SVR: Practical Engineering of a Fast 3D Meshing Algorithm [⋆]

Umut A. Acar[1], Benoît Hudson[2], Gary L. Miller[2], and Todd Phillips[2]

[1] Toyota Technological Institute
[2] Carnegie Mellon University

Summary. The recent Sparse Voronoi Refinement (SVR) Algorithm for mesh generation has the fastest theoretical bounds for runtime and memory usage. We present a robust practical software implementation of the SVR for meshing a piecewise linear complex in 3 dimensions. Our software is competitive in runtime with state of the art freely available packages on generic inputs, and on pathological worse cases inputs, we show SVR indeed leverages its theoretical guarantees to produce vastly superior runtime and memory usage. The theoretical algorithm description of SVR leaves open several data structure design options, especially with regard to point location strategies. We show that proper strategic choices can greatly effect constant factors involved in runtime.

1 Introduction

At last year's IMR conference we introduced a new meshing algorithm, Sparse Voronoi Refinement (SVR), which provided the typical guarantees for theoretical meshing algorithms, along with an unusual one that the algorithm ran in near-linear time [HMP06]. The goal in designing SVR was to create a meshing algorithm that was similar in implementation and style to many widely used meshing algorithms, but with the added benefit of very strong worst-case bounds on the runtime complexity and space usage. An additional achievement of SVR is that the algorithm can work in any fixed dimension d. More recently, we proved that the algorithm can be run in parallel, and we showed that the the Li-Teng sliver removal algorithm could easily be incorporated into the SVR framework [LT01, HMP07].

The main goals of the present work are twofold. First, to show preliminary results on a new implementation of the sequential version of this algorithm. Second, to discuss new data structures to empirically improve the run time of the point location parts of the algorithm, which may be of more generally applicable use. We focus on point location because both in theory and practice, the dominant cost of SVR and most other algorithms for meshing is the point location cost.

[⋆] This work was supported in part by the National Science Foundation under grants CCR-0122581 and Intel faculty gift.

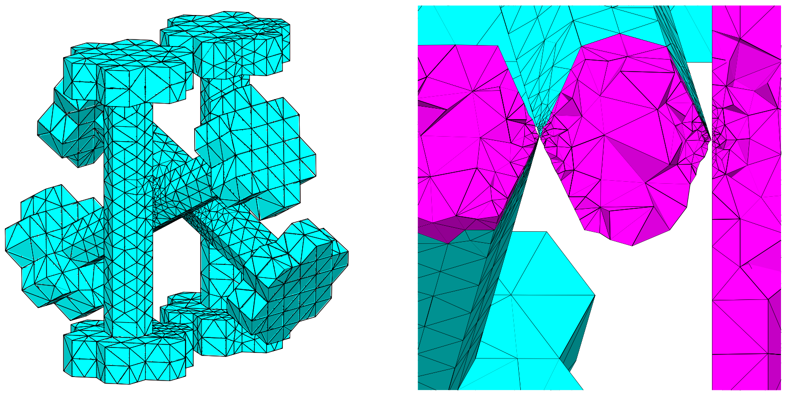

Fig. 1. Left: A radius/edge quality 2.0 mesh of an assemblage of four hexagonal dumbbells meshed by our software. Each dumbbell is described as a set of facets defining the two ends and the connecting rod. The output is a set of tetrahedra that fill space and the barbells while resolving the input. For visual effect we removed those tetrahedra in a standard postprocessing step. **Right:** detail on the nearest approach of three of the dumbbells. Notice that the tetrahedra grade smoothly away from the pinch point.

We compare SVR with two related codes: Pyramid by Shewchuk [She05a, She98], which is available by request to him; and TetGen by Si [Si07, Si06], which is available online. Our implementation of SVR compares very favorably to them, generating meshes that are of similar quality and size in less time. Furthermore, on pathological examples, prior codes run out of memory even at small input sizes whereas SVR sees no difficulty. Our implementation is available at `http://www.sparse-meshing.com`, free for the research community.

SVR produces a quality conforming mesh that is size-optimal in the number of vertices [HMP06]. One important concern in quality meshing is how we define the quality of tetrahedron. We use two separate measures of element quality in the algorithm and code. Both rely on the *circumball* of a tetrahedron, the smallest ball containing the tetrahedron's vertices. We denote it's radius (the *circumradius*) as R. The first quality measure we use is the radius-edge ratio of a tetrahedron: we we compare the circumradius R versus the shortest edge e of the tet. In a good-radius-edge tetrahedron, R/e must be less than some value ρ. In three dimensions, this metric is somewhat lacking, as it can admit poorly-shaped *slivers*. Radius-edge is still useful however, since in a good radius-edge mesh, every vertex has bounded degree [MTTW99].

The other quality measure is the radius/radius ratio: we compare the circumradius R to the radius r of the largest ball *inscribed* by the simplex. The ratio R/r of a good quality element must be less than some value σ. This quality criterion does not admit slivers, and is thus the one desired for output.

In the finite element method, an element with good radius/radius ratio is known to be numerically good under standard assumptions. The reason we use both measures is that is it not well understood how to avoid creating slivers during the algorithm (though we do scour them from the final output). The SVR algorithm provably never creates excessively bad radius-edge simplices even in the intermediate stages of the algorithm.

Time and Space Usage: Our code takes as input a Piecewise Linear Complex (PLC) [MTTW99]. Let n be the total number of input features (vertices, segments, polygons, etc). Let L/s be the *spread* of the input, i.e. the ratio of the diameter of the input space to the smallest pairwise distance between two disjoint features of the PLC.

SVR has worst case runtime bounded by $O(n \log L/s + m)$, where m is the number of output vertices. This runtime bound is a vast improvement over prior meshing algorithms for three and higher dimensions. For almost all interesting inputs, this bound is equivalent to $O(n \log n + m)$, which is optimal (using a sorting lower bound). SVR also has optimal output-sensitive memory usage $O(m)$, which means that even on pathological inputs it can process moderately large inputs entirely in memory.

2 Related Work

There have been several different approaches to the meshing problem. The idea of generating a mesh whose size is within a constant factor of optimal was first considered by Bern, Epstein, and Gilbert [BEG94] using a quadtree approach. A 3D extension was given by Mitchell and Vavasis [MV00], who later released an implementation under the name of QMG [Vav00].

Chew introduced a 2D Delaunay refinement algorithm [Che89] and showed termination. The quality of the initial triangulation was improved by adding the circumcenters of poor quality triangles as extra vertices. This produced a mesh with no small angles, but inserted many more new vertices than necessary. Ruppert [Rup95] extended this idea of adding circumcenters for 2D meshing to produce a mesh that was within a constant factor in size from the optimal and also handled line segments as input features. Shewchuk implemented Ruppert's algorithm in the very popular Triangle code [She05b], which has since been extended with various enhancements and remains actively maintained.

The extension of Ruppert's algorithm to 3D has been ongoing research. Some methods assume that that Ruppert's local feature size function is given [MTTW99]. Others refine a bad radius-edge ratio mesh directly [She98, MPW02]. These methods by themselves do not give quality meshes because they include slivers; a large number of other techniques have been concocted that aim to eliminate slivers while only slightly (at most linearly) increasing the output size. A 3D version of Ruppert's algorithm in conjunction with a sliver-eliminating post-process produces a quality, optimal-sized mesh. Shewchuk has implemented his higher-dimensional version of Ruppert's algorithm in Pyramid [She05a], but it has not yet achieved an official release and remains in alpha stage.

Of the many other algorithms for Delaunay refinement in 3D that have been proposed [She98, MPW02, CP03, CD03], none except SVR have eluded nontrivial runtime analysis. Trivial runtime bounds such as $O(m^3)$ be found in most cases. Simple examples can usually give bad worst-case performance for naive implementations of

```
S      -SVR
 1: while Volume Mesh is not Conforming do
 2:   R     a non-conforming tet
 3:   while Volume Mesh is not of Quality $\rho$ do
 4:     R     a poor-quality tet
 5:   end while
 6: end while

R
 7: if F   W   P      then
 8:   Insert a vertex from some nearby feature
 9: else if F   E         then
10:   Recursively R     all encroached lower-dimensional feature meshes
11: else
12:   Add a circumcenter as a new vertex in this Mesh
13: end if
```

Fig. 2. A very loose description of a much simplified SVR. Most of the runtime work is spent on point location in evaluating lines 7 and 9 (See Section 8). More detailed pseudo-code and the full algorithm can be found in [HMP06].

these algorithms. As mentioned, they will all suffer from intermediate size $\Omega(n^2)$ in the worst case.

2.1 Optimal Time Meshing Algorithms

Finding refinement algorithms that have provably good run times has also been of interest. Spielman, Teng, and Üngör [STÜ07] proved that Ruppert's and Shewchuk's algorithms can be made to run in $O(\lg^2 L/s)$ parallel steps. They did not, however, prove a work bound. Miller [Mil04] provided the first sub-quadratic time bound in 2D with a sequential work bound of $O((n \lg \Gamma + m) \lg m)$, where Γ is a localized version of L/s (in particular, $\Gamma \leq L/s$). In addition to an $O(n \lg L/s + m)$ sequential runtime, the SVR algorithm has been shown to be parallelizable to run in $O(\lg L/s)$ iterations with the same work [HMP07].

3 Overview of SVR

For a detailed description of SVR, refer to [HMP06]. Herein, we briefly review SVR for the purpose of describing the implementation needs of the algorithm. Coarse pseudocode for SVR is described Figure 2. The input is given as a piecewise linear complex (polygonal faces, segments, and vertices), that we will refer to as a set of *features*. Constraints on the input are described further in Section 4. Over the life of the algorithm, SVR maintains a separate mesh for each feature. A three dimensional volume mesh is maintained covering the entire domain.

In SVR, these feature meshes begin very coarsely, and initially do not conform to the input at all. The meshes are then gradually refined for two reasons: to maintain quality, and to eventually conform. Refinement is done by attempting to insert the circumcenter of a poor quality tetrahedron or of a non-conforming tetrahedron. But because the mesh does not yet conform, we may instead choose to W   , and insert a vertex from a nearby input feature instead of the circumcenter. We do this to make sure that new points are not created to close to the features or close to input points. Additionally, before the volume mesh can insert near a feature, we may first need to refine the feature itself. This is accomplished by protecting the feature with *protective balls* that cannot be entered by the volume mesh. If the volume mesh desires to insert a circumcenter that *encroaches* upon a lower-dimensional protected ball, it first *yields* to the lower-dimensional feature, and allows the latter to refine itself. After the lower-dimensional feature has refined, the volume mesh may warp to the point that was inserted into the lower-dimensional mesh. Facet meshes operate similarly with respect to their bounding segments: if a facet mesh wishes to eliminate a bad triangle (because it has poor quality, or because the volume mesh encroached upon it), it attempts to insert the circumcenter of the triangle, possibly yielding to encroached boundary segments and/or warping to points not yet resolved in the facet mesh. A putative new circumcenter may encroach on many protective balls simultaneously; if so, for good runtime we must make sure all the affected features refine themselves.

4 Input Format

The input format is that of Shewchuk's Pyramid, which is the obvious extension of the Triangle format to the third dimension: An input file starts with a header listing how many inputs vertices there are, then lists their coordinates. If there are segments to conform to, it then lists their number and describes each as a pair of vertex IDs. Finally, if there are polygons, it lists their number and describes each as a list of segment IDs. Holes and concavities are automatically found and need not be mentioned in the input.

For SVR to properly produce a quality mesh, users must somewhat restrict their input. As usual, our mesher needs clean inputs: the code will report an error if a polygon is not watertight, or if two polygons intersect each other geometrically but no intersection is mentioned in the input. More importantly, the algorithm sometimes also fails if two input polygons intersect at an angle less than 90°, even on clean input: the problem is that refinement on one of the faces may encroach on the other, which may loop back to encroach on the former. We can tell the mesher not to refine an element smaller than some minimum size, which will return us a mesh, albeit with possibly some bad elements. It remains active research to solve these issues in a more principled but practical manner.

Our runtime proofs also require additional properties of the polygonal features: (1) polygons must be defined by only a constant-bounded number of points, (2) the initial triangulation of each polygon must be of good quality, (3) the initial triangulation must not self-encroach. Violating any of these requirements impinges upon the runtime properties, but the code will still properly return an optimal-size, good-quality mesh. Polygons with n_i vertices on their boundary currently take time $O(n_i^2)$ to preprocess; it would not be hard to reduce this to $O(n_i \lg n_i)$ time, though it seems unnecessary for all inputs we have on hand since generally n_i is on the order of 4–20. The second and third condition are repaired automatically by adding more points to the boundary and interior when creating the initial triangulation; this causes the code to run only logarithmically slower than optimal.

Constant	Constraints	Default Value
Output Radius-Edge, ρ	$\rho > 2$	2.0
Output Radius-Radius, σ	$\sigma >> 3$	not set
Sliver Growth, B	$B > 2$	3.0
Perturbed Insertion, δ	$0 < \delta < 1$	0.1
Yielding Ratio, k	$0 < k < 1$	0.9

Fig. 3. Table of constants for SVR.

5 Algorithm Constants

Several constants for SVR are left to be chosen by the user. Figure 3 gives an overview of these constants and their defaults in our implementation.

The first constant, ρ, determines the radius-edge quality bound on every element in the final output mesh. For most numerical methods using the mesh, ρ will have a strong effect on the quality and runtime of the method used, with a lower ρ (better shaped elements) being preferred. Of course, driving ρ lower will necessarily increase the number of elements output by any meshing software, SVR included.

Many numerical methods also require the elimination of slivers [ELM+00]. We have extended SVR to eliminate extreme slivers using the Li Teng sliver removal algorithm [LT01]. The algorithm randomly perturbs circumcenter insertions by a factor of δ to ensure that any new slivers created must be larger by a factor of B, then recursively works on the larger slivers. Eventually no larger slivers can be created, and so the mesh is sliver-free. The constant σ gives an upper bound on the radius/radius ratio of any output tetrahedron. Like all quality bounds, tightening σ (reducing it) will increase output size. The published proofs [LT01, Li03] suggest the settings of δ and B listed above; unfortunately, for σ they yield a gargantuan number. Therefore, by default the sliver-removal code is off. However, the proof is known to be very loose, and in practice we can eliminate much worse slivers than is provable. See the experiments in Section 9.

5.1 Warping parameter

One of the more interesting constants to specify for SVR is the *warping parameter*, k. This parameter controls the behavior of the warp operation: when we decide to split a simplex s, we will warp to a point within distance $k \cdot R(s)$ of the center of s if there is one. A higher value of k will more aggressively look for a vertex to warp to, which intuitively would help reduce the output mesh size. However, a higher value of k also worsens the intermediate quality bound, which allows high-degree vertices and thus hurts runtime. This section discusses the tradeoff between k, runtime, and output size, and aims to find a reasonable default setting for k.

First, in our original paper, we proved that if $\rho k^3 > 2$, our algorithm will terminate with the aforementioned guarantees. This somewhat limits the user's choice of parameter settings. However, it is easy to show that we can internally use a ρ' that satisfies the requirement, and, once all points have been inserted into the mesh (and therefore k is rendered irrelevant), we can improve the mesh quality up to ρ. In other words, the user may ask for any k, and any $\rho > 2$.

Under this framework, we can then ask about the optimal setting of k in terms of mesh output size, or in terms of runtime, leaving ρ fixed. We ran an experiment on the point-cloud refinement code, tracking runtime and output size versus k. See Figure 4. The main finding is that in practice, the output size is indeed quite strongly affected by the value of k, whereas runtime is much less affected until k becomes very close to 1. Based on these results, we set the default k value to 0.9 as being a reasonable tradeoff between output size and runtime.

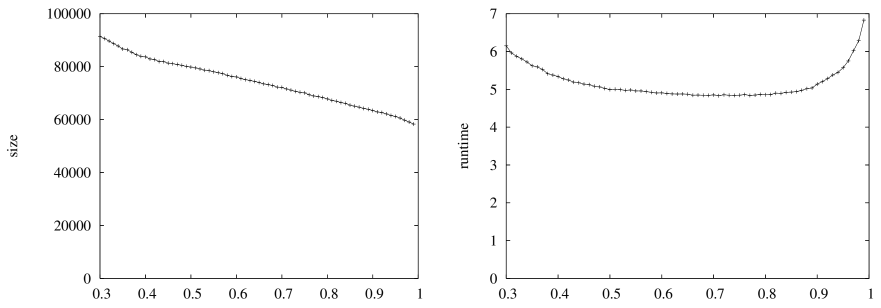

Fig. 4. *Left:* Number of vertices SVR outputs for the Stanford bunny, versus k. *Right:* Runtime of SVR on the Stanford bunny in arbitrary units, versus k. Notice the tradeoff between mesh size and runtime. Other natural and pathological examples showed similar tradeoffs.

Another experiment we ran was to see how to best choose the point to warp to. When there are many choices, it may be that one is better than another. In particular, we expected that it would be best to choose the point closest to the prospective Delaunay center among all available points to warp to; the intuition is that this is, in a sense, the “median” and we are, in a sense, sorting. Whether this is a good idea depends on the order in which valid warp points will be seen. In early implementations, the difference was substantial. However, in the current implementation, with the Voronoi shelling approach described in Section 8.4, the first point we find is far from any vertex, making it a good choice. Therefore, we currently report the first point we can safely warp to.

6 Implementation in C++

We implemented our algorithm in C++, in the hopes of being most accessible to the community, but still highly efficient. Almost all of the code is general-dimensional; the exception is in the numerical predicates, where we use Shewchuk’s specific-dimensional code [She97]. While the code is written in a general dimensional style, we use the C++ template mechanism to choose the dimension at compile-time. That is, an `SVR<2>` is a mesher in two dimensions, while an `SVR<3>` is in three dimensions. This allows the compiler to unroll loops, perform constant propagation on the dimension term, and otherwise optimize the code to achieve almost no overhead over a specific-dimensional code. The data structures are accessible via an API; we expect this will allow more easily experimenting with post-processes and variations on our algorithm.

7 Mesh Data Structures

Using any reasonable data structures, SVR will be faster than its competitors on pathological examples. However, to also be useful on more normal inputs, we must be somewhat careful. We implemented SVR in a way that sharply separated the implementation into four modules, so that we can easily change various pieces one by one: (1) basic data structures and I/O, (2) the topological mesh structure; (3) the geometry code and Delaunay structures; (4) the mesh refinement algorithm.

The basic data structures are essentially C++ Standard Templates Library (STL) structures optimized for our usage. The STL and the Boost libraries are extremely useful for quickly generating prototype code, but we have seen that many of their structures are either too large or too slow for production usage in many cases. When profiling indicated that STL libraries were limiting factors, we replaced them with some of our own versions, in particular to have greater control over memory allocation.

For the topological structure, we currently use a standard pointer-based simplicial complex (4 vertex pointers and 4 neighbour pointers per tetrahedron). Client code can optionally attach additional data using a template argument; the type of vertices is itself also specified by a template argument. Simplices are reference-counted to avoid complications and memory bugs when simplices are destroyed yet remain in the SVR work stack. Access is typically via a generic depth-first-search routine.

The API has changed little as we tried various types of structures; we therefore predict that, with relatively little effort, we could scale our implementation to much larger sizes than tested here simply by replacing the underlying structure with a compressed mesh structure [Bla05]. The topological structure is also largely compatible with the CGAL structures; one could write a thin adapter class to view one structure as the other. We chose to write our own structure in order to have full control over the implementation.

The Delaunay triangulation structure is a thin veneer over the simplicial complex. SVR computes the circumcenter and circumradius of every simplex (to test for quality), and does so repeatedly: simple empirical evidence shows about 10 times per simplex in 3d. Therefore, we compute these values only once, and cache them as data attached to each simplex. An LRU cache can easily be substituted for relatively little runtime overhead to reduce the per-simplex memory usage; this will become critical if we use a compressed mesh structure.

The geometric code is largely drawn from Shewchuk's notes on geometric calculations [She99] and from his public-domain library [She97, NBH01]. We use LAPACK for some calculations, but have found that for very low-dimensional operations, which dominate our geometric requirements, the cost of marshaling and unmarshaling data between our code and LAPACK dwarfs the cost of the numerical calculations, and it is thus advisable to write them by hand.

The point location structures of the mesher are the most performance-critical ones; we describe them separately in great detail in Section 8.

8 Point Location Data Structure

A point location data for SVR provides a good deal of room for variation in implementation. SVR requires that every time a vertex is considered for addition into the volume mesh, SVR locally searches for any lower dimensional feature meshes, so that the volume mesh might conform to these instead. Additionally, SVR searches for lower dimensional features that may need refinement before it can proceed with this volume mesh insertion. Essentially, a correspondence must be maintained between the volume mesh and the feature meshes because they do not conform to one another. In the following section, we describe implementing such a correspondence.

Recall that SVR runs in $O(n \lg L/s + m)$ time. The $n \lg L/s$ term is driven by the cost of two operations: finding a point to warp to, and testing for encroachment. We conflate these operations and call them both *point location*. The m term also includes these operations, plus some other work. Therefore, point location is the leading term in the asymptotics on most inputs, and even when it is not, it is a major constant factor. Profiling information on various inputs verifies that this prediction holds in practice in the current implementation: well over 20% of the runtime – and 33% of the cache misses – are directly due to point location, even ignoring any ancillary costs (malloc/free; memory overhead; cache evictions causing slowdowns elsewhere; etc).

The PLC input is described using a standard *incidence poset*: segments know their endpoints and any internal vertices, polygons are described using a set of boundary segments (and possible internal segments or vertices), etc. For simplicity, we compute the transitive closure of the poset, linking facets to vertices, etc. The structure of this linking between meshes is shown in Figure 5.

To quickly handle point location queries, we need to keep track of intersections between elements of one mesh and elements of the mesh of a lower-dimensional feature (and vice-versa). The published SVR algorithm used the Voronoi cells of the mesh vertices as elements, to make the proofs most succinct. Here we report on the differences between several choices, all asymptotically equivalent but with significant constant-factor differences.

We formalize this notion: we maintain a bipartite map between the abstract types of an `Upper` container and a `Lower` cell. See Figure 6. At the moment, we only have two `Lower` types: circumspheres (for features of dimension 1 and higher) and points (for features of dimension 0). Points are of course merely special cases of sphere with radius 0, but they are sufficiently special to merit distinct consideration. This formalism involving `Upper` and `Lower` is likely to extend to more interesting inputs such as curves.

The point location data structure will need to perform the following three operations quickly:

F W P : Given a Delaunay simplex s in a mesh M, determine whether there exists an point to be inserted in M (a child of M in the poset) that lies within the warp ball $B(c(s), k \cdot R(s))$.

F E : Given a point p chosen by F W P for s, determine the (possibly-empty) set of lower-dimensional protective balls b_i that p encroaches – that is, p lies in b_i.

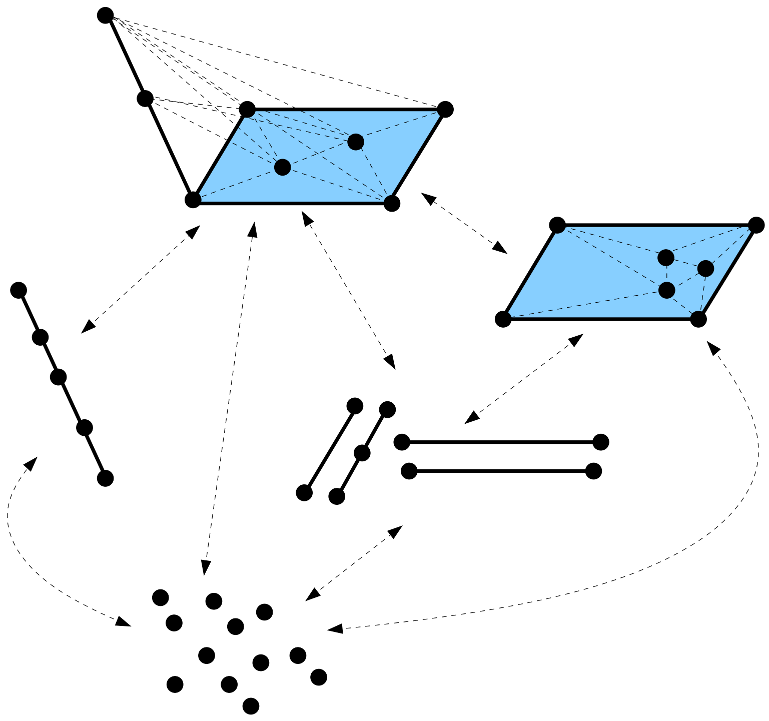

Fig. 5. The volume mesh at the top of the figure has features consisting of one facet, five segments, and several input points. The point location structure must link all of the meshes along the arrows shown. Note that these meshes may not all conform to each other; input points or feature refinements may not be resolved in the volume mesh. Linking two meshes involves tracking the intersections between every lower dimensional element and every higher dimensional element.

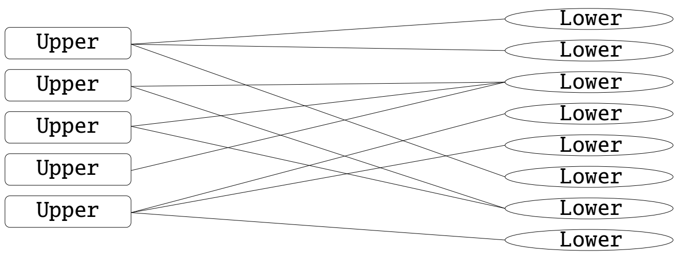

Fig. 6. We maintain a map, tracking intersections between `Upper` containers and `Lower` cells.

U : Given a point *p* chosen by F W P in a mesh *M*, and given the set of simplices before (C, the *cavity*) and after (S, the *star*) inserting *p*, update the point-location structures.

We compare four different choices of `Upper` in the following subsections. We have experimented with all four for maintaining the point location structure for points. For maintaining protected balls, we so far have only used the first method; trying other techniques remains future but high-priority work.

8.1 Circumballs as `Upper`

The easiest way to implement F W P and F E is to use the circumball $B(c(s), R(s))$ of each simplex s as the `Upper` elements. Indeed, any point that we may warp to is in $B(c(s), k \cdot R(s))$, a strict subset of the circumball. Similarly, any lower-dimensional feature encroached by a point chosen by F W P necessarily intersects the circumball.

U is only slightly more complicated: to compute the `Lower` s intersecting a new simplex s, we must find every lower-dimensional protected ball that intersects the circumball of s. It is a standard fact that the set of circumballs of the cavity C (the simplices before inserting p), plus the neighbours C_N of the cavity, covers the set of new circumballs. Thus, we can compute the `Lower` s intersecting $C \cup C_N$ and for each new simplex, filter this set to compute those intersecting the circumball of s.

8.2 Tetrahedra as `Upper`

When maintaining the uninserted points of a mesh, the approach in the prior section stores a point repeatedly since the circumballs of the mesh intersect. If instead we use the simplices directly, we avoid duplicates; this is the basis for in-simplex point location being the most traditional kind.

The lack of overlap between regions greatly speeds up the cost on contained points: we can perform about half as many intersection queries on average since we can shortcut execution at the first simplex of the star that matches the vertex. Furthermore, the star and cavity cover the same area, so we need only relocate the `Lower` s intersecting C, ignoring the neighbours. Finally, computing the set of `Lower` vertices interesting C is easier since we need not eliminate duplicates.

The cost of this approach is that now in F W P (s) we must find all the simplices that intersect the circumball of s. There are only a constant number of these, but finding them all is still expensive. Nevertheless, note that every simplex ever created is involved in an call, whereas only a small minority (empirically about one in 18, in three dimensions) ever have F W P called on them.

For maintaining lower-dimensional features, we do not gain the benefit of not having duplicates, which makes this technique unlikely to be useful. For uninserted points, however, this point location structure is substantially faster than in-sphere in our experiments.

8.3 Voronoi Cells as `Upper`

Traditionally, the Delaunay triangulation is the most common structure to use for point location, which made it the natural choice for the implementation. However, determining whether a point p is owned by a simplex s is expensive: either an in-sphere or an in-simplex test, both of which are essentially determinants of matrices of rank d, which becomes a major cost in U . Another choice is to use the dual

Voronoi diagram, as described in the original paper. A point p is in the Voronoi cell of a mesh vertex v if v is the nearest neighbour to p. If we know the old Voronoi cell in which p lies (call it $V(v)$), then updating after the insertion of one mesh vertex v' requires only two distance calculations: the distance $|p - v|$ and the distance $|p - v'|$. Furthermore, there are many fewer vertices than simplices, by a ratio of, empirically, 1:6 in three dimensions (both in the experiments we ran, and in prior reports [Bla05, *e.g.*]). Thus, the overhead of the lists is greatly reduced compared to using the simplices as the basic objects, which becomes important near the end of the algorithm.

Searching the `Upper` Voronoi cells for F W P has a disadvantage: we must search every Voronoi cell that intersects the warp ball. We can do this by searching the set associated with every vertex on every simplex whose circumball intersects the warp ball. In terms of implementation, this is only a slight increase in code compared to using the interior of simplices; but this is a much larger area being searched, so we may visit many more vertices that are not in the warp ball. In practice, we found this technique to be equal in runtime to using simplices for point location, although the memory usage is slightly reduced.

Beware of one pitfall when using Voronoi cells: Unlike simplices, vertices have long lifetime. Therefore, if each vertex has its own memory pool for the list of uninserted points in its Voronoi cell – or uses the STL :: class, which never shrinks –, vertices inserted early in the run of the algorithm will reserve a large amount of memory even when they no longer need to track many vertices. This is easily avoidable by actually releasing memory when it is no longer needed, or by using a single memory pool shared among all vertices.

8.4 Shelled Voronoi Cells as `Upper`

When looking for a point to warp to, we take a Delaunay ball and shrink it by a factor k. Therefore, any input point that is "near" a vertex need not be examined at all. However, the previously-described approaches take no note of this. An easy way to implement this is to store uninserted points in the Voronoi cells, as described above, but within each Voronoi cell, bucket the points according to distance from the Voronoi site – that is, into concentric shells of geometrically increasing radius (see Figure 7). Upon a F W P query, we ignore any bucket that lies entirely outside the query region; similarly, on U we do not try to reassign points in buckets that are closer to the old vertex than to the new. Asymptotically speaking, this converts $O(\lg L/s)$ distance calculations on an uninserted point chosen very late, into $O(1)$ distance calculations and $O(\lg L/s)$ divide-by-two operations. Indeed, in practice we saw the total runtime of the algorithm fall by half when we implemented shelling of Voronoi cells as compared to either Voronoi cells or in-simplex point location.

8.5 Further Refinements

As noted earlier, using the Voronoi diagram has the disadvantage that uninserted points rather far from the query ball can be accessed. Shelling does not reduce this tendency: shelling only means that uninserted points very close to a vertex will be

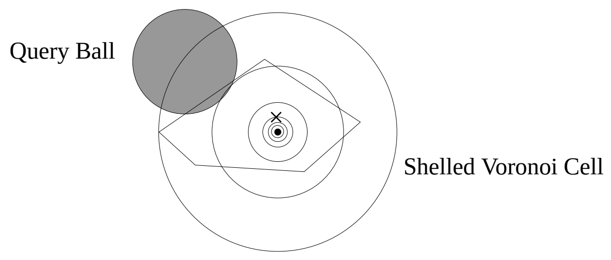

Fig. 7. Illustration of shelling and of a query on a shelled Voronoi cell. The shells are concentric around the Voronoi site, with radius halving at each step toward the center. While all the shells exist mathematically, in the implementation we only store those that include at least one uninserted point. During F W P , the uninserted point marked by an X is not visited because the annulus that contains X does not intersect the query ball.

ignored. Another view of the problem is that we now understand well how to store points near the mesh vertices, but the best strategy for storing point far from vertices is not yet clear. One possibility is a hybrid, using shelling for points close to mesh vertices, and in-sphere or in-simplex for points far from the vertices. Another possibility is to locate in the Voronoi cells of both the mesh vertices and the element circumcenters.

We have identified three times when we can completely avoid calling F W - P . Clearly, if there are no uninserted vertices remaining anywhere, we can simply skip the call. This occurs in about 10% of the insertions on the bunny dataset, 30% on the pathological input, and merely requires keeping a global or per-mesh counter. If we are retrying an insertion that caused a small sliver, we know we will not warp since the previous iteration would have warped if needed. How often this occurs depends on the sliver parameters. Finally, when we insert a point due to a crowded Steiner point, we can blindly insert any uninserted vertex in the Steiner point's Voronoi cell. This happens 5% percent of the time on the bunny dataset, almost never on the pathological input. Together, then, we see that about 15-30% of split operations can avoid this call.

9 Experiments

We performed some experiments on our implementation to determine the runtime and output size of our algorithm as compared to a few other equally-available codes. The experiments are on point-cloud inputs: as of this writing, the point-cloud code of both SVR and Pyramid is more mature than the feature-set code. We ran the experiments on a desktop 3.2 GHz Pentium D with 2 GB RAM running Linux 2.6. We compiled all applications using the gcc compiler version 4.2.1 with compiler flags `-m32 -O2 -g -fomit-frame-pointer -mtune=native -DNDEBUG` for both C and C++, except for one file in TetGen which cannot be optimized and had to be compiled `-m32 -O0`. All three codes currently assume 32-bit pointers, though this should be easy to fix in all cases. We compare our implementation (SVR) to Pyramid 0.50 [She05a] and to TetGen 1.4.2 [Si07]. For all meshes here, we required an output radius/edge quality of 2.0; unless otherwise noted, SVR did not perform sliver removal in these experiments. SVR used a value of 0.9 for its k parameter (see Section 5); Pyramid and TetGen use default values. We measure time by using the UNIX 'time' utility and summed "user" (cpu) and "system" times. All reported times are averaged over five runs.

9.1 Point cloud results

Input	SVR	Pyramid	TetGen	SVR	Pyramid	TetGen
Stanford Bunny ($n = 34890$)	4.62	6.35	12.4	59702	59040	74269
Line & Circle ($n = 2000$)	0.80	4.79	6.5	12119	14003	14573
Line & Circle ($n = 20000$)	7.62	N/A	N/A	120933	N/A	N/A
$50 \times 50 \times 50$ Grid ($n = 125000$)	11.30	15.96	45.9	129839	129929	130140
$100 \times 100 \times 100$ Grid ($n = 10^6$)	97.71	179.04	400.3	1016262	1017799	1018684

Table 1. Comparison of the SVR, Pyramid, and TetGen codes on a few point-cloud inputs. Both Pyramid and TetGen ran out of memory on the $n = 20000$ Line & Circle example, and could not complete; otherwise, all examples fit in memory. **Left**: Execution times (seconds of CPU plus system time) versus inputs. Average of 5 runs. **Right**: Output size, in vertices. All three methods produce meshes of approximately the same size.

Table 1 shows a comparisons of timings and output sizes for TetGen, Pyramid, and SVR on three inputs: (1) the Stanford Bunny, (2) line-and-circle (a pathological quadratic example), and (3) points on a grid. All these inputs are then bounded by a $6 \times 6 \times 6$ bounding box to avoid all boundary effects. SVR is the fastest on all inputs, increasingly so as complexity increases. Furthermore, as predicted by theory, both Pyramid and TetGen crash on very modestly-sized pathological inputs. Even with inputs of just 20,000 points, they can be made to try to allocate more memory than can be addressed with 32-bit pointers.

TetGen has a performance bug for meshing point clouds: it requires the user first invoke TetGen to produce and output a Delaunay mesh; then invoke TetGen again to load that mesh and refine it. On non-pathological input, this is a low-order effect. However, on pathological input the reloading of the mesh runs in time cubic in the number of vertices. This is a front-end issue and is not fundamental, so we subtract out the time of the intermediate output and reload in the TetGen times we report.

The number of vertices and tetrahedra output by all three codes is similar. During the development of SVR, we found that relatively minor changes in the code can often change the output size by 20% in either direction (often in opposite directions for different inputs), but not reliably so. As noted by Har-Peled and Üngör [HPU05], one of the few consistently useful heuristics is to work on the smallest simplex first, as measured by its shortest edge; in our algorithm, this happens to also improve cache efficiency since the smallest simplex is usually the most recently-create one.

Finally, we investigated the issue of slivers. On the bunny, radius/edge refinement using either SVR or Pyramid created several thousand simplices with dihedral angles flatter than 175°; in both, dihedral angles ranged from 0.01° to 179.98°. Tet-

Gen has a separate mesh optimization post-process which produced dihedral angles from 4.64° to 168.56°. In our implementation of SVR with Li-Teng sliver removal, we find we can achieve angles ranging only between 12.09° to 154.21° on the bunny when asking for radius/radius quality of 9.0 (recall from elementary geometry that an equilateral tetrahedron has radius/radius ratio 3.0) . Unfortunately, Li-Teng only provably works with very weak guarantees: at the best quality we demanded, the technique is very brittle and often fails depending on small changes in the parameter settings. This suggests two avenues for improvement: tweaking Li and Teng's algorithm to produce more practical bounds so that we need never go beyond the provable envelope, and/or the addition of a mesh optimization post-process to the SVR code.

9.2 PLC results

The vast majority models available in mesh repositories – even those that use the Pyramid input format we adopt – are triangulated surface meshes. Unfortunately, this is automatically an illegal input: the edges of a triangle obviously do not all meet at non-acute angles. TetGen merges adjacent triangles that are (nearly) coplanar; time did not permit us to implement this type of input grooming. Instead, we ran SVR on several PLC examples provided to us by Jonathan Shewchuk. None are strictly legal according to our input restrictions (see Section 4); in particular, they all have facets meeting at acute angles. Nevertheless, we were able to produce quality meshes of about the same size as Pyramid's output. We also synthesized our own examples, such as the ten-barbells example shown in the introduction, which satisfy all the theoretical requirements. On these examples, we were also able to remove slivers with dihedrals worse than about 5° or 175°, albeit not entirely reliably. The major outstanding issues in the feature-set SVR code come down to implementing some of the tricks that TetGen has to work with input that is, strictly speaking, illegal; and to speed up the provably correct code using many of the techniques we have discussed and used in the point-cloud code.

9.3 SVR profiling

We made heavy use of profiling (particularly using the Valgrind toolset) while optimizing the underlying data structures for point-cloud refinement. Similar optimization of the feature-set refinement code remains future work. On various examples, we find the following trends. First, a large fraction (about 20%) of the time is spent writing the output to an ASCII file. Applications for which time is critical will output in a faster and smaller format, so let us ignore this cost. On the platforms we tested on (Intel, AMD, and PowerPC), we find the processor issues about 0.7 to 1.0 instructions per clock cycle, which is in line with what we would expect from an unstructured code. The cache performance was good: essentially no I-cache misses, while 1.5% of data reads reach to L2 and only 0.3% to main memory. This can be explained by the fact that SVR is fundamentally parallel [HMP07]. Therefore, merely by using a work stack instead of a work queue, we achieve good data locality. Another illustration of this is that SVR could maintain about 4% CPU utilization in tests where the mesh did not fit in memory. By contrast, Pyramid under the same conditions suffers a 0.6% miss rate to main memory, and achieves less than 1% CPU utilization when it hits swap.

On the Stanford bunny example, the code issues about 5.5 billion instructions. Three major components each take almost exactly one billion instructions each. (1) Calls to Shewchuk's numerical predicates library, mostly in-sphere tests. (2) Computing circumcenters, and performing distance calculations for point location. (3) Topologically updating the Delaunay triangulation. The remainder of the time is taken up traversing the mesh to perform point location queries, maintaining the work queues, allocating memory and maintaining reference counts, and reading the input. We note that it is critical that we use memory pools to allocate small objects (list nodes, simplices, and so on).

Major improvements in point-cloud meshing time will require three improvements. Most prosaically, we need to make our code run on 64-bit architectures to take advantage of large memories; as the mesh size grows, we expect our lead over previous codes to widen. Next, using a compressed mesh structure [Bla05] would improve cache performance and allow us to mesh much larger examples entirely in memory. Finally, we cache circumcenters because we would otherwise repeatedly recompute the circumcenter; but the same effect is true of in-sphere tests. It is highly likely we could substantially reduce the cost of category (1) above by merging it with category (2). Major improvements in feature-set meshing time are low hanging fruit at the moment. After some more work to eliminate any remaining factors of two in sequential runtime, we expect eventually to look to parallelize the code, as we have theoretically proved can be done.

10 Conclusions

We have shown that it is possible to implement the theoretically described SVR algorithm to achieve practical robust software. The strong asymptotic runtime guarantees underlying the SVR algorithm are clearly evident on simple examples. Performance is competitive with existing codes on relatively well shaped inputs. For pathological inputs, the superiority of SVR is unquestionable.

In the SVR implementation, we have combined strong theory with a great deal of software engineering. Devising and implementing appropriate data structures was crucial to implementing practical software. As well, attention to geometric predicates is necessary for any robust code.

10.1 Extensions

The most obvious extension of the SVR code is to handle a richer set of input descriptions. Modern geometries require a mesher that can handle curved input surfaces and sharp corners; SVR on the other hand requires input to come from a highly restricted class (as do the algorithms underlying Pyramid and TetGen).

One of the elegant features of our SVR implementation is that although it is tuned for performance in three dimensions, the data structures are versatile enough for use in four or more dimensions, which is of possible use in emerging space-time meshing applications. The only routines we need for a higher dimensional implementation are fast yet robust in-sphere predicates in higher dimensions. Automated techniques for generating these predicates are known [NBH01], but the implementation is not currently available. Using exact arithmetic kernels such as are available with CGAL is a possibility, for small problems where the runtime is not critical.

Finally, we close by noting that SVR is known to be parallelizable [HMP07]. At the moment we feel there are still large constant factors to attack in the sequential implementation of the code; but as soon as those have been vanquished, the next obvious step will be to implement a multi-core version of this code. The current implementation has been explicitly geared toward leaving open that possibility.

Acknowledgments

We wish to acknowledge the help of Jonathan Shewchuk in offering many useful tips for the code and geometric predicates, for making available to us pre-release version of Pyramid, and for sending us some sample PLC inputs. We also acknowledge Hang Si's help in understanding TetGen; mesh drawings and mesh quality statistics were generated by TetGen.

References

[BEG94] Marshall Bern, David Eppstein, and John Gilbert. Provably good mesh generation. *J. Comput. Syst. Sci.*, 48(3):384–409, 1994.

[Bla05] Daniel K. Blandford. *Compact Data Structures with Fast Queries*. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, October 2005. CMU CS Tech Report CMU-CS-05-196.

[CD03] Siu-Wing Cheng and Tamal K. Dey. Quality meshing with weighted delaunay refinement. *SIAM J. Comput.*, 33(1):69–93, 2003.

[Che89] L. Paul Chew. Guaranteed-quality triangular meshes. Technical Report TR-89-983, Department of Computer Science, Cornell University, 1989.

[CP03] Siu-Wing Cheng and Sheung-Hung Poon. Graded Conforming Delaunay Tetrahedralization with Bounded Radius-Edge Ratio. In *Proceedings of the Fourteenth Annual Symposium on Discrete Algorithms*, pages 295–304, Baltimore, Maryland, January 2003. Society for Industrial and Applied Mathematics.

[ELM$^+$00] Herbert Edelsbrunner, Xiang-Yang Li, Gary L. Miller, Andreas Stathopoulos, Dafna Talmor, Shang-Hua Teng, Alper Üngör, and Noel Walkington. Smoothing and cleaning up slivers. In *Proceedings of the 32th Annual ACM Symposium on Theory of Computing*, pages 273–277, Portland, Oregon, 2000.

[HMP06] Benoît Hudson, Gary Miller, and Todd Phillips. Sparse Voronoi Refinement. In *Proceedings of the 15th International Meshing Roundtable*, pages 339–356, Birmingham, Alabama, 2006. Long version available as Carnegie Mellon University Technical Report CMU-CS-06-132.

[HMP07] Benoît Hudson, Gary L. Miller, and Todd Phillips. Sparse Parallel Delaunay Refinement. In *19th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 339–347, San Diego, June 2007.

[HPU05] Sariel Har-Peled and Alper Üngör. A Time-Optimal Delaunay Refinement Algorithm in Two Dimensions. In *Symposium on Computational Geometry*, 2005.

[Li03] Xiang-Yang Li. Generating well-shaped *d*-dimensional Delaunay meshes. *Theor. Comput. Sci.*, 296(1):145–165, 2003.

[LT01] Xiang-Yang Li and Shang-Hua Teng. Generating well-shaped Delaunay meshes in 3D. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium* *on Discrete algorithms*, pages 28–37, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[Mil04] Gary L. Miller. A time efficient Delaunay refinement algorithm. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 400–409, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[MPW02] Gary L. Miller, Steven E. Pav, and Noel J. Walkington. Fully Incremental 3D Delaunay Refinement Mesh Generation. In *Eleventh International Meshing Roundtable*, pages 75–86, Ithaca, New York, September 2002. Sandia National Laboratories.

[MTTW99] Gary L. Miller, Dafna Talmor, Shang-Hua Teng, and Noel Walkington. On the radius–edge condition in the control volume method. *SIAM J. Numer. Anal.*, 36(6):1690–1708, 1999.

[MV00] Scott A. Mitchell and Stephen A. Vavasis. Quality Mesh Generation in Higher Dimensions. *SIAM Journal on Computing*, 29(4):1334–1370, 2000.

[NBH01] Aleksandar Nanevski, Guy E. Blelloch, and Robert Harper. Automatic Generation of Staged Geometric Predicates. In *International Conference on Functional Programming*, pages 217–228, Florence, Italy, September 2001.

[Rup95] Jim Ruppert. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, 18(3):548–585, 1995.

[She97] Jonathan Richard Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry*, 18(3):305–363, October 1997.

[She98] Jonathan Richard Shewchuk. Tetrahedral Mesh Generation by Delaunay Refinement. In *Proceedings of the Fourteenth Annual Symposium on Computational Geometry*, pages 86–95, Minneapolis, Minnesota, June 1998. Association for Computing Machinery.

[She99] Jonathan Richard Shewchuk. Lecture notes on geometric robustness, 1999.

[She05a] Jonathan R. Shewchuk. Pyramid, 2005. Personal communication.

[She05b] Jonathan R. Shewchuk. Triangle, 2005. `http://www.cs.cmu.edu/~quake/triangle.html`.

[Si06] Hang Si. On refinement of constrained Delaunay tetrahedralizations. In *Proceedings of the 15th International Meshing Roundtable*, 2006.

[Si07] Hang Si. TetGen, 2007. `tetgen.berlios.de`.

[STÜ07] Daniel Spielman, Shang-Hua Teng, and Alper Üngör. Parallel Delaunay refinement: Algorithms and analyses. *IJCGA*, 17:1–30, 2007.

[Vav00] Stephen A. Vavasis. QMG, 2000. `http://www.cs.cornell.edu/home/vavasis/qmg-home.html`.