Complex-Valued Contour Meshing

Chris Weigle and David C. Banks

ABSTRACT

An isovalue contour of a function of two complex variables defines a surface in four-space. We present a robust technique for creating polygonal contours of complex-valued functions. The technique, Contour Meshing, generalizes well to larger dimensions.

CR Descriptors: G.1.5 [**Numerical Analysis]** Roots of Nonlinear Equations; G.1.6 [**Numerical Analysis**] Optimization; G.2.1 [**Discrete Mathematics**] Combinatorics; I.3.5 [**Computer Graphics**] Computational Geometry and Object Modeling; I.3.6 **[Computer Graphics]:** Methodology and Techniques; J.2 [**Physical Sciences and Engineering**] Mathematics and Statistics.

1 Introduction

This paper describes a recursive technique to construct a triangle mesh on an implicit surface (also called a variety or contour) in 4-space. The technique readily extends to large dimensions. We explore the resulting family of surfaces with an interactive 4D viewer in order to inspect the behavior of a complex curve as it sweeps through a singularity.

The *k*-dimensional analog of a surface is called a *k*-manifold. It is known that a *k*-manifold cannot necessarily be triangulated unless it is differentiable [Cairns]. An implicitly-defined contour might not be differentiable, so one might imagine that some exotic contour could arise that prevents any algorithm from triangulating it. Fortunately, a contour is guaranteed to be triangulable if the underlying function is algebraic [Waerden] or analytic [Koopman]; our aim of triangulating a contour in large dimensions is therefore not undertaken in vain.

Various triangulation techniques have been developed for computer graphics [Koide] [Lorensen] [Bloom88] and numerical analysis [Allgower90] [Allgower91]. Many of these techniques are designed specifically to construct triangulations only of curves, surfaces, or varieties of co-dimension one (several having been implemented as part of the Pisces project at the Geometry Center [Pisces]). The Contour Meshing algorithm, by contrast, is quite general. It consults *m* functions in *(k+m)*-dimensional space to yield an implicit *k*-dimensional contour.

Section 2 describes the Contour Meshing scheme used to locate isosurfaces in 4-space. Section 3 describes the methods used for viewing the implicit surfaces in 4-space and varying a contour parameter *t*. Finally, section 4 presents statistics resulting from triangulating the implicit surfaces.

Department of Computer Science
Mississippi State University
chris@cs.msstate.edu, banks@cs.msstate.edu

2 The Contour Meshing Algorithm

This section presents simple recursive routines that can be combined to triangulate *k*-dimensional contours. The contours arise from the set of scalar-valued functions $f_i: \mathbf{R}^n \rightarrow \mathbf{R}$ with $f_i = c_i$ for constants c_i, where $1 \le I \le n\text{-}k$. The first routine triangulates the domain; the second locates the contour.

2.1 Splitting a Cell into Simplexes

Recall that a square is a 2-cell, a cube is a 3-cell, a "hypercube" is a 4-cell, and so on (figure 1). Also recall that a triangle is a 2-simplex, a tetrahedron is a 3-simplex, a "hypertetrahedron" is a 4-simplex, and so on (figure 2). Euclidean space $\mathbf{R}^n$ is conveniently tiled by *n*-cells, but we wish to split the cells into simplexes prior to contouring. Triangulating the cells avoids ambiguities [Nielson] that arise when a contour crosses a cell. There are various ways to perform the split [Allgower91]. We chose a method that

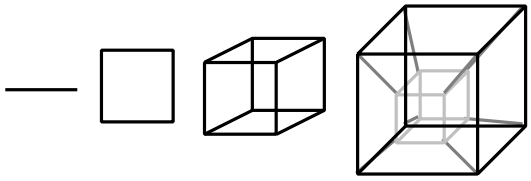

Figure 1. Examples of *n*-cells (*n* = 1, 2, 3, 4).

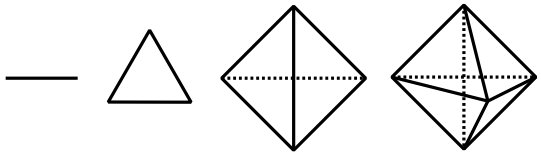

Figure 2. Examples of *n*-simplexes (*n* = 1, 2, 3, 4).

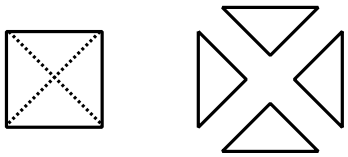

Figure 3. Splitting the 2-cell (left). The midpoint connects to each edge to produce four 2-simplexes.

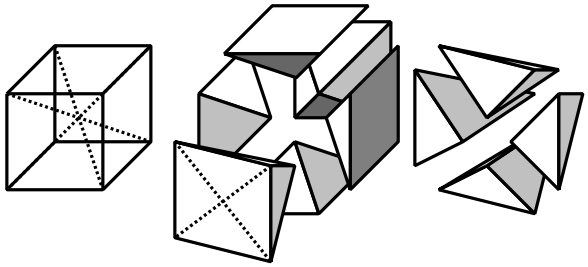

Figure 4. Splitting the 3-cell (left). The cell's midpoint connects to each face to produce six pyramids (middle). The midpoint of a pyramid's base produces four 3-simplexes (right).

can be formulated to work in arbitrary dimension and that can also be applied to other polytopes in addition to the *n*-cell.

Splitting a 2-cell

First consider the 2D case, splitting a 2-cell into four 2-simplexes (squares into triangles). Each edge in the cell can combine with the cell's midpoint to form a 2-simplex (figure 3). The process is analogous to barycentric subdivision of a simplex. The **splitC2** routine gives pseudocode for this operation in a 2-cell. The datatype for simplex and edge contains an array p[0..dim] of vertices.

```
splitC2 (square, simplex)
  simplex.p[2] ← midpoint (square)
  foreach edge in square do
    simplex.p[0..1] ← edge.p[0..1]
```

Splitting a 3-cell

Next consider the 3D case. We want to split a 3-cell into 3-simplexes (cubes into tetrahedra). Each face in the cell can combine with the 3-cell's midpoint to form a pyramid. If each face is further split into four triangles the result is four tetrahedra rather than one square pyramid, a step performed by the previous routine. The process is sketched in figure 4, and pseudocode is given as **splitC3**.

```
splitC3 (cube, simplex)
  simplex.p[3] ← midpoint (cube)
  foreach square in cube do
    splitC2 (square, simplex)
```

Splitting a 4-cell

Now consider the 4D case. We want to divide a 4-cell into 4-simplexes (hypercubes into hypertetrahedra). Again, we start by finding the midpoint of the cell. Then for each 3-cell "face" in the 4-cell (cube in the hypercube) we apply the previous routine, chopping the cube into its constituent tetrahedra. The tetrahedron, together with a midpoint, yields a 4-simplex. Pseudocode is given as **splitC4**.

```
splitC4 (hypercube, simplex)
  simplex.p[4] ← midpoint (hypercube)
  foreach cube in hypercube do
    splitC3 (cube, simplex)
```

Splitting an *n*-cell

This process generalizes to solve the problem of splitting *n*-cells into *n*-simplexes. The routines can be written recursively, passing each sub-cell to the next recursive level and terminating the subdivisions upon reaching the 1-cell (edge). Pseudocode for splitting the *n*-cell is given in split. Selecting the (*n*-1)-cells from an *n*-cell is simply a matter of combinatorics.

```
split (cell, simplex, n)
  if (n > 1) then
    simplex.p[n] ← midpoint (cell)
    foreach subcell in cell do
      split (subcell, vertex, n-1)
  else
    simplex.p[0..1] ← cell.p[0..1]
    /* simplex.p[0..n] now contains a simplex */
```

This midpoint-splitting scheme generates $2^{n-1}n!$ simplexes from an *n*-cell; there are other techniques [Allgower91] that generate only *n*! simplexes. The midpoint scheme is therefore easy to encode but inefficient in simplex production, a familiar trade-off. As section 4 shows, however, the actual number of contour-triangles produced is, in practice, much lower than this worst-case analysis would suggest.

2.2 Contouring the Simplexes

A contour of a 1-simplex (segment) is a single point *p* such that $f(p) = c$ for some constant *c*. Assume, without loss of generality, that $c = 0$. We will not treat the degenerate case where $f = 0$ identically across the entire simplex; the problem of simplexes not transverse to *f* is addressed elsewhere [Allgower91].

Contouring a 1-simplex

To determine whether the function *f* crosses zero on an edge, we check for a sign change at the endpoints. Thus, $f(v_1)\ f(v_2) \le 0$ implies (by the intermediate value theorem) that there is a zero-crossing in the interval between vertices v_1 and v_2. Note that the reverse implication does not hold; the mesh must be sufficiently fine to resolve a zero-crossing that lies within a simplex. In the pseudocode **contourS1** below,

the test for a zero-crossing is encoded by the **if**-statement. The contour point on an edge can be located with a linear approximation or with an iterative root-finding scheme. We leave the details to a "black-box" calculation performed by the routine **contourPoint**.

```
contourS1 (function, edge)
  if (function = 0) on edge then
    return contourPoint (function, edge)
```

Contouring a 2-simplex

A contour within a 2-simplex (triangle) is approximated by a segment. Each endpoint of the segment is a point that lies on an edge of the triangle. We loop over the edges (each edge being a sub-simplex of the triangle), locate zero-crossings, and then connect them to form a contour-segment in the triangle. The endpoints defining the contour-segment are found via a call to **contourS1**. Pseudocode for the 2D case is given below as **contourS2**.

```
contourS2 (function, triangle)
  foreach edge in triangle do
    point ← contourS1 (function, edge)
    append point to contour
  /* contour will be a line segment */
```

Contouring a 3-simplex

Bloomenthal describes how to find contours in 3-simplexes (tetrahedra) by enumerating combinations of zero-crossings into a table [Bloom94]. The table provides an explicit construction of a contour. By contrast, we seek a procedural approach that mimics the sequence of steps applied to a triangle. To construct the contour-polygon in a tetrahedron, we loop over its four triangular faces, then connect contour-segments found within them in order to form a contour-polygon in the tetrahedron. Note that the polygon may have either three or four edges. (The four-edge case may actually be non-planar, so calling it a polygon is technically an abuse of terminology.) The pseudocode is given below as **contourS3**.

```
contourS3 (function, tet)
  foreach triangle in tet do
    segment ← contourS2 (function, triangle)
    append segment to contour
  /* contour will be a polygon */
```

Contouring a 4-simplex

To find a contour in a 4-simplex, we loop through its 3-simplex "faces" and collect the contour-polygons they contain. Figure 5 illustrates the 3-simplexes that comprise a 4-simplex; the decomposition is a matter of combinatorics, and generalizes readily to *n*-simplexes. The contour-polygons form the boundary elements of the contour-polyhedron within the 4-simplex. Note that the "polyhedron" may actually be non-flat, even if it has flat polygonal faces. A contour-polyhedron in a 4-simplex that can assume a variety of shapes. These are shown in the appendix. The pseudocode **contourS4** demonstrates the method for finding contours in 4-simplexes.

```
contourS4 (function, hypertet)
  foreach tet in hypertet do
    polygon ← contourS3 (function, tet)
    append polygon to contour
  /* contour will be a polyhedron */
```

Contouring an n-simplex

As this sequence of examples shows, the edges and triangles of **contourS2** are promoted to triangles and tetrahedra in **contourS3,** then to tetrahedra and hypertetrahedra in **contourS4**. These individual routines can be subsumed by a single a recursive solution to find the contour in an *n*-simplex. The pseudocode is shown below as **contour**.

```
contour (function, simplex, n)
  if (n > 1) then
    foreach subsimplex in simplex do
      polytope ←
          contour (function, subsimplex, n-1)
      append polytope to contour
  else /* simplex is a line segment */
    if (function = 0) on simplex then
      return contourPoint (function, simplex)
```

Triangulating a contour-polytope

Once a contour-polytope has been extracted from a simplex, it can itself be approximated by simplexes. The routine **split** can be adapted to split a polytope into simplexes, and is given below as **triangulate**. Its only essential difference from **split** is that it loops over sub-polytopes (rather than sub-cells) along the boundary of the polytope.

```
triangulate (polytope, simplex, n)
  if (n > 1) then
    simplex[n] ← midpoint (polytope)
    foreach subpoly in polytope do
      triangulate (subpoly, simplex, n-1)
  else
    simplex[0..1] ← polytope[0..1]
```

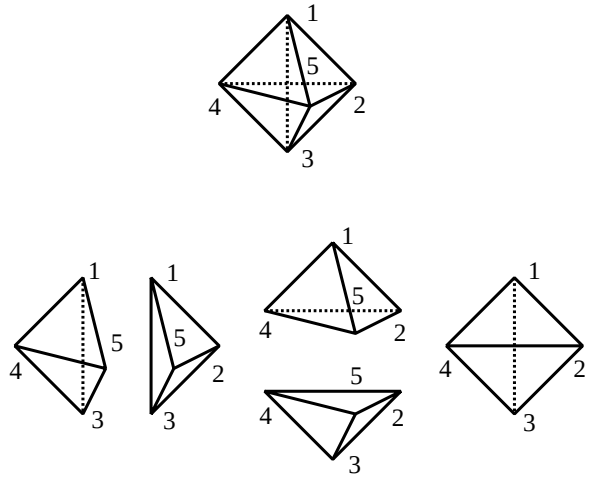

Figure 5. Decomposing the 4-simplex (top) into 5 tetrahedra (bottom). Vertices are numbered consistently to show the 5-choose-4 combinations of sub-simplexes.

Why would one want to triangulate the polytope? One reason is to provide simple graphics primitives for rendering an image on the screen. But the more important reason is that each simplex in a contour-polytope can be passed as an argument to the **contour** routine again. A function f_1 produces a contour-polytope which is split into simplexes. Each of the simplexes can be contoured against a function f_2 to find a lower-dimensional contour-polytope.

3 Contour Meshing Applied to CxC

This section presents the results of applying the Contour Meshing algorithm to the particular problem of triangulating a surface in 4-dimensional space. Such a surface arises in the context of studying complex curves.

A major area of interest to algebraic geometers is the structure of families of complex algebraic varieties; that is, the contours $\mathbf{C}_t$ defined by isovalues for a collection of polynomials parameterized by *t*. One contour can deform into another, but the deformation must preserve the algebraic structure of the variety. There are many difficult unsolved problems regarding families of varieties; an interesting case arises when a smooth variety deforms into a singular one (containing a cusp or a self-intersection).

Work of the last decade in the interaction between algebraic geometry and theoretical physics has increased interest in problems involving families of complex algebraic curves with additional structure such as vector bundles. The behavior of these bundles, as singularities develop in the underlying curves, has physical as well as mathematical significance. Even relatively simple questions are difficult to answer (for example, whether or not one of these curve-line bundle pairs can be deformed into any other such pair – i.e., whether the space parameterizing all these pairs is connected). Visualization methods that increase understanding of these families would be very interesting to mathematicians.

A complex curve, defined by $f: \mathbf{C}^2 \rightarrow \mathbf{C}$, can be represented by a surface in real 4-space. Each complex variable is specified as a real/imaginary pair, hence the domain is 4-dimensional and the function can be written as $f: \mathbf{R}^4 \rightarrow \mathbf{R}^2$. The real-valued constraints *real(f)* = 0 and *imaginary(f)* = 0 serve as the two functions f_1 and f_2 for the Contour Meshing algorithm. We first enumerate 4-cells that tile the space $\mathbf{R}^4$, then split each cell into 4-simplexes and solve $f_1 = 0$ to extract a contour-polyhedron. Its constituent tetrahedra are then contoured against $f_2 = 0$ to extract the contour-polygons that form the implicit surface.

Consider the reciprocal function $xy = t$ (so named because of its explicit form $y = t/x$), for real-valued x, y, and parameter t. Figure 6 shows contours for this function at three values of t. When t reaches zero the contour develops a singularity (the self-intersection at the origin), since $xy = 0$ has solutions $x = 0$ and $y = 0$ corresponding to the x- and y-axes.

Now consider complex-valued variables x, y, and parameter t. The function $xy = t$ defines an implicit surface in the 4-dimensional space spanned by two complex planes. When $t = 0$ the surface contains an isolated singularity at the origin, with two sheets intersecting transversely. It is especially difficult to visualize the manner in which two surfaces intersect in a single point; such an intersection can never be embedded in $\mathbf{R}^3$. Figures 7 and 8 show the results.

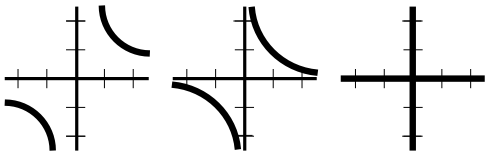

Figure 6. Contours for the real-valued reciprocal function $xy = t$ ($t = 2, 1, 0$).

Figure 7. Contours for the complex-valued reciprocal function $xy = t$ (see also plate 1).

Figure 8. Stereographically projected contours for complex-valued $xy = t$ (see also plate 2).

3.1 Stereographic Projection

After the Contour Meshing algorithm has generated an isosurface, we would like to investigate its shape in $\mathbf{R}^4$. Moreover, we would like to see the entire surface, despite its infinite extent.

To view the entire surface *f(x,y)* = 0, we must map an infinite space onto a finite one through a process called compactification. One way to accomplish this compactification is to use stereographic projection.

Stereographic projection maps $\mathbf{R}^n$ onto the n-dimensional sphere $\mathbf{S}^n$, which can be imbedded in $\mathbf{R}^{n+1}$. Consider projecting the plane $\mathbf{R}^2$ onto a 2-sphere $\mathbf{S}^2$ in $\mathbf{R}^3$ (figure 9). The south pole rests on the plane. A line through the north pole and a point p on the plane must meet a point q on the sphere; q is the stereographic projection of p. Points on the plane just below the sphere are mapped to the southern hemisphere, while points near infinity are mapped to a neighborhood of the north pole.

Stereographic projection leaves a hole at the north pole. Filling the hole compactifies the space. We explicitly cover the hole with polygons which connect the implicit surface's boundary (after projection) to the north-pole. Technically, these hole-filling triangles form a cone over the surface's boundary. Figures 7 and 8 show the complex function $f(x,y) = xy - t = 0$, before and after stereographic projection (with the hole filled).

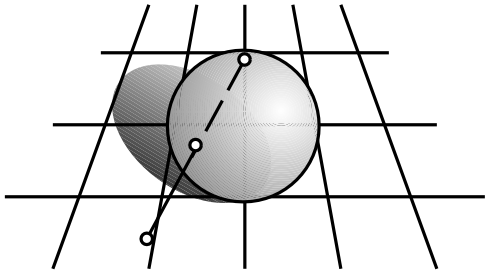

Figure 9. A stereographic projection.

3.2 Manipulating the surface

After stereographic projection, our complex-valued surface lies on a 4-sphere in $\mathbf{R}^5$. $\mathbf{R}^5$ allows 5 degrees of freedom for translations and 10 for rotation; four-dimensional space $\mathbf{R}^4$ allows 4 and 6 degrees of freedom, respectively. It is difficult to manipulate a graphical object in $\mathbf{R}^4$ using an input device in our 3-dimensional world; it is even harder if the object lies in $\mathbf{R}^5$. We therefore fix a particular projection of the surface from $\mathbf{R}^5$ to $\mathbf{R}^4$, where we can view it interactively using an interface developed in the Fourfront system [Banks]. The mechanics of interacting with surfaces in 4-space are described elsewhere [Banks] [Hanson] [NDview].

3.3 Time Varying Surfaces

The impetus for this work was to examine a set of isosurfaces parameterized by t, where at $t = 0$ the isosurface contains a singularity. We create a separate meshed surface for several values of t. All meshes are loaded into machine memory. On a graphics workstation the 4D viewer then animates them as a flipbook of surfaces. This approach requires a long mesh-generation step, and it limits the set of values of t that the user can animate. Ideally, only the isovolume through which t varies would be precomputed. That is, we could compute the one isovolume for $f(x,y,t) = 0$. Subcontours could then be extracted on-the-fly depending on the exact value of t that the user specifies. Then the interactive viewer could perform the final Contour Meshing to extract the isosurface. On a multiprocessor platform (or distributed over a cluster of machines) this might be a feasible approach, but a single-processor workstation is too slow to extract the isosurface in real time. The viewing system already requires all available computing power from a single-processor SGI Onyx Reality Engine or an SGI Indigo^2 High Impact, even though the isosurfaces at different values of t have been precomputed.

In his first session he expressed surprise that the family of isosurfaces deformed with a twist as it approached the singularity. He was also interested to see the actual geometry of the surface patches in the neighborhood of their 1-point intersection. Although he is well acquainted with the algebraic properties of the complex curves, this was his first experience visualizing them.

4 Application Performance and Statistics

Our visualization system contains two parts, one to extract the contours and one to interact with them. Both were developed on an SGI Indigo^2 with a 250MHz MIPS R4400 processor, 192 Mbytes of RAM, and a High Impact graphics engine. The mesh generator was written in C++ and the viewer in C and OpenGL. The following statistics describe the complex reciprocal function $f(x,y) = xy - t = 0$.

Table 1. Distribution of contour-polyhedra and contour-polygons resulting from the complex reciprocal function.

Triangles (out of 386,080) from	**%**
Case 2 (tetrahedral contour)	30.13
Case 3 (pyramid-shaped contour)	0.37
Case 4 (prism-shaped contour)	66.81
Case 5 (degenerate 4-simplex)	0.00
North-pole's cone over the boundary	2.69
Types of contour-polygons in 3-simplexes	
3-sided polygons	81.69
4-sided polygons	18.31

The statistics in tables 1 and 2 describe the mesh elements and system performance. The complex reciprocal function was evaluated within a 4-cell of size 4x4x4x4 centered about

the origin in $\mathbf{R}^4$ for 51 values of t ranging from -1 to 1. Table 1 shows what percentage of the triangles in the implicit surface arise from the various cases of contour-polyhedra in a 4-simplex (see the appendix), and what percentage of the polygons arise from the cases of contour-polygons in a 3-simplex. Tetrahedra and prism-shaped isovolumes are shown to dominate. Table 2 summarizes mesh-generation statistics for the same complex reciprocal function. The triangle-counts in table 2 do not include the polygons required to fill the hole in the stereographic projection. The table shows that fewer than 30 triangles were found, on average, in each 4-cell. This is significantly fewer than the worst-case analysis of $2^3 4!$ contour-polyhedra per 4-cell, 18 tetrahedra per polyhedron (case 4 with midpoint-triangulation), 4 contour-polygons per tetrahedron, which yields a total of 13,824 potential triangles per 4-cell.

Table 2. Mesh generation statistics for the complex reciprocal function.

$xy - t = 0$ **for complex** x, y, **and** t	
Number of values assumed by t	51
Number of 4-cells per value of t (4x4x4x4)	256
Number of 4-cells traversed (51×4^4)	13,056
Total number of triangles in 51 meshes	375,695
Average number of triangles per 4-cell	28.78

Increasing the resolution of the spatial tiling dramatically increases the number of cells to be traversed; with h cells in each of n dimensions, h^n cells result. However, the final contour mesh only triangulates the cells that contain the k-dimensional contour, so $O(h^k)$ is a more reasonable estimate of the asymptotic complexity of the mesh. Triangulating the complex reciprocal function (in a neighborhood of the origin) at h = 2, 4, and 8 cells per dimension yields 896 triangles in 2^4 4-cells, 4768 in 4^4 4-cells, and 16,992 in 8^4 4-cells (see plate 3). This production is consistent with quadratic growth of about $(16h)^2$ triangles.

Because it performs 4D-transformations, the interactive viewer is unable to match the rendering speed of ordinary 3D applications. Although SGI graphics engines can perform 4D rotations in hardware, they are not designed to perform 4D translations or 4D-to-3D projections. The interactive viewer renders approximately 60,000 4D triangles per second, which is about 40% of the performance for 3D triangles. For the complex reciprocal example, this limits us to an average of about 8 frames per second – barely enough to be interactive. We find that this performance, at least on our SGI workstations, is essentially independent of window size.

5 Future Work

One area of future work is to improve the implicit surface by recursive subdivision and to adjust the position of midpoints to satisfy $f_i = 0$. Because of the large number of polygons Contour Meshing produces, a parallel implementation of the algorithm could speed up this phase significantly. Another possibility for mitigating the burden of the number of polygons produced is to decimate the triangular mesh. Since the isosurfaces lie in $\mathbf{R}^4$, we are examining ways to modify existing 3D methods for mesh simplification.

6 Conclusions

Contour Meshing is an algorithm for triangulating a contour from a collection of functions in arbitrary dimension. The algorithm locates a contour-polytope in a simplex. Splitting a cell into simplexes and splitting a polytope into simplexes are steps that allow the algorithm to be applied recursively over multiple functions in large dimensions. We examined the sample case of triangulating an algebraic complex curve (producing a surface in real 4-dimensional space) and viewing the result interactively on a graphics workstation. The viewing system maintains acceptable interactivity for a modest number of 4-cells tiling the domain. The mesh-generator and interactive viewer are in use by an algebraic geometer interested in the exploration of complex curves.

Acknowledgments

We gratefully acknowledge the guidance of Tyler Jarvis (Brigham Young University, Department of Mathematics), who supplied the original problem of visualizing vector bundles over singular complex curves. We thank Jules Bloomenthal for insights into implicit-mesh algorithms. We thank the reviewers for pointers to a wealth of background material that significantly improved the quality of this paper.

References

[Allgower90] E. L. Allgower and K. Georg, Numerical Continuation Methods: An Introduction, Springer, Berlin, 1990.

[Allgower91] E. L. Allgower and S. Gnutzmann, Simplical pivoting for mesh generation of implicitly defined surfaces, *Computer Aided Geometric Design*, 8(4):305-325, 1991.

[Banks] David C. Banks, Interactive manipulation and display of two-dimensional surfaces in four-dimensional space. *1992 Symposium on Interactive 3D Graphics*, ACM Press, pp. 197-207, 1992.

[Bloom88] Jules Bloomenthal, Polygonization of implicit surfaces. *Computer Aided Geometric Design*, 5(4):53-60, 1988.

[Bloom94] Jules Bloomenthal, An implicit surface polygonizer. *Graphics gems IV*. Boston: Academic Press, 1994.

[Cairns] S. S. Cairns, Triangulation of the manifold of class one. *Bulletin of the American Mathematical Society*, 41:549-552, 1935.

[Hanson] Andrew Hanson and Robert A. Cross, Interactive visualization methods for four dimensions. *Proceedings of Visualization '93*. pp. 196-203, 1993.

[Hoffman] Christoff Hoffman and Jianhua Zhou, Visualizing surfaces in four-dimensional space. *Computer Aided Design* 23:83, 1991.

[Holt] Olaf Holt, Introducing ... NDView 1.0, http://www.geom.umn.edu/software/geomview /docs/NDview/, The Geometry Center, University of Minnesota.

[Koide] Akio Koide, Akio Doi and Koichi Kajioka, Polyhedral approximation approach to molecular orbital graphics, *Journal of Molecular Graphics*, 4(3):-, 1986.

[Koopman] B. O. Koopman and A. B. Brown. On the covering of analytic loci by complexes. *Transactions of the American Mathematical Society*, 34:231-251, 1932.

[Lorensen] William Lorensen and H. E. Cline, Marching cubes: a high resolution 3-D surface construction algorithm. *Proceedings of SIGGRAPH '87*, in *Computer Graphics* 4, 1987.

[Nielson] G. M. Nielson and Bernd Hamann,The asymptotic decider – resolving the ambiguity in marching cubes. *Proceedings of Visualization '91*, IEEE Computer Society Press (October), 1991.

[Waerden] B. L. van der Waerden, Topologische Begründung des kalküls der abzählenden geometrie. *Mathematische Annalen*, 102:337-362, 1929.

[Wicklin] Frederick Wicklin and Eric Streed, Pisces Project at the Geometry Center, http://www.geom.umn.edu /software/pisces/, The Geometry Center, University of Minnesota.

Appendix: Cases of 4-Simplex Contours

The number of 3-simplexes produced by **triangulate** can be reduced if the possible isovolumes of a 4-simplex are considered. It is also useful to consider the cases when constructing a table-driven contouring method, like enumerating the cases in Marching Cubes. There are five cases:

1. No contour;
2. A tetrahedron (figure 10 shows one, but not the only, case);
3. A pyramidal solid (figure 11 shows the only case);
4. A prism-shaped solid (figure 12 shows the only case);
5. The entire 4-simplex ($f = 0$ at all vertices).

A tetrahedron does not need to be midpoint-split into more tetrahedra. Only a non-simplex polytope needs to be split. Figures 10-12 show how this small optimization avoids splitting tetrahedra that can result from contouring and splitting.

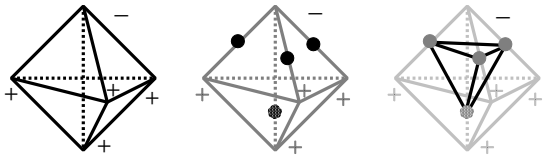

Figure 10. Case 2. In this 4-simplex (left) there are four vertices where *f* is positive and one where it is negative. Zero-crossings are shown as dots (middle). The resulting contour-polyhedron (right) has four triangular faces.

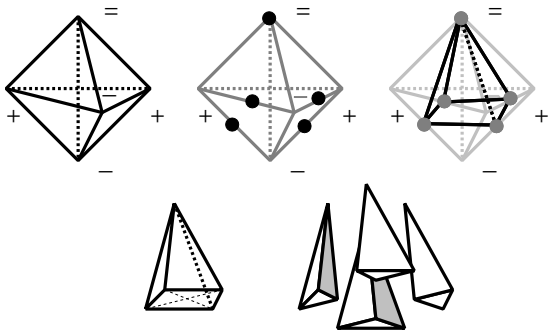

Figure 11. Case 3. In this 4-simplex (top left) there are two vertices where *f* is positive, two where it is negative, and one where it is exactly zero. Zero-crossings are shown by dots (top middle). The contour-polyhedron (top right) has four triangular faces and one rectangular face. This pyramid (bottom left) can be midpoint-split once at its base.

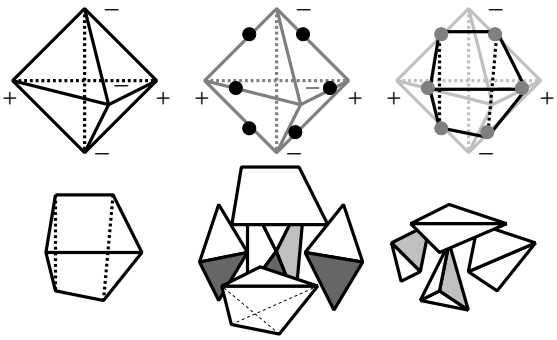

Figure 12. Case 4. This 4-simplex (top left) contains zero-crossings (top middle) that produce a contour-polyhedron with two triangular and three rectangular faces (top left). This prism (bottom left) can be midpoint-split into three pyramids and two tetrahedra (bottom middle). Only the pyramid-shaped components need to be split (bottom right).

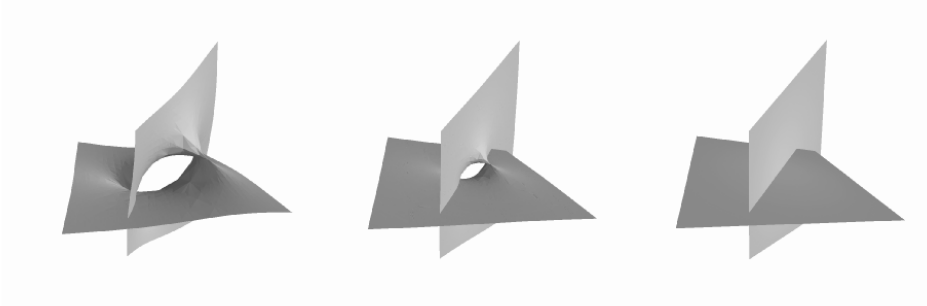

Plate 1. The complex reciprocal function $xy - t = 0$ realized as a surface in $\mathbf{R}^4$. Two patches of the surface nearly intersect (left and middle) as t approaches 0. They intersect as two planes at the origin (right) when $t = 0$. The 3D image makes the intersection appear to occur along a line; interactively rotating the surface in $\mathbf{R}^4$ demonstrates, however, that the only fixed point of the intersection is the origin.

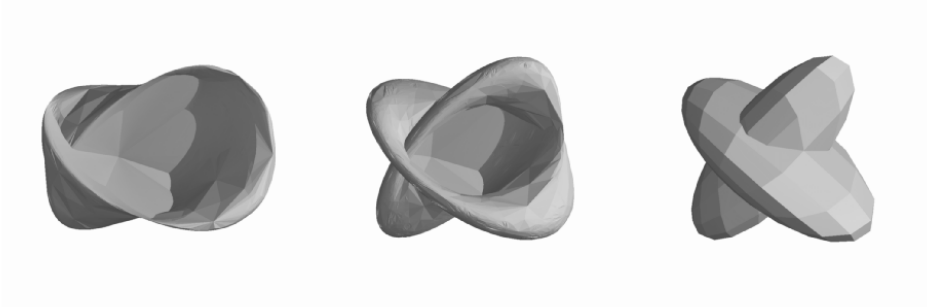

Plate 2. The surfaces shown in plate 1 are here stereographically projected to the 4-sphere in $\mathbf{R}^5$ and the neighborhood of the north pole is then covered. The intersecting planes in $\mathbf{R}^4$ become intersecting spheres after the stereographic projection (right).

Plate 3. The upper-left surface in plate 1 is shown here from a different viewpoint, revealing the hyperbolic shape (along the silhouette) of the real-valued reciprocal function contained within it. The surface mesh is generated over the same domain at three different resolutions of the spatial subdivision. From left to right: 2^4 4-cells yield 896 triangles; 4^4 4-cells yield 4768 triangles; 8^4 4-cells yield 16,992 triangles. The triangles are color-coded (see color plates) according to the individual cases of contour-polyhedra from which they arise: green = case 2 (tetrahedron); blue = case 3 (pyramid); orange = case 4 (prism).