Visualizing Arcs of Implicit Algebraic Curves, Exactly and Fast

Pavel Emeliyanenko[1], Eric Berberich[2], Michael Sagraloff[1]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
`[asm|msagralo]@mpi-sb.mpg.de`
[2] School of Computer Science, Tel-Aviv University, Tel-Aviv, Israel
`ericb@post.tau.ac.il`

Abstract. Given a Cylindrical Algebraic Decomposition of an implicit algebraic curve, visualizing distinct curve arcs is not as easy as it stands because, despite the absence of singularities in the interior, the arcs can pass arbitrary close to each other. We present an algorithm to visualize distinct connected arcs of an algebraic curve efficiently and precise (at a given resolution), irrespective of how close to each other they actually pass. Our hybrid method inherits the ideas of subdivision and curve-tracking methods. With an adaptive mixed-precision model we can render the majority of algebraic curves using floating-point arithmetic without sacrificing the exactness of the final result. The correctness and applicability of our algorithm is borne out by the success of our web-demo[1] presented in [10].

Key words: Algebraic curves, geometric computing, curve rendering, visualization, exact computation

1 Introduction

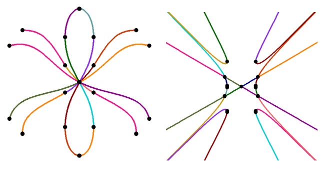


Fig. 1. "Spider": a degenerate algebraic curve of degree 28. Central singularity is enlarged on the left. Arcs are rendered with different colors.

In spite of the fact that the problem of rasterizing implicit algebraic curves has been in research for years, the interest in it never comes to an end. This is no surprise because algebraic curves have found many applications in Geometric Modeling and Computer Graphics. Interestingly enough, the task of rasterizing separate curve arcs,[2] which, for instance, are useful to represent "curved" polygons, has not been addressed explicitly upon yet. That is why, we first give an overview of existing methods to rasterize complete curves. For an algebraic curve $\mathcal{C} = \{(x, y) \in \mathbb{R}^2 : f(x, y) = 0\}$, where $f \in \mathbb{Q}[x, y]$, algorithms to compute a curve approximation in a rectangular domain $\mathcal{D} \in \mathbb{R}^2$ can be split in three classes.

[1] `http://exacus.mpi-inf.mpg.de`
[2] A curve arc can informally be defined as a connected component of an algebraic curve which has no singular points in the interior; see Section 2.

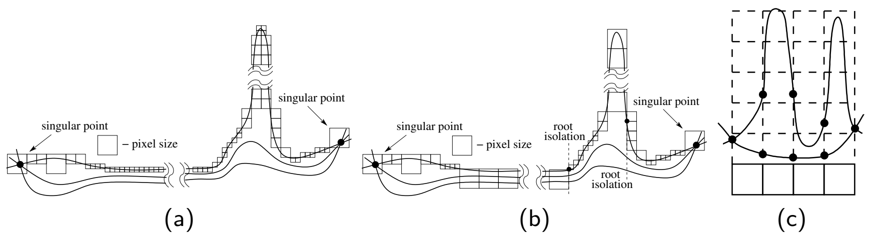


Fig. 2. (a) even with an exact solution of $f = 0 \wedge f_y = 0$, the attempt to cover a curve arc by a set of xy-regular domains using subdivision results in a vast amount of small boxes; (b) our method stops subdivision as soon as the direction of motion along the curve is uniquely determined; (c) exact root isolation at fixed positions (pixel boundary) can easily overlook high-curvature points of the curve

Space covering. These are numerical methods which rely on *interval analysis* to effectively discard the parts of the domain not cut by the curve and recursively subdivide those that might be cut. Algorithms [8, 17] guarantee the geometric correctness of the output, however they typically fail for singular curves.[1] More recent works [1, 4] subdivide the initial domain in a set of xy-regular subdomains where the topology is known and a set of "insulating" boxes of size $\leq \varepsilon$ enclosing possible singularities. Yet, both algorithms have to reach the root separation bounds to guarantee the correctness of the output. Altogether, these methods alone cannot be used to plot distinct curve arcs because no continuity information is involved.

Continuation methods are efficient because only points surrounding a curve arc are to be considered. They typically find one or more seed points on a curve, and then follow the curve through adjacent pixels/plotting cells. Some algorithms consider a small pixel neighbourhood and obtain the next pixel based on sign evaluations [5,20]. Other approaches [15,16,18] use Newton-like iterations to compute the point along the curve. Continuation methods commonly break down at singularities or can identify only particular ones.

Symbolic methods use projection techniques to capture topological events – tangents and singularities – along a sweep line. This is done by computing Sturm-Habicht sequences [7, 19] or Gröbner bases [6]. There is a common opinion that knowing exact topology obviates the problem of curve rasterization. We disagree because symbolic methods disrespect the size of the domain $\mathcal{D}$ due to their "symbolic" nature. The curve arcs can be "tightly packed" in $\mathcal{D}$ making the whole rasterization inefficient; see Figure 2 (a). Using *root isolation* to lift the curve points in a number of fixed positions also does not necessarily give a correct approximation because, unless x-steps are adaptive, high-curvature points might be overlooked, thereby violating the Hausdorff distance constraint; see Figure 2 (c).

[1] By geometrically-correct approximation we mean a piecewise linear approximation of a curve within a given Hausdorff distance $\varepsilon > 0$.

Given a Cylindrical Algebraic Decomposition (CAD) of $\mathcal{C}$, for each distinct curve arc we output a sequence of pixels which can be converted to a polyline approximating this arc within a fixed Hausdorff distance.

The novelty of our approach is that it is hybrid but, unlike [18], the roles of subdivision and curve-tracking are interchanged – curve arcs are traced in the original domain while subdivision is employed in tough cases. Also, note that, the requirement of a complete CAD in most cases can be *relaxed*; see Section 3.5. We start with a "seed point" on a curve arc and trace it in two opposite directions. In each step we examine 8 neighbours of a current pixel and choose the one crossed by the arc. In case of a tie, the pixel is subdivided recursively into 4 parts. Local subdivision *stops* as soon as a certain threshold is reached and all curve arcs appear to leave the pixel in one unique direction. From this point on, the arcs are traced collectively until one of them goes apart. When this happens, we pick out the right arc using the *root isolation* [13]; see Figure 2 (b).

According to our experiences, we can trace the majority of curves without resorting to exact computations even if root separation bounds are very tight. To handle exceptional cases, we switch to more accurate interval methods or increase the arithmetic precision.

2 Preliminaries

Arcs of algebraic curves. For an algebraic curve $\mathcal{C} = \{(x,y) \in \mathbb{R}^2 : f(x,y) = 0\}$ with $f \in \mathbb{Q}[x,y]$, we define its gradient vector as $\nabla f = (f_x, f_y) \in (\mathbb{Q}[x,y])^2$ where $f_x = \frac{\partial f}{\partial x}$ and $f_y = \frac{\partial f}{\partial y}$. A point $\mathbf{p} \in \mathbb{R}^2$ is called *x-critical* if $f(\mathbf{p}) = f_y(\mathbf{p}) = 0$, similarly $\mathbf{p}$ is *y-critical* if $f(\mathbf{p}) = f_x(\mathbf{p}) = 0$ and *singular* if $f(\mathbf{p}) = f_x(\mathbf{p}) = f_y(\mathbf{p}) = 0$. Accordingly, *regular* points are those that are not singular.

We define a *curve arc* as a single connected component of an algebraic curve which has no singular points in the interior bounded by two not necessarily regular end-points. Additionally, an *x-monotone* curve arc is a curve arc that has no x-critical points in the interior.

Interval analysis. We consider only advanced interval analysis (IA) techniques here; please refer to [12] for a concise overview. The First Affine Form (AF1) [14] is defined as: $\hat{x}_{AF1} = x_0 + \sum_{i=1}^{n} x_i\varepsilon_i + x_{n+1}\tilde{\varepsilon}$, where x_i are real coefficients fixed and $\varepsilon_i \in [-1,1]$ represent unknowns. The term $x_{n+1}\tilde{\varepsilon}$ stands for a cumulative error due to approximations after performing non-affine operations, for instance, multiplication. Owing to this feature, the number of terms in AF1, unlike for a classical affine form, does not grow after non-affine operations. Conversion between an interval $[\underline{x}, \overline{x}]$ and an affine form $\hat{x}$ proceeds as follows:

$$\text{Interval} \to \text{AF1:} \quad \hat{x} \;= (\underline{x} + \overline{x})/2 + [(\overline{x} - \underline{x})/2]\varepsilon_k, \quad \tilde{\varepsilon} \equiv 0,$$

$$\text{AF1} \to \text{Interval:} \quad [\underline{x}, \overline{x}] = x_0 + \left(\sum_{i=1}^{n} x_i\varepsilon_i + x_{n+1}\tilde{\varepsilon}\right) \times [-1,1],$$

here k is an index of a new symbolic variable (after each conversion k gets incremented). Arithmetic operations on AF1 are realized as follows:

$$\hat{x} \pm \hat{y} = (x_0 \pm y_0) + \sum\nolimits_{i=1}^{n} (x_i \pm y_i)\varepsilon_i + (x_{n+1} + y_{n+1})\tilde{\varepsilon},$$

$$\hat{x} \cdot \hat{y} = x_0 y_0 + \sum_{i=1}^{n} (x_0 y_i + y_0 x_i)\varepsilon_i + \left(|x_0| y_{n+1} + |y_0| x_{n+1} + \sum_{i=1}^{n+1} |x_i| \sum_{i=1}^{n+1} |y_i| \right) \tilde{\varepsilon}.$$

The Quadratic Form (QF) is an extension of AF1 that adds two new symbolic variables $\varepsilon^+ \in [0,1]$ and $\varepsilon^- \in [-1,0]$ to attenuate the error when an affine form is raised to even power, and a set of square symbolic variables ε_i^2 to capture quadratic errors:

$$\hat{x}_{QF} = x_0 + \sum_{i=1}^{n} x_i \varepsilon_i + x_{n+1}\tilde{\varepsilon} + x_{n+2}\varepsilon^+ + x_{n+3}\varepsilon^- + \sum_{i=1}^{n} x_{i+n+3}\varepsilon_i^2,$$

where $\varepsilon_i^2 \in [0,1]$. For reasons of space we refer to [14] for arithmetic operations on QF.

Another method is the so-called Modified Affine Arithmetic (MAA) [11]. It is more precise than AF1 and QF. We consider the 1D case here as the only relevant to our algorithm. To evaluate a polynomial $f(x)$ of degree d on $[\underline{x}, \overline{x}]$, we denote $x_0 = (\underline{x} + \overline{x})/2$, $x_1 = (\overline{x} - \underline{x})/2$ and $D_i = f^{(i)}(x_0) x_1^i / i!$. The interval $[\underline{F}; \overline{F}]$ is obtained as follows:

$$\underline{F} = D_0 + \sum_{i=1}^{\lceil d/2 \rceil} \left(\min(0, D_{2i}) - |D_{2i-1}| \right), \quad \overline{F} = D_0 + \sum_{i=1}^{\lceil d/2 \rceil} \left(\max(0, D_{2i}) + |D_{2i-1}| \right).$$

The efficient technique to further shrink the interval bounds is by exploiting the *derivative* information [12]. In order to evaluate a polynomial $f(x)$ on $X = [\underline{x}, \overline{x}]$, we first evaluate its derivative f' on X using the *same* interval method. If the derivative does not straddle 0, f is strictly monotone over X, hence the *exact* bounds are obtained as follows:

$$[\underline{F}; \overline{F}] = [f(\underline{x}), f(\overline{x})] \text{ for } f' > 0, \qquad [\underline{F}; \overline{F}] = [f(\overline{x}), f(\underline{x})] \text{ for } f' < 0.$$

The same approach can be applied recursively to compute the bounds for f', f'', etc. Typically, the number of recursive derivatives in use is fixed by a threshold chosen empirically. We rely on all three aforementioned interval methods in our implementation. The AF1 is a default method while QF and MAA are used in tough cases; see Section 3.5.

3 Algorithm

3.1 Overview

We begin with a high-level overview of the algorithm which is a further development of [9]. After a long-term practical experience we have applied a number of optimizations aimed to improve the performance and numerical stability of the algorithm. In its core the algorithm has an 8-way stepping scheme introduced in [5]; see Figure 3 (a). As the evidence of the correctness of our approach we use the notion of a *witness* (sub-)pixel.

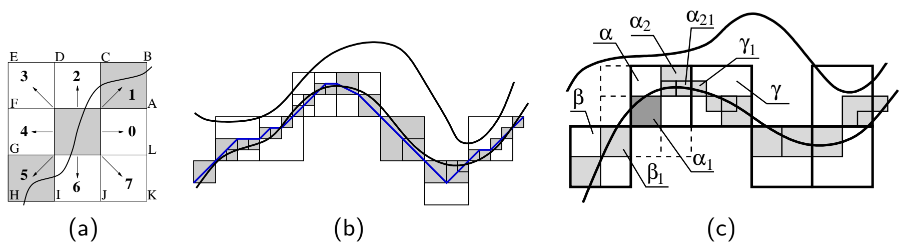


Fig. 3. (a) the 8-pixel neighbourhood with numbered directions, plotted pixels are shaded; (b) adaptive approximation of a curve arc and a polyline connecting witness (sub-)pixels (shaded); (c) more detailed view.

A "witness" (sub-)pixel is a box whose boundaries intersect only twice with an arc to be plotted and do not intersect with any other arc. We implicitly assign a witness (sub-)pixel to each pixel in the curve trace. Then, if we connect the witness (sub-)pixels from one end-point to another by imaginary lines, we obtain a piecewise linear approximation of a curve arc within a *fixed Hausdorff distance*; see Figure 3 (b).

Given a set of x-monotone curve arcs, we process each arc independently. The algorithm picks up a "seed point" on an arc and covers it by a witness (sub-)pixel such that the curve arc leaves it in to different directions. We trace the arc in two directions from the seed point until the end-points. In each step we examine an 8-pixel neighbourhood of a current pixel; see Figure 3 (a). If its boundaries are crossed only twice by the arc, we say that the *neighbourhood test* succeeds (see Section 3.2). In this case, we step to the next pixel using the direction returned by the test. Otherwise, there are two possibilities: **1.** the current pixel is *itself* a witness (sub-)pixel: we subdivide it recursively into 4 even parts until the test succeeds for one of its sub-pixels or we reach the maximal subdivision depth.[1] **2.** the current pixel has an assigned witness (sub-)pixel: we proceed with tracing from this witness (sub-)pixel. In both situations tracing at a sub-pixel level is continued until the pixel boundary is met and we step to the *next* pixel. The last sub-pixel we encounter becomes a *witness* of a newly found pixel. Details on the subdivision are given in Section 3.4 in terms of algorithm's pseudocode.

Suppose we start with a witness (sub-)pixel marked by $\boldsymbol{\alpha_1}$ in Figure 3 (c), its 8-pixel surrounding box is depicted with dashed lines. The pixel it belongs to, namely α, is added to the curve trace. Assume we choose the direction 1 from $\boldsymbol{\alpha_1}$ and proceed to the next sub-pixel $\boldsymbol{\alpha_2}$. The test fails for $\boldsymbol{\alpha_2}$. Thus, we subdivide it into 4 pieces, one of them ($\boldsymbol{\alpha_{21}}$) intersecting the arc is taken.[2] We resume tracing from $\boldsymbol{\alpha_{21}}$, its neighbourhood test succeeds and we find the next

[1] In this situation the algorithm restarts with increased precision; see Section 3.5. We define a subdivision depth k as the number of pixel subdivisions, that is, a pixel consists of 4^k depth-k sub-pixels.

[2] To choose such a sub-pixel we evaluate a polynomial at the corners of $\boldsymbol{\alpha_2}$ since we know that there is only one curve arc going through it.

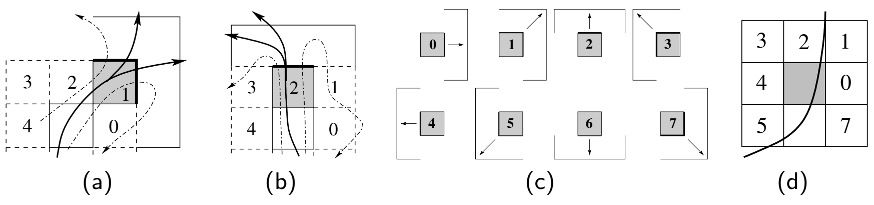


Fig. 4. (a) and (b): Depending on the incoming direction, the curve can leave the shaded pixel's neighbourhood along the boundary depicted with solid lines. Dash-dotted curves show prohibited configurations; (c) boundaries to be checked for all incoming directions; (d) the arc passes exactly between two pixels

"witness" (sub-)pixel (γ_1), its corresponding pixel (γ) is added to the curve trace. The process terminates by reaching one of the arc's end-points. Then, we trace towards another end-point from a saved sub-pixel β_1.

In Section 3.3 we present a technique to stop the local subdivision earlier even if the number of arcs in the pixel neighourhood is not one. Finally, in Section 3.5 we discuss the numerical accuracy issues of the algorithm and the termination criteria.

3.2 Counting the number of curve arcs

In this section we discuss the neighbourhood test. Due to x-monotony constraint, there can be no closed curve components inside a box enclosing an 8-pixel neighbourhood. Hence, the boundary intersection test suffices to ensure that only one curve arc passes through this box. We rely on the following consequence of Rolle's theorem:

Corollary 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 $[a; b]$.*

First, we sketch the basic version of the neighbourhood test, and then refine it according to some heuristic observations. The test **succeeds** if the procedure CHECK_SEGMENT given below returns "one root" for exactly 2 out of 9 sub-segments $AB, BC, \ldots, LA$, and "no roots" for the remaining ones; see Figure 3 (a). The test **fails** for all other combinations, resulting in pixel subdivision.

```
procedure CHECK_SEGMENT([a, b] : Interval, f : Polynomial)
    if $0 \notin \{[\underline{F}, \overline{F}] = f([a, b])\}$ then      ▷ evaluate $f$ at $[a, b]$ and test for 0 inclusion
        return "no roots"                                  ▷ interval does not include 0 $\Rightarrow$ no roots
    if sign(f(a)) = sign(f(b)) then                        ▷ test for a sign change at end-points
        return "uncertain"                                 ▷ no sign change $\Rightarrow$ even number of roots
    loop
        if $0 \notin \{[\underline{F}, \overline{F}] = f'([a, b])\}$ then                ▷ check interval for 0 inclusion
            return "one root"                              ▷ $f'$ does not straddle 0 $\Rightarrow$ one root
        Split [a, b] in 2 halves, let $[x, y]^+$ be the one at which f(x) has no sign change
        if $0 \in \{[\underline{F}, \overline{F}] = f([x, y]^+)\}$ then               ▷ check interval for 0 inclusion
            return "uncertain"                             ▷ $f$ straddles 0 $\Rightarrow$ nothing can be said for sure
```

```
12:         Let $[x, y]^-$ be the half at which $f(x)$ has a sign change
13:         $[a, b] \leftarrow [x, y]^-$                    ▷ restart with the refined interval
14:     end loop
15: end procedure
```

The search space can be reduced because we know the direction of an incoming branch. In Figure 4 (a), the algorithm steps to the shaded pixel in a direction "1" relative to the previous pixel. Hence, a curve must cross a part of its boundary marked with thick lines. The curve can leave its neighbourhood along a part of the boundary indicated by solid lines. Configurations shown by dash-dotted curves are impossible due to x-monotonicity. In Figure 4 (b) we enter the shaded pixel in direction "2", dash-dotted curves are again prohibited since, otherwise, the 8-pixel neighbourhood of a previous pixel (the one we came from) would be crossed more than twice by the curve resulting in a subdivision. Figure 4 (c) lists parts of the boundary being checked for all possible incoming directions. Thus, the neighbourhood test succeeds if CHECK_SEGMENT returns "one root" for exactly one of the "enabled" sub-segments respecting the incoming direction (and "no roots" for the remaining enabled ones).

In Figure 4 (d) the arc passes exactly between two pixels. In this situation we are free to choose any out of two "hit" directions, that is, 1 or 2 in the figure. We use additional exact zero testing to detect this. A grid perturbation technique presented in the next section makes this situation almost improbable.

3.3 Identifying and tracing closely located arcs, grid perturbation

To deal with tightly packed curve arcs, we modified the neighbourhood test in a way that we allow a pixel to pass this test once a new direction can uniquely be determined *even* if the number of arcs over a sub-segment on the boundary is more than one. We will refer to this as tracing in *coincide mode*. In other words, the test reports the coincide mode if CHECK_SEGMENT returns "uncertain" for one sub-segment and "no roots" for all the rest being checked (as before, the test fails for all other combinations leading to subdivision). From this point on, the arcs are traced *collectively* until one of them goes apart. At this position we *exit* the coincide mode by picking up the right arc using *root isolation* (see Figure 5)(a, b), and *resume* tracing with subdivision. The same applies to seed points – it is desireable to start already in coincide mode if the arcs at this position are too close. Typically, we enable the coincide mode by reaching a certain subdivision depth which is an evidence for tightly packed curve arcs (depth 5 works well).

Observe that, we need to know the *direction of motion* (left or right) to be able to isolate roots on exiting the coincide mode. Yet, it is absolutely unclear if we step in a vertical direction only. Then, we compute the direction using the tangent vector $(t_x, t_y) = (\partial f/\partial x, \partial f/\partial y)$ evaluated at the "seed" point.

In Figure 5 (c) the arcs are separated by the pixel grid thereby prohibiting the coincide mode: for example, the curve $f(x, y) = y^2 - 10^{-12}$ (two horizontal lines) when the grid origin is at $(0, 0)$. A simple remedy against this is to shift

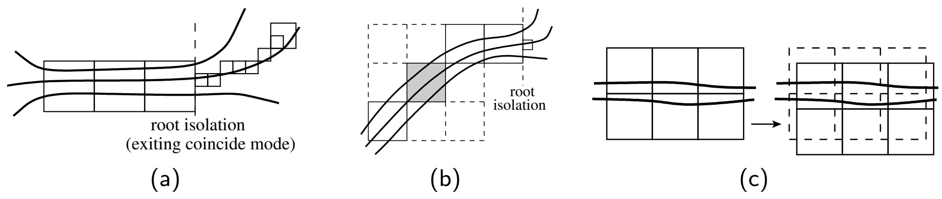


Fig. 5. (a) Tracing arcs collectively in "coincide mode"; (b) diagonal "coincide mode": the neighbourhood test succeeds for the shaded pixel even though the number of arcs crossing the boundary is more the one; (c) pixel grid perturbation.

the grid origin by an arbitrary fraction of a pixel (grid pertrubation) from its initial position before the algorithm starts.

Remark that, the coincide mode *violates* our definition of a witness (sub-)pixel. Nevertheless, the approximation is correct because an arc is guaranteed to lie within an 8-pixel neighbourhood even though this neighbourhood does not necessarily contain a single arc.

3.4 Pseudocode

We present the algorithm's pseudocode in terms of a procedure STEP computing the next pixel in a curve trace. The STEP gets the current pixel, its witness (sub-)pixel and a direction as parameters, and returns the next pixel, its witness and a new direction relative to the previous pixel. It is applied until tracing reaches one of the end-points, for the sake of simplicity, this test is not shown in the pseudocode. The neighbourhood test is performed by the test_pix, it returns a new direction in case the test succeeds (and optionally sets a global flag coincide_mode if tracing can be continued in "coincide mode") and returns -1 otherwise. The step_pix advances the pixel coordinates with respect to a given direction.

```
procedure STEP(pix: Pixel, witness: Pixel, d: Direction)
    new_d : Direction ← test_pix(pix, d)            ▷ check the pixel's neighbourhood
    if new_d ≠ −1 then
        p:Pixel ← step_pix(pix, new_d)               ▷ step to the next pixel and return it
        return {p, p, new_d}          ▷ a pixel, its witness sub-pixel and a new direction
    if coincide_mode then             ▷ curve branches go apart ⇒ need new seed point
        coincide_mode = false                    ▷ we stop tracing "coincide" branches
        {p, new_d} ← get_seed_point(pix, d)      ▷ get a new seed point and a direction
        return {get_pixel(p), p, new_d}           ▷ a pixel, its witness and a new direction
    if witness = pix then     ▷ witness sub-pixel is a pixel itself ⇒ perform subdivision
        {p: Pixel, new_d} ← SUBDIVIDE(pix, d)
    else                      ▷ otherwise continue tracing from the witness sub-pixel
        {p: Pixel, new_d} ← {witness, d}
    while get_pixel(p) = pix do         ▷ iterate until we step over the pixel's boundary
        new_d ← test_pix(p, new_d)                  ▷ check the pixel's neighbourhood
        if new_d = −1 then
            {p,new_d} ← SUBDIVIDE(p, new_d)
        p ← advance_pixel(p, new_d)
    end while
    return {get_pixel(p), p, new_d} ▷ a pixel, its witness sub-pixel and a new direction
end procedure

procedure SUBDIVIDE(pix: Pixel, d: Direction)
    sub_p: Pixel ← get_subpixel(pix, d)          ▷ get one of the four sub-pixels of 'pix'
    new_d: Direction ← test_pix(sub_p, d)
    if new_d = $-1$ then
        return SUBDIVIDE(sub_p, d)               ▷ neighbourhood test failed ⇒ subdivide
    return {sub_p, new_d}
end procedure
```

3.5 Addressing accuracy problems and relaxing CAD assumption

Numerical instability is a formidable problem in geometric algorithms because, on one hand, using exact arithmetic is unacceptable in all cases while, on the other hand, an algorithm must handle all degeneracies adequately if one strives for an exact approach. To deal with this, we use a *mixed-precision* method, that is, high precision computations are applied only when necessary. The following situations indicate the lack of numerical accuracy:

– reached the maximal subdivision depth (typically 12). This prevents the algorithm from being "stuck" at one pixel when the subdivision depth is too high. In this case it is reasonable to restart with increased precision;
– subpixel size is too small, that is, beyond the accuracy of a number type;
– no subpixel intersecting a curve arc has been found during subdivision. This indicates that the polynomial evaluation is not trustful anymore.

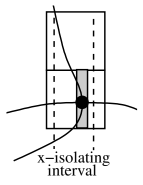
When any of this occurs, we restart the algorithm using a more elaborate IA method (QF and then MAA; see Section 2). If this does not work either, the arithmetic precision is increased according to a three-level model given below.

Level 0: all operations are carried out in double-precision floating-point. Polynomial coefficients are divided by the median scalar factor and converted to intervals of doubles. If the evaluation with such a polynomial results in an interval including zero, then the quantity is *reevaluated* using exact rational arithmetic.

Level 1: all operations are performed in bigfloat arithmetic. Accordingly, polynomial coefficients are converted to bigfloats. As before, a quantity is reevaluated with rational arithmetic if the evaluation with bigfloats is not trustful.

Level 2: exact rational arithmetic is used throughout all computations. At this level arbitrary subdivision depths are allowed and no constraints are imposed on a sub-pixel size.

Recalling the introduction, unless we deal with isolated singularities, our algorithm can proceed having only x-coordinates of x-critical points of $\mathcal{C}$ (resultant of f and f_y). Hence, an expensive **lifting phase** of a symbolic CAD algorithm can be avoided. However, since y-coordinates of end-points are not explicitly given, we



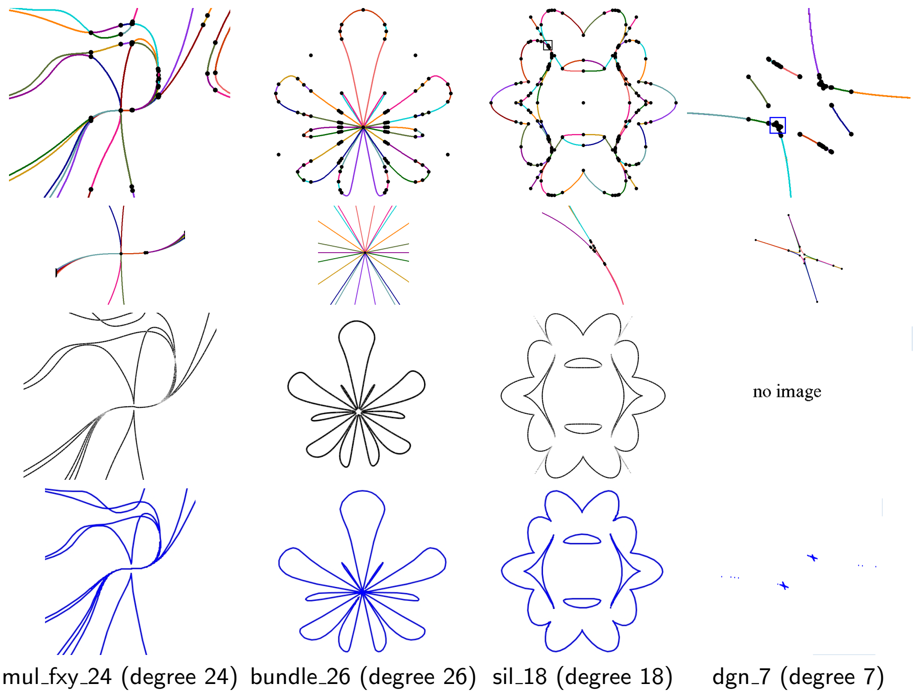


Fig. 6. First two rows: curves rendered using our method with zooming at singularities; **3rd row:** plots produced in Axel; **4th row:** plots produced in Maple. Curve degrees are given w.r.t. y-variable.

exploit the x-monotony to decide where to stop the arc tracing.
Namely, the tracing terminates as soon as it reaches a (sub-)pixel containing an x-isolating interval of an end-point, and there exists such a box inside this (sub-)pixel that the curve crosses its vertical boundaries only. This last condition is necessary to prevent a premature stopping alarm for arcs with a decent slope; see Figure to the right.

4 Results and conclusion

Our algorithm was implemented in the CGAL (Computational Geometry Algorithms Library, `www.cgal.org`) as part of *Curved_kernel_via_analysis_2* package [2]. The CGAL's development follows a *generic programming paradigm*. This enabled us to parameterize our algorithm by a number type to be able to increase the arithmetic precision without altering the implementation.

We tested our algorithm on 2.2 GHz Intel Core2 Duo processor with 4 MB L2 cache and 2 GB RAM under 32-bit Linux platform. Multi-precision number types were provided by CORE with GMP 4.3.1 support.[1] The CAD of an algebraic

[1] CORE: `http://cs.nyu.edu/exact`; GMP: `http://gmplib.org`.

Table 1. Running times in seconds. **Analyze:** the time to compute the CAD; **Render:** visualization using our method; and visualization using Axel and Maple 13 respectively.

Curve	Analyze	Render	Axel	Maple	Curve	Analyze	Render	Axel	Maple
mul_fxy_24	22.2	4.1	2.93	3.35	bundle_26	60.5	2.8	2.11	1.88
sil_18	35.2	6.4	265	2.22	dgn_7	0.61	0.82	timeout	3.1

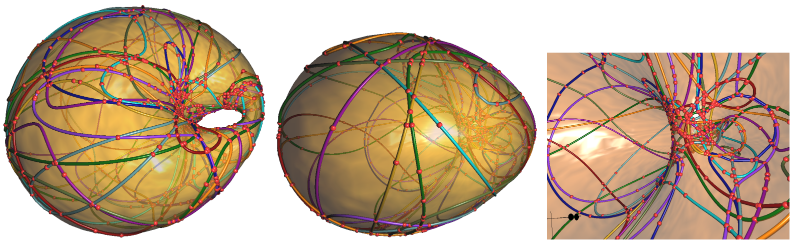


Fig. 7. Rendering curves on a surface of Dupin Cyclide (about 64000 rendered points)

curve was computed using [7]. We compared our visualization with the ones provided by Axel and Maple 13 software.[1] Axel implements the algorithm given in [1]. Due to the lack of implementation of the exact approach, we compared with a subdivision method. We varied the accuracy parameter ε from $5 \cdot 10^{-5}$ to 10^{-8} depending on the curve. The "feature size" (asr) was set to 10^{-2}. In Maple we used implicitplot method with $numpoints = 10^5$. Figure 6 depicts the curves[2] plotted with our method, Axel and Maple respectively. Notice the visual artifacts nearby singularities. Moreover, in contrast to our approach, the algorithms from Axel and Maple cannot visualize the arcs selectively. Table 1 summarizes the running times. Rendering the curve dgn_7 in Axel took more than 15 mins and was aborted, this clearly shows the advantages of using the coincide mode.

Figure 7 depicts an intersection of a Dupin Cyclide with 10 algebraic surfaces of degree 3 computed using [3]. Resulting arrangement of degree 6 algebraic curves was rendered in tiles with varying resolutions and mapped onto the Dupin Cyclide using rational parameterization. Visualization took 41 seconds on our machine.

To conclude, we have identified, that the interplay of a symbolic precomputation and a numerical algorithm delivers the best performance in practice, because, once the exact solution of $f = 0 \wedge f_y = 0$ is computed, rendering proceeds fast for *any* resolution. In contrast, the subdivision methods have to recompute the topological events for every new domain $\mathcal{D}$ due to the lack of "global" information. Moreover, they can often report a wrong topology if ε is not chosen small enough. Also, as mentioned in Section 3.5, the amount of symbolic computations required by our algorithm can be reduced substantially. Yet,

[1] Axel: `http://axel.inria.fr`; Maple: `www.maplesoft.com`.
[2] Visit our curve gallery at: `http://exacus.mpi-inf.mpg.de/gallery.html`

the current implementation is still based on a complete CAD, thus we have not been able to evaluate this in practice which is an object of future research.

References

1. L. Alberti, B. Mourrain: "Visualisation of Implicit Algebraic Curves". *Computer Graphics and Applications, Pacific Conference on* (2007) 303–312.
2. E. Berberich, P. Emeliyanenko: *CGAL's Curved Kernel via Analysis.* Technical Report ACS-TR-123203-04, Algorithms for Complex Shapes, 2008.
3. E. Berberich, M. Kerber: "Exact Arrangements on Tori and Dupin Cyclides". In: E. Haines, M. McGuire (eds.) *SPM'08.* ACM, Stony Brook, USA, 2008 59–66.
4. M. Burr, S. W. Choi, B. Galehouse, C. K. Yap: "Complete subdivision algorithms, II: isotopic meshing of singular algebraic curves". In: *ISSAC '08.* ACM, New York, NY, USA, 2008 87–94.
5. R. Chandler: "A tracking algorithm for implicitly defined curves". *IEEE Computer Graphics and Applications* **8** (1988).
6. J. Cheng, S. Lazard, L. Pe naranda, M. Pouget, F. Rouillier, E. Tsigaridas: "On the topology of planar algebraic curves". In: *SCG '09.* ACM, New York, NY, USA, 2009 361–370.
7. A. Eigenwillig, M. Kerber, N. Wolpert: "Fast and exact geometric analysis of real algebraic plane curves". In: *ISSAC '07.* ACM, New York, NY, USA, 2007 151–158.
8. G. Elber, M.-S. Kim: "Geometric constraint solver using multivariate rational spline functions". In: *SMA '01.* ACM, New York, NY, USA, 2001 1–10.
9. P. Emeliyanenko: *Visualization of Points and Segments of Real Algebraic Plane Curves.* Master's thesis, Universität des Saarlandes, 2007.
10. P. Emeliyanenko, M. Kerber: "Visualizing and exploring planar algebraic arrangements: a web application". In: *SCG '08.* ACM, New York, NY, USA, 2008 224–225.
11. I. V. Huahao Shou, Ralph Martin, et al.: "Affine arithmetic in matrix form for polynomial evaluation and algebraic curve drawing". *Progress in Natural Science* **12 (1)** (2002) 77–81.
12. R. Martin, H. Shou, I. Voiculescu, A. Bowyer, G. Wang: "Comparison of interval methods for plotting algebraic curves". *Comput. Aided Geom. Des.* **19** (2002) 553–587.
13. K. Mehlhorn, M. Sagraloff: "A Deterministic Descartes Algorithm for Real Polynomials", 2009. Accepted for ISSAC 2009.
14. F. Messine: "Extensions of Affine Arithmetic: Application to Unconstrained Global Optimization". *Journal of Universal Computer Science* **8** (2002) 992–1015.
15. T. Möller, R. Yagel: *Efficient Rasterization of Implicit Functions.* Tech. rep., Department of Computer and Information Science, Ohio State University, 1995.
16. J. Morgado, A. Gomes.: "A Derivative-Free Tracking Algorithm for Implicit Curves with Singularities." In: *ICCSA*, 2004 221–228.
17. S. Plantinga, G. Vegter: "Isotopic approximation of implicit curves and surfaces". In: *SGP '04.* ACM, New York, NY, USA, 2004 245–254.
18. H. Ratschek, J. G. Rokne: "SCCI-hybrid Methods for 2d Curve Tracing". *Int. J. Image Graphics* **5** (2005) 447–480.
19. R. Seidel, N. Wolpert: "On the exact computation of the topology of real algebraic curves". In: *SCG '05.* ACM, New York, NY, USA, 2005 107–115.
20. Z. S. Yu, Y. Z. Cai, M. J. Oh, et al.: "An Efficient Method for Tracing Planar Implicit Curves". *Journal of Zhejiang University - Science A* **7** (2006) 1115–1123.