
Implementing Ruppert’s Algorithm for Generic Curves in 2D

Barbara M. Anthony and Matthew D. Flatau

Southwestern University, Georgetown Texas
{anthonyb,flataum}@southwestern.edu

1 Introduction

While quality implementations exist for meshing straight-line inputs, fewer are available for handling curved inputs, even in 2D. Many are based on the well-known Ruppert’s algorithm [10] which has led to a large body of research in meshing. In this work, we provide a software package that handles a variety of smooth inputs in 2D, based on a minimal modification of Ruppert’s algorithm. Existing software for curved inputs lacks the elegance of Ruppert’s original algorithm [1, 3] (and subsequent improvements) or is application-specific [2]. In contrast, we seek to keep the core of our work as similar as possible to Ruppert’s algorithm to allow our software to benefit from related research and to be easily updated for additional input curve types. In particular, the differences in our implementation versus the original are limited to two pre-processing steps in the spirit of [7] and a generalized midpoint calculation.

2 Implementation

Ruppert’s algorithm [10] is given in Algorithm 1. We characterize the only modifications necessary to apply said algorithm on generic curved inputs in two dimensions. Next, we describe the classes of curves currently implemented.

2.1 Modifications for Generic Curves

For any generic curve, in order to use Ruppert’s algorithm with minimal modifications, it suffices to be able to split the curve in half, presplit the curve to obtain subsegments with restricted total variation in orientation, and intersect the curve with a circle for protecting small angles. While the first is easiest to state, such midpoint calculation is the only change to the body of Ruppert’s algorithm; the other two steps are necessary for preprocessing.

Algorithm 1 Ruppert's Algorithm

Require: Planar straight-line graph X , with vertices V and segments S **Require:** Angle between any pair of adjacent segments $\geq 60^\circ$ **Ensure:** Conforming Delaunay triangulation DT of X with all angles $\geq \alpha (< 20.7^\circ)$

```

1: Initialize bounding box and triangulate input vertices
2: while any segment  $s$  is encroached or any triangle  $t$  has some angle  $< \alpha$  do
3:   if  $s$  is encroached then
4:     add midpoint of  $s$  to  $V$ , update  $\{S, DT\}$ 
5:   else
6:      $p$  = the circumcenter of  $t$ 
7:     if  $p$  encroaches upon some segment  $s'$  then
8:       add midpoint of  $s'$  to  $V$ , update  $\{S, DT\}$ 
9:     else
10:      add  $p$  to  $V$ , update DT
11:    end if
12:  end if
13: end while

```

Midpoint Lines 4 and 8 of Algorithm 1 necessitate splitting an input segment in half, i.e., determining the midpoint and splitting appropriately. While trivial for a straight line segment, finding a suitable point for generic curves requires more thought. While straight segments are subdivided into two portions of equal length, the length requirement need not be strictly maintained for generic curves; rather, the split must occur at a point on the curve near to the center of the diametral circle determined by the curve's endpoints. (Preprocessing subdivides any closed curves into appropriate subsegments.)

Presplit For Algorithm 1 to succeed, segments can be arbitrarily long provided they do not curve 'too much'. In particular, curves must stay within the diametral ball of the endpoints, which can be ensured by restricting the total variation in orientation. For generic curves a 45° bound suffices; it can be relaxed to 90° for circular arcs [7].

Small Angle Circles Small input angles ($< 60^\circ$ is the theoretical bound, though in practice angles $> 51^\circ$ typically work [6]) inherently pose a problem for Algorithm 1. Typically such angles are handled by splitting adjacent segments at equal lengths and allowing certain small angles to remain in the final triangulation [5, 12]. Our implementation permits them to remain by explicitly adding a circle around a vertex with a small angle, as in [9], splitting the adjacent segments at equal lengths, i.e. the radius. The region inside said circles is now 'protected' and Ruppert's algorithm proceeds outside the circle.

2.2 Curve Classes

The generic curves that are first implemented include: (straight) line segments, circles, circular arcs, and Bézier curves (quadratic and cubic).

Midpoint Midpoint calculations for the indicated types are simple. The only complication arises from the possibility that numerous points may exist on a single curve. Thus there is some overhead in maintaining the relative ordering, and noting that midpoints are with respect to subsegments, not necessarily the curve as a whole. Likewise, knowing the endpoints of a circular arc does not suffice, since there are four possible circular arcs with a given radius and endpoints, reduced to two with a specified center, and determined uniquely if ordering of the endpoints is standardized. While no sophisticated calculations are required, curves each require their own class to handle the distinctions.

Presplit As mentioned, Algorithm 1 requires bounds on the total variation in orientation of input curves. Naturally straight line segments have zero variation. Circles and circular arcs can easily be presplit by finding the angle between points on the curve (if any) and subdividing appropriately. These calculations are aided by the total variation precisely matching the angle spanned by the associated arc. For Béziers, finding the total variation involves calculating angles between tangent lines. For quadratic Béziers, it is the angle between tangents at the endpoints, whereas for cubic Béziers a third interior point should be considered. In either case, midpoint insertion occurs and is repeated until the total variation of all subsegments satisfies the constraints.

Small Angle Circles Circles are inserted around small angles preventing Algorithm 1 from continuing to split indefinitely. The radius r for each small angle circle should be small enough so that no such circles overlap or touch disjoint curves; as such, one-third the distance to the nearest disjoint curve or input vertex suffices. Currently for Béziers a brute-force search is done using an approximation for the curve, with satisfactory results, but we hope to improve the efficiency with more sophisticated techniques. Finding the point on a straight line segment that is a distance r from a small angle vertex is trivial, but computing this intersection is more complicated for generic curves (explicit formula is known for circles/arcs, binary search is used for Béziers).

The final challenge to constructing the small angle circles lies in correctly ordering the intersection points. The most interesting case arises when the small angle circle intersects two or more curves which are tangent at the relevant vertex. Detecting such tangencies is not difficult, but since there can be arbitrarily many tangent curves at a given point (see Fig. 2(b)) a systematic way to determine the ordering is a must. As such, we maintain a curvature variable, indicating both how quickly a curve recedes from the straight line and in which direction. For instance, line segments have zero curvature while circles have a curvature based on the inverse of the radius of the curve.

3 Examples

We now provide examples illustrating the software package. Some examples explicitly show presplit points (blue), small angle circles (red), the resulting triangulations, and zoomed-in portions illustrating the detail in the mesh.

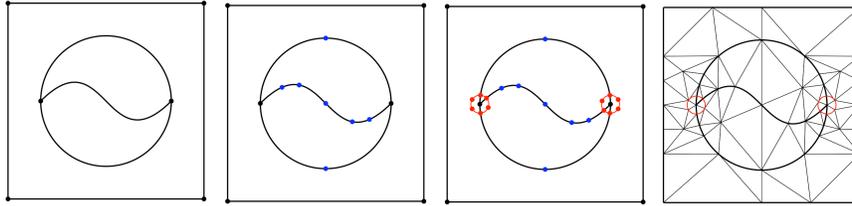


Fig. 1. From left to right: a smooth input with input vertices enlarged, presplit vertices inserted, small angle protection circles added, and the final triangulation after performing Ruppert's algorithm with midpoint modification.

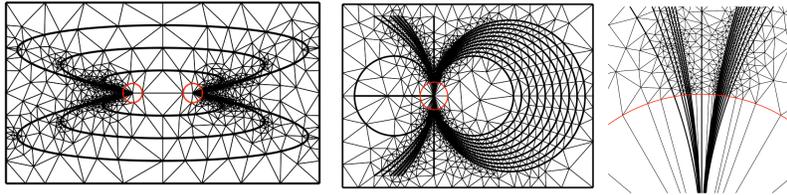


Fig. 2. A triangulation of multiple cubic Bézier curves (left), numerous curves (circles, Béziers, and a segment) tangent at the center (middle), and a zoomed-in view of the detailed mesh between tangent curves (right).

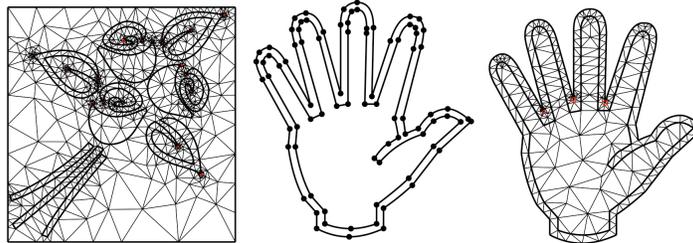


Fig. 3. A complex mesh (left), another input (middle), and resulting output (right).

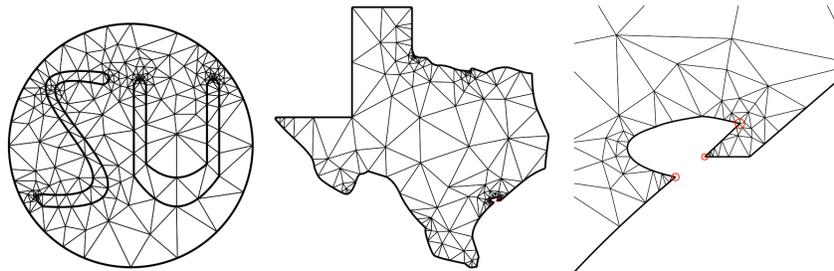


Fig. 4. A mix of curves (left), and a mesh of Texas (middle), zoomed in near the port of Houston (right) which contains small angles that are protected.

4 Conclusion and Future Work

Future extensions to the software include implementing higher-degree Bézier curves (a challenge as much in the output in SVG format as in the mesh creation) and other interesting curves. Improvements in efficiency will also be sought, including improving the radius computation and, as in [5], only protecting small angles when necessary. We anticipate releasing the full code under the MIT license in early 2011.

Acknowledgments Thanks to Nicolas Devillard for iniparser [4] available under the MIT license, Alexander Rand for some starter code from [8], and Jonathan Shewchuk for public domain exact geometric predicates [11].

References

1. C. Boivin and C. F. Ollivier-Gooch. Guaranteed-quality triangular mesh generation for domains with curved boundaries. *Int. J. Numer. Meth. Eng.*, 55(10):1185–1213, 2002.
2. D. Cardoze, A. Cunha, G. L. Miller, T. Phillips, and N. Walkington. A Bézier-based approach to unstructured moving meshes. *Proc. 20th Symp. Comput. Geom.*, pages 310–319, 2004.
3. S.-W. Cheng, T. K. Dey, and J. A. Levine. A practical Delaunay meshing algorithm for a large class of domains. *Proc. 16th Int. Meshing Roundtable*, pages 477–494, 2007.
4. N. Devillard. iniParser: stand-alone ini Parser library in ANSI C. <http://ndevilla.free.fr/iniparser/>.
5. G. L. Miller, S. E. Pav, and N. Walkington. When and why Delaunay refinement algorithms work. *Int. J. Comput. Geom. Appl.*, 15(1):25–54, 2005.
6. S. E. Pav. *Delaunay Refinement Algorithms*. PhD thesis, Carnegie Mellon University, May 2003.
7. S. E. Pav and N. Walkington. Delaunay refinement by corner lopping. *Proc. 14th Int. Meshing Roundtable*, pages 165–181, 2005.
8. A. Rand. DIR3 (Delaunay incremental refinement in 3D). <http://www.math.cmu.edu/~arand/DIR3/>.
9. A. Rand and N. Walkington. Collars and intestines: Practical conforming Delaunay refinement. *Proc. 18th Int. Meshing Roundtable*, pages 481–497, 2009.
10. J. Ruppert. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, 18(3):548–585, 1995.
11. J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Comput. Geom.*, 18(3):305–363, 1997.
12. J. R. Shewchuk. Mesh generation for domains with small angles. *Proc. 16th Symp. Comput. Geom.*, pages 1–10, 2000.