ÖAW Austrian Academy of Sciences

Doctoral Program
Computational Mathematics
Numerical Analysis and Symbolic Computation

JKU JOHANNES KEPLER UNIVERSITY LINZ

Computational geometry and combinatorial algorithms for the genus computation problem

Mădalina Hodorog Josef Schicho

DK-Report No. 2010-07 09 2010

A–4040 LINZ, ALTENBERGERSTRASSE 69, AUSTRIA

Supported by

Austrian Science Fund (FWF) Upper Austria

FWF Der Wissenschaftsfonds.

LAND OBERÖSTERREICH

DK sponsors:

- **Johannes Kepler University Linz** (JKU)
- **Austrian Science Fund** (FWF)
- **Upper Austria**

Computational geometry and combinatorial algorithms for the genus computation problem *

Mădălina Hodorog, Josef Schicho
Johann Radon Institute for Computational and Applied Mathematics
Austrian Academy of Sciences
Research Institute for Symbolic Computation
Johannes Kepler University Linz Austria
{madalina.hodorog, josef.schicho}@oeaw.ac.at

Abstract

Computational geometry and combinatorial algorithms play an important role in algebraic geometry. We report on several computational geometry and combinatorial algorithms needed for computing the genus of a plane complex algebraic curve with numeric coefficients.

1 Introduction

The genus computation problem is a classical subject in computer algebra. In [3] we have already presented a method for computing the genus of a plane complex algebraic curve, whose defining polynomial has numeric coefficients, based on the topology of singular points of the curve and on knot theory. The main idea is to divide the genus computation problem in several subproblems, to solve each of these subproblems and than to combine the solutions of these subproblems in order to obtain the solution to the original problem. In this paper, we describe in details the computational geometry and combinatorial algorithms that are required in order to solve the genus computation problem. We base some of these algorithms on general algorithms from computational geometry (e.g. Bentley-Ottman algorithm). Whenever required, we design new algorithms for solving the specific subproblems. We also include a concise description for the computational complexity of the algorithms.

2 The Genus Computation Problem

2.1 Formulating the genus computation problem

We formulate the genus computation problem in the following way: given a plane complex algebraic curve with numeric coefficients, we want to compute the value for the genus of this input curve. For the input curve $C = \{(z, w) \in \mathbb{C}^2 | F(z, w) = 0\} \in \mathbb{C}^2$ we are given its defining polynomial $F(z, w) \in \mathbb{C}[z, w]$

*The work is supported by the Austrian Science Funds (FWF) under the grant W1214/DK9

with numeric coefficients, and its degree d. We intend to compute the value for the genus of C denoted with$genus(C) \in \mathbb{Z}$ using the following formula:

$$genus(C) = \frac{1}{2}(d-1)(d-2) - \sum_{s \in Sing(C)} \delta_s,$$

where $Sing(C)$ is the set of all singularities of the input curve C, and with δ_s we denote the delta-invariant of the singularity s. We consider C as a real two-dimensioanl subset in $\mathbb{C}^2 \cong \mathbb{R}^4$, that is

$$C = \{(x, y, u, v) \in \mathbb{R}^4 | F(x, y, u, v) = 0\} \subset \mathbb{R}^4,$$

where $z = x + i * y, w = u + i * v$. In order to solve the problem we divide it into several interdependent subproblems, we solve each of these subproblems and then we combine the solutions of these subproblems to obtain the solution to our original problem. We divide the genus computation problem into the following subproblems: firstly, we compute all the singularities of the input curve; secondly, we compute the algebraic link of each singularity; then, we compute the Alexander polynomial of each algebraic link; we continue with computing the delta-invariant of each singularity from the Alexander polynomial; and finally, we compute the genus of the input curve from the delta-invariants of all the singularities.

2.2 Solving the genus computation problem

We shortly present the main ideas for solving each of the interdependent subproblems introduced to solve the genus computation problem. For more details, the reader is advised to consult [3].

Computing the singularities of the input curve

We compute the set of all the distinct real singularities of the curve both in the affine and in the projective space, that is the set:

$$Sing(C) = \{(z_0, w_0) \in \mathbb{C}^2 | F(z_0, w_0) = 0, \frac{\partial F}{\partial z}(z_0, w_0) = 0, \frac{\partial F}{\partial w}(z_0, w_0) = 0\}.$$

The algorithm for computing the set $Sing(C)$ is described in detail in [3]. In the future, we intend to compute the set of all the distinct complex singularities of the input curve C.

Computing the algebraic link of each singularity

We translate each of the singularity in the origin. For each of the translated singularity, we compute its topology called the algebraic link of the singularity. The algorithm for computing the algebraic link of a translated singularity is straightforward: we take the input curve with the translated singularity in the origin; we consider the sphere centered in the origin and of a small radius ϵ; we intersect the input curve with this sphere obtaining a real subset X in $\mathbb{R}^4$; we project X from $\mathbb{R}^4$ to $\mathbb{R}^3$ using the stereographic projection f:

$$f : S^3 \setminus \{P\} \subset \mathbb{R}^4 \rightarrow \mathbb{R}^3 \ ,$$

$$(x, y, u, v) \rightarrow (a, b, c) = (\frac{x}{\epsilon - v}, \frac{y}{\epsilon - v}, \frac{u}{\epsilon - v}) \ .$$

For small ϵ, the image of X under f is a link, called the algebraic link of the singularity. Using the identical algorithm described in [3], $f(X)$ is computed as an implicit curve in $\mathbb{R}^3$ given as the intersection of two surfaces G, H in $\mathbb{R}^3$:

$$f(X) = \left\{ (a, b, c) \mid \begin{array}{l} G := Re(F(\frac{2a\epsilon}{n}, \frac{2b\epsilon}{n}, \frac{2c\epsilon}{n}, \frac{-\epsilon + a^2\epsilon + b^2\epsilon + c^2\epsilon}{n})) = 0, \\ H := Im(F(\frac{2a\epsilon}{n}, \frac{2b\epsilon}{n}, \frac{2c\epsilon}{n}, \frac{-\epsilon + a^2\epsilon + b^2\epsilon + c^2\epsilon}{n})) = 0 \end{array} \right\},$$

where $n = 1 + a^2 + b^2 + c^2$, and $Re(F), Im(F)$ are the real, respectively the imaginary part of the complex polynomial $F(z, w)$.

We define a knot as a simple closed curve in $\mathbb{R}^3$ that does not intersect itself, and a link as a finite union of disjoint links. In fact $f(X)$ is a differentiable algebraic link which we denote with L, that is L is a differentiable simple closed curve in $\mathbb{R}^3$ that does not intersect itself, which arises as the intersection of an algebraic curve with a sufficiently small sphere. In order to compute the implicit simple closed curve L in $\mathbb{R}^3$ as the intersection of the two surfaces G, H in $\mathbb{R}^3$, we use the Axel algebraic geometric modeler, developed at INRIA, Sophia-Antipolis [5]. Axel uses certified algorithms [1] to compute a piecewise linear approximation $L^{'}$ of L which is isotopic with L. $L^{'}$ is computed as a graph denoted with $Graph(L)$, as described in Section 3. In fact, $Graph(L)$ is an piecewise simple closed curve approximation for L.

Computing the Alexander polynomial of each algebraic link

After we compute $Graph(L)$ an approximation of the algebraic link L, we are interested in computing the Alexander polynomial of $Graph(L)$. Our algorithm as described in [3] for computing the Alexander polynomial of an algebraic link L, takes as an input $D(L)$ the diagram of L as defined in Section 3. Therefore we need some algorithms to transform the approximation of the algebraic link $Graph(L)$ into the corresponding diagram of the link $D(L)$. All these algorithms are computational geometry and combinatorial algorithms, and they result from the graph data structure returned by Axel. We describe in detail these algorithms in Section 3.

Computing the delta-invariant of each singularity

Based on Milnor's research [4], from the Alexander polynomial of each algebraic link of each singularity we deduce a formula for the delta-invariant of each singularity of the input curve. For each singularity, we denote with m the number of variables in the Alexander polynomial of its corresponding algebraic link and with d the degree of the same Alexander polynomial. In the case of a univariate Alexander polynomial of an algebraic link of a singularity s, the delta-invariant of s denoted with δ_s is computed with the formula $\delta_s = \dfrac{d}{2}$. In the case of a multivariate Alexander polynomial of an algebraic link of a singularity s, δ_s is computed with the formula $\delta_s = \dfrac{d + m}{2}$.

Computing the genus of the input curve

From the delta-invariants of all the singularities of the input curve C, we compute the genus of the curve denoted with $genus(C) \in \mathbb{Z}$ using the formula:

$$genus(C) = \frac{1}{2}(d-1)(d-2) - \sum_{s \in Sing(C)} \delta_s,$$

where d is the degree of the input curve C, $Sing(C)$ in the set of all singularities of C and δ_s is the delta-invariant of the singularity s of the input curve C.

3 Computational geometry and combinatorial algorithms for the genus computation problem

All the computational geometry algorithms presented in this paper are implemented in Axel [5], written in C++ with Qt Script for Applications (QSA) on a Macintosh 2.2 GHz Intel Core 2 Duo, with 2 GB 667 MHz DDR2 SDRAM and Mac OS X, version 10.5.8 operating system. All the algorithms are included in one of Axel's plugin, which is called GENOM3CK, Symbolic numeric techniques for GENus cOMputation of plane Complex algebraiC Curves using Knot theory. Firstly, we introduce some notations, that we will use during the rest of this paper.

Notations

- A graph is given as a pair $G = \langle P, E \rangle$, where P is a set of points in the 3-dimensional space together with their euclidean coordinates, and E is a set of edges connecting them, that is $V = \{p = (a, b, c) \in \mathbb{R}^3\}$ and $E = \{(i, j) \mid i, j \in P\}$;
- A point in the graph is given as a 4-tuple $p(index, x, y, z)$, where $index \in \mathbb{Z}$ uniquely identifies each point in the graph, and $(x, y, z) \in \mathbb{R}^3$ are the euclidean coordinates of the point.
- An edge in the graph is given as a 2-tuple $e(source, destination)$, where *source* is the index of the source point of e and *destination* is the index of the destination point of e. For simplicity reasons, we denote the pair $e(source, destination) := e(s, d)$.
- We use the dot notation . for accessing the elements of a tuple, i.e. $p.index$, $p.x$, $p.y$, $p.z$, $e.s$, $e.d$;
- For a random point $p(index, x, y, z)$, we introduce the following notations:
  - $\texttt{point3D}(index) = (x, y, z) \in \mathbb{R}^3$
  - $\texttt{point2D}(index) = (x, y) \in \mathbb{R}^2$
  - $\texttt{xcoord}(index) = x \in \mathbb{R}$
  - $\texttt{ycoord}(index) = y \in \mathbb{R}$
  - $\texttt{zcoord}(index) = z \in \mathbb{R}$

- For a random edge $e(a, b)$ with $a, b \in \mathbb{Z}$, we use the notations:
  - $\texttt{source}(e) = a \in \mathbb{Z}$
  - $\texttt{destination}(e) = b \in \mathbb{Z}$
- Given a vector (or a list), we access its i-th component using the underscore notation for the index i, i.e we use sw_0 to denote the first element of a vector (or list) denoted SW. We consider that the indexes of a vector (or a list) start from 0. We also distinguish between the name of the list, which is denoted with upper case letters (i.e. SW), and the elements of the list themselves that are denoted with lower case letters (i.e. sw_0).
- Given a vector or a list called for example SW, we use $\texttt{length}(SW)$ to denote its length.
- In order to state that an object is not empty we will use the predicate symbol $IsNotEmpty(object)$ which will be true when the object is not empty, and false otherwise.
- We will use the `Null` pointer as in the C++ programming language, whenever an algorithm has "nothing" as its returning value.

Remark. For $Graph(L)$, which is the approximation of a differentiable algebraic link computed with Axel [5], each *index* appears two times in the set of points P of the graph $Graph(L)$, because $Graph(L)$ is in fact a circuit. A circuit is a path which ends at the point it begins, and a path is a sequence of consecutive edges in the graph. In the rest of this paper, we will use the $Graph(L)$ notation to refer to the data structure returned by Axel, but we will always think of this $Graph(L)$ as a circuit, that is a special type of graph which contains a sequence of consecutive edges and that ends at the point it begins.

Moreover, for $Graph(L) \in \mathbb{R}^3$ with the set of points $p_i = (x_i, y_i, z_i) \in \mathbb{R}^3$ we consider its projection in $\mathbb{R}^2$ with the set of points $p_i = (x_i, y_i) \in \mathbb{R}^2$. We also consider no vertical edges in the projection. We consider the edges of $Graph(L)$ to be "small" edges, i.e. the projection of any edge of $Graph(L)$ has at most one double point. In the rest of this paper, whenever we refer to the projection of $Graph(L)$, we consider the projection as described here, with no vertical edges and only "small" edges.

BASIC COORDINATE GEOMETRY ALGORITHMS

First of all, we design several algorithms from coordinate geometry, which we call basic coordinate geometry algorithms, since they are straightforward algorithms and they do not imply a lot of complicated computations. These algorithms will allow us:

- to compute the slope of a line, given two points in $\mathbb{R}^2$;
- to compute the Y-intercept of a line, given two points in $\mathbb{R}^2$;
- to compute the equation of a line, given its slope and its Y-intercept;
- to compute the equation of a line, given two points in $\mathbb{R}^2$;
- to decide whether a point in $\mathbb{R}^2$ lies on a line of a certain equation.

We denote the slope of a line in the coordinate plane system Oxy with m. We define m as the ratio of the change in the y-value over the corresponding change in the x-value, between two distinct points on the line. We consider $A(a,b),\ B(u,v)$, two points with given coordinates in $\mathbb{R}^2$. Based on the definition, the slope of the line AB denoted with m is defined as the ratio $m = \dfrac{\textit{change in y-value}}{\textit{change in x-value}} = \dfrac{v-b}{u-a}$, assuming that $(u-a) \neq 0$. We notice that if $(u-a) = 0$, then the value of the slope m is ∞, thus undefined. In fact if $m = \infty$, then the corresponding line is a vertical line, i.e. a parallel line to the y-axis. We remember also that if two lines have the same slope, then they are parallel.

Algorithm 1 `GetSlope`(A, B)

Require: $A(a,b),\ B(u,v) \in \mathbb{R}^2$ two points in $\mathbb{R}^2$
Ensure: m_{AB} the slope of the edge (and line) AB

1: **if** $(u-a) \neq 0$ **then**
2: **return** $m_{AB} = \dfrac{v-b}{u-a}$
3: **else**
4: **print** `zero denominator!`
5: **end if**

We denote the Y-intercept of a line in the coordinate plane system Oxy with n. We define n as the distance on the y-axis from the origin $O(0,0)$ to the point where the line intercepts the y-axis of the coordinate plane system Oxy. If we consider $P(0,n)$ to be the point where the line intercepts the y-axis, then the Y-intercept equals the y-coordinate of the point P. Given the same points $A(a,b),\ B(u,v)$ as in the previous algorithm and supposing the equation of the line AB determined by the two points is $y = mx + n$, where $m = \dfrac{v-b}{u-a}$ is the slope of the line as computed with the algorithm `GetSlope`(A, B) and n is the Y-intercept of the line, we can compute the value of the Y-intercept as $n = y - mx = y - \dfrac{v-b}{u-a}x$. The point $B(u,v)$ belongs to the line AB, so

we obtain the value for the Y-intercept to be $n = v - \frac{v-b}{u-a}u = \frac{b \cdot u - a \cdot v}{u-a}$, assuming that $(u-a) \neq 0$.

Algorithm 2 `GetYIntercept`(A, B)

Require: $A(a,b),\ B(u,v) \in \mathbb{R}^2$ two points in $\mathbb{R}^2$
Ensure: n_{AB} the yIntercept of the edge (and line) AB

1: **if** $(u-a) \neq 0$ **then**
2: **return** $n_{AB} = \frac{b \cdot u - a \cdot v}{u-a}$
3: **else**
4: **print** `zero denominator!`
5: **end if**

A line in the coordinate plane system Oxy, with slope m and Y-intercept n, has the defining equation $y = m * x + n$ or equivalently $m * x - y + n = 0$. Given the same points $A(a,b),\ B(u,v)$ with $m = \frac{v-b}{u-a}$ and $n = \frac{b \cdot u - a \cdot v}{u-a}$ computed as in the previous algorithms for $(u-a) \neq 0$, we get the following form for the defining equation of the line AB, $y = \frac{v-b}{u-a} * x - \frac{b \cdot u - a \cdot v}{u-a}$. A straightforward computation produces the equation of the line $AB : (b-v)x + (u-a)y + (a \cdot v - b \cdot u) = 0$. We notice that for the equation of a line computed from two points in $\mathbb{R}^2$, it is enough to return the coefficients of the polynomial equation in x, y, as described in the following algorithm.

Algorithm 3 `EqnLine`(A, B)

Require: $A(a,b),\ B(u,v) \in \mathbb{R}^2$ two points in $\mathbb{R}^2$
Ensure: The real coefficients $m,\ n,\ p$ for the equation of the line $AB : mx + ny + p = 0$

1: $m = (b-v)$
2: $n = (u-a)$
3: $p = a \cdot v - b \cdot u$
4: **return** (m, n, p)

In the rest of this paper, we will think of an edge from the projection of the data structure $Graph(L)$ as a line in the plane coordinate system Oxy. We notice that if we are given an edge as a pair $e(s,d)$, where s is the index of its source point and d is the index of its destination point, then we can compute the equation of the edge using the algorithm `EqnLine`(A, B) where the coordinates of the source and destination points of e are given by $A(xcoord(s), ycoord(s))$, and $B(xcoord(d), ycoord(d))$. Thus if we are given a point $Q(q,r)$ and an edge as a pair $e(s,d)$, we can first compute the equation of the given edge and then decide whether the point $Q(q,r)$ belongs to the edge $e(s,d)$ or not. If we suppose that the equation of the edge returned by the algorithm `EqnLine`(A, B) is $mx + ny + p = 0$, then we can compute: $value = m * q + n * r + p$. The point $Q(q,r)$ belongs to the edge $e(s,d)$ if and only if $value = 0$.

We notice that all the basic coordinate geometry algorithms require $O(1)$ constant time and constant space/memory for their computation.

Algorithm 4 `EvalAtPointEqnLine`$(Q, e(s,d))$

Require: $Q(q,r) \in \mathbb{R}^2$ a point in $\mathbb{R}^2$
$e(s,d)$, an edge in $\mathbb{R}^2$
Ensure: The real *value* of the equation of the edge e evaluated at the point Q

1: $a = \texttt{xcoord}(s), b = \texttt{ycoord}(s)$
2: $u = \texttt{xcoord}(d), v = \texttt{ycoord}(d)$
3: Take $A(a,b), B(u,v)$
4: (m,n,p)=EqnOfLine(A,B)
5: $value = m * q + n * r + p$
6: **return** *value*

Why do we need all the computational geometry and combinatorial algorithms? All the algorithms referring to basic coordinate geometry will be used in more elaborate computational geometry and combinatorial algorithms, whose main purpose is to transform the $Graph(L)$ data structure returned by Axel, which represents the piecewise linear algebraic link (which is in fact the approximation of the differentiable algebraic link computed with the stereographic projection method) into the corresponding diagram of the algebraic link $D(L)$. The diagram of the algebraic link $D(L)$ is needed as input for the algorithm that computes the Alexander polynomial of the algebraic link (see [3]). By definition a regular projection is a linear projection for which no three points on the knot project to the same point, and no vertex projects to the same point as any other point on the knot. A crossing point is an image of two knot points of such a regular projection from $\mathbb{R}^3$ to $\mathbb{R}^2$. Having defined these notions, we can define the diagram of an algebraic link to be the image under regular projection together with the information at each crossing point telling which brach goes over and which goes under. We thus classify the crossing points in overcrossings and undercrossings. Moreover, a diagram together with an arbitrary orientation of each knot in the link is called an oriented diagram. We remember that the projection of $Graph(L)$ has no vertical edges and consists only of "small" edges (i.e. the projection of any edge has at most one crossing point). In order to transform the projection of $Graph(L)$ into $D(L)$, we need to compute the elements of $D(L)$, i.e. the number of knot components in the diagram, plus for each knot component its crossings with their types, and its arcs in the diagram.

ALGORITHM FOR DETECTING THE CROSSING POINTS OF A KNOT DIAGRAM

For each algebraic link of a singularity, Axel will return a $Graph(L)$ data structure. The following algorithm which is an adapted version of the Bentley-Ottman algorithm computes the intersection points of all the edges of the projection of $Graph(L)$ and some extra information:

- for each intersection point p the pair of edges (e_i, e_j) that contains p;
- and each pair of edges (e_i, e_j) is ordered, i.e. e_i is under e_j in $\mathbb{R}^3$.

We notice that these computed intersection points together with the extra information basically coincide with the crossings of $D(L)$. Our adapted Bentley-Ottman algorithm operates in several computational steps, which we describe in detail.

Step 1 (Ordering criteria). The edges of the projection of $Graph(L)$ are oriented from left to right and they are ordered in a list $E = \{e_0, ..., e_N\}$ as follows: (1) by the x-coordinates of their source points; (2) if the x-coordinates of the source points of two edges coincide, then the two edges are ordered by the two slopes of their supporting lines; (3) if the x-coordinates of the source points and the slopes of two edges coincide, then the two edges are ordered by the y-coordinates of their destination points. The ordering criteria is necessary for the correctness of the algorithm.

Step 2 (Sweep line paradigm). We consider a vertical sweep line l that sweeps the plane from left to right. While l moves, it intersects several edges from E. The list of edges that intersect l at one point during the sweeping process, denoted SW, is called the sweep list. SW changes while l sweeps the plane. The algorithm is based on the key observation that SW is updated only at certain points of the edges from E called event points. The sweep list SW is ordered in this algorithm by the y-coordinates of the intersections of the edges of E with the sweep line L. We notice that each *index* appears two times in E since $Graph(L)$ is a circuit. Due to this property, we can manage SW in a simpler way in our adapted Benttley-Ottman algorithm than in the original version.

Step 3 (Initialization). We consider E the list of ordered edges as described in *Step 1*, and SW the sweep list as described in *Step 2*. We denote with I the list of intersection points of all the edges of the projection of $Graph(L)$, and with EI the list of all pairs of intersection edges that contain the intersection points. At the end of the algorithm, the i-th element of the list EI will represent the pair of edges that contains the i-th intersection point from the list I, with the extra information that the first edge from the pair of edges is under the second edge from the pair of edges in $\mathbb{R}^3$. In the initialization step of our adapted Bentley-Ottman algorithm, E contains all the ordered edges of the projection of $Graph(L)$, SW contains always the first two edges of E, while I and EI are empty.

Step 4 (Sweep list management). While we traverse E from its third position to its end (the first two position of E are inserted in SW in the initialization step), we need to insert the current edge $e_m(s_m, d_m)$ from E in SW in the right position and that is:

- we search for an edge $e_n(s_n, d_n)$ in SW such that its destination coincide with the source of $e_m \in E$, i.e. $d_n = s_m$; if we find such an $e_n \in SW$ we replace it with $e_m \in E$;
- if such an edge $e_n \in SW$ does not exist, we insert e_m in SW depending on its position against the current edges from SW. We assume $SW = \{e_{i_1}, e_{i_2}, e_{i_3}, ..., e_{i_k}\}$, with $e_{i_q} \in E$ for all $q \in \{1, ..., k\}$. There exists a unique index j with $0 \leq j \leq k$ such that $ycoord(s_m)$ is larger than the y-coordinates of all the intersections of $e_{i_1}, ..., e_{i_j}$ with l and smaller than the y-coordinates of all the intersections of $e_{i_{j+1}}, ..., e_{i_k}$ with l. This index j can be found by checking all the signs of the determinants $det\big[\big(xycoord(s_m), 1\big), \big(xycoord(s_{i_j}), 1\big), \big(xycoord(d_{i_j}), 1\big)\big]$. Then we insert e_m in SW between the two edges e_{i_j} and $e_{i_{j+1}}$ and we obtain $SW = \{e_{i_1}, e_{i_2}, ..., e_{i_j}, e_m, e_{i_{j+1}}, ..., e_{i_k}\}$.

Each time we insert an edge from E into SW on the right position we have to additionally update SW depending on the event points encountered:

- we test each inserted edge in SW against its two neighbors for intersection. If an intersection point p is found we report it together with the ordered pair of edges that contains it. In addition we swap the edges that intersect in SW. As opposed to the original Bentley-Ottman algorithm after swaping the edges in SW, we do not test the edges against their new neighbors for intersections because we consider only "small" edges.
- we test each inserted edge in SW against its two neighbors for common destination. In addition, when two edges are swapped in SW after reporting their intersection point, we test them against their new neighbors for common destination. Whenever we find two consecutive edges with common destinations we erase them from SW. As opposed to the original Bentley-Ottman algorithm after deleting edges from SW, we do not test the new neighbors for intersection because we consider only "small" edges.

Firstly, we need a basic algorithm to compute the intersection point of two edges e_1, e_2, assuming that this intersection point exists. We call this algorithm `FindIntersection`(e_1, e_2). This algorithm uses an auxiliary algorithm `ComputeIntersection`(e_1, e_2), which effectivelly computes the coordinates of the intersection point $p(x, y)$ of the two edges (e_1, e_2). In order to find the coordinates of the intersection point we solve a linear system of two equations in the x, y unknowns determined by the defining equations of the two edges e_1, e_2. We first extract the coordinates of the source and the destination points of each edge e_1, e_2. Using the `GetSlope` and the `GetYIntercept` algorithms, we then compute the slopes and the Y-intercept of the two edges , denoted with m_1, n_1 for e_1 and with m_2, n_2 for e_2. We get the linear system of equations determined by the equations of the two edges:

$$\begin{cases} m_1 x - y + n_1 = 0 \\ \\ m_2 x - y + n_2 = 0 \end{cases} \tag{1}$$

We assume that $m_1 - m_2 \neq 0$ and using Cramer's rule to solve this system we get as solutions to (1) the coordinates of the intersection point $p(x, y)$ with $x = \dfrac{n_2 - n_1}{m_1 - m_2}$ and $y = \dfrac{m_1 \cdot n_2 - m_2 \cdot n_1}{m_1 - m_2}$.

Algorithm 5 `ComputeIntersection`(e_1, e_2)

Require: $e_1(s_1, d_1), e_2(s_2, d_2)$ two edges in $\mathbb{R}^2$

Ensure: $(x, y) \in \mathbb{R}^2$ the coordinates of the intersection point of the pair of edges (e_1, e_2) with $I(x, y) = e_1 \cap e_2$

1: $a_1 = \texttt{xcoord}(s_1), b_1 = \texttt{ycoord}(s_1), u_1 = \texttt{xcoord}(d_1), v_1 = \texttt{ycoord}(d_1)$
2: $a_2 = \texttt{xcoord}(s_2), b_2 = \texttt{ycoord}(s_2), u_2 = \texttt{xcoord}(d_2), v_2 = \texttt{ycoord}(d_2)$
3: Take $A_1(a_1, b_1), B_1(u_1, v_1)$
4: Take $A_2(a_2, b_2), B_2(u_2, v_2)$
5: $m_1 = \texttt{GetSlope}(A_1, B_1), n_1 = \texttt{GetYIntercept}(A_1, B_1)$
6: $m_2 = \texttt{GetSlope}(A_2, B_2), n_2 = \texttt{GetYIntercept}(A_2, B_2)$
7: **if** $(m_1 - m_2 \neq 0)$ **then**
8: x=$\dfrac{n_2 - b_1}{n_1 - m_2}$, y=$\dfrac{m_1 \cdot n_2 - m_2 \cdot n_1}{m_1 - m_2}$
9: **else**
10: **print** `zero denominator!`
11: **end if**
12: **return** (x,y)

We describe the algorithm `FindIntersection`(e_1, e_2) which test whether two edges e_1, e_2 intersect. If the edges intersect, then the algorithm uses the algorithm `ComputeIntersection`(e_1, e_2) to compute the coordinates of the intersection point $p(x, y)$ of (e_1, e_2). If the edges do not intersect, then the algorithm returns the `Null` pointer. We now describe the test for deciding whether two edges e_1, e_2 intersect or not. For the given edges $e_1(s_1, d_1), e_2(s_2, d_2)$ we first extract the coordinates of the source and destination points. We assume that e_1 has the source $A_1(a_1, b_1)$ and the destination $B_1(u_1, v_1)$, and e_2 has the source $A_2(a_2, b_2)$ and the destination $B_2(u_2, v_2)$. We then compute the equations of the two edges using their slopes and their Y-intercept, i.e. m_1, n_1 for e_1 and m_2, n_2 for e_2. Thus, we compute the equations of the two edges denoted with $L_1(x, y)$, $L_2(x, y)$. From the computation we obtain: $L_1(x, y) : m_1 \cdot x - y + n_1 = 0$ and $L_2(x, y) : m_2 \cdot x - y + n_2 = 0$. If the edges e_1 and e_2 intersect, then the following two conditions have to be simultaneously fulfilled:

1. condition 1: we consider $L_1(x, y)$ the equation of the edge e_1. If e_1 intersects e_2, then A_2 and B_2 have to be on opposite semiplanes determined by e_1, i.e. the condition $L_1(A_2) \cdot L_1(B_2) < 0$ has to be fulfilled;

2. condition 2: we consider $L_2(x, y)$ the equation of the edge e_2. If e_2 intersects e_1, then A_1 and B_1 have to be on opposite semiplanes determined by e_2, i.e. the condition $L_2(A_1) \cdot L_2(B_1) < 0$ has to be fulfilled;

Algorithm 6 `FindIntersection`(e_1, e_2)

Require: $e_1(s_1, d_1), e_2(s_2, d_2)$ two edges in $\mathbb{R}^2$

Ensure: If the two edges e_1 and e_2 intersect, then return their intersection point, otherwise return the `Null` pointer

1: $a_1 = \texttt{xcoord}(s_1), b_1 = \texttt{ycoord}(s_1), u_1 = \texttt{xcoord}(d_1), v_1 = \texttt{ycoord}(d_1)$
2: $a_2 = \texttt{xcoord}(s_2), b_2 = \texttt{ycoord}(s_2), u_2 = \texttt{xcoord}(d_2), v_2 = \texttt{ycoord}(d_2)$
3: Take $A_1(a_1, b_1), B_1(u_1, v_1)$
4: Take $A_2(a_2, b_2), B_2(u_2, v_2)$
5: $(m_1, n_1, p_1) = \texttt{EqnLine}(A_1, B_1)$
6: $(m_2, n_2, p_2) = \texttt{EqnLine}(A_2, B_2)$
7: value$_1$=`EvalAtPointEqnLine`$(A_2, e_1(s_1, d_1))$
8: value$_2$=`EvalAtPointEqnLine`$(B_2, e_1(s_1, d_1))$
9: value$_3$=`EvalAtPointEqnLine`$(A_1, e_2(s_2, d_2))$
10: value$_4$=`EvalAtPointEqnLine`$(B_1, e_2(s_2, d_2))$
11: **if** $(m_1 = m_2)$ **then**
12: **return** `Null` { the edges are parallel}
13: **end if**
14: **if** (value$_1$∗value $_2 < 0$) and (value$_3$∗value$_4 < 0$) **then**
15: **return** (x,y)=`ComputeIntersection`(e_1, e_2) { the edges intersect }
16: **else**
17: **return** `Null` { the edges do not intersect}
18: **end if**

Secondly, we need a basic algorithm to introduce a current edge $e(s, d)$ from E into the right position in SW as described in *Step 4*. Thus, the algorithm keeps the sweep list SW ordered. We call this basic algorithm `InsertSW`(e, SW). The algorithm `InsertSW`(e, SW) uses an auxiliary algorithm `ComputeDet`(A, B, P). This auxiliary algorithm computes the values of the determinant formed by the coordinates of three points $A(a, b), B(u, v), P(m, n)$. For instance, we assume that the edge $e(s, d)$ has the point $A(a, b)$ as its source point and the point $B(u, v)$ as its destination point. Given the point $P(m, n)$ we want to test whether P lies above or below the edge e in $\mathbb{R}^2$. If the value of the determinant formed by the three points A, B, P computed with the algorithm `ComputeDet`(A, B, P) is positive, then the point P is above the edge e. If the value of the determinant is negative, then the point P is below the edge e, and if the value of the determinant is zero, then the three points are collinear. Moreover, if we assume that the point P is the source point of another edge $f(s, d)$ in $\mathbb{R}^2$, then by computing the value of the determinant formed by the three points A, B, P, we can decide the position of the edge f toward the edge e. If the value of the determinant is positive, then f is above e; if the value of the determinant is negative, then f is below e, and if the value of the determinant is zero, then f and e lie on the same line, i.e. they have a common index point (either source or destination).

We describe the `InsertSW`(e, SW) algorithm. We assume that the edge $e(s, d)$ has the point $P(m, n)$ as its source point. We consider an arbitrary edge at position i from the sweep list denoted with $sw_i(s, d)$, and for which the source point is $A(a, b)$ and the destination point is $B(u, v)$. We notice that, if we assume that the sweep list SW is ordered as described in *Step 4* depending

Algorithm 7 `ComputeDet`(A, B, P)

Require: $A(a, b), B(u, v), P(m, n) \in \mathbb{R}^2$

Ensure: det(A,B,P) $\in \mathbb{R}$, the value of the determinant formed by the three points A,B,P.

1: det(A,B,P)= -u∗b + m∗b + a∗v - m∗v - a∗n + u∗n
2: **return** det(A,B,P)

on the y-coordinates of its edges, then the position of the edge $e(s, d)$ towards the edges from the sweep list can be one of the three cases (Figure 1): either the edge $e(s, d)$ is below sw_i and all the edges from the sweep list SW and there are no other edges below e from SW in $\mathbb{R}^2$; or the edge $e(s, d)$ is above sw_i and all the edges from the sweep list SW and there are no other edges above e from SW in $\mathbb{R}^2$; or finally, the edge $e(s, d)$ is above the edge sw_i from the sweep list, thus e is above all the edges which are below sw_i and e is below all the other edges from SW.

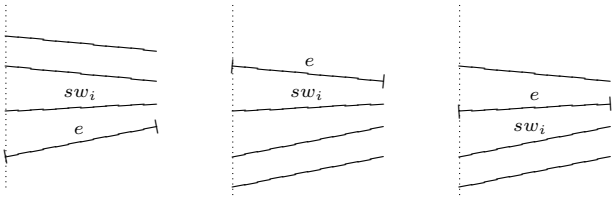

Figure 1: Position of an edge towards the edges from the sweep list

Algorithm 8 `InsertSW`(e, SW)

Require: $e(s, d)$ an edge in $\mathbb{R}^2$
SW the sweep list as described in *Step 2*
Ensure: the modified sweep list SW in which we insert e on the right position as described in *Step 4*

1: **for all** $i = 0$ to `length`(SW) **do**
2: **if** we find sw_i in SW s.t. `source`(e) = `destination`(sw_i) **then**
3: insert e instead of sw_i in SW
4: **else**
5: take $P \leftarrow$ source point of e
6: take $A \leftarrow$ source point of sw_i
7: take $B \leftarrow$ destination point of sw_i
8: $value$ =`ComputeDet`(A, B, P)
9: **if** $(value < 0)$ and (there are no edges in SW below e) **then**
10: insert e before sw_i in SW {e is below sw_i and above no edges}
11: **else if** $(value \geq 0)$ and (there are no edges in SW above e) **then**
12: insert e after sw_i in SW {e is above sw_i and below no edges }
13: **else**
14: insert e after sw_i in SW {e is above sw_i and below several edges}
15: **end if**
16: **end if**
17: **return** SW
18: **end for**

We assume that we know the algorithm `FindIntersection`(e_1, e_2) for finding the intersection point of two edges e_1, e_2, if this intersection point exists, together with its auxiliary algorithm `ComputeIntersection`(e_1, e_2); and the `InsertSW`(e, SW) algorithm for inserting an edge e from E into SW on the right position, together with its auxiliary algorithm `ComputeDet`(A, B, P). We describe the `SweepPlane` algorithm. In this algorithm, `HandleCase1, HandleCase2, HandleCase3` are three procedures which manage the sweep list SW differently, depending on the position on which each edge is inserted in SW. We call a non-trivial position of the sweep list SW, a position of SW different from its first or last position, i.e. i-th of SW is non-trivial if and only of $i \in \{1, ..., length(SW) - 1\}$. If e is inserted on the first position in SW, then we call the `HandleCase1` procedure. If e is inserted on the last position in SW, then we call the `HandleCase2` procedure, and if e is inserted on a non-trivial position in SW then we call the `HandleCase3` procedure. The sweep plane algorithm basically operates in the same way as described below.

Algorithm 9 `SweepPlane`$(G\langle P, E\rangle)$

Require: G the projection of the graph data structure $Graph(L)$ with
P the set of points with $p_i(index, x_i, y_i)$
E the set of ordered edges among the points in P with $e_i(s_i, d_i)$ for $s_i, d_i \in P$
Ensure: I the list of intersection points among all the edges of E
EI the list of pairs of intersection edges which contain the intersection points

1: $E \leftarrow \{e_0, e_1, e_2, ..., e_n\}$ and $SW \leftarrow \{e_0, e_1\}$
2: $I \leftarrow \emptyset$ and $EI \leftarrow \emptyset$
3: **for** $i \leftarrow 2$ to `length`(E) **do**
4: $p \leftarrow$ `InsertSW`(e_i, SW) { the position on which e_i is inserted in SW}
5: **if** $(p = 0)$ **then**
6: `HandleCase1` {e_i is inserted on the first position in SW}
7: **else if** $(p =$ `length`$(SW))$ **then**
8: `HandleCase2` {e_i is inserted on the last position in SW}
9: **else**
10: `HandleCase3` {e_i is inserted on a non-trivial position in SW}
11: **end if**
12: **end for**
13: **return** $\langle I, EI\rangle$

We now describe in detail the three auxiliary procedures `HandleCase1, HandleCase2, HandleCase3` needed for the `SweepPlane` algorithm. Whenever we insert an edge e on the first position in the sweep list SW, we want to test it for intersection with its right neighbour, since in this case we know that the left neighbour of e does not exist. We call an intersection point of two edges degenerate if and only if the intersection point coincide with the destination point of the two edges. Since we do not want to detect degenerate intersection points, we always test e and its neighbour for intersection using the `FindIntersection` algorithm if and only if the two edges do not have the same destination point. If an intersection point is detected, then we insert it in the list of intersection points. In addition, the corresponding pair of intersection edges is inserted in the list of pairs of intersection edges. Moreover, whenever an intersection point p is reported for the pair of edges (e, f) from SW, we have to swap the order of the two edges in the sweep list.

In general, if e and its neighbours have the same destination points, we have to erase them from the sweep list SW in order to assure the correctness of the algorithm. If two neighbouring edges with the same destination points are not erased from SW, then the algorithm will not detect all the intersection points.

We notice that when we insert e on the first position in SW, e can have different neighbours depending if it intersects its right neighbour or not. On the one hand, if e intersects its right neighbour, then the two intersecting edges are swapped in SW. After the swapping operation, e will have a new right neighbour and also a left neighbour which coincides with the original right neighbour of e (see Figure 2). Thus e has to be tested for common destination points both with its left and right neighbour. On the other hand, if e and its original right neighbour does not intersect but they have the same destination points, then we erase them from the sweep list (see Figure 3).

0	1	2	...	n
e	sw_1	sw_2	...	$sw-n$

$\Longrightarrow$

0	1	2	...	n
sw_1	e	sw_2	...	sw_n

Figure 2: Insertion on the first position with intersection detected

0	1	2	...	n
e	sw_1	sw_2	...	$sw-n$

$\Longrightarrow$

0	1	2	...	n
e	sw_1	sw_2	...	sw_n

Figure 3: Insertion on the first position with no intersection detected

We now describe the `HandleCase1` procedure.

Algorithm 10 `HandleCase1`

Require: Same requirements as in `SweepPlane` algorithm
Ensure: I and EI as in `SweepPlane` algorithm

1: **if** `ycoord`$(sw_p) \neq$ `ycoord`(sw_{p+1}) **then**
2: $v =$ `FindIntersection`(sw_p, sw_{p+1}) { exclude degenerate case }
3: **end if**
4: **if** `IsNotEmpty`(v) **then**
5: insert the intersection point v to the list I
6: insert the pair of edges (sw_p, sw_{p+1}) to the list EI
7: swap(sw_p, sw_{p+1}) { intersection detected }
8: **if** `ycoord`$(sw_{p+1}) =$ `ycoord`(sw_{p+2}) **then**
9: erase the edges sw_{p+1}, sw_{p+2} from SW { assure correctness }
10: **end if**
11: **end if**
12: **if** `ycoord`$(sw_p) =$ `ycoord`(sw_{p+1}) **then**
13: erase the edges sw_p, sw_{p+1} from SW { assure correctness }
14: **end if**
15: **return** $\langle I, EI \rangle$

Whenever we insert an edge e on the last position in the sweep list SW, we want to test it for intersection with its left neighbour, since in this case we know that the right neighbour of e does not exist. As in the previous case, since we do not want to detect degenerate intersection points, we always test e and its neighbour for intersection using the `FindIntersection` algorithm only if the two edges do not have the same destination point. If an intersection point is detected, then we insert it in the list of intersection points. Moreover, the corresponding pair of intersection edges is inserted in the list of pairs of intersection edges. As in the previous case, whenever an intersection point p is reported for the pair of edges (e, f) from SW, we have to swap the order of the two edges in the sweep list. In addition, we have to check whether e and its new left neighbour or its right neighbour (which coincide after the swapping with the original left neighbour) have comon destination points (see Figure 4). If they do, then we erase them from SW. Finally, if e and its original left neighbour do not intersect but they have the same destination points, then we erase them from the sweep list SW (see Figure 5).

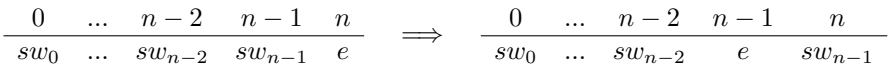

Figure 4: Insertion on the last position with intersection detected

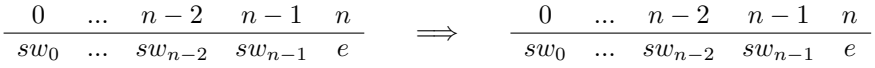

Figure 5: Insertion on the last position with no intersection detected

Algorithm 11 `HandleCase2`

Require: Same requirements as in `SweepPlane` algorithm
Ensure: I and EI as in `SweepPlane` algorithm

1: **if** $\texttt{ycoord}(sw_{p-1}) \neq \texttt{ycoord}(sw_p)$ **then**
2: $v = \texttt{FindIntersection}(sw_{p-1}, sw_p)$ { exclude degenerate case }
3: **end if**
4: **if** `IsNotEmpty`(v) **then**
5: insert the intersection point v to the list I
6: insert the pair of edges (sw_{p-1}, sw_p) to the list EI
7: swap(sw_{p-1}, sw_p) { intersection detected }
8: **if** $\texttt{ycoord}(sw_{p-1}) = \texttt{ycoord}(sw_{p-2})$ **then**
9: erase the edges sw_{p-1}, sw_{p-2} from SW { assure correctness }
10: **end if**
11: **end if**
12: **if** $\texttt{ycoord}(sw_{p-1}) = \texttt{ycoord}(sw_p)$ **then**
13: erase the edges sw_{p-1}, sw_p from SW { assure correctness }
14: **end if**
15: **return** $\langle I, EI \rangle$

Whenever we insert an edge e on a non-trivial position in the sweep list SW, we want to test it for intersection with both its left and right neighbour. We apply the same strategies as before, which are splitted by case, taking care to include all the possible existing cases. Firstly, we consider the case of e and its left neighbour. If the two edges have different destination points, we test them for intersection. If an intersection is detected, then we report it together with the pair of edges that contains it, and we swap the order of the edges of intersection in SW. In addition, if e has common destination points with its neighbours, we erase them from SW (see Figure 6). Secondly, we consider the case of e and its right neighbour. We test the two edges for intersection if they have different destination points. If an intersection point is detected, then we report it together with the pair of edges that contains it, and we swap the order of edges of intersection in SW (see Figure 7). In addition, if e has common destination points with its neighbours, we erase them from SW. Finally, if e does not intersect any of its neighbours, then we have to test it against its left and right neighbour for common destination points (see Figure 8). Whenever two consecutive edges from the sequence $(\texttt{left-neighbour}(e), e, \texttt{right-neighbour}(e))$ have a common destination point, they are erased from the sweep list. Since we have considered only "small" edges, the edge e cannot intersect in the same time both its right and left neighbour, so this case was not included in the treatment of the algorithm.

...	$i-2$	$i-1$	i	$i+1$	...
...	sw_{i-2}	sw_{i-1}	e	sw_{i+1}	...

$\Rightarrow$

...	$i-2$	$i-1$	i	$i+1$	...
...	sw_{i-2}	e	sw_{i-1}	sw_{i+1}	...

Figure 6: Insertion on a non-trivial position with intersection on the left

...	$i-1$	i	$i+1$	$i+2$	...
...	sw_{i-1}	e	sw_{i+1}	sw_{i+2}	...

$\Rightarrow$

...	$i-1$	i	$i+1$	$i+2$	...
...	sw_{i-1}	sw_{i+1}	e	sw_{i+2}	...

Figure 7: Insertion on a non-trivial position with intersection on the right

...	$i-1$	i	$i+1$	...
...	sw_{i-1}	e	sw_{i+1}	...

$\Rightarrow$

...	$i-1$	i	$i+1$	...
...	sw_{i-1}	e	sw_{i+1}	...

Figure 8: Insertion on a non-trivial position with no intersection detected

Algorithm 12 `HandleCase3`

Require: Same requirements as in `SweepPlane` algorithm
Ensure: I and EI as in `SweepPlane` algorithm

1: **if** `ycoord`$(sw_{p-1}) \neq$ `ycoord`(sw_p) **then**
2: $v =$ `FindIntersection`(sw_{p-1}, sw_p) { exclude degenerate case }
3: **end if**
4: **if** `IsNotEmpty`(v) **then**
5: insert the intersection point v to the list I
6: insert the pair of edges (sw_{p-1}, sw_p) to the list EI
7: swap(sw_{p-1}, sw_p) { intersection detected }
8: **if** `ycoord`$(sw_{p-1}) =$ `ycoord`(sw_{p-2}) **then**
9: erase the edges sw_{p-1}, sw_{p-2} from SW { assure correctness }
10: **end if**
11: **end if**
12: **if** `ycoord`$(sw_p) \neq$ `ycoord`(sw_{p+1}) **then**
13: $v =$ `FindIntersection`(sw_p, sw_{p+1}) { exclude degenerate case }
14: **end if**
15: **if** `IsNotEmpty`(v) **then**
16: insert the intersection point v to the list I
17: insert the pair of edges (sw_p, sw_{p+1}) to the list EI
18: swap(sw_p, sw_{p+1}) { intersection detected }
19: **if** `ycoord`$(sw_{p+1}) =$ `ycoord`(sw_{p+2}) **then**
20: erase the edges sw_{p+1}, sw_{p+2} from SW { assure correctness }
21: **end if**
22: **end if**
23: **if** `ycoord`$(sw_{p-1}) =$ `ycoord`$(sw_p) =$ `ycoord`(sw_{p+1}) **then**
24: erase the edges sw_{p-1}, sw_p, sw_{p+1} from SW { assure correctness }
25: **end if**
26: **if** `ycoord`$(sw_{p-1}) \neq$ `ycoord`(sw_p) and `ycoord`$(sw_p) =$ `ycoord`(sw_{p+1}) **then**
27: erase the edges sw_p, sw_{p+1} from SW { assure correctness }
28: **end if**
29: **if** `ycoord`$(sw_{p-1}) =$ `ycoord`(sw_p) and `ycoord`$(sw_p) \neq$ `ycoord`(sw_{p+1}) **then**
30: erase the edges sw_{p-1}, sw_p from SW { assure correctness }
31: **end if**
32: **return** $\langle I, EI \rangle$

As output to the `SweepPlane` algorithm we obtain I the list of intersection points, and EI the list of pairs of intersection edges that contain each intersection point from I. For instance, if p_i is the i-th intersection point from I, then the i-th pair of edges (e_i, f_i) from EI represents the pair of intersection edges that contain the intersection point $p_i = e_i \cap f_i$. We need an algorithm that orderes each pair of intersection edges. In general, we say that the pair of intersection edges (e_1, e_2) that contains the intersection point p is ordered if and only if e_1 is under e_2 in $\mathbb{R}^3$. For both edges e_1, e_2 we extract the coordinates of their source and destination points in $\mathbb{R}^2$. We denote the source and destination points of e_1 with $A_1(a_1, b_1), B_1(u_1, v_1)$, and the source and destination points of e_2 with $A_2(a_2, b_2), B_2(u_2, v_2)$. Assuming that the two edges e_1, e_2 intersect, we compute the coordinates of their intersection point $p(x, y)$ in $\mathbb{R}^2$ with the `ComputeIntersection` algorithm. We compute the equations of the two edges e_1, e_2 with the `EqnLine` algorithm. We denote the equation of e_1 with $L_1(x, y)$, and the equation of e_2 with $L_2(x, y)$. We remember that e_1, e_2 are projections of the original edges of $Graph(L)$ from $\mathbb{R}^3$ that we denote with $e_1^{'}, e_2^{'}$. For both $e_1^{'}, e_2^{'}$ we extract the coordinates of their source and destination points in $\mathbb{R}^3$, that we denote with $A_1^{'}(a_1, b_1, c_1), B_1^{'}(u_1, v_1, w_1)$ for $e_1^{'}$, and with $A_2^{'}(a_2, b_2, c_2), B_2^{'}(u_2, v_2, w_2)$ for $e_2^{'}$. The intersection point $p(x, y) = e_1 \cap e_2$ from $\mathbb{R}^2$ has two corresponding points in $\mathbb{R}^3$: a point $p_1^{'}(x, y, z_1)$ which lies of $e_1^{'}$ with the special property that p and $p_1^{'}$ split e_1 and $e_1^{'}$ in the same proportion factor α_1 since e_1 is the projection of $e_1^{'}$; and a point $p_2^{'}(x, y, z_2)$ which lies of $e_2^{'}$ with the special property that p and $p_2^{'}$ split e_2 and $e_2^{'}$ in the same proportion factor α_2 since e_2 is the projection of $e_2^{'}$. We notice that $p_1^{'}$ and $p_2^{'}$ differ only by their z-coordinate. In fact, if $z_1 < z_2$ then $e_1^{'}$ is under $e_2^{'}$ in $\mathbb{R}^3$. Since e_1, e_2 are the projections of $e_1^{'}, e_2^{'}$, if $z_1 < z_2$ then e_1 is under e_2 in $\mathbb{R}^3$. In this way, the criteria for ordering the pair of intersection edges (e_1, e_2) of the intersection point $p(x, y)$ reduces to computing the corresponding coordinates z_1, z_2 as described above. In order to compute z_1, z_2 we need an algorithm that computes the proportion factors α_1, α_2. For instance, we compute α_1 (in order to compute α_2 or any other proportion factor we proceed in the same way). We consider $A_1(a_1, b_1), B_1(u_1, v_1)$ and $L_2(x, y)$ as computed above. We compute α_1 from the equation $\alpha_1 \cdot L_2(A_1) + (1 - \alpha_1) \cdot L_2(B1) = 0$. We get $\alpha_1 = \dfrac{L_2(B_1)}{L_2(B_1) - L_2(A_1)}$, as described in the `GetAlpha`(A_1, B_1, e_2) algorithm below.

Algorithm 13 `GetAlpha`(A, B, f)

Require: $A(a,b), B(u,v) \in \mathbb{R}^2$ two points in $\mathbb{R}^2$ that determine the edge e
e is the projection of the edge $e^{'}$ from $\mathbb{R}^3$
$f(s,d)$ an edge in $\mathbb{R}^2$ which intersects e in $P(x,y)$
P on e is the projection of $P^{'}(x,y,z)$ on $e^{'}$
Ensure: α s.t. P and $P^{'}$ split e and $e^{'}$ in the same proportion

1: consider $L(x,y)$ the equation of f
2: (`EvalEqnLine` is not called for $L(x,y)$ as it is included in `EvalEqnLine`)
3: compute $v_1 = L(a,b)$ by evaluating $A(a,b)$ in $L(x,y)$: `EvalEqnLine`(A, B, f)
4: compute $v_2 = L(u,v)$ by evaluating $B(u,v)$ in $L(x,y)$: `EvalEqnLine`(A, B, f)
5: compute $\alpha = \dfrac{v_2}{v_2 - v_1}$
6: **return** α

We continue with computing the z_1 coordinate of $p_1^{'}$ as described above. We assume we have computed α_1 with the `GetAlpha`(A_1, B_1, e_2) algorithm. For e_1 the projection of $e_1^{'}$ from $\mathbb{R}^3$, we know $A_1^{'}(a_1, b_1, c_1), B_1^{'}(u_1, v_1, w_1)$ the coordinates of the source and destination point of $e_1^{'}$ in $\mathbb{R}^3$. Thus we compute $z_1 = \alpha_1 \cdot c_1 + (1 - \alpha_1) \cdot w_1$ as described in the `GetZCoordinate`(A_1, B_1, e_2) algorithm below.

Algorithm 14 `GetZCoordinate`(A, B, f)

Require: $A(a,b), B(u,v) \in \mathbb{R}^2$ two points in $\mathbb{R}^2$ that determine the edge e
$A^{'}(a,b,c), B^{'}(u,v,w) \in \mathbb{R}^2$ two points in $\mathbb{R}^3$ that determine the edge $e^{'}$
e is the projection of the edge $e^{'}$ from $\mathbb{R}^3$
$f(s,d)$ an edge in $\mathbb{R}^2$ which intersects e in $P(x,y)$
P on e is the projection of $P^{'}(x,y,z)$ on $e^{'}$
Ensure: z s.t. P and $P^{'}$ split e and $e^{'}$ in the same proportion

1: $\alpha =$ `GetAlpha`(A, B, f)
2: $z = \alpha \cdot c + (1 - \alpha) \cdot w$
3: **return** z

We assume that we know the `GetAlpha` and `GetZCoordinate` algorithms. For a pair of intersection edges (e_1, e_2) of the intersection point $p(x,y)$ in $\mathbb{R}^2$, we compute the corresponding z_1, z_2 coordinates as described above. If $z_1 < z_2$ for the pair of intersection edges (e_1, e_2) of the intersection point $p(x,y)$, then e_1 is under e_2 in $\mathbb{R}^3$ and thus the pair (e_1, e_2) is ordered. If $z_1 > z_2$ for the pair of intersection edges (e_1, e_2) of the intersection point $p(x,y)$, then e_1 is over e_2 in $\mathbb{R}^3$. Thus we swap the two edges in the original pair of intersection edges (e_1, e_2) and we obtain a new pair of intersection edges (e_2, e_1) which is ordered. After applying the `SweepPlane` algorithm on the projection of a $Graph(L)$ data structure returned by Axel, we always apply the `ArrangeEdgesIntersect` algorithm described below on EI the list of pairs of intersection edges returned by `SweepPlane` to order all the pairs of intersection edges in the list.

Algorithm 15 `ArrangeEdgesIntersect`(EI)

Require: EI as returned by the `SweepPlane` algorithm
I as returned by `SweepPlane` the list of intersection points of all the edges of the projection of $Graph(L)$
EI the list of pairs of edges of all the intersection points I
Ensure: The ordered list EI s.t. for each pair $(e_i, f_i) \in EI$ e_i is under f_i in $\mathbb{R}^3$

1: **for all** $i = 0$ to `length`(EI) **do**
2: consider the pair of intersection edges (e_i, f_i) from EI
3: consider A_1, B_1 the source and destinaton points of e_i
4: consider A_2, B_2 the source and destinaton points of f_i
5: $z_1 =$ `GetZCoordinate`(A_1, B_1, f_i)
6: $z_2 =$ `GetZCoordinate`(A_2, B_2, e_i)
7: **if** $(z_1 > z_2)$ **then**
8: swap(e_i, f_i)
9: **end if**
10: **return** EI
11: **end for**

Computational complexity of the algorithm used for detecting the crossing points of a knot diagram. The algorithm used for detecting the crossing points of a knot diagram require $O(n)$ computational (running) time. We notice that the computational time is linear in n the number of edges of the (algebraic) link diagram. The algorithm is an adapted version of the usual Bentley-Ottman algorithm, whose computational time is $O(nlogn + Ilogn)$, where n represents the number of edges for which we compute all the intersection points and I represents the number of reported intersection points. The adapted version of the Bentley-Ottman algorithm behaves better because we can use simpler data structures for the event queue and the status of the algorithm: we use lists instead of balanced binary search trees; moreover, we need only one list for the status of the algorithm which we call the sweep list and we denote with SW, since the next event is always determined on the fly, when we traverse the consecutive edges of the link diagram, that always share a common *index*. We easily notice that the `ComputeIntersection`(e_1, e_2), the `FindIntersection`(e_1, e_2) and the `ComputeDet`(A, B, P) algorithms together with the three procedures `HandleCase1`, `HandleCase2`, and `HandleCase3` require $O(1)$ constant computational time; and that the `InsertSW`(e, SW) algorithm require $O(sw)$ linear computational time in the current lenght of the sweep list SW which is always smaller than n. It follows that the computational time for `SweepPlane`$(G\langle P, E\rangle)$ is $O(n)$. We notice that the algorithm `SweepPlane`$(G\langle P, E\rangle)$ requires $O(n)$ linear space/memory for the computation, where n denotes the total number of edges of E, the set of edges of G, which represents the projection of the graph data structure $Graph(L)$. In fact, n represents the number of edges of the link diagram. In addition, the algorithms `GetAlpha`(A, B, f), `GetZCoordinate`(A, b, f) require $O(1)$ constant computational time, thus `ArrangeEdgesIntersect`(EI) requires $O(I)$ linear computational time and $O(I)$ linear computational space in the number of reported intersection points. It follows that `ArrangeEdgesIntersect` is an output-senzitive

algorithm.

ALGORITHM FOR DETECTING THE KNOT COMPONENTS OF AN ALGEBRAIC LINK

For every differentiable algebraic link of a singualrity computed with the stereographic projection method described in Section 2.2, Axel returns a $Graph(L)$ data structure which is an piecewise linear algebraic link which is the approximation of the differentiable algebraic link. If the differentiable algebraic link has several differentiable knot components, then the $Graph(L)$ data structure contains also several piecewise linear algebraic knots that are the approximations of the differentiable knot components. For computing the diagram of the algebraic link, we certainly need to compute all the piecewise linear knot components. From now on we consider only piecewise linear algebraic knot components, which we will simply call knot components. Thus, the following computational algorithm constructs the knot components of the diagram of a link from the projection of $Graph(L)$. It also returns the total number of knot components. We consider E as in the previous algorithm. We denote a positive edge in $\mathbb{R}^2$ for which $\texttt{xcoord}(s) < \texttt{xcoord}(y)$ with $e(s,d)$, and its corresponding negative edge for which $\texttt{xcoord}(s) > \texttt{xcoord}(y)$ with $-e(d,s)$. We notice that the positive edges are oriented from left to right, while the negative ones are oriented from right to left. We denote the knot components with $K_j, j \in \mathbb{N}$. All K_j must satisfy the following two properties:

1. for each edge $e_k(s_k,d_k) \in K_j$ there exists $e_{k+1}(s_{k+1},d_{k+1}) \in K_j$ with $d_k = s_{k+1}$; in this case, we call e_{k+1} the right consecutive edge of e_k, and we call e_k, e_{k+1} two suitable consecutive edges for a knot component.

2. for $K_j = \{e_0(s_0,d_0),...,e_n(s_n,d_n)\}$: $d_n = s_0$. This assures that the knot component is always a circuit in the graph.

We need an algorithm which contructs all the knot components $K_j, j \in \mathbb{N}$ with the two properties. As opposed to the list E which contains only positive edges oriented from left to right, we notice that each list K_j will contain both positive and negative edges. We show how the first knot component of a link can be computed from the projection of $Graph(L)$ data structure. We initialize the first knot K_0 with the first edge $e_0(s_0,d_0)$ from E. Next we look for the edge e_n in E which has a common index, either source or destination, with d_0. If we find $e_n(d_0,d_n) \in E$ then we insert $e_n(d_0,d_n)$ in K_0 as a positive edge. If we find $e_n(s_n,d_0) \in E$ then we insert $-e_n(d_0,s_n)$ in K_0 as a negative edge. In this case, we first need to swap the source and the destination points of the positive edge $e_n(s_n,d_0)$ to obtain its negative edge correspondent $-e_n(d_0,s_n)$, and then we insert $-e_n$ in K_0. We call e_n the right consecutive edge of e_0. After we insert e_n in K_0 we erase it from E. We will always find such an edge e_n in E, because each *index* such as d_0 appears two times in E. We continue with inserting edges in K_0 from E until the *destination* of an inserted edge coincide with s_0 the source of the first edge from K_0. We apply the same strategy to constructs all the knots K_i of $D(L)$ until E is empty, increasing i each time a new knot starts being constructed. At the end of the algorithm, the index i returns the total number of knot components of $D(L)$. We notice that all the knot components that are constructed from E have always a counterclockwise

orientation. In Figure 9 we give a simple example of how to compute such knot components from E the set of ordered edges.

Example. We consider $E = \{e_0, e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}\}$. The algorithm consists of two steps (see Figure 9):

- Step 1: We initialize the first knot component with $K_0 = \{e_0\}$. We continue with inserting edges from E into K_0 into the right direction (either from left to right or from right to left) until two consecutive edges have the same source index. While inserting the edges in K_0, we erase them from E. We obtain: $E = \{\cancel{e_0}, \cancel{e_1}, e_2, e_3, \cancel{e_4}, \cancel{e_5}, e_6, e_7, \cancel{e_8}, e_9, \cancel{e_{10}}, e_{11}\}$ $K_0 = \{e_0, e_4, e_{10}, -e_8, -e_5, -e_1\}$
- Step 2: We initialize $K_1 = \{e_2\}$. We proceed in the same way as in Step 2 and we obtain: $E = \{\cancel{e_2}, \cancel{e_3}, \cancel{e_6}, \cancel{e_7}, \cancel{e_9}, \cancel{e_{11}}\}$ $K_1 = \{e_2, e_6, e_{11}, -e_9, -e_7, -e_3\}$ The algorithm terminates because $E = \emptyset$.

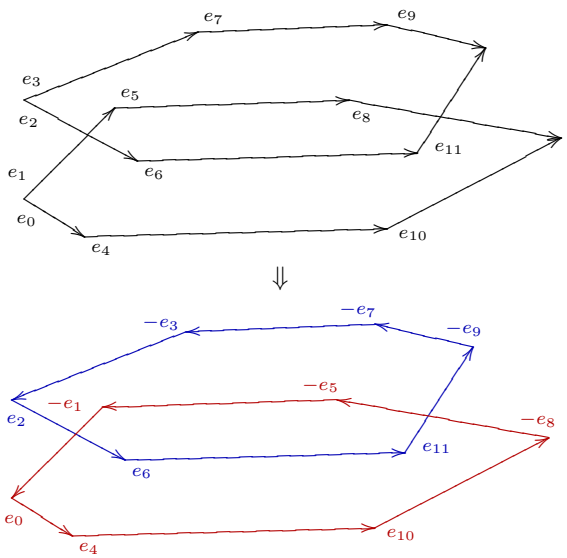

Figure 9: Creating the knot components of an algebraic link

We now describe the `CreateKnots` algorithm.

Algorithm 16 `CreateKnots`$(G\langle P, E\rangle)$

Require: G the projection of the graph data structure $Graph(L)$ with
P the set of points with $p_i(index, x_i, y_i)$
E the set of ordered edges as described in *Step 1* among the points in P with $e_i(s_i, d_i)$ for $s_i, d_i \in P$

Ensure: all the knot components K_j with $j \in \mathbb{N}$ constructed from E with the two properties (1), (2) and
$index$ the number of knot components

1: $index \leftarrow -1$
2: **while** `IsNotEmpty`(E) **do**
3: $index++$
4: consider $K_{index} \leftarrow \emptyset$ a knot component of the link
5: insert e_0 in K_{index}
6: erase e_0 from E
7: **repeat**
8: consider e the last edge in K_{index}
9: **for all** $i \leftarrow 0$ to `length`(E) **do**
10: find e_i the right consecutive edge of e in K_{index} with a common index (either source or destination) with `destination`(e)
11: **if** `destination`$(e) =$ `source`(e_i) **then**
12: insert e_i after e in K_{index}
13: erase e_i from E
14: **end if**
15: **if** `destination`$(e) =$ `destination`(e_i) **then**
16: $swap($`source`$(e_i),$ `destination`$(e_i))$
17: insert e_i after e in K_{index}
18: erase e_i from E
19: **end if**
20: **end for**
21: **until** two suitable consecutive edges have the same source points
22: **end while**
23: **return** $\langle\{K_j, j \in \mathbb{N}\}, index\rangle$

Computational complexity of the algorithm used for detecting the knot components of an algebraic link. The algorithm used for detecting the knot components of an algebraic link denoted with `CreateKnots`$(G\langle P, E\rangle)$ require $O(n^3)$ computational (running) time. We notice that the computational time is polynomial in n the number of edges of the (algebraic) link diagram. In addition the computational memory requiered for the algorithm is $O(n)$ linear in the number of edges of the (algebraic) link diagram.

ALGORITHM FOR DETECTING THE ARCS OF A KNOT DIAGRAM AND FOR DECIDING THE TYPE OF CROSSINGS

For the diagram of a link we can always distinguish between the type of its crossings. A crossing is lefthanded if the underpass traffic goes from left to right or it is righthanded if the underpass traffic goes from right to left. We

denote a lefthanded crossing with -1 (or LH) and a righthanded crossing with $+1$ (or RH), see Figure 11. Moreover, we define the notion of arcs of the diagram of a link. An arc is the part of a diagram between two undercrossings. The algorithm presented in this paragraph constructs the arcs for each knot component of the link. It also decides the type of crossings (righthanded or lefthanded) for each knot component. For constructing the arcs, we consider E as in the previous algorithms. This algorithm operates on the outputs of the previous two algorithms, i.e. the list of intersection points I together with the list of ordered pairs of intersection edges EI, and the lists of edges for all the knot components $K_i, i \in \mathbb{N}$. The key point of the algorithm is to search in K_i all the undergoing edges from EI and to splitt them in two parts. For instance in Figure 10, we consider a diagram of a trefoil knot and we compute its arcs. We assume that for $E = \{e_0, ..., e_n, e_m, ..., e_l, e_k..., e_t, e_s, ..., e_{last}\}$, we compute the following outputs with the previous two algorithms:

$$I = \{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}, EI = \{(-e_n, e_m), (e_l, e_k), (e_s, -e_t)\}$$

$$K_0 = \{e_0, ..., e_k, ..., e_s, ..., e_m, ..., e_l, ..., -e_t, ..., -e_n, ..., -e_1\}$$

We search the three undergoing edges $-e_n, e_l, e_s$ one by one in K_0 and we replace them with $-e_n \to (-e_n^d, -e_n^u), e_l \to (e_l^d, e_l^u), e_s \to (e_s^d, e_s^u)$ obtaining:

$$K_0^{'} = \{e_0, .., e_k, .., e_s^d, e_s^u, .., e_m, .., e_l^d, e_l^u, .., -e_t, .., -e_n^d, -e_n^u, .., -e_1\}.$$

From the definition of an arc, we conjecture that an arc contains the list of edges from a modified knot component $K_i^{'}, i \in \mathbb{N}$ starting with an edge of type $e_j^u, j \in \mathbb{N}$ from $K_i^{'}$ and ending with the next consecutive edge of type $e_k^d, k \in \mathbb{N}$ from $K_i^{'}$. While we insert the edges from $K_i^{'}$ into the list of edges representing the arcs we erase them from $K_i^{'}$. Thus from the modified loop $K_0^{'}$ we compute the following three arcs until $K_0^{'}$ is empty:

$$K_0^{'} = \{e_0, .., e_k, .., e_s^d, \cancel{[e_s^u, .., e_m, .., e_l^d]}, e_l^u, .., -e_t, .., -e_n^d, -e_n^u, .., -e_1\}$$

$$arc_0 = \{e_s^u, ..., e_m, ..., e_l^d\}$$

$$K_0^{'} = \{e_0, .., e_k, .., e_s^d, \cancel{[e_l^u, .., -e_t, .., -e_n^d]}, -e_n^u, .., -e_1\}$$

$$arc_1 = \{e_l^u, .., -e_t, .., -e_n^d\}$$

$$K_0^{'} = \{\cancel{[e_0, .., e_k, .., e_s^d]}, \cancel{[-e_n^u, .., -e_1]}\}$$

$$arc_2 = \{e_n^u, .., -e_1, e_0, .., e_k, .., e_s^d\}$$

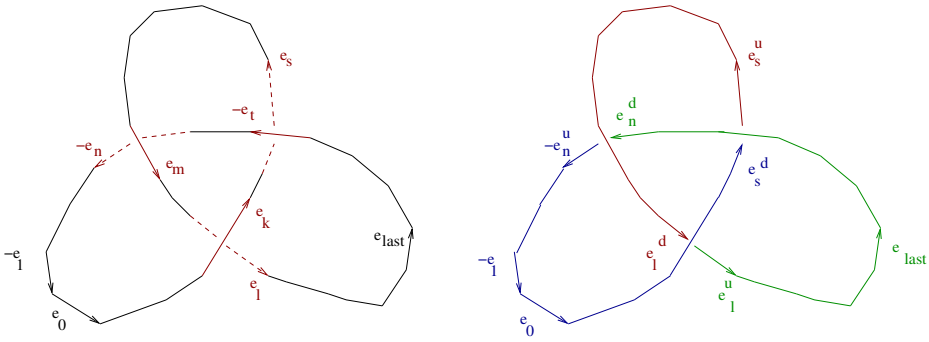

Figure 10: Creating the arcs of a trefoil knot diagram

Algorithm 17 `CreateArcs`$(\langle I, EI\rangle, \langle\{K_j, j \in \mathbb{N}\}, index\rangle)$

Require: I and EI as computed by the `SweepPlane` algorithm
$K_j, j \in \mathbb{N}$ and $index$ as computed by the `CreateKnots` algorithm

Ensure: for each knot component $K_j, j \in \mathbb{N}$ all the arcs of its corresponding diagram are computed

1: **for all** $i \leftarrow 0$ to `length`(EI) **do**
2: consider (e_i, f_i) the i-th pair of intersection edges with e_i under f_i in $\mathbb{R}^3$ as returned by `ArrangeEdgesIntersect`(EI)
3: split the undergoing edge of such a pair $e_i \rightarrow (e_i^d, e_i^u)$
4: search for e_i in all of the K_j knot components and when found replace it in the corresponding K_j where it was found with $e_i \rightarrow (e_i^d, e_i^u)$
5: **end for**
6: $m \leftarrow -1$
7: **for all** $l \leftarrow 0$ to $index$ **do**
8: consider K_l the l-th knot component
9: **while** `IsNotEmpty`(K_l) **do**
10: $m++$
11: consider $arc_m \leftarrow \emptyset$ an arc of K_l
12: insert all the edges from K_l between the first edge of type e^u and the first consecutive edge of type e^d into arc_m
13: delete all the edges inserted in arc_m from K_l
14: **end while**
15: **end for**
16: **return** $\{arc_m, m \in \mathbb{N}\}$

For deciding the type of crossings, we observe that in each knot component for a positive edge $e_i(s_i, d_i) : xcoord(s_i) < xcoord(d_i)$ and for a negative edge $-e_j(s_j, d_j) : xcoord(s_j) > xcoord(d_j)$. Each type of crossing depends on the pair of intersection edges (e_{under}, e_{over}) that contains the corresponding intersection point, and that is: (i) on the orientation of e_{under}, and e_{over}, i.e. whether they are oriented from left to right (positive) or from right to left (negative); (ii) on the comparison relation between the slope of e_{under} and the slope of e_{over}. Depending on these three parameters, we have 2^3 possible cases for deciding the type of crossings. For instance, we consider a crossing c determined by the pair

of ordered edges $\big(-e_l(s_l,d_l), e_k(s_k,d_k)\big)$, for which $-e_l$ is the undergoing edge and e_k is the overgoing edge in $\mathbb{R}^3$. We have $xcoord(s_l) > xcoord(d_l)$ for the negative undergoing edge e_l, and $xcoord(s_k) < xcoord(d_k)$ for the positive overgoing edge e_k. If additionally we suppose that $slope(e_l) < slope(e_k)$, then c is a lefthanded crossing. For instance, in Figure 11 we can decide each type of crossings: $c_1 = (-e_n, e_m)$ is a lefthanded crossing, since $\texttt{xcoord}(s_n) > \texttt{xcoord}(d_n)$, $\texttt{xcoord}(s_m) < \texttt{xcoord}(d_m)$,and $slope(e_m) < slope(-e_n)$; using the same reasoning, $c_2 = (e_l, e_k), c_3 = (e_s, -e_t)$ are both lefthanded crossings. We describe the `DecideTypeCrossings` algorithm. For this algorithm we use the `slope`(e) procedure, which computes the slope of $e(s,d)$. This procedure first extract the x, y coordinates of the source and destination points of e obtaining $A(\texttt{xcoord}(e.s), \texttt{ycoord}(e.s)), B(\texttt{xcoord}(e.d), \texttt{ycoord}(e.d))$. Then it computes its slope as described in `GetSlope`(A, B). We notice that both the `CreateKnots`, `CreateArcs` and `DecideTypeCrossings` algorithms contain also combinatorial aspects.

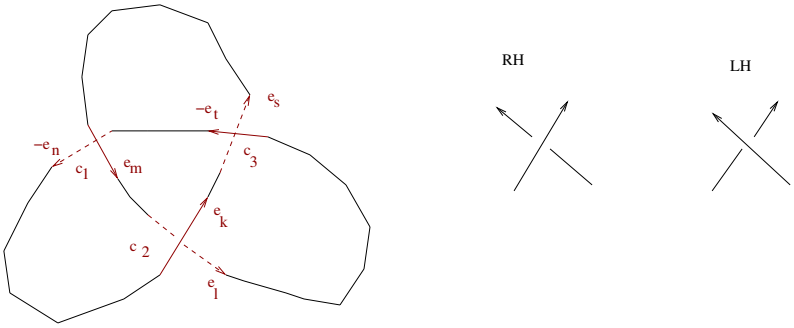

Figure 11: Deciding the type of crossings of a knot diagram (RH or LH)

Algorithm 18 `DecideTypeCrossings`(EI)

Require: EI as computed by the `SweepPlane` algorithm

Ensure: T a list of $\{-1, 1\}$ elements with the same `length` as EI
where the i-th element from T represents the type of the crossing corresponding to the i-th pair of intersection edges (e_i, f_i) from EI

1: **for all** $i \leftarrow 0$ to `length`(EI) **do**
2: consider (e_i, f_i) the i-th pair of intersection edges with e_i under f_i in $\mathbb{R}^3$ as returned by `ArrangeEdgesIntersect`(EI)
3: **if** (`xcoord`$(e_i.s) <$ `xcoord`$(e_i.d)$ and `xcoord`$(f_i.s) <$ `xcoord`$(f_i.d)$ and `slope`$(e_i) >$ `slope`(f_i)) or
(`xcoord`$(e_i.s) <$ `xcoord`$(e_i.d)$ and `xcoord`$(f_i.s) >$ `xcoord`$(f_i.d)$ and `slope`$(e_i) <$ `slope`(f_i)) or
(`xcoord`$(e_i.s) >$ `xcoord`$(e_i.d)$ and `xcoord`$(f_i.s) <$ `xcoord`$(f_i.d)$ and `slope`$(e_i) <$ `slope`(f_i)) or
(`xcoord`$(e_i.s) <$ `xcoord`$(e_i.d)$ and `xcoord`$(f_i.s) <$ `xcoord`$(f_i.d)$ and `slope`$(e_i) >$ `slope`(f_i)) **then**
4: $t_i \leftarrow 1$
5: **end if**
6: **if** (`xcoord`$(e_i.s) <$ `xcoord`$(e_i.d)$ and `xcoord`$(f_i.s) <$ `xcoord`$(f_i.d)$ and `slope`$(e_i) <$ `slope`(f_i)) or
(`xcoord`$(e_i.s) <$ `xcoord`$(e_i.d)$ and `xcoord`$(f_i.s) >$ `xcoord`$(f_i.d)$ and `slope`$(e_i) >$ `slope`(f_i)) or
(`xcoord`$(e_i.s) >$ `xcoord`$(e_i.d)$ and `xcoord`$(f_i.s) <$ `xcoord`$(f_i.d)$ and `slope`$(e_i) >$ `slope`(f_i)) or
(`xcoord`$(e_i.s) <$ `xcoord`$(e_i.d)$ and `xcoord`$(f_i.s) <$ `xcoord`$(f_i.d)$ and `slope`$(e_i) <$ `slope`(f_i))
then
7: $t_i \leftarrow -1$
8: **end if**
9: **end for**
10: **return** $T = \{t_i, i \leftarrow 0, ..., \texttt{length}(EI)\}$

Computational complexity of the algorithm used for detecting the arcs of a knot diagram and for deciding the type of crossings. We denote with n the total number of edges of E, the set of edges of G, which represents the projection of the graph data structure $Graph(L)$. We denote with I the number of intersection points of the edges of the link diagram as computed with `SweepPlane` algorithm, and with k the number of knot components of the link as computed with the `CreateKnots` algorithm. With these notations, we observe that `CreateArcs` algorithm requires $O(n * k * I)$ computational time. In a way, this time is linear in the number of edges, intersection points and knot components of the algebraic link, because the number of intersection points I and the number of knot components k will always be smaller than n, the numbers of edges of the link diagram. With the same notations, we observe that the algorithm `CreateArcs` requires $O(n + k + I)$ computational space.

In addition, the `DecideTypeCrossings`(EI) algorithm requires $O(I)$ linear computational time and linear computational space in the number of intersection points I computed by the `SweepPlane` algorithm.

4 Conclusion

We described several computational geometry and combinatorial algorithms for solving the genus computation problem of plane complex algebraic curves whose defining polynomials have numeric coefficients. The algorithms depend on the geometric properties of the specific problems that they solve. We use the library GENOM3CK ([3]) included in the Axel algebraic geometric modeler (see [5]) for their implementation. The set of algorithms prove to be efficient in producing the required solutions. The efficiency of the set of algorithms is indicated by both the tests experiments but also by the analysis of each of the algorithms. As we have thourughly described in each separate section, the complexity of the algorithms is at most polynomial.

References

[1] Alberti, L., Mourrain, B.: Regularity Criteria for the Topology of Algebraic Curves and Surfaces. IMA Conference on the Mathematics of Surfaces, 1-28, 2007.

[2] Berg, M. de, Krefeld, M., Overmars, M., Schwarzkopf, O.: Computational Geometry: Algorithms and Applications. Second Edition. Springer, 2008.

[3] Hodorog, M., Schicho, J.: A Symbolic-Numeric Algorithm for genus computation.

[4] Milnor J.: Singular Points of Complex Hypersurfaces. Annals of Mathematics Studies. Princeton University Press and the University of Tokyo Press. Priceton, New Jersey, 1968.

[5] Wintz, J.: Algebraic Methods for Geometric Modelling. PhD. Thesis. University of Nice. Sophia-Antipolis, 2008.

Technical Reports of the Doctoral Program "Computational Mathematics"

2010

2010-01 S. Radu, J. Sellers: *Parity Results for Broken k-diamond Partitions and (2k+1)-cores* March 2010. Eds.: P. Paule, V. Pillwein

2010-02 P.G. Gruber: *Adaptive Strategies for High Order FEM in Elastoplasticity* March 2010. Eds.: U. Langer, V. Pillwein

2010-03 Y. Huang, L.X.Châu Ngô: *Rational General Solutions of High Order Non-autonomous ODEs* June 2010. Eds.: F. Winkler, P. Paule

2010-04 S. Beuchler, V. Pillwein, S. Zaglmayr: *Sparsity optimized high order finite element functions for H(div) on simplices* September 2010. Eds.: U. Langer, P. Paule

2010-05 C. Hofreither, U. Langer, C. Pechstein: *Analysis of a non-standard finite element method based on boundary integral operators* September 2010. Eds.: B. Jüttler, J. Schicho

2010-06 M. Hodorog, J. Schicho: *A symbolic-numeric algorithm for genus computation* September 2010. Eds.: B. Jüttler, R. Ramlau

2010-07 M. Hodorog, J. Schicho: *Computational geometry and combinatorial algorithms for the genus computation problem* September 2010. Eds.: B. Jüttler, R. Ramlau

2009

2009-01 S. Takacs, W. Zulehner: *Multigrid Methods for Elliptic Optimal Control Problems with Neumann Boundary Control* October 2009. Eds.: U. Langer, J. Schicho

2009-02 P. Paule, S. Radu: *A Proof of Sellers' Conjecture* October 2009. Eds.: V. Pillwein, F. Winkler

2009-03 K. Kohl, F. Stan: *An Algorithmic Approach to the Mellin Transform Method* November 2009. Eds.: P. Paule, V. Pillwein

2009-04 L.X.Chau Ngo: *Rational general solutions of first order non-autonomous parametric ODEs* November 2009. Eds.: F. Winkler, P. Paule

2009-05 L.X.Chau Ngo: *A criterion for existence of rational general solutions of planar systems of ODEs* November 2009. Eds.: F. Winkler, P. Paule

2009-06 M. Bartoň, B. Jüttler, W. Wang: *Construction of Rational Curves with Rational Rotation-Minimizing Frames via MöbiusTransformations* November 2009. Eds.: J. Schicho, W. Zulehner

2009-07 M. Aigner, C. Heinrich, B. Jüttler, E. Pilgerstorfer, B. Simeon, A.V. Vuong: *Swept Volume Parameterization for Isogeometric Analysis* November 2009. Eds.: J. Schicho, W. Zulehner

2009-08 S. Béla, B. Jüttler: *Fat arcs for implicitly defined curves* November 2009. Eds.: J. Schicho, W. Zulehner

2009-09 M. Aigner, B. Jüttler: *Distance Regression by Gauss–Newton–type Methods and Iteratively Re–weighted Least–Squares* December 2009. Eds.: J. Schicho, W. Zulehner

2009-10 P. Paule, S. Radu: *Infinite Families of Strange Partition Congruences for Broken 2-diamonds* December 2009. Eds.: J. Schicho, V. Pillwein

2009-11 C. Pechstein: *Shape-explicit constants for some boundary integral operators* December 2009. Eds.: U. Langer, V. Pillwein

2009-12 P. Gruber, J. Kienesberger, U. Langer, J. Schöberl, J. Valdman: *Fast solvers and a posteriori error estimates in elastoplasticity* December 2009. Eds.: B. Jüttler, P. Paule

2009-13 P.G. Gruber, D. Knees, S. Nesenenko, M. Thomas: *Analytical and Numerical Aspects of Time-Dependent Models with Internal Variables* December 2009. Eds.: U. Langer, V. Pillwein

Doctoral Program
"Computational Mathematics"

Director:

Prof. Dr. Peter Paule
Research Institute for Symbolic Computation

Deputy Director:

Prof. Dr. Bert Jüttler
Institute of Applied Geometry

Address:

Johannes Kepler University Linz
Doctoral Program "Computational Mathematics"
Altenbergerstr. 69
A-4040 Linz
Austria
Tel.: ++43 732-2468-7174

E-Mail:

office@dk-compmath.jku.at

Homepage:

http://www.dk-compmath.jku.at

Submissions to the DK-Report Series are sent to two members of the Editorial Board who communicate their decision to the Managing Editor.