Universität des Saarlandes, FR Informatik
Max-Planck-Insitut für Informatik, AG 1

Visualization of Points and Segments of Real Algebraic Plane Curves

Masterarbeit im Fach Informatik
Master Thesis in Computer Science
von / by

Pavel Emeliyanenko

Erstgutachter / first examiner

Prof. Dr. Kurt Mehlhorn
MPI Informatik, Saarbrücken

Zweitgutachterin / second examiner

Prof. Dr. Nicola Wolpert
Hochschule für Technik, Stuttgart

Saarbrücken, February 2007

Hilfsmittelerklärung (Non-plagiarism statement)

Hiermit versichere ich, dass ich diese Arbeit selbstandig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Saarbrücken, 13. February 2007

Acknowledgements

First of all I would like to thank Prof. Dr. Kurt Mehlhorn, my supervising professor at Saarland University, for offering me this stimulating topic as well as Algorithm Engineering course that absolutely enriched my knowledge in Algorithms. I am grateful to Dr. Michael Sagraloff, one of my thesis advisors, for his guidance, availability and for the pragmatic suggestions especially in theoretical questions which also greatly expanded my mathematical background. Special thanks to Eric Berberich, another my thesis advisor, for correcting my thesis, constructive criticism and useful suggestions in designing and documenting my code. I also thank to Michael Kerber, the author of AlciX library, who kindly provided me a plenty of examples to test my algorithm and answered a lot of questions related to his library. Last but not the least I would like to thank all IMPRS department for offering me the scholarship, this way helped me to successfully accomplish this work.

Abstract

This thesis presents an exact and complete approach for visualization of segments and points of real plane algebraic curves given in implicit form $f(x, y) = 0$. A curve segment is a distinct curve branch consisting of regular points only. Visualization of algebraic curves having self-intersection and isolated points constitutes the main challenge. Visualization of curve segments involves even more difficulties since here we are faced with a problem of discriminating different curve branches, which can pass arbitrary close to each other. Our approach is robust and efficient (as shown by our benchmarks), it combines the advantages both of curve tracking and space subdivision methods and is able to correctly rasterize segments of arbitrary-degree algebraic curves using double, multi-precision or exact rational arithmetic.

Contents

Chapter 1

Introduction

Problem statement

Planar implicit curves are of great interest in Computer-Aided design and Computer Graphics. They are very useful because of their ability for general function description, for example, to represent the intersection of two parametric surfaces in $\mathbb{R}^3$. A plane algebraic curve is defined as the zero set of a bivariate polynomial $Z(f) = \{(x, y) \in \mathbb{R}^2 : f(x, y) = 0\}$ and might have singularities such as cusps, self-intersections and isolated points, which constitute the interesting cases for us.

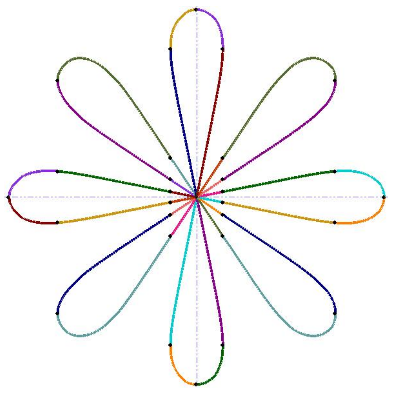


Fig 1.1: Erdõs lemniscate of degree 16 decomposed into sweepable segments. Each segment is drawn in its own color

Algebraic curves can be decomposed into a set of arcs, such that each of them does not contain singularities in the interior. We call such arcs *segments* of an algebraic curve (as an example see Figure 1.1). Computing and visualizing these segments is an important fundamental operation in Computational Geometry. For example, one can draw polygons whose boundaries are defined by segments of generic algebraic curves and perform operations on them.

The problem of visualization of implicit curves is well-studied and there are several classes of algorithms proposed in the literature. Some of them focus at special types of algebraic curves (such as rational curves [AG91, Blo88] or curves defined by fixed-degree algebraic polynomials), or can handle only special kinds of singularities [MY95], another can visualize any implicit function but sacrifice exactness and may incorrectly rasterize some details.

Moreover, to the best of your knowledge, none of them can visualize distinct curve segments. This is no surprise, since for visualization of complete curves the local curve's topology does not play a vital part. Indeed, most of the visualization algorithms interpret a curve as a complete object, without decomposing it into a set of potentially smaller objects. Whereas drawing segments separately imposes additional constraints, because the visualization process depends on the actual curve structure: here several curve segments can lie very close to each other, and discriminating them is, in some cases, a non-trivial task.

The main contribution of this thesis is to provide an algorithm and its implementation for visualizing segments and points of arbitrary-degree plane algebraic curves. Our algorithm is *exact*, since our output is a pixel approximation of the true mathematical result. It is reasonable that some details might be not visible at a certain resolution, but as the curve topology is known, these small details are exposed at a higher resolution.

Furthermore, our approach is *complete*, meaning that we allow as input algebraic curves of arbitrary degree and can handle highly degenerate cases: essentially, when the distance between two curve branches lies beyond the limits of double-precision arithmetic. We developed a technique to separately visualize curve segments located *arbitrarily* close to each other using only *double precision* arithmetic in most cases. If this does not suffice we switch to multi-precision floating-point or exact rational computations.

Computational efficiency is also one of the important goals in our approach. The most time-critical part of segments visualization is separation of them at locations where several branches almost coincide and the difference between them is unobservable. In these cases we stop space subdivision at a certain level, beyond which the curve branches are inseparable by sight. From this point we continue tracing the curve branches in a one "bundle" without any impact on the visual quality. At location where the curve branches again go apart we pick up the right one by using a real root isolation.

Our method is essentially a composite of the ideas taken from space subdivision and curve tracking methods equipped with techniques facilitating separation of closely located curve branches. We use the *exact* method to decompose a curve into a set of x-monotone segments from the AlciX library [Ker06] as a pre-processing step. X-coordinates of segments' end-points are given by real algebraic numbers which can be refined up to an arbitrary precision. Having the exact computation of the curve's topology is an intrinsic requirement for correctness and robustness of our approach. We operate on distinct segments delimited by end-points which can also lie at infinity. We employ pixel techniques to always stay within the level of detail given by the current pixel resolution and omit drawing of potentially imperceptible parts.

If we focus only on drawing x-monotone segments – which is the main objective of this work – then a straightforward solution, for example, could be as follows: *"for each x coordinate within the range between two segment's end-points find an appropriate y coordinate by isolating roots of a univariate polynomial at this x and picking up the one corresponding to our segment.*[1]"

Although rather trivial, this solution is inefficient by definition since it requires to isolate real roots of a univariate polynomial at every pixel comprised the approximation of a curve branch. We do not consider such "heavy weapons" any further. The interested reader can find a detailed description of real root isolation technique in [EKK$^+$05, RZ04]. Moreover, the algorithm presented in this work is able to visualize more "general" segments which are not necessarily x-monotone.

Related Work

Algebraic curves were widely adopted in fields of Computational Geometry, Geometric Modeling, Computer Graphics, etc. That is why an extensive research was carried out to develop visualization algorithms for algebraic curves. These algorithms can be grouped into three main classes which we consider now.

Representation conversion. The idea behind this class of algorithms is to find a parametric representation for an implicit curve provided that it is easier to render parametric curves [AG91, Blo88]. Only rational curves (the curves with geometric genus[2] zero) admit a global parameterization. However, a local analytic parameterization always exists in a neighbourhood of a regular point of an implicit curve. This is a point $\mathbf{p}$ such that $f(\mathbf{p}) = 0$ and $\nabla f(\mathbf{p}) \neq 0$. This allows us to visualize rational implicit curves by means of algorithms for visualization of parametric curves.

Space subdivision. A space subdivision algorithm partitions a space into subspaces recursively, discarding those subspaces that do not intersect the curve [MG03, MG04b, Tau94]. The recursive subdivision terminates when the resulting approximation to the curve by a set of small subspaces is within a given tolerance (a pixel size). Robust algorithms can be implemented by using algebraic techniques and interval (or affine) arithmetic [MSV$^+$02, FS04], algebraic and rational techniques [KM95], floating-point arithmetic [She96]. In theory, the advantage of this class of algorithms relies on the fact that they can render implicit functions of arbitrary com-

[1]It is supposed that we have enough information to uniquely describe a curve segment. Later we give three definitions of a curve segment, one as the refinement of another, which clarifies the statement.

[2]For definition of geometric genus see [Gib98].

plexity but, typically, they are more computationally expensive than algorithms from the other classes and not applicable to segments.

Curve tracking. The idea here is to compute a sequence of points in a way similar to parametric curves. This method has its roots in the Bresenham's algorithm for rendering circles, which is basically a continuation method in image space. Some methods start at a seed point on the curve, then look at the adjacent pixels to find out if any of them is the next pixel along the curve, thus exploiting the discrete nature of pixel images [Cha88]. Other methods, having a current point, consider a small (circular) neighbourhood of it and use some numeric methods (for example, Predictor-Corrector and False Position method) to obtain the next point along the curve [MG04a, MY95]. Altogether, these methods are attractive because they concentrate effort on where it is really needed and may adapt the computed approximation to the local geometry of the curve (for example, with respect to the curvature). Unfortunately, some of them break down at singular points, another can handle singularities of special types only. Besides that, this approach requires a complete set of seed points covering the whole curve which is not trivial to produce.

There are a lot of approaches for the visualization of implicit curves in the literature. Each of them has its own advantages and drawbacks. Below we describe why no one of them can be used to visualize curve segments.

Representation conversion algorithms are applicable to only a small family of algebraic curves and, therefore, are not considered further in this work.

Space subdivision algorithms are well-suited to render complete curves but unable to rasterize a certain curve segment, since they look at a curve "entirely" and without exploiting the continuity of curve segments.

In spite of the fact that *curve tracking* algorithms are basically continuation methods, rendering of complete curves is still less constrained. Indeed, in ambiguous situation, when several curve branches are discovered surrounding the current tracking point, one can continue tracking any of these branches, saving the remaining ones beforehand. In case of drawing curve segments, such uncertainty is intolerable.

Furthermore, even this method works not in all situations. Different authors [MY95, MG04a, ZsYzMj$^+$06] suggest different approaches how to deal with multiple points during curve tracing. But all of them concur in how to detect multiple points. This can be explained as follows: "*divide the circle around the multiple point into* n *sufficiently small equal parts to get the points* $P_0P_1 \ldots P_{n-1}$ *on the circle. If the function* $f(x, y)$ *has sign change between points* P_i *and* P_{i+1} *where* $i = 0, 1, \ldots n-2$ *there is a segment of the implicit curve passing through*

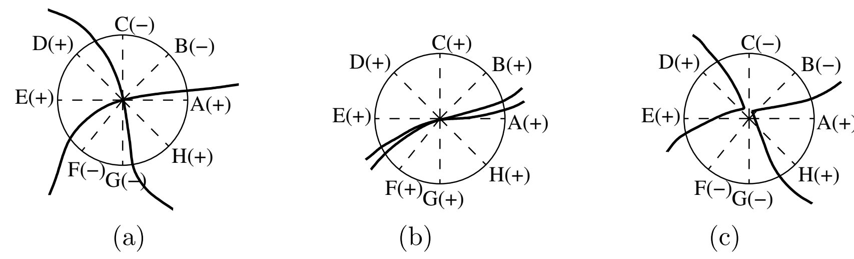


Fig 1.2: A "nice" multiple point (a); a multiple point with closely located branches (b); and a situation where two closely located cusps look like a multiple point (c)

this small arc P_iP_{i+1}. If the number of such arcs greater than 3 the centre of the circle is the multiple point".

It surely works for "nice" cases as depicted in Figure 1.2(a) but as curve branches come closer to each other it might happen that no sign changes will be registered, see Figure 1.2(b). The statement "*divide into n sufficiently small equal parts*" requires further specification. By the same token, there could be some cases outwardly similar to a singular point but, in fact, consisting of two cusps (Figure 1.2)(c).

Our approach combines ideas of curve tracking (to exploit locality and continuation) and space subdivision methods (to separate closely located curve branches). We rely on the approach originally introduced by [Cha88]. In particular, in each step of the algorithm we have the current pixel (already drawn) and 8 possible directions to follow (this is also called "8-way stepping scheme", see Figure 1.3)(a).

In the original algorithm the next pixel is chosen from among eight nearest neighbours of the current pixel by searching for a sign difference in two function evaluations at midpoints between consecutive pixels. However, in some cases this method can fail. That is why, we use range analysis instead of sign computations, which result in more trustworthy curve intersection tests.

Local subdivision solves the problem of discarding neighbouring curve branches. Whenever the neighbourhood test fails to determine the next pixel – because several curve branches occurred in the neighbourhood – we apply the subdivision. As a result, we end up with a sequence of subpixels approximating our curve branch – see Figure 1.3(b).

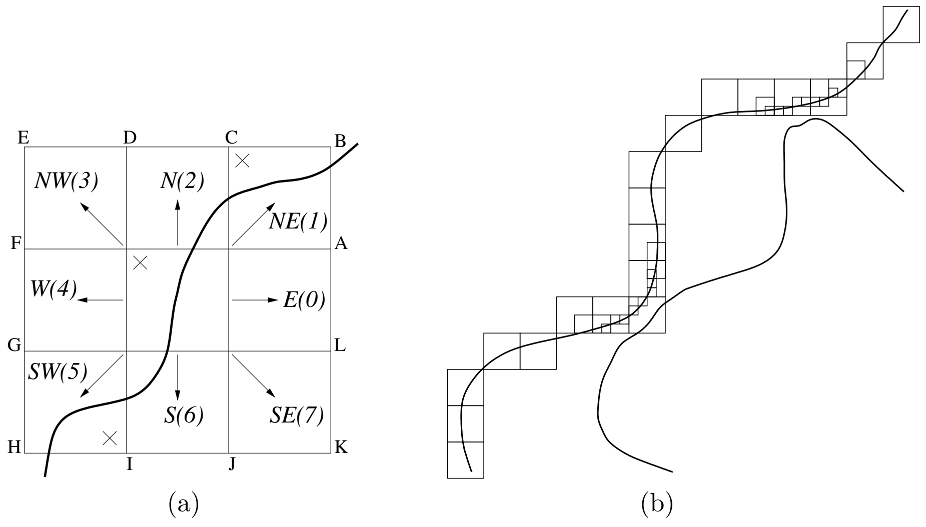


Fig 1.3: 8-way stepping scheme, capital letters define the directions: S - *south*, W - *west*, NE - *north-east* and so on (a); and approximation of a curve branch using adaptive subdivision (b)

Outline

The content of the Thesis consists of four chapters. In Chapter 2 we present the **Mathematical background**. First, we give a definition of algebraic curves and list the types of singularities. Then, we formally define a curve segment to clarify the problem statement. Afterwards, we discuss advantages and drawbacks of various range analysis methods, and at the end of this chapter we present a technique to count the number of curve branches crossing a line-segment based on the Mean Value theorem.

Chapter 3 is devoted to the **Algorithm** itself. We give a pseudo-code for the main procedure and discuss the 4- and 8-way stepping schemes developed to slightly different approaches. We point out the differences between them and survey the advantages of each. The chapter ends with algorithm details such as finding "seed" points, stopping criteria, handling isolated points and it discusses important optimizations such as tracing of visibly coincide branches.

In Chapter 4 we give an introduction to our **Implementation**. We describe the overall structure and different parts of the EXACUS library, outline some important details of the implementation, give running time measurements and compare our algorithm with a sample implementation of the space subdivision approach.

The Thesis ends with **Conclusions** for the accomplished work and outlines the possible directions of future research.

Chapter 2

Mathematical background

Algorithms for visualization of algebraic curves rest on a large number of algebraic operations. In this chapter we will introduce the algebraic concepts and mathematical tools that are fundamental for our computations and enable us to deal with algebraic curves.

First, we will give necessary notations and definitions to make our work self-contained and accessible to the non-experienced reader. Afterwards, we give an overview of range analysis methods and formulate the algorithmic ideas underlying our approach.

2.1 Algebraic curves

The objects we consider and manipulate in our work are real plane algebraic curves represented by polynomials in two variables with real coefficients. A non-zero bivariate polynomial $f \in \mathbb{R}[x, y]$ can be written as:

$$f(x, y) = \sum_{i=0}^{n} \sum_{j=0}^{m} a_{ij} x^i y^j.$$

Such a polynomial can be treated in two different fashions. In *hierarchical* representation, we treat f as a univariate polynomial in the outermost variable y with coefficients in $\mathbb{R}[x]$. In this case we define the degree of f with respect to the outermost variable y as (the x-degree can be defined in a similar way):

$$deg_y(f) = \max\{j : j \leq m, \sum_{i=0}^{n} a_{ij} x^i \neq 0\}.$$

In *flat* representation, we consider f as a sum of its *monomials* $a_{ij} x^i y^j$ with $a_{ij} \neq 0$. The *total degree* of f is defined by:

$$deg(f) = \max\{i + j : i \leq n, j \leq m, a_{ij} \neq 0\}.$$

If f is a nonzero constant, then f has degree 0.

We define an *algebraic curve* in the following way: Let f be a polynomial in $\mathbb{R}[x, y]$. We set $C(f) = \{(\alpha, \beta) \in \mathbb{R}^2 : f(\alpha, \beta) = 0\}$ and call $C(f)$ the *algebraic curve* defined by f. Remark that if two algebraic curve $C(f)$ and $C(g)$ have the same real zero sets it does not necessarily imply that polynomials f and g are the same up to multiplication of scalars. This is only true for curves without multiple components which have infinite real zero set, however there are curves without real zero set at all.

For an algebraic curve $C(f)$ we define its *gradient vector* as $\nabla f = (f_x, f_y) \in (\mathbb{R}[x, y])^2$ with $f_x = \frac{\partial f}{\partial x}$ and $f_y = \frac{\partial f}{\partial y}$. With the help of the gradient vector we can characterize points lying on the curve $C(f)$. We call a point $(\alpha, \beta) \in \mathbb{R}^2$ of $C(f)$ *singular* if $(\nabla f)(\alpha, \beta) = (f_x(\alpha, \beta), f_y(\alpha, \beta)) = (0, 0)$. Accordingly, the remaining points are called *non-singular* or *regular*. The geometric interpretation is that for every point (α, β) of $C(f)$ there exists a unique line tangential to the curve $C(f)$. This tangent is perpendicular to $(\nabla f)(\alpha, \beta)$. Non-singular points of $C(f)$ are the ones that admit a unique tangent line to the curve. The remaining points, such as self-intersections and isolated points, we call singularities. There are only finitely many singular points. In particular, an algebraic curve defined by a polynomial of degree d has at most $(d - 1)(d - 2)/2$ singular points.

Let (α, β) be a non-singular point of $C(f)$. We call (α, β) an *extreme point* if it has a vertical tangent, i.e., $(\nabla f)(\alpha, \beta) = c \cdot (1, 0)$. We refer to singularities and extreme points of $C(f)$ collectively as the *critical points* of $C(f)$. The set of this critical points is exactly the set of intersection points of the curves $C(f)$ and $C(f_y)$.

2.2 Curve segments

Having a comprehensive classification of the curve points, we are able now to give an adequate definition of a curve segment which is necessary to understand the overall problem correctly. We start with the most intuitive definition, then refine it further to satisfy the problem requirements.

Definition 2.1. ***Regular segment.*** *A regular curve segment is a single connected branch of an algebraic curve consisting of regular points only, bounded by two end-points, which are not necessarily regular.*[1]

The second definition refines the previous one in a way that segments do not have critical and extreme points inside (in other words, they are x-monotone):

[1]Please note that one (or two) of segment's end-points can lie at "infinity". In this case we will call the segment *unbounded.*

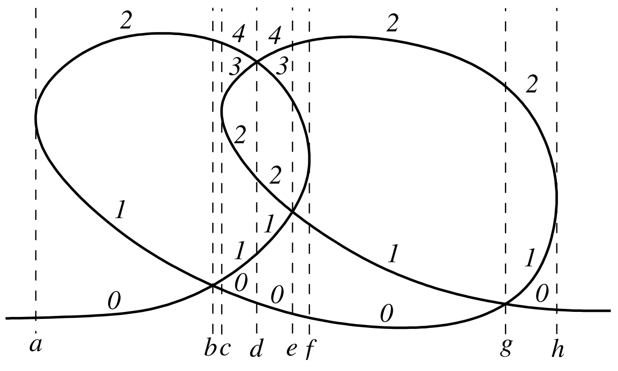


Fig 2.1: Partition of a curve into segments with corresponding arc-numbers

Definition 2.2. ***x-monotone segment.*** *An x-monotone curve segment is a single connected branch of an algebraic curve consisting of non-critical points only, bounded by two (possibly critical) end-points.*

The third refinement partitions a curve into a set of potentially smaller segments suitable for Gaps (Generic Algebraic Curves and Surfaces, for details see section 4.1.2).

Definition 2.3. ***Sweepable segment.*** *A sweepable curve segment is a single connected branch of an algebraic curve defined by the range of x-coordinates between two critical points and an arc number - a rank of a segment amongst all segments with the same x-coordinates range, which is constant in segment's interior.*

Note that the requirement to have a segment defined between two consecutive critical points immediately implies x-monotonicity as well as the fact that it consists of non-critical points only. In Figure 2.1 one can see a decomposition of the curve into sweepable segments. Each vertical line indicates a self-intersection or extreme point. Segments are assigned the arc-numbers from bottom to top, starting with 0.

One can notice, that the segments do not necessarily have critical end-points. Such a decomposition of a curve into segments admits natural ordering of them: we can enumerate segments in lexicographical order, first by the starting x-coordinate, and then by the arc-numbers among all segments with the same x-coordinate range.

The algorithm presented in this work can render segments meeting the first definition, but, strictly speaking, this definition is not of much use, since segments satisfying it can be arbitrary complex in the interior. The effort of rendering such complex segments is comparable to the effort of rendering a complete curve, which is not the main goal of this work.

Further we will refer to the last definition of a curve segment unless it is stated explicitly.

2.3 Range analysis

The aim of range analysis is to find the range of a function (usually a polynomial) in one or several variables over an input interval. In practice, finding an exact range is difficult, and it is more usual to find a range which includes the actual range. Information about the range of a function f, and related functions such as its partial derivatives, inverse, etc. are of considerable interest to people working in the fields of numerical and functional analysis, differential equations, linear algebra, approximation and optimization theory and other disciplines.

Range analysis has many important applications in CAGD and computer graphics, including the plotting and localisation of implicit curves and surfaces.

First we give an overview of the classical technique of Interval Arithmetic (IA), then we consider Affine Arithmetic (AA) which is more accurate but still has essential drawbacks. And finally, we describe Modified Affine Arithmetic (MAA) [SMW$^+$] which is known to be the one of the best methods for polynomial evaluation over an interval. This is due to the fact that it does not suffer from under-conservatism problem as it obeys all commutative, associative and distributive laws.

2.3.1 Interval arithmetic

Interval arithmetic is a technique for numerical computation where each uncertain quantity is represented by an interval of floating-point numbers. These intervals are added, subtracted, multiplied, etc. in such a way that each computed interval is guaranteed to contain the unknown value of the quantity it represents. More precisely, a quantity $x \in \mathbb{R}$ is represented by an interval $[a, b]$, where a and b are floating-point numbers (including $\pm\infty$), such that $a \leq x \leq b$. If $\mathbf{x}$ and $\mathbf{y}$ are intervals and $\odot$ denotes one of the arithmetic operators $+, -, \times$ and $/$, then $x \odot y$ is defined by:

$$\mathbf{x} \odot \mathbf{y} = \{x \odot y \mid x \in \mathbf{x}, y \in \mathbf{y}\}$$

Primitive arithmetic operations can be extended to intervals:

$$\begin{aligned}
[a, b] + [c, d] &= [a + c, b + d], \\
[a, b] - [c, d] &= [a - d, b - c], \\
[a, b] \times [c, d] &= [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)], \\
[a, b]/[c, d] &= [a, b] \times [1/d, 1/c] \quad \text{provided } 0 \notin [c, d].
\end{aligned}$$

Note that lower bounds are rounded downwards and upper bounds - upwards. Similar formulae can be given for extending the elementary functions to intervals.

The natural interval extension of a bivariate polynomial $f(x, y)$, denoted by $\mathbf{f}(\mathbf{x}, \mathbf{y})$, is obtained by replacing each occurrence of x and y in $f(x, y)$ by intervals $\mathbf{x}$ and $\mathbf{y}$, and evaluating the resulting interval expression using the above definitions. The result is itself an interval.

Unfortunately, the range estimates given by standard interval arithmetic tend to be too large, especially in complicated expressions or long iterative computations. The main reason for the overestimation is that IA implicitly assumes the arguments (the quantities) to primitive operations to be independent from each other. Thus, if there are any mathematical constraints between those quantities then not all combinations of values in the given intervals will be valid. In that case, the interval obtained by Interval arithmetic may be much wider that the exact range of the result quantity.

Example 2.1. *To see this, let us compute x^2 over an interval $[x] = [-1, 2]$. The application of the above rules gives $[-2, 4]$, while the exact range for $[x]^2$ is $[0, 4]$.*

A significant property of IA is that the form in which the polynomial is expressed can affect the result. There are modifications of the standard IA which exploit this property such as IA using centered form originally introduced by Moore [Moo66] and Horner form.

Despite the fact that rearranging the function can give tighter bounds on the result, IA still is not as accurate as Affine arithmetic, which we consider next.

2.3.2 Affine arithmetic

Affine arithmetic [FS04] is an alternative approach to IA that is more resistant to overestimation, since AA keeps track of first-order correlations between computed and input quantities; these correlations are automatically exploited in primitive operations, such that the boundaries obtained by applying AA in most cases are much tighter than the ones obtained with IA. Moreover, AA implicitly provides a geometric representation for the joint range of related quantities that can be exploited to increase the efficiency of interval methods.

Below we explain the main concepts of Affine arithmetic. In AA a quantity x is represented by an expression of the form:

$$\hat{x} = x_0 + x_1\varepsilon_1 + \cdots + x_n\varepsilon_n,$$

which is an affine expression (first-degree polynomial) on the noise symbols ε_i with floating point coefficients x_i, in other words, an affine form. Each noise symbol ε_i is a symbolic real variable whose value is unknown and lies in the interval $[-1; +1]$ and is independent from the other noise symbols. The coefficient x_0 is called the *central value* of the affine form $\hat{x}$. The coefficients $x_1, \ldots, x_n$ are called the *partial derivations* associated with the noise symbols $\varepsilon_1, \ldots, \varepsilon_n$ in $\hat{x}$. The number n of noise symbols depends on the affine form: different affine forms can use a different number of noise symbols, some of which may be shared with other affine forms. New noise symbols are created during computation.

Affine forms provide interval bounds for the corresponding quantities: if a quantity x is represented with the form $\hat{x}$ as above, then $x \in [x_0 - r_x, x_0 + r_x]$, where $r_x = |x_1| + \cdots + |x_n|$ is called the *total deviation* of $\hat{x}$. Conversely, if $x \in [a, b]$, then x can be represented with the affine form

$$\hat{x} = x_0 + x_1\varepsilon_1,$$

where $x_0 = (b + a)/2$ and $x_1 = (b - a)/2$. That is, AA algorithms can input and output intervals, and so AA can be used as a replacement for IA. However, affine forms give additional information that can be exploited to further bound the joint range of quantities.

As for interval arithmetic, computations in AA are performed by first extending primitive operations and functions to operate on affine forms, and then combining these primitives to compute arbitrarily complex functions. Given two affine forms:

$$\hat{x} = x_0 + x_1\varepsilon_1 + \cdots + x_n\varepsilon_n$$

$$\hat{y} = y_0 + y_1\varepsilon_1 + \cdots + y_n\varepsilon_n$$

and three real numbers α, β, and γ, then:

$$\alpha\hat{x} + \beta\hat{y} + \gamma = \alpha x_0 + \beta y_0 + \gamma + (\alpha x_1 + \beta y_1)\varepsilon_1 + \cdots + (\alpha x_n + \beta y_n)\varepsilon_n.$$

Extending non-affine operations requires using good affine approximation to the exact result and append an extra term to bound the error of this approximation. Here we consider only multiplication, which is basically only one non-affine operation required to polynomial range analysis. The product of two affine forms is:

$$\hat{x} \cdot \hat{y} = x_0 y_0 + \sum_{i=1}^{n}(x_0 y_i + y_0 x_i)\varepsilon_i + \sum_{i=1}^{n} x_i\varepsilon_i \cdot \sum_{i=1}^{n} y_i\varepsilon_i.$$

So we can write the following affine form for the product:

$$\hat{x} \cdot \hat{y} = x_0 y_0 + \sum_{i=1}^{n} (x_0 y_i + y_0 x_i)\varepsilon_i + z_{n+1}\varepsilon_{n+1},$$

where

$$|z_{n+1}| \leq \left| \sum_{i=1}^{n} x_i \varepsilon_i \cdot \sum_{i=1}^{n} y_i \varepsilon_i \right|$$

is an upper bound for the approximation error. The simplest bound is

$$z_{n+1} = \sum_{i=1}^{n} |x_i| \cdot \sum_{i=1}^{n} |y_i|.$$

which is at most four times the error of the best affine approximation, but is very easily computed. For details on approximation of non-affine operations see [FS04].

As mentioned before, the affine form are more stable against overestimation problem, which has its roots in dependency between variables. To be precise, two or more affine forms can share noise symbols (a noise symbol is *shared* when it appears with a nonzero coefficient in all affine forms). When this happens, the quantities represented by those affine forms are not completely independent: they have a partial dependency for each noise symbol shared by their affine forms.

However, the literature [FS04, MSV$^+$02] says that AA still has an over-conservatism problem.
Example 2.2. *Let $\hat{x} = 0 + \varepsilon_1 + \varepsilon_2$, $\hat{y} = 0 + \varepsilon_1 - \varepsilon_2$. The exact range of $\hat{x} \times \hat{y}$ is ${\varepsilon_1}^2 - {\varepsilon_2}^2 = [-1, 1]$, while AA gives $[-4, 4]$. Besides, AA does not obey the distributive law. For example, in AA, $\hat{x} \times (\hat{y} - \hat{y})$ is zero, however $\hat{x} \times \hat{y} - \hat{x} \times \hat{y}$ is not zero.*

Using AA, one can easily test whether a curve given by $f(x, y) = 0$ crosses a rectangle $[\underline{x}, \overline{x}] \times [\underline{y}, \overline{y}]$. First, we convert the quantities into affine forms: $\hat{x} = x_0 + x_1\varepsilon_x$, $\hat{y} = \underline{y_0} + y_1\varepsilon_y$, then calculate the expression $f(\hat{x}, \hat{y})$. If the computed interval $[\underline{F}, \overline{F}]$ includes 0, then there is possibly a curve branch inside the rectangle. Of course, due to over-conservatism, Affine Arithmetic may err but only in one direction: it may report intersection, although there is no one. But for rectangles small enough this test will succeed, moreover in section 2.3.4 we consider a technique allowing to compute *exact* bounds using AA in a number of cases.

2.3.3 Modified affine arithmetic

Now we consider another method to evaluate a function values over an interval. The method can be seen as an extension of standard Affine Arithmetic. The idea was introduced in [SMW+].

Unlike IA and AA, which do not obey the distributive law, MAA satisfies all the commutative, associative and distributive laws, because it keeps all powers of noise symbols without approximation. In this respect, there is no difference between MAA and real arithmetic.

Here we give an overview only for 1D case as the only relevant case in application to our problem (for 2D case we refer to [SMW+, MSV+02]).

Taylor expansion of a univariate polynomial of degree d results in the following equation:

$$f(x) = f(x_0) + \sum_{i=1}^{d} \frac{f^{(i)}(x_0)}{i!}(x - x_0)^i$$

Now we want to compute $f(x)$ over $\mathbf{x} = [\underline{x}; \overline{x}]$. Thus, we take

$$x_0 = (\underline{x} + \overline{x})/2,\ x_1 = (\overline{x} - \underline{x})/2,\ \text{and } x - x_0 \in x_1[-1, 1].$$

This yields to the following:

$$f(\mathbf{x}) = f(x_0) + \sum_{i=1}^{d} \frac{f^{(i)}(x_0)}{i!} x_1^i [-1, 1]^i.$$

Observe that $[-1, 1]^{2n} = [0, 1]$ and $[-1, 1]^{2n+1} = [-1, 1]$, so our equation simplifies to:

$$f(\mathbf{x}) = f(x_0) + \sum_{i=1}^{\lceil d/2 \rceil} \frac{f^{(2i-1)}(x_0)}{(2i-1)!} x_1^{2i-1}[-1, 1] + \sum_{i=1}^{\lfloor d/2 \rfloor} \frac{f^{(2i)}(x_0)}{(2i)!} x_1^{2i}[0, 1].$$

Now let denote $D_i = f^{(i)}(x_0)x_1^i/i!$, then the boundaries $[\underline{F}; \overline{F}]$ are obtained as follows:

$$\overline{F} = D_0 + \sum_{i=1}^{d} \left\{ \begin{array}{cc} \max(0, D_i), & \text{if } i \text{ is even} \\ |D_i|, & \text{otherwise} \end{array} \right\},$$

$$\underline{F} = D_0 + \sum_{i=1}^{d} \left\{ \begin{array}{cc} \min(0, D_i), & \text{if } i \text{ is even} \\ -|D_i|, & \text{otherwise} \end{array} \right\}.$$

Remarkably, this formula turns out to be more precise than direct application of affine arithmetic to $f(x)$. Since we handle all dependencies between quantities manually. However, the experienced reader may object whether it is worth using complicated interval methods when AA offers quite satisfiable results in most cases which is true to a certain extent. On the other hand, the inaccuracy of AA may require, for instance, more subdivision steps of the algorithm, which nullifies the advantages of AA.

Further development of MAA method can be found in [SMW$^+$05]. The proposed method leads to slight performance increase in 2D case which is, however, negligible in 1D case.

2.3.4 Recursive derivative information

Using the derivative of $f(x, y)$ can provide extra information, which can help to make the determination of the bounds for $f(x, y)$ on $[\underline{x}, \overline{x}] \times [\underline{y}, \overline{y}]$ more precise. The idea is that before evaluating $f(x, y)$ over $[\underline{x}, \overline{x}] \times [\underline{y}, \overline{y}]$ using *any* range analysis method, we first evaluate $\partial f/\partial x$ and $\partial f/\partial y$ over $[\underline{x}, \overline{x}] \times [\underline{y}, \overline{y}]$ using the same range analysis method as used to evaluate f itself. If both resulting derivative intervals do not straddle 0, then f increases or decreases monotonically on going across the interval in x and y. Thus, exact bounds of $f(x, y)$ over $[\underline{x}, \overline{x}] \times [\underline{y}, \overline{y}]$ can be obtained immediately as shown below:

- if $\frac{\partial f}{\partial x} > 0$ and $\frac{\partial f}{\partial y} > 0$, then $\underline{F} = f(\underline{x}, \underline{y})$, $\overline{F} = f(\overline{x}, \overline{y})$;
- if $\frac{\partial f}{\partial x} > 0$ and $\frac{\partial f}{\partial y} < 0$, then $\underline{F} = f(\underline{x}, \overline{y})$, $\overline{F} = f(\overline{x}, \underline{y})$;
- if $\frac{\partial f}{\partial x} < 0$ and $\frac{\partial f}{\partial y} > 0$, then $\underline{F} = f(\overline{x}, \underline{y})$, $\overline{F} = f(\underline{x}, \overline{y})$;
- if $\frac{\partial f}{\partial x} < 0$ and $\frac{\partial f}{\partial y} < 0$, then $\underline{F} = f(\overline{x}, \overline{y})$, $\overline{F} = f(\underline{x}, \underline{y})$.

The same approach can also be used recursively — to get the bounds on $\partial f/\partial x$, one can use its derivatives, i.e., $\partial f^2/\partial x^2$, $\partial f^2/\partial x \partial y$ and so on. The process must terminate whenever a derivative is a constant function.

The recursive derivative methods use not only first derivative but all higher derivative information possible in trying to find the exact bounds of $f(x, y)$ over $[\underline{x}, \overline{x}] \times [\underline{y}, \overline{y}]$, and therefore they are more accurate.

AA equipped with the recursive derivative information offers the best compromise between accuracy and computation efficiency as reported by [MSV$^+$02].

2.4 Counting the number of curve branches

Separation of curve branches (or curve segments) relies on the fact that one is able to compute (or estimate) the number of curve branches crossing a line-segment or passing through a rectangle. This finally leads to the problem of counting the number of roots of a univariate polynomial over an interval.

As it could seem at first sight, this problem is not trivial and there are quite complicated and computationally expensive algorithms to solve it. Here we only survey existing approaches, referring to [Per06] for the complete description. All of them are founded on the *Descartes's Law of Signs* and its generalization known as *Budan-Fourier theorem.* This can be formulated as follows: Let the **number of sign changes**, $V(a)$, in a sequence $a = a_0 \dots a_p$, of elements in $\mathbb{R}\backslash\{0\}$ is defined by induction on p by:

$$V(a_0 \dots a_p) = \left\{ \begin{array}{ll} V(a_1 \dots a_p) + 1 & \text{if } a_0 a_1 < 0 \\ V(a_1 \dots a_p) & \text{otherwise} \end{array} \right\}, \text{ with } V(a_0) = 0,$$

and let P be a univariate polynomial of degree p in $\mathbb{R}[X]$, we denote by $Der(P)$ the list of derivatives $P, P', \dots, P^{(p)}$, and by $n(P; (a, b])$ the number of roots of P in $(a; b]$ counted with multiplicities.

Theorem 2.1. *(Budan-Fourier theorem) Let P be a univariate polynomial of degree p in $\mathbb{R}[X]$. Given a and b in $\mathbb{R} \cup \{-\infty, +\infty\}$*

- $V(Der(P); a, b) \geq n(P; (a, b])$,
- $V(Der(P); a, b) - n(P; (a, b])$ *is even.*

In general it is not possible to conclude much about the number of roots over an interval using only theorem 2.1, this is only a "building block" for complicated algorithms. There are several techniques which give the exact number of roots over an interval via computing:

- signed remainder sequences;
- signed subresultant polynomials;
- signature of a quadratic form.

Although these methods provide the exact number of polynomial roots over an interval, it is too expensive to use them in a real-time visualization. That is why, we base our tests on the result obtained from the Mean Value theorem explained below. These tests at some sense constitute the "core" of our approach. Mainly, we distinguish two cases: the number of intersections of a curve with a line-segment and the number of curve branches passing through a rectangular area.

2.4.1 The Mean Value theorem

The Mean Value Theorem is an important theoretical tool in Calculus.

Theorem 2.2. *If $f(x)$ is defined and continuous on the interval $[a, b]$ and differentiable on (a, b), then there is $c \in (a, b)$ such that*

$$f'(c) = \frac{f(b) - f(a)}{b - a}.$$

In particular, we are interested in a special case of the Mean Value that is also known as Rolle's theorem.

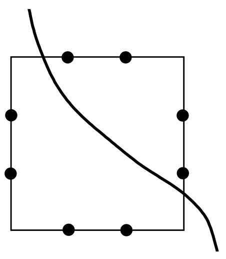


Fig 2.2: A boundary-intersection test

Theorem 2.3. *(Rolle's Theorem) If $f(x)$ is defined and continuous on the interval $[a, b]$ and differentiable on (a, b), and $f(a) = f(b)$, then there exists some c in the interval (a, b) such that $f'(c) = 0$*

Remarkably, this theorem leads to the following result:

Corollary 2.1. *If for a differentiable function $f(x)$ its derivative $f'(x)$ does not straddle 0 in the interval $[a; b]$, then $f(x)$ has at most one root in $[a; b]$.*

In other words, the presence of the first derivative allows to argue about the number of roots of a univariate polynomial over an interval or the number of curve branches crossing a line segment.

2.4.2 Boundary intersection of a curve with a rectangle

We developed two similar techniques to test whether only one curve branch intersects with a rectangle which reflect different stepping schemes considered later. The first technique is based on boundary intersection tests.

Assume we have a rectangular area whose boundary is subdivided evenly into small subsegments (see Figure 2.2). We claim that there is only one curve branch crossing this rectangular area if a line-segment intersection tests presented below succeeds for *exactly* two subsegments on the boundary. Apart from that we do not take into consideration possible presence of closed curve components within the area since this would violate our definition of a sweepable segment, and therefore one must encounter a segment end-point inside this area.

Essentially, it is not always possible to distinguish whether the curve has no intersections or more than one intersection with a line-segment relying on range analysis tests only.[2] Therefore, for each line-segment comprising a rectangle our

[2]Indeed, only real root counting techniques like the ones mentioned in the beginning of ssection 2.4 can give an accurate answer immediately. However, applying them in a real-time for each rasterized pixel would be rather costly. That is why, we decided in favour of range

algorithm outputs either “no intersections”, or “one intersection”, or “possibly more than one intersection”. The overall boundary test succeeds if exactly two subsegments on the boundary have “one intersection” result, and the remaining ones have “no intersections” correspondingly. The last case – “possibly more than one intersection” – indicates that nothing can be said for sure on the current scale, and the area of interest must be refined. Meaning of this step will be clear after presenting the overall approach in chapter 3, now it is important that this case corresponds to the test failure. Following our needs, we will call the procedure “estimation of the number of curve branches” further in our discussion.

Assume we need to estimate the number of intersections of an algebraic curve $C(f)$ with a line segment (horizontal or vertical) $[a; b]$. This is analogous to estimate the number of roots of a polynomial $f(x, y_0)$ or $f(x_0, y)$ where one coordinate is fixed over an interval $[a; b]$. We propose the following approach:

1. calculate the range of values of $f(x)$ over an interval $[a; b]$ using range analysis. If the resulting interval does not contain zero, $f(x)$ has no root in $[a; b]$ and we are done;
2. otherwise we need to estimate the number of possible roots of $f(x)$ counted with multiplicities over this interval. To this end, we evaluate the sings at interval end-points.
    (a) If there is no sign change, the number of roots is even and possibly more than 0, since range analysis gives a zero interval. At this point we are unable to distinguish between “no roots” and “more than one root” cases. In any case, we report “more than one root”, which corresponds to test failure as mentioned above.
    (b) If there is a sign change at end-points, we apply the following recursive procedure:
        i. check whether the first derivative $f'(x)$ straddles 0 on $[a; b]$ using range analysis. If it does not, we have only *one* root of $f(x)$ and our test succeeds (due to the Mean-Value theorem);
        ii. otherwise partition the segment in two halves ($[a; c]$ and $[c; b]$). Observe that $f(x)$ must have a sign change only at one of them, say at $[a; c]$. Select the half where $f(x)$ has no sign change, i.e., $[c; b]$ and test it with range analysis. If range analysis returns a zero interval, again nothing can be said concretely at a moment because we have potentially more than one root along the interval. Therefore, we report “more than one root” and boundary intersection test ends with a failure.
        iii. otherwise, if a computed interval over $[c; b]$ does not contain zero, we apply the complete recursive procedure to the other half $[a; c]$.

analysis equipped with subdivision technique.

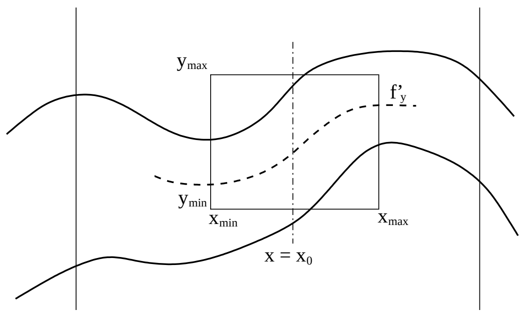


Fig 2.3: Extension of the Mean Value theorem to 2D

Observe that if our intersection test succeeds for two subsegments one has two locations where a curve branch enters and leaves the rectangle indicating that only one curve branch crossing it.

2.4.3 2D intersection of a curve with a rectangle

This is a 2D counterpart to the procedure in section 2.4.2.[3] The intuition behind it is that between two curve branches of $C(f)$ lies a branch of a partial derivative such that the roots of a univariate polynomial, obtained by fixing one of the coordinates of f, are separated by the root of its derivative. In our case we are interested in the first partial derivative w.r.t. y-coordinate as we deal with x-monotone curve branches.

Let us consider a *sweepable area* in Figure 2.3 bounded by two vertical lines. Sweepable area denotes a region in the plane defined by x-coordinates of two consecutive critical points of a curve, in other words, curve segments crossing this area are *x-monotone* and do not have singularities in the interior.[4] The following theorem allows to estimate the number of curve branches in the sweepable area.

Theorem 2.4. *Assume $R = [x_{min}; x_{max}] \times [y_{min}; y_{max}]$ is a rectangle in the sweepable area. Then, there is only one branch of $C(f)$ in R if $C(f_y)$ does not have zeros in this rectangle. The reverse is not true: namely, $C(f_y)$ may have zeros in R but there is only branch of $C(f)$ in R.*

Proof. In Figure 2.3 one can see a rectangle R in the sweepable area and two branches of $C(f)$. If we fix the x-coordinate at some $x_0 \in [x_{min}; x_{max}]$, then,

[3] We call it "2D intersection test" to distinguish from the boundary intersection test.

[4] Note that the sweepable area is not necessarily bounded by vertical lines – if a curve is not closed and has no singular and extreme points, then the sweepable area covers the whole plane.

by the Mean Value theorem, between two roots of $f(x_0, y)$ there must be a root of $f_y(x_0, y)$. We may choose any x-coordinate in the range $[x_{min}; x_{max}]$ and, since the segments are x-monotone, the number of intersections of a vertical line with a curve will remain constant on going across the interval (always two intersections with two branches). In other words, there must be a branch of $C(f_y)$ between two x-monotone branches of $C(f)$.

Now one needs two show that if there are two branches of $C(f)$ in the interior of R, then $C(f_y)$ also goes through the rectangle R. If there is a vertical line at x_0 between x_{min} and x_{max} which intersects simultaneously with both branches of $C(f)$ inside the rectangle then by the Mean Value theorem there is a root of $f_y(x_0, y)$ in the range $[y_{min}; y_{max}]$ and assuming the continuity of $C(f_y)$ we are done. If there is no such a vertical line (see Figure 2.3) we proceed as follows. We know that the branch of the derivative lies in between the curve branches and, hence, the derivative must cross the boundary of R earlier than the branch of $C(f)$ on entering the rectangle. By the same token, on leaving the rectangle the derivative must cross a boundary of R after the branch of $C(f)$. As a result, we have that $C(f_y)$ enters and leaves the rectangle, since it cannot be discontinuous, it also has zeros inside R. $\square$

We developed a procedure to check if only one curve branch of $C(f)$ exists inside a rectangle $[x_{min}; x_{max}] \times [y_{min}; y_{max}]$. As before, we are unable to count the number of branches precisely relying on range analysis, therefore, we distinguish the following cases: "no curve branches inside a rectangle", "one curve branch", and "possibly more than one curve branch". The latter case corresponds to the test failure saying that the number of curve branches cannot be determined on the current "resolution", and the rectangular area must be refined, this will be discussed in chapter 3. Here is the approach:

1. compute $f(x, y)$ over $[x_{min}; x_{max}] \times [y_{min}; y_{max}]$ using the range analysis. If the resulting interval does not contain zero, the curve does not cross this rectangle and we are done;

2. otherwise compute $\partial f/\partial y$ over $[x_{min}; x_{max}] \times [y_{min}; y_{max}]$ again using range analysis. If $C(f_y)$ does not have zeros inside this rectangle, we have only one curve branch inside it and report and our test succeeds.

3. if range analysis returns an interval containing zero for f_y, we potentially have more than one curve branch within the rectangle. In this case, 2D intersection test fails and we report "possibly more than one curve branch over rectangle".

It is important that in practice we use only approach from section 2.4.2. There are several reasons for that: one of them is that one-dimensional range analysis is much faster. We consider this in detail in the next chapter after presenting the overall algorithm.

Chapter 3

Algorithm

This chapter presents the algorithm for visualization of segments and points of algebraic curves. We begin with a short discussion to have an intuitive feeling about the algorithm. Then we present the pseudo-code and consider two different stepping schemes (4- and 8-way) along with technical details. Afterwards, we discuss stopping criteria of our algorithm and plotting isolated points. As possible optimisations we consider a special case of handling coincide branches. Practise has shown that this simple idea can give at least tenfold performance boost in complicated cases.

3.1 Overview

We already mentioned in the introduction that our approach has its roots in curve tracking and space subdivision methods. We start with clear and simple idea introduced by [Cha88]. His algorithm offers the best performance for nicely-behaving curves, i.e., when curve does not have singularities.

We generalized his algorithm as follows: for each pixel, consider its neighbourhood (see Figure 1.3)(a) and check whether *only* one curve branch enters and leaves it.[1] This can be done using techniques given in section 2.4. If there is only one curve branch we choose the next pixel. Otherwise we subdivide the current one into four subpixels, we consider these subpixels with a certain priority explained later and select the one intersected with the curve branch. Then, we again consider a neighbourhood of this subpixel, if there is only one branch inside the area we make a step forward. Otherwise we partition this subpixel recursively into four subpixels and repeat all computations. As we know that a curve segment has no singularities in the interior the recursion must stop at some subdivision level. Thus, the sequence of subpixel should finally bring us

[1]4- and 8-way stepping schemes consider different pixel neighbourhoods however this is an object of later discussion, the overall approach has the same structure for both schemes.

to the next pixel in the approximation.

As the evidence to the correctness of our approach we developed the notion of *witness* subpixel.

Definition 3.1. *A "witness" pixel is such a pixel which intersects only with one curve branch we rasterize.*
We implicitly assign a witness pixel to each pixel in our approximation. Apparently, *the approximation of a curve segment is correct if there is a path of witness pixels from one end-point to another.* Note that a witness pixel does not necessarily have pixel's size, it can amount to an integer fraction of a pixel depending on how close curve branches come to each other at this location.

Our algorithm starts at a seed point, in other words, at a point which definitely lies on the curve branch. Assume the seed point is chosen in the middle of the segment such that at the starting point we have two different directions to follow. We approximate this seed point with a subpixel, small enough such that its boundaries intersect with the given curve branch only, i.e., with a witness pixel. We assign this witness pixel to the starting pixel (a pixel with the seed point in the interior). In most cases they have equal size, meaning that there are no other curve branches surrounding the seed point.

We start with examining the neighbourhood of the starting pixel (see Figure 1.3). If there are only two intersections – only two pixels from the neighbourhood intersect with the curve – we mark one direction as taken and immediately obtain the next pixel. We need to state explicitly that term "intersection" can be understood differently. Various definitions lead to various issues, that is why we do not focus on a concrete mathematical meaning now. Exact statements will be given later in this section. Anyway, regardless the concrete interpretation the overall structure of the algorithm remains unchanged.

Otherwise, if there are more than two intersections, we continue tracking from the *witness* pixel – we cannot subdivide the starting pixel itself, since we are not sure whether only *one* curve branch passes through it. If the neighbourhood test for the witness pixel succeeds we obtain the next subpixel, otherwise it is subdivided into 4 parts. Among these parts, the one intersecting the curve branch is selected. Intrinsically, we can choose anyone which intersects a curve branch, but it is profitable to select the one located closer to the pixel's boundary in the current pixel's direction. This observation will be clarified in the next section, where we consider as an example a part of a curve branch tracked by the algorithm.

If the neighbourhood test for this selected subpixel again fails we subdivide it further, recursively. Assume the neighbourhood test succeeds in this case, then we can uniquely identify the next subpixel. Let us call it subpixel **a**. To determine the next *pixel* in the approximation we test whether **a** lies in the pixel *different* from the one we started tracing from. If it does, we immediately obtain the next pixel. Simultaneously **a** becomes its witness pixel. Otherwise, we continue tracking from **a** until the pixel's boundary is reached. With this approach we finally end up with a sequence of witness pixels which approximates the given curve branch.

Another important issue is how to distinguish incoming and outgoing branches. Indeed, each time the neighbourhood test succeeds we have two curve branches and need to select the one corresponding the current tracking direction. Chandler [Cha88] solved this problem as follows: each time the algorithm makes a step it stores a direction been taken – a number from 0 to 7 in Figure 1.3(a) – then, the opposite direction (i.e., the direction we came from) is calculated as $(d + 4) \bmod 8$ with d being a taken direction. Such that the branch leaving the pixel in direction $(d+4) \bmod 8$ is treated as an incoming branch and, thus, can be discarded. Although this works pretty well in the original algorithm – under assumption that we do not have singularities and closely located branches – in some cases it can determine wrong "backward" direction. We point out these cases later along with the modification to overcome this difficulty.

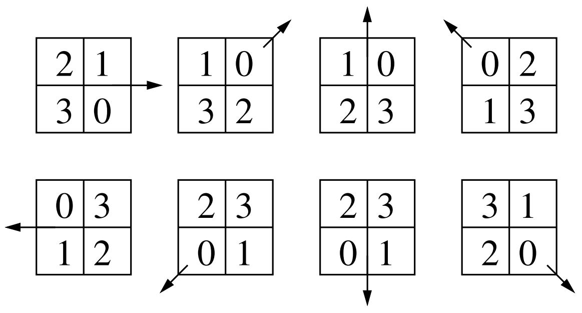


Fig 3.1: A sequence along which subpixels are checked. Arrow shows the backward direction.

3.2 Algorithm details

3.2.1 Pseudo-code

The algorithm's pseudo-code requires the following data types and procedures to be defined:

$\mathsf{Pixel} = \{\mathsf{x}, \mathsf{y}, \mathsf{sub\_x}, \mathsf{sub\_y}, \mathsf{level}\}$: a pixel defined by integer coordinates $(\mathsf{x}; \mathsf{y})$, subpixel coordinates $(\mathsf{sub\_x}, \mathsf{sub\_y})$ relative to the pixel's boundary and subdivision level (0 for pixels, 1 for 1/4 of pixel, 2 for 1/16 of pixel, etc)

Direction : a number $0 \dots 7$ specifying the one out of 8 possible directions in 8-pixel neighbourhood.

function test_neighbourhood(p : Pixel, back_dir : Direction) : tests 4 or 8 surrounding pixels of pixel p (depending on the stepping scheme used), and returns a new direction in case the neighbourhood test succeeds, otherwise returns -1. The direction d is treated as backward direction, that is the direction in which the incoming branch leaves the pixel.

function get_pixel(subp : Pixel) : returns a pixel to which subpixel subp belongs. If subdivision level of subp is 0, the pixel itself is returned.

function get_subpixel(p : Pixel) : subdivides pixel subp into 4 parts and returns the one intersected with the curve branch.

function advance_pixel(p : Pixel, d : Direction) : advances pixel coordinates (for pixels) or subpixel coordinates (for subpixels) in direction d, returns a new pixel.

function subdivide(p : Pixel, d : Direction) : recursively subdivides pixel p into subpixels until the next subpixel can be uniquely determined, returns a subpixel for which neighbourhood test succeeded and a new direction. Notice that this next subpixel is not necessarily a “witness” however intersects with only one curve branch: it becomes “witness” when it lies in a pixel different from the one we came from.

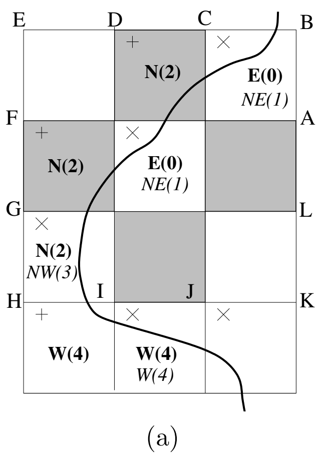
The procedure step below gets the current pixel pix, its witness pixel witness and a direction d and returns the next pixel in the approximation, its witness pixel and a current tracing direction relative to the previous pixel.

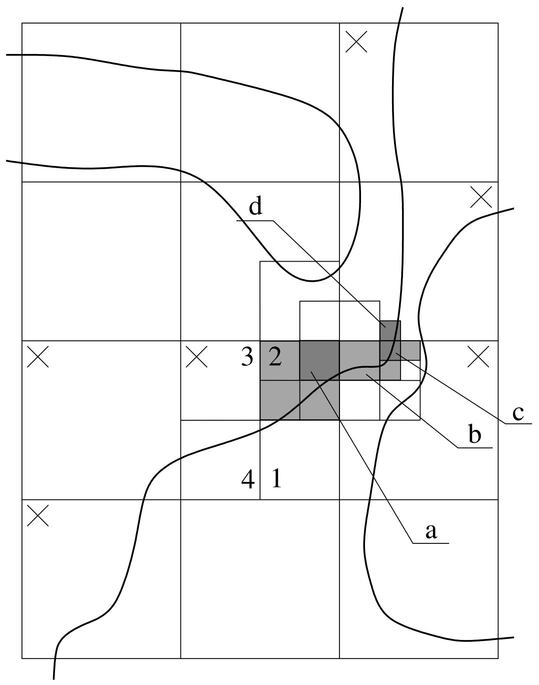
```
procedure step(pix : Pixel, witness : Pixel, d : Direction)
1.   new_d : Direction ← test_neighbourhood(pix, d)
2.   if new_d $\neq -1$ then
3.        p : Pixel ← advance_pixel(pix, new_d)
4.        return {pix, pix, d}
5.   if witness = pix then // if a witness subpixel is a pixel itself we perform subdivision
6.        {p : Pixel, d} ← subdivide(pix, d)
7.   else // otherwise we start tracing from the witness subpixel
8.        p : Pixel ← witness
9.   while get_pixel(p) = pix // iterate until we get a new pixel
10.       new_d ← test_neighbourhood(p, d)
11.       if new_d $= -1$ then
12.            {p, new_d} ← subdivide(p, d)
13.       p ← advance_pixel(p, new_d)
     // returns a pixel, its witness subpixel and a new direction
14.  return {get_pixel(p), p, new_d}
end
```

```
procedure subdivide(p : Pixel, d : Direction)
1.   sub_p : Pixel ← get_subpixel(p, d)
2.   new_d ← test_neighbourhood(sub_p, d)
3.   if new_d $= -1$ then
4.        return subdivide(sub_p, d)
5.   return {sub_p, new_d}
end
```

Remarks. Notice that the procedure test_neighbourhood is essentially the most important one. It takes a pixel and examines its surrounding pixels for the purpose of intersection with a curve using range analysis. Its realization differs for 4- and 8-way stepping schemes depending on the way we treat intersections between a curve and a pixel. We consider both of them.

The procedure get_subpixel selects a subpixel from among 4 possible which is intersected by a curve. The testing of subpixels proceeds along a sequence of most likely possibilities. In particular, the subpixel located near the pixel's boundary in direction opposite to the current direction of motion is checked first. Then, its two nearest neighbours, and at the end the remaining subpixel, see Figure 3.1. The number inside a subpixels shows the order of testing, an arrow defines a backward direction.





(a) (b)

Fig 3.2: Pixel approximation of a curve branch with relative pixel directions (a); and curve tracing with 4-way stepping scheme, plotted pixels are marked with x's (b)

3.2.2 4-way stepping scheme

This scheme means that we do not step in diagonal directions, only horizontal/vertical directions are allowed. Let us consider Figure 3.2(a). From the central-top pixel one can step to dark-colored pixels only. We mark pixels plotted by using 4-way stepping scheme with +'s, bold-face letters denote relative pixel directions. Pixels plotted by using both schemes are marked with x's, and italic letters denote pixel directions in case of 8-way stepping scheme.

Remark that the stepping schemes differ in how we count pixel-curve intersections. In particular, in case of 4-way scheme we use the following definition of a pixel-curve intersection: *a curve intersects with a pixel if there is a curve branch in the interior of this pixel.*

At first glance it sounds trivial but in section 3.2.3 we will see the difference after introducing the definition of intersection for 8-way stepping scheme. Now we consider the application of 4-way stepping scheme on the example. Let us take look at Figure 3.2)(b).[2] Assume we came from the bottom-left pixel. From this pixel the algorithm selects a north pixel as the only one intersecting by the curve branch. At this pixel the neighbourhood test again succeeds and we step in the east direction.

While testing the neighbourhood of the next pixel, we find out that two of its neighbours intersect the curve. Thus, the pixel is subdivided into 4 parts (1, 2, 3 and 4 in the figure). We further consider one of them intersecting the curve branch (2 in the figure). The subpixels are verified in the order as depicted in Figure 3.1 depending on the backward direction.

We continue curve-tracking from the subpixel "2" in the figure. The neighbourhood test for it fails as we have more than 2 intersections, this leads to another subdivision step. Finally, in the last subdivision step we find out that only one subpixel intersects with the curve branch (the one denoted by **b** in the figure). This subpixel lies in the area enclosed by another pixel, and, hence, the next pixel is determined. Simultaneously, **b** becomes its "witness" pixel.

Next iteration of the algorithm starts from this pixel. Again, several subdivision steps are required, until the neighbourhood of the subpixel denoted by **b** allows to obtain the next subpixel **c**. Repeating this procedure, we end up with a subpixel **d**, which lies in the next pixel. As before, **d** becomes its "witness" pixel.

Repeating these steps we finally result in a path consisting of "witness" pixels approximating the curve branch.

3.2.3 8-way stepping scheme

In this case we can step in any of 8 possible directions as depicted in Figure 1.3)(a). Besides that, we exploit another definition of curve-segment intersection: *a curve intersects with a pixel if its "exterior" boundaries are intersected by the curve.*

For instance, in Figure 1.3(a) segments AB and BC are "exterior" boundaries for pixel NE, segment CD — for pixel N, segments DE and EF — for pixel NW. In this case a pixel neighbourhood consists of 8 pixel (counting diagonal directions).

Assume as before we came from the bottom-left pixel – see Figure 3.3. Then, we select the *north-east* pixel, as the only one crossed by the curve.

[2]Curve branches depicted in the figure certainly do not hit our definition of a curve segment given in section 2.2, but since our algorithm can deal with more general segments we consider this as an example without loss of generality.

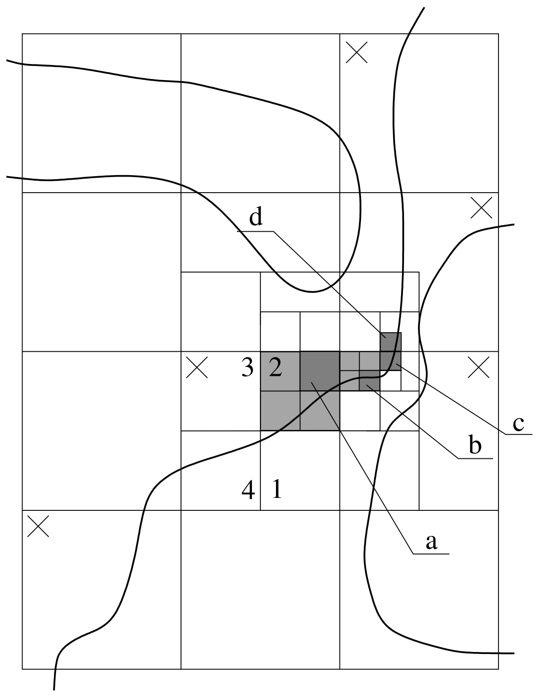


Fig 3.3: Curve tracing with 8-way stepping scheme, plotted pixels are marked with x's

The neighbourhood of this pixel does not allow to determine the next pixel uniquely, that is why we subdivide it into 4 parts (1, 2, 3 and 4 in the figure). We select the subpixel marked with "2" and proceed by testing its neighbourhood. The subdivision level is again not enough to decide unambiguously which pixel is the next one. In the next subdivision step we select the subpixel denoted by **a** in the figure and successfully determine the next subpixel. This next subpixel lies in the interior of another pixel, thus the *east* pixel is determined. At the same time, this subpixel is saved as its "witness" pixel.

Next iteration we start from the east pixel. Since there are several curve branches passing through its neighbourhood, we have to go down to subpixel level again. But unlike for the previous pixel, now we have a certain "witness" pixel, therefore we continue by considering its neighbourhood. The subdivision level is again too small. Thus, we subdivide the "witness" pixel into 4 parts and choose the one denoted by **b** in the figure. From there we step to the subpixel **c**, since it is only one crossed by the curve branch.

In the next algorithm step we obtain a subpixel **d** which in its turn belongs to the *north* pixel, therefore the next pixel is successfully determined. **d** is stored as its “witness” pixel.

As a result, the reader may observe slightly different behaviour of two schemes on the same example. In the next section we give technical details in realization of these two schemes and point out advantages and drawbacks of both.

3.2.4 Plotting isolated points and vertical segments

[3]As mentioned in the introduction, an important feature of implicit curves, which makes the visualization of them more complicated, refers to the presence of *isolated* points. A simplest equation defining an isolated point as intersection of two imaginary lines is: $x^2 + y^2 = 0$.

Technically, an isolated point is a singular point – since a gradient ∇f vanishes in it – and in our case is represented as a degenerate segment whose end-points coincide.

To draw such a segment we look for an appropriate y-coordinate of the point. For this purpose, we use a real root isolation algorithm (for details see [EKK+05, RZ04]) on a given x-interval to obtain a set of ordered y-coordinates. Among them we select the one corresponding to a certain arc-number. As a result, a pair sufficiently refined intervals in x and y direction approximates an isolated point.

Processing vertical segments is a similar to the technique explained above: indeed, a vertical segment is uniquely determined by its end-points. Therefore, we compute isolating rectangles for segment’s end-points and connect them by a straight line.

3.2.5 Choosing the “seed” point and stopping criteria

All curve-tracking algorithms require a starting (seed) point which lies on the curve. There are a lot of methods how to find such a point: some algorithms use subdivision, another apply heuristics such as Monte-Carlo method, or require user interaction and so on. In our case a starting point has to be determined precisely since our goal is to visualize a distinct curve segment.

Remember that a curve segment is defined by two end-points and an arc number which is constant in the interior of the segment. X-coordinates of the end-points are specified by isolating intervals whereas y-coordinates are given implicitly by

[3]Note that the information on various kinds of degeneracies of curve segments is supposed to be provided as an input to the algorithm, referring to the current implementation this topic is highlighted in section 4.1.2

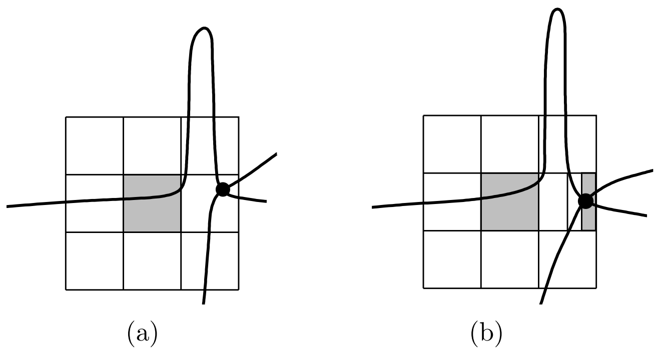


Fig 3.4: Stopping criterion (a); and the advanced version (b)

arc numbers.

We suggest the following method to compute a seed point: pick up some x-coordinate $\mathbf{x}$ within the range between two segment end-points, isolate roots of a univariate polynomial at this x-coordinate, order the set of computed y-intervals in increasing order, and select the one interval which corresponds to the segment's arc number. Afterwards, a pixel with x-coordinate $\mathbf{x}$ and y-coordinate lying on the boundary of y-isolating interval is refined (subdivided) to be intersected only with one curve branch we rasterize.

As a result we have an isolating rectangle (subpixel) which is guaranteed to intersect with exactly one curve branch and contains a desired seed point. Then, we check the neighbourhood of a pixel containing this seed point. If the test returns two distinct directions, the starting "witness" pixel is found, otherwise we perform the neighbourhood test recursively on the subpixel level. From the "witness" pixel one can move to any out of two distinct directions, storing the remaining one beforehand.

Another important question concerns stopping criteria. At first glance stopping criteria look rather simple: tracing terminates once a pixel containing an end-point is met in the neighbourhood of the current pixel. Pixels containing the end-points are again computed by means of real root isolation (for details see [EKK$^+$05, RZ04]). Recall, that x-coordinates of the end-points and the segment's arc number are enough to compute isolating rectangles enclosing the end-points.

However, this is not the solution for all cases. In Figure 3.4(a) one can see the situation where one of the neighbouring pixels of the current pixel (dark-colored) contains a singular point. In such a case our stopping criteria will alarm and the further tracing in that direction will stop, nevertheless the curve contains a high-curvature undulation near the end-point which is not rasterized. We developed the ideas allowing to recognise a number of such situations. Since the curve segments we render are *x-monotone*, we are guaranteed to be away from high-curvature regions if the algorithm has reached a pixel (or subpixel) enclosed in a rectangle containing the end-point whose bottom/top boundaries do not have intersections with the curve – in Figure 3.4(b) this rectangle is dark-colored. To find such a rectangle, we start by testing the horizontal sides of the end-point pixel with range analysis. If at least one test succeeds, the pixel is divided in two parts and the part containing the end-point is tested again. Recursion stops when the rectangle is small enough such that range analysis tests fail for both of its horizontal sides.

This approach seems to be good enough to handle a number of problematic cases, but if a curve has almost vertical slope near the end-point, horizontal sides of such an "isolating" rectangle would be miserably small and the algorithm, therefore, would have to perform a great number of subdivision steps to reach a subpixel lying inside this rectangle. That is why, as a practical solution we suggest to restrict the number of rectangle subdivisions by the maximal subdivision level dependent on the precision of arithmetic currently used (for detailed information on the maximal subdivision level refer to section 4.2.2).

From the conservative point of view, we sacrifice completeness of our algorithm in that way. However, when the visualization of high-curvature "peaks" constitutes a problem at certain fixed resolution, scaling the region of interest exposes previously hidden details – in any case the knowledge of exact curve topology guarantees that curve undulations whose sizes are comparable with the pixel size cannot be overlooked. From this position the problem looks somewhat strained: indeed, the curve peaks of one pixel width does not carry too much information about a curve segment.

3.3 Realization features

The two considered schemes substantially differ in realization of the procedure test_neighbourhood. With the help of mathematical tools described in section 2, we can now formulate the sequence steps required by this procedure.

3.3.1 4-way stepping scheme

In this case all tests to refine pixels rely on the technique explained in section 2.4.3. Let us again consider Figure 3.2(a). Our 4-pixel neighbourhood comprises only dark-colored pixels in the figure. Assume we are in the central pixel marked with direction **E**. Steps of the procedure test_neighbourhood:

1. check all 4 neighbouring pixels for intersecting with a curve using range analysis. If more than two of them intersect with a curve, we require sub-

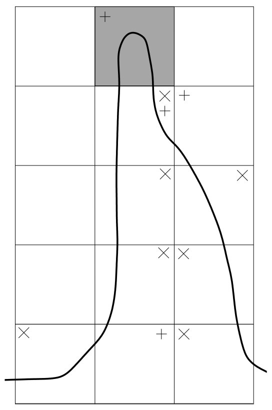


Fig 3.5: Behaviour of schemes nearby a high-curvature point, x's – pixels plotted by both schemes, +'s – by 4-way stepping scheme only

division. Indeed, there should at most 2 intersections – one for incoming branch and one for outgoing branch – otherwise the next pixel cannot be determined uniquely. Assume we have found two pixels – the northern and the southern one in the figure. We proceed as follows:

2. drop the pixel corresponding to backward direction.[4] As a result we are left with the upper-center pixel (in Figure 3.2)(a). Here a difficulty shows up: let us consider Figure 3.5. Assume we are in the upper-center pixel containing a high-curvature point and are considering its neighbourhood. Thus, by simply dropping the pixel we came from one may find oneself in a situation where no next pixel is determined. That is why, we need to keep track of all such suspicious situations and perform subdivision to remove all doubts.
3. obtain the number of curve branches crossing the remaining pixel. We use the technique from section 2.4.3 to ensure that only one curve branch passes through the pixel. In case of ambiguity subdivide the pixel into 4 parts and test each of them separately.

The way how this scheme works can be understood via the notion of *tunnelling*. If one conceives of variable-width tunnels – consisting of "witness" pixels – surrounding each curve branch, which are sufficiently narrow, such that the tunnels of two closely passing branches do not intersect. Such "tunnels" separate curve branches from one another and, therefore, allow to render them separately. The approach given above is robust in the sense that all pixels intersected with a curve branch are plotted, but in some cases this conservatism leads to redundant

[4]Recall that backward direction is defined by $(d + 4) \bmod 8$ where d is the current direction of motion. We number directions as: *east*(0), *north*(2), *west*(4) and *south*(6), see Figure 1.3(a). Although in 4-way stepping scheme we use only 4 directions, we left the numbering unchanged, in order to elude the confusion.

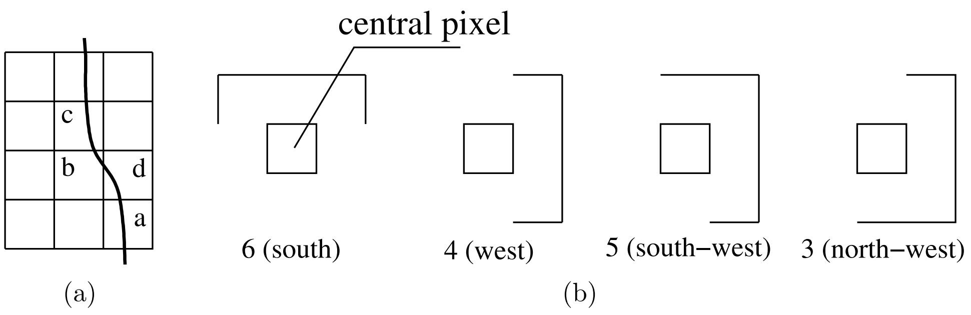


Fig 3.6: Wrong backward direction (a); and fragments of the neighbourhood frontier checked by range analysis depending on the backward direction (b)

subdivision steps and plotting some pixels more than once.

3.3.2 8-way stepping scheme

The ability to step in diagonal directions imposes additional difficulties. In particular, in some cases backward direction may be determined incorrectly causing the algorithm to suspend or even to crush.

For instance, let us examine the situation depicted in Figure 3.6(a). Suppose we came from the pixel denoted by **a**. The next pixel found by the algorithm is **b**, and, then, pixel **c**. At the pixel **c** we determine backward direction by formula $(d + 4)$ mod 8 where $d = 2$ (north) as we came from the pixel **b**. The formula gives us direction 6 (south) pointing to the pixel **b**. Whereas the right backward direction is 7 (south-east), as curve intersects the frontier of pixel **d**!! Therefore no curve branch can be discarded leaving us in a situation where more than one pixel intersects with the curve. This will cause redundant subdivision steps.

To overcome this difficulty we admit that a curve branch may pass not necessarily through the pixel pointed to by the backward direction but also through its neighbouring pixels. That is why, for every backward direction we factor out certain parts of the frontier as shown in Figure 3.6(b), and only the remaining parts are tested by range analysis.

Now we can formulate steps of the procedure test_neighbourhood for 8-way stepping scheme. Assume we are also in the central pixel with NE direction in Figure 3.2(a). As we came from the south-west the backward direction is 5 (south-west). Steps of the procedure test_neighbourhood:

1. discard a certain part of the frontier depending on the current backward direction and test the remaining one by means of range analysis – in our example we check only line segments DB, BK, and KJ. If more than one

intersection found, we possibly have more than one curve branch in the neighbourhood and have to proceed with subdivision. Indeed, there must be *only* one intersection – as we already discarded the incoming branch – otherwise the next pixel couldn't be determined and we need to subdivide. In Figure 3.2(a) only the segment DB intersects with the curve.

2. split up the obtained segment into smaller subsegments corresponding to pixel directions, and check them separately (subsegments DC and CB in our case). By the same argument, there must be only one subsegment crossed by the curve. In our case this is a segment CB. Now we need to ensure that indeed only *one* curve branch intersects the selected segment.
3. to compute the number of curve branches crossing the segment we use a technique based on first-derivative tests from section 2.4.2. If the test succeeds we return a new direction, otherwise we proceed by subdivision.

The algorithm given above requires some remarks.

Intrinsically, a pixel direction can be conceived of as though we save a previous pixel each time we perform the algorithm step, but this previous pixel is given in "local" coordinates relative to its neighbouring pixels. Essentially, direction is the only way to exploit the continuity of curve branches.

Looking at Figure 3.6(b) one may think that by cutting off the frontier the algorithm can overlook some curve branches and, thus, be misled. For instance, when a curve enters and leaves pixel's neighbourhood within one segment. However, this cannot happen since having two close curve branches would cause subdivisions earlier during tracing of previous pixel(s). Indeed, if a curve has an x- or y-extreme point (we consider general segments here) nearby the current pixel such that directions of incoming and outgoing branches differ rather little, because of continuity of the curve more than one branch must be detected in the previous neighbourhood tests, and therefore we approach this current pixel at subpixel scale.

3.3.3 Comparison of schemes

In this section we will try to summarize all advantages and drawbacks of both schemes.

First of all, with 4-way stepping scheme we trace a curve a bit slower, because steps in diagonal directions are forbidden. On the other hand, we use 2D range analysis with this scheme, which gives some advantages. Namely, in Figure 3.5 with 4-way stepping scheme all pixels crossed by the curve are plotted, whereas 8-way stepping scheme misses some pixels. This is because we use 1D range analysis (since 2D range analysis do not allow to step in diagonal directions) and test only the boundaries with 8-way scheme.

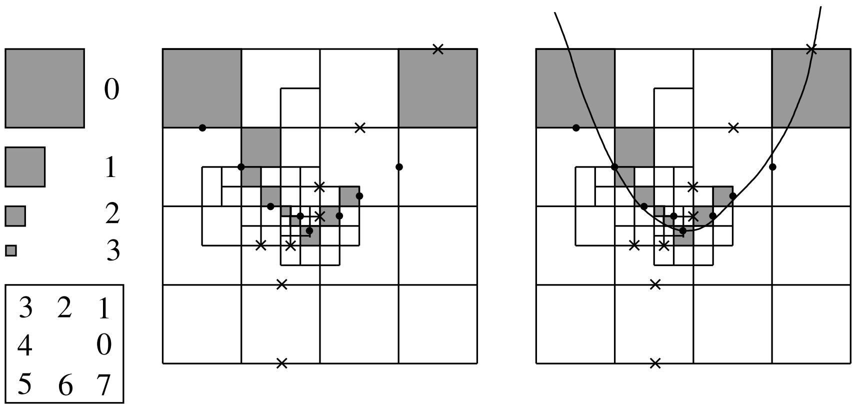


Fig 3.7: Tracing of a "flattened" parabola

1D range analysis is much faster than 2D which together with faster curve tracing gives 8-way scheme an advantage over 4-way scheme. Moreover, the size of an area which can be left out by 8-way scheme is at most 2×2 pixels. Therefore, in a picture taken with resolution 800×600 the absence of such tiny structures is almost unnoticeable. However, this is a theoretical observation: the tests have shown that nearby high-curvature points range analysis tends to over-estimate results quite considerably and such regions are subdivided with high probability.

As an example see Figure 3.7 where several steps of 8-way adaptive subdivision while rendering a "flattened" parabola are reconstructed from the log file. In the figure dots define registered intersection resulting in successful steps whereas crosses mean "redundant" or incorrect intersections which lead to subdivisions. One can observe that due to smaller steps all pixels intersecting with the curve are rasterized.

That is why, we would suggest using 8-way stepping scheme as it is more flexible and requires only 1D range analysis, which is fast and quite easy to implement.

3.3.4 Round-off arithmetic errors and possible threats

Unlike the ideal case, real computations often use approximated quantities which result in round-off errors. For exact algorithms this is a formidable problem, since inexactness of computations may lead to algorithm's hang-up or faulty exit. Situations where round-off errors may have a serious impact on the algorithm's execution are often referred to as "degenerate" cases. In this section we outline such situations and possible ways to avoid them. We do not draw a special attention to a certain stepping scheme. The rules given in this section are common for both schemes.

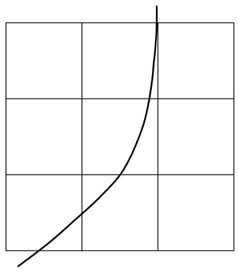


Fig 3.8: A situation where the next pixel may be determined incorrectly due to round-off errors

Remark that, for inexact arithmetic we represent polynomial coefficients by intervals, this ensures that our approximation always encompasses the "true" result. Evaluation of such a polynomial at a point is also an interval.

Let us consider Figure 3.8. Here the curve branch passes very close to pixel's boundary, so it might happen that because of round-off errors an incorrect pixel will be chosen. For example, if one uses double-precision arithmetic and evaluation of a polynomial at pixel's boundary near the curve branch returns an interval containing zero, the sign of such an evaluation we cannot trust. Indeed, an interval including zero is the result of round-off errors, 10^{-17} is a limit for double-precision and when the computed quantity is comparable with this limit further computations with it are unreliable. To prevent this, we keep track of all polynomial evaluations, and, whenever the precision limit is exceeded, trigger the flag indicating possible imprecise result. In this case, the same quantity is evaluated once again using modular arithmetic to test for exact zero. Again, if the quantity is not evaluated to zero, it is recomputed with exact rational arithmetic.

Another challenge occurs when the curve branch goes directly through a pixel's boundary. Exact arithmetic returns zero in this case and we need to decide which pixel to choose. In fact, we are free to select anyone out of two, but in order to exclude ambiguous situations we stick to the following rule: "for horizontal segments select the left one, for vertical – the lower one". This method has its roots in *symbolic perturbation* technique, for details please refer to [MN00].

Although the cases considered here are the most important ones, we are still not free from round-off errors. It can happen that double-precision arithmetic is not sufficient to visualize a curve branch, therefore we suggest to use a three-level scheme with gradually increasing arithmetic precision. At the first level we use double arithmetic and store polynomial coefficients as double intervals,

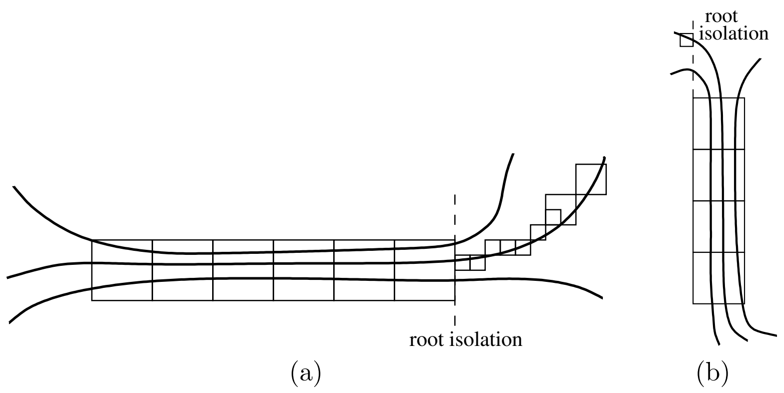


Fig 3.9: Rendering of "coincide" branches, horizontal (a) and vertical (b) cases

at the second level we use multi-precision arithmetic for all computations and store polynomial coefficients as single multi-precision floating-points. At the top level – only exact rational computations are involved.

3.4 Optimization

Practical experiments have shown that the largest amount of time – more than 90% – the algorithm spends in tracing closely located curve branches. When subdivision level reaches 12[5] or more, the computational effort becomes large to an extent that it nullifies other advantages of the algorithm and implementation, such as 1D range analysis with recursive derivative information, adaptive pixel size or caching of precomputed data. It can be understood, since at level 12 one has to trace at most $2^{12} = 4096$ subpixels to reach the pixel's boundary, i.e., to find the next pixel. As a result, *the whole matter boils down to the fact that we spend more than 90% of time in plotting less than 10% of all rasterized pixels.*

Fortunately, in most cases this difficulty can be avoided. We present a technique to suppress high subdivision levels, which in complicated cases gives a significant performance boost and, simultaneously, allows us to render most of algebraic curves using only *double* arithmetic.

3.4.1 Dealing with visibly coincide branches

Let us look at Figure 3.9(a). There is a part of a curve where three curve branches come very close to each other, such that all of them share the same

[5]A subdivision level defines how many times a pixel was subdivided. For instance, $level = 0$ defines an entire pixel, $level = 1$ – 1/4 of pixel, $level = n$ means that a pixel consists of 4^n parts.

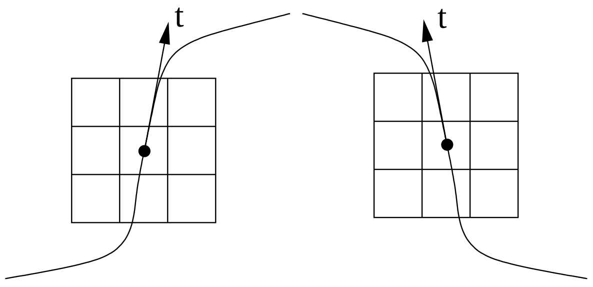


Fig 3.10: Towards computing direction of motion

pixels within a small region (or strip). We call such branches *coincide* branches, because they visibly coincide in the resulting approximation.[6]

Assume we want to draw a middle segment. Our approach will start at a seed point and before entering the strip will make the subdivision level large enough to build up a small subpixel "tunnel" around the middle branch, whose boundaries are not crossed by the curve. The "tunnel" will lead us in the correct direction not allowing to step aside. However, it's clear that subpixeling does not improve the picture within this strip any further, since the pixels comprising it are shared by all three branches at the same time. Moreover, if we want to draw the remaining two segments, we will have to repeat all expensive computations, whereas plotted pixels are *the same.*

That is why, we suggest *to not discriminate branches any further once a tracing direction can be determined uniquely.* Indeed, in Figure 3.9(a) one can see that the direction of motion can be computed, despite the fact that there are more than one branch in the neighbourhood for all pixels comprising the strip. As soon as we detected such a situation we switch on the flag indicating the tracing of "coincide" branches. Later, at location where these branches grow apart or another branch comes closer, in other words, in all such situations where the next pixel cannot be determined uniquely, we have nothing to do but computing a new "seed" point. Really, allowing branches to coincide we, thereby, lose the information about the true arrangement of curve branches, i.e., plotted pixels are not "witness" pixels anymore. That is why, a real root isolation is the only way to "catch" the required branch on leaving the strip.

Although this technique looks nice and easy to implement, account must be taken of different directions along which branches coincide. Let us consider Figure 3.9(b). Here branches coincide in a vertical direction, this imposes addi-

[6]to remove any confusions: we use term "coincide branches" to accentuate the fact that at some locations several curve branches may occupy the same pixels in a resulting approximation, however, we do not mean that several branches really have common zero sets.

tional difficulties in determining a new seed point by exiting a strip of coincide branches. Point is that we have to know toward which end-point we are currently moving in order to obtain an x-coordinate of the new seed point. The end-point we approach can only be determined after we stepped in left or right direction[7], since a curve segment is *x-monotone*. However, if we move in a vertical direction all the time, it is absolutely unclear which end-point we approach because a curve segment may be not y-monotone. In the next section we discuss how the direction of motion can be computed.

Observe that, if a curve branch is almost vertical on coming out the strip of coincide branches one may result in a *gap* between the last pixel of the strip and a pixel containing the new seed point. But, as a curve segment is x-monotone we can close the gap by drawing a line segment between these points. To exploit this idea for general non x-monotone segments one needs to find additional intricate criteria to continue tracing after leaving "coincide branches mode" what is a tough task.

3.4.2 Computing direction of motion

In most cases direction of motion can be determined immediately at the starting pixel: indeed, "left" directions point to "lower" end-point whereas "right" directions lead to "upper" end-point. However, if two vertical directions encountered at the starting pixel, determine direction of motion turns out to be non-trivial. In this case we suggest a method based on computations of a tangent vector.

Assume there is a point $\mathbf{p}$ on the curve and the neighbourhood test for a pixel containing the point reports two vertical directions (see Figure 3.10). If a curve segment is x-monotone, the tangent vector at $\mathbf{p}$ must be directed slightly to the left or to the right depending on the curve slope. In particular, we evaluate $\partial f/\partial x$ and $\partial f/\partial y$ at the starting point $\mathbf{p}$, the corresponding tangent vector is as follows:

$$t = (t_x, t_y) = (-\partial f/\partial y, \partial f/\partial x).$$

If $t_y < 0$ the vector is reversed, that way one ensures that t always "looks" upwards. Then, we use the following rule: if $t_x > 0$, upward direction leads to the right end-point and downward direction – to the left end-point. Otherwise, upward direction leads to the left end-point and downward direction – to the right end-point. Note that the case $t_x = 0$ corresponds to the vertical segment and, therefore, cannot happen.[8]

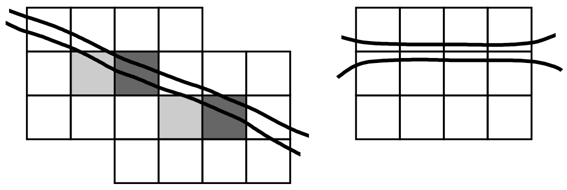


Fig 3.11: Cases where coincide branches technique does not work properly

3.4.3 Caveats

Note that we need to warn against possible cases where this technique does not work and can even slow down the performance. In Figure 3.11 to the left one can see the case where two curve branches go in a diagonal direction close to each other. Since a pixel grid is fixed, we encounter the case when two branches are treated as coincide at one pixel and not coincide in the next pixel, and so on. Namely, in light-colored pixels branches coincide, whereas in dark-colored – do not. Such an alternate behaviour will cause a real root isolation at every second pixel which is equivalent to rendering a curve segment pixel-by-pixel using a straightforward approach given in Introduction.

Another problematic case is shown in the figure to the right. Here again because of rigidness of a pixel grid two curve branches can lie on different sides of pixel's boundary. That is, in spite of close location of the curve branches we cannot apply our technique, and, therefore, need to subdivide.

Although these problems are in general unavoidable, we can reduce their probability by forbidding tracing of coincide branches until a certain subdivision level is reached. In other words, we trace a curve branch with normal subdivision method while a subdivision level is sufficiently small, in that way decreasing the probability of the above situations. Our tests have shown that switching to tracing coincide branches at subdivision level 5 offers rather stable performance not leading to "blow-up" of the number of auxiliary seed points.

3.5 Correctness proof

In this section we try to summarize the features of our algorithm to make the reader convinced that our algorithm indeed works.

[7] according to our numbering the set of "left" directions consists of directions 3 (north-west), 4 (west) and 5 (south-west); whereas "right" directions are: 1 (north-east), 0 (east) and 7 (south-east).

[8] Rasterization of vertical segments is discussed in section 3.2.4.

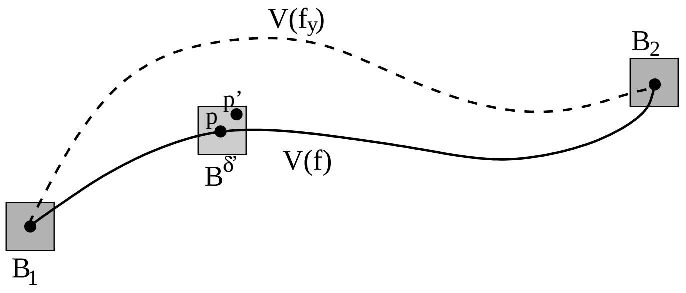


Fig 3.12: Towards showing the correctness of the algorithm

We start with a notion of “witness” pixel. Recall that such a pixel intersects only with a curve branch we want to rasterize. This is achieved by testing the boundaries of its 8-pixel neighbourhood (we consider this proof only in application to 8-way stepping scheme) by range analysis. We use the consequence of the Mean-Value theorem from section 2.4.1 to ensure that indeed only one curve branch passes through a “witness” pixel.

Our goal is to build up a sequence of such “witness” pixels connecting two end-points of a segment. Note that the “continuity” of our sequence is stated by definition: in case of 8-way stepping scheme adjacent pixels share either a common side or a common vertex.

Now we will show that a sequence of “witness” pixels can be computed in a number of steps limited by a global parameter dependent on the curve topology only. In other words, we mean that the number of subdivision steps required to determine the next pixel is always limited. To show this we take $C = V(f)$ (vanishing locus of f) excluding parts covered by open boxes B_1 and B_2 around segment end-points, since at non-singular end-points the derivative f_y intersects with a function (see Figure 3.12):

$$C' = C \setminus B_1 \cup B_2,$$

therefore $\exists\ \delta-$neighbourhood $U(C')$ of C' with respect to $\|\cdot\|_\infty$, such that :

$$V(f_y) \cap \overline{U(C')} = \emptyset, \text{ the set } \overline{U(C')} \text{ is compact, hence :}$$

$$\exists\ \varepsilon > 0 : \inf_{\overline{U(C')}} |f_y| = \varepsilon.$$

Now let us consider a box $B_p^{\delta'}$ with a centre $p \in C'$ where $\delta' < \delta$, Figure 3.12. Note that,

$$B_p^{\delta'} \subset \overline{U(C')}, \text{ and therefore } \min_{B_p^{\delta'}} |f_y| \geq \varepsilon.$$

Our goal is to estimate a range of f_y computed by range analysis over $B_p^{\delta'}$.[9] There exists a fixed M, such that for a small enough δ' the computed range is as follows:

$$[f_y(p) - M \cdot \delta'; f_y(p) + M \cdot \delta'],$$

where M is determined by the polynomial f and defining area of f: for example, let us estimate a range of values of a univariate polynomial g over a small interval $\mathbf{x} = [\underline{x}; \overline{x}]$. We define $g = \sum_{i=0}^{n} a_i x^i$, and $x_1 = \overline{x} - \underline{x}$, then:

$$g(\mathbf{x}) = a_0 + \mathbf{x}[a_n \mathbf{x}^{n-1} + \cdots + a_1] \subseteq a_0 + \mathbf{x} \cdot \sum_{i=1}^{n} |a_i|, \text{ if } \mathbf{x} \text{ is small enough.}$$

This implies that:

$$g(\mathbf{x}) \subseteq [a_0 - x_1 \cdot M; a_0 + x_1 \cdot M], \text{ where } M = \sum_{i=1}^{n} |a_i|.$$

Now let go back to the proof: assume δ' is small enough such that $M \cdot \delta' < \varepsilon$, therefore computed range of f_y over $B_p^{\delta'}$ does not include 0. We claim that the subpixel size cannot become smaller than $B_p^{\delta'}$. Really, the neighbourhood test must succeed at this subdivision level because range analysis will return a non-zero interval for f_y, since our choice of δ' takes into account the maximal possible overestimation of range analysis methods. Therefore, on subpixel grid of size δ' only "true" intersections are registered, that is we can step to the next subpixel.

Another important issue concerns termination of the algorithm. Assume we rasterize the curve branch on a grid of size δ'. With the above argument we have shown that this is the smallest possible grid size. We need to show that the steps in backward direction never take place which otherwise could possibly lead to algorithm's hang-up. This is ensured by an appropriate selection of boundaries to be checked during the neighbourhood test. In Figure 3.6(b) one can see that boundaries corresponding to backward directions are never tested. Moreover, for an x-monotone segment (unless segment is vertical[10]) we can always determine the current tracing direction – towards left or right end-point. Therefore, in case of tracing toward the left end-point we prohibit to follow any "right" directions such as $SE(7)$, $E(0)$ and $NE(1)$; respectively while tracing toward the right end-point we prohibit taking any "left" directions – $NW(3)$, $W(4)$ and $SW(5)$.

Now we consider intersection points of a curve segment with a grid of size δ'. Remark that we take only those points which result from intersection of a neighbourhood frontier of a subpixel on the grid with the curve segment in a direction

[9]We assume *exact* range analysis over $\mathbb{Q}$ throughout the proof in order to stay away from arithmetic precision errors. As mentioned in section 3.3.4, depending on the current subdivision level, first we switch to multi-precision, and then to rational arithmetic.

[10]Visualization of vertical segments is discussed in section 3.2.4.

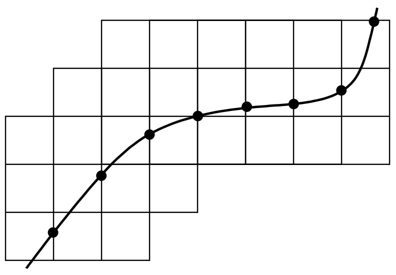


Fig 3.13: Intersection “leaving” points

of motion along the segment, in other words we consider only “leaving” points, see Figure 3.13. The number of such intersection points with one grid-line is finite[11] and limited by the degree of f. Now observe that the number of grid-lines is also finite as we cannot go further than a certain subdivision level, and the intersection points form a strictly monotone sequence with respect to x-coordinate (since curve segments are x-monotone), therefore we must reach a segment end-point in a finite number of steps.

To complete our proof one should also pay attention to various “degenerate” situations where inexactness of arithmetic may lead to the wrong decision, we refer to section 3.3.4 where this problem is discussed in detail.

[11]With the exception, when a horizontal or vertical curve segment coincides with a grid line, however in this case reachability of a segment end-point is beyond question.

Chapter 4

Implementation

4.1 Overview of the Exacus

The EXACUS project is being developed in the Max-Planck-Institute für Informatik in Saarbrücken, it is part of the European Union's Effective Computational Geometry project and ACS project.[1] EXACUS[2] is a set of software libraries for Efficient and Exact Algorithms for Curves and Surfaces. The libraries are designed according to the generic programming paradigm with C++ templates, as it is widely known from the *Standard Template Library*, STL. EXACUS inherited some ideas originated in CGAL.[3] Currently the work is underway to port all EXACUS libraries to CGAL.

EXACUS uses several external libraries: BOOST (interval arithmetic), GMP and CORE (number types), NTL (modular gcd computation), LEDA (number types, graphs, other data structures, and graphical user interface), CGAL (number types and arrangements) and Qt (graphical user interface). At the current state of development EXACUS consists of the following libraries:

- **Library Support** (LiS) is a foundation layer, it offers basic services such as configuration, assertion, file IO (`LiS/file_io.h`), memory management, handle template (`LiS/Handle.h`), generic algorithms, and other low level support for software libraries.
- **NumeriX** (NiX) comprises number types, algebraic constructions to build types from types, such as polynomials (over a number type), vectors, and matrices, computing determinants, gcds, resultants and subresultants, isolating real roots of polynomials, provides a support for real algebraic numbers, and more. As this library offers basic functionality for management

[1]http://acs.cs.rug.nl
[2]www.mpi-inf.mpg.de/projects/EXACUS
[3]www.cgal.org

of number types we consider relevant parts of it later in detail.

- **ConiX** (CnX) operates on conics in the plane, predicates on conics and their intersections, and computes the arrangement of conics in the plane.
- **CubiX** (CbX) contains predicates on and arrangements of planar curves defined implicitly by bivariate polynomials of degree up to 3.
- **QuadriX** (QdX) collects the representation of quadrics in space, predicates on quadrics, their intersection curves and points, the representation of projected curves, and the arrangement of quadrics in space.
- **AlciX** (AcX) computes geometric properties of real algebraic plane curves of arbitrary degree. These properties contain the topology of the curve as well as the location of singular points and vertical asymptotes. This library is of particular interest for us because it provides our algorithm with the exact input consisting of a set of sweepable segments (see section 2.2).
- **SweepX** (SoX) implements a generic sweep-line algorithm to compute the planar arrangement of segments of algebraic curves and generalized boolean operations on regions bounded by curved arcs. Users of SweepX can either provide their own point and segment classes, or, which is more convenient, use SweepX's own Generic Algebraic Points and Segments (Gaps) based on user-supplied classes for the analysis of curves and curve pairs (as it done ConiX, CubiX, QuadriX and AlciX). SweepX also provides an interface for `CGAL::Arrangement_2`.

As the Exacus libraries are built on class templates, the source code is mainly concentrated in header files. From the Exacus root directory `EXACUS` one can find the source code of a certain library by following the path: `EXACUS/`*library-name*`/include/`*library-abbrev*`/`. For example: `EXACUS/NumeriX/include/NiX/`. The library concepts are defined in `EXACUS/`*library-name*`/manual/concepts/`.

Detailed instructions on how to configure and install Exacus can be found in `EXACUS/developers_manual/manual.ps`. There also listed naming and coding conventions for the Exacus libraries. In particular, if a header file contains a single class or data structure then its name is used as a file name. All classes and files contain a reference documentation in Doxygen format. The reader with theoretical background can access implementation details of Exacus algorithms by starting from `EXACUS/doc.html`.

4.1.1 NumeriX

Now we will outline important parts of the NumeriX library.

Each number type supported by NumeriX belongs to the one of a rich layer of number type concepts[4], which are refinements of one another (for details see [BEH+05]). A concept defines which arithmetic operations supports a number type, whether it is ordered, bounded or exact, which type conversions are allowed.

The properties of number types are collected in appropriate traits classes. Each number type `NT` knows the (most refined) concept it belongs to; it is encoded in `NT_traits<NT>::Algebra_type`. The usual arithmetic and comparison operators are realized through C++ operator overloading.

Polynomials are provided by class `Polynomial< NT >` with coefficients over a number type `NT` which should be the model of a certain concept. Algebraic operations on polynomials include: gcds, resultant computation, factorization by multiplicities. Multivariate polynomials are defined recursively by nesting templates, i.e.,
`Polynomial< Polynomial< NT > >`. For example, a bivariate polynomial $f(x, y)$ is represented as univariate polynomial in y over univariate polynomials in x over some integral domain.

Such a representation makes difficulties when operations w.r.t. the inner variable are required. To overcome these difficulties the files `NiX/bivariate_polynomial_hacks.h` and `NiX/trivariate_polynomial_hacks.h` provide special operations for corresponding polynomials without additional runtime overhead. These operations include, for example, swapping inner and outer variables in polynomial equation (`transpose_bivariate_polynomial`), substitution of the inner variable x with a number (`substitute_x`), differentiation w.r.t. the inner variable (`diff_x`).

Real Algebraic numbers are realized in `Algebraic_real` class. The class represents a real root of a square-free polynomial. The representation consists of the defining polynomial and an isolating interval, which is an open interval containing exactly one root of the polynomial. The polynomial is guaranteed to be non-zero at end-points of the interval. If the root has exact rational value, then the interval is collapsed to a single rational value. The representation is automatically simplified to a rational number or to a one root number if a degree of defining polynomial is 1 or 2 respectively.

[4]A *concept* is a set of syntactical and semantic requirements on a template parameter, and a type is called a *model* of a *concept* if it fulfils these requirements and can thus be used as an argument for the template parameter

Isolating intervals for all real roots of a polynomial are computed by various approaches including the Descartes Method [RZ04]. All real roots are cross-linked, meaning that if one number was used to factorize the defining polynomial, all linked numbers benefit from this simplification.

Interval refinement plays an outstanding role in algorithms which compute isolating intervals. The refinement step is available via the following interface: `refine()` bisects the isolating interval; `strong_refine(m)` refines the isolating interval until m is outside the closed interval; `refine_to(lo, hi)` intersects the current interval with the isolating interval $(lo; hi)$.

ArithmeticTraits concept declares a well-formed collection of the main number types (such as integer, rational and real types) and polynomial types over them. Currently EXACUS provides `LEDA_arithmetic_traits` and `CORE_arithmetic_traits` constructed out of number types taken from LEDA and CORE libraries respectively.

4.1.2 SweepX and Gaps

SWEEPX implements a sweep-line algorithm for line segments, which handles all types of degeneracies in a simple unified event type. The sweep-line approach offers an algorithm to reorder curve segments continuing through a common intersection point in linear time.

A *geometric traits class* for curve segments provides an interface between the generic sweep-line algorithm and the actual geometric operations. The geometric traits class needs a type `Point_2`, representing end-points and intersection points, and a type `Segment_2` representing curve segments. The interface requirements imposed by the point and segment templates are collected in concepts AlgebraicCurve_2 and AlgebraicCurvePair_2.

Classes `Algebraic_curve_segment_2` and `Algebraic_curve_point_2` define a generic (degree-independent) representation of segments and points on algebraic curves and operations on them, which are suitable for all algorithms in SWEEPX. These classes are implemented as part of GAPS, they contain all predicates and constructors needed for generic sweep-line algorithm `SoX::sweep_curves()` and to use `CGAL::Arrangement_2` with every instantiated version of GAPS. They are based on the analysis of a pair of curves as implemented by models of AlgebraicCurvePair_2.

As mentioned before, segment and point class templates are implemented module GAPS. The implementation can handle segments and points of arbitrary algebraic curves. The application libraries have only to provide a curve analysis and a curve pair analysis depending on the actual curve type. Note that points on algebraic curves are defined by isolating intervals in x-range while y-coordinates are specified implicitly by the arc numbers, see definition of a sweepable segment in section 2.2.

Below we list the most relevant properties and methods of classes `Algebraic_curve_segment_2` and `Algebraic_curve_point_2` used in this work.

Segment_2:

- `source()` and `target()` return a source and a target point of a curve segment respectively (both of type `Algebraic_curve_point_2`).
- `support()` returns the supporting algebraic curve of a segment.
- `arcno()` returns the arc number in the interior of a segment.
- `is_degenerate()` indicates whether a segment is degenerate (i.e., source = target).
- `is_reversed()` indicates whether a segment is reversed, i.e., source > target in lexicographical order.
- `is_vertical()` indicates whether a segment is vertical.
- `is_bounded()` indicates whether at least one of the segment's end-points lies at infinity. Not that infinity is not infinity in the sense of projective geometry, but in an "infimaximal" meaning: an infinite x-coordinate is smaller/larger than all finite event points.
- `rational_in_x_range()` returns a rational number within the x-range of a segment, this number can be used as the "seed" point for our algorithm.

Point_2:

- `x()` returns an x-coordinate of a point.
- `curve()` returns a supporting curve of a point.
- `arcno()` returns the arc number of a curve segment at this point.
- `is_infinity_tag_set()` indicates whether a point is an "end-point" of a vertical line at plus or minus infinity. This is useful when, for example, a point is the end-point of a vertical unbounded segment.
- `infiniteness_in_x()` returns a three-valued result, indicating whether the x-coordinate of a point is finite, lies at $-\infty$ or at $+\infty$.
- `infiniteness_in_y()` returns a three-valued result, indicating whether the y-coordinate of a point is finite, lies at $-\infty$ or at $+\infty$. The last two options are valid when a segment has asymptotic behaviour.

It is only left to say some words about unbounded curves and segments, since the curve renderer must be able to distinguish between ordinary and unbounded segments. To include unbounded segments and curves into a uniform representation of sweepable segments, there are symbolic "end-points" defined in Gaps. For instance, a hyperbola $xy - 1 = 0$ is represented as a pair of segments: one from ($x = -\infty$; $arcno = 0$) to ($y = -\infty$); and the other one from ($y = +\infty$) to ($x = +\infty$; $arcno = 0$).

This is done in the following way: first minus- and plus-infinity are symbolically added to the range of x-coordinate values. Then, the x-coordinates for "asymptotic" and unbounded end-points are perturbed. Namely, the x-coordinate of an end-point approaching the pole from the left (the "asymptotic" end-point) is perturbed by $-\varepsilon$, the lower end-point of a vertical line by $-\varepsilon^2$, the upper end-point of a vertical line by ε^2, and an end-point of a curve approaching a pole from the right by ε.

An x-coordinate of an end-point is represented by an object of type `NiX::X_with_tendency`. Which in addition to the x-coordinate provides a method `is_tending_to()`. This method returns the amount by which the x-coordinate was perturbed. This information allows to easily discriminate various end-point types.

4.1.3 AlciX

The AlciX library implements a curve analyzer for arbitrary-degree algebraic curves for Gaps(for details see [Ker06]). The main class `Algebraic_curve_2` is instantiated with `Arithmetic_traits` template argument which is the model of ArithmeticTraits concept.

In the present work the AlciX library is used to decompose an algebraic curve into a set of sweepable segments (for definition of a sweepable segment see section 2.2). A generic curve-decomposition algorithm is realized in functor `SoX::Curve_to_segments` as a part of Gaps. It is instantiated with `Segment_2` and `Algebraic_curve_2` (`AcX::Algebraic_curve_2` in our case) template parameters. Then, the operator call executes the curve analyzer and returns a list of sweepable segments.

Below we describe the most relevant methods of class `Algebraic_curve`:

- `f()` returns the defining polynomial of an algebraic curve.
- `event_info_at_x(Algebraic_real x)` intersects a curve with a vertical line at `x` and returns an object of type `Vert_line` describing the curve's topology at this x-coordinate. This data is used in order to obtain isolating intervals for y-coordinates of segment's end-points.
- `analyse()` analyses curve's topology, in fact, it decomposes the curve into sweepable segments.
- `refine_all(Rational precision)` refines isolating intervals for x-coordinates of all event points until a certain `precision` is reached.

`Vert_line` class is a successor of `SoX::Event1_info`. It describes the topology of a curve over an event x-coordinate. It gives the following information: how many real arcs (without multiplicities) intersect and how many isolated points lie on the vertical line at this x-coordinate; has the curve a vertical line; is there a vertical asymptote and how many arcs approach it from left and right and at minus and plus infinity.

The most relevant methods of this class are the following:

- `refine_to(int j, Rational eps)` refines the y-coordinate of the jth intersection point between the curve and this event line until certain precision.
- `upper_boundary(int j)` returns an upper boundary of the jth isolating interval of the y-coordinate.
- `lower_boundary(int j)` returns a lower boundary of the jth isolating interval of the y-coordinate.
- `interval_length(int j)` returns the length of the jth isolating interval.

4.2 Curve renderer design

In this section we discuss the main implementation details of the curve renderer. Note that the curve parts of EXACUS contain different implementations for visualization. CONIX and QUADRIX use parameterization of curve segments, while CUBIX exploits a subdivision algorithm to visualize complete curves and ALCIX has no visualization at all.

The main goal of this work is to provide a unified reliable and exact visualization algorithm for CONIX, CUBIX and QUADRIX libraries and, in particular, a visualization algorithm for the ALCIX library. The algorithm is realized as part of GAPS since it can be used to render arbitrary plane algebraic curves.

The code of the curve renderer is concentrated into 5 main classes: `SoX::Gfx_GAPS_2`, `SoX::Curve_renderer_2`, `SoX::Affine_form`, `SoX::Subdivision_2` and `CGAL::Qt_widget_Curve_renderer_2`. We consider them one after another.

4.2.1 `Gfx_GAPS_2`

`SoX::Gfx_GAPS_2` is a successor of `SoX::GAPS_2`, and serves as a wrapper class for the curve renderer. Its main functionality consists in providing a convenient interface to the renderer, and switching the renderer to use multi-precision or exact rational arithmetic if necessary. `SoX::Gfx_GAPS_2` is instantiated with three template parameters: `GAPS_2` – an instantiation of `SoX::GAPS_2` to be used as a base class, `Float` – a floating-point number type on which the curve renderer operates, `BigFloat` – a multi-precision floating-point number type for the exact computations.

The public interface of the class contains the following methods:

- `setup(x_min, y_min, x_max, y_max, res_w, res_h)` initializes the renderer with the boundaries of a drawing window and pixel resolution.

- `draw(Segment_2 seg, OutputIterator& points)` draws a distinct curve segment, the result is returned as a list of pixel coordinates.
- `draw(Point_2 seg, OutputIterator& points)` draws a point on a curve, the result is returned as pixel coordinates of a point.

Note that the drawing functions also handle `SoX::Insufficient_precision_exception` which is thrown whenever double precision is not enough to visualize segment or point of an algebraic curve.

4.2.2 `Curve_renderer_2`

The algorithm is encapsulated in the class `SoX::Curve_renderer_2`. This class is instantiated with 3 template parameters. `Segment_2` and `Point_2` are the types of a curve segment and a curve point respectively, obtained from `Gfx_GAPS_2`. `NT` is a main floating-point number type used in all internal computations – user can either choose double, multi-precision or exact rational arithmetic. The curve renderer provides the same set of public methods as `SoX::Gfx_GAPS_2` does, that is why we do not describe it here.

Now we will illustrate some details of the curve renderer implementation.

Polynomial conversion

First of all, one needs to convert an input polynomial defining an algebraic curve to a suitable format. In the current implementation input polynomials are defined over integers which can be arbitrary long. Direct conversion to floating-point representation is not always possible and may cause the information loss. That is why, we first normalize all polynomial coefficients by dividing them by the maximal coefficient and, then, convert the resulting quantities into floating-point representation. This is done by means of the following adaptor classes:

```
template <class NT>
struct Max_coeff
{
    template <class X>
    NT operator()(const NiX::Polynomial<X>& p) const
    {
        typename NiX::Polynomial<X>::const_iterator it = p.begin();
        Max_coeff<NT> max_coeff;
        NT max(max_coeff(*it));
        while(++it != p.end()) {
            NT tmp(max_coeff(*it));
            if(max < ABS(tmp))
                max = ABS(tmp);
        }
        return max;
    }
    NT operator()(const NT& x) const
    { return ABS(x); }
};
template <class Result, class Coeff, class Rational, class From_rat,
        class Make_exact>
struct Transform
{
    Rational max;
    typedef Result result_type;
    From_rat from_rat;
    Make_exact make_exact;
    template <class X>
    Result operator()(const NiX::Polynomial<X>& p)
    {
        Transform<typename Result::NT, Coeff, Rational, From_rat,
            Make_exact> tr;
        tr.max = max;
        return Result(LiS::mapper(p.begin(),tr),
                LiS::mapper(p.end(), tr));
    }
    Result operator()(const Coeff& x)
    {
        Result tmp = static_cast<Result>(from_rat(Rational(x)/max));
        make_exact(tmp);
        return tmp;
    }
};
```

The first of them computes the maximal coefficient, and the second divides all polynomial coefficients by the maximal one. The purpose of Make_exact and From_rat adaptors will be explained later in section 4.2.4.

Main data structures

Below we describe the main data structures used by the renderer. The visualization algorithm mainly operates on pixels and subpixels. Our goal was to use a common data structure for pixels and subpixels which facilitates a subdivision process, it is represented as follows:

```
template <class Integer>
struct Pixel_2
{
    //! pixel coordinates relative to the drawing area
    int x, y;
    //! subpixel coordinates relative to pixel's boundary
    Integer sub_x, sub_y;
    //! subdivision level: 0 - for pixels, 1 - for 1/2 pixels, etc
    unsigned level;
};
```

Here `x` and `y` stand for pixel coordinates, whereas `sub_x` and `sub_y` denote relative subpixel coordinates w.r.t. the pixel's boundary. Subdivision `level` counts how many times the subdivision procedure was applied to a pixel. Any level greater then 0 defines a subpixel, in particular, a pixel contains 4^n subpixels with subdivision level n – in other words 2^n parts in each direction.

Such an integer representation of subpixels helps us to keep away from a great number of round-off errors which are unavoidable while working with floating-point representation. Note that the type of subpixel coordinates is given as a template parameter. This is necessary, since integers would have been enough to store coordinates for up to 31st subdivision level only, which is not sufficient when the exact number types are involved. In the latter case, subpixel coordinates could be represented, for instance, by `CORE::BigInt` number type.

Having this data structure we can outline the most important parts of the implementation. Suppose we have found a subpixel of a pixel `p` (among 4 possible) which is crossed by the curve branch. It is given by the `index` from 0 to 3 – this index is returned by the procedure get_subpixel (see section 3.2.1). Then, the subdivision step results in the following sequence of operations (one assumes that subpixels are numbered in the proper order):

```
    Pixel_2 new_p;
    new_p.x = p.x;
    new_p.y = p.y;
    new_p.level = p.level + 1;
    new_p.sub_x = (p.sub_x<<1) + (index&1);
    new_p.sub_y = (p.sub_y<<1) + (index>>1);
```

After each successful step of the algorithm we advance pixel coordinates in a certain direction (the procedure advance_pixel from section 3.2.1). This is done in the following way:

```
void advance_pixel(Pixel_2& pix, int new_dir)
{
    int x_inc = Intern::directions[new_dir].x,
        y_inc = Intern::directions[new_dir].y;
    if(pix.level == 0) {
        pix.x += x_inc;
        pix.y += y_inc;
    } else {
        Integer x = pix.sub_x + x_inc,
                y = pix.sub_y + y_inc,
                pow = (one << pix.level) - 1;
        (x < 0 ? pix.x-- : x > pow ? pix.x++ : x);
        (y < 0 ? pix.y-- : y > pow ? pix.y++ : y);
        pix.sub_x = x & pow;
        pix.sub_y = y & pow;
    }
}
```

Here `new_dir` represents a direction $(0 \ldots 7)$ of stepping, and the array `Intern::directions` stores x/y increments in a certain direction.

The procedure get_pixel returning a pixel to which a given subpixel belongs is implemented trivially with our representation. One only needs to compare pixel coordinates of two pixels. This test is performed each time a new subpixel is found. Whenever the test reports that pixel coordinates of these two pixels are different – pixel's boundary was crossed – a new pixel is added to the list of plotted pixels.

Cache design

Another important issue of the curve renderer design is a caching mechanism. Reuse of already precomputed quantities becomes crucial for the performance, especially at high subdivision levels. The renderer caches univariate polynomials, function evaluations and y-coordinates of segments' end-points.

Caches are implemented as hashed containers with LRU capabilities, based on `boost::multi_index` class template from the Boost library.[5] Univariate polynomials are hashed by one coordinate – x or y – depending on which coordinate in a bivariate polynomial is fixed. Polynomial evaluations cache uses composite hash keys constructed from 2 point coordinates. End-points coordinates are hashed by unique IDs which have all models of *Point_2* concept.

The precached data can be reused during rasterization of different curve segments (or points) which have the same supporting curve. LRU property ensures that the most recently used entries are located on the top of the cache. Such that, by exhaustion of the allocated space, the least recently used entries are replaced by the new ones.

Moreover, cache data structures are grouped at a higher level called "metacache". Metacache manages a list of several cache instances and one of them is always active at a time. One cache instance is assigned to one supporting curve. Cache instances replace each other depending on which supporting curve is currently used. As a result, we do not have an additional overhead, when segments of two (or more) different curves are drawn alternately.

[5]www.boost.org

Automatic selection of arithmetic precision

We already mentioned that round-off errors may become fatal for the algorithm and it is not always possible to prevent them by using advanced techniques explained in section 3.3.4. That is why we have to switch to a higher-precision arithmetic. In this section we outline cases when increasing of arithmetic precision is necessary and how the mechanism is realized.

We support three levels of arithmetic precision, they differ depending on how polynomial coefficients are represented and which number type is used in all internal computations:

1st level: polynomial coefficients are stored as `double` intervals supported by `boost::interval` library. All computations are performed in double arithmetic.

2nd level: polynomial coefficients are represented by single multi-precision arithmetic numbers (such as `CORE::BigFloat`), all computations are performed in multi-precision arithmetic.

3rd level: only exact rational computations are involved.

At first glance, it would be reasonable to use only two levels: double and multi-precision arithmetic. Indeed, for the latter case one can gradually increase the precision until the algorithm succeed. However, our approach exploits only rational computations and the only irrational quantities involved are dimensions of the drawing window (they can be easily rationalized). At some point it is advantageous to use only rational arithmetic, since multi-precision arithmetic with too large mantissas is considerably slow.

Note that, at first two levels all polynomial evaluations are equipped with round-off tests. If the sign of computed quantity cannot be uniquely determined it is evaluated once again with exact arithmetic:

```
NT res = evaluate(poly, c1);
if(error_bounds) {
    res = evaluate_modular(var, c, key, true);
    if(res != 0)
        res = from_rat(evaluate_rational(var, c, key, true));
    error_bounds = false;
}
```

Here `evaluate` stands for generic evaluation procedure, `error_bounds` flag is set when an interval representing the computed quantity is not strictly positive or negative. In particular, for double arithmetic checking for incorrect boundaries is trivial since the result of evaluation is an interval itself. For multi-precision arithmetic we use an interface provided by a certain number type, for instance, a function `isZeroIn()` for `CORE::BigFloat` returns true if a representation interval includes zero. A function `evaluate_modular` is used for fast zero tests, and `evaluate_rational` calculates with exact rational arithmetic.

Different precision levels are supported through an exception mechanism. There are three criteria indicating that the current arithmetic precision is not sufficient:

- reached the maximum subdivision level specified for the number type. This limit can be set by the user through *Curve_renderer_traits*. The maximum level essentially depends on the integer number type used to store subpixel coordinates (for example, 31 for 32-bit integers), and on the precision of the floating-point number type. However, we suggest to keep a subdivision level not greater than 12-13, since too high subdivision levels indicate the lack of arithmetic precision, in this case it's advantageous to switch to more precise computations rather than tracing a curve branch at high subdivision level.
- a subpixel size is beyond the limits of arithmetic precision: a subpixel is too small such that round-off errors are comparable with the quantity itself (10^{-17} for double precision);
- a subpixel was not found: it might happen that during subdivision of a pixel no intersections with a curve branch were registered, meaning that the right subpixel cannot be determined. In this case we have probably taken a wrong direction, again because of inexact computations.

All other erroneous situations finally lead to one of these cases.

4.2.3 Affine arithmetic support

Support for affine arithmetic is implemented in the class template `SoX::Affine_form`. This class realizes basic operations on an affine form as it is presented in theory. It is instantiated with a template parameter specifying a floating-point number type. The class supports usual arithmetic operations such as `+, -, *` via C++ operators. An affine form can be initialized with a fixed value or with an interval defining a variable quantity. It provides means to convert from the affine representation back to the interval.

In fact we implemented two methods for range analysis: Affine Arithmetic and Modified Affine Arithmetic explained in section 2.3.2. Both use recursive derivative information. Although more precise, our tests showed MAARD method requires expensive computations, therefore we have fixed on using AARD method.

4.2.4 Curve renderer traits

A class template `Curve_renderer_traits` provides usual type-conversions and also specifies a type-dependent information necessary for the renderer. The class is initiated with a polynomial coefficient number type, this can be either a floating-point or rational number type. Let us consider the specialization of this class for number type `NT`:

- `Float` is a number type used in all internal computations. Note that it can differ from the number type used to store polynomial coefficients. For example, we represent coefficients by intervals of doubles but all computations are performed over doubles.
- `Integer` is an integer number type used to store subpixel coordinates. In particular, we use machine `int` with double-precision floating point, and `CORE::BigInt` for `CORE::BigFloat`.
- `To_integer` adaptor converts `Float` to integers.
- `From_rational` adaptor converts from rational representation to `Float`. Rational number type is specified as a template argument.
- `To_rational` adaptor converts back from `Float` to rational number type.
- `To_machine_int` adaptor converts from `Integer` number type to machine `int`.
- `Convert_poly` serves for conversion from a rational polynomial to a polynomial over number type `NT`.
- `Make_exact` adaptor is used to set the error component of a multi-precision floating point number after each inexact operation, such as division, square root, etc.
- `Extract_eval` converts the result of a polynomial evaluation from a number type `NT` to `Float`. It also sets the flag `error_bounds` if the resulting interval is not strictly positive or negative.
- `Precision_limit` compares a given quantity with the limit of current arithmetic precision, returns true if this limit is exceeded.
- `MAX_SUBDIVISION_LEVEL` constant defines the maximum subdivision level for the number type. Setting up too big values for inexact number types does not raise the precision of computations.

The specialization of this class for the number type `NiX::Interval` looks as follows:

```
template <>
struct Curve_renderer_traits<NiX::Interval> {
    typedef int Integer;
    typedef double Float;            //! a floating-point number type
    typedef NiX::Interval Coeff;     //! a type of polynomial coefficients
    struct To_integer {
        typedef Float argument_type;
        typedef Integer result_type;
        Integer operator()(const Float& x) const
        { return static_cast<Integer>(floor(x)); }
    };
    struct From_rational {
        typedef Float result_type;
        template<class Rational>
        Float operator()(const Rational& x) const
        { return NiX::to_double(x); }
    };
    struct To_rational {
        typedef Float argument_type;
        typedef CORE::BigRat result_type;
        CORE::BigRat operator()(const Float& x) const
        {
            return static_cast<CORE::BigRat>(x);
        }
    };
    struct To_machine_int {
        typedef Integer argument_type;
        typedef int result_type;
        int operator()(const Integer& x) const
        {
            return x;
        }
    };
    struct Convert_poly { //! converts a rational polynomial to internal
                          //! representation
        typedef NiX::Polynomial<NiX::Polynomial<Coeff> > result_type;
        template <class Rational>
        result_type operator()(const
            NiX::Polynomial<NiX::Polynomial<Rational> >& poly)
        {
            typename NiX::NT_traits<NiX::Polynomial<
                NiX::Polynomial<Rational> > >::To_Interval to_interval;
            return to_interval(poly);
        }
    };
    //! converts the result of evaluation from Coeff to Float number type
    //! sets error_bounds flag whenever round-off errors may occur
    struct Extract_eval {
        typedef Coeff argument_type;
        typedef Float result_type;
        Float operator()(const Coeff& x, bool& error_bounds) const
        {
            Float l = x.lower(), u = x.upper(), mid = (l+u)/2;
            error_bounds = (l < 0 && u > 0);
            return mid;
        }
    };
    //! not necessary for doubles since it is not an exact number type
    struct Make_exact {
        typedef Float argument_type;
        typedef void result_type;
        void operator()(const Float& x) const
        { }
    };
    //! compares a given quantity with the precision limit of the number
    //! type, returns true if this limit is exceeded
    struct Precision_limit {
        typedef Float argument_type;
        typedef bool result_type;
        bool operator()(const Float& x) const
        {
            return (ABS(x) <= 1e-16);
        }
    };
    //! maximum subdivision level for the curve renderer by exceeding which
    //! Insufficient_rasterize_precision_exception is thrown
    static const unsigned MAX_SUBDIVISION_LEVEL = 16;
};
```

4.2.5 Subdivision_2

Space subdivision algorithm was implemented for testing purposes to compare a visual quality and efficiency of our approach. As a range analysis tool we employ Recursive Taylor approach presented in [SMW+05]. The algorithm is fairly simple: in order to rasterize an implicit curve $C(f)$ over a given rectangular interval $[\underline{x}, \overline{x}] \times [\underline{y}, \overline{y}]$ where $f(x, y)$ is a bivariate polynomial we use the following recursive procedure:

```
procedure quad_tree($\underline{x}, \overline{x}, \underline{y}, \overline{y}$)
1.  $F \leftarrow$ range_evaluation($\underline{x}, \overline{x}, \underline{y}, \overline{y}$)
2.  if $\underline{F} \le 0 \le \overline{F}$ then
3.      if $\overline{x} - \underline{x} <$ pixel_size and $\overline{y} - \underline{y} <$ pixel_size then
4.          plot_pixel($\underline{x}, \overline{x}, \underline{y}, \overline{y}$)
5.      else subdivide($\underline{x}, \overline{x}, \underline{y}, \overline{y}$)
end

procedure subdivide($\underline{x}, \overline{x}, \underline{y}, \overline{y}$)
1.  $\hat{x} \leftarrow (\underline{x} + \overline{x})/2$
2.  $\hat{y} \leftarrow (\underline{y} + \overline{y})/2$
3.  quad_tree($\underline{x}, \hat{x}, \underline{y}, \hat{y}$)
4.  quad_tree($\underline{x}, \hat{x}, \hat{y}, \overline{y}$)
5.  quad_tree($\hat{x}, \overline{x}, \hat{y}, \overline{y}$)
5.  quad_tree($\hat{x}, \overline{x}, \underline{y}, \hat{y}$)
end
```

Here range_evaluation($\underline{x}, \overline{x}, \underline{y}, \overline{y}$) denotes application of Recursive Taylor method on a rectangular interval $[\underline{x}, \overline{x}] \times [\underline{y}, \overline{y}]$.

The public interface of the class consists of three methods:

- `setup(x_min, y_min, x_max, y_max, res_w, res_h)` initializes the class with boundaries of the drawing window and pixel resolution.

- `set_polunomial(poly)` sets up the supporting bivariate polynomial for algebraic curve.
- `draw()` rasterizes a given curve over `[x_min, y_min]x[x_max, y_max]` region.

4.2.6 Providing an interface to CGAL

In light of forthcoming integration of EXACUS and CGAL libraries it is extremely important to provide an intuitively clear and easy-to-use interface with our algorithm to CGAL.

The visualization mechanism of the CGAL library is based on the class `CGAL::Qt_widget`. It extends the standard functionality of Qt's `QWidget`[6] with the ability of drawing various CGAL objects, such as: Delaunay triangulations, arrangements of line segments and circular arcs, polyhedrons, etc. These graphical primitives are supported through C++ output operator interface. In particular, each graphic object provides an output operator for `CGAL::Qt_widget` and outputs its graphical representation to the context. C++ operator overloading allows to built a uniform and simple interface for all CGAL primitives. Moreover, the set of supported graphic objects can be easily extended in future versions.

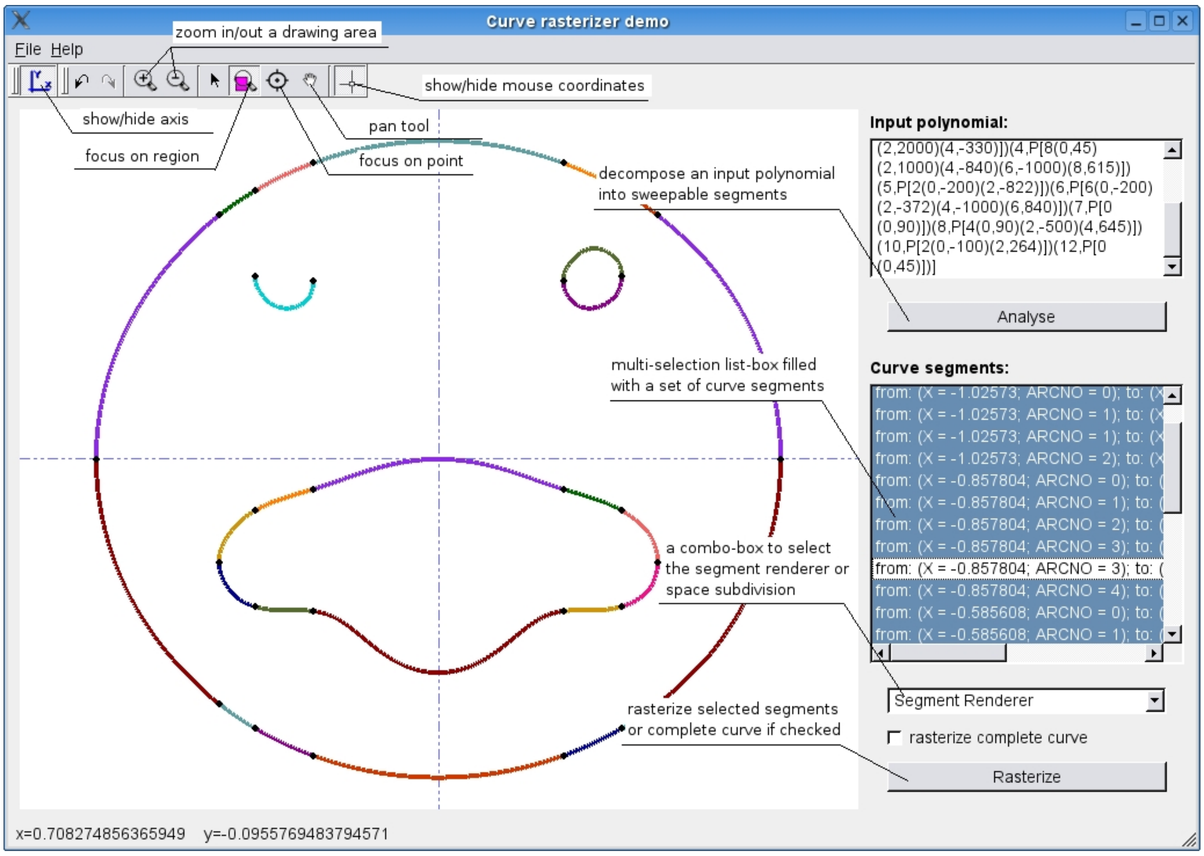
Our interface to CGAL is implemented in a file `Qt_widget_Curve_renderer.h` which must be located in CGAL header directory. There are two public functions to visualize algebraic curve segments and points:

```
//! outputs an algebraic curve segment to Qt_widget
template <class AlgebraicCurvePair_2>
Qt_widget& operator << (Qt_widget& ws,
    const SoX::Algebraic_curve_segment_2<
        SoX::Algebraic_curve_point_2<AlgebraicCurvePair_2> >& seg);

//! outputs an algebraic curve point to Qt_widget
template <class AlgebraicCurvePair_2>
Qt_widget& operator << (Qt_widget& ws,
    const SoX::Algebraic_curve_point_2<AlgebraicCurvePair_2>& pt);
```

With CGAL interface, visualization of curve segments and points is rather trivial. Note that, one does not need to instantiate `Gfx_GAPS_2` explicitly: this happens automatically in a corresponding output operator. The curve renderer is initialized with the drawing window and pixel resolution obtained from the hosting `CGAL::Qt_widget` object. It also automatically reacts on any rescalings or shifts of the widget's drawing window and changes its parameters appropriately. The usage of CGAL interface illustrates the following example:

[6]Further information about Qt widgets can be obtained from http://doc.trolltech.com


Curve rasterizer demo
File Help
zoom in/out a drawing area
show/hide mouse coordinates
show/hide axis
focus on region
pan tool
focus on point
decompose an input polynomial into sweepable segments
multi-selection list-box filled with a set of curve segments
a combo-box to select the segment renderer or space subdivision
rasterize selected segments or complete curve if checked
Input polynomial:
Analyse
Curve segments:
Segment Renderer
rasterize complete curve
Rasterize

```
(2,2000)(4,-330)])(4,P[8(0,45)
(2,1000)(4,-840)(6,-1000)(8,615)])
(5,P[2(0,-200)(2,-822)])(6,P[6(0,-200)
(2,-372)(4,-1000)(6,840)])(7,P[0
(0,90)])(8,P[4(0,90)(2,-500)(4,645)])
(10,P[2(0,-100)(2,264)])(12,P[0
(0,45)])]
```

```
from: (X = -1.02573; ARCNO = 0); to: (X
from: (X = -1.02573; ARCNO = 1); to: (X
from: (X = -1.02573; ARCNO = 1); to: (X
from: (X = -1.02573; ARCNO = 2); to: (X
from: (X = -0.857804; ARCNO = 0); to: (
from: (X = -0.857804; ARCNO = 1); to: (
from: (X = -0.857804; ARCNO = 2); to: (
from: (X = -0.857804; ARCNO = 3); to: (
from: (X = -0.857804; ARCNO = 3); to: (
from: (X = -0.857804; ARCNO = 4); to: (
from: (X = -0.585608; ARCNO = 0); to: (
from: (X = -0.585608; ARCNO = 1); to: (
```

x=0.708274856365949 y=-0.0955769483794571

Fig 4.1: The xalci demo program with the description of user interface

```
namespace SoX {
typedef GAPS_2<AcX::Algebraic_curve_pair_2<Algebraic_curve_2> >
    GAPS_2_inst;
// extract Segment and Point types
typedef GAPS_2_inst::Segment_2 Segment_2;
typedef GAPS_2_inst::Point_2 Point_2;
} // namespace SoX

SoX::Segment_2 seg;
CGAL::Qt_widget widget;
...
widget << seg; // output a curve segment to Qt_widget
```

4.3 The demo program

The executable demo program of the algorithm can be found in `AlciX/demos` directory and is called `xalci` (see Figure 4.1). The program demonstrates usage of the AlciX library as well as our curve renderer.

Working with `xalci` is straightforward. First, the user inputs a bivariate polynomial.[7] Then, analyzes it by clicking the "Analyze" button. A set of sweepable segments result from the curve's decomposition is displayed in the list-box. The list-box is multi-selectable, such that the user can select certain segments to rasterize. To visualize a curve completely one should activate the check-box "rasterize complete curve". Curve segments are rasterized in different colors to make them distinguishable by sight, segments' end-points are marked with big black dots.

Curve segments in the list-box are represented in the following way:

```
 (from:(X = <x_coord_1>, ARCNO = <arcno_1>); to:(X = <x_coord_2>,
   ARCNO = <arcno_2>); segment arcno:  <seg_arcno> | vertical),
```

where `<x_coord_1/2>` and `ARCNO = <arcno_1/2>` denote x-coordinate and the arc-number of an end-point respectively, `<seg_arcno>` defines the segment's arc-number for non-vertical segments (for details on segment arc-numbers see section 2.2).

4.4 Benchmarks

To convince the reader about the correctness and efficiency of our approach, we have measured running times to visualize a set of algebraic curves from the test suite. Aside from the visual quality we are also concerned with the computational performance of the algorithm. That is why we compared the output of our algorithm with a general subdivision algorithm implemented within the frame of our work directly for testing purposes. The latter algorithm employs Recursive Taylor as 2D range analysis tool described in [SMW$^+$05], unlike general Affine Arithmetic this method offers rather better visual quality although loses in performance.

To measure the running time, we used `xalci` on a Pentium IV 2,8 Ghz dual core with 1024 KB cache and 1 GB of RAM under Linux. The demo program was compiled with the GNU C++ compiler v3.3.5 with optimizations (`-O2`) and with all assertions switched off (`-DNDEBUG`). The exact number types were provided by the Core library version 1.7. The times were measured with `LiS::User_timer` which reports on the times allocated by the operating system. We provide the

[7]Through the specificity of the AlciX library, one can only input polynomials with integer coefficients. However, this is not a severe restriction, since integer coefficients can be arbitrarily large and, thus, can approximate any real numbers with the desired precision.

running times needed for curve analysis using the ALCIX library and for curve rasterization separately, as well as the overall running time. ALCIX was set up to use modular gcd computations from the NTL library which results in a significant speed-up.

Our test suite contains examples kindly provided by the author of the ALCIX library ([Ker06]) and also some "famous" algebraic curves taken from the Internet:

- grid_3 is a grid constructed out of vertical and horizontal lines, the total degree is 3.
- apple_7 is an apple-like curve of total degree 7. This curve already occurs in [SW05].
- infinitesimal_7 is a curve of total degree 7 with an infinitesimal singular point in the origin.
- L4_circles is the union of 4 circles w.r.t. the L_4 norm, the total degree is 16. The curve is already occurred in [Ker06, SW05].
- ten_circles is the union of 10 circles, the total degree is 20. The curve was taken from [Ker06].
- lemniscate_16 is Erdõs lemniscate of degree 16, i.e., the curve given by an implicit equation $|z^8 - 1| = 1$ with $z = x + yi$.
- bundle_26 is a curve of total degree 26, it depicts the curvature of degree 10 lemniscate.
- rand_10 is a curve of total degree 10 with a lot of grid points.
- spiral_38 is a Taylor expansion of order 40 approximating Fermat's spiral in the origin.
- inf_7_mul_der_23 is a curve of total degree 23, produced by multiplying infinitesimal_7 with its partial derivatives several times. It contains several curve branches coming in a "bundle" near the origin.

Our implementation of a space subdivision method uses Recursive Taylor approach as mentioned before. All partial derivatives are precomputed and stored in a tree-like structure in order to avoid duplicates, as, for instance, $f'''_{xxy} = f'''_{xyx}$. Practice has shown, that for high-degree polynomials the overhead for calculations with high-order derivatives increases quite rapidly. That is why, we suggested to use computations with partial derivative up to the order 6 and switch to the standard Affine Arithmetic for higher-order derivatives. This does not influence the visible quality, but reduces dependency of the running on polynomial's degree.

Figures 4.2 and 4.3 display graphs of the curves in the order they are listed above. Figures 4.4 and 4.5 display the same curves rendered with Recursive Taylor method. One may observe some visible artifacts, especially in rasterizations of curves lemniscate_16, bundle_26 and inf_7_mul_der_23. Figure 4.6 shows focusing at a singular point for curves apple_7, infinitesimal_7, then follows two consecutive

Curve	Analyze	Rasterize	Zoom on point	Subdivision
grid_3	0.008	0.09	–	0.65
apple_7	0.168	0.16	0.43	4.42
infinitesimal_7	0.02	0.116	0.136	2.08
L4_circles	16.2	0.83	3.89	32.66
ten_circles	57.99	13.36	–	115.7
lemniscate_16	22.19	0.808	1.28	36.6
bundle_26	108.2	6.97	8.25	186
rand_10	22.3	6.5	–	19.0
spiral_38	229.5	68.9	–	512.2
inf_7_mul_der_23	44.7	16.0	19.82	283.5

Table 4.1: Time measurements

zooms of L4_circles, and at the end, zooming at a singularity of inf_7_mul_der_23. Remark that, the curve L4_circles has the following equation:

$$\begin{aligned} f(x,y) &= ((x+1)^4 + (y+1)^4 - 2) \cdot ((x+1)^4 + (y-1)^4 - 2) \cdot \\ &\quad ((x-1)^4 + (y+1)^4 - 2) \cdot ((x-1)^4 + (y-1)^4 - 2 - 10^{-5}). \end{aligned}$$

In other words, it consists of four circles w.r.t. the L_4 norm, the radius of one of them is slightly enlarged causing new singularities close to the origin. This can be observed in the figure.
Table 4.1 shows the time measurements for our algorithm and Recursive Taylor approach, the pictures were taken with resolution 1024×700, all running times are given in seconds. The column "Analyze" represents the time required to decompose a curve into sweepable segments using the ALCIX library; "Rasterize" denotes the time taken to rasterize a complete curve using our approach; "Zoom on point" specifies the time to rasterize a curve after zooming on a singular point, this typically takes more time since curve branches come closer to each other in this region; the last column gives the times for space subdivision approach. From the table one can see that our approach outperforms the space subdivision. Moreover, for complicated curves analyze phase generally takes more time than rasterization. This can be put as an advantage of our approach: indeed, we are only required to analyze the curve *once* and, then, can zoom and shift it arbitrary or visualize distinct segments without any additional overhead. On the other hand, once a polynomial degree gets higher the effort required to refine the coordinates of segments' end-points grows quite rapidly. Such that for complicated curves a refinement phase typically takes more time than the "pure" visualization. However, this time depends on the concrete algorithm used to manage and refine intervals and, therefore, is not the object of our discussion.

The table also shows a strong dependency of the space subdivision method on the polynomial's degree. This is due to the fact that the subdivision method employs 2D range analysis which involves calculations of mixed derivatives whose number growth quadratically w.r.t. the degree of a polynomial, whereas our approach needs only 1-dimensional analysis and, thus, complexity of calculations depends linearly on the polynomial's degree.

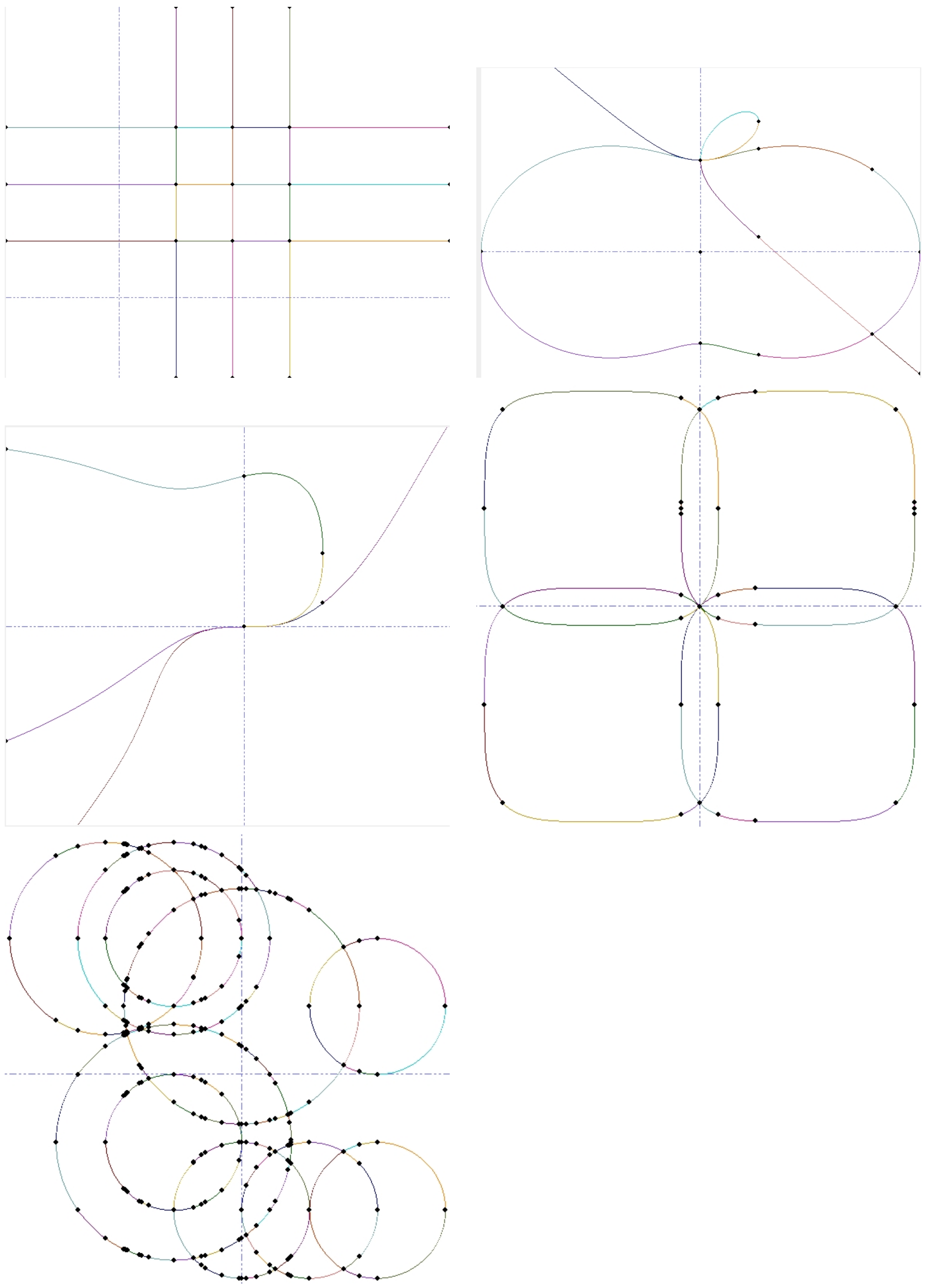


Fig 4.2: Segment renderer: grid_3, apple_7, infinitesimal_7, L4_circles and ten_circles

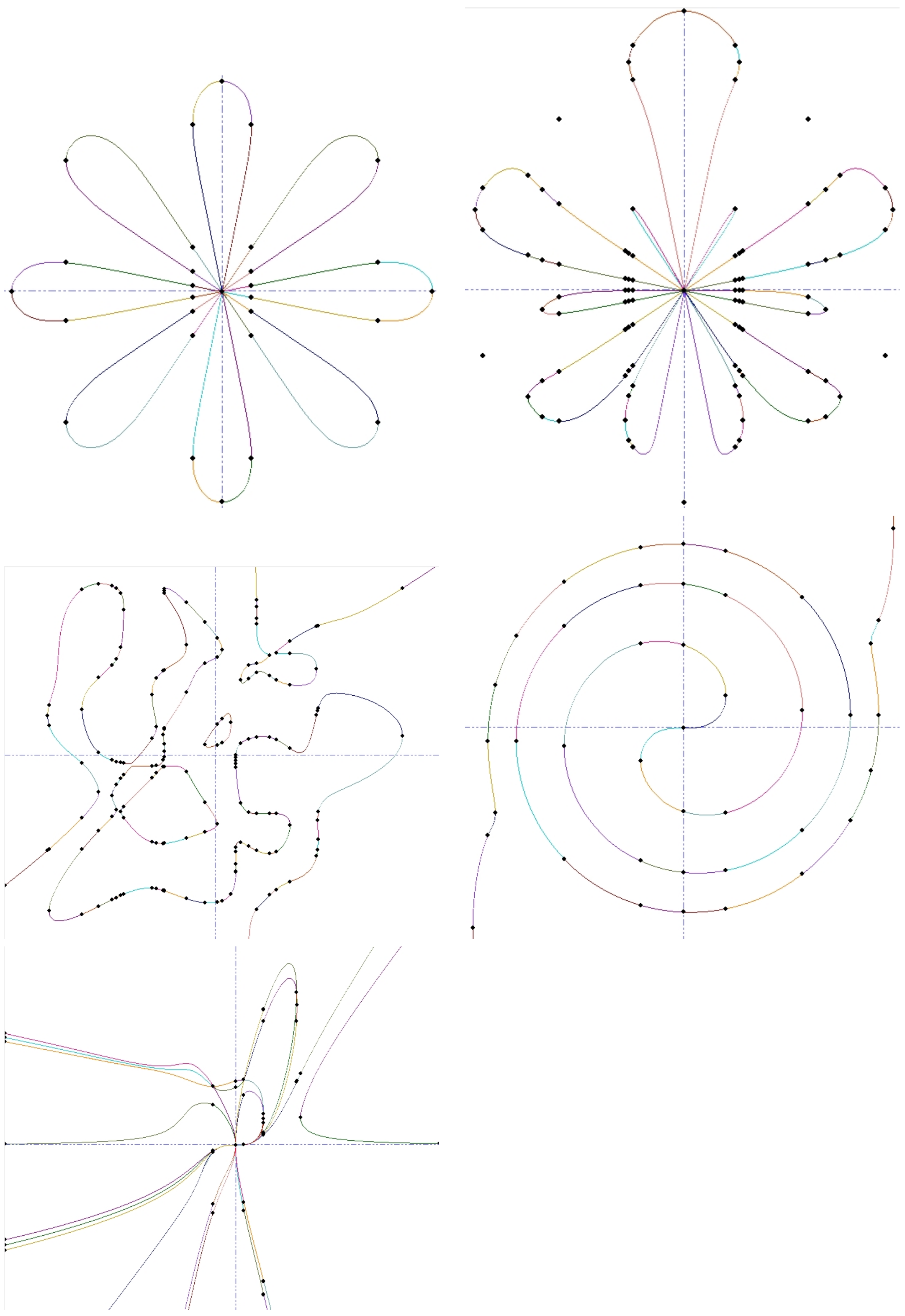


Fig 4.3: Segment renderer: lemniscate_16, bundle_26, rand_10, spiral_38 and inf_7_mul_der_23

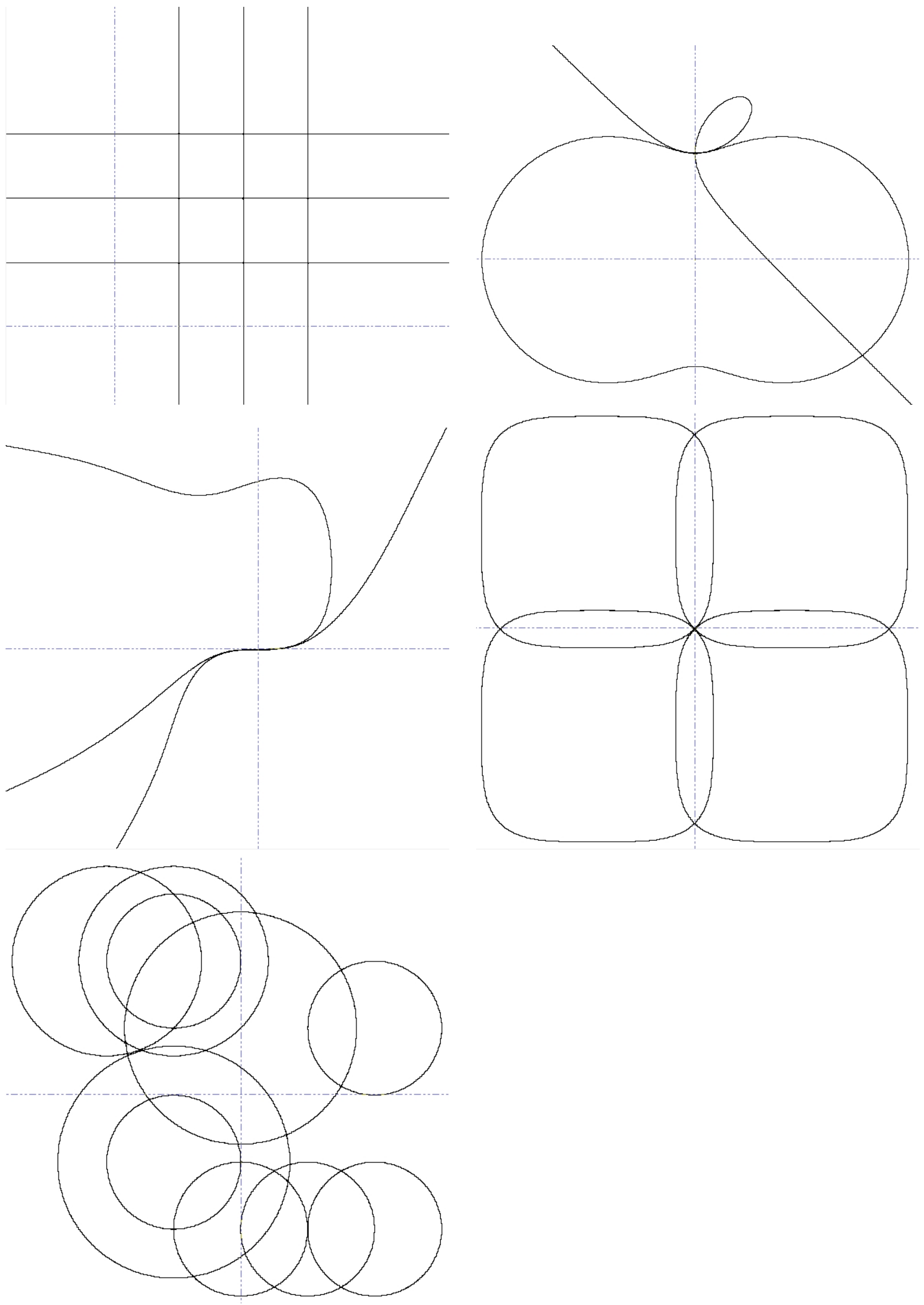


Fig 4.4: Space subdivision: grid_3, apple_7, infinitesimal_7, L4_circles, L4_circles and ten_circles

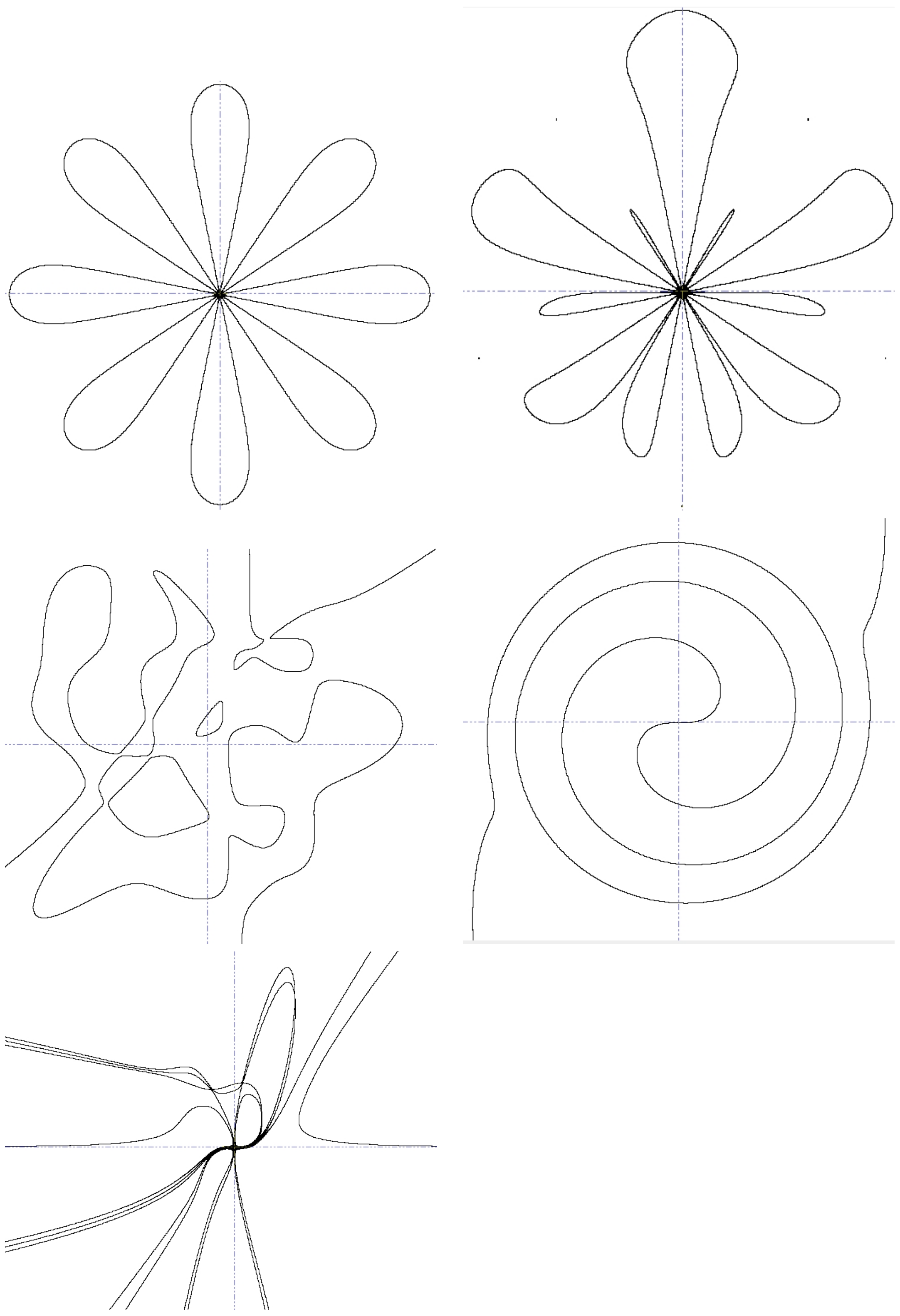


Fig 4.5: Space subdivision: lemniscate_16, bundle_26, rand_10, spiral_38 and inf_7_mul_der_23

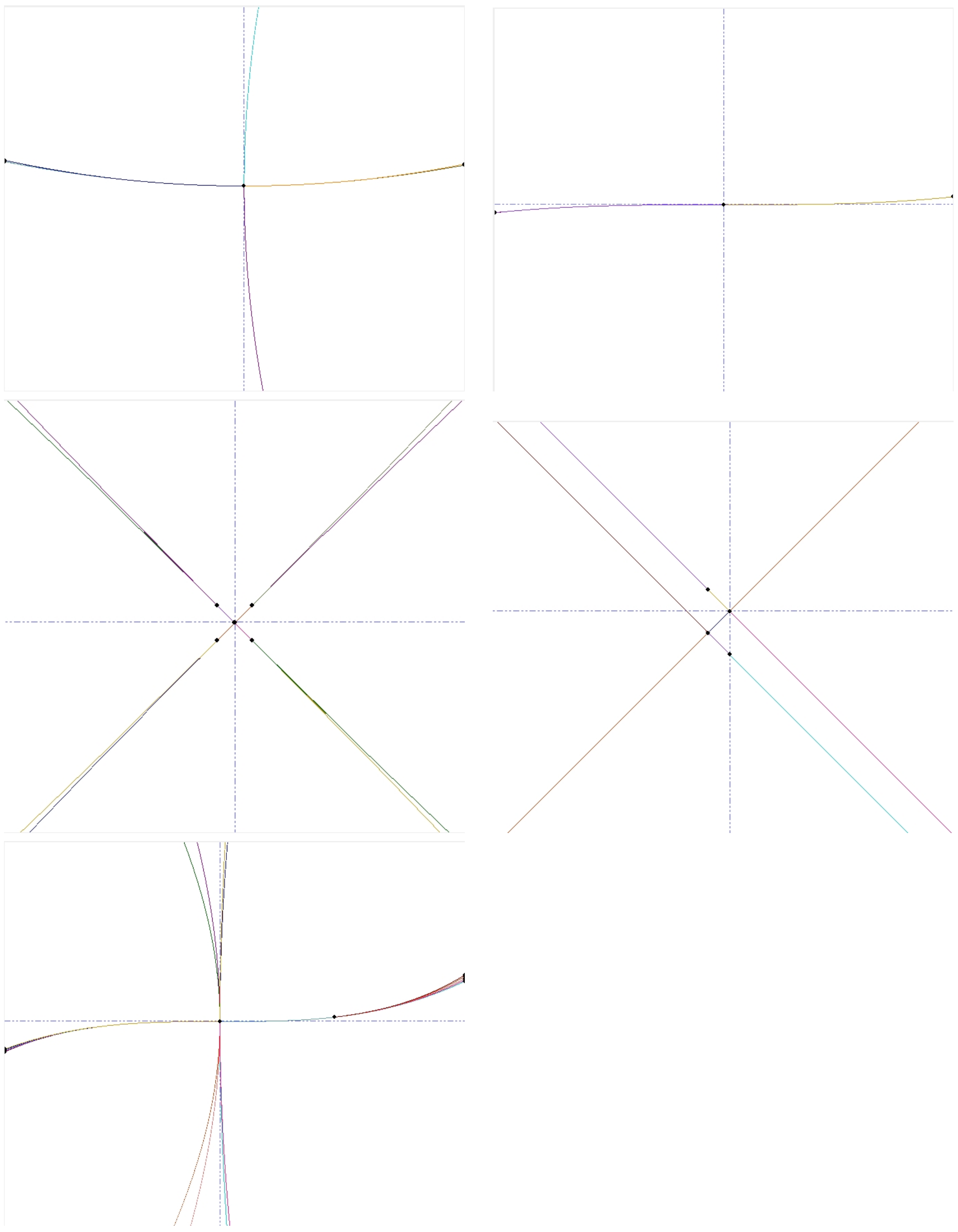


Fig 4.6: Focus on point: apple_7, infinitesimal_7, L4_circles, L4_circles and inf_7_mul_der_23

Chapter 5

Conclusion

Summary and Outlook

Within the frame of this thesis the algorithm for visualization of points and segments of algebraic curves was developed and implemented as part of Exacus libraries.

Due to adaptive subdivision and optimization technique we can handle highly degenerate cases and discriminate curve segments located very close to each other. By keeping track of round-off errors our approach offers correct and reliable visualization using only double-precision arithmetic. However, to handle exceptional cases we developed a three-level model to gradually increase the arithmetic precision in case of need.

With the help of this algorithm one is able draw polygons whose boundaries are defined by segments of arbitrary-degree plane algebraic curves. Moreover, the algorithm can be used to unify a curve rasterization procedure for all Exacus libraries.

As a future development in this area, one, for instance, may think of extending the algorithm ideas to rasterize algebraic curves or surfaces in 3D or to adapt the existent approach to rasterize complete algebraic curve without costly pre-processing step.

Performance tuning may also be an object of future research: indeed, in the long run the overall performance greatly depends on used range analysis method. That is why, enhancing the present range analysis techniques to make them less expensive and to obtain more tight bounds may result in a significant performance increase. However, the performance partly depends on a curve analysis technique, which should also evolve accordingly to achieve competitive results.

Bibliography

[AG91] E. Allgower and S. Gnutzmann. Simplicial pivoting for mesh generation of implicitly defined surfaces. *Computer Aided Geometric Design*, 8, 1991.

[BEH$^+$05] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, L. Kettner, K. Mehlhorn, J. Reichel, S. Schmitt, E. Schömer, and N. Wolpert, editors. EXACUS*: Efficient and Exact Algorithms for Curves and Surfaces.* Springer-Verlag, 2005. Proceedings of the 13th Annual European Symposium Algorithms (ESA'05).

[Blo88] J. Bloomenthal. Polygonalisation of implicit surfaces. *Computer Aided Geometric Design*, 5, 1988.

[Cha88] R. Chandler. A tracking algorithm for implicitly defined curves. *IEEE Computer Graphics and Applications*, 8, 1988.

[EKK$^+$05] A. Eigenwillig, L. Kettner, W. Krandick, K. Mehlhorn, S. Schmitt, and N. Wolpert. A Descartes algorithm for polynomials with bit-stream coefficients. *CASC 2005 : computer algebra in scientific computing*, 2005.

[FS04] L. Figueiredo and J. Stolfi, editors. *Affine Arithmetic: Concepts and Applications.*, volume 37, 2004. SCAN'2002 International Conference.

[Gib98] C. G. Gibson. *Elementary geometry of algebraic curves: an undergraduate introduction.* Cambridge University Press, New York, NY, USA, 1998.

[Ker06] M. Kerber. Analysis of real algebraic plane curves. Master's thesis, Saarland University, Saarbrücken, Germany, 2006.

[KM95] S. Krishnan and D. Manocha, editors. *Numeric-symbolic algorithms for evaluating one-dimensional algebraic sets.*, volume 10, 1995. Proceedings of ISSAC '95.

[MG03] J. Morgado and A. Gomes., editors. *A non-uniform binary space partition algorithm for 2D implicit curves.*, volume 2669 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003. Proceedings of the International Conference on Computational Science and Its Applications (ICCSA'2003), also 2nd International Workshop on Computer Graphics and Geometric Modeling (CGGM'2003).

[MG04a] J. Morgado and A. Gomes., editors. *A derivative-free tracking algorithm for implicit curves with singularities.*, volume 3039 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004. Proceedings of the 4th International Conference on Computational Science (ICCS'2004), also 3rd International Workshop on Computer Graphics and Geometric Modeling (CGGM'2004).

[MG04b] J. Morgado and A. Gomes. Rendering implicit curves with isolated points through binary space partitioning. *The Virtual - Portuguese Journal of Computer Graphics*, 2004.

[MN00] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computng.* Cambridge University Press, Cambridge, UK, 2000.

[Moo66] R. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1966.

[MSV$^+$02] R. Martin, H. Shou, I. Voiculescu, A. Bowyer, and G. Wang. Comparison of interval methods for plotting algebraic curves. *Computer Aided Geometric Design*, 19, 2002.

[MY95] T. Möller and R. Yagel. Efficient rasterization of implicit functions. Technical report, Department of Computer and Information Science, Ohio State University, 1995.

[Per06] S. Perikleous. Algorithms in real algebraic geometry i: The univariate case. 5, 2006.

[RZ04] F. Rouillier and P. Zimmermann. Efficient isolation of polynomial's real roots. *Computational Applied Mathematics*, 162, 2004.

[She96] J. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete and Computational Geometry*, 18, 1996.

[SMW$^+$] H. Shou, R. Martin, G. Wang, A. Bowyer, and I. Voiculescu. Affine arithmetic in matrix form for polynomial evaluation and algebraic curve drawing. *Progress in Natural Science 12(1).*, 77.

[SMW$^+$05] H. Shou, R. Martin, G. Wang, A. Bowyer, and I. Voiculescu. A recursive taylor method for algebraic curves and surfaces. 2005.

[SW05] R. Seidel and N. Wolpert, editors. *On the Exact Computation of the Topology of Real Algebraic Curves.* ACM Press, 2005. Proceedings of the twenty-first annual symposium on Computational geometry.

[Tau94] G. Taubin. Rasterizing algebraic curves and surfaces. *IEEE Computer Graphics and Applications*, 14, 1994. IEEE Computer Society Press, Los Alamitos, CA, USA.

[ZsYzMj$^+$06] Y. Zheng-sheng, C. Yao-zhi, O. Min-jae, K. Tae-wan, and P. Qun-sheng. An efficient method for tracing planar implicit curves. *Journal of Zhejiang University - Science A*, pages 1115–1123, 2006.