Computing Contour Trees in All Dimensions

Hamish Carr
Dept. of Computer Science
Univ. of British Columbia

Jack Snoeyink
Dept. of Computer Science
Univ. of British Columbia
& UNC Chapel Hill

Ulrike Axen
School of EECS
Washington State Univ.

Abstract

We show that contour trees can be computed in all dimensions by a simple algorithm that merges two trees. Our algorithm extends, simplifies, and improves work of Tarasov and Vyalyi and of van Kreveld et al.

1 Introduction

Many imaging technologies and scientific simulations produce data in the form of sample points with intensity values. This data may be converted into geometric models by segmentation, often involving thresholding or taking level sets, or the data may be studied in situ using similar tools. In this paper, we focus on one tool that can help in choosing threshold values or in interactive exploration of such data: the *contour tree*.

Contour trees were proposed by van Kreveld et al. [8] for computing isolines on terrain maps in geographic information systems. With terrain maps, a surface model is computed from elevation values at sample points in the plane. Isolines, often called contours, are the curves consisting of points at a given height that can be seen on any topographic map. Contours can be traced from a surface model relatively easily, given a starting point, or "seed" on each. van Kreveld et al. use the contour tree to generate "seed sets" for any query height value.

We are particularly interested in data from X-ray crystallography for studying protein molecules. Our data arrives as points in $\mathbb{R}^3$, either on a lattice or irregularly sampled, with intensity values. These values are extended to $\mathbb{R}^3$ by using the data points in a decomposition of $\mathbb{R}^3$ into simplices and interpolating linearly.

The contour tree allows us not only to compute seed sets for tracing isosurfaces, but also gives important values of the parameter where topological changes occur in the level sets; these changes may correspond to important phenomena such as chemical bonds. While van Kreveld et al. do discuss the extension of their approach to $\mathbb{R}^3$, their algorithm runs in quadratic time, which is prohibitive.

Tarasov and Vyalyi [6] gave an $O(N \log N)$ algorithm for computing contour trees in $\mathbb{R}^3$, where N is the number of simplices in the decomposition of the data. They resolve all multiple singularities and replace them by simple singularities, then perform three sweeps through the data. We later describe their algorithm and the handling of singularities in more detail, but their approach multiplies the number of simplices by a factor of 360, which is again prohibitive.

Our algorithm for contour trees begins with Tarasov and Vyalyi's idea of three passes through the data, but makes the following simplifications and improvements. The first two sweeps can be replaced by linear-time recursive procedures that produce two trees containing the nodes of the contour tree. By sorting only these nodes, we can form the contour tree by a simple merge procedure. The resulting algorithm handles multiple singularities and extends to all dimensions. Because there are some applications in which multiple singularities must be replaced by simple singularities, we also observe that Tarasov and Vyalyi's approach to resolving singularities can be extended to all dimensions.

After preliminary definitions in Section 2, we define contour trees and look at their properties in Section 3. We give our algorithm to construct contour trees in Section 4. Our observation on resolving singularities is in Section 5.

2 Definitions and Preliminaries

Suppose that we are given a set of n points $\{p_1, p_2, \ldots, p_n\}$ in a fixed-dimensional space $\mathbb{R}^d$, with corresponding scalar measurements $\{h_1, h_2, \ldots, h_n\}$. We assume that the h_i are unique, perhaps by perturbation of our data.

To extend the data to the entire space, we choose a simplicial mesh with vertex set $\{p_1, p_2, \ldots, p_n\}$, and a piecewise-linear function f to interpolate values from the data points given, such that

1. f is a linear function within each simplex, and
2. $f(p_i) = h_i$ for all $i = 1, \ldots, n$.

A *level set of f for some value x* is the set $\{p \in \mathbb{R}^d | f(p) = x\}$. Topologically, a level set may consist of 0, 1, or more connected components. Under our assumptions of uniqueness and linear interpolation, these connected components will be of dimension $\leq (d-1)$.

In 2-D, a connected component is called an *isoline,* and in 3-D an *isosurface.* We will sometimes use *contour* as a general term for a connected component of a level set in a space of arbitrary dimension.

The field of Morse theory [1, 2, 4] studies the changes in topology of level sets of f as the parameter x changes. Points at which the topology of the level sets change are called *critical points.* Morse theory requires that the critical points are isolated – i.e. that they occur at distinct points and values. A function that satisfies this condition is called a *Morse function.* All points other than critical points are called *regular points* and do not affect the number or genus of the components of the level sets. Our definition of f – as a linear interpolant over a simplicial mesh with unique data values at vertices – ensures that f is a Morse function, and that the critical points occur at vertices of the mesh [1].

If we think of the parameter x as time and watch the evolution of the level sets of f over time, then we will see components of level sets appear, split, change genus, join, and disappear. The *contour tree*, which we define next, is a graph that tracks components of the level set as they split and appear or join and disappear.

3 Definition and properties of the contour tree

Here we define the contour tree and augmented contour tree, and consider their properties.

3.1 The contour tree

The *contour tree for a Morse function* is defined as a graph in which:

1. each leaf vertex represents the creation or deletion of a component at a local extremum of the parameter
2. each interior vertex represents the joining and/or splitting of two or more components at a critical point
3. each edge represents a component in the level sets for all values of the parameter between the values of the data points at each end of the edge.

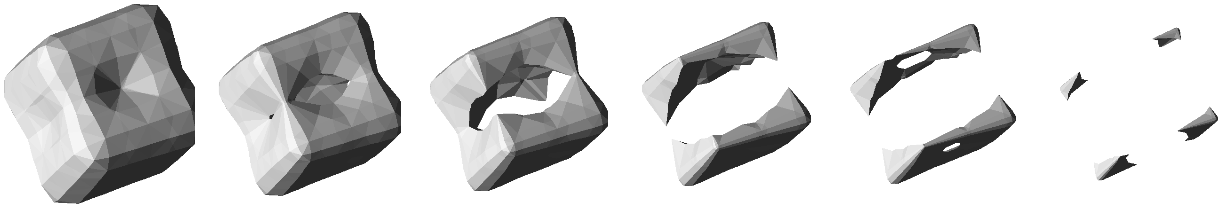

Figure 1: Level sets of $f(x)$ as x increases

We refer to the vertices and edges of the contour tree as *supernodes* and *superarcs,* respectively. This graph has been shown to be a tree [8], hence the name *contour tree.*

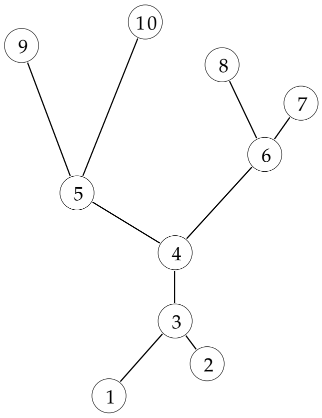

Figure 2: Contour tree for Fig 1

Figure 1 illustrates the level sets of a function that, as the parameter x increases, evolve from a solid, to a hollow ball, to a single component, to two "cushions," to two rings, to four sticks. Figure 2 illustrates the corresponding contour tree. Starting from the top and decreasing the parameter, we see four leaves corresponding to the sticks. These merge in pairs (5,6) forming rings and cushions. (Note that the changes in genus from disk to torus to disk are not reflected in the contour tree — even though the connected components change topology, each can still be traced from a single seed point.) The cushions join at (4). Then there is one component of the level set until at (3) it encloses a hollow, at which point the level set splits into an inner boundary and an outer boundary. The inner boundary then contracts and disappears at (2).

3.2 The contour tree as recording topological events

We can describe the contour tree as recording what happens to components of the level set in response to certain *events* that correspond to the critical points, if we continue to think of the parameter values as time. First, we need some notation to describe the components.

A component is created either by appearing, separated from all existing components, or by an existing component splitting to become two or more new components. Similarly, a component is destroyed either by collapsing down to a single point and disappearing, or by joining with another component to make a new, combined component. Each component is assigned a name, C_α^β, based on the time α when it is created and the time β when it is destroyed. If we know only the creation time, α, then we say that the name C_α is partially assigned.

Thus, when the parameter h becomes equal to the value of a critical point, the set of possible changes is strictly limited to:

i) A new component C_h is created at a local minimum.

ii) An existing component, C_k is destroyed at a local maximum: we will rename the component C_k^h.

iii) Two or more existing components, $C_{k_1}, C_{k_2}, \ldots C_{k_m}$, are joined into a new component at a saddle point. These components are destroyed – their names are completed to $C^h_{k_1}, C^h_{k_2}, \ldots C^h_{k_m}$ – and a new component C_h is created.

iv) The topological genus of an existing component is changed at a saddle point.

v) An existing component C_k is split into two or more new components at a saddle point. This involves destroying C_k, renaming it C^h_k, and creating several new components C_h.

vi) Any combination of iii) – v). Both splits and joins can occuring at a highly-degenerate multi-saddle.

We treat vi) saddle points with splits, joins, and changes of genus as consisting of changes of genus to zero or more of the components involved, followed by an optional join, an optional split, and optional changes of genus for all components involved. This simplifies processing of such points. As we will note later, changes of genus do not affect the contour tree.

If we could determine types of events, then we could construct the contour tree by a sweep through the parameter values. Each component of the level set is created at a critical point of type i), iii), or v), and is destroyed at a critical point of type ii), iii) or v). We call such a critical point a *supernode*. For each component, we connect the supernode where it is created and the one where it is deleted by an edge called a *superarc*. The components then have a 1-1 relationship with the superarcs.

One important observation can be made based on the treatment of these events:

Lemma 3.1 *The completed names for contour components are unique.*

Proof: The join events create names $C^h_{k_1}$ and $C^h_{k_2}$ only when partial components C_{k_1} and C_{k_2} are different. By the uniqueness of event values, we know that $k_1 \neq k_2$. ∎

3.3 The augmented contour tree

For some purposes, such as the generation of isosurfaces, information about regular points is also required. We augment the contour tree with the regular points to produce an *augmented contour tree*.

For each component, we sweep through the space from the value at which it appears to the value at which it disappears. Each data point swept through by this component is then assigned to the superarc to which the component corresponds. These points become *nodes* in the contour tree. Clearly, the supernodes will also be nodes: thus, all points in the dataset are nodes.

Along each superarc, we sort the associated nodes, and connect them in sorted order by *arcs*. This constructs a single path from the supernode at one end of the superarc to the other.

3.4 Previous work

van Kreveld et al. [8] reported the first efficient algorithm for constructing contour trees. This algorithm performs the extraction in $O(N \log N)$ time in 2-D data fields, and $O(N^2)$ time in higher dimensions, where N is the number of simplices (triangles) in the mesh of the n data points. The algorithm performs a sweep from low to high value, maintaining each component of the level set, and examines the data set locally to determine when saddle points are encountered and how to deal with them. Runtime is kept to $O(N \log N)$ in the plane by a clever method of merging the contours. Multi-saddle points are treated as a set of ordinary saddle points.

Tarasov and Vyalyi [6] presented a $O(N \log N)$ algorithm for 3-D data fields. Their algorithm performs three sweeps: one sweep to identify joins, a second to identify splits, and a third to combine the results of the two sweeps. Again, the level set is maintained at all times during the sweep. Multi-saddle points are dealt with by a complicated preprocessing step (see Section 5). Runtime is again kept to $O(N \log N)$ by the same method of merging the contours. Finally, boundary effects at the edge of the dataset are handled by special cases inside the algorithm.

In both algorithms, two factors contribute to the runtime: the initial sort takes $O(n \log n)$ time, and maintaining the level sets takes $O(N \log N)$ time. Bounds on number of simplices, N, in terms of the number of vertices, n, in fixed dimensions are $N = \Omega(n)$ and $N = O\left(n^{\lceil d/2 \rceil}\right)$. In any fixed dimension, it is possible to construct a mesh such that $n = \Theta(n)$.

4 A new contour tree algorithm

We propose a new algorithm for constructing augmented contour trees and augmented contour trees with the following characteristics:

1. Input is assumed to be a simplicial mesh of n vertices and N simplices, with data values measured at each vertex,
2. Time requirements of $O(n \log n + N\alpha(N))$ for constructing augmented contour trees, in any number of dimensions,
3. Time requirements of $O(n + N\alpha(N) + t \log t)$ for constructing a contour tree of t nodes, in any number of dimensions,
4. Space requirements of $O(N)$ for the mesh and $O(n)$ additional working storage,
5. Simple treatment of multi-saddle points.

We will describe the algorithm in three subsections. We follow Tarasov and Vyalyi [6] by first identifying contour joins and splits, but we build a *join tree* and a *split tree*, as described in Section 4.1. By merging these two trees, in Section 4.2, we obtain the contour tree. We discuss some implementation issues in Section 4.3.

4.1 Join and split trees

Define a *join component* to be a connected component of the set $\{p \in \mathbb{R}^d \mid f(p) \leq x\}$. We will label a join component J_α^β if it is created at α and destroyed at β – where we think of the parameter as time in the same way as when we name level set components. By this definition, if two points belong to the same component of the level set, then they must belong to the same join component. Thus, each join component corresponds to at least one component of the level set, and possibly more.

Define the *join tree* as a graph whose edges represent join components. One vertex, the root of the tree, represents the entire space. Other leaf vertices represent the creation of a join component at a local minimum, and internal vertices represent the merge of two or more join components. Since components can only merge, it is clear that this graph is a tree.

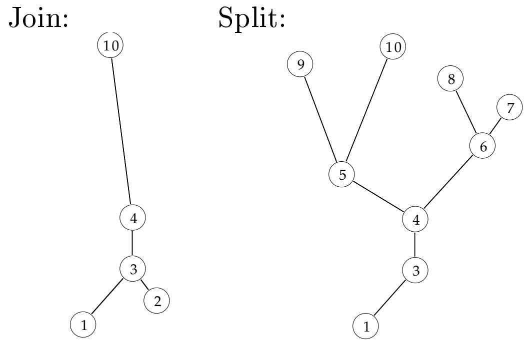

Figure 3: Join and split trees for the example of Figure 1

We also define the *split tree*, which is what we obtain when we construct the join tree using components of the sets $\{p \in \mathbb{R}^d \mid f(p) \geq x\}$ in order of decreasing parameter x. Together, the join and split trees contain all the supernodes of the contour tree. Figure 3 illustrates the join and split trees for our example from Figure 1.

Lemma 4.1 *The join and split trees for a Morse function f have the following properties:*

1. *Each node in the join or split tree is a supernode in the contour tree.*
2. *Each edge in the join or split tree represents a union of components from the contour tree.*
3. *Suppose that a v is a non-root leaf in the join or split tree that is not an internal node of the other tree. The join or split edge incident on v represents a single component of the contour tree.*

Proof: We can establish these properties by observing the Morse events. Details omitted for this abstract. ∎

The join tree can be constructed in nearly-linear time.

Lemma 4.2 *The join tree can be constructed using union-find in $O(N\alpha(N))$ operations.*

Proof: We must identify the critical points in the mesh and their parameter values, and decide which are nodes of the join tree. Since critical points occur only at mesh vertices, it suffices to check, at each vertex p_i, whether p_i is a local minimum, or whether there are two or more components that join when the parameter reaches the parameter value h_i. By our interpolation, it suffices to know if two or more vertex neighbors of p_i are in different components for $h_i - \epsilon$. This can be tracked using the set union-find structure of Tarjan [7].

We could perform a sweep to determine our unions and finds, but that would require sorting all parameter values. To avoid this, we use the following recursive algorithm. Start by placing on a stack all vertices whose parameter values are greater than their neighbors. Then, to determine the component for a vertex v, recursively determine the components for all neighboring vertices of lower parameter values. The components for some of these vertices may have already been computed, in which case the stored value is used and that branch of the recursion returns immediately.

If there are no neighbors with lower parameter values, then v is a local minimum that begins a new component. If all neighbors with lower parameter values are in the same component, then v is not a join node. On the other hand, if these neighbors belong to two or more components, then v is a join node and we perform a union operation on the neighbors.

This recursion terminates, because it always progresses to vertices of lower parameter values. Moreover, recursion starts once from each vertex. This implies that it takes $O(N)$ union-find operations. These operations take $O(N\alpha(N))$ time in total, where α is the slowly-growing inverse Ackermann function. ∎

Actually, one can refine the analysis of running time to $O(N\alpha(t, N))$, where t is the number of local minima, which is one greater than the number of unions performed. In dimension two, special union-find algorithms can eliminate the superlinear factor. (In practice, however, these would probably require more operations than standard union-find.)

We may wish to augment the join tree with extra internal nodes corresponding to some or all of the vertices in the mesh. By doing so we form an *augmented join tree.*

Lemma 4.3 *The augmented join tree with t nodes can be constructed in $O(N\alpha(N) + t\log t)$ operations.*

Proof: As we construct the join tree, we may make the association between vertices and the join components that first contain them. To augment the join tree with the nodes corresponding to these vertices, we simply need to refine the associated edges in the join tree. We sort the new nodes by parameter value and insert them as degree two vertices in their associated edges. ∎

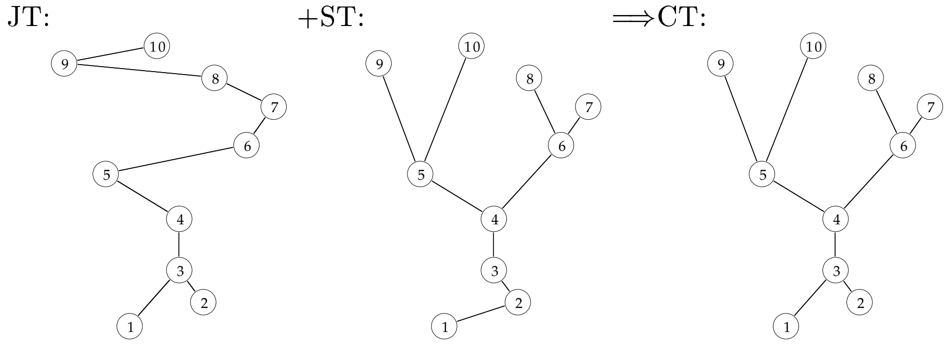

Figure 4: Augmented join and split trees merge to form the contour tree

The *augmented split tree* is defined and constructed in a similar manner. In the next section, we show how to form the contour tree from the augmented join and split trees.

4.2 Merging to form the contour tree

Use the algorithm of the previous section to compute a join tree and a split tree, and augment each with the nodes of the other to form the augmented join tree JT and augmented split tree ST. Figure 4 illustrates the results on our example from Figure 1. In this section we show how to merge JT and ST in linear time to form the contour tree, CT. If one augments JT and ST with all vertices of the mesh, then the same merge algorithm computes the augmented contour tree.

We identify a leaf in JT or ST that we can add as a supernode to CT, and remove from both JT and ST. We proceed inductively, generating one additional contour tree superarc at each step.

Lemma 4.4 *The join and split trees, JT and ST, can be merged to form the contour tree in time proportional to their size.*

Proof: We assume that we have augmented join and split trees, JT and ST, which contain all nodes for the portions of the contour tree CT that have not yet been constructed. At the leaves of JT and ST, we may have some portions of CT constructed.

We also assume that if a non-root leaf node v of JT (or ST) is incident on more than one unconstructed contour component, then v is a split (or join) node in the other tree. This holds initially by Lemma 4.1(3).

Choose a non-root leaf of JT or ST that is not a split/join node of the other tree. Since there are more leaves than split/join nodes, this can always be done. Because the the cases are symmetric, we may assume that a leaf v of JT is chosen, and not the root.

We move v and its incident edge from JT to the contour tree CT. In ST, either v is a degree 2 node or v is the root. In the former case, we suppress v in ST while maintaining the connection; in the latter we delete v from ST. We then have restored the property that JT and ST are trees on the same set of nodes.

We must argue that any node that becomes a leaf in one of the trees and has more than one incident contour component must be a split/join node in the other tree. In fact, the only node u that can possibly become a leaf by our changes is the parent node of v in join tree JT. If two contour components start from u, however, then u is a split node, and appears as such in ST. Thus, our merge can proceed by induction. ∎

If we assemble the pieces, we obtain the following results.

Theorem 4.5 *The augmented contour tree for a function on n data points, interpolated over a mesh with N simplices in* $\mathbb{R}^d$*, for fixed dimension d, can be computed in* $O(n \log n + N\alpha(N))$ *time. If the contour tree has t nodes, it can be computed in* $O(t \log t + N\alpha(N))$ *time. Both algorithms use* $O(n)$ *working space in addition to the* $O(N)$ *for the mesh.*

4.3 Implementation issues

In the full paper we report in more detail on the implementation issues for X-ray crystallography applications. These include implicitly decomposing a 3D lattice into simplices, embedding the data set in a field of zeros to avoid special boundary conditions, and perturbation to make all values unique.

5 Resolving multiple singularities

The algorithm described by Tarasov and Vyalyi [6] requires *simple singularities*, so they describe a method for breaking multi-saddle points into multiple simple singularities in time $O(N \lg N)$. Although our algorithm handles multi-saddles, their method is of independent interest for computation of Morse singularities in higher dimensions; if non-simple singularities are resolved, then a general function on a complex K is a Morse function. We therefore briefly show that their method applies in all dimensions. We assume familiarity with concepts of PL topology such as barycentric subdivisions, star, and link [3].

We first summarize the subdivision and perturbation given in [6] and extend it trivially to general dimensions. We then considerably simplify the proof that this method resolves non-simple singularities, and we extend it to all dimensions. Assume that K is a m-dimensional simplicial complex, $m \geq 3$, in $\mathbb{R}^d$ and f is a general function on K, (i.e., $f(v) \neq f(w)$ for any pair of vertices $v, w \in K$). The first step is to construct the barycentric subdivision, sd$K$, and extend f linearly over sd$K$. This yields a new function f_0 with the property that no two critical points are adjacent, but which may not be a general function. A small perturbation described in [6] transforms f_0 into a general function f_1 over $K_1 = \mathrm{sd}K$.

Now the star of each non-simple singularity is further refined. Let v be a non-simple saddle point. For each k-dimensional simplex in the link of v, Lk(v), a new so-called k-vertex is added in the star of v, St(v), as follows. For each vertex w in Lk(v), a corresponding 0-vertex is added on the edge vw, at a point which is $\frac{1}{4}$ distance from v to w. For each k-simplex σ in Lk(v), $k \geq 1$, a k-vertex is added in the $(k+1)$-simplex formed by v and σ, at $\frac{1}{3}$ distance from v to the barycenter of σ. See Figure 5 for an illustration in 2 dimensions.

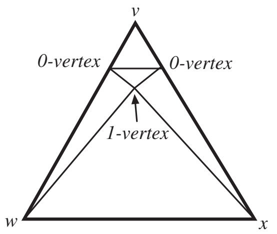

Figure 5: The subdivision of a 2-simplex vwx at a non-simple singularity v.

Simplices of this subdivision are defined as follows. Let σ be a m-simplex in St(v), i.e., a simplex of highest dimension; it contains m 0-vertices. These together with v form a new m-simplex. The rest of σ is then a prism with two $(m-1)$-simplices as bases. Now each cell containing a 1-vertex is star triangulated from the 1-vertex, then each 2-vertex defines a star triangulation to form tetrahedra, and so on up to the $(m-1)$-vertex, where the star triangulation results in m-simplices.

The neighborhoods of all non-simple singularities are refined in this manner, yielding a new complex K_2. Now f_1 is extended over K_2 to yield a new function f_2. By definition, $f_1 = f_2$ at all vertices common to K_1 and K_2. We now describe the extension of f_1 to f_2, again very similar to that described in [6].

Let h be a linear function over $\mathbb{R}^d$ that has different values at all vertices of K_2, and let H be the maximum difference between any two values of h on K_2, i.e., $H = \max_{v,w}\{h(v) - h(w)\}$. Let δ be the minimum gap between successive values of f_1 on K_1. For each vertex u added in the star of a non-simple singularity v, let

$$f_2(u) = f_1(v) + \frac{\delta}{2H}(h(u) - h(v)).$$

Function f_2 on K_2 now has the property that all singularities are simple, i.e., that the level set at $f_2(v)$ divides $\mathrm{St}(v)$ into at most three components. Indeed, it is easy to see that all former regular points and simple singularities are still regular or simple (see [6]), so we will restrict ourselves here to proving that a former non-simple singularity v is regular, and that all points added in K_2 are either regular or simple. To see that v is a regular point, notice that after the local refinement around v, $\mathrm{St}(v)$ consists only of the simplices formed by 0-vertices and v. f_2 is by construction linear over $\mathrm{St}(v)$ and so v must be a regular point. Now we use an inductive proof to show that the added k-vertices are either regular or simple. We define the *restricted star* or *restricted link* to be the restriction of the star or link of an added point u to simplices formed only by vertices added in K_2.

Lemma 5.1 *All k-vertices, $k \geq 0$, added in the subdivision around non-simple singularities are either regular points or simple singularities of f_2.*

Proof: Let u be a 0-vertex. u is adjacent to two original vertices from K_1: the non-simple singularity v, and the vertex w which was used to construct u. Otherwise, u is only adjacent to other added vertices. Since f_2 is linear over the simplices formed by v and the added vertices, the level set at $f_2(u)$ divides the restricted $\mathrm{St}(u)$ into at most two connected components, one with values greater than $f_2(u)$ and the other with values less than $f_2(u)$. w either belongs to one of those connected components or it forms its own connected component. Thus, u is either a regular point or a simple singularity.

Now let u be a k-vertex, $k \geq 1$. By construction, u is not adjacent to any vertices of K_1 other than the vertices of the k-simplex which define u. Again, the restricted $\mathrm{St}(u)$ and $\mathrm{Lk}(u)$ can be broken by the level set at $f_2(u)$ into at most 2 components. We now make the inductive assumption that a $(k-1)$-simplex $\sigma \in \mathrm{Lk}(u)$ from K_1 divides $\mathrm{Lk}(u)$ further into at most three components and show that under this assumption, a k-simplex from K_1 in $\mathrm{Lk}(u)$ cannot divide $\mathrm{Lk}(u)$ further into more than three connected components. Let $\sigma \in \mathrm{Lk}(u)$ be a $(k-1)$-simplex from K_1, and let $w \in \mathrm{Lk}(u)$ be the additional vertex from K_1 that forms a k-simplex in $\mathrm{Lk}(u)$. There are three cases to consider.

1. Suppose first that some vertices of σ have value in f_2 greater than $f_2(u)$ and others have value less than $f_2(u)$. Then w necessarily belongs to one of the existing connected components.
2. Suppose σ belongs to one of the connected components of the restricted $\mathrm{Lk}(u)$. Then $\mathrm{Lk}(u)$ without w consists of at most two components, and w can increase this to at most three components.
3. Finally, assume that σ forms a separate connected component. w is adjacent to both σ and vertices of the restricted

Lk(u), so regardless of the value at $f_2(w)$, w will belong to an existing component.

These three cases complete the proof. ■

Note that in the proof we do not need to distinguish between boundary simplices and interior simplices.

Acknowledgments

This work has been supported by NSERC through a postgraduate fellowship and a research grant.

References

[1] T. F. Banchoff. Critical points and curvature for embedded polyhedra. *J. Diff. Geom.*, 1:245–256, 1967.

[2] J. W. Milnor. *Morse Theory.* Princeton University Press, Princeton, NJ, 1963.

[3] C. P. Rourke and B. J. Sanderson. *Introduction to Piecewise-Linear Topology.* Springer-Verlag, 1972.

[4] Y. Shinagawa, T. L. Kunii, and Y. L. Kergosien. Surface coding based on Morse theory. *IEEE Comput. Graph. Appl.*, 11:66–78, Sept. 1991.

[5] S. Tarasov and M. Vyalyi. Some pl functions on surfaces are not height functions. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 113–118, 1997.

[6] S. P. Tarasov and M. N. Vyalyi. Construction of contour trees in 3D in $O(n \log n)$ steps. In *Proc. 14th Annu. ACM Sympos. on Comput. Geom.*, pages 68–75, 1998.

[7] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22:215–225, 1975.

[8] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. Contour trees and small seed sets for isosurface traversal. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 212–220, 1997.