www.cadanda.com
Computer-Aided Design AND APPLICATIONS™

C2 Approximation of Planar Curvilinear Profiles by Cubic B-Splines

Martin Held[1] and Dominik Kaaser[2]

Universität Salzburg, FB Computerwissenschaften, A-5020 Salzburg, Austria
[1]held@cosy.sbg.ac.at [2]dominik@kaaser.at

ABSTRACT

We introduce an efficient algorithm for computing C^2 approximations of a set of planar curvilinear profiles by means of uniform cubic B-splines. The resulting approximations are guaranteed to lie within a user-specified tolerance of the input profiles, with asymmetric and even one-sided tolerances being supported by our algorithm. Furthermore the input topology is reflected by our approximation. The input profiles may consist of straight-line segments and circular arcs.

Extensive experiments with synthetic and real-world data sets show that our algorithm works very nicely in practice. In particular, it supports the approximation of profiles with up to 10000 input segments and arcs in less than ten seconds on a standard PC. We use a top-down fitting scheme to generate a (hopefully) small number of B-spline segments. This allows our approximation algorithm to run in $O(n \log n)$ time and $O(n)$ space, where n denotes the number of input segments and arcs that form the profiles.

Keywords: approximation, B-spline, C2-continuity, tolerance zone, Voronoi diagram.

1. INTRODUCTION

1.1. Motivation and Prior Work

Smooth approximations of planar curvilinear profiles that consist of straight-line segments and circular arcs are useful both from a theoretical and a practical point of view. Our goal is to represent one or more open or closed profiles by a comparatively small number of higher-order primitives within a user-specified tolerance, thereby achieving C^2 continuity. Applications like the simplification of geographic entities (such as islands within a lake or river) in a geographic information system also require us to maintain the topology of the input. That is, the approximation curves generated are not allowed to self-intersect or to intersect each other, and if a profile P_1 lies within another closed profile P_2 then the approximation of P_1 shall also lie within the approximation of P_2. (Actually, in VLSI applications we may even be asked to guarantee a minimum clearance among the approximation curves in order to prevent undesired connectivity.) For certain applications like the approximation of a tool path the user may wish to specify a tolerance that is non-symmetric or even one-sided [11].

However, we are not aware of algorithms that allow the simultaneous approximation of a set of planar profiles by C^2 curves such that the user has full control over the approximation tolerance and such that the input topology is preserved. Of course, many algorithms for the G^2- or C^2-approximation of one curve with limited control over the tolerance are known. For instance, Behar et al. [2] present an algorithm for a G^2-smoothing of an open polygon by cubic A-splines. Curvature-continuous approximations of an offset curve are described in [1,3].

If one forfeits G^2 continuity then the work by Heimlich and Held [6] is applicable: It allows the approximation of multiple closed polygons by G^1 curves consisting of either biarcs or cubic Bézier curves such that the input topology is preserved, while retaining full control over the approximation tolerance. Recent algorithms by Drysdale et al. [5] and Mayer and Pisinger [12] allow the approximation of one polygon by the minimum number of tangent-continuous circular arcs within a tolerance region, which needs to be present as part of the input. Neither of these latter two approach seems to be extensible to G^2 continuity, though.

1.2. Results Achieved

We describe an algorithm to compute C^2 approximations of a set of planar curvilinear profiles that

Computer-Aided Design & Applications, 11(2), 2013, 206–219, http://dx.doi.org/10.1080/16864360.2014.846092

Taylor & Francis
Taylor & Francis Group

consist of straight-line segments and circular arcs. As in [6], the algorithmic vehicle for guaranteeing a user-specified maximum (Hausdorff) distance between the input profiles and the approximation curves, for achieving the simplicity of the approximation curves, and for maintaining the input topology is the use of a tolerance zone: Every profile (and its approximation curve) is contained in its own portion of the tolerance zone whose width is chosen such that the user-specified bound on the Hausdorff distance is obeyed. To this goal we extend the definition of a tolerance zone of [6] to open curves. To compute the boundary of the tolerance zone, we use traditional (Voronoi-based) offsetting.

We employ uniform cubic B-splines as approximation primitives to achieve C^2 continuity. Cubic B-spline segments are defined upon four control vertices each. Our B-spline curves use so-called approximation nodes ("a-nodes") as control vertices. These a-nodes are generated within the tolerance zone and placed on its medial line. De Boor's subdivision algorithm [4] is used for approximate (but conservative) tests for containment of a B-spline segment in the tolerance zone.

Since consecutive B-spline segments share a common set of control vertices rather than only one a-node , standard greedy-like schemes [6,9] for fitting "long"primitives to the input cannot be adapted easily to B-splines. To overcome this problem we propose a top-down approach that refines an initially coarse approximation: We start with one coarse B-spline and refine it by successively adding new a-nodes if the B-spline is not contained entirely in the tolerance zone, thus splitting the coarse B-spline into smaller ones which, together, provide a better fit to the input.

We implemented our algorithm in C++, based on the Voronoi package VRONI/ArcVRONI [7,10]. Extensive experiments with synthetic and real-world data sets show that our algorithm works very nicely in practice. Our top-down approach yields a smaller number of B-spline segments than the standard greedy-like fitting schemes. It is also significantly faster, processing profiles with up to 10000 input segments and arcs in less than ten seconds on a standard PC. From a theoretical point of view, our approximation algorithm runs in $O(n \log n)$ time and requires $O(n)$ space, where n denotes the total number of straight-line segments and circular arcs that form the input profiles. Due to the use of Voronoi-based computations, our approximation algorithm is completely immune to input noise: As the approximation tolerance is gradually increased, the number of B-spline segments used decreases gradually.

Figure 1 shows a sample input together with its Voronoi diagram (in green) and the boundary curves of the tolerance zone (in blue and magenta); the actual B-spline approximation computed by our algorithm is shown in orange.

Since VRONI/ArcVRONI was developed and is being maintained by the first author's research group, it was natural to base our implementation on VRONI/ArcVRONI. We note, though, that we do not exploit any specific feature of VRONI/ArcVRONI besides its capability to compute Voronoi diagrams and offset curves. Thus, any code for computing Voronoi diagrams of straight-line segments and circular arcs would do for replicating our work.

2. TOLERANCE ZONE

2.1. Review of Basics

The input $\mathcal{P}$ for our algorithm is formed by a set of *curvilinear profiles*, where each profile is a planar curve formed by a sequence of primitives such that the $(i+1)$-st primitive starts in the end point of the i-th primitive. We allow straight-line segments and circular arcs as input primitives. All profiles need to be simple curves that are pairwise disjoint. (A curve is "simple" if it has no self-intersections.) In order to support a consistent notion of (local) sidedness we require the orientation of the line segments and circular arcs to be consistent.

The straight-line segments, circular arcs and vertices of $\mathcal{P}$ are called "sites" of $\mathcal{P}$. As in [6] we make extensive use of the Voronoi diagram, $\mathcal{VD}(\mathcal{P})$, of $\mathcal{P}$. Very roughly, the Voronoi diagram of $\mathcal{P}$ partitions the plane into cells, so-called Voronoi cells, such that

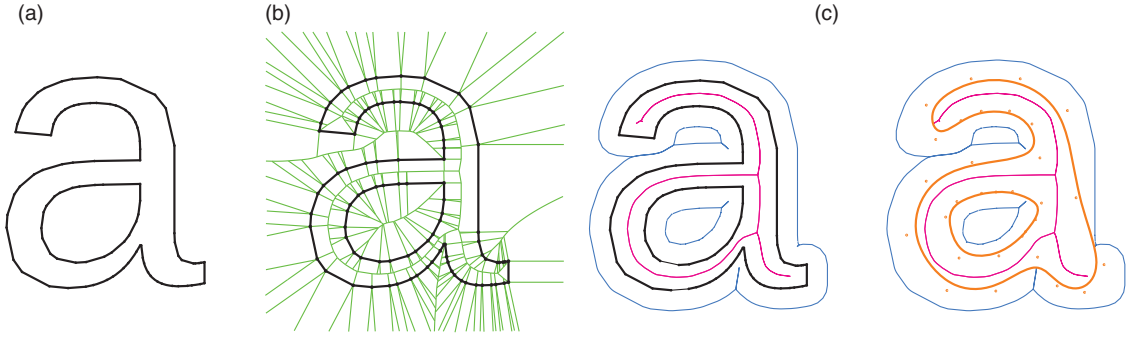

Fig. 1: From left to right: (a) The input consists of the outline of the letter "a", described by two polygons, (b) the Voronoi diagram (green) of the input, (c) the tolerance zone boundaries (blue, magenta) and the approximation (orange) within the tolerance zone.

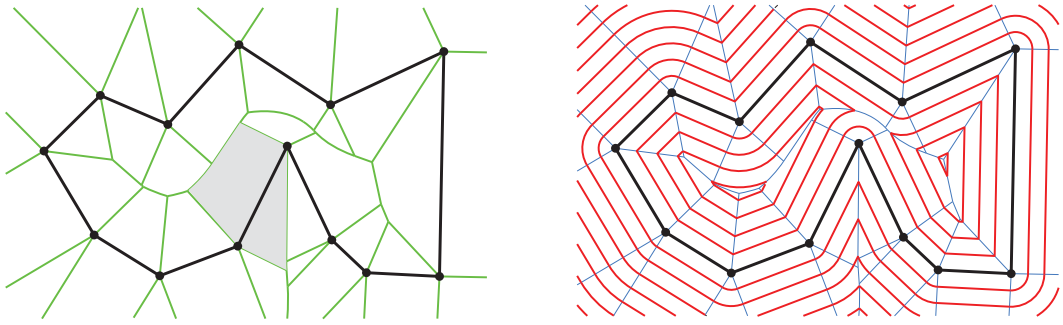

Fig. 2: Left: Voronoi diagram (in green) and the Voronoi cell of one straight-line segment (shaded in light grey); Right: Family of offset curves (in red).

each cell is defined by a site of $\mathcal{P}$ and consists of all points of the plane closer to that site than to any other site. See Fig. 2, which depicts (a finite portion of) the Voronoi diagram (in green of a polygon (in black); the Voronoi cell of one straight-line segment of $\mathcal{P}$ is shaded in light grey. We denote the Voronoi cell of a site s of $\mathcal{P}$ by $\mathcal{VC}(\mathcal{P}, s)$.

Every Voronoi cell is bounded by Voronoi edges. Points on Voronoi edges are equidistant to two sites of $\mathcal{P}$. Voronoi nodes, which are formed by the intersection of Voronoi edges, are equidistant to three (or more) sites. A Voronoi edge defined by two straight-line segments of $\mathcal{P}$ lies on the angular bisector of these two line segments. (Every Voronoi edge is a portion of a conic, i.e., a straight-line segment, parabolic arc, hyperbolic arc, or elliptic arc.) The theory of Voronoi diagrams tells us the Voronoi diagram $\mathcal{VD}(\mathcal{P})$ is a planar graph that consist of $O(n)$ edges and nodes if $\mathcal{P}$ is formed by n line segments and circular arcs.

The Voronoi diagram of a set of n (non-intersecting) straight-line segments and circular arcs can be computed in $O(n \log n)$ time and $O(n)$ space, both in the worst case [14] and in the expected case [10]: Yap [14] employs a divide-and-conquer algorithm, while [10] relies on a randomized incremental construction of the Voronoi diagram. That is, Held and Huber [10] start with a (trivial) Voronoi diagram of two sites and then insert sites, one after the other and in a randomized order, and appropriately update the Voronoi diagram for each site inserted. Reference is given to [8] for an up-to-date survey on the computation and application of Voronoi diagrams of straight-line segments and circular arcs.

An offset with offset distance d of $\mathcal{P}$ is the set of curves traced out by the center of a disk with radius d that rolls along $\mathcal{P}$. (For oriented curves it is also common to distinguish between left and right offset.) Once $\mathcal{VD}(\mathcal{P})$ is known, a family of offset curves of $\mathcal{P}$ can be computed easily. The relation between the Voronoi diagram and offset curves of $\mathcal{P}$ can be recognized by observing how the offset curves change when the offset distance is modified: Imagine that one starts with offset distance $d = 0$ and continuously increases d. Then the endpoints of the (moving) offset primitives trace out the Voronoi diagram, see Fig. 2. Thus, by intersecting offset primitives with bisectors of the appropriate Voronoi cell one can compute all offset curves of $\mathcal{P}$ for a specific offset distance d in $O(n)$ time. We refer to [8] for an explanation of Voronoi-based offsetting.

2.2. Definition of the Tolerance Zone

Heimlich and Held [6] define a signed distance $d_s(\mathcal{P}, p)$ between the input $\mathcal{P}$ and a query point p that depends on the nesting level of the polygons. (Recall that their algorithm is restricted to closed polygons.) That signed distance is used to define the tolerance zone of $\mathcal{P}$ as the set of points p whose signed distance to $\mathcal{P}$ is lower-bounded by the left tolerance d_l and upper-bounded by the right tolerance d_r. Heimlich and Held then go on to select individual tolerance bands within the tolerance zone.

Clearly, their approach does not extend to our setting as our profiles need not form closed curves. Thus, we lack the notion of a nesting level. In order to retain the benefits of their approach we base our tolerance zones on signed distances to the individual sites. Specifying the meaning of "left" and "right" relative to an individual line segment or circular arc is a bit tricky for points that are far away from that site. Fortunately, we can restrict our attention to the *cone of influence* of a site. The *cone of influence*, $CI(s)$, of

- a straight-line segment s is the closure of the strip bounded by the normals through its endpoints;
- a circular arc s is the closure of the cone bounded by the pair of rays originating in the arc's center and extending through its endpoints;
- a point s is the entire plane.

Then, the signed distance $d_s(s, p)$ of a point $p \in CI(s)$ to an oriented straight-line segment s of $\mathcal{P}$ is given by the standard (Euclidean) distance of p to l, multiplied by -1 if p is on the left side of the supporting line of s, see Fig. 3. The signed distance of a point p to a circular arc s of $\mathcal{P}$ is defined analogously. For the

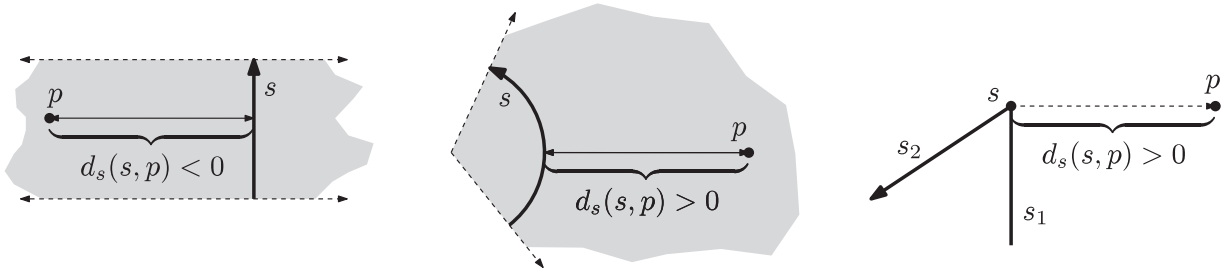

Fig. 3: Signed distances to a straight-line segment, circular arc, and a vertex; the cones of influence of the line segment and the arc are indicated by light grey.

distance of a point p to a vertex s of $\mathcal{P}$ we take the standard distance between p and s, and multiply it by -1 if the ray from s to p is locally on the left side of s_1 and s_2, where s_1 and s_2 are the sites of $\mathcal{P}$ that share s as a common vertex.

We are now ready to define the tolerance zone of a site s of $\mathcal{P}$, see Fig. 4:

$$\mathcal{TZ}_{site}(s, \mathcal{P}, d_l, d_r) := \{p \in \mathcal{VC}(\mathcal{P}, s) : d_l < d_s(s, p) < d_r\}.$$

We note $\mathcal{TZ}_{\text{site}}(s, \mathcal{P}, d_l, d_r)$ is well-defined since the Voronoi cell of every site of $\mathcal{P}$ is contained in its cone of influence [19] and, thus, our definition of a signed distance is applicable. For open profiles we take only inner vertices into account. That is, we omit the half-disk caps around the start and end vertices that occur in traditional offset curves. The resulting tolerance zones of vertices form disk sectors or ring sectors, depending on whether the tolerances are one-sided and disjoint from the input. The tolerance zone of $\mathcal{P}$ is defined as the union of all tolerance zones of all sites:

$$\mathcal{TZ}(\mathcal{P}, d_l, d_r) := \bigcup_{s \in \mathcal{P}} \mathcal{TZ}_{\text{site}}(s, \mathcal{P}, d_l, d_r).$$

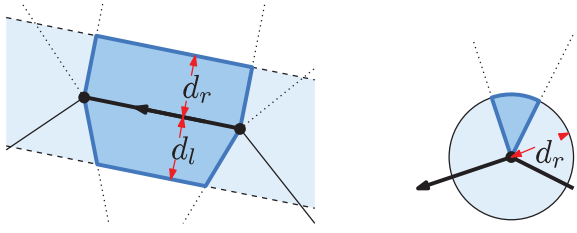

Fig. 4: The tolerance zone of a straight line segment (left) and a vertex (right).

2.3. Algorithms to Compute the Tolerance Zone Boundary

Our algorithm for computing the boundary of the tolerance zone $\mathcal{TZ}(\mathcal{P}, d_l, d_r)$ runs in three stages and returns a list of vertices that define the tolerance zone. We presuppose the existence of the Voronoi diagram of the input profiles $\mathcal{P}$, which we compute using VRONI/ArcVRONI [7,10].

The *clearance radius* of a point p relative to $\mathcal{P}$ is the radius of the largest disk centered at p that does not contain any site of $\mathcal{P}$ in its interior. We take advantage of the fact that VRONI/ArcVRONI assigns a clearance-based parameterization $f : [t_{\min}, t_{\max}] \to \mathbb{R}^2$ to every Voronoi edge e, where $t_{\min}$ is the minimum and $t_{\max}$ is the maximum clearance of points of e. Hence, the coordinates of a point p of e with clearance t can be obtained by evaluating f: We have $p = f(t)$, and the minimum distance of p from $\mathcal{P}$ is given by t. (VRONI/ArcVRONI splits every conic Voronoi arc at its apex, such that the clearance is guaranteed to increase or decrease strictly when traversing a Voronoi edge.)

Although the actual approximation algorithm processes the input profiles sequentially one after the other, the computation of the Voronoi diagram needs to be carried out on the entire input $\mathcal{P}$. (Otherwise, we could not guarantee the disjointness of the tolerance zones of individual profiles!) The following three stages compute one side of the boundary of the tolerance zone for a profile P of $\mathcal{P}$.

Stage 1: Collect Nodes of Voronoi Cells. In the first stage we add all nodes of Voronoi cells defined by sites of the current profile P to a list. We start in the first input vertex of P and its corresponding Voronoi cell and run along the Voronoi edges in the appropriate direction, adding one node after another to a list until we return to a node located directly on $\mathcal{P}$. Such nodes can be identified by their clearance parameters, as nodes located directly on $\mathcal{P}$ are guaranteed to have a clearance equal to zero. We continue this procedure with the next Voronoi cell until we reach the last vertex of P. The resulting list forms a path given by consecutive nodes over the Voronoi diagram starting in the first and ending in the last vertex of P. (Of course, for a closed profile this means that we have returned to the vertex from which we started.) The manner in which we constructed this list implies that every pair of consecutive nodes in this list is connected by a Voronoi edge.

This algorithm is detailed in pseudo-code in Alg. 1. It takes the list of profile vertices and a flag indicating the side for the computation with respect to

input : list of vertices p of a profile, side lr ("left" or "right")
output: list of Voronoi nodes

```
edge ← getVoronoiDiagramEdge(site(p0, p1)) ; // use VRONI to obtain a pointer to VD-cell of edge
repeat
  | edge ← next edge in Voronoi cell of site(p0, p1);
until edge starts in p0 and edge is on the side lr;
for i ← 1 to length(p) do
  | repeat
  |   | add start node of edge to output;
  |   | edge ← next edge in Voronoi cell of site(pi−1, pi);
  | until edge ends in pi;
  | reverse orientation of e;
  | if pi has tolerance zone on side lr and i < length(p) − 1 then
  |   | repeat
  |   |   | add start node of edge to output;
  |   |   | edge ← next edge in Voronoi cell of pi;
  |   | until edge ends in pi;
  |   | reverse orientation of e;
  | end if
end for
```

Alg. 1: Collect nodes

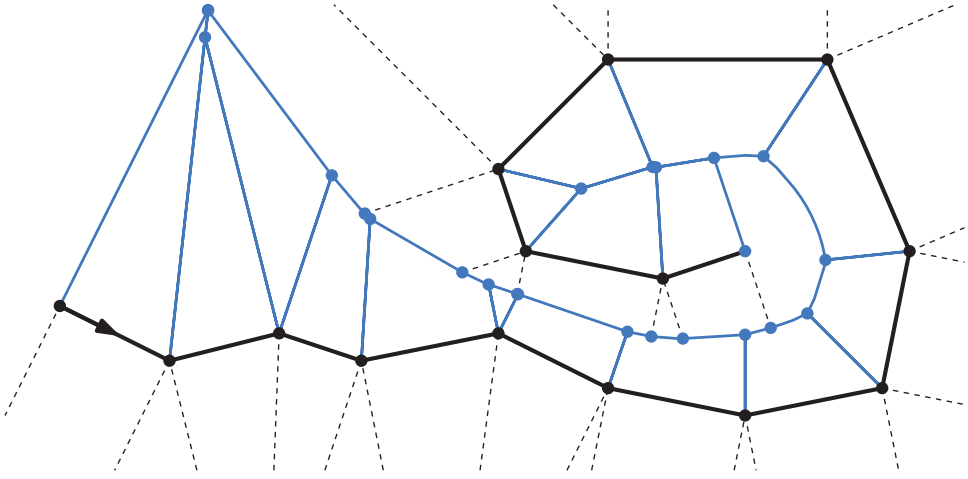

Fig. 5: The Voronoi edges traversed after Alg. 1 are shown in solid; all nodes of the Voronoi cells on one side of the profile get lined up in a list.

the profile's orientation as input. We assume that when traversing the boundary of a Voronoi cell, every Voronoi edge is oriented with respect to its start and end nodes according to the current traversal. That is, a traversal of a Voronoi cell of a site s with vertices p and q will begin with a Voronoi edge that starts in p, traverse one half of the the Voronoi cell until a Voronoi edge ending in q is encountered, continue with a Voronoi edge (on the other side of s) that starts in q, traverse the second half of the Voronoi cell of s, until reaching back to p. This orientation does not reflect the edge orientation as given by VRONI/ArcVRONI, but the particular orientation can be extracted easily from the context. Figure 5 shows the Voronoi edges traversed after Alg. 1 on the left of our sample profile.

Stage 2: Skip Nodes Outside of Tolerance Zone. In the next stage we make use of the clearance parameters of the nodes in the list to skip all those nodes that leave the conventional offset d, defined as either $|d_l|$ or $|d_r|$ depending on the side of the profile we are interested in. We run through the list until we find a pair of consecutive nodes n_i, n_{i+1} with clearances $\mathrm{clr}(n_i) < d < \mathrm{clr}(n_{i+1})$. We conclude that node n_i lies within the tolerance zone while n_{i+1} lies outside. Since every two consecutive nodes in the list are connected by a Voronoi edge, we use VRONI/ArcVRONI to compute the exact center p of the clearance disk with radius d on the Voronoi edge between n_i and n_{i+1}. The resulting point p lies exactly on the boundary of the tolerance zone and is inserted into the list between nodes n_i and n_{i+1}. Since we are currently outside of the tolerance zone we raise a boolean flag outside.

We continue in the list and remove all nodes from the list until we find a pair of consecutive nodes n_j and n_{j+1} with clearances $\mathrm{clr}(n_j) > d$ and $\mathrm{clr}(n_{j+1}) < d$. Again, we evaluate the Voronoi edge data and compute the position of the point q that lies exactly on the boundary of the tolerance zone. The point q is inserted into the list and we drop the boolean flag outside to indicate that we have returned into the tolerance zone.

```
input  : list of nodes, tolerance d
output: filtered list of nodes

outside ← false;
for ν ∈ input do
    t ← clr(ν) ;                                              // use VRONI
    if outside then
        if t < d then                                  // returning into tolerance zone
            outside ← false;
1           insert(node with clearance d);
        else
            delete(ν);
        end if
    else
        if t > d then                                    // leaving the tolerance zone
            outside ← true;
2           insert(node with clearance d);
            delete(ν);
        end if
    end if
end for
```

Alg. 2: Skip nodes

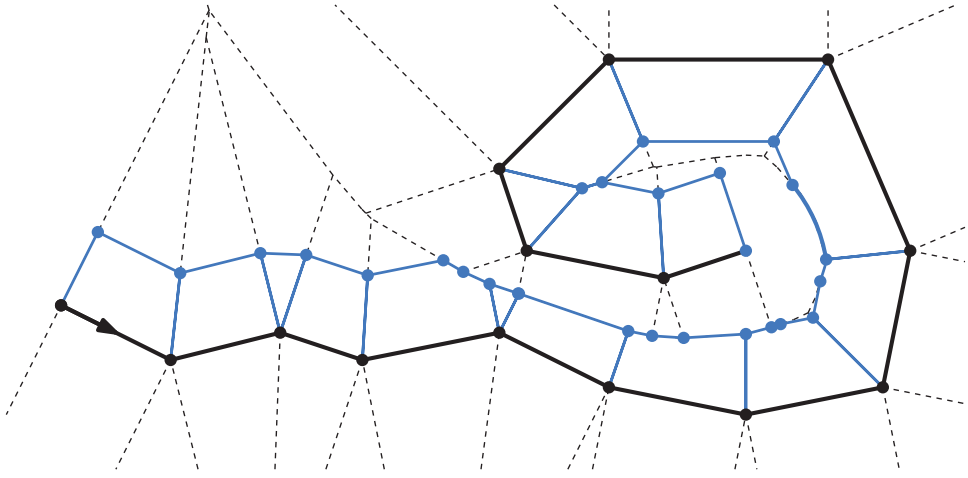

Fig. 6: The list after Alg. 2. Nodes that exceed the maximum distance to the profile are skipped.

We continue this procedure until we reach the end of the list. We now know that $d(n_i, P) \leq d$ holds for every node n_i in the resulting list of nodes. Note that in Alg. 2 the nodes added to the list in lines marked 1 and 2 do not necessarily coincide with nodes of the Voronoi diagram, but rather lie on the edge defined by the previous and the current node in the list. The result of this stage for our sample profile is shown in Fig. 6.

Stage 3: Removing Trees Within the Tolerance Zone. In the second stage we used the clearance parameters of the Voronoi nodes to remove all nodes of Voronoi cells that leave the tolerance zone of the profile. The result is the boundary of the tolerance zone with additional Voronoi edges attached to it in its interior. In the third stage we discard all those nodes that do not define the boundary of the tolerance zone. These structures attached to the boundary form trees. Hence, we can iterate over the list and remove all leaves as we pass by, ultimately removing all trees.

The manner in which we constructed the list of nodes implies that leaves correspond to triples (n_i, n_{i+1}, n_{i+2}) of nodes such that $n_i = n_{i+2}$, i.e., the first and the third node have the same position. Thus, we scan for such constellations of consecutive nodes and remove them from the list. If our current set of three nodes does not allow any deletion, we move on. This algorithm is summarized in Alg. 3, and the result for our sample profile is shown in Fig. 7. The spikes (shown in red) ensure that the symmetric Hausdorff distance is not exceeded.

After the tolerance zone on one side of a profile has been constructed, we turn our attention to the other side and repeat the three stages outlined above in order to compute the second boundary of the tolerance zone of P. (Note that this may actually mean computing a second boundary on the same side of

```
input  : list of nodes
output: final list of boundary nodes of tolerance zone
for i ← 1 to length(input)−2 do
    if node_{i−1} = node_i then                          // remove coinciding nodes
        delete(node_i);
        i ← i − 1;
    end if
    if node_{i−1} = node_{i+1} then                      // remove leaves
        delete(node_i);
        i ← i − 1;
    end if
end for
```

Alg. 3: Remove trees

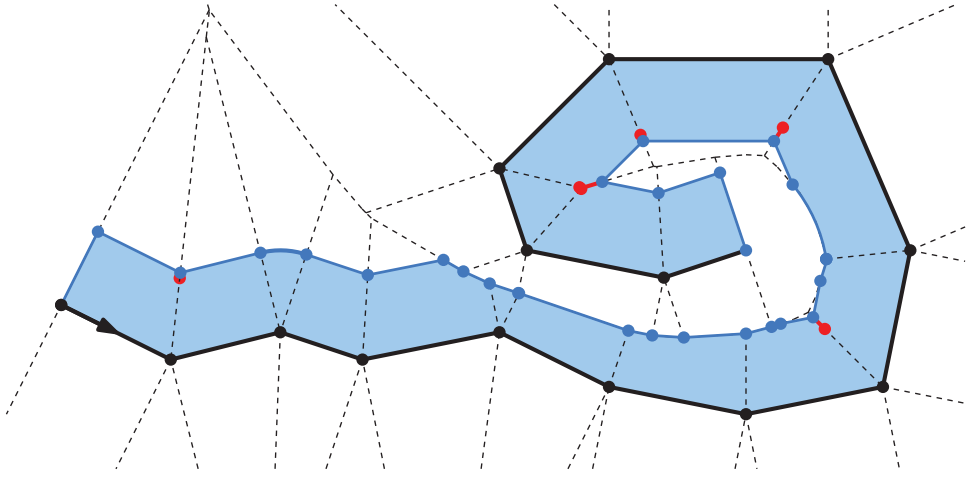

Fig. 7: After Alg. 3 all Voronoi edges that belong to trees within the tolerance zone have been removed.

the profile if a one-sided tolerance was requested by the user.)

2.4. Properties of the Tolerance Zone

Suppose that we constrain the approximation curve of every profile P of $\mathcal{P}$ to the interior of the tolerance zone of P. Our Voronoi-based construction ensures that the tolerance zones of every pair of profiles do not intersect in their relative interiors. (Actually, if required by a particular application, we can even guarantee a minimum clearance distance between pairs of tolerance zones.)

Let us denote the set of approximation curves of $\mathcal{P}$ by $\mathcal{A}$. The construction of $\mathcal{TZ}(\mathcal{P}, d_l, d_r)$ ensures that

- for an approximation A of $P \in \mathcal{P}$ the signed distances of (all points of) A from P are lower-bounded by d_l and upper-bounded by d_r, thus respecting the tolerances imposed by the user,
- all curves of $\mathcal{A}$ are disjoint, and that
- the topology of $\mathcal{P}$ is preserved: If a profile P_1 is contained inside of some other profile P_2 then the approximation A_1 of P_1 is contained inside of the approximation A_2 of P_2.

We note that for symmetric tolerances $d := -d_l = d_r > 0$ the relation $\mathcal{A} \subset \mathcal{TZ}(\mathcal{P}, -d, d)$ can also be expressed in terms of the directed Hausdorff distance from $\mathcal{A}$ to $\mathcal{P}$:

$$\mathcal{A} \subset \mathcal{TZ}(\mathcal{P}, -d, d) \Rightarrow h(\mathcal{A}, \mathcal{P}) \leq d \text{ where } h(\mathcal{A}, \mathcal{P}) := \max_{a \in \mathcal{A}} \min_{p \in \mathcal{P}} d(a, p).$$

Thus,

$$\mathcal{A} \subset \mathcal{TZ}(\mathcal{P}, -d, d) \wedge \mathcal{P} \subset \mathcal{TZ}(\mathcal{A}, -d, d) \Rightarrow H(\mathcal{A}, \mathcal{P}) \leq d,$$

where $H(\mathcal{A}, \mathcal{P})$ denotes the (symmetric) Hausdorff distance between $\mathcal{A}$ and $\mathcal{P}$. If the second condition $\mathcal{P} \subset \mathcal{TZ}(\mathcal{A}, -d, d)$ or, more generally, $\mathcal{P} \subset \mathcal{TZ}(\mathcal{A}, -d_r, -d_l)$ is omitted then the approximation curves are not constrained to pass through disc sectors with radii d (respectively $-d_l$, d_r) centered at the vertices of the profiles of $\mathcal{P}$. This may result in a poor approximation if $\mathcal{P}$ contains sharp corners or circular arcs whose radii are tiny relative to (the absolute values of) the approximation tolerances.

How can we ensure $\mathcal{P} \subset \mathcal{TZ}(\mathcal{A}, -d_r, -d_l)$ if this is requested by the user? We resort to the concept of *offset spikes* introduced by Heimlich and Held [6] to guarantee that no input vertex is further away than

d_l (or d_r) from the boundary of its tolerance zone. Roughly, this means generating spikes that emanate at the vertices of the tolerance zone and grow along the Voronoi diagram towards the vertices of $\mathcal{P}$ in order to constrain the approximation curves closer to the input vertices; see the red nodes and edges in Fig. 7. We refer to [6] for details on the construction of the offset spikes.

3. APPROXIMATION ALGORITHM

3.1. Approximation Primitive

We use uniform cubic B-splines as approximation primitives in order to achieve a C^2 continuous approximation. A cubic B-spline is a parametric curve composed of a linear combination of B-spline basis functions of degree three, where the form of the linear combination is governed by the position of the so-called control points. Each spline segment is defined by four vertices of the B-splines control polygon, and consecutive spline segments share three common vertices, see Fig. 8. For uniform B-splines the B-spline basis functions are simply shifted copies of each other. We note that each segment of a cubic B-spline is contained in the convex hull of its four control points. We refer to textbooks on spline modeling (e.g., [13]) for an introduction to B-splines.

In our approximation algorithm we will have to check frequently whether a cubic B-spline is contained in the tolerance zone. In order to facilitate this task we replace all circular arcs of the boundary of the tolerance zone by straight-line segments, with a constant number of line segments per circular arc. (This approximation is carried out in such a way that those line segments stay inside of the tolerance zone.) If we know that the start point of a cubic B-spline lies within the tolerance zone, it suffices to check whether the cubic B-spline intersects the boundary of the tolerance zone in order to check for containment within the tolerance zone.

For such a segment/spline intersection test we use de Boor's subdivision algorithm [4]. De Boor's algorithm is a fast and numerically stable algorithm for the evaluation of B-spline curves: It computes a point u that splits a B-spline segment into two sub-segments, and replaces a control polygon (p_1, p_2, p_3, p_4) by the two polygons (q_1, q_2, q_3, q_4) and (q_2, q_3, q_4, q_5), see Fig. 9. The new control polygon is related to the old control polygon by the so-called splitting matrix:

$$\begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \end{pmatrix} = \frac{1}{8} \begin{pmatrix} 4 & 4 & 0 & 0 \\ 1 & 6 & 1 & 0 \\ 0 & 4 & 4 & 0 \\ 0 & 1 & 6 & 1 \\ 1 & 1 & 4 & 4 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{pmatrix} \tag{1}$$

Repeated insertions of control points can be used to obtain a refinement of the original control polygon while leaving the curve defined by the control points unchanged. (Actually, one can show that the sequence of control polygons obtained by de Boor's subdivision algorithm converges to the B-spline curve defined by these.)

In order to check whether a uniform cubic B-spline *ucbs* intersects a straight-line segment ℓ of the border of the tolerance zone, we check whether ℓ intersects the convex hull of *ucbs*. If ℓ does not intersect the convex hull of *ucbs* then ℓ cannot intersect *ucbs*. Otherwise, we use de Boor's algorithm to subdivide *ucbs*, and recurse on the two halves. If the maximum intersection tolerance is reached then we report an intersection. (Hence, we err on the safe side.)

The validity of this intersection check follows from the fact that every curve segment of a B-spline is contained within the convex hull of its four control points [13], see Fig. 8. This intersection check is summarized in pseudo-code in Alg. 4.

We note that we may have to check one B-spline curve *ucbs* for intersection with several straight-line segments of the tolerance zone. Since the subdivision of a B-spline curve follows a regular pattern that does

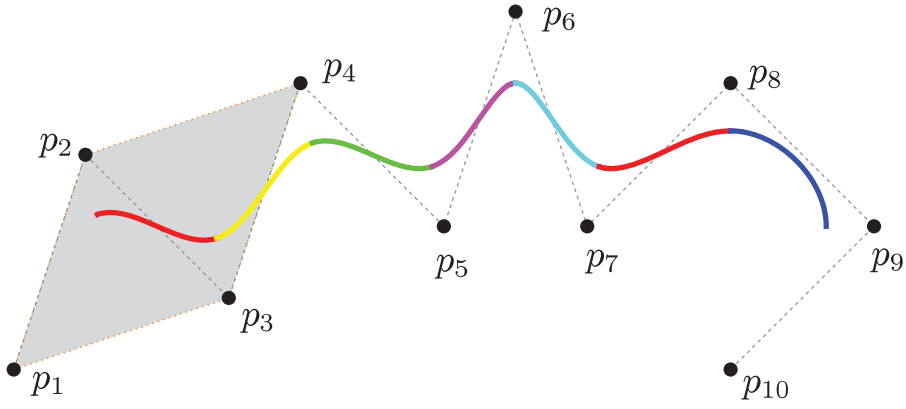

Fig. 8: A uniform cubic B-spline that consist of seven segments. The convex hull of the first four control points (of the red segment) is shaded in light grey.

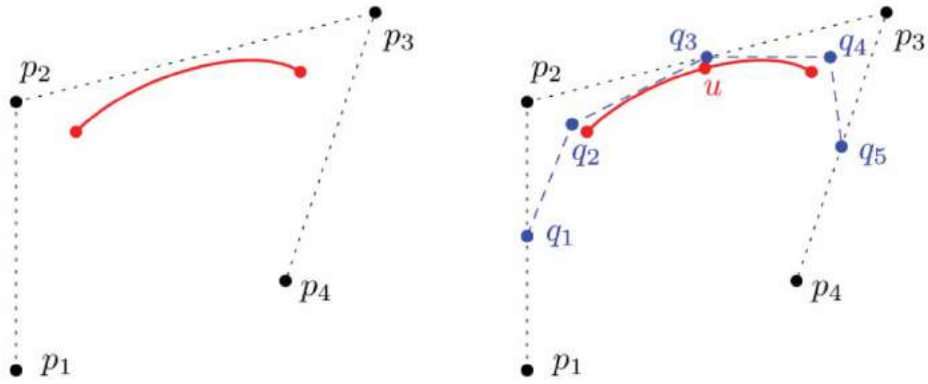

Fig. 9: Illustration of de Boor's subdivision process of a cubic B-spline.

constants: intersection tolerance
function `intersect(` p_1, p_2, p_3, p_4, ℓ `)`
  input : Control points p_1, p_2, p_3, p_4, straight-line segment ℓ
  output : false if an intersection can be ruled out, true otherwise

  if ℓ *intersects* $CH(\{p_1, p_2, p_3, p_4\})$ **then**
    if $\|p_1 - p_4\| \leq$ *intersection tolerance* **then**
      return true;
    else

$$\begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \end{pmatrix} \leftarrow \frac{1}{8} \begin{pmatrix} 4 & 4 & 0 & 0 \\ 1 & 6 & 1 & 0 \\ 0 & 4 & 4 & 0 \\ 0 & 1 & 6 & 1 \\ 1 & 1 & 4 & 4 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{pmatrix};$$

      return `intersect(` q_1, q_2, q_3, q_4, ℓ `)` ***or*** `intersect(` q_2, q_3, q_4, q_5, ℓ `)`;
    end if
  else
    return false;
  end if

Alg. 4: The function `intersect` uses the convex hull of the control points of the cubic B-spline to either rule out an intersection or to split the B-spline into two halves and recursively check for an intersection.

not depend on external parameters, we might end up computing the same control vertices over and over again. We can refine algorithm:intersection and save computational time by storing the control vertices in a binary tree $\mathcal{T}$, where each node of $\mathcal{T}$ corresponds to one subdivision. During an intersection check of *ucbs* with a line segment we start at the root of $\mathcal{T}$ and descend $\mathcal{T}$ in the same way as we recurse on portions of *ucbs*. If a particular intersection check requires a subdivision whose corresponding control vertices are not stored in $\mathcal{T}$ then we compute the vertices by means of Eqn. (1). Of course, we also store them appropriately in a new node of $\mathcal{T}$ in order to make them available for future intersection checks. Thus, the binary tree $\mathcal{T}$ reflects all subdivisions of the same B-spline curve carried out within the individual intersection tests.

3.2. Computing an Approximation

We adopt the approach by Heimlich and Held [6] and compute a set of points within the tolerance zone that will become the "support"for our approximation primitives. (In [6], "support"means start point or end point of a biarc that is used for the approximation.) These so-called *approximation nodes* ("a-nodes") are placed on the Voronoi diagram of the tolerance zone. More precisely, the a-nodes are placed on the *medial line* of the Voronoi diagram of the tolerance zone: An edge of the Voronoi diagram of the tolerance zone (restricted to the interior of the tolerance zone) belongs to the medial line if and only if it is defined by one edge of the left boundary and one edge of the right boundary of the tolerance zone, see Fig. 10.

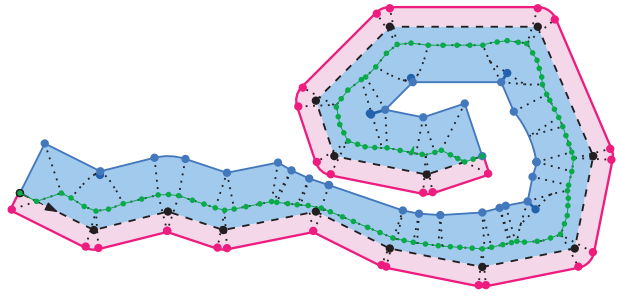

Fig. 10: The approximation nodes are sampled on the medial line of the tolerance zone.

We would prefer to sample every edge of the medial line by uniformly spaced a-nodes such that the distance between two neighboring a-nodes is only a fraction of the width $d_r - d_l$ of the tolerance zone. However, such a sampling scheme does not allow to bound the number of a-nodes in terms of the number n of input sites. Hence, in order to avoid that the complexity of the actual approximation algorithm grows unboundedly, we allow only a total of $c \cdot n$ a-nodes, for some constant c. (E.g., $c := 10$ turned out to be a decent choice in our tests.) This total number of a-nodes is assigned to the nodes and edges of the medial lines in a uniform way. That is, long edges get more a-nodes than short edges, and a very short edge might have no a-node assigned to its relative interior at all.

Once the a-nodes have been fixed, the goal is to find a small number of primitives that stay within the tolerance zone and use a subset of the a-nodes as support. Naturally, this leads to some form of a greedy-like approach, where one tries to find long primitives that span as many as a-nodes as possible. Note that [6] does not use a length comparison based on the actual curve length but, for complexity reasons, uses the number of passed a-nodes as a measure of distance. The results reported in [6] indicate that, for approximations by biarcs or straight-line segments, the best performance is achieved by a doubling-and-bisection strategy [9]: An exponential growth phase is followed by a binary search to obtain the next approximation primitive.

These methods have been proven to generate reliable results when used with approximation primitives that are defined upon exactly two a-nodes. Since consecutive B-spline segments share a common set of control vertices rather than only one a-node, there is no obvious way how a greedy scheme is adapted best to our needs. We observed that when choosing the a-node for which the current primitive passes the maximum number of a-nodes, chances are good that the next primitive will manage to pass only a rather small number of a-nodes. In particular, stretching the distance between two control vertices may distort the following spline segments such that no further valid primitives can be found.

To overcome this problem we propose a top-down approach that refines an initially coarse approximation recursively: For every profile of $\mathcal{P}$ we compute an initial B-spline curve that consists of a small number of segments. This initial approximation curve is then refined by subsequently adding a-nodes as new control vertices until we obtain an approximation curve that fits through the tolerance zone.

The algorithm summarized in pseudo-code in Alg. 5 requires a vector of a-nodes and a description of the left and right boundary of the tolerance zone of the profile to be approximated. It is essential that the data type for the a-nodes allows constant-time random access. Also, every a-node has a pointer to the corresponding segments of the left and right tolerance zone boundary. Thus, we can test for containment in the tolerance zone locally by enumerating the relevant boundary segments and carrying out individual intersection tests between these segments and the B-spline under construction.

input : vector of a-nodes a, boundary of tolerance zone
output: list of approximation primitives r
data : empty list of node indices n

for $i \leftarrow 1$ **to** 3 **do**
  `pushFront(0)` into n;
  `pushBack(length(`a`) -1)` into n;
end for
$c \leftarrow 0$;
while $c + 3 <$ `length(`n`)` **do**
  if `Primitive(`n_c, n_{c+1}, n_{c+2}, n_{c+3}`)` *is invalid* **then**
    `insert(`$\lfloor (n_c + n_{c+1})/2 \rfloor$`)` at position $c+1$ into n;
    `insert(`$\lfloor (n_{c+1} + n_{c+2})/2 \rfloor$`)` at position $c+3$ into n;
    `insert(`$\lfloor (n_{c+2} + n_{c+3})/2 \rfloor$`)` at position $c+5$ into n;
  else
    $c \leftarrow c + 1$;
  end if
end while

Alg. 5: Top-down approach (for an open input chain)

The initial approximation curve is obtained as follows for an open profile: One can show analytically that the start point of a B-spline is bound to coincide with a point u if its first three control points are identical to u. Analogously, a B-spline ends in v if its last three control points coincide with v. Hence, to approximate an open profile we add the start and end point of the profile three times each as the first and the last a-nodes, thus obtaining the initial approximation. Then the approximation proceeds as summarized in Alg. 5: When a segment of the B-spline is invalid since it does not fit into the tolerance zone then we add three new control points.

The algorithm as given in Alg. 5 does not rely on recursion but rather uses the resulting list of indices to iteratively refine the approximation primitives. This algorithm can easily be extended to support closed profiles. We use a circular iterator on the a-nodes that allows a traversal from the end of the vector back to the start. With an appropriate choice of the starting nodes and properly adapted loop conditions, the algorithm for closed profiles follows.

3.3. Runtime Complexity

In a preprocessing phase, all profiles of $\mathcal{P}$ are scanned for degeneracies: Collinear and co-circular vertices as well as zero-length segments and arcs are removed. Of course, $O(n)$ time suffices for the preprocessing, where n denotes the total number of segments and arcs of $\mathcal{P}$.

The computation of the tolerance zone boundary relies on the Voronoi diagram of $\mathcal{P}$, and the placement of the a-nodes requires knowing all medial lines, which are subsets of the Voronoi diagram of the tolerance zone. The theory of Voronoi diagrams tells

us that the Voronoi diagram of a set of m (non-intersecting) straight-line segments and circular arcs can be computed in $O(m \log m)$ time and $O(m)$ space, both in the worst case [14] and in the expected case [10]. Since the number of edges and vertices of the tolerance zone is linear in n, the total complexity of both Voronoi computations is $O(n \log n)$. The computation of the tolerance zone runs in $O(n)$ time, as does the computation of the a-nodes.

For the top-down approach, every split step requires a containment test that can involve a linear number of intersection checks of the B-spline curve with edges of the boundary of the tolerance zone. The merge step can be done in constant time by simply joining two lists of consecutive control vertices together. Note that every edge/spline intersection check involves one or more steps of de Boor's subdivision, where the number of steps depends on an intersection tolerance rather than on n. Our experiments indicate that a small number of subdivision steps suffices for every intersection check.

If we regard one such intersection check as a constant-time operation then the total runtime complexity of the top-down approximation is $O(n \log n)$, requiring linear $O(n)$ space. Finally, these complexities sum up to an overall $O(n \log n)$ runtime complexity and $O(n)$ space.

4. EXPERIMENTAL EVALUATION

We implemented this approximation algorithm in C++, based on the Voronoi code VRONI/ArcVRONI [7,10]. We ran an extensive series of tests on 22124 individual input files. Our input files comprise both contrived and real-world data, with one or more polygons or open/closed curvilinear profiles per file. Our real-world data sets include cross-sections of CAD parts, tool paths, fonts, maps and outlines of countries, and road and river networks.

In order set up tests that can be run automatically and repeatedly in a consistent way, we used half of the mean segment/arc length as an automatically chosen symmetric approximation tolerance. (Using one fixed approximation tolerance for all polygons does not make sense since it may be far too large for one profile and, thus, cause a drastic data compression, while it may be far too small for another polygon and, thus, effectively disable any approximation.) In all tests the approximation curve was also required to lie within a tolerance zone of the original input. (That is, our algorithm had to generate offset spikes.)

We ran our tests on a machine equipped with an INTEL Core i7-2600 processor with four cores backed by 16 GB of memory. Our test machine runs Ubuntu 10.4 LTS with the Linux 3.0.0 x64 kernel as a 64 bit operating system. Since our implementation does not benefit from hyper-threading we simultaneously executed four instances of our implementation to fully utilize all cores available.

Figure 11 collects the results of our statistics on the runtime and on the memory consumption of our approximation code. Both plots indicate that the code performs as predicted by the theoretical analysis. No significant difference in the results was observed between tests of synthetic and real-world input data. Hence, we do not bother to present essentially the same plots twice.

We also compared our new top-down approach with the doubling-and-bisection algorithm proposed by Held and Eibl [9]. Our tests use the same automatic setup of tolerances and a-nodes for both types of approximation algorithms. The y-axes of the plots in Fig. 12 show ratios of the form

$$\frac{\text{Result(Top} - \text{Down)}}{\text{Result(Doubling} - \text{and} - \text{Bisection)}}$$

for approximations of profiles by straight-line segments (top row), biarcs (middle row), and uniform cubic B-splines (bottom row). The left column of plots shows the runtime ratios, whereas the right column shows the ratios of the number of output primitives. The strips highlighted represent the 2σ-environment

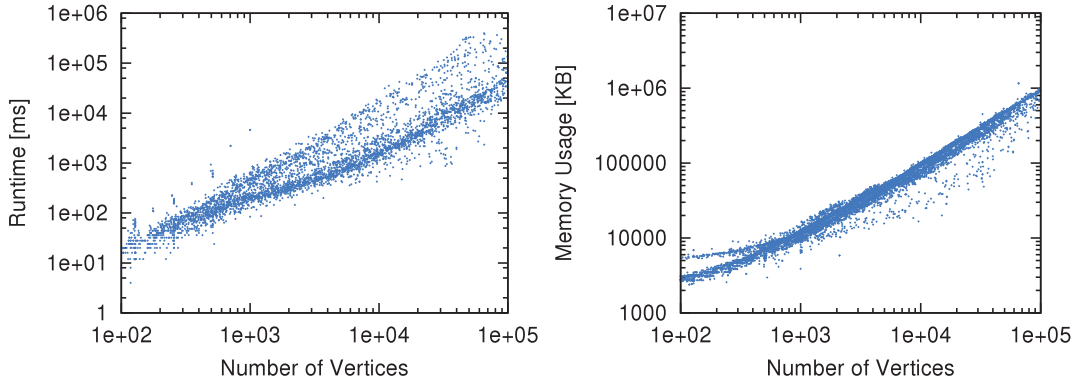

Fig. 11: Runtime plot (left) and memory consumption (right) of our implementation. Each point in these plots represents one test run on one specific input file.

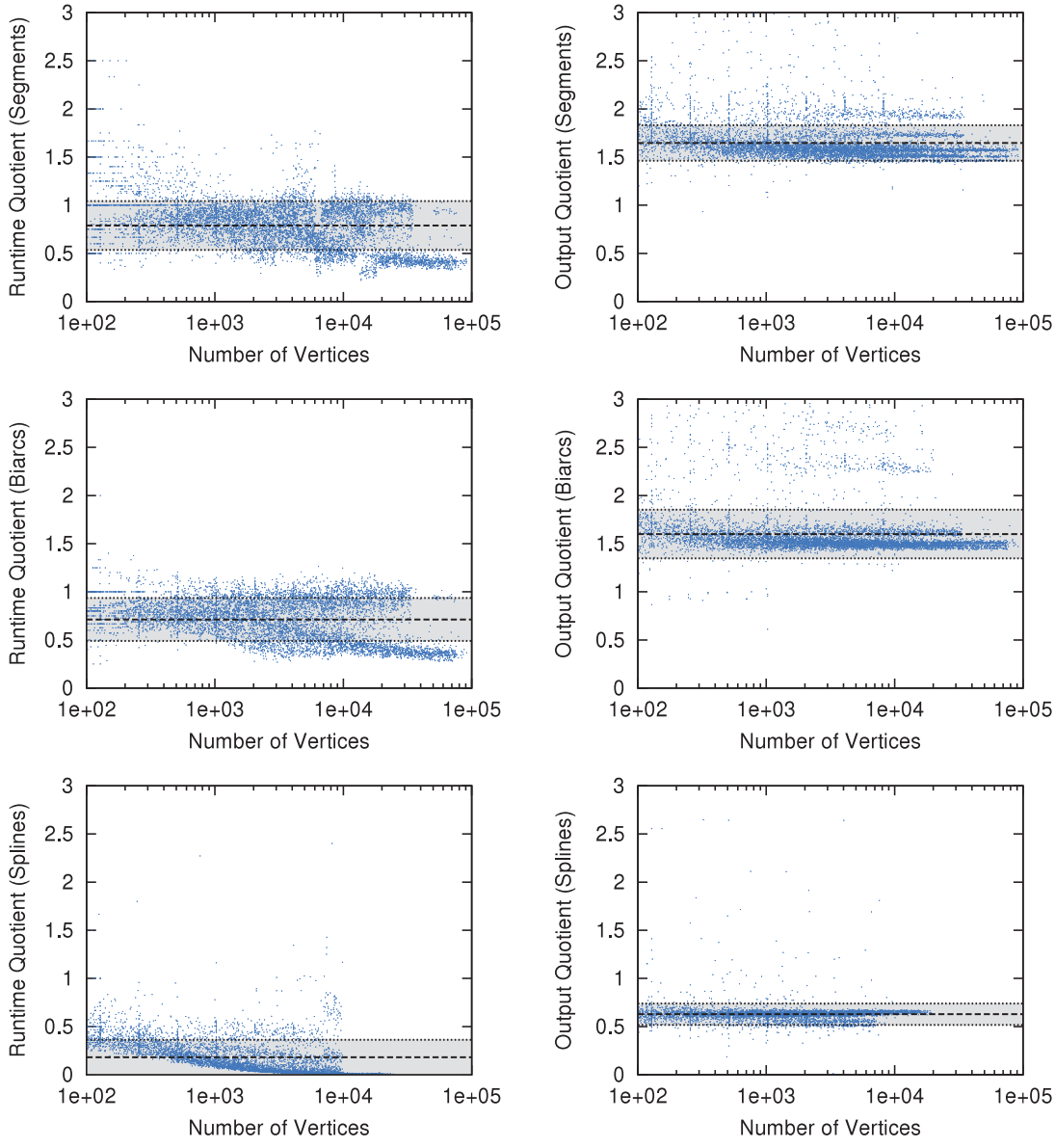

Fig. 12: Comparison of output and runtime for top-down and doubling-and-bisection methods.

around μ, where σ is the standard deviation from the mean μ.

The plots indicate that for approximations by line segments and biarcs the top-down approach is about 25% faster than doubling-and-bisection but generates about 60% more primitives for the approximation curves. Thus, from a user's perspective the top-down approach is likely to be regarded as inferior to doubling-and-bisection. The situation is quite different for approximations by uniform cubic B-splines: Our new top-down approach is both significantly faster and generates fewer primitives than doubling-and-bisection.

Recall that we used half of the mean segment/arc length as an automatically chosen symmetric approximation tolerance for our tests. Let $rt(\varepsilon)$ denote the runtime for some specific input relative to a symmetric tolerance ε. The left plot in Fig. 13 depicts ratios $rt(10 \cdot \Delta)/rt(\Delta)$, where Δ denotes the default tolerance chosen automatically for an input, and the right plot of Fig. 13 depicts ratios $rt(\Delta)/rt(0.1 \cdot \Delta)$. The plots indicate that, on average, increasing or decreasing the approximation tolerance by a factor of 10 tends to result in a multiplication of the runtime by a constant factor, albeit with a noticeable variation for complex inputs.

5. DISCUSSION AND CONCLUSION

We present a new algorithm for approximating a set of open and closed curvilinear profiles by means

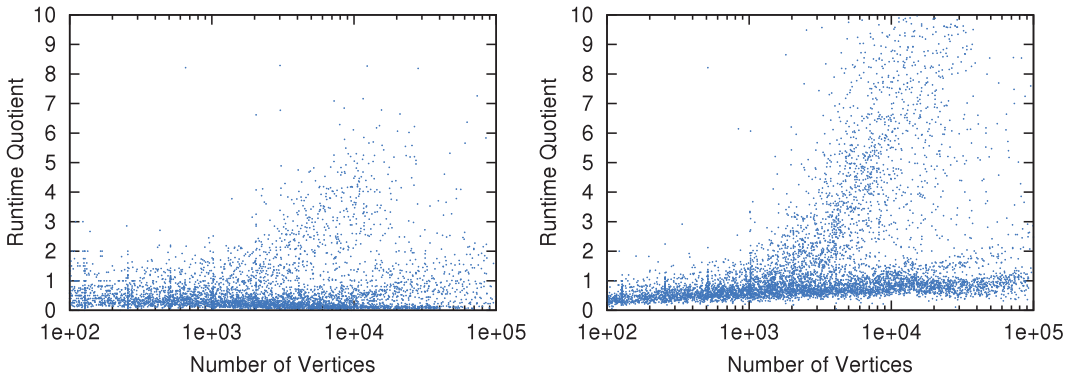

Fig. 13: Impact of the approximation tolerance on the runtime.

of uniform cubic B-splines, i.e., by C^2 continuous primitives. The algorithm was implemented in C++, based on the Voronoi package VRONI/ArcVRONI [7, 10]. Extensive tests run on 22124 synthetic and real-world data sets indicate that our approach is reliable and robust enough for practical applications.

An analysis of the runtime complexity and memory consumption shows that our algorithm runs in $O(n \log n)$ time and has a linear memory footprint. We compared our top-down fitting scheme to previously published greedy methods and conclude that our new approach, although slightly faster, tends to produce more output when used for approximations with biarcs and straight-line segments. However, our approach is both faster and more efficient in terms of smaller output size when used for approximations with uniform cubic B-splines.

Our algorithm is a genuine extension of the work by Heimlich and Held [6] to C^2-approximations of open and closed profiles since it retains its pros: A set of simple profiles is approximated by a set of simple uniform cubic B-splines, thereby maintaining the topology of the input. Asymmetric and one-sided tolerances are supported. Due to the use of Voronoi computations and Voronoi-based offsetting, our algorithm is completely immune to input noise: As the approximation tolerance is gradually increased, the number of approximation primitives used decreases gradually. Thus, our algorithm can be used for obtaining a data compression that maintains important characteristics of the input.

The single biggest drawback of our approach is the use and placement of a-nodes as control points for the B-splines. There is little justification for placing a-nodes on the medial line of a tolerance zone besides stating that this works decently in practice and admitting that no one has any insights in a better approach. A very recent algorithm by Maier and Pisinger [12] allows to approximate a closed polygon by a minimum number of circular arcs and line segments without relying on a-nodes, but it seems very difficult to extend their scheme to approximations with B-splines.

However, a poor distribution of the a-nodes may result in a large number of approximation primitives that could have been avoided with a better placement of the a-nodes. Figure 14 shows a sample tolerance zone of a polygon with n vertices that could have been approximated by using only four line segments (red square), while a poor placement of the a-nodes results in a wiggly approximation that consumes $O(n)$ line segments (orange polygon). It is obvious that one can modify this example such that similar results occur for an approximation with B-splines. Thus, devising a refined strategy for the placement of the a-nodes constitutes an important task for future work.

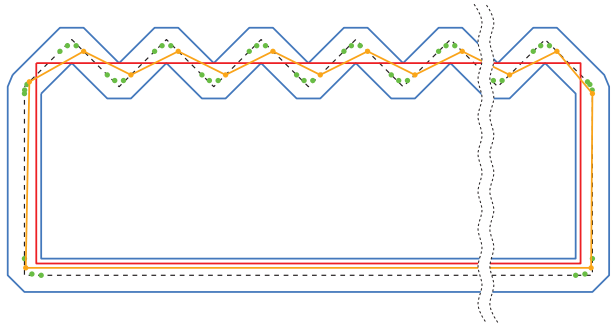

Fig. 14: An inappropriate placement of the a-nodes may cause the number of approximation primitives to sky-rocket.

REFERENCES

[1] Ahn, Y. J.; Hoffmann, C.M.; Kim, Y.S.: Curvature Continuous Offset Approximation Based on Circle Approximation Using Biquadratic Bézier Curves, Comput. Aided Design, 43(8), 2011, 1011–1017. doi:10.1016/j.cad.2011.04.005.

[2] Behar, S.; Estrada, J.; Hernández, V.; León, D.: Smoothing of Polygonal Chains for 2D Shape Representation Using a G2-Continuous Cubic A-Spline, In Proc. CIARP'05, volume 3773 of Lecture Notes Comput. Sci., 42–50, 2005. doi:10.1007/11578079_5.

[3] Chuang, S.-H.F.; Shih, J.-L.: A Novel Approach for Computing C2-Continuous Offset of NURBS Curves, Int. J. Avd. Manuf. Technology., 29(1), 2006, 151–158. doi:10.1007/s00170-004-2484-x.

[4] de Boor, C.: On Calculating with B-Splines, J. Approx. Theory, 6, 1972, 50–62.

[5] Drysdale, R. L. S.; Rote, G.; Sturm, A.: Approximation of an Open Polygonal Curve with a Minimum Number of Circular Arcs and Biarcs, Comput. Geom. Theory and Appl., 41, 2008, 31–47. doi:10.1016/j.comgeo.2007.10.009.

[6] M. Heimlich, M. Held. Biarc Approximation, Simplification and Smoothing of Polygonal Curves By Means of Voronoi-Based Tolerance Bands. Int. J. Comput. Geom. Appl., 18(3):221–250, June 2008. doi:10.1142/S0218195908002593.

[7] Held, M.: VRONI: An Engineering Approach to the Reliable and Efficient Computation of Voronoi Diagrams of Points and Line Segments, Comput. Geom. Theory and Appl., 18(2), 2001, 95–123. doi:10.1016/S0925-7721(01)00003-7.

[8] Held, M.: VRONI and ArcVRONI: Software for and Applications of Voronoi Diagrams in Science and Engineering, In Proc. 8th Int. Symp. Voronoi Diagrams in Science & Engineering, pages 3–12, Qingdao, China, June 2011. doi:10.1109/ISVD.2011.9.

[9] Held, M.; Eibl, J.: Biarc Approximation of Polygons within Asymmetric Tolerance Bands, Comput. Aided Design, 37(4), 2005, 357–371. doi:10.1016/j.cad.2004.06.001.

[10] Held, M.; Huber, S.: Topology-Oriented Incremental Computation of Voronoi Diagrams of Circular Arcs and Straight-Line Segments, Computer Aided Design, 41(5), 2009, 327–338. doi:10.1016/j.cad.2008.08.004.

[11] Held, M.; Spielberger, C.: Improved Spiral High-Speed Machining of Multiply-Connected Pockets, Computer Aided Design & Applications, to be published.

[12] Maier, G.; Pisinger, G.: Approximation of a Closed Polygon with a Minimum Number of Circular Arcs and Line Segments, Comput. Geom. Theory and Appl., 46(3), 2013, 263–275. doi:10.1016/j.comgeo.2012.09.003.

[13] Prautzsch, H.; Boehm, W.; Paluszny, M.: Bézier and B-Spline Techniques, Springer-Verlag, 2002. ISBN 978-3540437611.

[14] Yap, C. K.: An O(n log n) Algorithm for the Voronoi Diagram of a Set of Simple Curve Segments, Discrete Comput. Geom., 2(4), 1987, 365–393.