verbose=True)

    print('Write file', flush=True)
    record = {}
    record['voronoi'] = voronoi
    with open(sceneFileName, 'wb') as f:
        pickle.dump(record, f)

else:
    print('use: {} coordinateFile sceneFile maxEmptyArea'.format(sys.argv[0]))
```

BIBLIOGRAPHY

[1] Adrian Bondy, U. M. (2008). *Graph theory*. Graduate texts in mathematics 244. Springer, 3rd corrected printing. edition. (Cited on pages 55 and 60.)

[2] Aghababa, M. P. (2012). 3d path planning for underwater vehicles using five evolutionary optimization algorithms avoiding static and energetic obstacles. *Applied Ocean Research*, 38:48 – 62. (Cited on page 1.)

[3] Barkema, M. and Newman, G. (1999). *Monte Carlo Methods in Statistical Physics*. Oxford Uinversity Press. (Cited on page 26.)

[4] Bertsekas, D. P. (1999). *Nonlinear programming*. Athena Scientific, 2nd edition. (Cited on page 32.)

[5] Bhattacharya, P. and Gavrilova, M. L. (2008). Roadmap-based path planning - using the voronoi diagram for a clearance-based shortest path. *IEEE Robot. Automat. Mag.*, 15(2):58–66. (Cited on page 46.)

[6] Canny, J. F. (1988). *Complexity of Robot Motion Planning*. ACM Doctoral Dissertation Award. The MIT Press. (Cited on page 10.)

[7] Choset H., e. a. (2005). *Principles of Robot Motion: Theory, Algorithms, and Implementations*. Intelligent Robotics and Autonomous Agents series. MIT. (Cited on pages 7, 9, and 11.)

[8] de Berg, M., Cheong, O., van Kreveld, M., and Overmars, M. (2008). *Computational Geometry Algorithms and Applications*. Springer-Verlag, Berlin Heidelberg, 3 edition. (Cited on pages 9, 23, 24, 33, 43, and 48.)

[9] de Boor, C. (1978). *A Practical Guide to Splines*. Springer-Verlag. (Cited on pages 15, 17, 18, and 21.)

[10] Dijkstra, E. (1959). A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271. (Cited on pages 9, 48, and 50.)

[11] do Carmo, M. P. (1976). *Differential geometry of curves and surfaces.* Prentice Hall. (Cited on pages 22 and 63.)

[12] Farin, G. E. (1990). *Curves and surfaces for computer aided geometric design - a practical guide (2.ed.).* Computer science and scientific computing. Academic Press. (Cited on pages 1, 15, 17, 19, 21, and 63.)

[13] Farouki, R. T. (2008). *Pythagorean-Hodograph Curves: Algebra and Geometry Inseparable*. Geometry and Computing 1. Springer-Verlag Berlin Heidelberg, 1 edition. (Cited on page 1.)

[14] Fortune, S. (1986). A sweepline algorithm for voronoi diagrams. In Aggarwal, A., editor, *Symposium on Computational Geometry*, pages 313–322. ACM. (Cited on pages 24 and 46.)

[15] G. Farin, J. Hoschek, M.-S. K. (2002). *Handbook of Computer Aided Geometric Design*. North Holland, 1 edition. (Cited on page 1.)

[16] Giannelli, C., Mugnaini, D., and Sestini, A. (2016). Path planning with obstacle avoidance by G^1 PH quintic splines. *Computer-Aided Design*, 75–76:47 – 60. (Cited on page 1.)

[17] Goerzen, C., Kong, Z., and Mettler, B. (2009). A survey of motion planning algorithms from the perspective of autonomous uav guidance. *Journal of Intelligent and Robotic Systems*, 57(1):65–100. (Cited on pages 7, 8, and 9.)

[18] Ho, Y.-J. and Liu, J.-S. (2009). Collision-free curvature-bounded smooth path planning using composite bezier curve based on voronoi diagram. In *CIRA*, pages 463–468. IEEE. (Cited on pages 1 and 46.)

[19] Ho, Y.-J. and Liu, J.-S. (2010). Simulated annealing based algorithm for smooth robot path planning with different kinematic constraints. In Shin, S. Y., Ossowski, S., Schumacher, M., Palakal, M. J., and Hung, C.-C., editors, *SAC*, pages 1277–1281. ACM. (Cited on page 26.)

[20] Hrabar, S. (2008). 3d path planning and stereo-based obstacle avoidance for rotorcraft uavs. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 807–814. IEEE. (Cited on page 1.)

[21] Hughes J.F., e. a. (2013). *Computer Graphics: Principles and Practice*. 3rd Edition. Addison-Wesley Professional, 3 edition. (Cited on pages 1 and 41.)

[22] James D. Foley, e. a. (1995). *Computer Graphics. Principles and Practice in C*. Addison-Wesley, 2 edition. (Cited on pages 1 and 41.)

[23] Kirkpatrick, S., Gelatt, C. D. J., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*. (Cited on page 26.)

[24] Knuth, D. E. (1977). A generalization of dijkstra's algorithm. *Information Processing Letters*, 6(1):1 – 5. (Cited on pages 48 and 50.)

[25] Kroumov, V. and Yu, J. (2009). 3d path planning for mobile robots using annealing neural network. In *Networking, Sensing and Control, 2009. ICNSC '09. International Conference on*, pages 130–135. (Cited on page 1.)

[26] LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press. (Cited on pages 7, 8, 12, 55, and 60.)

[27] Li, Z., Meek, D., and Walton, D. (2006). A smooth, obstacle-avoiding curve. *Computers & Graphics*, 30(4):581 – 587. (Cited on page 1.)

[28] Maekawa, T., Noda, T., Tamura, S., Ozaki, T., and ichiro Machida, K. (2010). Curvature continuous path generation for autonomous vehicle using b-spline curves. *Computer-Aided Design*, 42(4):350–359. (Cited on page 1.)

[29] Metropolis, N. and Ulam, S. (1949). The monte carlo method. *J. Am. Stat. Assoc.*, 44:335. (Cited on page 26.)

[30] Okabe A., e. a. (2000). *Spatial tessellations: Concepts and applications of Voronoi diagrams*. Wiley Series in Probability and Statistics. Wiley, 2ed edition. (Cited on page 49.)

[31] Paden, B., Cáp, M., Yong, S. Z., Yershov, D. S., and Frazzoli, E. (2016). A survey of motion planning and control techniques for self-driving urban vehicles. *CoRR*, abs/1604.07446. (Cited on page 7.)

[32] Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. (1992). *Numerical Recipes*. Cambridge University Press. (Cited on pages 23 and 43.)

[33] Pukelsheim, F. (1994). The three sigma rule. *The American Statistician*, 48(2):88–91. (Cited on page 28.)

[34] Richard H. Bartels, John C. Beatty, B. A. B. (1995). *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling (The Morgan Kaufmann Series in Computer Graphics)*. The Morgan Kaufmann Series in Computer Graphics. Morgan Kaufmann, 1 edition. (Cited on pages 15 and 17.)

[35] Salomon, D. (2006). *Curves and Surfaces for Computer Graphics*. Springer. (Cited on pages 15, 17, and 22.)

[36] Schneider, P. J. and Eberly, D. (2002). *Geometric Tools for Computer Graphics*. Elsevier Science Inc., New York, NY, USA. (Cited on pages 33, 34, and 37.)

[37] Sobol', I. M. (1994). *A primer for the Monte Carlo method*. CRC Press. (Cited on page 26.)

[38] Stoer, J. and Bulirsch, R. (1992). *Introduction to numerical analysis*. Springer-Verlag, New York. (Cited on page 23.)

[39] Šeda, M. and Pich, V. (2008). Robot motion planning using generalised voronoi diagrams. In *Proceedings of the 8th Conference on Signal Processing, Computational Geometry and Artificial Vision*, ISCGAV'08, pages 215–220, Stevens Point, Wisconsin, USA. World Scientific and Engineering Academy and Society (WSEAS). (Cited on page 46.)

[40] Wah, B. W. and Wang, T. (1999). Constrained simulated annealing with applications in nonlinear continuous constrained global optimization. In *ICTAI*, pages 381–. IEEE Computer Society. (Cited on page 26.)

[41] Yang, K. and Sukkarieh, S. (2008). 3d smooth path planning for a uav in cluttered natural environments. In *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 794–800. IEEE. (Cited on page 1.)

ACRONYMS

CAD Computer-Aided Design

CAGD Computer-Aided Geometric Design

CAM Computer-Aided Manufacturing

CHP Convex Hull Property

LR Lagrangian Relaxation

MCM Monte Carlo Method

OOP Object Oriented Programming

OTF Obstacle Triangular Face

PCLT Probability central limit theorem

PDF Probability Density Function

PE Probable Error

PH Pythagorean Hodograph

RRT Rapidly-expanding Random Tree

SA Simulated Annealing

UAV Unmanned Aerial Vehicle

UML Unified Modeling Language

VD Voronoi Diagram

VTK Visualization Tool Kit

XML eXtensible Markup Language

INDEX