Dynamic Mesh Refinement

Benoît Hudson

CMU-CS-07-162
December 2007

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Gary L. Miller, Chair
Anupam Gupta
Daniel D.K. Sleator
Umut A. Acar, Toyota Technological Institute at Chicago
Jonathan R. Shewchuk, University of California, Berkeley

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

This work was supported in part by the National Science Foundation under grants
ACI-0086093, CCR-0122581, and CCR-0085982.

Keywords: Computational Geometry, Scientific Computing, Mesh Refinement, Dynamic Algorithms

Abstract

Mesh refinement is the problem to produce a triangulation (typically Delaunay) of an input set of points augmented by *Steiner* points, such that every triangle or tetrahedron has good *quality* (no small angles). The requirement arises from the applications: in scientific computing and in graphics, meshes are often used to discretely represent the value of a function over space. In addition to the quality requirement, the user often has input segments or polygons (generally, a piecewise linear complex) they would like see retained in the mesh; the mesh must *respect* these constraints. Finally, the mesh should be *size-conforming*: the size of mesh elements should be related to a particular sizing function based on the distance between input features.

The static meshing problem is increasingly well-understood: one can download software with provable guarantees that on reasonable input, the meshes will have good quality, will respect the input, and will be size-conforming; more recently, these algorithms have started to come with optimal runtimes of $O(n \lg(L/s) + m)$, where L/s is the spread of the input. As a first result, I present experimental results of the first time-optimal code, available online at `sparse-meshing.com`.

Increasingly, static meshing is insufficient: users want to modify the mesh over time. Throwing away the old mesh and remeshing from scratch is a common approach, but that suffers from slow runtime, and from reinterpolation error because the old and new meshes may be almost unrelated. Mesh *stability* analyzes the correspondence between meshes for two inputs. The main theoretical result of this thesis is an algorithm that has provable bounds on stability: upon inserting or removing a feature that in the final mesh is represented using k points, the mesh only modifies $O(k \lg(L/s))$ mesh simplices.

Finally, stability can be exploited to produce an efficient *dynamic* algorithm. Under the *self-adjusting computation* framework, with a small amount of additional effort, I show that my algorithm can be dynamized to run in $O(k \lg(L/s))$ time per update, using $O(n \lg(L/s) + m)$ space.

I owe a great debt to my parents, Richard Hudson and Gisèle Chevalier, who raised me a scientist and an academic. It took me 23 years, but when I said I'd write a thesis when I'd grow up, I wasn't kidding.

My best teachers in France and in Canada all taught me geometry, which speaks well for the field. At Brown, Roberto Tamassia showed me you could both be a geometer and a computer scientist; he and the rest of the CS department gave me an early taste for teaching and research. Mark Johnson in Linguistics and Carlé Pieters in Geology let me discover that though I was a computer scientist by training and trade, applying my knowledge to the pursuit of the sciences made it all the more exciting.

My advisor, Gary Miller, somehow managed to instill upon me the ability to do research. Through constant meetings occasionally turning into shouting matches with him and with Todd Phillips, we have muddled through vast wastelands of bad ideas and somehow managed to pick out one or two good ones in passing. Umut Acar, for reasons unclear, took a chance that perhaps we should work together; no algorithms would have been dynamized without his help. Jonathan Shewchuk has always been in the background, if not the foreground. Much of my meshing work bears his imprimatur, as does my tea cabinet.

Contents

Chapter 1

Introduction and background

In scientific computing, graphics, and in many geometric processing problems, a key task is to take an input geometry and tile it with a collection of small objects that are easier to handle, such as triangles, tetrahedra, or perhaps cubes — a **mesh**. The ancients already did this in order to produce mosaics (using tiles instead of the more modern tendency to use pixels). Topologically, each of the smaller elements should have a small description, to make them easy to manipulate and easy to reason about, hence the use of triangles and cubes. Geometrically, applications impose requirements on the shape of each element: in the applications I consider, the requirement will be that adjacent elements have similar size (for an appropriate definition of size), and that each element has bounded distortion (for an appropriate definition of distortion).

The goal here is to represent a function f that is continuous over space, and operate on it. On each triangle (or on each tetrahedron, in three dimensions), we can set a value at each vertex and linearly interpolate within the element. This yields a piecewise linear approximation $\hat{f}$ to f. Assuming that f has bounded first and second derivatives in all directions, if every element of the mesh is not too large, and matches a certain quality criterion described below, then $\hat{f}$ is a good approximation to f under the H_1 norm: that is, the gradient of f approximates the gradient of $\hat{f}$.

Mesh refinement is the task of taking as input the description of a geometric object, possibly *refining* it by adding additional vertices, and producing as output a set of vertices and triangles (or tetrahedra) that tile space and all have good quality. Of course, applications using the mesh will run in time governed by the size of the mesh, so it is desirable to output as few additional vertices and as few triangles as possible. Finally, in a timestepping finite element simulation, such as would be used to solve a hyperbolic PDE, under some commonly-used solution methods, the maximum allowed length of the timestep is governed by the size of the smallest element. Therefore, mesh elements must not be too small.

1.1 Quality measures

In a finite element application, f will be a property such as heat whose value we are trying to calculate while simulating a physical process. The approximation quality of $\hat{f}$ defines how accurate the simulation is. Classically, Babuška and Aziz [BA76] proved that in order for the Finite Element Method to produce an accurate solution, the mesh over which the simulation is run must not contain any angles between two segments that are close to $180°$ — that is, if the largest angle in the mesh is $180 - \epsilon°$, they proved an error bound that is a constant function of ϵ. This gives rise to the **no large angles** condition in mesh refinement. In three dimensions, the angles of interest are face angles (the angles between two segments on a triangular face) and dihedral angles (the angles between two triangular faces on a tetrahedron). Solid angles do not affect the solution quality.

Typically, the values at each vertex that define $\hat{f}$ will be computed by solving a linear system of equations $Ax = b$, where the entries of A are affected by the shape of the elements. Many solvers have their runtime regulated in part by the condition number of A. While large face or dihedral angles will cause $\hat{f}$ to be a bad approximation of f, small face, dihedral, or solid angles will cause the condition number to degenerate [She02]. Therefore, in meshing, the goal is usually to produce meshes with no angle close to $0°$, which is called the **no small angles** condition. A mesh with no angle of any type close to zero has no angle of any type close to $180°$: the no small angles condition is strictly harder to satisfy than the no large angles condition.

The analyses of meshing algorithms are much simplified by using different notions of quality than the angles. In two dimensions, saying that every angle in a triangle is larger than α is equivalent to saying that it has radius/edge ratio no larger than ρ (see Figure 1.1). In such a case, I say the triangle has "good" radius/edge ratio. Another equivalent definition is the aspect ratio, which is variously defined (see Figure 1.2). In three and higher dimensions, these definitions all generalize in the obvious way. However, their correspondences are not maintained. In particular, a simplex with good radius/edge ratio may have a small angle; such a simplex is called a **sliver**. See Figure 1.3.

Since the early days of theoretically-proved automatic meshing, it has been known how to provably produce meshes with good radius/edge ratio (although developing software to do so has been a greater challenge); techniques can typically prove that they will output a mesh with radius/edge ratio no worse than $\sqrt{2}$ in two dimensions, or 2 in three

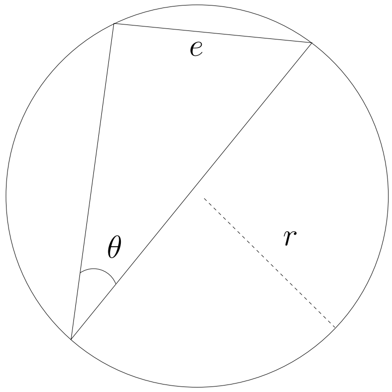


Figure 1.1: Illustration of the radius/edge ratio quality criterion. Draw the circumscribing circle of a triangle; it has radius r. Let e be the length of shortest edge of the triangle. The **radius/edge ratio** is r/e. In two dimensions, if θ is the smallest angle, then $r/e = \frac{1}{2\sin\theta}$. Therefore, a triangle with small radius/edge ratio has no small angles. The correspondence only holds in two dimensions.

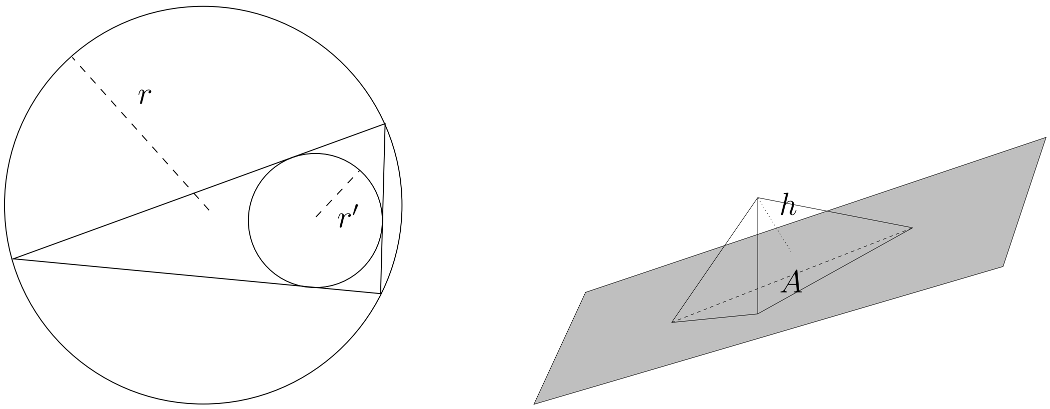


Figure 1.2: Illustration of two common definitions of the **aspect ratio criterion**. Draw the circumscribing circle of a triangle; it has radius r. (A) Draw the inscribed circle of a triangle; it has radius r'. A triangle has good aspect ratio if $r/r' \leq \sigma$ for some constant σ. This is often also called the **radius ratio**. (B) Alternatively, let A be the area of the largest $(d-1)$-face of the simplex. Let h be the height of the simplex — the distance from the remaining point to the plane defined by the smallest face. Then the simplex has good aspect ratio if $A/h^{d-1} \leq \sigma$. The two quality measures are equivalent up to constants. Unfortunately, the literature uses the same term for both, and also sometimes a for different power of these criteria, including their reciprocal.

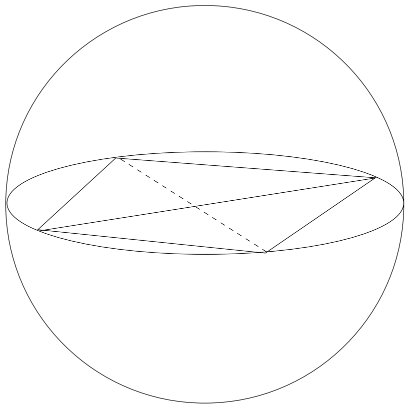


Figure 1.3: A sliver in three dimensions. The four points are equally spaced about the diameter of the sphere. Therefore, the radius/edge ratio is $\sqrt{2}$. However, the largest ball inscribed to the tetrahedron has zero radius, so the aspect ratio is infinite (equivalently, the height of the fourth point off any face is zero). Note also that the sliver has dihedral angles of both $0°$ and $180°$, but any two neighbouring segments form a $45°$ or $90°$ angle. In higher dimension, a sliver is defined by Li to be a simplex that has good radius/edge, but either the entire simplex has bad aspect ratio, or one of the faces is itself a sliver [Li03].

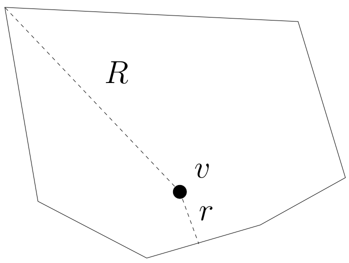


Figure 1.4: The Voronoi aspect ratio of v is defined by the distance R from v to the farthest point in its Voronoi cell (*i.e.* the largest radius of any Delaunay simplex around v) and by the radius r of the largest ball at v that does not leave the Voronoi cell (which is half the distance to the nearest neighbour). If $R/r \leq \tau$, the Voronoi cell has good aspect ratio. Note that if for all cells, $R/\,\mathrm{NN}(v) \leq \rho$, then the Delaunay triangulation has radius/edge ratio at most ρ. Conversely, for any Delaunay mesh with radius/edge ratio bound ρ, there is a constant τ_ρ that bounds the Voronoi aspect ratio of the same point set [Tal97] (this fact is trivial in two dimensions but requires substantial proof in higher dimension).

dimensions. Much less is known about eliminating slivers; see the Related Work section for details. However, many techniques exist that take as input a mesh with every element having good radius/edge ratio, and produce as output one with no slivers, with varying levels of theoretical and practical success (the provable aspect ratios are tiny, but in practice they generally eliminate dihedral angles smaller than about $1°$ to $5°$, and larger than about $175°$ to $179°$ depending on the technique and implementation). Furthermore, as proved by Miller *et al*, slivers do not affect solution quality in the Control Volume Method, which is a simulation technique comparable to the Finite Element Method [MTT$^+$96]. All that matters for the Control Volume Method is that the Voronoi cell of every point have good aspect ratio (as defined in Figure 1.4), a quality that is implied by ensuring the simplices all have good radius/edge ratio. Alternatively, we can directly achieve good aspect ratio Voronoi cells. As we discovered while developing SVR, and I also found while developing this thesis, it is substantially more convenient to theoretically analyze the use of the Voronoi aspect ratio condition.

1.2 Input description

The task of describing the input geometry is a field of its own (namely, CAD). At minimum, an input geometry should allow for specifying points that will appear in the mesh. The mesh should then fill a certain *domain* Ω. I take the domain to be a box, sufficiently larger than the input geometry that no meshing activity occurs near the boundary of the box (see Section 4.4). Quite naturally, the user may also want to ensure that certain segments, denoted as a pair of points, appear in the mesh. In three dimensions, the user may also want to ensure that certain polygons appear. Such an input is termed a *Piecewise Linear Complex* by Miller *et al* [MTT$^+$96]. Copying their definition:

Definition 1.2.1 (Piecewise Linear Complex [MTT$^+$96, Definition 3.2]) *A **piecewise linear complex (PLC)** is a set $\mathcal{X}$ of polytopes with the following properties:*

- *The set is closed under taking boundaries: For each $f \in \mathcal{X}$, the boundary of f is a union of polytopes in $\mathcal{X}$.*
- *$\mathcal{X}$ is closed under intersection: For any two f and f' in $\mathcal{X}$, the intersection $f \cap f'$ is itself a polytope in $\mathcal{X}$.*
- *If $\dim(f \cap f') = \dim(f)$ then $f \subseteq f'$ and $\dim(f) < \dim(f')$*

The algorithms and software I present in this thesis further require that if two polytopes f and f' intersect, then either $f \subseteq f'$ or they form an angle of at least 90° — they must intersect at **non-acute** angles. This has two effects. For one, it limits the number of higher-dimensional polytopes that are incident on any single polytope (for instance, it limits the number of segments incident on a point). The other effect is that if we place the center of a ball b on a feature f, and b does not intersect the boundary of f, then any other feature f' that b intersects is disjoint from f.

The input description is somewhat restrictive: it does not allow for curves, nor does it allow for for even moderately sharp angles. It is likely possible to extend the algorithms herein with relatively minimal difficulty to allow for input that also allows arcs and Bézier curves, perhaps building upon the work of Ollivier-Gooch and Boivin [BOG02]. There is substantially less work on meshing with curved surfaces, but there does not seem to be a fundamental difficulty to overcome. Acute angles are a much harder problem; see the Related Work in Section 1.7.

Given a PLC $\mathcal{X}$, I need to define a notion of what it means to represent $\mathcal{X}$ using a mesh. A mesh M is itself a PLC. Following the literature, I say that M **respects** $\mathcal{X}$ if every polytope in $\mathcal{X}$ appears as a union of polytopes in M. For example, a segment in $\mathcal{X}$ should show up as a set of edges of the mesh simplices, and a polygon in $\mathcal{X}$ should appear as a set of triangles in the mesh. In practice, this is essentially impossible: due to numerical error, it is very likely that the set of segments in the mesh that ostensibly respects a segment in $\mathcal{X}$ in fact do not coincide. Using exact arithmetic would solve this issue, but requires asymptotically more time and space. Using robust numerical predicates [She97a] only somewhat mitigates the issue. In the algorithms, I assume a model of computation where we can apply the standard arithmetic operations to real numbers of exact precision in constant time, and leave it to future work to analyze the algorithms under a floating-point model.

A useful metric for the input space is its *spread*. Let L be the length of a side of the box that defines the meshing domain. Let s be the shortest distance between any two disjoint features. Then the spread is L/s. The quality measures each imply that at most, two neighbouring elements in the output mesh have about the same size, to within a constant. Therefore, the quantity $\log L/s$ appears repeatedly in this thesis and in the literature. Assuming integer inputs with words of length w, $L \leq 2^w$ while $s \geq 1$. Normally the assumption is that $w = O(\lg n)$, so $L/s \in \mathrm{poly}(n)$. With floating point input (where a floating-point number consists of a word-sized integer for the exponent, and a word-sized integer for the mantissa), the spread can be exponential in n.

In the application at hand — meshing for scientific computing and graphics — the length of a timestep is commensurate with the size of the smallest element, which must of course be commensurate with s. If we intend to simulate a period of simulated time of about $\Theta(L)$, then it will take $\Theta(L/s)$ timesteps to do this. This motivates taking on the assumption, common in computational geometry, that even when accepting floating point or real input we can consider the spread to be only polynomial in n.

1.3 Sizing functions

A very important definition, from Ruppert [Rup95] and extended to higher dimension by Miller *et al* [MTT$^+$96], is that of the local feature size, which produces a continuous function over space that describes the spacing between input features near any point.

Definition 1.3.1 (Local feature size) *Given a PLC $\mathcal{X}$, the local feature size at a point x in space, denoted* $\operatorname{lfs}(x)$*, is the radius of the smallest ball centered that x that encloses a pair of disjoint features f and f' (*i.e.*, $f \cap f' = \emptyset$). The local feature size is a Lipschitz function: for any two points x and y,* $\operatorname{lfs}(x) \leq \operatorname{lfs}(y) + ||xy||$.

In two dimensions, the following holds: consider a mesh of a domain Ω, where the mesh respects the input $\mathcal{X}$ and every element has good radius/edge quality. It may or may not be Delaunay. Ruppert proves [Rup95] that every point $x \in \Omega$ lies in a triangle τ that has circumradius r such that $r \leq O(\operatorname{lfs}(x))$. This implies, not entirely trivially, that even in the smallest possible valid mesh, the number m_{opt} of triangles obeys $m_{opt} \in \Omega(\int_\Omega \frac{1}{\operatorname{lfs}^2(x)} dx)$. At the same time, Ruppert shows an algorithm that produces a mesh where every point $x \in \Omega$ lies in a triangle with radius $r \geq \Omega(\operatorname{lfs}(x))$. This implies that the mesh his algorithm produced had size $m \in O(\int_\Omega \frac{1}{\operatorname{lfs}^2(x)} dx)$. In other words, $m \in \Theta(m_{opt})$: the mesh that Ruppert's algorithm produces is constant-competitive with the optimal mesh that any algorithm could produce.

The proof depends on the mesh being of good aspect ratio. In three dimensions, it still holds for any good aspect ratio simplicial mesh. However, the statement that $r \in O(\operatorname{lfs}(x))$ is no longer true when demanding only good radius/edge meshes. Shewchuk [She98b] shows an example of two skew edges, which generate tiny local feature size where the two edges almost meet. However, a sliver can resolve both edges. This discorrespondence complicates the analysis of the size guarantees of higher-dimensional meshing algorithms. Meshing algorithms that produce good radius/edge meshes, including the ones I mention here, typically still prove that $r \geq \Omega(\operatorname{lfs}(x))$.

Instead of discussing the size of the meshes my algorithms produce relative to the optimal radius/edge mesh, I will show that the meshes are **size-conforming**. That is, the local feature size defined by the vertices of the final output mesh is a function not far different than the local feature size defined by the input. More formally, define the **local mesh size** at $x \in \Omega$, $\operatorname{lms}(x)$ as the distance from x to the second-nearest vertex of the output mesh.

Definition 1.3.2 (Size-conforming) *A mesh made up of a finite set of vertices $V \subset \mathbb{R}^d$, such that the mesh respects an input PLC $\mathcal{X}$ and tiles a domain Ω, is **size-conforming** if for every $x \in \Omega$,*

$$\operatorname{lms}(x) \leq \operatorname{lfs}(x) \leq c \operatorname{lms}(x)$$

for some constant c.

As proved by Ruppert (after the obvious generalization to higher dimension), a size-conforming mesh contains $\Theta(\int_{x \in \Omega} \operatorname{lfs}^{-d}(x) dx)$ elements. This is the same number of elements as in the smallest mesh that respects $\mathcal{X}$ and has good aspect ratio. Algorithms exist that take as input a size-conforming mesh with good radius/edge quality, and produce a good aspect ratio mesh of either exactly the same size or only linearly more vertices [ELM+00, CDE+00, LT01, Li03].

An interesting aside is that in the field of surface reconstruction, the local feature size, defined only on the manifold being reconstructed, is the distance to the medial axis. This alternative description matches the description I gave here everywhere on $\mathcal{X}$, assuming the angle between intersecting segments is obtuse (strictly larger than $90°$); but if the angle is right or acute, the medial axis touches the surface and thereby defines regions with $\operatorname{lfs}(x) = 0$. The surface reconstruction definition of local feature size is well defined on curved inputs. This is another reason to expect that extending the techniques here to be able to mesh curved input will be a smaller change than finding algorithms that can handle acute angles.

Often in scientific computing, the local feature size is insufficiently fine: to visualize why, think of an eddy swirling in the middle of the ocean. We will need small elements simulate it accurately, even though the geometry suggests using huge elements there since the coastline is so far away. In the present work, I abstract away from such requirements; they can be imposed for instance by adding a few additional points where further refinement is needed.

1.4 Static and dynamic algorithms

Traditionally, the meshing problem has been to take an input description (such as a PLC) and produce a quality mesh. Practitioners assume that a mesh will be used for hundreds of simulations, each one running for minutes to hours. Therefore, the computation time spent generating the mesh is no object; instead, mesh quality, mesh size, and most of all correctness are far more important than runtime.

Increasingly, however, we are interested in simulating processes where the geometry changes during the simulation: blood cells moving through a channel, pump blades rotating, valves opening, etc. Another realm of interest is in optimizing the shape of an engineering component: the task is to take an input geometry, run a simulation and compute a quantity of interest (such as, for a heart pump, the maximum strain any blood cell will suffer), then to automatically change the geometry slightly and re-run the simulation. If the new geometry is better with respect to the quantity of interest, repeat the process, in a hill-climbing approach.

In the algorithms literature, there are two closely related concepts that are sometimes conflated into the term "dynamic." According to the definitions I prefer to use, in the **dynamic** setting, we take a fixed input and compute the solution to a problem. Then, the adversary can add or remove part of the input, and ask us to update the solution accordingly the problem. Let n be the greater of the size of the input before or after the change. An algorithm *responds* to the change in time $O(f(n))$ if it can, starting with the initial output and some other data structures, change the output to be a valid answer to the new problem, in time $O(f(n))$. In terms of meshing, one could use a dynamic meshing algorithm to maintain a quality mesh as, for example, a crack creeps through the domain, one break at a time; or when a solenoid valve is essentially instantaneously opened. It is in this sense that the algorithm I present that forms the core of the thesis is a dynamic algorithm.

In a **kinetic** setting, the input size does not change over time [Gui98]. Instead, the input (assumed to be geometric) changes continuously over time. It is assumed that the input will change discontinuously only at specific points in time. The goal is to maintain a valid output and some internal data structures as the input continuously moves. The analysis will normally compare the number and cost of changes in the internal data structures, against the number of changes to the output. A kinetic mesher would be one that maintains a quality mesh as fluid entrains a blood cell, for example. The kinetic meshing problem (or, more commonly, the *moving mesh problem*) is beyond the scope of my thesis, though there are deep links between the dynamic and kinetic settings.

History independence, in the sense defined by Micciancio [Mic97], states that an algorithm will produce an identical output given an input without regard to the history of changes. That is, if after a long series of changes, kinetic or dynamic, we were to rerun the algorithm from scratch on the current input, it would produce an output that is equivalent to the one that was dynamically maintained. The notion of equivalence is usually obvious: In the meshing problem, equality is defined by the coordinates of the mesh vertices, and the topology of the mesh. The advantage of history independence is that it limits the need to analyze the correctness of our algorithms to the simpler, static case; the disadvantage is that in some cases — meshing might come to mind, the algorithm has a lot of freedom in choosing the output. Specifying that the dynamization will be history-independent limits that freedom.

When an algorithm is history independent, it is well defined what the output is, given the static algorithm that is needed to compute the initial solution. This brings up the question of automatically dynamizing a static algorithm. The task is to simulate rerunning the static algorithm from scratch, while hopefully spending time related to a notion of a difference between the two runs of the algorithm. Acar *et al* formalized one such approach, calling it **self-adjusting computation (SAC)**, and provided algorithms to indeed efficiently simulate a static algorithm under dynamic or kinetic changes to the input [Aca05, ABBT06, ABT06]. We can describe the run of a program essentially as a circuit (I use more formal notions later), where each gate reads in a bounded number of operands and produces a result. Updating to run on a new input involves propagating values through the circuit, but also creating new sub-circuits (operations that are performed in one but not the other run), and eliminating old sub-circuits. The algorithm that performs this propagation of changes is imaginatively called the **change propagation** algorithm. In order to avoid propagating changes through parts of the circuit that it will later decide are not needed, SAC maintains an order maintenance structure that defines a total ordering on the gates — namely, the order in which the static algorithm initially ran the corresponding operations. The change propagation algorithm uses a priority queue to ensure that it re-executes the program in order, and thereby avoids updating a computation present in the old circuit, but no longer present in the new one. Efficient updates depend on the following definitions:

Definition 1.4.1 (Trace [Aca05, Definition 8]) *The **trace** is an ordered, rooted tree that describes the execution of a program P on an input. Every node corresponds to a function call, and is labeled with the name of the function; its arguments; the values it read from memory; and the return values of its children. A parent-child relationship represents a caller-callee relationship.*

Definition 1.4.2 (Cognates and Trace Distance [Aca05, Definition 12]) *Given two traces*

*T and T' of a program P, a node $u \in T$ is a **cognate** of a node $v \in T'$ if u and v have equal labels. We say a program is **concise** if a node $u \in T$ does not have a cognate in T other than itself. The **trace distance** between T and T' is equal to the symmetric difference between the node-sets of T and T', i.e., distance is $|T| + |T'| - 2|C|$ where C is the set of cognates of T and T'.*

Definition 1.4.3 (Monotone Programs [Aca05, Definition 15]) *Let T and T' be the trace of a concise program with inputs that differ by a single insertion or deletion. We say P is **monotone** if operations in T happen in the same order as their cognates in T' during a pre-order traversal of the traces.*

The main theorem of Acar [Aca05] states that for monotone programs, the time for change propagation is the same as the trace distance if the priority queue overhead can be bounded by a constant. For the theorem, we say that a program is **O**$(f(n))$**-stable** for some input change, if the distance between the traces T, T' of the program with inputs I and I', where I' is obtained from I by applying the change, is bounded by $O(f(n))$. For the proofs, I will generally abstract away from trace nodes and use a more fuzzy notion of an "operation," and show that there are at most a constant number of trace nodes per operation.

Theorem 1.4.4 (Update time [Aca05, Theorem 34]) *If a program P is monotone under a single insertion/deletion, and is $O(f(n))$-stable, and if the priority queue can be maintained in $O(1)$ time per operation, then change propagation after an insertion/deletion takes $O(f(n))$ time.*

Given that SAC is history-independent, the space usage of SAC is bounded by the static runtime of the algorithm. Even if the update time is large, SAC stores only an amount of memory linear in the length of the current trace. In some cases, it may be possible to further reduce the memory usage, as I will briefly discuss when dynamizing the quadtree algorithm.

The method I use to prove my algorithm are $O(f(n))$-stable will be an exercise quite familiar to anyone who has designed a parallel algorithm. Indeed, the key idea is to define a notion of dependency: operation a *depends* on operation b if b must be computed before a. To prove a parallel bound, we must quite explicitly reason about dependencies and show that any path of dependency is short. The dynamic setting is both more stringent and more permissive: not only must every dependency path be short, there must not be too many paths, because the change propagation is run in series. On the other hand, a dependency path may be long without causing issues in the dynamic response time if change propagation can stop propagation (because an operation being propagated wrote the same value to its output as it had written before).

In this thesis, I will give stability bounds for various algorithms. The way in which they are written strongly implies that the algorithms are also parallelizeable. Naively parallelizing them at the moment will add a logarithmic term in the runtime. It is likely that one could use methods other than SAC to dynamize (for instance, a hand-coded specialization thereof), without substantially changing the theoretical analysis. In particular, the bounds I give also imply that even after throwing away the mesh re-running the static algorithms from scratch, the new mesh will mostly match the old, avoiding reinterpolation error.

1.5 Bucketed priority queues

Meshing algorithms (my own included) frequently recommend performing operations in a particular order that depends on the geometry of the items in the queue. For instance, it may be advantageous to process tetrahedra in order of largest radius first as per Miller [Mil04], or shortest shortest-edge first as per Har-Peled and Üngör [HPÜ05]. The most natural approach is to put the tetrahedra to be processed in a priority queue, keyed by their characteristic length (the length of the shortest edge, or the reciprocal of the radius). Then the algorithm can INSERT all the tetrahedra into a priority queue, and call DELETEMIN to decide which element to process next. Generally in meshing, the DECREASEKEY operation is not needed; a DELETE operation may be useful for tetrahedra that are destroyed before being processed, but it is in practice both simpler and faster to simply ignore a result from DELETEMIN that refers to a deleted simplex.

In a comparison model, computing the exact minimum will take $O(\lg |Q|)$ per operation, where $|Q|$ is the number of items in the priority queue. However, it is often the case that we need only approximate the order in the priority queue: if the true minimum has key l, then it is safe to report another item if the other item has key $l' < \gamma l$, for some constant $\gamma > 1$ that depends on the application. On its own, this allowance to approximate exponentially reduces the runtime, but the runtime still depends on $|Q|$.

However, in my application we know yet more: when an item with key l is removed from the queue and processed, the processing may add new items on the queue. Those new items will have key $\Theta(l)$, so they will go near the "front" of the priority queue. This allows achieving constant-time approximate priority queue operations. The structure is as follows: I store a sorted linked list. Each node in the linked list is a pair consisting of a number l, by which I sort; and a bucket — an unsorted set of items whose key lies in

$[l, \gamma l)$. INSERT on an item with key k searches from the head of the list until it finds the bucket that contains k, then adds the item to that bucket. If there is no such bucket, the insertion algorithm creates a new bucket and links it into the list. Adding to a bucket will take constant time, and by the assumption that new items have key with a constant factor of the previously-deleted item's key, only $O(1)$ buckets will need to be considered before finding the appropriate bucket or determining there is no such bucket. A DELETEMIN operation simply looks up the head of the stored list and removes an arbitrary item from that bucket, deleting the linked list node if the bucket is thereby emptied. Therefore, under these two assumptions — that we may approximate the priority queue order, and that items are created close in keyspace to the previously-deleted item — the priority queue costs are $O(1)$ per insertion and deletion.

When creating a new bucket, the question arises as to what number the bucket should take. For simplicity, we would prefer that buckets do not overlap; this is not critical, but shedding this requirement complicates the analysis without simplifying the implementation. The easiest way to ensure the non-overlap condition is for the first insertion into the bucketed queue to use a bucket number equal to the key of the item being inserted, and to subsequently use powers of γ times the first bucket number. Thus, when a new bucket is to be created, we look at the number of the next-smaller (or next-larger) bucket, and multiply (or divide) by γ until we find an appropriate number. We should be careful for the case of removing the item in the queue that has the smallest key, and there remaining only items with much larger keys — the repeated division approach would then take time logarithmic in the ratio between the deleted key and the next-smallest key. To handle this, we can simply remember the number of the last bucket that we deleted.

Alternately, if numbers are in a floating-point representation, then the bucket number can be read directly as the exponent of the key. For example, in the SVR implementation, I use a bucketed priority queue on tetrahedra in which the key is r^2, the square of the circumradius. Reading the floating-point exponent as an integer therefore buckets according to $[r, \sqrt{2}r)$. This can be a substantial constant factor faster than repeated division, and only requires storing a small integer rather than a floating-point double.

In the static case, there are many possible implementations of such a priority queue beyond the one I have mentioned. The details are immaterial to my later proofs, so I generalize to the following definition:

Definition 1.5.1 (γ-bucketed priority queue) *A **γ-bucketed priority queue** is a structure that supports* INSERT *and* DELETEMIN *calls. When the smallest key in the set is l, then* DELETEMIN *is guaranteed to spend $O(1)$ time before returning a value whose key is in $[l, \gamma l)$. Having deleted an item with key $[l, \gamma l)$, we can now* INSERT *items with key $\Theta(l)$ in*

A bucket is a pair $\langle$ $\mathbb{R}$, a set of items $\rangle$
Q is a record { buckets: sorted list of buckets, last: $\mathbb{R}$ }
The list is sorted by the bucket numbers, smallest first.

INSERT(Q, k: real, v: item)
Find the bucket $\langle l, S\rangle$ such that $k \in [l, \gamma l)$
if there is no such bucket **then**
    $S \leftarrow$ nil
    if there is no bucket **then**
        $l \leftarrow k$
        Set the "last" field to k
    else
        Compute i such that $k \in [\gamma^i\text{last}, \gamma^{i+1}\text{last})$
        $l \leftarrow \gamma^i\text{last}$
    end if
    Insert the bucket $\langle l, S\rangle$ into the buckets list in sorted order
end if
Add v to the list S

DELETEMIN(Q)
Read the first bucket of Q as $\langle l, S\rangle$
Set the "last" field to l
Remove the first element v of S
If S is now empty, remove the bucket
return v

Figure 1.5: The γ-bucketed priority queue described in the text. DELETEMIN is clearly deterministically constant-time. INSERT has a potentially expensive operation in finding the appropriate bucket; the assumption that new items will be close in size to the current minimum limits this cost to be constant. Computing i may be expensive or inexpensive, depending on machine model: with $\gamma = 2$ on an floating-point machine, it is constant time, whereas if only multiplication is allowed, it has the same asymptotic cost as finding the appropriate bucket.

constant time. More generally, when the minimum key in the queue is $\min$*, we can insert an item with key* k *in time* $O(|\log_\gamma \frac{k}{\min}|)$.

In the dynamic case, it is difficult to analyze the stability and response time of this γ-bucketed priority queue without knowing how it will be used. I therefore defer these questions until the dynamic analysis of my meshing algorithms, which will refer to the specific algorithm I described here.

1.6 Claims

Here are the claims I make for this thesis:

- The first implementation of a time-optimal Delaunay refinement code in two and three dimensions. On point clouds in three dimensions, it is the fastest known refinement code.
- The first dynamic algorithm for maintaining a size-conforming quadtree or octree over a point cloud, or over a suitable PLC $\mathcal{X}$. If a feature f intersects m_f quadtree cells when it is present, then adding it to the PLC or removing it from the PLC takes $O(m_f \lg L/s)$ time. The data structures to support this consume $O(n \lg L/s + m)$ space.
- The first dynamic algorithm for maintaining a quality, size-conforming simplicial mesh that respects a suitable PLC $\mathcal{X}$, in the same time and space bounds as to produce the quadtree. The algorithm works in any fixed dimension d.
- A sufficient condition for being able to provably ignore boundary effects while meshing.
- A new framework to use to guide the choice of Steiner points. The new framework gives somewhat greater freedom than traditional approaches, which is likely to be useful from both the theoretical and practical point of view.

1.7 Related work

Chew gave the first algorithm to provable produce a quality mesh in two dimensions [Che89]. His algorithm did not however produce a size-conforming mesh; instead, it produced a nearly uniform mesh, where all triangles were about the same size. Meshes generated by Chew's algorithm are therefore often must larger than needed. Ruppert adapted Chew's algorithm and proved that his own algorithm produces a graded mesh of good quality, with only a constant factor more vertices than is optimal [Rup95, first published 1992]. He lay the foundations (discussed in Section 1.3) for analyzing the size of a mesh as compared to the optimal. Both these algorithms use so-called Delaunay refinement: as an invariant, they maintain the Delaunay triangulation [Del34] or Constrained Delaunay triangulation [Che87] of the input and any additional Steiner points inserted so far. Iteratively, they detect a triangle t which is unsatisfactory (has bad radius/edge quality, for example). Recall that the Delaunay triangulation is defined as those triangles whose circumscribing disc is empty. Therefore, if a triangle is unsatisfactory, it is quite natural and effective to insert the center of that disc, and recompute the Delaunay triangulation. In three dimensions, the pattern was repeated by Chew [Che97] and later Shewchuk [She98b], who first showed how to produce a quality radius/edge mesh with uniform element size, then with graded elements. None of these algorithms claim any interesting runtimes, and indeed some lower-bound examples exist that can make the algorithms take quadratic time. The situation is even worse in three dimensions, where the initial Delaunay triangulation can have size $\Omega(n^2)$ even as the output has size only $O(n)$.

The chief practical difficulties with the Delaunay refinement algorithms have been the difficulty they have handling of small input angles, and the tendency in three dimensions to produce slivers. It is known in two dimensions that Ruppert's algorithm works as-is on inputs with angles of 60° or more between any two segments. The same holds in three dimensions, but this is small comfort since two polygonal facets must meet at non-acute angles, and segments are normally only used to bound facets. Various simple tricks can be used to ensure that Ruppert's algorithm terminates despite much smaller angles. Even defining what the proper input should be in these cases is tricky: if the input has an angle of 1°, it is fundamentally impossible to both respect the input and produce a mesh with no angle less than 20°. Simple tricks such as Shewchuk's "terminator" can produce meshes with no small angles except "near" a small input angle. Sadly, in three dimensions, no simple tricks are known that have interesting provable bounds. Eliminating slivers from three-dimensional meshes is an industry unto itself. Among results that can provably eliminate slivers, I outline the work of Edelsbrunner *et al* and of Cheng *et al* [ELM$^+$00, CDE$^+$00], which showed that slivers are brittle objects that disappear when faced with minor perturbations, even without adding new Steiner points. Alternatively, Chew showed how to eliminate slivers in a uniform mesh by being careful when adding Steiner points, inserting not the circumcenter but a point close to the circumcenter [Che97]. Randomly choosing a point near the circumcenter, checking whether it will created a sliver, and trying again if it did, we can be assured that no slivers will remain in the output. Li and Teng extended this to graded meshes by allowing the creation of slivers, if they are substantially larger than the simplex being processed; the larger slivers can then recursively be addressed [LT01, Li03]. While these techniques provably produce meshes with aspect ratio at most some constant σ, the value of σ they prove they can achieve is miniscule. Labelle [Lab06] demonstrated that by running standard Delaunay refinement but using lattice points rather than circumcenters, he can prove that the output mesh will have no dihedral angle less than $30°$ for point cloud input. Lattice refinement has not yet been extended to handle PLC features. It is very likely that both the Li-Teng technique and Labelle's lattice refinement can be used in a dynamically-stable code.

Closely related to the Delaunay meshing algorithms are the ball-packing algorithms [MTT$^+$96, Tal97, LTU99b, LTU99a]. These operate by computing a set of non-overlapping balls with radius driven by a spacing function such as the local feature size. The mesh vertices are at the center of each ball. These can be also made to take account of PLC input [MTT$^+$96]. In spirit, Delaunay refinement is just a flavour of ball-packing refinement: both involve inserting points at the center of an empty ball. Unfortunately, ball packing quite explicitly adds as many points as possible and has in practice been found to create large meshes [LTU99b]. Related to dynamic updates, Miller, Talmor, and Teng [MTT99] showed how to use ball packing techniques to coarsen a mesh, as might be required to do after removing a feature from the input.

Simultaneously with the developments in the realm of Delaunay refinement, Bern, Eppstein, and Gilbert showed how to use a quadtree to produce a quality mesh [BEG94, first published 1990]; Bern, Eppstein and Teng later parallelized the algorithm [BET99, first published 1993]. The technique defines a quadtree whose squares are size-conforming, then warps the corners of the quadtree cells to respect the input points. In the presence of segments, it treats the intersection between a segment and the side of a quadtree cell as an input point. Finally, they use a stencil to show how to triangulate the resulting shapes. Mitchell and Vavasis soon extended this to higher dimension [MV00, first published 1992]. Bern *et al.* showed that if the input is defined by n vertices, and the output contains m cells, then their algorithm runs in $O(n \lg n + m)$ time on integer or floating-point input. Mitchell and Vavasis show the same assuming constant-time tests to see whether a quadtree cell and a polygon intersect, barring which their algorithm can degenerate to $O(mn)$ time; it is sufficient to require that the polygons are defined by a bounded number of segments on their boundary. In other words, quadtree-based meshes have asymptotically the same optimal size as Delaunay-based meshes, but can be produced asymptotically faster. However, in practice, algorithms based on quadtrees output many more points than do Delaunay refinement algorithms.

The first interesting running time proved on Delaunay refinement was by Spielman, Teng, and Üngör, who showed that Ruppert's algorithm can be run in $O(\lg^2 L/s)$ parallel rounds, with each round running in $O(\text{polylog}(m))$ parallel time [STÜ07, first published 2002]. They did not explicitly show a bound on the work, though one can trivially deparallelize the algorithm to find that in two dimensions, it runs in near-linear work, albeit with large polylogarithmic factors. Miller [Mil04] proved the first practical subquadratic sequential time bound on a variant of Delaunay refinement, with time essentially $O((n + m) \lg L/s)$ on PLC input in two dimensions. Around this time, Üngör showed that by choosing a different point in the circumball, one that he termed the *off-center*, he could produce a somewhat smaller mesh in practice than what Delaunay refinement can offer [Üng04]. The new point has the advantage that computing it is a more local operation: its coordinates depend only on the shortest edge, unless the triangle is of almost good radius/edge quality. The locality allowed Spielman, Teng, and Üngör to shave one logarithmic factor off their analysis [STÜ04]. More excitingly, it also allowed Har-Peled and Üngör to use a quadtree for point location and off-centers for Steiner points to choose a set of points whose Delaunay triangulation forms a quality radius/edge mesh in $O(n \lg n + m)$ time [HPÜ05]. Finally, using the Delaunay triangulation plus a very naive point location structure, Hudson, Miller, and Phillips developed the first optimal-time $O(n \lg L/s + m)$ Delaunay meshing algorithm that handles PLC input (also, the first subquadratic space algorithm in three or higher dimensions) [HMP06]; this quickly led to the development of a parallel algorithm in $O(\lg L/s)$ rounds of parallel depth $O(\lg m)$ each, with no additional work [HMP07b].

Various other meshing techniques have been proposed. Longest edge bisection chooses a Steiner point for a bad-quality triangle by bisecting the longest edge, rather than inserting the circumcenter. Adding a number of details, it appears that this technique is soon to be proven correct and size-conforming in two dimensions. Advancing front methods start at the boundary of the PLC features and build triangles or tetrahedra from there out toward the boundary of the domain. These methods typically face some difficulties on complicated geometries, though they are not insurmountable. The Har-Peled and Üngör technique is reminiscent of an advancing-front technique in that the off-centers it inserts are spawned from regions with small local feature size and grow outward to regions with larger local feature size. Labelle and Shewchuk recently showed how to mesh an implicitly-defined surface, rather than a PLC, in such a way as to obtain strong guarantees on the dihedral angles [LS07]. Given a poor-quality mesh, a huge number of mesh optimization passes have been proposed that move the mesh vertices around in hopes of improving the quality of the elements: at the International Meshing Roundtable, there are normally several sessions each for tetrahedral and hexahedral mesh optimization. Many are guaranteed never to reduce mesh quality, and most are guaranteed not to insert or delete any vertices. Typically, they act locally, which implies that it should be dynamically stable to run a few passes of a mesh optimizer after running my algorithms.

Shewchuk implemented Ruppert's algorithm for two dimensions [She96], and his own in three dimensions [She98b]. Triangle has been quite successful. Pyramid has not yet been publicly released. Mitchell and Vavasis implemented QMG [MV00], also known as Qmesh, which is based on the octtree algorithm; this code is sadly no longer supported. Hang Si has implemented Delaunay refinement algorithms along with substantial engineering to handle real-world inputs, and packaged it as TetGen [Si06].

Commercially, the most successful codes are based on TetMesh. This is a major piece of software encompassing a huge number of heuristics and mesh optimization passes which, in practice, appear to often produce small and good quality meshes. TetMesh lacks theoretical guarantees; when one meshing heuristic fails, the advice is to change which heuristic to use. Many in the scientific computing world, especially in National Labs run by the US Department of Energy, prefer to use so-called hexahedral meshes, which use distorted cubes as their basic element. Producing hexahedral meshes appears to be very difficult both from a theoretical and a practical standpoint.

Kinetic meshing has attracted some attention. There are two main approaches taken in the literature. The easiest is to remesh from scratch [KFCO06, BWHT07]. Unfortunately, it is of course quite wasteful to entirely throw away an almost good mesh. Worse, the two meshes may differ in every triangle, causing reinterpolation error when copying values from the old mesh to the new. Another approach is to locally remesh [CCM$^+$04, MBF04]. It is easy to implement locally improving the quality. However, to be size-conforming (and size-optimal), we must also *coarsen* the mesh [MTT99, LTU99b]. Sadly, it is unclear how to make mesh coarsening a local operation: coarsening runs in $O(m \lg m)$ time, which is asymptotically more expensive than remeshing, although in practice is much faster.

Dynamic meshing is a much less well-trodden field. Topological changes during kinetic meshing are typically handled in an *ad hoc* manner, and not seen as dynamic changes. Chew has mentioned in personal communication that in a fracture simulation project with which he collaborated, many of the mesh updates were exactly dynamic updates as I have described; to those, their technique was to remesh from scratch. Nienhuys and van der Stappen describe a technique to remesh locally to simulate the new surfaces caused by a scalpel cut through simulated tissue; they remesh locally and add a few new points [NvdS04]. Experimentally, they claim interesting results; unfortunately, they theoretically prove neither runtime nor correctness. Coll, Guerrieri, and Sellarès described a dynamic remeshing algorithm based on mesh optimization [CGS06]. Sadly, they did not analyze the runtime of their technique, though they did prove its correctness.

1.7.1 Relations to prior work

The SVR algorithm is something of a hybrid of quad-tree and Delaunay refinement techniques. Like the naive quadtree algorithm, SVR always maintains a quality mesh (with, therefore, low degree) and refines top-down, only at the last step recovering the input. Like a Delaunay refinement approach, SVR maintains the Delaunay triangulation of the set of points it has thus far processed, and uses circumballs of simplices as its fundamental objects. The code is in the tradition of Triangle, Pyramid, TetGen, and QMG: cross-platform codes implementing provably good algorithms, and cost-free for research use.

My dynamically stable mesher has a longer pedigree, even ignoring proof techniques. It directly uses a quadtree, and therefore is a quadtree algorithm. Borrowing the idea of Har-Peled and Üngör, however, the quadtree points are never inserted the output, and instead it is only used for point location in lieu of maintaining a Delaunay mesh. The gap balls with which I define legal insertions are from Talmor and her thesis work (with Miller *et al.*) on ball-packing and Delaunay refinement. Mine is a parallel algorithm, though I use the parallelism only in order to get good dynamic stability; it fits under the paradigm first developed by Spielman *et al* [STÜ07, STÜ04]. Finally, the encroachment and yielding rules of Chapter 6 are from SVR. As with the algorithm of Har-Peled and Üngör, a claim could also be made that mine is an advancing front algorithm, though it was not conceived as such.

The general technique I adopt is to use a graded, size-conforming quadtree as a point location structure to choose Steiner points. A number of other dynamic point location structures have been developed, some of them based on quadtrees. In particular, Eppstein, Goodrich, and Sun show how to dynamically maintain what they term a *skip quadtree* under point insertions and deletions [EGS05], work that shares a surprising number of keywords with this work. Because of the highly restrained way in which I use the quadtree I build, it can answer the relevant range queries in $O(1)$ time (see Lemma 4.3.3), and can support insertions of new Steiner points also in constant time. It is unclear how to use skip quadtrees to support these operations without paying an additional $O(\lg n)$ term in the runtime; it is equally unclear how to represent higher-dimensional PLC features in a skip quadtree. My algorithms run in two phases: first they build the point location structure, then they query the structure and build the mesh. The runtime (both static and dynamic) is at the moment dominated by the first phase. A faster point location structure may speed it up in practice. However, any replacement structure must take care to keep the query cost in constant time: an additional logarithmic term on queries would make the time of the second phase dominate the asymptotics.

Those who have witnessed the slow development of dynamic convex hull algorithms may be surprised by my claims of having discovered a relatively simple algorithm that outputs a Delaunay triangulation (note that the Delaunay problem in dimension d is equivalent to the convex hull problem in dimension $d+1$). The dynamic Delaunay triangulation problem is famously difficult, because the insertion of a single point can cause linear change in the output triangulation. However, examples of this behaviour require maintaining a mesh with very skinny triangles. In a quality mesh, no skinny triangle is present in the output. Indeed, the algorithms I present never represent a very skinny triangle even in intermediate stages of the algorithm.

Chapter 2

The SVR algorithm

The first result of this thesis is an implementation of the Sparse Voronoi Refinement (SVR) algorithm, which is the first optimal-time Delaunay refinement algorithm. This chapter reviews the previously published algorithm [HMP06] including sketches of the proofs to clarify the degrees of freedom we can work with in the implementation. The main new content of this chapter are a longer description of the intuition behind SVR and the problems faced by traditional Delaunay refinement methods; and a more exhaustive algorithm listing that has previously appeared.

2.1 Traditional Delaunay refinement

Traditional Delaunay refinement algorithms first construct a *conforming* Delaunay triangulation of the input (or a *constrained Delaunay* mesh), which respects the input but has elements of arbitrarily bad quality. Next, the algorithm iteratively finds a simplex of bad radius/edge condition, and removes it from the mesh by calling SPLIT. A simplex appears in the Delaunay triangulation if and only if its circumball is empty of any other points, so by inserting any point in the circumball and recomputing the Delaunay triangulation will remove the bad-quality simplex. Chew [Che89, Che97] showed that a procedure that inserts the center of the circumball will terminate; Ruppert [Rup95] showed it would terminate with a mesh of nearly optimal size (within a constant factor), while Shewchuk [She98b] first extended Ruppert's technique to three dimensions.

To handle features, the algorithms will snap candidate circumcenters to the boundaries as the need arises. A simplex of a PLC feature will appear in the mesh if and only if it has some circumscribing ball that is empty of any mesh vertices; in particular, it will appear if

TRADITIONALDELAUNAYREFINEMENT($\mathcal{X} \subset \mathbb{R}^d$, ρ)

Let M be a conforming Delaunay triangulation of $\mathcal{X}$
$Q \leftarrow$ the set of all bad-quality Delaunay simplices of M
while Q is non-empty **do**
    Let $s \leftarrow \text{TOP}(Q)$
    if s is no longer in the mesh, **skip**
    SPLIT(s)
end while

SPLIT(s: a simplex of dimension i)

Let $p \leftarrow \text{circumcenter}(s)$
if p encroaches a feature's simplex s' of dimension $i' < i$ **then**
    Yield: SPLIT(s')
else
    Perform a Delaunay insertion of p into M, creating a vertex v
    for each simplex $s \in \text{link}(v)$ **do**
        if s has bad radius/edge quality, add s to Q
    end for
end if

Figure 2.1: The traditional Delaunay refinement algorithm, due to Chew [Che89], Ruppert [Rup95], and Shewchuk [She98b], and refined by many others. The algorithm first computes a conforming or constrained Delaunay triangulation of the input (for simplicity, I describe the conforming Delaunay case here), then iteratively improves it, taking care that the triangulation continues to conform to the input. In three dimensions, this is not always possible, which requires an auxiliary structure to maintain unresolved boundary facets [She98b, MPW02].

the smallest-radius circumscribing ball is empty. Though this is not a necessary condition, it is relatively easy to describe and analyze. Accordingly, it is common in the literature to define the circumball of a lower-dimensional simplex as that (unique) smallest-radius circumscribing ball. To maintain the invariant that the circumball of a lower-dimensional simplex is empty, when inserting a triangle's circumcenter would violate the invariant, traditional refinement (and also SVR) will *yield* and insert the segment's midpoint first. In higher dimension, this procedure may recursively yield from circumcenters of full-dimensional simplices to $(d-1)$-dimensional ones down to segments (1-dimensional simplices).

The runtime of traditional Delaunay refinement has been difficult to analyze. In two dimensions, it is clear that each insertion can be made to run in time linear in the size of the mesh at that point, which gives us an uninteresting time bound of $O(m^2)$. When the order of operations is left arbitrary as in Chew's, Ruppert's, and Shewchuk's algorithms, examples (see Figure 2.2) exist where the output size is $\Theta(n)$ — just under $3n$, using Triangle —, yet each vertex insertion modifies a linear number of triangles, giving us a lower bound of $\Omega(n^2)$. Miller worked around this difficulty by proposing to order the work queue Q largest circumradius first. This produces an optimal-time $O(n \lg n + m)$ algorithm when there are no segment features (because the priority queue can be approximate), and one that runs in a logarithmic factor slower than optimal when there are features (because the priority queue must then be strict). Conversely, Üngör recognized that one gets a smaller output size by working on the shortest edge first, in which case the worst-case example applies. Therefore, practical codes can deterministically be made to take quadratic time on admittedly non-practical inputs.

In three or higher dimensions, the situation is far simpler. It is well known that there are point sets where the Delaunay triangulation has size $\Theta(n^{\lceil n/2 \rceil})$ [McM70]. In particular, the moment curve, where point p_i has coordinates $\langle i, i^2, \ldots, i^d \rangle$, achieves this bound. For an example that comes up in application, consider a galaxy with a central quasar. Almost all the mass of the galaxy except the quasar will be concentrated almost on a circle, while the quasar will emit jets perpendicular to the galactic plane. We can model this as a set of n_1 points on the circle $x^2 + y^2 = 1$ with $z = 0$, and another set of n_2 points on the line $x = y = 0$. See Figure 2.3. The Delaunay triangulation of this input has exactly $n_1(n_2 - 1)$ tetrahedra (which is $\Theta(n^2)$ when $n_1 = n_2$): consider a consecutive pair of points on the line, and another consecutive pair of points on the circle. We can expand a ball out with on its surface the two points on the line by moving normal to the line. In this plane, we can choose to move in the direction of the midpoint between the two points on the circle. Clearly, the ball will never intersect any other points on the line, since it is being grown orthogonal to it. Equally clearly, the first points on the circle that ball will intersect are the pair that we have chosen. This witnesses that the tetrahedron formed by the four chosen points is Delaunay. By symmetry, we have proven that *every* successive pair on the circle forms a Delaunay triangle with *every* successive pair on the line. On the other hand, after refining this pathological example, we can prove [HMP07a] that the output size (both number of vertices and number of tetrahedra) will be only linear in the number of vertices in the input. This prediction is borne out by experiments using the SVR implementation, which show that the answer is $m = 42n$ tetrahedra.

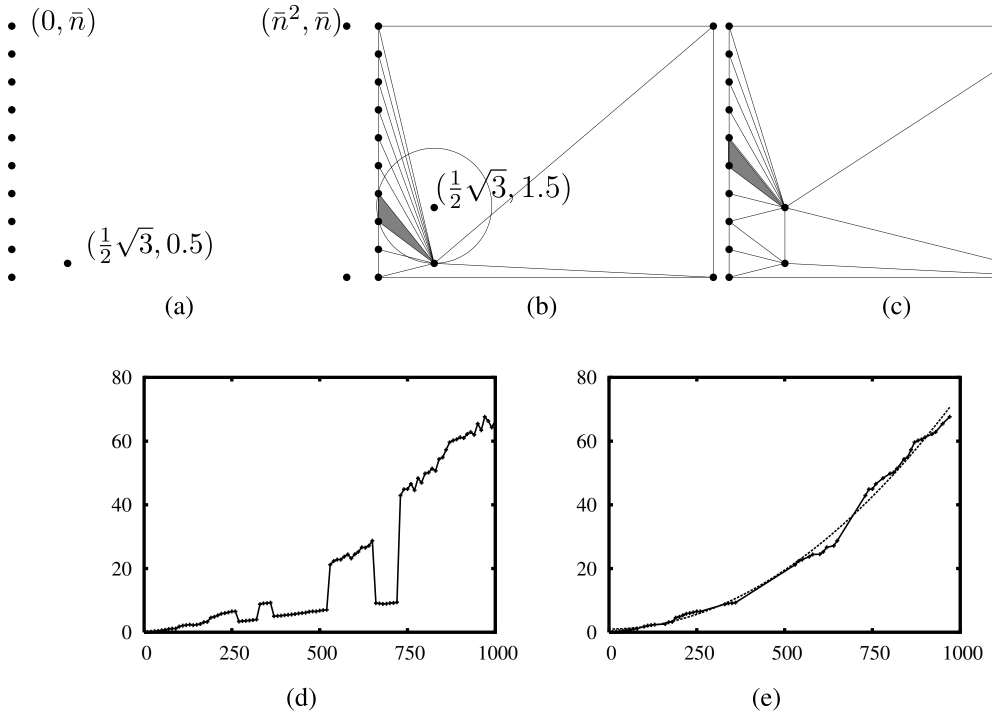


Figure 2.2: An example, modified from one of Jernej Barbic (personal communication), that can require Ruppert's algorithm to run in quadratic time when processing triangles in order of shortest edge first (which is what Triangle approximates). Ruppert reportedly previously developed an equivalent example, also unpublished. (a) Let $\bar{n} = n - 4$. We place $n-3$ points along the y axis so that the separation between successive points is about 1 but grows slightly, by a tiny ϵ (not pictured). We place a point at the off-center of the bottom-most edge, so that it subtends a quality triangle. To complete the area, we place two more points far enough that every point on the y axis has a Delaunay edge with the off-center point. (b) When choosing to process triangles shortest edge first, the shaded triangle will be chosen, which inserts a new vertex directly above the old off-center. Note that this new point is the off-center of the shaded triangle, not just its circumcenter. (c) Inserting the chosen point requires updating all but a small number of triangles, and leads us to essentially the same situation as before, proving the quadratic runtime. (d) Runtime of Triangle, version 1.5, running on an unloaded Mac Pro 3 GHz Intel Core2 Duo. I presume the runtime is highly variable because Triangle only approximates the smallest-first order required to exhibit the pathology. (e) The upper envelope of the runtimes, compared to a $\Theta(n^2)$ fit.

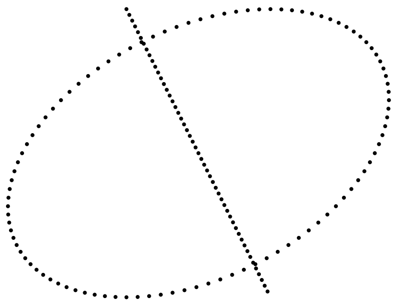


Figure 2.3: Pathological example with 50 points on a line and 100 on a circle.

2.2 SVR intuition

The difficulty of analyzing, even in two dimensions, the runtime of traditional Delaunay refinement comes from the fact that in the intermediate meshes, vertices have almost arbitrary degree — up to linear. Indeed, the problem in three dimensions is that every vertex has linear degree. However, in the final output, every vertex has bounded degree: at most 12 in two dimensions when the angle bound is $30°$ (easily computed by dividing $360°$ into $30°$), and still a constant function of ρ in higher dimension [MTTW95, Tal97]. The last statement suggests that one way to bound the degree, a topological quantity that affects runtime, is to bound the quality, a geometric quantity perhaps easier to analyze. SVR is thus built around the following invariant: *every intermediate mesh has reasonable radius/edge quality* — this implies that the mesh is always *sparse*. Achieving the user-demanded bound ρ is too demanding and unnecessary, so we allow intermediate meshes to degrade to some ρ' that is a constant function of ρ, of the dimension d, and of other user-specified parameters independent of the input configuration.

We can thus describe SVR as an algorithm that alternates between two goals: first and foremost, it must maintain quality. Thus, the initial state is to hold a good-quality triangulation of a bounding box around the input, which does not resolve the input at all. Whenever the mesh contains an element of bad quality (worse than ρ), SVR states that we must process that element first, as if running traditional Delaunay refinement (albeit with a slightly modified SPLIT procedure). Only once the mesh is again of good quality will SVR try to resolve the input. A simplex that contains an input point in its circumball is clearly not a simplex that will be output, so SVR will also SPLIT that simplex.

Recall that to eliminate a simplex, we need only insert some point within its circumball.

Traditional methods insert the circumcenter. However, in SVR, not all of the input is included in the mesh; the circumcenter may be very close to an input point, which would create an artificially small feature. Conversely, if indeed there is an uninserted input point within the circumball, SVR may as well use it instead of the circumcenter — unless the input point lies too near a current mesh vertex (namely, a previously-inserted input point). To preferentially use an uninserted input point, but ensure that it is far from any existing mesh vertex, SVR searches for a point in the shrunken ball $B(c, kr)$ where $B(c, r)$ is the circumball of the given simplex, and k is a positive constant less than 1. See the next section (Figures 2.5 through 2.8) for a full algorithm listing and description.

Proofs are much simplified by using the *Voronoi* diagram as the intermediate mesh, rather than using the Delaunay triangulation; thence the name of our algorithm. The two are duals of each other, so a data structure for one is a data structure for the other. The type of the intermediate structure is an implementation detail: its purpose is not to be triangular (or tetrahedral, or Voronoi), but rather to provide for fast range queries when looking for a point in $B(c, kr)$, to provide a mechanism for determining whether the mesh is locally of good quality, and to drive recovery of uninserted points and features. A key philosophy of the design of SVR was that implementation details should not be enshrined within the algorithm or its proofs, in the hopes that this would allow substantial room for constant-factor improvements.

The proof that SVR produces a quality mesh of small size is relatively unenlightening; it follows the same line of argument as Ruppert's original algorithm [Rup95]; see also the size-conformance proofs in the later chapters of this thesis. In essence, the algorithm works because it inserts roughly the same vertices that traditional refinement would insert. The difference is that SVR inserts the vertices in a different order.

SVR's runtime is bounded to be in $O(n \lg L/s + m)$. Fundamentally, there are only two proofs underlying this runtime. First, we prove that the mesh quality never get worse than some ρ' which is a constant function of ρ, k, and the dimension. Individual insertions may degrade the quality of the mesh, but an inductive argument shows that the degradation is limited. In essence, splitting a good-quality element may create mediocre-quality elements, but splitting mediocre elements cannot create bad elements. This implies that the mesh is of bounded degree. The bounded degree in turn implies that most operations are constant time: in particular, inserting a vertex will take only constant time, establishing the $O(m)$ term of the runtime. Second, we show, using a packing argument, that any input point participates in only $O(\lg L/s)$ range queries, and that Steiner points inserted on lower-dimensional features are involved in only $O(1)$ range queries; using a similar argument, we show the same holds for updating the range query structure. Given the bounded-quality assumption, each query is constant time; this provides the $O(n \lg L/s)$

term, and also contributes to the $O(m)$ term. In other words, point location is almost the entire cost of mesh refinement using SVR.

2.3 Algorithm Listing

SVR suffers from some schizophrenia as to whether it is truly a Voronoi-based or a Delaunay-based algorithm. Generally, it is easiest to state the proofs in the Voronoi and the algorithms in the Delaunay. Converting between the two is generally not hard; I use the word 'cell' to be deliberately vague about which I mean, whereas I use the term 'Voronoi cell' or 'Delaunay simplex' in those few times where it is important to be specific.

The fundamental invariant in SVR is that the mesh being maintained in memory always has good quality. To achieve this, we initially only represent part of the input. Points of the input that are not yet inserted are kept in a point location structure related to the mesh; see Section 3.1 for a discussion of a few different such structures; in the present chapter, the structure is deliberately left vague. While the intermediate mesh always has bounded radius/edge ratio (or bounded Voronoi aspect ratio), that quality bound is in general some $\rho' > \rho$. To maintain quality ρ', and to eventually recover the input, the mesher must alternate between ensuring good quality by eliminating skinny cells (those with quality worse than ρ) and eliminating crowded Voronoi cells (those that contain an uninserted point of the input). Priority is given to eliminating skinny cells. For ease of analysis and programming, the crowded and skinny cells are all put onto a queue, whose responsibility it is to properly prioritize the different types of events.

A cell on the queue needs to be either eliminated (if it is a simplex) or shrunken (if it is a Voronoi cell). To shrink a Voronoi cell $V(v)$, we must insert a mesh vertex u such that some points in $V(v)$ will be closer to u than to v; this corresponds exactly to inserting a point in the circumball of one of the Delaunay simplices that has v as a vertex. Therefore, eliminating a Delaunay simplex or shrinking a Voronoi cell are fundamentally the same operation. Inserting a new vertex anywhere within a circumball eliminates the corresponding Delaunay simplex; however, we must take care of two things:

- If the vertex corresponds to a Steiner point (a point not in the input), it must not violate the size conformality requirement by being close to an input feature.
- If the vertex is an input point, it must not violate the intermediate quality guarantee by being close to a mesh vertex.

We defined a procedure that accounts for both these requirements, illustrated in Fig-

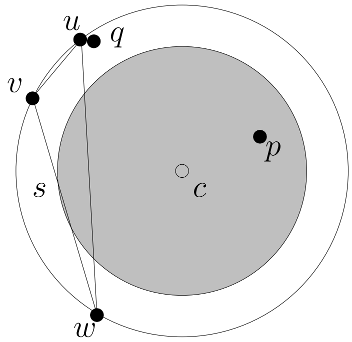


Figure 2.4: If a simplex s is undesirable for some reason, we wish to insert a point in its circumball, the outer circle pictured. Any point in the ball will destroy the simplex, but inserting an uninserted point q very near a current mesh vertex would create a new simplex of arbitrarily bad quality. Therefore, the search is over the shrunken ball of radius kr, shown shaded. If there is an uninserted p as there is in this illustration, it will be inserted; if not, the center c will be inserted.

ure 2.4. Given an undesirable Voronoi cell $V(v)$, we say that the largest-radius Delaunay simplex that has v as a vertex is itself undesirable. Given an undesirable Delaunay simplex, the two most natural points to insert are the circumcenter of the simplex being eliminated, or an input point that lies within the circumball. The rule is described as follows: first, consider the circumcenter. Search out to a radius kr, with $k < 1$, from the circumcenter. Upon finding an input point within that radius, *warp* to the input point and use it instead; if there is no such input point, insert the center. Setting $k = 0$ violates the first condition (we create arbitrarily small features that should not exist); setting $k = 1$ violates the second (we emulate Ruppert/Shewchuk refinement, allowing arbitrarily bad intermediate quality). The proofs of correctness and runtime apply for any $k \in (0, 1)$, which means that choosing k is an engineering question that I relegate to the next chapter.

A critical detail is that we *must*, when processing a crowded Voronoi cell $V(v)$ of a Steiner point, insert an uninserted input point $p \in V(v)$. Intuitively, we know this is safe, because p must be far from v (otherwise, the warping procedure would have chosen p over v); it is therefore desirable from a constant-factor standpoint to blindly insert p. The fact that it is required for termination is less obvious, and holds only for some k: essentially, without this rule, we might insert Steiner points in rings bracketing p but never inserting p itself.

The method for implementing the query for whether $B(c, kr)$ contains a point, or whether a cell $V(v)$ is crowded, is purposely left undefined here as an implementation detail; I overview several alternatives in the next chapter.

Handling input features is done by maintaining a mesh M_f for each polytope $f \in \mathcal{X}$.

SVR($\mathcal{X}$, ρ, k)

Create Q, ordered according to COMPARE
INITIALIZE the meshes, the point location structures, and the work queue Q,
while Q is not empty **do**
    $w \leftarrow$ the highest-priority work item
    SPLIT(w)
end while
Output M

Figure 2.5: The SVR main loop.

Skinny cells in low-dimensional meshes are prioritized ahead of skinny cells in the higher-dimensional and the full-dimensional mesh. When a new vertex v appears in the mesh for f, meshes for the polytopes that contain f (i.e. that have f on their boundary) add v to their list of uninserted points. Given a Delaunay simplex $s \in M_f$, consider the circumball of s. If, in a mesh M_{f+} for a higher-dimensional feature f^+, that ball contains no points, and if all the vertices of s appear in M_{f+}, then clearly s is Delaunay and thus appears as a simplex in M_{f+}. SVR maintains the invariant that every mesh M_{f+} *protects* the diametral balls of all the Delaunay simplices of all its lower-dimensional features and keeps those balls empty. To achieve this, when M_{f+} considers adding a mesh vertex v, it first checks whether that vertex lands within a diametral ball b that it is protecting. If indeed v *encroaches* upon the protected ball, then M_{f+} enqueues the simplex corresponding to b, giving it the highest possible priority, and also re-enqueues whatever work it was processing that led it to encroach on a protected ball, with obviously lower priority.

Encroached cells are processed before skinny cells, which are processed before crowded cells. In the event of a tie, events are processed lowest-dimension first (an encroached segment before an encroached triangle, but an encroached tetrahedron before a skinny triangle). It should be noted that we need not explicitly handle low-dimensional crowding: if the ambient-dimensional mesh M decides to insert a point p that crowds a simplex s in low dimension, it will immediately cause encroachment of p on s, which will be resolved by inserting p into the lower-dimensional mesh.

I divide the algorithm into four parts. The main loop initializes, then iterates until the queue is empty (Figure 2.5). Initialization creates the data structures needed for the algorithm (Figure 2.6). The SPLIT operation handles checking for encroachment and warping (Figure 2.7). The INSERT operation performs the mesh and point location structure updates when we finally do in fact insert a vertex (Figure 2.8). The work queue is ordered according to COMPARE.

INITIALIZE

Create a bounding box B large enough to ignore boundary effects
Let $M \leftarrow \text{Delaunay}(B)$
For each $f \in \mathcal{X}$, let $M_f \leftarrow \text{Delaunay}(f)$
Initialize the point location structure P with M and $\mathcal{X}$
For each $f \in \mathcal{X}$, initialize P_f with M_f, f, and subpolytopes of f
for every vertex $v \in M$ **do**
    If v is crowded, add a work item CROWDED(v)
end for
for every simplex s in every mesh M_f **do**
    If s has radius/edge quality worse than ρ, add SKINNY(s)
    If s is encroached in a higher-dimensional mesh, add ENCROACHED(s)
end for

Figure 2.6: Initialization: create the meshes for each feature and for ambient space. Create point location structures. Seed the work queue.

```
SPLIT(w)
if w = CROWDED(v) and v has containing dimension d then
    Choose an arbitrary point p that crowds v
    Find a simplex s around v whose circumball includes p
    Call SPLIT(p, s)
    If SPLIT did not insert any points, add w back to the work queue
else if w = CROWDED(v) and v has containing dimension i < d then
    SPLIT the simplex around v with largest radius
else {w refers to a simplex s in M_f}
    if w also refers to a point p then
        let c ← p
    else
        Compute the circumcenter c of s, with radius r
        if P_f knows about a point p in the ball B(c, kr) then
            Warp: c ← p
        end if
    end if
    if c encroaches upon a simplex s' of a lower-dimensional feature f' then
        Add w back to the work queue
        Yield: Add ENCROACHED(s') to the work queue
    else
        INSERT(c, s)
    end if
end if
```

Figure 2.7: Handling a work item, which will inevitably lead to splitting a simplex by inserting a point in its circumball. Splitting may warp to an input point, or it may temporarily yield to one or more lower-dimensional simplices it encroaches upon.

INSERT(p, s)

Let M_s be the mesh of which s is an element, and f_s the corresponding PLC polytope.
Perform a Delaunay insertion of p into M_s
Let (K, C) be the sets of simplices resp. destroyed and created by p
Update P_{f_s} accordingly
for each feature f_s^+ that has f_s as a subpolytope **do**
    Update $P_{f_s^+}$ to ignore K and take account of C and p
end for
if p has containing dimension $\dim(f_s) < d$ **then**
    Look up in P the vertex v whose Voronoi cell contains p
    Add CROWDED(v) to the work queue
end if

Figure 2.8: Inserting a point into a mesh in SVR. This requires updating the appropriate mesh, but also updating the set of protected balls and uninserted points that higher-dimensional meshes maintain via the point location structures. Also, when points are created, we mark a corresponding vertex of the top-dimensional mesh as being crowded.

COMPARE(w_1, w_2)

ENCROACHED comes first. If both are encroached, lower dimension comes first.
SKINNY comes first; if both are skinny, lower dimension comes first.
CROWDED comes last.
Break ties arbitrarily.

Figure 2.9: Items on the work queue are ordered according to COMPARE. Two work items that match both on reason (ENCROACHED, SKINNY, or CROWDED) and on dimension are ordered arbitrarily. This ordering can be computed in constant time.

Chapter 3

A Practical Implementation of SVR

Having reviewed the algorithm and proof sketches, I now report on a C++ implementation of the static algorithm, and some significant constant-factor runtime improvements we developed for the implementation. This chapter is expanded from a report presented at the International Meshing Roundtable [AHMP07].

3.1 Point location structure

During SPLIT, two types of point location query must be performed: first, we ask whether there is an uninserted point in the warp region, as described in Figure 2.4. Next, we ask whether the point chosen to be inserted encroaches upon any lower-dimensional protected ball. Updating these structures is left to INSERT and INITIALIZE. Recall that SVR runs in $O(n \lg L/s + m)$ time. The first term is due entirely to these range queries and point location operations; the second term also includes some such costs. This suggests that these operations will drive the runtime, and indeed we were able to extract substantial speedups by focusing on them. This section overviews the development of a data that in practice offers great advantage over the more naive methods described in the theoretical papers. To date, the best technique is only applied to the uninserted points; the code to handle input features is still young.

In order to retain the asymptotic guarantees, the query structure must be able to respond to a range query looking for an uninserted point in an empty ball $B(c, kr)$ in constant time, plus the time to consider the points themselves. During a range query, every uninserted point p that is considered (and potentially rejected) must be at distance $||cp|| \in O(r)$. The same holds also for updates: upon inserting a vertex v, the structure must only perform reads or writes relating points p such that $||vp|| \in O(\mathrm{NN}(v))$. Finally, the analogous statements must be true for checking for encroachment.

By circumball: When we perform the queries, we have in hand a simplex that we are destroying, and even after warping, any point that we insert will be within the circumball of the simplex. Therefore, it is intuitive to maintain points and lower-dimensional protected balls associated with the circumballs of the mesh simplices. Then, to check whether we need to warp, we simply look up the set of points in the circumball of the present simplex and find one that lies in the kr shrunken ball. To check whether the chosen point encroaches, we query each of the protected balls that intersect the circumball. The simplicity of this approach is obvious. There are some significant disadvantages unfortunately: an uninserted point may be present in the circumballs of several simplices, which duplicates the storage. It also requires duplicate elimination when updating, to make sure that the same point is not added repeatedly to a single circumball. Some care must be taken to make sure that points are indeed assigned to the circumballs in which they lie: it is best to use a robust INSPHERE predicate for this.

By simplex: The traditional point location structure for triangulations, dating back at least to Kirkpatrick [Kir83], uses the triangles to drive point location. That is, points are kept in the triangles. To test whether a point lies within the query region, is somewhat more complicated than above: we iterate over the set of triangles that intersects the query, and in each triangle, iterate over all the points it contains. Conversely, updates are cheaper: we simply reallocate the points in all the triangles that were destroyed. Each point appears exactly once, and will lie in exactly one of the new triangles (if it lies on a segment, break ties arbitrarily). During this reassignment, it is critical to ensure that numerical errors do not cause a point to fail to be assigned to any triangle. I used robust ORIENT3D predicates for this. In my admittedly inexaustive experiments, the savings in memory and reassignment time compared to using circumballs were quite substantial. Recognize that there is a tradeoff between reassignment time and query time: storing in the circumballs gives for faster queries, but slower reassignment, as compared to simplices. However, every query leads to the destruction of a large number of simplices (on average six in two dimensions, about 18 in three dimensions in our experiments). In other words, substantially more reassignment is performed than queries.

By Voronoi: Determining whether a point p lies within a simplex, or determining whether it lies within the circumball of a simplex, is an expensive operation: both involve solving the determinant of a dense rank $d+2$ matrix. Using the dual Voronoi diagram gives a substantially cheaper update: when a new mesh vertex is inserted, uninserted points in the affected Voronoi cells (that is, the cells of the vertices that in the Delaunay triangulation form the link of the new vertex) need only be tested to see whether the new vertex is closer

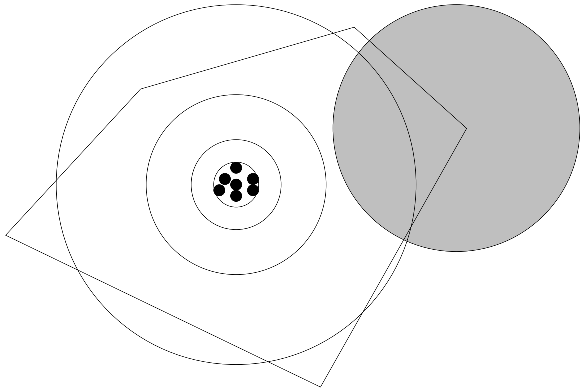


Figure 3.1: Illustration of the intuition for why using the Voronoi with concentric shells can give a dramatic benefit over naively using the Voronoi diagram. Any concentric shell that the query ball does not intersect need not be searched; this allows SVR to avoid checking the dense cluster of points.

or further than the vertex that currently owns them. This is also an operation that requires far less numerical robustness: being off by a small ϵ may position the point in the wrong Voronoi cell, but will never entirely lose track of the point; and queries are essentially unaffected by such small errors, since they will search both Voronoi cells in any case. Queries are more expensive even than the simplex-based structure, for the simple reason that the set of Voronoi cells that covers the query disk also covers a much larger area outside the query disk than do the equivalent simplices. In implementation, there was no immediate benefit to using the Voronoi diagram for point location instead of using the simplices.

The definition of crowding needs to be changed when using Voronoi cells: now, a Voronoi cell is crowded, rather than a simplex. When a Voronoi cell of a Steiner vertex is crowded, it is guaranteed that any uninserted input point is far from the vertex (or it would have warped), so we can blindly insert any one of the vertices. When it is a Voronoi cell of an input point, the SVR algorithm specifies to insert the farthest Voronoi node. Since the code now uses Voronoi cells for point location, this is what I described in Figure 2.7.

One pitfall which befell me was that vertices have a long lifetime, unlike simplices; but they only lose vertices over that lifetime. Therefore, the structure that stores the set of points in the Voronoi cell must shrink over time. In particular, the standard `std::vector` class does not shrink, which then causes the point location structures to require asymptotically more space.

By Voronoi, in concentric shells: Consider a dense cluster of points, as in Figure 3.1.

Early in the run of SVR, some simplex will warp to one of the points, call it v, when choosing a point to insert, and all the remaining points will be assigned to v. Those points will then be checked for relocation $O(\lg L/s)$ times each, and will each time reply that they are closer to v than to the new vertex. A fix for this is to assign the points in concentric shells around v, with the radius of each being a constant factor larger than the previous. When a new vertex v' appears, we know not to check any point in any concentric shell that is entirely closer to v than to v'. In other words, points will only be checked for reassignment when the local mesh size has shrunk to be commensurate with their distance from v. Similarly, queries inspect all the points in each concentric shell, from the outermost until reaching a concentric shell too close to the vertex for the query to give a positive answer. Asymptotically, this is no change: constructing the concentric shells takes $O(\lg L/s)$ per input point to find the appropriate shell, using a structure very similar to the γ-bucketed priority queue described earlier (Section 1.5). However, the operations are cheaper, and it allows the memory hierarchy to safely evict from cache the geometric coordinates of the uninserted points. During development, switching to concentric shells immediately halved the runtime as compared to using the Voronoi diagram naively, or using simplices.

Despite the saluatory effect of concentric shells with respect to the number of point relocations that are performed, relocations are still a large expense. On an IEEE floating point machine given a point p, we can find the index of the appropriate concentric shell around v by computing the square of the distance, then right-shifting by 53 bits to retain only the exponent of the squared distance. Using this directly means that the concentric shells grow by a factor $\sqrt{2}$; I did not experiment with changing the growth factor by modifying the exponent. Note also that the exponent is an integer rather than a double, which reduces the memory overhead of each concentric shell. Such low-level tricks as this one, which only avoids computing a square root and a few divisions, have a surprisingly important effect.

Other ideas: In some circumstances, no point location needs to be done at all. In particular, if there is no remaining uninserted input, we can short-cut the query. On the bunny dataset, this occurs in the final 10% of point insertions; on the pathological input, this is closer to 30% of insertions (see Section 3.4 for the description of the experimental inputs). Detecting this situation is cheap: simply a mesh-wide counter.

The concentric shells avoid wasting time negatively answering range queries with respect to points close to a mesh vertex. Another weakness in Voronoi-based point location, which the concentric shells do not improve, is that a query originating from one side of the Voronoi cell will touch points on the opposite side of the cell. Worse, if the Voronoi cell is of mediocre aspect ratio, and the query comes from a short side, the outermost shells will still be inspected. In other words, we do not yet have a good solution to point location for points far from a vertex. Possibilities include using a hybrid: Voronoi shells near the vertices, simplices in open space. Another possibility is to use the Voronoi cells of both mesh vertices and element circumcenters. These remain future work. Given the success of the prior speedups, Amdahl's Law currently limits the effect of any further improvement. However, I optimistically claim that point location will likely become the limiting factor again as SVR proceeds to tackle increasingly large problems.

3.2 Design and implementation

The goal of the SVR codebase, beyond merely implementing the algorithm, is to provide a useful library for writing arbitrary-dimensional meshing code, and to provide a richer API than is often provided in meshing codes for accessing the mesh (in particular, for modifying it). For ease of interface with other codes, the SVR implementation is in C++. For efficiency in the face of generality, I heavily use the parametric polymorphism of the C++ template mechanism, which modern compilers can compile reasonably well; I avoid polymorphism based on dynamic dispatch due to its runtime cost. The code can be roughly divided into four modules: (1) support classes and I/O. (2) The mesh topology. (3) The mesh geometry, and geometric predicates. (4) The meshing algorithm itself.

Given that the code is in C++, I made extensive use of the Standard Template Library (STL) and of the Boost libraries. These provide for type-safe and reasonably efficient data structures. However, "reasonably" was not always fast enough; as the need arose, I replaced certain components with my own. In particular, I implemented a memory allocation routine based on a global freelist, for fast allocation of small items (up to and including the size of a simplex). I also created a hash table that for some uses is substantially faster than the GCC implementation of the standard hash table, and other similar structures. When iterating over a set maintained as a hash table, no order is explicit. The default STL implementation is to order them in an order related to their hash values, which in turn are often based on pointer values. Given that any change in memory allocation may change all pointers subsequently allocated, this makes debugging and performance profiling difficult: a seemingly innocent `printf` may cause quite different behaviour. I therefore make a concession in the direction of debuggability here: I never iterate over hash tables, but rather I simultaneously maintain a linked list for any set over which iteration will be required.

The topological structure is a standard pointer-based simplicial complex, with $d + 1$ vertices and $d+1$ neighbour pointers per mesh element. It takes template arguments for the dimension d, the type of the vertices, and for optional user data to attach to each simplex. In an attempt to avoid memory errors, simplices are reference counted. This means it is safe to hold a handle to a simplex even after destroying it during meshing; conversely, it means that accidentally holding a handle to a simplex leads to memory overuse. Access to the simplicial complex is via two mechanisms: a generic depth- (or breadth-) first search routine which takes a closure as an argument, which is the one client code usually uses; and safe low-level access to navigation routines for specialized use. Simplices are kept in a fixed topological orientation. All access is properly oriented, which avoids sign-flipping bugs. I implement the "switch" access routines defined by Brisson [Bri93]. A handle to a simplex in three dimensions can be used to denotes a vertex, edge, triangle, and tetrahedron. A 0-switch changes which vertex is being denoted, but leaves fixed the edge, triangle, and tetrahedron. A 1-switch changes only the edge, etc. A single switch operation flips the orientation of the handle, so I require that switch operations occur in pairs. The topological structure implements insertion of a vertex by deleting a given set of simplices, and replacing them with the star that has the new vertex at its apex, and a simplex for each of the exterior faces of the deleted set. I do not implement flips or deletion of vertices.

I define the geometric structures on top of the topology. In this layer, vertices must have coordinates. A Delaunay triangulation has two, possibly distinct, associated dimensions: the *ambient* dimension of the points, and the dimension of the object it is meshing (a polygon in three dimensions, for example). I call the latter the *topological* dimension. For triangulations with topological dimension less than the ambient, we compute a basis for the plane in which the triangulation lies using repeated squaring of the moment matrix to compute the largest eigenvectors. While points are maintained in ambient space, the INSPHERE test used while updating the Delaunay triangulation is defined in the basis, which, using Shewchuk's predicates [She97a], allows for *consistent* predicate calculations. Circumcenter computations are done directly in ambient space, in the hope of avoiding accumulating error from projecting into the basis of the plane, computing the circumcenter, then projecting back to the ambient basis.

The library includes a number of geometric primitives — points, matrices, circumcenter computation, and geometric predicates such as `incircle` or `orient3d`. These are largely based on suggestions or published work by Shewchuk [She96, She99]. As a runtime optimization — one with major effect — each simplex also maintains its circumcenter and radius. On a 32-bit machine, this data approximately doubles the in-memory size of a simplex; in future work, I intend to implement a way to use an LRU cache to reduce the overhead.

The API of the geometric structures is arbitrary-dimensional. However, for the most part, they currently only work in one, two, or three dimensions, and will give a compiler error in four or higher dimension. Circumcenter computation will work in higher dimension, but it is currently implemented using LAPACK. I replaced that code with special-dimensional code because the data marshalling costs to communicate between my code and LAPACK dwarfed the cost of the numerical computation itself, by a factor of almost 4:1. LAPACK assumes that matrices will be large, whereas these geometric primitives typically involve at the most a 5×5 matrix. Fast geometric predicates could be produced in higher dimension, albeit at high labour cost; an automatic technique [NBH01] exists but is at the moment unimplemented due to bitrot. For working but slow code, one could use one of the many generators of exact predicates, such as those bundled with CGAL.

The mesh algorithms and structures are described in great detail in other sections. The API of the mesh algorithms takes in a description of a PLC and a set of constants regulating the meshing (this is for development, as the default settings are acceptable), and returns an instance of a Delaunay triangulation. The user then has access to the same APIs I used to implement the refinement algorithms.

3.3 Numerical robustness

SVR requires accurately making a number of numerical computations, particularly insphere tests and computing the circumcenter of a simplex. For a sphere defined by a full-dimensional simplex, Shewchuk has defined a fast but exact numerical predicate and released the code into the public domain [She97a]. However, this code does not compute, for example, whether a point p lies within a ball $B(c, kr)$ where c is defined based on a triangle in space, and r is the circumradius of that triangle, reduced by a factor k. Nanevski *et al*. extended Shewchuk's work, and designed and implemented an automatic method of generating new fast but exact predicates [NBH01]. Sadly, the code no longer works due to bit-rot. Various other authors have produced support and compilers for exact predicates, which generally produce correct but very slow code.

A minor issue is that Shewchuk's predicates assume that computations will not underflow into denormalized numbers (and that neither will they overflow into infinity). In experiments, I started seeing underflow issues occurring when points were separated by a distance of about 10^{-5}; simply scaling the entire space fixed this problem. It appears that the problem of generating fast numerical predicates for numbers with small exponents is not yet solved in a downloadable package.

Most of the predicates used by SVR are robust by nature. The key invariant of SVR, that elements in intermediate meshes have good quality, ensures that every computation will return unambiguous results in two dimensions; indeed, profiling information indicates that Shewchuk's predicates almost always give the correct answer even only using the non-robust calculation, except when computing the initial triangulation, which has a number of cocircularities.

Unfortunately, slivers bypass that guarantee in three dimensions by being of bad aspect ratio but being present in our intermediate meshes. The standard computation of the circumcenter involves solving a linear system $Ax = b$ where $\det A$ is the volume of the simplex. A sliver has (almost) zero volume, which makes this a degenerate system. Similarly, the in-sphere predicate on a sliver tries to compute the determinant of a matrix whose rows, on a perfectly flat sliver, are linearly dependent. Consequently, even exact computation returns that every point in space is on the surface of the circumsphere of the sliver. On point clouds examples, SVR does not appear to suffer from these sliver-based issues. Sadly, they arise with some frequency on PLC inputs. It remains open how to resolve this issue.

3.4 Experiment inputs

For the runtime experiments on point-cloud, I used three classes of input. The first is the set of points that define the Stanford Bunny. This is a set of 34890 points, sampled roughly uniformly from a smooth surface; it is a standard example in computer graphics and in the meshing community. Amenta, Attali, and Devillers [AAD07] recently showed that given a uniform sample in three dimensions from a manifold in two dimension, the Delaunay triangulation of this has linear size. Therefore, one should expect that SVR would have only limited benefit over prior Delaunay refinement algorithms in this case.

The second example is a pathological case, described in Figure 2.3. I evenly sample $n/2$ points on a vertical line, and $n/2$ points on a circle centered on the line, and place the assemblage in a sufficiently large bounding box. This example was one of the driving examples for the development of SVR: any technique that starts by computing the Delaunay triangulation of the input is doomed to build about $n^2/4$ tetrahedra just on the initialization step here, whereas SVR only takes space commensurate with the final size of the mesh, which in experiments is $42n$ tetrahedra.

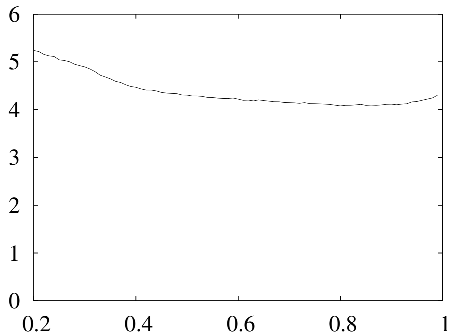
Finally, I use a regular $k \times k \times k$ integer grid. The grid is trivial to mesh: its Delaunay triangulation has radius/edge ratio $1/\sqrt{2}$, so we never insert Steiner points except in the expanded bounding box. As we very recently proved [HMP07a], in the bounding box outside a convex shape, the number of points inserted is linear in the number of points on the surface of the shape, so there are k^3 points in the cube and only $\Theta(k^2)$ Steiner points outside it. The goal of this example was to test the cache performance of the implementations.

In addition to the point-cloud inputs, I ran SVR on some examples of PLC inputs donated by Shewchuk and by Phillips. Some of Shewchuk's examples, and most examples to be freely found online, include triangulated surfaces. These automatically are invalid input for SVR, as they clearly have angles between segments much worse than $90°$, and in fact worse than the $60°$ lower bound that might avoid infinite refinement; indeed, SVR cannot mesh inputs consisting of triangulated surfaces. Most of Shewchuk's examples include angles between faces of slightly less than $90°$. Despite violating the theoretical guarantees, these inputs can be meshed by SVR. Many of the examples include holes in the input polygons that are not specified in the input. I added a hack that tries to detect the holes, which successfully handles some but not all inputs; ideally, one would implement a truly correct hole-detecting routine. The examples by Todd Phillips were engineered to quite precisely match the requirements of SVR as published in the theoretical papers: each facet is convex, has $O(1)$ points on its surface, and has bounded aspect ratio. Segments on the boundary of a facet are not encroached by other vertices of the facet, and all input angles are non-acute.

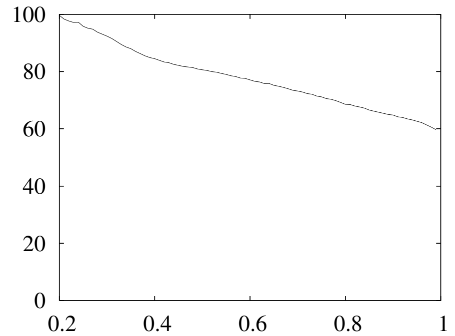
3.5 Parameter settings

There are two main knobs to turn in SVR: the radius/edge quality bound ρ, and the warping parameter k. The proofs show that SVR will have optimal runtime for any constant $k < 1$, and $k^d\rho > \sqrt{2}^{d-1}$. However, the parameter k only affects the algorithm when there are uninserted points. One trick we can employ is to use, internally, some $\rho' > \sqrt{2}^{d-1}/k^d$, to define what makes a skinny simplex. Once all uninserted points have been brought into the mesh, we can then improve the mesh quality to the user-specified ρ by adding a low-priority class of events: any simplex that has quality better than ρ', but worse than ρ, is added to the queue as a MODERATELYSKINNY with priority less than CROWDED. Clearly, if no elements are crowded, every vertex is in the mesh. The SPLIT operation can be left as-is, although the search for a point to warp to is wasted time since there is no point anywhere that it could find.

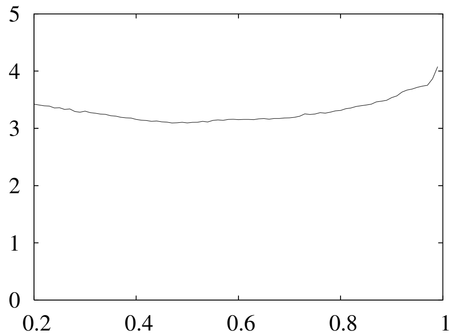
In other words, the user may demand any quality ρ so long as it is strictly greater than two in three dimensions, and any $0 < k < 1$. Users will typically want a mesh with ρ as small as possible. The proofs that require $\rho > 2$ are very likely loose on examples that arise in practice, since we and others note that inputs can be meshed with rather smaller values for ρ with little difficulty, so I set a default value of $\rho = 2$.



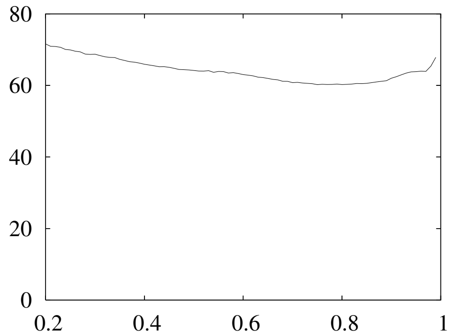
(a) Stanford Bunny: runtime



(b) Stanford Bunny: size



(c) Line & Circle: runtime



(d) Line & Circle: size

Figure 3.2: *Left:* Runtime in seconds, versus k. *Right:* Number of vertices that SVR outputs in thousands, versus k. See the discussion in the main text.

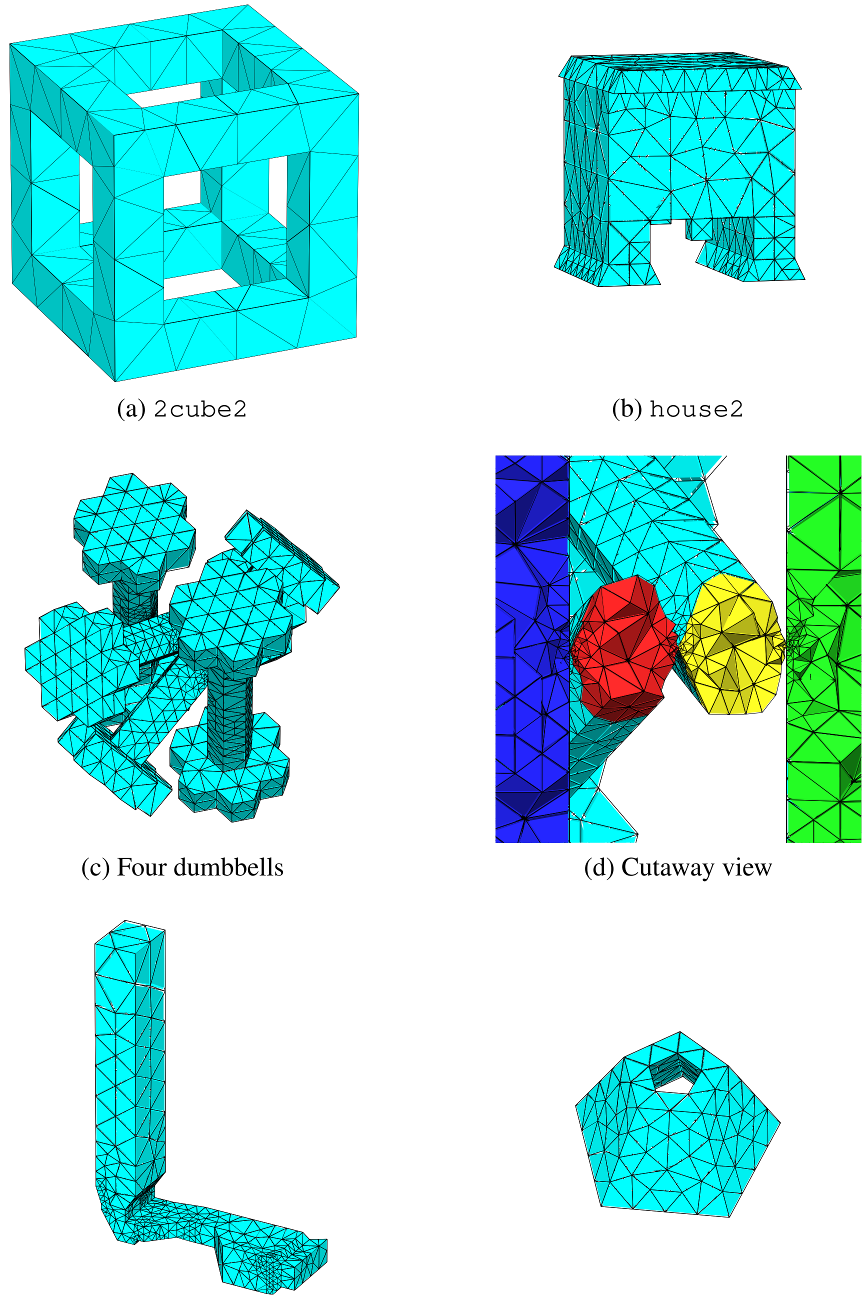
The effect of the value of k is less well understood from the axioms. Intuitively, one would expect that as k approaches unity, the number of Steiner points in the output should decline, since SVR will more frequently be allowed to warp rather than inserting a Steiner point. Conversely, as k shrinks to zero, it should insert increasingly many vertices. Thus, if runtime is no object, one should set k to 1.0 (modulo possible numerical errors arising from using inexact arithmetic to determine if a point is in the warp region, although these should be easy to fix). On the other hand, runtime is the main reason to use SVR. Clearly, as k gets very small, SVR will insert a huge number of Steiner points, and these insertions will regulate the runtime. As k gets very close to 1, we should expect the quality of the skinniest element seen during refinement to get worse. Since the quality bounds the degree of mesh vertices, this means we lose the bounds on runtime. As a lower-order yet still significant effect, increasing k increases the volume of the ball that must be searched, which also increases runtime.

I ran an experiment to determine the optimal setting of k. I tested SVR with $\rho = 2$ and varying k from 0.20 to 0.99 in steps of 0.01 on two inputs: the Stanford bunny, and the $n = 10,000$ pathological example. Figure 3.2 summarizes the results. As expected, in general output size increases as k is made smaller. Also, in general runtime increases at very low values of k, because insertions predominate; and at very high values of k, because of bad quality. It is not clear why the pathological line and circle example sees an increase in mesh size with high values of k. Given the results on these two examples, and giving greater credence to the non-pathological example as being indicative of standard behaviour, we set the default k to 0.9 and use that in all further experiments.

3.6 Experimental results

Pictures of a set of PLC examples meshed by SVR appear in Figure 3.3. The `2cube2` and `house2` examples are credited to Shewchuk; the four dumbbells example is from Phillips; the `nalco1a` example is from Ollivier-Gooch; and `vava` is from Vavasis (the latter two communicated to me by Shewchuk). These show that indeed, SVR is able to mesh some PLCs, even when strictly speaking they violate the $90°$ requirement on angles between faces (witness the base of `house2` and the bottom-right of `nalco1a`; the base of `vava`, not visible in the picture, also has an acute angle). The meshes generated grade from small features to large, which is especially visible in `nalco1a` and in the cutaway view of the four dumbbells. SVR meshes a volume; to generate these pictures, as a post-processing pass I did a walk starting from the vertices of the bounding box, removing any simplex that could be reached without crossing a triangle of a PLC facet. My code works for convex faces; the test of whether a triangle belongs to a PLC facet is imperfect for non-convex faces, which covers up a hole in `house2`.

To evaluate the runtime of SVR with respect to prior codes, I ran some experiments on point-cloud inputs. I have not yet had the opportunity to spend significant effort on speeding up the handling of PLCs, so I report no such numbers here. I compare to the codes Pyramid [She98b] and TetGen [Si06]. The experiments were run on a desktop 3.2 GHz Pentium D, with 2 GB of RAM running Linux 2.6. I used the gcc compiler, version 4.2.1, and flags `-DNDEBUG -m32 -O2 -g -fomit-frame-pointer -mtune=native`. Results under older versions of gcc, different compiler flags, and on different platforms, are qualitatively similar. One file in TetGen cannot be optimized, for reasons that escape me (it is Shewchuk's numerical predicates library, which works perfectly well under optimization in both SVR and Pyramid); that file is compiled with the `-m32 -O0` flags. The `-m32` flag is required because all three codes currently only support 32-bit pointers on



(a) `2cube2`

(b) `house2`

(c) Four dumbbells

(d) Cutaway view

(e) `nalco1a`

(f) `vava`

Figure 3.3: Pictures of the meshes SVR generates for a number of PLC inputs. Each example was meshed using the default settings of $k = 0.9$ and $\rho = 2.0$. The entire volume was meshed; for display, I removed all simplices reachable from the exterior without crossing a PLC facet.

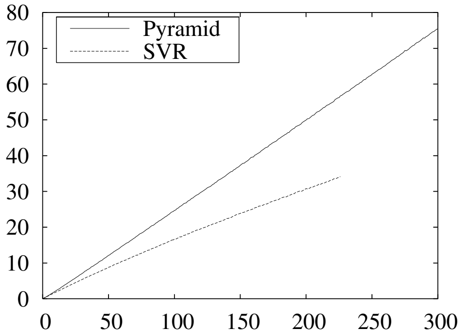
Input	SVR	Pyramid	TetGen	SVR	Pyramid	TetGen
Stanford Bunny ($n = 34890$)	4.62	6.35	12.4	59702	59040	74269
Line & Circle ($n = 2000$)	0.80	4.79	6.5	12119	14003	14573
Line & Circle ($n = 20000$)	7.62	N/A	N/A	120933	N/A	N/A
50^3 Grid ($n = 125000$)	11.30	15.96	45.9	129839	129929	130140
100^3 Grid ($n = 10^6$)	97.71	179.04	400.3	1016262	1017799	1018684

Figure 3.4: 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.

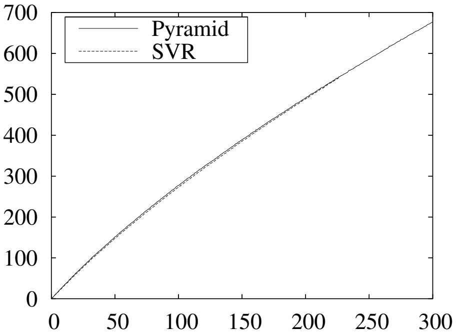
this combination of platforms and compilers. *A priori* there is no reason to believe that the update to 64-bit machines would be at all difficult for any of the codes, and indeed Pyramid is known to run in 64-bit mode on other combinations of platform and compiler.

The experiments use the default parameter settings of $k = 0.9$ for SVR, and $\rho = 2.0$ for all three codes. I measure time using the UNIX 'time' utility, summing the reported user and system times, averaged over five runs. The examples are those described in Section 3.4: the Stanford bunny; the pathological example, in two sizes; and the grid, in two sizes. Figure 3.4 summarizes the result of the experiments. As expected, on the pathological inputs, SVR is substantially faster. Indeed, even with a modest number of input points — just 10^4 points on the line and 10^4 on the circle, both Pyramid and TetGen run out of memory. Each simplex takes at least 8 words to describe (4 words for the vertices, and 4 words for the neighbours), which is 32 bytes; the approximately 10^8 simplices in the initial Delaunay that both Pyramid and TetGen try to compute therefore consume 3.2 GB, which exceeds the limit of addressable memory for 32-bit user programs under Linux. Interestingly, SVR is also faster on non-pathological inputs, and scales better than Pyramid or TetGen do as the input size grows. The output size of all three programs is similar, which shows that SVR does not trade away good output size for its theoretical guarantees.

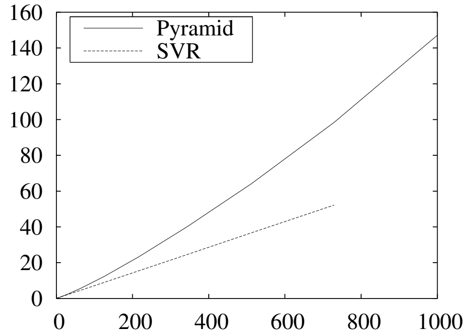
TetGen does not support directly generating a quality mesh from a point-cloud input; instead, it requires first computing its Delaunay triangulation, then invoking TetGen a second time to improve the quality. It is clear from discussions with Si that this is an oversight; and, in particular, it is disastrous from a runtime standpoint for the pathological examples. In deference to the fact that this is an easily fixed bug, I report the runtime after subtracting the cost of outputting the intermediate mesh to file and re-reading it into memory. This slightly reduces the precision of the runtime numbers for TetGen.



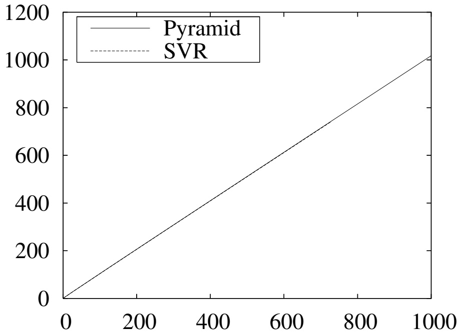
(a) 27 Bunnies: runtime



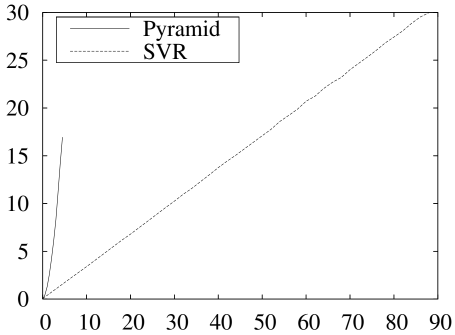
(b) 27 Bunnies: size



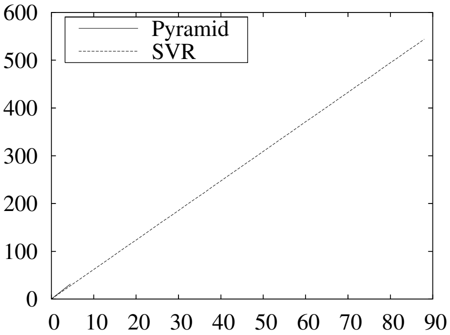
(c) Grid: runtime



(d) Grid: size



(e) Line & Circle: runtime



(f) Line & Circle: size

Figure 3.5: Scaling examples comparing SVR and Pyramid. **Left:** Time on the y axis is in seconds. Input size on the x axis is in thousands of points. **Right:** Number of points (in thousands) in the output *vs*. number of points in the input (in thousands). Jobs were killed upon allocating over 500 MB of memory.

I ran some experiment to see how well SVR scales to larger inputs. Two of the point-cloud inputs (the grid and line and circle) were described in an obviously scalable manner. For the Bunny, I made 27 copies of the bunny in a $3 \times 3 \times 3$ grid, a total of 940518 points, and randomly sampled from this set. These experiments were run using the same compiler settings as before, but on a factory-standard MacBook Pro with a 2.16 GHz Intel Core 2 Duo processor and 1 GB of RAM. In order to be able to run a large number of examples overnight without getting stuck on a thrashing job, I had the system automatically kill jobs that allocated more than 500 MB as reported by `ps`. The results, in Figure 3.5, show that SVR scales well with input size: at all sizes, it runs faster than Pyramid, and increasingly so as the input size increases. However, SVR uses more memory, due to the caching of circumcenters; the 500 MB cutoff affected SVR sooner than it affected Pyramid, except on the pathological examples. The output size difference between Pyramid and SVR is negligible: it is nigh-impossible to distinguish the two curves without a magnifying glass. I also ran another experiment, not shown in the figure, with a $150 \times 150 \times 150$ grid. SVR on this input exceeded the bounds of physical memory by a factor of two. Despite this, it was able to consume 4% of CPU, indicating good location of reference as would be expected from this fundamentally parallel algorithm.

3.6.1 Cache performance and profiling

During the development of SVR, I made heavy use of the Valgrind toolchain to study the cache performance and instruction profile of running code. The SVR code is an unstructured numerical program; therefore, systems folklore estimates we should expect to issue about one instruction per clock cycle if we are CPU-bound, because of the heavy cost of branching. This is approximately what we in fact do see, although somewhat unsurprisingly the ratio appears to be falling as I reduce the number of computations needed to compute the mesh.

A large fraction of runtime — approximately 20% on the Stanford bunny example — is spent outputting the file to disk. One of the goals of SVR is to allow using the mesh through the API, instead of needing to write it to disk and translate it to another in-memory format, so I ignore this cost in the profiling discussion. Of the remaining time, more than one third of the number of instructions issued and of the cache misses (and thus, one assumes, over a third of the total runtime) are related to geometric computations. There are principally three reasons for SVR to perform geometric calculations, each of them taking an equal fraction of the runtime: (1) computing circumcenters, (2) computing the `insphere` predicate when inserting a point, and (3) computing distances during point location. To reduce the cost of computing circumcenters, I store the circumcenters I have computed; it is unclear whether there is any more savings to be performed here. It should be possible to use the stored information to help with `insphere`: in many cases, simply checking the distance of the query point to the circumcenter of the simplex should give us a sufficiently accurate answer, and only if not should we need to call the exact predicate. The question of point location was the focus of Section 3.1; as mentioned there, it is quite likely that further improvement could occur. Another third of the time is related to topological changes in the simplicial complex. I expended a significant amount of effort optimizing this code, and find it unlikely to be able to much improve its performance without a substantially different data structure such as a compressed mesh à la Blandford *et al* [BBCK05, Bla05].

Cache efficiency is good, though not stellar. The instruction cache essentially never misses. However, on modestly-sized examples (such as the bunny), about 1.5% of data reads missed in L1 and had to fetch from L2 cache. About one in five of those — 0.3% overall — of data reads went to main memory. Keeping in mind that latency on a modern machine is in the teens of cycles for a miss to L2 cache, and low hundreds for main memory, this suggests that the program is likely to be frequently spending time stalled waiting for memory, but not predominantly so. This in turn implies that any further reduction in CPU time will need to include reductions in the size of the working set.

I should note that SVR handles its own memory allocation. Using the system `malloc` and `free` routines is disastrous both in terms of cache efficiency and in instruction count. Furthermore, to be generic, `malloc` and `free` need to tag each allocated block with an additional word of data to store the size of the allocated block. But the vast majority of allocations that SVR performs are all of the same size: usually list nodes (two words) and tetrahedra (19 words); furthermore, SVR knows the size of these allocations and deallocations at compile time. I therefore use freelists, one per size class, for memory allocation. The system `malloc` is only invoked for items larger than 20 words, or, for small items, one megabyte at a time. This substantially reduces the relative overhead for the size counters and eliminates the vast majority of calls of `malloc` and `free`. Allocating and freeing usually takes only two memory accesses in this framework, and similarly little computation. Freelists have a tendency to cause items allocated close in time to be allocated close in address space, although this specific benefit decreases over time as the freelist becomes mixed.

A major cost of using C++ is that complicated memory allocation schemes are almost impossible to explore. In particular, to avoid memory leaks, I must maintain reference counts, which has a known negative cache-performance impact: dropping a pointer to the head of a list of handles to simplices in a reference-counted environment means iterating over the list and touching each simplex, while doing so in a garbage collected environment costs zero time. Furthermore, a copying garbage collector is free to move simplices around to create memory-space locality between topological neighbours — or even to compress the mesh.

The most important remaining method to reduce the runtime is to reduce the number of Steiner vertices in the output. This would reduce the memory footprint, reduce the number of in-circle and circumcenter computations, and reduce the number of topological changes. It is also desirable as a goal unto itself. One may hope that recent work by Üngör on producing small meshes in two and three dimensions will help in this respect.

3.7 Extensions and future goals

Moderately small input angles seem to mostly work, usually. Given how often small angles appear in inputs, this is an insufficient level of reliability. Cheng, Dey, Levine, and Ramos have produced provable algorithms and a correct but relatively slow implementation for small angles [CDR07, CDL07]. Their work also handles piecewise *smooth* complexes, the obvious extension of PLCs to the curved surfaces common in CAD inputs. Their framework does not immediately extend to SVR's framework, but it is highly likely possible to marry the two to get a mesher that is provably fast and always terminating with small meshes on clean CAD input.

Slivers, as mentioned before, cause numerical robustness issues even while meshing, especially on PLC inputs. The geometric primitives and predicates need to be analyzed and fixed to handle slivers in order to obtain truly robust code.

We proved that the SVR algorithm can be made parallel [HMP07b]. The parallel algorithms community seems to view mesh refinement as a good new toy problem for their automatic and semi-automatic parallelization techniques [KCP06]. While it is largely known how to parallelize traditional dense matrix problems, unstructured problems like meshing are still open. SVR fits well in the category of what is starting to be known as workset parallel programs: SVR maintains a work queue, many of whose items are independent. I developed the current SVR code with an eye on its future parallelization, and hope that it will not be overly onerous to extend it to the shared-memory parallel case. In the near future, I believe that it is worth pursuing constant-factor improvements: it seems likely that we can halve the runtime of the single-threaded SVR, whereas achieving that level of improvement usually takes three or four processors in a parallel program. A further benefit to waiting is that commodity machines are likely to by then have eight cores, at which point parallelization will become more important.

Chapter 4

Dynamic Meshing for Point Clouds

Consider an input consisting of a list of points, denoted by their coordinates, lying in a box of appropriate size (see Section 4.4). The goal is to provide a dynamic algorithm that can handle adding or removing a point from the input. While meshing point clouds is far simpler than handling the full generality of also conforming to PLC inputs, nevertheless, several of the core concepts needed to analyze higher-dimensional inputs already arise. Indeed, despite the apparent oversimplification, this is the longest chapter of the thesis.

In joint work with Acar [AH06], we conjectured that SVR was a dynamically stable algorithm, based on some preliminary experimental results that indicated as much. I can now prove that conjecture false, in the adversarial setting (it may still hold against a weakened adversary). Consider the example in Figure 4.1. In SVR, a Steiner point may "warp" to the input early on — indeed, for realistic parameter settings, it will always do so in the very first iteration of the main loop. Every Steiner point inserted from then on depends, if only in the low-order bits, on the coordinates of the first point that was chosen. By the same token, while input points have fixed coordinates, their point location is through simplices or vertices whose coordinates depend on the first insertion. Changing the first point is therefore extremely expensive for dynamic stability. In an adversarial setting, unfortunately, the adversary may place a point p in a position where, when the adversary toggles whether or not p is part of the input, SVR is required to change its decision as to which vertex it should insert first. This then can force SVR to change its point location structures and to change the location of almost every Steiner point it inserts, which means the response time can be no faster than linear in the size of the output. Randomization does not help: it is easy to create examples with only one choice of legal warp move in SVR, but an arbitrary number of uninserted points. The example shows that SVR is not dynamically stable, and cannot be made dynamically stable without substantial modification.

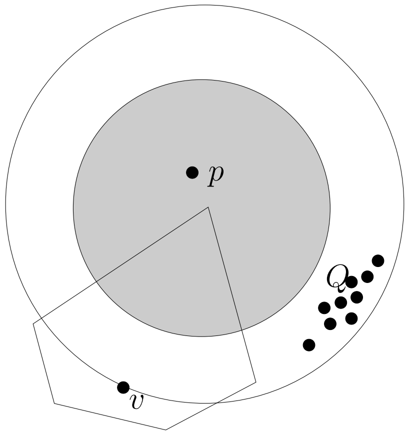


Figure 4.1: Even with randomization, the adversary can force SVR to make a deterministic choice of warp location and thus change the position in the point location structures of an arbitrary number of points.

This chapter presents a different meshing algorithm for point clouds that is both efficient in the static case, and dynamically stable. The chapter is in three sections. I start by building a quadtree that conforms to the local feature size. I show that a relatively naive quadtree construction algorithm is dynamically stable. Next, I show how to produce a good quality mesh using a very general technique. This second section does not concern itself with runtime complexity at all; instead, it shows that with an appropriate choice of Steiner points to add, we can arrive with a set of points whose Delaunay triangulation is a quality mesh of the input point cloud. Finally, in a third section I show how to use the quadtree to quickly implement the technique of the second section. In the end, then, we see an algorithm that provably produces a mesh that respects the input, has good radius/edge quality, is size-conforming, and in the face of a point being added or removed from the input, responds in $O(\lg L/s)$ time, which is an optimal time bound. The first three sections ignore any unusual effects near the boundary of the domain, where Voronoi cells become unbounded. To close the chapter, I show that how to specify the input domain such that no special handling is required for these boundary effects, which greatly simplifies both proofs and implementations.

4.1 Building a quad-tree for point location

Borrowing definitions from Bern, Eppstein, and Gilbert [BEG94], a *cell* is an axis-aligned hypercube of the specified dimension; I denote the length of a side of a cell c using $|c|$. A

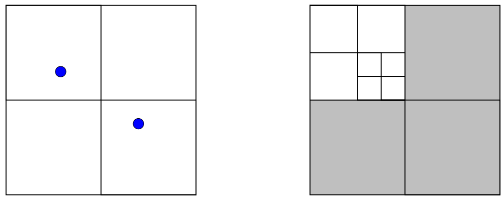


Figure 4.2: **Left**: four crowded cells. The upper-left and lower-right cells each have a point, so they crowd each other. In prior work, the lower-left and upper-right cells would not be said to be crowded; this is a constant factor improvement that complicates the proofs, but one which an implementation might find important in practice. **Right**: ill-graded cells. The shaded cells are four times larger than one of their neighbours. Note that neighbourhood goes through corners.

quad-tree is a cell subdivided into smaller cells. Two cells in the quad-tree are neighbours if they share a corner. During meshing, we track the location of points relative to the cells. A cell c is said to be **crowded** if there are at least two input points that lie in the union of c and of its neighbours — that is, either c contains two points, its neighbours contain two points, or c and one of its neighbours each contain one point. A cell is **ill-graded**[1] if it has a neighbour c' such that $|c|/|c'| \geq 4$. We say that a quadtree is *well-graded* if every unsplit cell is both well-graded and uncrowded. It is not hard to prove that a well-graded quad-tree produces a size-conforming, good quality mesh once triangulated. Note that this mesh does not respect the input; it merely conforms to its local feature size.

Figure 4.3 shows the algorithm for building a quad-tree. The algorithm starts with a bounding box (square) of the the point set, with side length L. It maintains a **work queue** Q of work items, *i.e.*, pointers to cells that need to be split, and a mapping from each cell to the set of input points that it contains. Q is partitioned into $\lg L/s$ buckets such that the bucket Q_i is a queue containing the cells of size exactly $L/2^i$. The "top" of the queue has the largest element. For dynamic stability, we will also require that each bucket maintain FIFO order (it must be a queue); this is not necessary for static runtime. Cells may be pushed repeatedly onto the queue to no ill effect. The algorithm iterates over the work queue until it is empty. On each iteration, it finds a cell that has not yet been split, splits it, reassigns points, and pushes new work onto the queue.

We now wish to establish three theoretical results: First, that the output quad-tree is, as we claim, size-conforming. Second, that we can compute it in $O(n \lg L/s)$ time. Third, that upon appropriate updates (namely, adding or a removing a point from the input), we can respond to the change and simulate running from scratch in only $O(\lg L/s)$ time.

[1]The original term was "unbalanced." Our experience in conference reviews shows that the "balance" term is confusing: quite reasonably, many think it refers to the depth of the quad-tree. Grading is a term commonly used in engineering for the very concept we wish to express.

```
BUILDQT(𝒫: point set, L: real, d: int)
P: map from quad-tree cells to points of 𝒫
Q: a 2-bucketed priority queue of cells, keyed by size, largest first
Add the association [0, L]^d → 𝒫 to P
if [0, L]^d is crowded (that is, |𝒫| > 2) then
    Q ← {[0, L]^d}
end if
while |Q| > 0 do
    c ← DELETEMIN(Q)
    Split c into 2^d smaller cells {c_i}
    for each point p ∈ P(c) do
        Find the c_i that contains p
        Add p to P(c_i)
    end for
    Delete the entry P(c)
    for every neighbour c' of c do
        if c' is ill-graded (|c'| = 2|c|) then
            enqueue c' onto Q
        end if
    end for
    for every child c_i of c do
        for every neighbour c'_i of c_i do
            if c_i is crowded (that is, |P(c_i) ∪ P(c'_i)| ≥ 2) then
                enqueue c_i onto Q
                skip to the next child
            end if
        end for
    end for
end while
```

Figure 4.3: Static algorithm for building the point location quad-tree given a point-set input. This algorithm runs in time $O(n \lg L/s)$. See also Figure 4.4 which contains the changes that allow self-adjusting computation to achieve fast response time.

Intuitively, the mesh is size-conforming because it separates any two points by at least one cell (and thus cells are not too large), but only splits further to achieve good grading (thus cells are not too small). The main asymptotic cost for refinement is to relocate points in Line 10; when a point is split, the size of the cell it occupies falls by half. This can only happen $O(\lg L/s)$ times per point, which gives the runtime and response time bounds.

4.1.1 The quad-tree is size-conforming

Lemma 4.1.1 *If a cell c is crowded, then its parent c^+ was crowded.*

Proof: The neighbours of c are also neighbours of c^+, unless they are children of c^+. Points that crowd c therefore also crowd c^+. ∎

Lemma 4.1.2 (Crowded cells are small) *During the algorithm, all unprocessed crowded cells are of the size of the smallest cells in the mesh, or exactly twice that size.*

Proof: Initially, this is trivially true (there is only one cell in the mesh). Later, consider the cell c^+ that was split to create a crowded cell c. According to Lemma 4.1.1, c^+ was itself crowded, and thus by induction was the smallest or nearly-smallest cell in the mesh. Upon splitting c^+, c^+ is the largest bad cell in the mesh: there cannot be any larger crowded cells, or the work queue would have returned them instead. Therefore, all crowded cells are of size $|c^+|$ or $|c^+|/2$. Having split c^+, the new cells are half the size of c^+. Cells crowded by c were already crowded by c^+, so only the new cells can be newly-crowded, and the new cells are all the smallest cells in the mesh. ∎

Lemma 4.1.3 (Cell size lower-bounded by lfs**)** *When a cell c is created, for all points $x \in c$ it is the case that* $\mathrm{lfs}(x) \leq 4.5\sqrt{d}|c|$.

Proof: When c is created, it is because the algorithm is splitting c^+. There are two reasons to split c^+: for crowding or for grading. If for crowding, then c^+ and a neighbour c' contain two inputs p and q in $c^+ \cup c'$. The two inputs are, at the farthest, on opposite corners of c^+ and c': $|pq| \leq \sqrt{d}(|c^+| + |c'|)$. Because c' is crowded, and because the queue orders largest-first, $|c'| \leq |c^+|$. This proves that there is a point $y \in c^+ \cup c'$ where $\mathrm{lfs}(y) \leq \sqrt{d}|c^+|$. The local feature size at $x \in c$ is thus no larger than $\mathrm{lfs}(x) \leq \mathrm{lfs}(y) + ||xy|| \leq \frac{9}{2}\sqrt{d}|c|$.

Alternately, if c^+ was split for grading, there was a neighbour c' that was much smaller: $|c'| = |c^+|/4 = |c|/2$. By induction (since c' was created before c), we can assume that $\mathrm{lfs}(y) \leq 4.5\sqrt{d}|c'|$ for all points $y \in c'$. The Lipschitz condition lets us know that

$\text{lfs}(x) \leq \text{lfs}(y) + ||xy||$. We can choose y to be the closest point in c' to x, which may be as far as the length of the diagonal of c^+: $||xy|| \leq 2\sqrt{d}|c|$. Overall then, we get $\text{lfs}(x) \leq (4.5\sqrt{d}/2 + 2\sqrt{d})|c| = \frac{17}{8}\sqrt{d}|c| < 4.5\sqrt{d}|c|$. ■

Theorem 4.1.4 (Size-conforming) *The mesh output by* BUILDQT$(\mathcal{P})$ *is size-conforming: for any point* $x \in [0, L]^d$*, the cell* c_x *that contains* x *satisfies*

$$k_{low}|c_x| \leq \text{lfs}(x) \leq k_{hi}|c_x|$$

For constants k_{low} *and* k_{hi} *depending only on dimension.*

Proof: Lemma 4.1.3 shows the upper bound on lfs. The lower bound is as follows: If x is on the medial axis of the point cloud, then there are at least two points p and q equidistant to x, which define $\text{lfs}(x)$. Only one of the two can be in c_x and its neighbours or else c_x would be crowded; wlog. let q be outside the neighbourhood of c_x. Then $\text{lfs}(x) = |xq| \geq |c'| \geq |c_x|/2$ (otherwise, c' would cause bad grading with c_x).

Otherwise, x is off the medial axis. Let y be the closest point on medial axis that minimizes $|xy| + \text{lfs}(y)$; this quantity is $\text{lfs}(x)$. Both terms are positive, so we know that $\text{lfs}(x)$ is at least as large either of them in isolation. If c_y is not a neighbour of c_x, then $\text{lfs}(x) \geq |xy| \geq |c'| \geq |c_x|/2$. But if c_y is in a neighbour of c_x, then by the argument above, $\text{lfs}(x) \geq \text{lfs}(y) \geq |c'|/2 \geq |c_x|/4$.

The constants are $k_{hi} = 4.5 \times \sqrt{d}$ and $k_{low} = 1/4$. Importantly, however, in cells that contain input, $\text{lfs}(x) \geq |c|/2$. ■

4.1.2 BUILDQT runs in $O(n \lg L/s)$ time

Lemma 4.1.5 (Always-quality) *At any point during the run of the algorithm, if* c *and* c' *are neighbours, then their sizes differ by a factor at most four.*

Proof: Consider a cell c^+ that is split, creating c, and consider a neighbour c' of c. By Lemma 4.1.2, if c^+ is a crowded cell, then the mesh is well-graded and neighbours differ in size by a factor at most 2: $|c'| \leq 2|c^+|$. On the other hand, if c^+ is an ill-graded cell, then by the queue order we can assume that $|c'|$ must again be at most $2|c^+|$ because otherwise c' would have been processed first. In either case, c is half the size of c^+, so c and c' differ in size by at most a factor four. ■

Lemma 4.1.6 *At every step of the algorithm, cells in the quad-tree have* $O(1)$ *neighbours.*

Proof: Since the size between neighbours differs only by a factor of four, we can only pack at most six neighbours per side of a cell (four along the side, plus one on each end), for a total number of cells equal to $6^d - 4^d$. ■

Lemma 4.1.7 (Splitting is fast) *Except for relocating points, handling a work queue event takes constant time.*

Proof: In any reasonable representation, checking whether a cell has been split is trivial, as is doing the structural modification to the quadtree. Relocating points we have explicitly excluded from the accounting in this lemma. Iterating over children $\times$ neighbours is a constant number of iterations since there are 2^d children and $O(1)$ neighbours. Checking for grading is obviously constant time. The sets $P(c_i)$ and $P(c_i')$ may be large, but when checking whether their union contains more than one point we only need to count to two and stop. ■

Lemma 4.1.8 *Any given input point p is relocated at most $O(\lg L/s)$ times over the course of the algorithm. Each relocation costs $O(1)$ time.*

Proof: A point p initially lies in the root cell $[0, L]^d$. Every time it is relocated, the size of its new cell is exactly half the size of its old cell. Because the output is size-conforming, the smallest cell is of size $\Omega(s)$, so p can only be relocated $\lg(L^d/\Omega(s)) \in O(\lg L/s)$ times. Relocating only takes the time to find the new cell out of 2^d new cells, and thus takes $O(1)$ time. More precisely, it takes d greater-than tests because we are using an axes-aligned quadtree. ■

Theorem 4.1.9 *The* BUILDQT *routine runs in time $O(n \lg L/s)$.*

Proof: Points are relocated $O(\lg L/s)$ times each and there are n points. Furthermore, every split costs $O(1)$ additional time. There are $O(m)$ splits because the output is size-conforming, and $m \in O(n \lg L/s)$ because the input is a point cloud. ■

4.1.3 BUILDQT is $O(\lg L/s)$-stable

To establish the runtime of our dynamic algorithm, we determine the *stability* of the output relative to changes in the input. The arguments will be familiar to designers of parallel algorithms — indeed, we draw on packing arguments from prior parallel meshing results [STÜ04, STÜ07, HMP07b]. Our runtime is regulated in large part by the data dependence structure of our algorithm. We must show that dependence paths are at most $O(\lg L/s)$ long. Unlike in parallel algorithms, we must also show that the dependences cannot fan out: even constant fanout would give us a runtime of $O(\mathrm{poly}(L/s))$, which is completely unacceptable.

Formalizing the notion of stability, consider a run of our algorithm. It reads in the points, performs some operations, reads and writes to memory, then returns an output. We can define an *execution trace* in the following way: operations and memory locations are nodes; there is an edge from a memory location a to an operation f if f reads a; and there is an edge from operation f to memory location b if f writes b. If f reads a and writes b, we say that b has a *data dependency* on a. The *dynamic stability* of one point p is the symmetric difference between the sets of nodes in trace T_1 where p is not present, and the nodes in another trace T_2 where p is present. Note that this is a symmetric difference, so that the stability of adding and removing the same point are equal.

To abstract away from memory locations and return to the more comfortable world of input points and quadtree cells, I say a cell c *blames* p if the operation that splits c is a trace descendent of the memory location that stores p. Thus, a cell c blames a point p if p is one of the (possibly many) points that crowds c. Inductively, c also blames p if c is made to be ill-graded because a neighbouring cell c' was created by a split, and c' blames p. Note that a cell may blame its splitting on many points; indeed, it will always blame at least two points.

If we consider a given cell c and a point p that it blames, then the distance in inductive hops from c to p is at most $O(\lg L/s)$: in every hop, we either directly blame p, or we blame p through a neighbour of half the size. Thus the trace is a shallow graph; it remains to be shown that the number of descendents of an input point (the number of cells that blame it) is bounded.

Lemma 4.1.10 *Assume p is blamed for the split of a cell c. Then $||pc|| \in O(|c|)$.*

Proof: If c is being split for crowding, then p is either within c or is in a neighbour c' of c, and $|c'| = |c|$. Thus $||pc|| \leq |c|$. If instead c is being split for grading, then we can follow the causal chain that leads to a cell c' that was split for crowding by p. Label the chain c_i with $c_0 = c$ and $c_k = c'$. Because of the grading condition, we know that $|c_i| = 2|c_{i+1}|$ and thus $|c| = 2^k|c'|$. The distance we can travel along the chain is maximized if the chain follows the diagonal of the cells, a total distance of $(1 + 2^k)\sqrt{d}|c'|$. Finally, c' either contains p or neighbours an equal-sized cell that contains p. Thus the distance from p to c is at most $(2 + 2^k)\sqrt{d}|c'| = (1 + 2^{1-k})\sqrt{d}|c|$, with $k \geq 1$ and $d \geq 1$. ■

Lemma 4.1.11 *Any point p is blamed for at most $O(\lg L/s)$ splits.*

Proof: Given a size class 2^l, the prior lemma showed that any cell of size 2^l that is blamed on p must have distance at most $O(2^l)$. In dimension d, these cells have volume $(2^l)^d$ each, and must fit within a volume of $O(2^{ld})$. Therefore, there can be only $O(1)$ splits in size class l that are blamed on p. Because the output is size-conforming (Theorem 4.1.4), there are $O(\lg L/s)$ size classes. ■

At this point we have accounted for the stability of the mesh itself: only $O(\lg L/s)$ will be created or destroyed upon a single-point insertion. In the scientific computing application, this limits numerical error due to reinterpolation. However, to establish the dynamic response time of the algorithm, we must also account for changes in structures only used internally. The principal such structure is the point location structure — the assignment of points to cells. To account for point location costs, we need to be a bit more careful about blame. If a split relocates a point, there are two possibilities: the split is due to crowding, or the split is due to grading. Lemma 4.1.2 implies that splits due to grading only occur on cells with at most one point inside, so paying for the relocation is only a constant extra cost. Splits due to crowding may be very costly, but the presence or absence of a point p only changes the decision about whether to split a crowded cell c if p is exactly the second point in the cell and its neighbours. This allows us to cut the causal chain and only have a point q blame its relocation on p when p is exactly the second point in the cell.

Lemma 4.1.12 *Only $O(\lg L/s)$ point location decisions blame any given input point p.*

Proof: As seen in Lemma 4.1.8, every point is reassigned at most $O(\lg L/s)$ times during the algorithm. What is left is to see how many other points are reassigned because of the presence of p that would not otherwise be reassigned (*i.e.*, their containing cell was split because p was present, but would not have been split were p absent).

There are two reasons a point can be reassigned: either it is in a crowded cell being split, or it is in an ill-graded cell being split. A reassignment due to a crowded cell c can only be affected if the point p was either in the cell c or in a neighbour c' of c. Furthermore, we know that there was exactly one other point in c or c' — otherwise the algorithm would split regardless of the presence or absence of p. On the other hand, Lemma 4.1.2 implies that any ill-graded cell c must be uncrowded — c therefore only has one point inside. In other words, if a split reassigns any points, it reassigns exactly one point. The set of splits is $O(\lg L/s)$-stable, and thus so is the set of point reassignments. ■

Putting these observations together, we get the final result of this section:

Theorem 4.1.13 *The* BUILDQT *algorithm is $O(\lg L/s)$-stable under single point insertions and deletions.*

4.1.4 BUILDQT response time is $O(\lg L/s)$

The stability bound just proved does not yet immediately translate into a dynamic algorithm. However, it suggests that we can use self-adjusting computation (SAC) to automatically dynamize BUILDQT. Under the SAC framework, an execution of BUILDQT is seen as a set of *operations*, which I will define shortly. Acar [Aca05] proved that under certain assumptions, SAC can respond to a change in time equal to the stability bound. The assumptions are (1) the program is *concise* — no operation is done twice, and (2) the program is *monotone* — in every trace that performs operations both a and b, their relative order is the same: a occurs before b.

We can think of an operation as being a sequence of machine instructions, such that the entire sequence can be run in constant time. Thus, BUILDQT line 8 ($c \leftarrow$ DELETEMIN(Q)) is an operation because we use a 2-bucketed approximate priority queue, in which DELETE-MIN is constant time. By contrast, relocating points in the loop starting line 10 is not an operation, though each iteration of the loop is. For simplicity, I describe the body of the main loop as a single operation, which implements the loop starting line 10 as a recursive function call.

Lemma 4.1.14 BUILDQT *is concise.*

Proof: A crowded cell c_i is only added to the priority queue once, by its parent (line 23). It cannot also be added to the queue as an ill-graded cell, because until c_i is processed, no smaller cell can be processed. Similarly, an ill-graded cell c' will also only be added once (line 17: when it is added to the work queue, until c' is processed, no other smaller cell is processed. Therefore, each iteration of the main loop only occurs (at most) once per cell. Finally, a point cannot be relocated from a cell c to a sub-cell c_i of c more than once, since c is only split once. ■

Monotonicity is harder to prove, at least in part because BUILDQT as presented is not monotone. Consider two execution traces of BUILDQT, T_- and T_+, where the input for the latter differs from the input from the former in that it has an additional point p. In T_+, a cell may be split because it contains p and another point; in T_-, it may be that the cell was split because it was ill-graded. To repair this, I change the algorithm in the following two ways: (1) the set of neighbours that are checked in line 17 must be traversed in a canonical order (for example, clockwise in two dimensions); (2) the queue will now store not only the names of cells to be split, but also a "reason" to split it. The reason field is for crowded cells (those enqueued in line 23) is the cell itself. When a cell c being split due to crowding enqueues its neighbours in line 17, the reason field for those ill-graded cells is set to c. Finally, when an ill-graded cell being split with reason c enqueues its neighbours, the reason field for remains is c. More briefly, every cell enqueued directly or indirectly by the split of a crowded cell has the same "reason" field. See Figure 4.4 for the modified, monotone algorithm.

This modified algorithm, I claim, is monotone. To prove monotonicity, consider two operations a and b such that there is an input that induces a trace where a occurs before b. I say that the program is *monotone with respect to a and b* if in *every* trace where a and b both appear, a occurs before b. I distinguish three classes of operations: splits due to crowding (those enqueued in line 24); splits due to grading (those enqueued in line 18); and point location operations (lines 12 and 13). The proof is in parts, showing that for any a drawn from one class of operations and b drawn from another class, the program is monotone relative to them.

Fact 4.1.15 BUILDQT-DYN *is monotone relative to pairs of split operations that are both due to crowding.*

Proof: Crowded cells are enqueued immediately when they are created; therefore, they are dequeued in order of size. This shows that if $|c_1| \neq |c_2|$, their relative order is determined by their relative size. If instead $|c_1| = |c_2|$, then consider the crowded cells c_1^+ and c_2^+ that enqueued them. If c_1^+ and c_2^+ are the same cell c, then the fact that the algorithm uses a canonical ordering for numbering children of a node means that c_1 and c_2 are enqueued in the same relative order in any trace where c is split for crowding. Each bucket of the 2-bucketed priority queue maintains a deterministic ordering of its items in each bucket (LIFO, for example). Therefore, the order in which c_1 and c_2 are dequeued is uniquely determined by the order in which they are enqueued, which we argued was the same for all traces.

If instead c_1^+ and c_2^+ are not the same cell, then the relative order of c_1 and c_2 on the queue is determined by the relative order of c_1^+ and c_2^+ (again assuming deterministic ordering of the queue). We can iterate this argument until the least common ancestor of c_1 and c_2, which we just argued pushes its children on the queue in the same order in all traces. Thus, two crowded cells are dequeued and split in the same relative order in any trace in which they are both crowded. ■

As a proof aid for dealing with ill-graded cells, consider the full quadtree of infinite depth (*i.e.* the object that results from the infinite process of splitting cells in breadth-first order). Looking at a cell c, in this infinite quadtree, we can identify a unique set C' of neighbouring cells of twice the size. This is the set of cells that, in the worst case, c could enqueue due to grading when c is split. In turn, the cells in C' have neighbours of twice the size. I define the **shadow** of a cell c as the transitive closure of this neighbourhood

BUILDQT-DYN($\mathcal{P}$: point set, L: real, d: int)

```
P: map from quad-tree cells to points of 𝒫
Q: a 2-bucketed priority queue of pairs of cells (c, r), keyed by |c|, largest first
Each bucket of Q must be deterministically ordered based on insertion order (LIFO,
for example)
add the association [0, L]^d → 𝒫 to P
if [0, L]^d is crowded (that is, |𝒫| > 2) then
    Q ← {[0, L]^d}
end if
while |Q| > 0 do
    (c, r) ← DELETEMIN(Q)
    split c into 2^d smaller cells {c_i} in a canonical order
    for each point p ∈ P(c) do
        find the c_i that contains p
        add p to P(c_i) with a tag of r
    end for
    delete the entry P(c)
    for every neighbour c' of c in a canonical order do
        if c' is ill-graded (|c'| = 2|c|) then
            enqueue (c', r) onto Q
        end if
    end for
    for i = 0 to 2^d − 1 do
        for every neighbour c'_i of c_i do
            if c_i is crowded (that is, |P(c_i) ∪ P(c'_i)| ≥ 2) then
                enqueue (c_i, c_i) onto Q
                skip to the next child
            end if
        end for
    end for
end while
```

Figure 4.4: The algorithm of Figure 4.3, modified so that the program is *monotone* (Definition 1.4.3). The modifications are: I clarify the order of each bucket of the priority queue, I tag queue entries with a “reason” field r, and I tag point location entries with the “reason” field. Under self-adjusting computation, this algorithm has response time $O(\lg L/s)$ when adding or removing a point from $\mathcal{P}$.

operation: intuitively, the shadow of c is the set of all the cells that c could ultimately cause to be enqueued due to grading. Thanks to the order of the priority queues, after splitting c, the entire shadow of c is split before any other cell c' with $|c'| \leq |c|$ will be processed: the shadow of c consists only of cells larger than c. Effectively, BUILDQT-DYN is a recursive function: after splitting a cell, we "clean" the effect of splitting it, removing any ill-graded cells; the priority queue serves as the stack in the recursive calls. In particular this means that after splitting a crowded cell c, but before splitting another crowded cell, every split performed has c for its reason field; I call this the **shadow property**. We can now return to proving the monotonicity lemmas.

Fact 4.1.16 BUILDQT-DYN *is monotone relative to any pair of split operations whose corresponding queue elements* (c_1, r_1) *and* (c_2, r_2) *have different reason fields* $r_1 \neq r_2$.

Proof: The shadow property implies that the relative order of c_1 and c_2 depends only on the relative order of r_1 and r_2, which are both cells that were split due to crowding. We already proved that BUILDQT-DYN is monotone with respect to such r_1 and r_2. ■

Fact 4.1.17 BUILDQT-DYN *is monotone relative to any pair* (c_1, r) *and* (c_2, r).

Proof: The algorithm requires imposing a canonical order on the neighbours of the cell r (for example, in two dimensions, counterclockwise from the lower-left corner would be a natural choice), and enqueueing the large neighbours in that order. This imposes a total order in which the neighbours will be dequeued in any trace. Recursively, the shadow property imposes a total order on all splits that could conceivably be done with reason r. ■

Fact 4.1.18 BUILDQT-DYN *is monotone relative to any two point location operations, or to any one point location operation and a split.*

Proof: While splitting a cell c, until all points in c are relocated, no other splits occur; thus the relative order of point relocations and splits is identical to the relative order of the split of c and other splits. Within a cell c that was created upon splitting a cell c^+, the order of point locations is determined by the order of point locations when splitting c^+. Inductively this reduces to the order of point locations in the initial node $[0, L]^d$, which is input. ■

Lemma 4.1.19 (BUILDQT-DYN monotonicity) *The dynamically-stable* BUILDQT-DYN *algorithm, as described in Figure 4.4, is concise and monotone.*

Adding a tag is a constant-time operation — just one or two new arguments to each function, which the program reads but whose value it ignores. Specifying the order of queues and lists will only cost time if the ordering is expensive to maintain. However, FIFO or LIFO orderings work and can be maintained in constant time. Thus, the static runtime of the dynamically-stable BUILDQT-DYN is identical to that of the original BUILDQT-DYN algorithm. The question that remains is whether the stability analysis still holds in the face of distinguishing cells that in the prior analysis were seen to be identical. Consider a cell c whose split operation is re-executed by the change propagation algorithm when updating the trace for T_1 to generate the trace for T_2. The re-execution will occur only if the reason field on the priority queue changes. The reason field r will only change if r was not crowded in T_1 but is crowded in T_2, which implies that r contains or is a neighbour of the new point p. Thus, c *blames* p and is indeed accounted for in the stability analysis. Equally, this shows that a cell whose split is re-executed contains at most one point in T_1, and at most two in T_2; therefore, the re-executed point location costs are only constant per re-executed split. Together with the conciseness and monotonicity result, this shows:

Theorem 4.1.20 *When run under the self-adjusting computation framework,* BUILDQT-DYN *can construct a graded quad-tree in $O(n \lg L/s)$ time over an input of n points with the closest pair of points being at distance s from each other. Furthermore, when a point is added to the input,* BUILDQT-DYN *can respond to the change in time $O(\lg L/s)$, where s is the distance between the closest pair after adding the point. Similarly,* BUILDQT-DYN *can respond to the removal of a point from the input in time $O(\lg L/s)$, where s is the closest pair distance before removing the point. After responding to the change, the quadtree is indistinguishable from a quadtree built from scratch over the new input.*

4.2 Choosing Steiner points

The prior section showed how to produce statically, and update dynamically, a size-conforming quadtree. This is not yet a mesh: it does not conform to the points themselves, only to the spacing function between points; it is also not triangular, which may be problematic for some users. Bern *et al.* described how to warp the points of the quadtree, and triangulate the result. However, quadtrees have many more points than are necessary: consider for instance that splitting a single cell may add as many as five points, as compared to circumcenter refinement à la Ruppert. The current section shows a conceptual algorithm, and proves it correct. In this section, I reach for the most general algorithm that builds a quality, size-conforming mesh; the hope is that by proving correct a very wide class of possible Steiner points, later authors (including myself) can generate an efficient algorithm that outputs very few points in practice. The class of Steiner points I describe encompasses both circumcenters (as per Ruppert's algorithm [Rup95]) and off-centers (as per Üngör [Üng04]), and many choices beyond those.

I start with some definitions. Recall that I will be producing a set of Steiner points whose Delaunay triangulation is a quality mesh. It is therefore quite natural to develop a fixation on d-dimensional spheres and the balls they enclose. Assume the existence of a finite set of points $\mathcal{P} \subset \mathbb{R}^d$, and a bounded domain $\Omega \subset \mathbb{R}^d$. I denote a ball b with its center at c and with radius r as $B(c, r)$. Unless clearly stated otherwise, this is an open ball. A point on the surface of b is not considered to be in the ball. I use two special types of balls, as follow:

Definition 4.2.1 (Gap ball [Tal97]) *A **gap ball** on a vertex v is any ball $b = B(c, r)$ such that (1) c is within the domain Ω, (2) v is on the surface of b, (3) there is no point in $\mathcal{P}$ that lies in the open ball b.*

Definition 4.2.2 (Circumball) *Given a d-simplex s composed of $d + 1$ points in general position, the circumball of s is the ball $b = B(c, r)$ such that every vertex of s lies on the surface of b (that is, b is the ball corresponding to the circumscribing sphere of s). Given instead an i-simplex s, $i < d$, the circumball is the ball whose center is on the affine plane of s and that has all $i + 1$ vertices on its surface. Equivalently, the circumball is the smallest ball that circumscribes s.*

Recall that Delaunay entitled his seminal paper *Sur la sphère vide* (On the empty sphere): a d-simplex is *Delaunay* if, and only if, its circumball is empty of any points in $\mathcal{P}$. Clearly, the circumball of a Delaunay simplex is a gap ball on each vertex of the simplex. An lower-dimensional simplex (a segment, or triangle in three dimensions, etc) is said to be Delaunay if, there is at least one empty ball that circumscribes the simplex. That empty ball is a gap ball on every vertex of the simplex. The circumball of a simplex is a particular circumscribing ball; a lower-dimensional simplex may be Delaunay even if its circumball is non-empty; however, if the circumball of a simplex is indeed empty, then the simplex is Delaunay.

Consider now the Voronoi cell of a vertex v which has nearest neighbour u. By the triangle inequality, it is clear that the circumball of the segment vu is empty. Indeed, this circumball is the smallest-radius gap ball around v. More generally, the center of any gap ball on v lies within the Voronoi cell of v. Therefore, a sufficient witness to the poor quality of the Voronoi cell of v is a pair of gap balls: one describing the distance to the

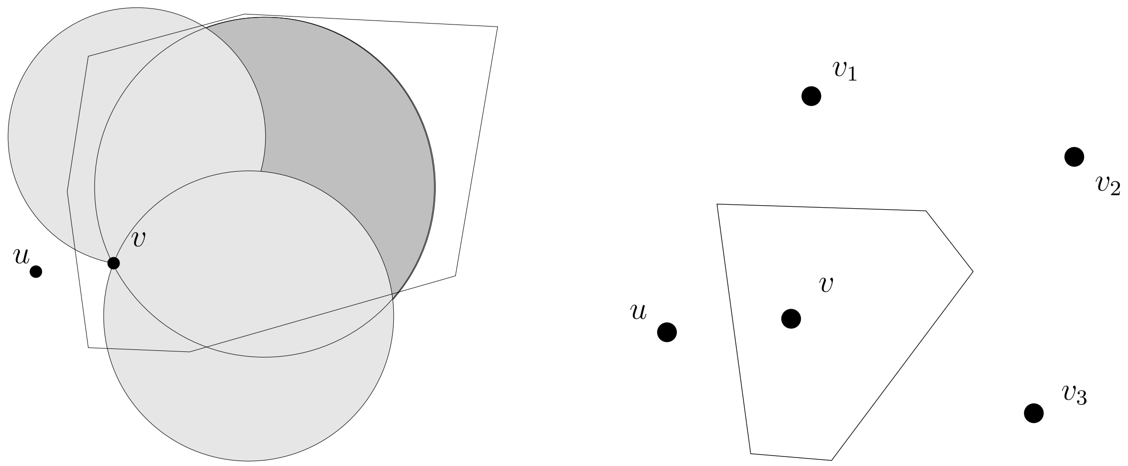


Figure 4.5: Completing v. **Left:** The vertex v has a bad aspect ratio Voronoi cell, as evidenced by the gap balls on v, which are much larger than the distance $||uv|| = \mathrm{NN}(v)$. The Voronoi cell could be arbitrarily larger than is shown. **Right:** After iteratively inserting v_1 through v_3 at the centers of large gaps, no large gap remains, and the Voronoi cell of v is of good quality.

nearest neighbour, and one with radius $r > \rho\,\mathrm{NN}(v)$ where ρ is the user-specified quality threshold. That is, we need not compute the Voronoi cell.

This produces an obvious algorithm: starting with the initial point set, look at each point in turn. If it has good quality, we can move on. If not, then there is at least one offensively large gap ball. Add the center of that ball, and check again, until the vertex is completely surrounded by points that hide any large gaps from it:

Definition 4.2.3 (Complete) *A vertex v is said to be **complete** if it has no gap ball of radius larger than $\rho\,\mathrm{NN}(v)$.*

The claim is that this procedure produces the quality mesh we would like to have. Clearly, the mesh respects the input point set. Equally clearly, the mesh is of good quality, except perhaps near the boundary of the mesh. For now, ignore the boundary effects; I will show in Section 4.4 that we can set up the domain such that it is indeed safe to ignore boundary effects. What is left is to show is that the mesh thusly created respects the local feature size (which also shows that the procedure does, in fact, terminate).[2]

[2]It is more natural to write the lemma as $\mathrm{NN}(v) \in \Omega(\mathrm{lfs}(v))$, because NN changes over time and is controlled by the algorithm, whereas lfs is fixed by the input. However, writing it in the form $\mathrm{lfs}(v) \in O(\mathrm{NN}(v))$ is more convenient in the proofs.

Lemma 4.2.4 *When the iterative routine chooses to insert an off-center v,*

$$\operatorname{lfs}(v) \in O(\operatorname{NN}(v))$$

Proof: The point v is being inserted in order to complete a mesh vertex u, whose nearest neighbour is u'. By definition, v is the center of an empty ball of radius at least $\rho\,\operatorname{NN}(u)$, on the surface of which lies u. This sets $\operatorname{NN}(v) \geq \rho\,\operatorname{NN}(u)$. If both u and u' are input points, then $\operatorname{lfs}(u) = \operatorname{NN}(u) \leq \operatorname{NN}(v)/\rho$. By the Lipschitz condition, $\operatorname{lfs}(v) \leq \operatorname{lfs}(u) + ||uv||$, into which we substitute the prior bounds to conclude $\operatorname{lfs}(v) \leq \operatorname{NN}(v)\frac{\rho+1}{\rho}$. Thus, for $\boxed{c \geq \frac{\rho+1}{\rho}}$ the statement holds.

Otherwise, let U be the newer of u and u'. By induction, we know that when U was inserted, $\operatorname{lfs}(U) \leq c\,\operatorname{NN}(U)$, and that at the time, $\operatorname{NN}(U) \leq ||uu'||$. Again appealing to the Lipschitz condition, $\operatorname{lfs}(v) \leq \operatorname{lfs}(U) + ||Uv||$. The distance from U to v is at most $||uv|| + ||uu'||$, so again we can substitute to get $(\frac{1+c}{\rho} + 1)\,\operatorname{NN}(v)$. Thus, for $c \geq (\frac{1+c}{\rho} + 1)$ the statement holds. This evaluates to $\boxed{c \geq \frac{\rho+1}{\rho-1}}$. The two boxed constraints are simultaneously satisfiable for any $\rho > 1$. ■

Theorem 4.2.5 (Completing is size-conforming) *There exists a constant c_{hi} that depends only on the dimension and on ρ such that after completing the mesh until no incomplete vertex is left, for every point x in the Delaunay d-simplex τ_x that contains x,*

$$\operatorname{lms}(x) \leq \operatorname{lfs}(x) \leq c_{hi}\operatorname{lms}(x)$$

Proof:

All points of the input are in the output, so the lower bound is obvious. The upper bound is implied by Lemma 4.2.4. Given an arbitrary point x, consider its nearest neighbour v, which is a vertex of the mesh (see Figure 4.6). In other words, x lies in the Voronoi cell of v. When x is on the boundary of the Voronoi cell, it is equidistant between v and a second point. Therefore, the smallest distance that x can have between it and its second-nearest point is $\operatorname{lms}(x) \geq \operatorname{NN}(v)/2$. By the Lipschitz condition, we know that $\operatorname{lfs}(x) \leq \operatorname{lfs}(v) + ||xv||$. Because the Voronoi cell of v is of good quality, $||xv|| \leq \rho\,\operatorname{NN}(v) \leq 2\rho\operatorname{lms}(x)$. The first term is an inductive argument:

Let u be the nearest neighbour of v. If v was inserted after u, then by Lemma 4.2.4, we know that $\operatorname{lfs}(v) \leq c\,\operatorname{NN}(v)$. On the other hand, if u was inserted after v, then we only know that at the time u was inserted, $\operatorname{lfs}(u) \leq c\,\operatorname{NN}(u)$. But v was a neighbour of u at that time, so $\operatorname{lfs}(u) \leq c||uv|| = c\,\operatorname{NN}(v)$. This allows us to use the Lipschitz condition to compute the local feature size at v: $\operatorname{lfs}(v) \leq \operatorname{lfs}(u) + \operatorname{NN}(v) \leq (1 + c)\,\operatorname{NN}(v)$.

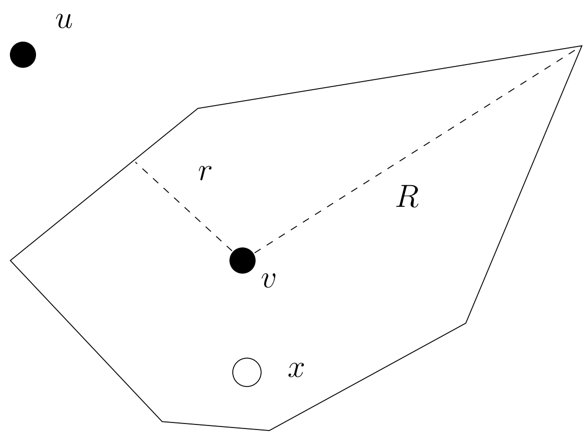


Figure 4.6: Illustration of the proof of Theorem 4.2.5. The hollow point x is an arbitrary point in space; it lies in the Voronoi cell of a mesh vertex v, whose nearest neighbour is u. The cell of v has out-radius R and in-radius r.

Substituting back in to $\mathrm{lfs}(x) \leq \mathrm{lfs}(v) + ||xv||$ we get that there exists a constant c_{hi}, namely $2(1 + c + \rho)$, such that $\mathrm{lfs}(x) \leq c_{hi} \,\mathrm{lms}(x)$. ∎

4.3 Efficient algorithm

Implementing the conceptual algorithm requires efficiently finding large empty gaps. The key to efficiency will be to work in a bottom-up fashion: we complete vertices in (approximate) order of their distance to their nearest neighbour, shortest distance first. Lemma 4.3.1 shows that work only progresses: when a new vertex v is created while completing a vertex u, $\mathrm{NN}(v)$ is substantially larger than $\mathrm{NN}(u)$. Thanks to this progress property, I can prove that when the algorithm computes a gap ball, the gap ball only intersects a bounded number of the cells of the quadtree constructed by BUILDQT. This shows that the postprocess to turn a graded quadtree into a quality Delaunay mesh takes only linear time in the size of the output. Philosophically, this makes sense: building the quadtree is analogous to sorting, and this postprocess is simply a filter process over the sorted structure.

The main loop invariant this algorithm maintains is that every vertex with nearby nearest neighbour is complete. The priority queue is ordered by the nearest neighbour, smallest first. The following shows that work in one bucket only creates work in a strictly later bucket. Therefore, once a bucket relating to length range $[l, \rho l)$ has been fully processed, every vertex that has a nearest neighbour closer than ρl is complete.

CHOOSESTEINERS($\mathcal{P}$, QT, P, ρ, γ)

$\mathcal{P}$: set of points in $\mathbb{R}^d$
QT: graded quadtree on $\mathcal{P}$, such as from BUILDQT
P: map storing the correspondence between quadtree cells and vertices
Q: work queue, bucketed with factor ρ
for each $p \in \mathcal{P}$ **do**
    let c be the quadtree cell that contains p
    add a COMPUTENN(p) event, with key $|c|/2$, to Q
end for
while Q not empty **do**
    $w \leftarrow$ pop(Q)
    if w is a COMPUTENN event on v **then**
        Compute the exact nearest neighbour u (Steiner or input) to v
        if $||uv|| \leq \hat{\mathrm{NN}}(v)$ **then**
            add a COMPLETE(v) event, with key $||uv||$, to Q
        end if
    else if w is a COMPLETE event on v **then**
        while $\exists$ a gap ball $b = B(x, r)$ with $r \in (\rho\,\mathrm{NN}(v), \gamma\,\mathrm{NN}(v))$ **do**
            Let c_x be the cell that contains x
            Add $c_x \leftrightarrow x$ to P
            Add COMPLETE(x), with key $||xv||$, to Q
        end while
    end if
end while

Figure 4.7: Static algorithm to compute Steiner points, given a graded quadtree. The parameter ρ must be strictly greater than 1. The parameter γ affects runtime; it must be at least as large as ρ.

Lemma 4.3.1 (Completing only leaves large incompletes) *Consider an incomplete vertex v with nearest neighbour u. While working to complete v, any new vertex v' thus created has a nearest neighbour (namely, v) at distance at least $||vv'|| \geq \rho||uv||$.*

Proof: The vertex v' is the center of a ball that has v on its surface, which proves that v is the nearest neighbour to v'. The radius of that ball is, by construction, at least $\rho||uv||$. ■

Given the loop invariant, we can now prove that both primitive operations of the CHOOSESTEINERS algorithm (namely, computing the nearest neighbour, and finding a large gap ball) can be implemented in constant time. Computing the nearest neighbour for a COMPUTENN event can be done using Dijkstra's algorithm. Nodes of the search graph are quadtree cells and mesh vertices, and the distance is the Euclidean distance from the object to v; there is an edge in the search graph from a cell to its neighbours, and from a cell to the mesh vertices it contains. This search is essentially a sweep, growing a circle out of v, which stops upon reaching a mesh vertex. The runtime is thus regulated by the number of quadtree cells the final query circle intersects. Computing a gap ball of radius $r \in (\rho\,\mathrm{NN}(v), \gamma\,\mathrm{NN}(v))$ during COMPLETE can be implemented using a similar set of searches. In either case, the query comes down to being an empty circle with radius at most $O(\mathrm{NN}(v))$.

Lemma 4.3.2 *In a partial mesh where all vertices with nearest neighbour close than l are complete, for any point $x \in \Omega$, if* $\mathrm{lfs}(x) < l/2$*, then x lies in the Voronoi cell of a complete vertex.*

Proof: I start with a simple observation: at any point $y \in \Omega$, there is at least one vertex within distance $\mathrm{lfs}(y)$ of y (in fact, there are at least two); we can write this as $\mathrm{NN}(y) \leq \mathrm{lfs}(y)$. Let v be the vertex that has x in its Voronoi cell. The Lipschitz condition says that $\mathrm{lfs}(v) \leq \mathrm{lfs}(x) + |xv|$. But $|xv| = \mathrm{NN}(x) \leq \mathrm{lfs}(x)$, so we have $\mathrm{lfs}(v) \leq 2\,\mathrm{lfs}(x) < l$. Finally, $\mathrm{NN}(v) \leq \mathrm{lfs}(v) < l$. Therefore, v is complete. ■

Lemma 4.3.3 (Gap searches are fast) *In a partial mesh where all Delaunay edges shorter than l are complete, any gap ball on a vertex v with nearest neighbour* $\mathrm{NN}(v) \in \Theta(l)$*, and with radius $r \in \Theta(l)$, intersects at most $O(1)$ quad-tree cells.*

Proof: Consider the cell, c_v, that contains v. By the size-conforming theorem (4.1.4), the cell has size $\Theta(\mathrm{NN}(v)) = \Theta(l)$, as do its neighbours. A cell c takes up volume $|c|^d$, so the gap ball can only intersect a constant number of cells of size $\Theta(|c_v|)$. In other words, if the gap ball is to intersect a large number of cells, it must intersect some small cells. For a contradiction, assume the gap ball does, indeed, intersect a small cell, of size less $\epsilon|c_v|$ for some ϵ to be described later. Let c_- be one such cell, and let x be a point common to c_- and the gap ball. Walk along the ray from x to v, stopping upon crossing k cells (for a particular constant k to be described), and letting the stopping point be called y. Then there is an empty ball centered at y that intersects at least $k + 1$ cells.

The last cell visited in this walk must have size at most $2^k|c_-|$ since the quadtree was well-graded. The local feature size at y is no greater than $c_{hi}2^k|c_-|$; that is, there is a Delaunay edge in the input with length at most $c_{hi}2^{k+1}|c_-|$. Using $\epsilon = \frac{c_{hi}}{c_{low}}2^{k+1}$ ensures that this length is strictly less than l, which means that the algorithm already completed the edge. But then y must lie in a Voronoi cell of good aspect ratio, as proved above. Therefore, the empty ball centered at y, which we previously proved intersected $k + 1$ cells, is contained in the union of circumballs of complete simplices. Completed simplices are size-conforming (they will be in the output); and since quad-tree cells also are, the empty ball centered at y can only be intersecting a constant number of quad-tree cells. Set k to that constant. Then we have a contradiction, which leads us to the conclusion that all cells that the gap ball on v with radius $\Theta(l)$ intersects only quadtree cells of size $\Theta(l)$, and thus only intersects a constant number of them. ■

Finally, recall that in a quality radius/edge mesh, every vertex has bounded degree [MTTW95]. This proves that the loop in a COMPLETE(v) event has a bounded number of iterations. Therefore, every event is processed in constant time.

4.3.1 Delaunizing

The CHOOSESTEINERS algorithm computes a set of points and never explicitly computes their Delaunay triangulation. However, a simplex constructed of complete vertices is never destroyed. Therefore, we can simply note this and output the simplex immediately at that time; every simplex of the output is reported since every simplex in the output is made up of complete vertices. Alternatively, as proposed by Har-Peled and Üngör, we can run a third pass that does explicitly compute the Delaunay: for each output vertex (Steiner or input), compute its set of Delaunay balls. Since every vertex is complete, every search will be a gap of radius $r < \rho\,\mathrm{NN}(v)$, which is constant time. Since every vertex in the output has bounded degree, it takes $O(1)$ time per vertex, or $O(m) \subset O(n \lg L/s)$ total time.

4.3.2 Static runtime

Theorem 4.3.4 *After running* BUILDQT, CHOOSESTEINERS, *and computing the Delaunay triangulation of the output points, we have constructed a mesh that respects the input, has radius/edge quality no worse than ρ, and is size-conforming. Furthermore, running these algorithms takes a total of $O(n \lg L/s)$ time.*

Proof: The output points include the input points, so the mesh trivially respects the input. Upon completing all the vertices, we have a quality mesh, modulo boundary concerns to be addressed in Section 4.4. Lemma 4.2.5 shows that the output is also size-conforming. Computing the quadtree is the bulk of the runtime, at $O(n \lg L/s)$ time. Computing the Steiner points and the Delaunay triangulation takes only time linear in the output size, which is $O(n \lg L/s)$ in the worst case, but much smaller in common cases. ■

4.3.3 Dynamic stability

To establish dynamic stability of CHOOSESTEINERS, I use the same argument as in the section on BUILDQT: namely, when a Steiner point u is added to the mesh, it is added to complete a vertex v. Naturally, u *blames* v. However, the coordinates of u also depend on a few other vertices, in particular the nearest neighbour of v, and any other vertices on the surface of the gap ball whose center defines u. Finally, while completing, the search will iterate over $O(1)$ quadtree cells, which may have changed due to changes in the input, so we must blame those as well.

In the BUILDQT section, the argument was that every step of blame cut a characteristic length in half, and thus if a quadtree cell blamed a point p, the cell had characteristic approximately equal to the distance from p. The situation is slightly more complicated now, because the vertices on the surface of the gap ball may have characteristic as large as that of v. Indeed, we can construct an example (see Figure 4.8) where the characteristic does not grow, and geometric information can leak arbitrarily far away. Spielman, Teng, and Üngör show how to sidestep this issue by colouring the work queue in linear work and constant parallel depth in their parallel algorithms [STÜ07, STÜ04] (they call it computing a maximal independent set, but this is a misnomer as the sets are large, but not maximal). The CHOOSESTEINERS algorithm modified to use colouring is in Figure 4.10.

Work in CHOOSESTEINERS is ordered according to a bucketed priority queue, with bucketing factor ρ. Two vertices in the bucket $[l, \rho l)$ are independent if we are sure they can be completed in parallel without the one affecting the other. This happens if the off-centers of one are not contained within the gap balls of the other. Recall that the gap balls

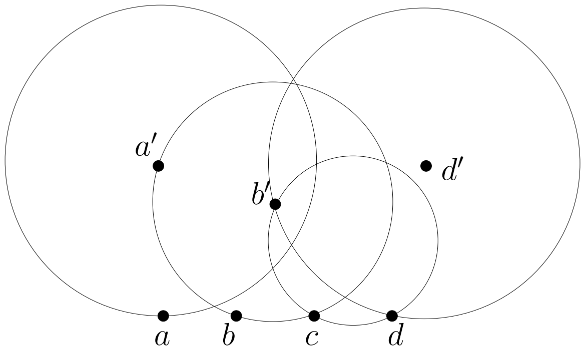


Figure 4.8: An illustration of a long chain of dependency. The input is n (here, $n = 4$) equally-spaced points on a line. The first vertex to be processed, a, adds an off-center a' at distance 2ρ directly above itself. Its neighbour b, casting a similar gap ball, runs into a' and thus creates a b' somewhat off from the vertical. Vertex c casts a gap ball and finds it to have small radius, and thus adds no off-center. The chain continues with d force to add d' slightly off from the vertical. Clearly, the dependency can propagate arbitrarily far. However, it would be safe to add a' and a hypothetical d'' at distance 2ρ directly above d: a and d are in a sense *independent*. Explicitly exploiting the independence allows breaking artificially long chains of dependency.

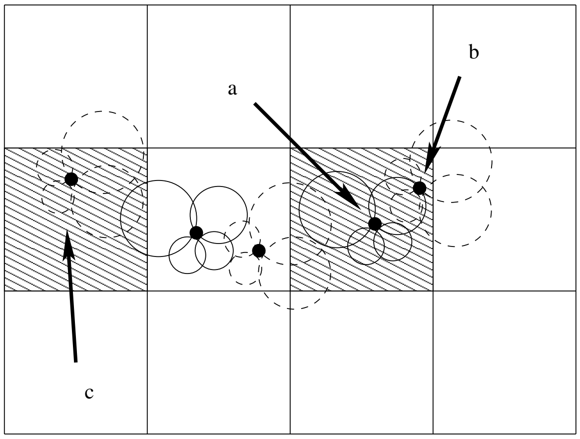


Figure 4.9: The grid used to determine independence. Vertices that lie in a shaded square cannot conflict with vertices lying in a separate shaded square. Only $O(1)$ vertices lie in any given square, so they can be processed serially. After cycling through all 3^d ways of regularly shading squares, every vertex has been assigned an order in which it will be processed.

CHOOSESTEINERS($\mathcal{P}$, QT, P, ρ)

$\mathcal{P}$: set of points in $\mathbb{R}^d$
QT: graded quadtree on $\mathcal{P}$, such as from BUILDQT
P: map storing the correspondence between quadtree cells and vertices
Q: work queue, bucketed with factor ρ
for each $p \in \mathcal{P}$ **do**
    let c be the quadtree cell that contains p
    add a COMPUTENN(p) event, with key $|c|/2$, to Q
end for
while Q not empty **do**
    Let W be the set of smallest items on Q
    if There are any COMPUTENN items in W **then**
        for each COMPUTENN(v) item in W **do**
            Compute the exact nearest neighbour u (Steiner or input) to v
            add a COMPLETE(v) event, with key $||uv||$, to Q
        end for
        loop again
    end if
    Let G be a grid on $[0, L]^d$ with grid elements of size $3\gamma\rho$
    for each COMPLETE(v) item in W **do**
        Add a pointer to v from the grid element of G that contains v
    end for
    Use G to colour W using Δ colours
    for each colour $i \in [1 \ldots \Delta]$ **do**
        for each element COMPLETE(v) of W with colour i **do**
            Complete v
        end for
    end for
end while

Figure 4.10: Dynamically-stable algorithm to compute Steiner points, given a graded quadtree. Unlike in the static algorithm, we now need to carefully sequence the COMPLETE(v) events within a bucket in order to avoid long chains of dependency.

of a vertex v have radius at most $\gamma\,\mathrm{NN}(v) < \gamma\rho l$. Therefore, if u and v are at distance $3\gamma\rho l$, they must be independent. The task now is to colour the vertices; each colour is an independent set.

To compute the colouring, make a grid with side length $3\gamma\rho l$. Note that this grid is unrelated to the quadtree. A vertex v at a point $\langle x_1, \ldots, x_d \rangle$ is assigned to the grid square $\langle \ldots, \lfloor x_i/(3\gamma\rho l) \rfloor, \ldots \rangle$. It would be a bad idea to create the grid explicitly; instead, we can use a hash table to store only the non-empty grid squares. If there are n_l vertices in the bucket, then this takes expected time $O(n_l)$.

Each grid square will have at most a constant k vertices within it (I prove this in Lemma 4.3.5). In a given grid square, it may or may not be possible to process two vertices in parallel; pessimistically, the colouring will use k colours in each grid square. Adjacent grid squares may also have non-independent vertices, so if two grid squares are adjacent, they must be coloured at different times. However, if we evenly sample every other (in L_∞ distance) square from the grid, all vertices in one sampled square s are independent of all those in another sampled square s': every point in s is at least $3\gamma\rho l$ from every point in s' (see Figure 4.9). Colour the vertices in this sample, using up to k colours. Then shift the sample by one square, and repeat. Each shift takes k colours; we only need 2^d shifts to ensure that every square has been sampled.

Lemma 4.3.5 *Each grid element contains at most a constant number of vertices.*

Proof: The volume of a grid element while processing a bucket of size $[l, \rho l)$ is $(3\gamma\rho l)^d$. Every point being coloured has nearest neighbour $\mathrm{NN}(v) \geq l$ and thus has an empty ball around it of volume $V_d(l/2)^d$, of which at least a 2^{-d} fraction must lie within the grid element. Thus, the constant is no more than $(12\gamma\rho)^d V_d$ where V_d is the volume of the unit sphere in dimension d. This proof is loose. ■

The ordering chosen by the colouring is congruent with an ordering of the work items in a given bucket, and the prior proofs did not depend on the ordering within buckets, so the algorithm remains correct. Colouring takes linear time: every vertex in the bucket $[l, \rho l)$ is inspected once, and every non-empty square is inspected once. There are no more non-empty squares than vertices. This means the static runtime is unmodified from before: $O(m)$ time to choose the Steiner points. What remains to be proved is the dynamic stability bound. The blame argument can now be articulated fully:

Lemma 4.3.6 (Blame within rounds packs) *Consider two Steiner points u and v created while processing events with nearest neighbour in $[l, \rho l)$. If v blames u, then $||uv|| \in O(l)$.*

Proof: If, at some point during the `while` loop, the gap balls of u intersect those of v, then the claim is obvious since the gap balls have radius $O(l)$. If not, then there is a path $v = v_0, v_1, \ldots, v_k = u$ of blame, whose gap balls intersect pair-wise. Any two adjacent vertices v_i and v_{i+1} are within $O(l)$ of each other. By the construction of independent sets, k is a constant. Therefore, the distance from v to u is at total of at most $O(l)$. ■

Lemma 4.3.7 (Blame across rounds packs) *If, while processing events with nearest neighbour in $[l, \rho l)$, the algorithm creates a Steiner point v, and v blames a vertex u inserted when processing smaller events, then $||uv|| \in O(l)$.*

Proof: If v blames u directly, then u is on the gap ball that defines v. The distance from v to u in this case is at most $O(\rho^2 l)$. Otherwise, blame is via a chain $v = v_0, v_1, \ldots, v_k = u$. Among a set of v_i inserted while processing the same bucket $[l', \rho l')$, we just saw that the distance is at most $O(l')$. We can therefore compress the chain to $v = v'_0, v'_1, \ldots, v'_{k'} = u$ where each vertex is in a different bucket. The distance from v'_i, processed in bucket $[l', \rho l')$, to v'_{i+1} processed in a smaller bucket is $O(l') \subset O(l\rho^{-i})$. The total distance from v to u, then, is the convergent sum $\sum_i O(l\rho^{-i}) \subset O(l)$. ■

4.3.4 Main theorem

Theorem 4.3.8 (Dynamic Stability of ChooseSteiners) *Under single-point additions and removals, the* ChooseSteiners *algorithm, composed with the* BuildQT *algorithm, is $O(\lg L/s)$ dynamically stable. This leads to an algorithm that responds to changes in the same time.*

Proof: If a Steiner point v blames an input point p, then by Lemma 4.3.7, the distance $||vp|| \in \Theta(\mathrm{NN}(v))$. Therefore, only $O(1)$ Steiner points created in each bucket blame p due to chains of off-centers. We already know from Theorem 4.1.11 that only $O(1)$ quadtree cells of any given size blame p; therefore, only $O(1)$ Steiner points in each bucket blame p due to changes in the quadtree. Given there are $O(\lg L/s)$ buckets, this bounds the total number of Steiner points that blame p. Finally, each Steiner point has $O(1)$ simplices in the final Delaunay mesh since it is of good quality; only $O(\lg L/s)$ triangles will change.

Monotonicity is ensured by the standard arguments. The change propagation queue is slightly more complicated. ComputeNN events can be bucketed by ρ, and LIFO within a bucket, to properly maintain the order of operations. The processing of Complete events is in two parts: the first must be bucketed by ρ and LIFO, to determine the colouring. The second must additionally be bucketed by the colour. This still yields a constant-time priority queue. ■

4.4 Bounding box to ignore boundary effects

With the provable algorithm in hand, I can now explain how to choose an appropriate meshing domain for the input. The bounding shape I propose is a box, $[0, L]^d$, with the faces split into N_{box}^{d-1} grids; there are thus $N_{box}^d - (N_{box} - 2)^d$ vertices defining the box. The input will be restricted to lie in a concentric box with side length l. The values N_{box} and l are functions of each other, and of ρ and the dimension.

We mentioned above that a complete mesh was a quality mesh, up to boundary effects. Miller, Pav, and Walkington [MPW02] proved that in a mesh with no encroached boundary, the circumcenter of every simplex lay inside the domain. Under this condition, then, any simplex with bad radius/edge ratio would have an associated gap ball of large radius; conversely, if the mesh is complete but there is a bad radius/edge ratio simplex, it must be that the domain boundary is encroached. Thus, it suffices to choose N_{box} and l such that at all points during the algorithm, no matter the order in which we choose to complete vertices, the boundary remains unencroached. Clearly, l must be small enough that none of the points in the input encroach the boundary. But this must also be true for the off-centers of input vertices, and recursively for the off-centers of Steiner vertices. Generally, increasing N_{box} increases l.

Lemma 4.4.1 (We can ignore boundary effects) *Given used-defined constants l, N_{box}, and ρ that satisfy $l \le (1 - \frac{2\sqrt{d-1}}{N_{box}-1}(\frac{\sqrt{2}}{\rho-1} + 1))L$, no off-center generated while completing the mesh encroaches upon the boundary box.*

Proof: If a vertex v encroaches upon the boundary, it lies in the circumball of an i-simplex τ, with $i < d$. That is, the distance from v to the boundary can be no greater than $r(\tau)$. This simplex is composed from $i + 1$ points on a regular grid with spacing $L/(N_{box} - 1)$, so $r(\tau) = \sqrt{i}L/(N_{box} - 1)$. At most, $i = d - 1$, and $r(\tau) = \frac{L\sqrt{d-1}}{N_{box}-1}$.

Assuming it is a Steiner vertex v_0 that encroaches, then it was created as the off-center of some v_1. The nearest neighbour of v_0 is, therefore, v_1. Furthermore, $\mathrm{NN}(v_0) \ge \rho\,\mathrm{NN}(v_1)$. In turn, v_1 may be input or Steiner. Since our goal is to calculate how far from the boundary the input must lie, in the worst case, v_1 is a Steiner created recursively by a vertex v_2. This defines a chain v_0, v_1, $\ldots$, v_∞. For any v_i in the chain, we have $\mathrm{NN}(v_i) \ge \rho\,\mathrm{NN}(v_{i+1})$, which unrolls to $\mathrm{NN}(v_i) \le \rho^{-i}\,\mathrm{NN}(v_0)$. We also have that $||v_i v_{i+1}|| = \mathrm{NN}(v_i)$. The distance from v_i to the boundary is thus at most $||v_i v_0||$ plus the distance from v_0 to the boundary, which is at most $r(\tau)$. Calculating the sum to v_∞ shows that the input can only cause encroachment on the boundary if $(L - l)/2 < \sum_{i=1}^{\infty} \rho^{-i}\,\mathrm{NN}(v_0) + r(\tau)$. Finally, a proof by Shewchuk [She98b, Lemma 1] shows that if v_0 encroaches any simplex of the boundary, then there is a boundary vertex p such that $\mathrm{NN}(v_0) \leq ||v_0 p|| \leq \sqrt{2}r(\tau)$. Putting it all together, we see that it suffices to satisfy $\frac{L-l}{2} \geq (\frac{\sqrt{2}}{\rho-1} + 1)r(\tau)$, where $r(\tau) = \frac{\sqrt{d-1}L}{N_{box}-1}$, to ensure that no input vertex or Steiner point can ever encroach upon the boundary. ■

We are now free to optimize l and N_{box} at will for any given ρ. One reasonable setting has the length of a side in the bounding box be the same as l: that is, $L/(N_{box}-1) = l$. For $\rho = 2$, this means that $N_{box} = 7$ satisfies the equations in two dimensions, and $N_{box} = 9$ in three dimensions. The former yields a box with 24 points on it, while the latter has 386 points. This validates the assumption made much earlier in this chapter that we can ignore boundary effects.

Chapter 5

Handling Input Segments

In the previous chapter, we saw how to maintain a mesh over a dynamically changing point cloud. This is likely to be sufficient for *e.g.* astrophysics simulations, but many applications will have additional features, such as an airfoil (in aeronautics), mountain ranges (in meteorology), reactor walls (in nuclear engineering), faults (in geology), and so on, which must appear in the output mesh. The current chapter shows how to extend the point-cloud algorithm to the case where we have constraints that are linear segments.

The input description is as follows: all input lies in a box $[(L-l)/2, (L+l)/2]^d$, plus the corners of a grid as described in Section 4.4. Input points are given as an index and a point. Input segments are given as a pair of point indices (not coordinates). For my proofs, I require that the segments meet at non-acute angles ($90°$ or more). Input segments must not intersect, except for meeting at common endpoints; similarly, input segments must not intersect input points that are not their endpoints. I do not require that segments form a convex, manifold, or even a connected shape. Similarly, points may be isolated in space and need not be the endpoint of a segment. The algorithm will add Steiner points that lie either exactly on a segment, or in the ambient space. The *containing dimension* of a point is 0 if it is an input point, 1 if it lies on a segment, or d if it lies in ambient space.

As before, the algorithm first produces a size-conforming quadtree, then chooses locations for Steiner points. The local feature size is now defined (following Ruppert) as the distance between two features, points or segments, which are disjoint — that is, they do not share an endpoint. In order to produce a mesh that *respects* the segments of the input, I will ensure that the Delaunay triangulation of the Steiner points includes the segments as a collection of Delaunay edges, possibly adding points on the segment as needed. On point-cloud inputs, the Voronoi aspect ratio quality that we may demand is any ρ strictly greater than one. When segments are involved, the best quality we can provably guarantee is $\rho > 2$.

The updates I will allow are now of two types: the user can add or remove a new segment; or the user can add or remove a point that is not the endpoint of any segment. To remove a point that is an endpoint, the user must first remove every segment on the point. As before, the user may not change L; also, the user may not move a point without removing all its segments, removing the point, re-adding it in its new location, and re-adding all the segments. Clearly, any segment being added must not cross another segment, and any new segment must be at a non-acute angle with existing segments — otherwise, the input would be illegal.

The dynamization argument will be almost verbatim from the prior section. Building the quadtree is only slightly complicated by the presence of segments. Choosing Steiner points is more strongly affected, but fundamentally the dynamization argument for point clouds rested on the fact that a vertex v with nearest neighbour $\mathrm{NN}(v)$ only created new Steiner points with nearest neighbour $\rho\,\mathrm{NN}(v)$, and thus (combined with the colouring argument), only $O(1)$ Steiner points of any given size could be affected by an input point p. I will show that in the presence of segments, a vertex v only creates new Steiner points with nearest neighbour $\frac{\rho}{2}\,\mathrm{NN}(v)$, with one exception where I need to argue with one additional indirection: v creates a Steiner point that creates a Steiner point with a distant nearest neighbour.

A critical difference is that a new segment may change the local feature size over a large area. Certainly, no history-independent dynamic algorithm can update in time faster than linear in the number of points, call it m_f, that an optimal mesh puts onto the new segment. No matter where the algorithm places its Steiner points, the adversary can place a facet such that it does not pass through the current set of points and thereby require at least m_f work: dropping the history independence condition can only save us a factor of two (we may be able to ignore deletions), which shows that no dynamic algorithm can respond faster than in $\Omega(m_f)$ time. I will show that the algorithm of this section updates in $O(m_f \lg L/s)$ time, a logarithmic factor off optimal.

As before, the overall algorithm follows the stencil:

REFINEPSLG(L, $\mathcal{P}$: points in $[0, L]^d$, $\mathcal{S}$: segments, ρ: quality bound)

1: QT $\leftarrow$ BUILDQT$(L, \mathcal{P}, \mathcal{S})$
2: $P' \leftarrow$ CHOOSESTEINERS(QT$, \rho)$
3: $M \leftarrow$ TRIANGULATE$(\mathcal{P} \cup P', T)$

5.1 Building the quad-tree

Building the quadtree (see Figure 5.1) to be lfs_1-conforming is very similar to building an lfs_0-conforming quadtree. I add one mapping, from quadtree cells to the segments that cross that cell; and I generalize the definition of crowding. Recall that lfs_1 is defined as the smallest ball centered at a given point p that touches a pair of non-intersecting features (segments or points); therefore, I split a cell if it intersects such a pair. Of course, as was the case with points, this is insufficient if features happen to lie within ϵ of a boundary between two cells. Thus I check that two neighbouring cells do not between them contain two non-intersecting features.

5.1.1 Analysis

I now proceed to prove certain properties about the quadtree the algorithm constructs. Namely: the quadtree is graded and size-conforming. Furthermore, each split takes constant time except for relocating the features, and each feature is relocated a bounded number of times, which yields the desired runtime. Finally, any cell c being split is at distance $O(|c|)$ from any feature f it blames, which yields the dynamic stability bound. The proof of size conformality is identical to that of Theorem 4.1.4. The runtime and dynamic stability bounds need the following proofs:

Lemma 5.1.1 (Constant-time splits) *Barring Lines 12–17, each iteration of the "while" loop in* BUILDQT *takes* $O(1)$ *time.*

Proof: Checking whether c is a leaf cell is a constant-time operation. There are $O(1)$ neighbours of the cell c being split in the while loop. Therefore, checking them for grading is $O(1)$ time. Similarly, there are a constant number of new children; creating them and linking them in will take constant time. Finally, the doubly-nested "for" loop is over a constant number of children, times a constant number of neighbours, and thus is a constant number of iterations.

The sets $\hat{P}$ and $\hat{S}$ may be larger than constant size. However, they are only used in a single conditional; if they are large, then the value of the conditional is true and we can short-cut execution. Conversely, if the conditional is false, the sets were small. This is obvious in the test $\hat{P} \geq 2$. To test whether all segments in $\hat{S}$ share an endpoint, we merely keep track of the intersection of the sets of endpoints of $\hat{S}$. As soon as that set is empty, we can stop and answer the conditional affirmatively. Every vertex in the PLC has degree at most 2^d because segments must be at least orthogonal to each other, so we will find an

BUILDQT($L, \mathcal{P}, \mathcal{S}$)

$T \leftarrow [0, L]^d$
$Q_{\text{qt}} \leftarrow [0, L]^d$
initialize P to hold pointers from $[0, L]^d$ to every $p \in \mathcal{P}$
initialize S to hold pointers from $[0, L]^d$ to every $s \in \mathcal{S}$
while Q_{qt} not empty **do**
    $c \leftarrow$ POP(Q_{qt})
    if c is not a leaf cell, skip
    for each neighbour c' of c **do**
        if $|c'| = 2|c|$ then add c' to Q_{qt}
    end for
    Split c into 2^d children
    for each $p \in P(c)$ **do**
        Add p to the P entry of the unique child that contains p
    end for
    for each $s \in S(c)$ **do**
        Add s to the S entry of every child that intersects s
    end for
    for each child c^- of c **do**
        for each neighbour c' of c^- **do**
            $\hat{P} \leftarrow P(c^-) \cup P(c')$
            $\hat{S} \leftarrow S(c^-) \cup S(c')$
            if $|\hat{P}| \geq 2$
            or $|\hat{P}| = 1$ and it is not an endpoint of S
            or not all segments in $\hat{S}$ share a common endpoint **then**
                add both c^- and c' to Q_{qt}
            end if
        end for
    end for
end while

Figure 5.1: The algorithm to build a size-conforming quad-tree. The splitting rules are expanded to split if two disjoint features lie in neighbouring cells (two points, a point and a segment, or two segments).

empty set as soon as we see at most twice that many segments (twice, since cells c^- and c' may both intersect any single segment). If $\hat{P}$ contains exactly one element, intersect $\hat{P}$ with the set of endpoints, again in constant time. ■

To make the analysis easier, I define a set of **pseudo-vertices** on each segment. If a segment f intersects fewer than κ cells, where κ is any constant at least 2, the pseudo-vertices are just the two endpoints and the edge is considered to be "short". If f intersects more cells, the edge is considered "long" and I split it: conceptually create a pseudo-vertex at the midpoint, and split f into subsegments, f' and f'', at the midpoint of f. Both subsegments will be barely short: after only a constant number of further splits that affect f', f' will become long (and similarly f''). The set of midpoints forms a well-graded, lfs-conforming mesh of the segment. Remember, however, that these are not real vertices: I never construct them in the algorithm — only in the analysis.

Lemma 5.1.2 (Query structure update costs) *Lines 12–17 run $O(\lg L/s)$ times per input point, and $O(\lg L/s + m_f)$ times for segment number i, where $m_f = \int_f \mathrm{lfs}^{-1}(x)dx$ is the number of points needed to size-resolve the segment f.*

Proof: At any step in the algorithm, an input point p lies in exactly one cell, c. If c is later split, p is relocated to c', which takes $O(1)$ geometric tests since c only has 2^d children. The size of c' is exactly half that of c, so this can only occur logarithmically many times, with the numerator being $O(L)$ and the denominator, because the quadtree conforms to local feature size, no smaller than $\Omega(s)$.

If a short segment f intersects only one cell, it behaves exactly as a point. Indeed, if f intersects only κ or fewer cells, then all the cells that f intersects are within a factor 2^κ in size of each other because the quadtree is graded. Therefore, we can meaningfully identify with f a characteristic size: the size of the smallest cell it intersects. After splitting κ cells that intersect f, the characteristic must have fallen by at least half. Any segment that intersects κ cells in the final output is therefore only relocated $O(\lg L/s)$ times.

To account for long segments, note that any segment is short at the beginning of the algorithm. Account for it as above until f intersects more than κ cells. In other words, the initial location of f within the mesh costs $O(\lg L/s)$ relocations. Afterwards, notice that the subsegment f' will be relocated $O(1)$ times before being conceptually split again. Every midpoint is associated with two subsegments, each of which is relocated $O(1)$ times, so the number of relocations is $O(\lg L/s + m_f)$ in total. ■

Note also the following: Uncrowded but ill-graded cells will have only zero or one vertex inside, and zero to 2^d segments inside — otherwise, they would be crowded.

Theorem 5.1.3 BUILDQT *runs in time* $O(n \lg L/s + m)$ *to produce a size-conforming graded quad-tree.*

Proof: The quadtree conforms to the local feature size (Theorem 4.1.4), so the number of splits is $O(m)$. Each split takes $O(1)$ time except for maintaining P and S (Lemma 5.1.1). The maintenance cost is $O(\lg L/s + m_f)$ for each segment or input point (with $m_f = 0$ for points), which sums to $O(n \lg L/s + \sum_f m_f) \subseteq O(n \lg L/s + m)$. ■

Theorem 5.1.4 *After adding or removing a feature from the input,* BUILDQT *can respond in time* $O(m_f \lg L/s)$.

Proof: For a point feature, this was proved by Lemma 4.1.11. For a segment feature, it is easy to see that Lemma 4.1.10 applies: any cell c being split due to f has $||cf|| \in O(|c|)$, where the distance is defined according to the nearest approach between c and f. Consider now the point x on f that is closest to c. That point is $O(|c|)$ from one of the pseudo-vertices (possibly much closer), because the local feature size at x is at most $\text{lfs}(c) + ||cx||$, both terms of which are $O(|c|)$. Therefore, cells that blame f for being split pack around pseudo-vertices: each pseudo-vertex is blamed for $O(1)$ cells of each size, there are $O(\lg L/s)$ sizes of cells and $O(m_f)$ pseudo-vertices; a total, as claimed, of $O(m_f \lg L/s)$. Since every split is constant time (Lemma 5.1.1) except for point location charges, this drives the response time.

Given that a cell that is split according one input but not the other must hold few features, point location costs are $O(m_f + \lg L/s)$ for f and at most $O(\lg L/s)$ for all others, which is dominated by the worst-case cost of the splits.

As before, I appeal to the standard tricks of creating artificial dependencies to ensure conciseness and monotonicity of all operations, and to implement the change propagation priority queue in constant time per operation. ■

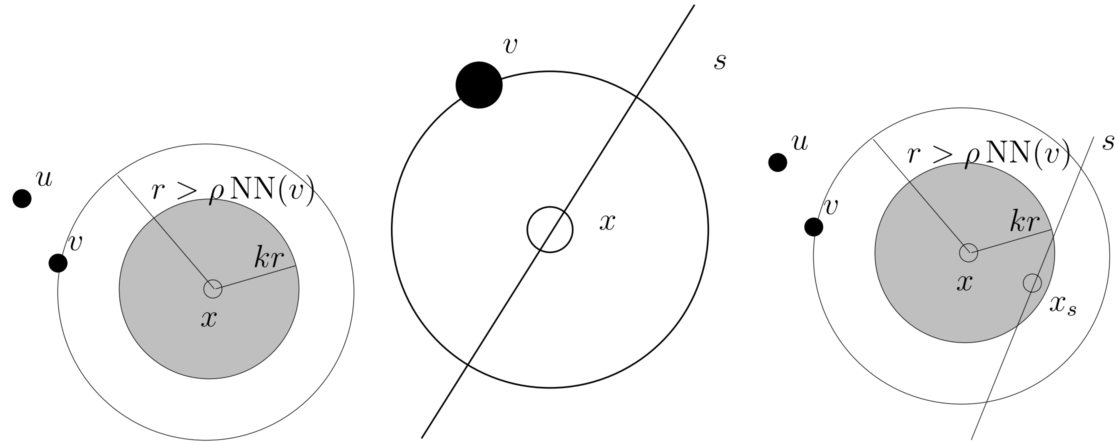
5.1.2 Practicalities

In the implementation of SVR, we found that the largest constant-factor costs of the algorithm were the point and range queries — not surprisingly, since they cause the $O(n \lg L/s)$ term of the total runtime analysis, whereas each point creation only affects the $O(m)$ term. Therefore, I wish to note some shortcuts to the BUILDQT implementation. Relocation of points is very cheap: d floating-point comparisons are all that is required to determine which of the 2^d sub-cells contains the cell. A point may lie exactly on the boundary between two cells; we must take some care to make sure to use greater-than or greater-or-equal tests consistently, so that cells are well-formed and non-overlapping. Ensuring that only one cell contains any one point avoids requiring any code to detect duplicates in $\hat{P}$: calling a cell crowded because a point is in both a cell and its neighbour is an annoyingly common bug in simple quad-tree implementations. Relocation of segments is slightly more expensive, but equally straightforward. To determine whether all segments in $\hat{S}$ share an endpoint, we compute the intersection of all their endpoints, short-cutting when the partial intersection set becomes empty. Duplicates are idempotent in this process, which is helpful given that duplicates are unavoidable.

Empty cells need not be split for crowding. Indeed, in the original BEG algorithm, the definition of crowding explicitly excludes empty cells. Not splitting empty cells for crowding clearly does not affect the grading guarantee, and only helps to achieve size-conformality with fewer cells. Therefore, if c^- or c' are empty, we can skip the body of the doubly-nested for loop, saving both time and memory. Alternately, if the conditional would return "true" for c^- in isolation, we need not iterate over all neighbours of c^- and perform the test. We also need not add all neighbours of c^- to the queues: the neighbour c' was already tested against c, unless it was empty (in which case it need not be split), or unless it is itself also a child of c (in which case it will shortly be checked). Finally, if c was crowded on its own (it contained two points, or two non-intersecting segments), then clearly all its children c^- are crowded: every child is a neighbour of every other child.

5.2 Choosing Steiner points with segments

Having built a size-conforming quadtree, we now proceed to create the Steiner points. In the prior chapter, the only requirement was that we complete all vertices. Now we have the significant new requirement to make sure that the Delaunay triangulation of the Steiner points (plus the original input) *respects* the constraining segments. To ensure this, the concept of completion will be expanded. Far from segments, Steiner points will be chosen as before. Near a segment, however, Steiner points will be in a sense snapped to the segment. Philosophically, this snapping dates to Chew [Che89] and Ruppert [Rup95] in that it is based on gap balls. The particular mechanism I use here is novel: as was the case with point clouds, the Delaunay balls that Chew and Ruppert use may be too expensive to compute, so I blow up gap balls only up to a distance related to the quality requirement ρ and the local feature size. This mechanism is the main new contribution of the current chapter. The development of the algorithm is again first conceptual to prove structural properties, and later made to be efficient.



(a) Consider a gap ball on v of radius $r \geq \rho\,\mathrm{NN}(v)$. If the smaller ball $B(x, kr)$ is empty of all segments, even including ones non-disjoint from v, then it is permissible to insert x.

(b) Consider a gap ball on v whose center lies on a segment s on which v does not lie. Then s is *encroached*, and it is permissible to insert x.

(c) Consider a gap ball on v of radius $r \geq \rho\,\mathrm{NN}(v)$. If a point x_s on a segment s lies in the smaller ball $B(x, kr)$, it is permissible to add x_s. Notably, s need not be disjoint from v.

Figure 5.2: List of permissible insertions for completing a vertex v with nearest neighbour u, in the possible presence of a constraining segment s. The COMPLETE algorithm iteratively adds permissible points until there is no permissible insertion around v.

5.2.1 Conceptual algorithm

First, I update slightly the definition of the nearest neighbour of a vertex v: it is now the nearest mesh vertex to v, or the distance to the nearest segment s that is disjoint from any segment on which v lies: $\mathrm{NN}(v)$ is the diameter of the smallest gap ball at v that touches a mesh vertex, or that touches a segment disjoint from any segment on which v lies (of which there may zero, one, or several).

To complete a vertex, I suggest a list of permissible insertions (see Figure 5.2); each one describes a situation that proves that the mesh does not satisfy our requirements of quality and of respecting the input. Once there are no permissible insertions, the mesh is complete. In the case of point clouds, there was only one permissible insertion: If v has a gap ball with radius $r > \rho\,\mathrm{NN}(v)$, then it is permissible to insert the center of this ball; this corresponds to harboring a simplex of bad radius/edge quality. The presence of segments complicates matters: we need to ensure we do not insert Steiner points overly near input segments, which would violate the size-conformality condition. If v has a gap ball $b = B(x, r)$ with radius $r > \rho\,\mathrm{NN}(v)$, and if there is a segment s that passes through the smaller concentric ball $B(x, r/2)$, it is permissible to insert any point $x_s \in s \cap B(x, r/2)$. If no segment passes through this shrunken ball, then x is permissible to insert. The factor of one half is optimal, but not critical; any constant strictly less than one will work. To establish that $1/2$ is optimal, in the description and proofs I use k. This idea of shrinking the gap ball by a factor k is from SVR, though there it appears in a slightly different context.

To ensure that input segments appear in the Delaunay triangulation of the output, following Chew I require that the circumball of any subsegment be empty of points — that it not be *encroached*. Given a segment s and a vertex v not on s, if v has a gap ball $b = B(x, r)$ whose center happens to lie on s, then v encroaches s. This is congruent with Chew's definition of encroachment: if there is an empty ball centered on s that has a vertex v on its surface, then clearly some subsegment of s is encroached by v. Given such a gap ball, it is permissible to insert the center x. This offers more choice of x than in Chew's algorithm (or than the equivalent operation in Ruppert's, Shewchuk's, and various other such algorithms): in my formulation x need not be the midpoint of an encroached subsegment.

5.2.2 A complete mesh with segments is size-conforming

Lemma 5.2.1 *There is a set of constants c_i with $i \in \{0, 1, d\}$, and $c \equiv \max c_i$, that depend only on ρ, such that when* COMPLETE*(v) inserts a point x with containing dimension i,*

$$\mathrm{lfs}(x) \leq c_i \,\mathrm{NN}(x)$$

Proof: Clearly, $c_0 = 1$; it is only defined for ease of reference. The proof follows the set of permissible moves closely, with a case for every move. Each case constrains the value of some c_i; after computing the constraints, I show they are satisfiable.

Almost every line is an inductive statement, with the induction being over the order in which points are inserted. Consider a vertex v, and its current nearest neighbour u. If v postdates u, then $\mathrm{lfs}(v) \leq c\,\mathrm{NN}(v)$. On the other hand, if u postdates v, then instead $\mathrm{lfs}(v) \leq \mathrm{lfs}(u) + ||uv||$ where $||uv|| = \mathrm{NN}(v)$ by definition. When u was inserted, its nearest neighbour was no further than v, so the induction yields that $\mathrm{lfs}(u) \leq c||uv||$. In either case, $\mathrm{lfs}(v) \leq (1 + c)\,\mathrm{NN}(v)$ in the current mesh.

Case 5.2a: The Lipschitz condition tells us $\mathrm{lfs}(x) \leq \mathrm{lfs}(v) + ||vx||$. The former term is, inductively, $\mathrm{lfs}(v) \leq (1 + c)\,\mathrm{NN}(v)$. We know that $\mathrm{NN}(v) \leq r/\rho$; also, given that the smaller ball is empty, $r \leq \mathrm{NN}(x)/k$. Thus, $\mathrm{lfs}(x) \leq \frac{1+c+\rho}{k\rho} \mathrm{NN}(x)$. We deduce $\boxed{c_d \geq \frac{1+c+\rho}{k\rho}}$.

Case 5.2b: If v is of containing dimension 0 or 1, then v and s are from disjoint features (because of the non-acute input assumption, a ball centered on s cannot intersect a segment s' that intersects s except at the common endpoint of s and s') and $\mathrm{lfs}(x) \leq ||vx|| = \mathrm{NN}(x)$. This constrains $\boxed{c_1 \geq 1}$. Otherwise, v is a vertex of containing dimension d. By the Lipschitz condition, $\mathrm{lfs}(x) \leq \mathrm{lfs}(v) + ||vx||$. We know inductively that $\mathrm{lfs}(v) \leq (1+c_d)\,\mathrm{NN}(v)$. Clearly, $\mathrm{NN}(v) \leq ||vx||$, by which we conclude $\mathrm{lfs}(x) \leq (2 + c_d)\,\mathrm{NN}(x)$ and $\boxed{c_1 \geq 2 + c_d}$.

Case 5.2c: If $\mathrm{NN}(x_s)$ is defined by a second segment, then $\mathrm{NN}(x_s) = \mathrm{lfs}(x_s)$ and $\boxed{c_1 \geq 1}$. Otherwise, induction and Lipschitz at x give, as in the proof of Case 5.2a, $\mathrm{lfs}(x) \leq (\frac{1+c}{\rho} + 1)r$. However, we are not inserting x but rather x_s. The distance $||xx_s||$ is at most kr. Also, since $\mathrm{NN}(x_s)$ is defined by a vertex and the gap ball centered at x is empty of points, the distance from x_s to its nearest neighbour is at least $\mathrm{NN}(x_s) \geq (1-k)r$. Plugging this in to $\mathrm{lfs}(x_s) \leq \mathrm{lfs}(x) + ||xx_s||$ yields that $\mathrm{lfs}(x_s) \leq (\frac{1+c}{\rho} + 1 + k)r \leq (\frac{1+c}{\rho} + 1 + k)\frac{1}{1-k}\,\mathrm{NN}(x_s)$. All in all, this constrains $\boxed{c_1 \geq \frac{1+c}{(1-k)\rho} + \frac{1+k}{1-k}}$.

Case 5.2b shows that $c_1 > c_d$, and therefore $c = c_1$. Substituting in, we therefore get the pair of constraints $\boxed{c_1 \geq \frac{(2k+1)\rho+1}{k\rho-1}}$ from Cases 5.2b and 5.2a, and $\boxed{c_1 \geq \frac{(1+k)\rho+1}{(1-k)\rho-1}}$ from Case 5.2c, assuming both $k\rho > 1$ and $(1-k)\rho > 1$. This means that we can allow the user to demand $\rho > 2$, independent of dimension, by setting $k = 1/2$. At this setting, we can easily evaluate the constants: $c_0 = 1$, $c_1 = \frac{4\rho+2}{\rho-2}$, and $c_d = \frac{\rho+6}{\rho-2}$. ■

Lemma 5.2.2 *Upon completing the mesh, it contains no overly large mesh elements: for every mesh vertex v of containing dimension less than d, we have that* $\mathrm{NN}(v) \leq 2\,\mathrm{lfs}(v)$.

Proof: The vertex v lies on a feature f. The local feature size at v is defined by a ball that touches, at a point y, a feature f' disjoint from f: $\mathrm{lfs}(v) \equiv ||vy||$. Let u be the vertex on f' closest to y. Clearly, the nearest mesh vertex to v is no farther than u: $\mathrm{NN}(v) \leq ||uv||$. The ball $B(y, ||uy||)$ is an empty ball — if v were within the ball, v would not be complete according to Case 5.2b. Therefore, $||uy|| \leq ||vy|| = \mathrm{lfs}(v)$. This proves

$$\mathrm{NN}(v) \leq ||uv|| \leq ||uy|| + ||yv|| \leq \mathrm{lfs}(v) + \mathrm{lfs}(v)$$

■

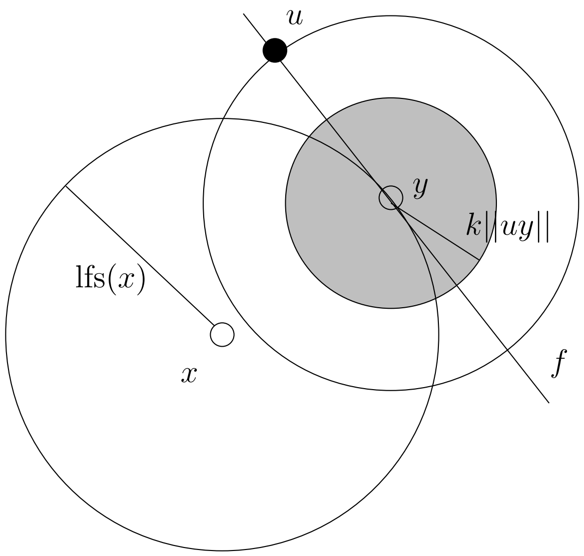


Figure 5.3: Illustration of the proof of Lemma 5.2.3.

Lemma 5.2.3 *Upon completing the mesh, for every point $x \in \Omega$, we have that*

$$\mathrm{lms}(x) \leq 10\,\mathrm{lfs}(x)$$

Where $\mathrm{lms}(x)$ denotes the local mesh size (the local feature size induced by the vertices of the mesh).

Proof: The local feature size at x is defined by a ball centered at x which on its surface intersects a feature f at some point y (see Figure 5.3); $\mathrm{lfs}(x) = ||xy||$. Take u to be the nearest mesh vertex to y that lies on the feature f. If f is a point, then $y = f = u$. More generally, f may be a segment. By the Lipschitz condition on lms, we know that $\mathrm{lms}(x) \leq \mathrm{lms}(u) + ||ux||$. Lemma 5.2.2 proved that $\mathrm{lms}(u) \leq 2\,\mathrm{lfs}(u)$. We can apply the Lipschitz condition, this time on lfs, to deduce $\mathrm{lfs}(u) \leq \mathrm{lfs}(x) + ||ux||$. So we know $\mathrm{lms}(x) \leq 2\,\mathrm{lfs}(x) + 3||ux||$. The distance from u to x is $||ux|| \leq ||xy|| + ||yu|| = \mathrm{lfs}(x) + ||yu||$. Because the mesh is complete, we know that the ball $B(y, ||uy||)$ is empty of points or features — otherwise, Case 5.2b would apply to either u or another vertex. Thus, $||uy|| \leq \mathrm{lfs}(y)$. At the same time, $\mathrm{lfs}(y) \leq \mathrm{lfs}(x) + ||xy|| = 2\,\mathrm{lfs}(x)$. Therefore, $||ux|| \leq 3\,\mathrm{lfs}(x)$. Plugging back in to the inequality before, we have $\mathrm{lms}(x) \leq 10\,\mathrm{lfs}(x)$. ■

Theorem 5.2.4 *A complete mesh is a size-conforming mesh, its Voronoi diagram has good aspect ratio, and its Delaunay tessellation respects the input.*

Proof: Lemmas 5.2.1 and 5.2.3 prove that the mesh is size-conforming: at any point in the meshing domain, the spacing induced by the mesh vertices is within a constant factor of the spacing induced by the segments and input points.

A vertex with a bad aspect ratio Voronoi cell would not be complete according to one of Cases 5.2a or 5.2c. Lastly, if a vertex v were to lie within the diametral ball of a subsegment of s, then v would not be complete according to Case 5.2b. This shows that every segment s is composed of subsegments, each of which is individually Delaunay. ■

5.3 Efficient algorithm

I adapt the CHOOSESTEINERS algorithm from the previous chapter, but with the new definition of completion. See Figure 5.6. The efficient implementation of completion depends on using a bucketed priority queue, where each bucket containing vertices with $\mathrm{NN}(v) \in [l, O(l))$. The constant here will differ from the constant (ρ) in the prior chapter. The proof that this is efficient depends on Lemma 4.3.3, which stated the algorithm can quickly search for a gap while processing the smallest remaining bucket. Its proof depended upon the claim that every new vertex v created by COMPLETE had its nearest neighbour such that v would be processed in a later bucket, as proved for point clouds in Lemma 4.3.1. This is clearly true for points created by Cases 5.2a, or 5.2c, as long as we the bucketing constant is $k\rho$ (that is, $\rho/2$): each bucket contains vertices with $\mathrm{NN}(v) \in [l, k\rho l)$. However, Case 5.2b violates this assumption. Indeed, consider two edges passing near each other, as in Figure 5.4. There is a gap at a with center on cd that inserts a vertex with nearest neighbour at distance ϵ! Fundamentally, the issue is that completing vertices is insufficient to getting good runtime when the local feature size is dictated by a pair of segments. For this reason, I define an operation DISENCROACH(c, s), detailed in Figure 5.5. As the CHOOSESTEINERS algorithm progresses, DISENCROACH inserts vertices on s so that points on s with small local feature size always have a nearby complete vertex.

Lemma 5.3.1 *When a vertex v is inserted as a result of* DISENCROACH*,* $\mathrm{lfs}(v) \leq c_1 \, \mathrm{NN}(v)$.

Proof: If the empty ball centered at v touched a disjoint segment, or touched a vertex that is not an endpoint, then $\mathrm{lfs}(v) = \mathrm{NN}(v)$. The remaining case in which v is indeed created is that the empty ball touched a vertex u of containing dimension d. Then v is the center of a gap ball on u that has its center on a segment; thus DISENCROACH emulates Case 5.2b of COMPLETE, which was already analyzed. ■

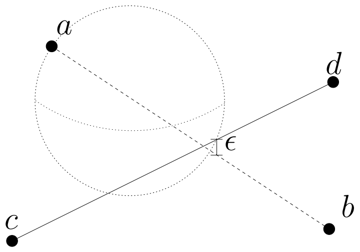


Figure 5.4: The maximal gap at a along the input segment ab causes Case 5.2b of COMPLETE to create a new vertex u on cd with $\mathrm{NN}(u) \approx ||ab||/2$, near where the two skew segments almost meet. The new vertex u can in turn, by the same case, insert a new vertex v with $\mathrm{NN}(v) \approx \epsilon$.

DISENCROACH(c, s)

for each $x \in c \cap s$ **do**
    Grow a ball $b = B(x, r)$ until it touches a disjoint segment or a mesh vertex v
    If b touched a vertex v that lies on s, continue the loop.
    Otherwise, add x
end for

Figure 5.5: A routine to avoid the problem described in Figure 5.4. Having computed the quadtree, we already have a good estimate of the local feature size everywhere. Therefore, we can add points on segments even before finding points to complete in the area.

Lemma 5.3.2 DISENCROACH *inserts at most* $O(1)$ *vertices in a quadtree cell* c*.*

Proof: The segment s has length at most $\sqrt{d}|c|$ through c. According to Lemma 5.3.1, any vertex inserted by DISENCROACH has nearest neighbour no closer than $\text{lfs}(v)/c_1$. Because c contains one input segment, no neighbour does, which lower bounds $\text{lfs}(x) \geq |c|/2$ everywhere in c. In other words, every vertex has $\text{NN}(v) \geq |c|/2c_1$. Only $2c_1\sqrt{d}$ vertices can be fit in this cell. ∎

Lemma 5.3.3 *During* CHOOSESTEINERS*, if all work with key* l *or less has been processed, then for any point* x *on any segment* s*, with* $\text{lfs}(x) < (1-k)\rho l$*, there is a vertex* v *such that* $||xv|| \leq \text{lfs}(x)$*.*

Proof: Let c be the cell in which x lies. DISENCROACH explicitly ensures that every point x in $c \cap s$ has a vertex v on s within distance $\text{lfs}(x)$. Since there is a feature in c, no neighbour of c can contain a disjoint feature; therefore, $\text{lfs}(y) \geq |c|/2$ for all y in c — in particular, $\text{lfs}(x) \geq |c|/2$. If $\text{lfs}(x) < (1-k)\rho l$, this implies $|c|/2 < (1-k)\rho l$. Given that $\frac{|c|}{2(1-k)\rho}$ is precisely the key associated with $|c|$, c has been processed, so DISENCROACH has been called and x has a nearby vertex. ∎

Lemma 5.3.4 *During* CHOOSESTEINERS*, while processing the bucket* $[l, (1-k)\rho l)$*, when a new vertex* x *is inserted, then either* $\text{NN}(x) \geq (1-k)\rho l$*, or every vertex* y *that* x *in turn inserts has* $\text{NN}(y) \geq (1-k)\rho l$*.*

Proof: When Case 5.2a inserts x, $\text{NN}(x) \geq (1-k)\rho\,\text{NN}(v)$ (since there may be a segment within the gap ball, but none near the center); meanwhile, $\text{NN}(v) \geq l$. The same holds for Case 5.2c. For points created by DISENCROACH, $\text{NN}(x) = \text{lfs}(x) \geq |c|/2 \geq l(1-k)\rho$. The only truly interesting case is Case 5.2b.

Consider the point x on s that is being inserted due to encroachment. Its nearest neighbour, and the point currently being completed, is v. If v is an input point or a containment dimension 1 vertex, then x and v are on disjoint features; therefore, $\text{lfs}(x) \leq ||xv||$. We know $\text{lfs}(x) \geq (1-k)\rho l$ or else there would be a vertex within distance $\text{lfs}(x)$ of x, contradicting that $B(x, ||vx||)$ is an empty ball. Therefore, $||vx|| \geq (1-k)\rho l$.

The only remaining case is Case 5.2b, where v is a containment dimension d vertex. Given that x is a containment dimension 1 point, we saw earlier in this proof that any vertex y that x will create will have $\text{NN}(y) \geq (1-k)\rho\,\text{NN}(x)$. We also know that $\text{lfs}(x) \geq (1-k)\rho l$ because $B(x, ||vx||)$ is empty of points; this further shows that the ball is also empty of disjoint segments, leaving only v to be the nearest neighbour of x: $\text{NN}(x) = ||vx||$.

CHOOSESTEINERS($\mathcal{P}$, $\mathcal{S}$, QT, P, S, ρ, γ, k)

P: map storing the correspondence between quadtree cells and vertices
S: map storing the correspondence between quadtree cells and segments
Q: work queue, bucketed with factor $(1-k)\rho$
for each cell c in QT **do**
    If c contains a point p, enqueue a COMPUTENN(p) event with priority $k_{low}|c|$
    Otherwise, if c contains a segment s, enqueue a DISENCROACH(s, c) event with priority $\frac{|c|}{2(1-k)\rho}$.
end for
while Q not empty **do**
    $w \leftarrow$ POP(Q)
    if w is a COMPUTENN(p) event **then**
        Grow a ball $b = B(p, r)$ until it touches a disjoint segment or mesh vertex
        Add a COMPLETE(p) event with priority r
    else if w is a DISENCROACH(s, c) event **then**
        Disencroach s over c
        For each new vertex v (if any), INSERT(v)
    else if w is a COMPLETE(u) event **then**
        Complete u
        For each new vertex v (if any), INSERT(v)
    end if
end while

INSERT(v)

Add a COMPLETE(v) event with priority $\mathrm{NN}(v)$
for each v' in the link of v **do**
    Add a COMPLETE(v') event with priority $||vv'||$
end for

Figure 5.6: Algorithm to complete a mesh such that it is quality, size-conforming, and respects input points and segments. The map S does not change during the algorithm, since no new segments are added. The map P changes as new vertices are added. To ensure short dependency paths, use gridding and colouring as mentioned in the previous chapter. Parameter ρ must be strictly larger than 2; γ must be a constant larger than ρ; k should be $1/2$.

Finally, the distance from v to s is certainly no further than $||vx||$, so the nearest neighbour to v is $||vx|| \geq \mathrm{NN}(v)$. Stringing these together shows:

$$\mathrm{NN}(y) \geq (1-k)\rho\,\mathrm{NN}(x) \geq (1-k)\rho||vx|| \geq (1-k)\rho\,\mathrm{NN}(v) \qquad \blacksquare$$

The flurry of proofs we just saw tells us the following: when CHOOSESTEINERS processes an event from the work queue, it may add items to the current bucket, but never to a smaller bucket. This is sufficient for the proof of Lemma 4.3.3. Therefore, I can claim the runtime is fast:

Theorem 5.3.5 (Static Meshing with Segments is Fast) *The* CHOOSESTEINERS *routine described in Figure 5.6 runs in $O(m)$ time. In conjunction with* BUILDQT, *this produces a quality, size-conforming mesh that respects the input points and segments, in total time $O(n \lg L/s + m)$.*

Proof: A COMPUTENN event involves casting a single query of radius at most $\mathrm{lfs}(p) \leq c_{hi}(1-k)\rho l$ around p to find its nearest neighbour, and thus takes $O(1)$ time per input vertex, a total of $O(n)$ time. Every query to support a DISENCROACH event involves casting a query of radius at most $\mathrm{lfs}(x) \leq k_{hi}(1-k)\rho l$ around a point $x \in c \cap s$. This results in either inserting a new vertex, or finding an old vertex. After $O(1)$ queries, the event has been processed. This event occurs at most once per quadtree cell, so $O(1)$ time per cell, and there are $O(m)$ cells. A COMPLETE event involves casting at most a constant number of queries, each of radius at most $\gamma\,\mathrm{NN}(v) \in O(\mathrm{lfs}(v))$ around a mesh vertex v. Thus each query takes $O(1)$ time, and there are $O(m)$ mesh vertices in the output. $\blacksquare$

Theorem 5.3.6 (Dynamic Meshing with Segments is Fast) *The* CHOOSESTEINERS *routine is $O(m_f \lg L/s)$-stable to adding a feature f that has m_f mesh vertices on it after adding it, or to removing a feature with m_f mesh vertices before removing it, where $m_f = 1$ for input points. Combined with* BUILDQT, *this gives an $O(m_f \lg L/s)$ response time to dynamic changes.*

Proof: The first order of business is to prove that we can colour the work queue appropriately. While processing bucket $[l, (1-k)\rho l)$, two COMPLETE events at distance $2\gamma\rho l$ are independent. Similarly, two DISENCROACH events at distance $2k_{hi}l$ are independent. COMPUTENN events are all independent. Therefore, we can colour the work queue using the technique of gridding space to achieve a constant number of colours. A bucket may need to be processed twice: a vertex of containing dimension d may create a COMPLETE event in the same bucket due to Case 5.2b. However, the new event will be for a vertex of containing dimension 1, so this will not be repeated, as proved in Lemma 5.3.4. Thus, after a constant number of iterations, the smallest event to be processed will have grown by a factor $(1-k)\rho$.

Stability: If a Steiner point v blames a feature f, then let x be the closest point on f to v. That events grow after a constant number of rounds proves that $||vx|| \in O(\mathrm{NN}(v))$; the proof is analogous to that given in Lemma 4.3.7. Consider the vertex u on f that is closest to x. Because of the non-encroachment condition (Case 5.2b), $||ux|| \leq ||vx||$. Therefore, Steiner points that blame f pack around the vertices that lie on f, which proves that at most $O(m_f \lg L/s)$ vertices blame f.

Response time: Finally, after the standard arguments for conciseness, monotonicity, and change propagation priority queue; and after composing BUILDQT and CHOOSESTEINERS, it becomes clear that the response time to adding (or removing) an input point is $O(\lg L/s)$, while the response time to adding (or removing) an input segment which, when present, has m_f points on it is $O(m_f \lg L/s)$. ■

5.4 Remarks

The idea of ensuring that input segments appear in the Delaunay triangulation by ensuring that each of their subsegments has an empty diametral ball is due to Chew [Che89], and was widely adopted [Rup95, She97b, She98b, LT01, MPW02, for example]. Lee and Lin [LL85] define the "generalized Delaunay triangulation," thanks to Chew [Che87] now known by the name of Constrained Delaunay Triangulation (CDT), which explicitly ensures that the subsegments appear even if their diametral balls are non-empty, which has been used with great success in two dimensions. It seems likely that the set of permissible insertions could be adapted to conform to the philosophy of CDT-based meshing. However, my focus is on higher-dimensional meshing, and extending the CDT even to three dimensions is non-trivial: the CDT of an input does not always exist, even when it does it may be NP-hard to compute it, and even when it is provably non-hard to compute it, implementing the algorithms remains non-easy [She98a].

Cases 5.2c and 5.2a cause my algorithm to only be able to produce meshes with Voronoi aspect ratio 2.0 in two dimensions. Traditionally, algorithms are able to produce meshes with radius/edge ratio $\sqrt{2}$. It is not yet clear to me whether the difference is a proof theoretic one, or an algorithmic one. If it is algorithmic, I suspect that less aggressive yielding would be the solution. Another idea is to use diametral lenses rather than diametral balls, which is used especially in two dimensions to improve the quality bound.

Chapter 6

Dynamic Meshing for PLC Input

Having shown how to produce a quality mesh over point cloud input and maintain it through dynamic updates; then how to additionally handle input segments, I can finally discuss the final details for handling full-dimensional features. Only two changes are required: one, to define the input; the other, to redefine the details of inserting a new mesh vertex. At the close of this chapter, we have the main result of this thesis.

The input I desire to handle is a *Piecewise Linear Complex* (PLC), see Section 1.2. Intuitively, a PLC is what one would draw in a CAD program: a set of piecewise linear objects — *features*: vertices (dimension 0), segments (dimension 1), polygons (2), and so on. Features must intersect in a well-behaved manner: If two features f and f' intersect, the intersection is required to itself be a feature f''. Geometrically, each feature must be a closed manifold. If a feature is not closed, it is ill-defined to produce a mesh of the feature. If a feature is non-manifold, it is trivial to subdivide it into a set of manifold features.

In terms of a programmatic description, the simplest is to list a set of input points, each one a pair of a unique index and a point $p \in \mathbb{R}^d$. Then, list a set of segments, each one an index for the segment, and two indices of input points. Then, a set of polygons as an index for the polygon, followed by indices for the segments and points that the polygon intersects (including its boundary). For a PLC in dimension greater than three, the format is generalized in the obvious manner.

I assume the existence of a black box query for deciding whether a hypercube in $\mathbb{R}^d$ intersects a feature; this query should take constant time. I further assume the existence of a query that returns, given an arbitrary point $x \in \mathbb{R}^d$, and a feature f, the closest point $y \in f$, again in constant time. Both are easily implementable for convex features with boundaries of constant size (independent of the description length — even if there are many internal features, only the size of the boundary matters). The question of implementing these queries for a non-convex feature, or in constant time for a feature with more than a constant-bounded number of vertices on their boundary, is left for the reader.

The goal will be to output a set of points whose Delaunay triangulation *respects* the PLC. A feature in dimension i is respected if it is tiled by a set of i-simplices, each of which is Delaunay according to the output point set. As is historically common, I ensure a stronger condition: each of the i-simplices has a ball centered on the simplex that goes through all of its $i + 1$ points, termed the *diametral ball*. That ball will be empty of any points.

In order for the algorithm to provably terminate, any two features must intersect at orthogonal or obtuse angles. This is so that two equal-dimensional features that intersect in a common lower-dimensional feature do not encroach each other (i.e. they do not insert points into each others' diametral balls), which could cause an infinite loop. Defining the *local feature size* at any point x, denoted $\text{lfs}(x)$, as the radius of the smallest ball that intersects two features f and f' that are mutually disjoint, I will prove that in the output, every mesh vertex v has a neighbour not much farther than local feature size, but also not much closer. That is, the output is size-conforming.

It is interesting to note that another definition of local feature size, used in surface reconstruction, is that the local feature size of a point x on a manifold surface is the distance from x to the medial axis of the surface. The medial axis of a piecewise linear surface touches the surface precisely when the surface has an angle of $90°$ or less. The mesh my algorithm outputs is size-conforming, so if the local feature size is zero, then the number of vertices required would be infinite.

Dynamic updates that take place must leave an input that matches all the requirements above. In particular, in order to remove *e.g.* a segment from the input, all polygons that name that segment must first be removed. Otherwise, those polygons would now be ill-formed, in that they would not be closed manifolds. Similarly, when adding a new feature f, it must not intersect another feature except in a lower-dimension subfeature of f, and even then, intersections must be at non-acute angles.

The exposition here is much abbreviated from prior chapters, because so much is in common with the prior claims, algorithms, and proofs. I highlight only the differences, except that I provide full algorithm descriptions for completeness and to help in an eventual implementation.

BUILDQT(L, $\mathcal{P}$, $\mathcal{X}$)
1: $T \leftarrow [0, L]^d$
2: $Q_{\text{qt}} \leftarrow [0, L]^d$
3: initialize F to hold pointers from $[0, L]^d$ to every $f \in \mathcal{X}$
4: **while** Q_{qt} not empty **do**
5:     $c \leftarrow \text{POP}(Q_{\text{qt}})$
6:     **if** c is not a leaf cell, skip
7:     **for each** neighbour c' of c **do**
8:         if $|c'| = 2|c|$ then add c' to Q_{qt}
9:     **end for**
10:     Split c into 2^d children
11:     **for each** $f \in F(c)$ **do**
12:         Add f to the F entry of every child that intersects f
13:     **end for**
14:     **for each** child c^- of c **do**
15:         **for each** neighbour c' of c^- **do**
16:             $\hat{F} \leftarrow F(c^-) \cup F(c')$
17:             **if** not all features in $\hat{F}$ share a common input point **then**
18:                 add both c^- and c' to Q_{qt}
19:             **end if**
20:         **end for**
21:     **end for**
22: **end while**

Figure 6.1: The algorithm to build a size-conforming quad-tree for PLC input.

6.1 Building the quad-tree

Building the quadtree (see Figure 6.1) to be lfs-conforming to the PLC input is almost identical to doing so for segments. Since a feature may now be a point, a segment, or a higher-dimensional object, I simplify the exposition to speak only of “features” rather than specifying what kind.

Checking whether f intersects a cell in Line 12 invokes the black-box constant-time query to see whether f intersects a cell c'; this is where this query is used.

If two features are non-disjoint, then in particular, they must share an input point. This means that to see whether a cell contains two disjoint features, it is sufficient to test if all the features share a common point. Testing for a common point can be an expensive operation if the description of a feature is allowed to grow large; but the test must operate in constant time. For this reason (and in order to implement the black box from above in constant time), I require the description of a feature to name only a constant-bounded number of input points. Violating this, and having a feature with Δ points on it, adds a factor $O(\Delta)$ to the runtimes I prove.

The proofs of the prior chapter all apply, modulo replacing the word "segment" with the word "feature." Therefore, I claim the following theorem holds:

Theorem 6.1.1 BUILDQT *produces a size-conforming quadtree over a PLC input $\mathcal{X}$, in time $O(n \lg L/s + m)$. To the addition or removal of a feature f which, when present, intersects m_f quadtree cells,* BUILDQT *responds in $O(m_f \lg L/s)$ time.*

6.2 Choosing Steiner points with features

In the case of choosing Steiner points for segments, there were three rules. The first eliminated large gaps far from any segments, which is the sole operation needed when operating on point cloud inputs — this is the rule that ensures the output will have good quality. The second eliminated encroachments near segments so that they would appear in the output mesh. The third handled the case of a large gap whose center would be near a segment, *yielding* to the segment if needed. The most obvious generalization of these rules to three dimensions requires seven cases: (1) a large gap with no nearby features. (2) A large gap with a nearby segment. (3) A large gap with a nearby face but no nearby segment. (4) A large gap with a nearby face and an almost nearby segment but not nearby enough that rule 2 applies. (5) An encroached segment. (6) An encroached facet with no nearby segment. (7) An encroached facet with a nearby segment that is not encroached. Clearly, to produce an arbitrary-dimensional algorithm, I need a more tractable way to generalize.

Fundamentally, we only need two rules, which I describe in Figure 6.2. The first rule handles large gaps. The second rule handles encroachments. We always insert a point in a large gap, or in an encroachment gap. However, instead of inserting the center of the gap, it might be better to *yield* to insert a point on a lower-dimensional feature. Let k be a constant less than 1. Consider a ball $B(x, r)$ with a center x that we are considering for insertion, where x has containing dimension i. If the gap ball contains a lower-dimensional input feature within distance kr, then I propose to yield to that feature and insert the closest point y on that feature (or, indeed, any point within distance kr of x). Recursively, y may need yield again. For ease of notation in the proofs below, define the complement of k as

$\bar{k} \equiv (1 - k)$.

6.2.1 A complete mesh is a good mesh

If we repeatedly call COMPLETE, described in Figure 6.2 on every vertex until no new vertices are added, the resulting mesh is size conforming. To do so, I need to first know the effect of yielding:

Lemma 6.2.1 (Yielding effects) *Let v be a vertex with a gap $B(x, r)$, which subsequently yields to insert a vertex y. Let x have containing dimension a. Then $||xy|| \leq (1 - \bar{k}^{d-1})r$ if $a = d$ (that is, if x is being inserted to ensure quality), and $||xy|| \leq (1 - \bar{k}^{d-2})r$ if $a < d$ (that is, x is being inserted for encroachment). If the nearest neighbour of y is a disjoint feature, then* $\mathrm{NN}(y) = \mathrm{lfs}(y)$. *Otherwise,* $\mathrm{NN}(y) \geq \bar{k}^{d-1}$ *if $a = d$, and* $\mathrm{NN}(y) \geq \bar{k}^{d-2}$ *if $a < d$.*

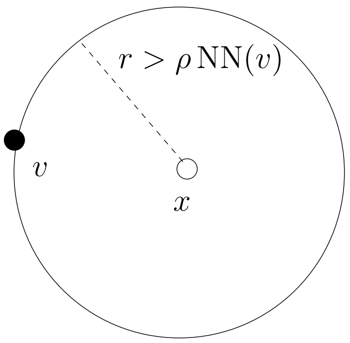
Proof: Let $x = y_0, y_1, \ldots y_i = y$ be the sequence of yields that led from x to y. Clearly, the more yields are performed, the farther y is from x, and the closer to the edge of the gap ball y is. Therefore, we can assume that in the worst case, $y = 1$.

Let a be the containing dimension of x. At most $a - 1$ yields can have been performed, since segments do not yield. Let r_j be the radius of the ball during the jth yield; $r_0 = r$. Clearly, $r_j = \bar{k}^j r$. Also, when y_i yields to y_{i+1}, we know that $||y_i y_{i+1}|| \leq k r_j$. Thus, the total distance from x to y is at most $\sum_{j=0}^{a-2} ||y_j y_{j+1}|| \leq kr \sum_{j=0}^{a-2} \bar{k}^j \leq (1 - \bar{k}^{a-1})r$.

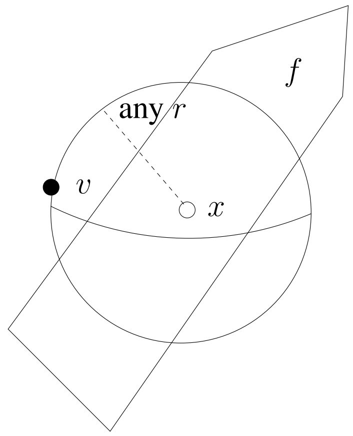
If the nearest neighbour of y is a disjoint feature, then $\mathrm{NN}(y) = \mathrm{lfs}(y)$. Otherwise, the nearest neighbour of y is a vertex. We know there are no vertices within the ball $b = B(x, r)$, and on every yield, the remaining ball is nested within b. Therefore, the nearest neighbour of y is at distance $\mathrm{NN}(y) \geq r - ||xy|| \geq \bar{k}^{a-1} r$. ■

The two rules will iteratively insert points, possibly yielding at every step. Assume the existence of two constants, c_i and c_d. I claim that when a vertex is inserted that had containing dimension d, it has $\mathrm{lfs}(v) \leq c_d \, \mathrm{NN}(v)$. If it has containing dimension $i < d$, then $\mathrm{lfs}(v) \leq c_i \, \mathrm{NN}(v)$. Let $c = \max(c_i, c_d)$. By the inductive argument we have seen twice before, if these two claims hold, then at any time during the algorithm after v was inserted, it is the case that $\mathrm{lfs}(v) \leq (1 + c) \, \mathrm{NN}(v)$. I now proceed to prove the claims. As before, each case of the proof constrains the constants; if the constraints can all be simultaneously satisfied, then the proof holds.

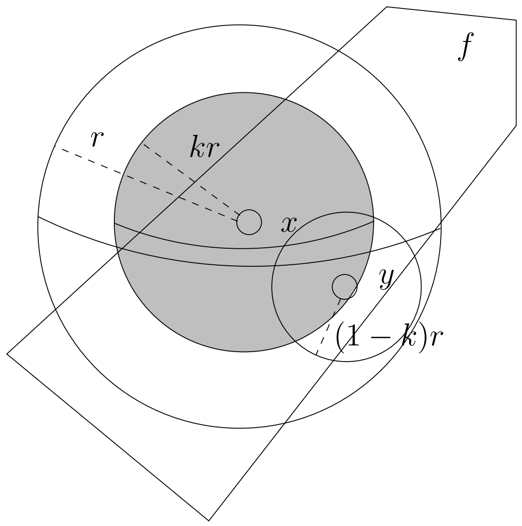
Lemma 6.2.2 *Assume the* COMPLETE *algorithm is processing a gap ball $B(x, r)$ on a vertex v, with $r > \rho \, \mathrm{NN}(v)$ — that is, assume it is inserting a point due to bad quality in*



(a) For quality: consider inserting the center of any gap ball on v of radius $r \geq \rho\,\mathrm{NN}(v)$.



(b) For encroachment: consider inserting the center of a gap ball on v, if $x \in f$ and f is disjoint from any feature on which v lies.



(c) Yielding: If we are considering inserting x, of containing dimension i, check whether there is a feature f of containing dimension $j < i$ within kr of x. If so, recursively consider y on f, with a ball of radius $(1-k)r$. If not, insert x.

Figure 6.2: The rules for handling PLC features. The COMPLETE algorithm consists of checking if either of the first two cases apply. If so, it chooses a permissible gap ball $B(x, r)$ and recursively checks whether x should yield according to the third rule.

the mesh. If it succeeds in inserting x, then $\mathrm{lfs}(x) \leq c_d \,\mathrm{NN}(x)$*. If it instead yields and inserts a vertex y of containing dimension $i < d$, then* $\mathrm{lfs}(y) \leq c_i \,\mathrm{NN}(y)$*.*

Proof: If x is inserted, then $\mathrm{lfs}(x) \leq \mathrm{lfs}(v) + r$. Inductively, $\mathrm{lfs}(v) \leq (1+c)\,\mathrm{NN}(v)$. We know that $\mathrm{NN}(v) \leq r/\rho$. Finally, since we did not yield, $r \leq \mathrm{NN}(x)/k$. This sums to $\mathrm{lfs}(x) \leq (\frac{1+c}{\rho} + 1)\,\mathrm{NN}(x)/k$, which satisfies the Lemma assuming $\boxed{c_d \geq \frac{1+c+\rho}{k\rho}}$.

In the other case, y was inserted as the result of (possibly recursive) yielding. If the nearest neighbour of y is a disjoint feature, $\mathrm{lfs}(y) = \mathrm{NN}(y)$ which limits $\boxed{c_i \geq 1}$. Otherwise, I proceed using the Lipschitz condition as usual: $\mathrm{lfs}(y) \leq \mathrm{lfs}(v) + ||vy||$. Lemma 6.2.1 bounds $||vy|| \leq (2 - \bar{k}^{d-1})r < 2r$ and $r \leq \mathrm{NN}(y)/\bar{k}^{d-1}$. The argument above shows that $\mathrm{lfs}(v) \leq \frac{1+c}{\rho} r$. Thus,

$$\mathrm{lfs}(y) \leq (\frac{1+c}{\rho} + 2)r \leq \frac{1+c+2\rho}{\bar{k}^{d-1}\rho}\,\mathrm{NN}(y)$$

This requires that $\boxed{c_i \geq \frac{1+c+2\rho}{\rho\bar{k}^{d-1}}}$. ■

Lemma 6.2.3 *Assume the* COMPLETE *algorithm is processing a gap ball $B(x,r)$ on a vertex v, with x centered on a feature f. If x is inserted, let $y = x$. Otherwise, x yields to a point y. In either case,* $\mathrm{lfs}(y) \leq c_i \,\mathrm{NN}(y)$*.*

Proof: If $\mathrm{NN}(y) = \mathrm{lfs}(y)$ then we are done.

Otherwise, Lemma 6.2.1 proves that $\frac{\mathrm{NN}(y)}{\bar{k}^{d-2}} \geq r$. If v has containing dimension less than d, then v and x lie on different features: $\mathrm{lfs}(y) \leq ||vy|| \leq (2 - \bar{k}^{d-2})r < 2r$. Thus $\mathrm{lfs}(y) \leq \frac{2}{\bar{k}^{d-2}}\,\mathrm{NN}(y)$. This constrains $c_i \geq \frac{2}{\bar{k}^{d-2}}$, which is overshadowed by the next paragraph.

If instead v has containing dimension exactly d, then we can inductively assume that $\mathrm{lfs}(v) \leq (1+c_d)\,\mathrm{NN}(v)$. When v was inserted, v did not yield to the feature on which x lies, so $k\,\mathrm{NN}(v) \leq ||vx|| = r$. The Lipschitz condition and Lemma 6.2.1 give us $\mathrm{lfs}(y) \leq \mathrm{lfs}(v) + ||vy|| \leq (\frac{1+c_d}{k} + 2)r$. Given $r \leq \frac{\mathrm{NN}(y)}{\bar{k}^{d-2}}$, we can conclude $\mathrm{lfs}(y) \leq (\frac{1+c_d}{k\bar{k}^{d-2}} + \frac{2}{\bar{k}^{d-2}})\,\mathrm{NN}(y)$. This constrains $\boxed{c_i \geq \frac{1+c_d+2k}{k\bar{k}^{d-2}}}$. Notice also that this shows that $c_i > c_d$. ■

Lemma 6.2.4 (Spacing is not too small) *There exist constants c, c_i, and c_d such that when* COMPLETE *adds a vertex v,* $\mathrm{lfs}(v) \leq c\,\mathrm{NN}(v)$*.*

Proof: From the boxes in the two prior proofs, we know that

$$c \geq c_i \geq \frac{1 + c + 2\rho}{\bar{k}^{d-1}\rho}$$

In other words, $c \geq \frac{2\rho+1}{\bar{k}^{d-1}\rho-1}$, so long as $\bar{k}^{d-1}\rho > 1$. Simultaneously,

$$c \geq c_i \geq \frac{1 + c_d + 2k}{k\bar{k}^{d-2}}$$

where $c_d \geq \frac{1+c+\rho}{k\rho}$. Thus, $c \geq \frac{k\rho+1+\rho+2k^2\rho}{\bar{k}^{d-2}k^2\rho-1}$, so long as the denominator is strictly positive. We conclude from this exercise that with k and ρ set such that both $\bar{k}^{d-1}\rho > 1$ and $k^2\bar{k}^{d-2}\rho > 1$, COMPLETE outputs a mesh with spacing no smaller than local feature size. ■

We can optimize k accordingly. Clearly, we may as well have $k\bar{k}^{d-2} = \bar{k}^{d-1}$; otherwise, one constraint or the other is harder to satisfy. This argues for setting $k = (1-k)$, namely $k = 1/2$. Then, to ensure that $\bar{k}^{d-1}\rho > 1$, we need that $\rho \geq 2^{d-1}$. Note that the d term there is an upper bound: if our input is in ambient dimension d but only has features of dimension at most i, we can allow $\rho > 2^{i-1}$. In other words, for point clouds in arbitrary dimension, $\rho > 1$ as we saw two chapters ago; for segments, $\rho > 2$ as we saw in the previous chapter. For PLC inputs in dimension 3, $\rho > 4$.

Theorem 6.2.5 *A mesh over a domain Ω with a constraining PLC that is complete is size-conforming, quality, and respects the PLC, as long as $\rho > d$.*

Proof: Lemma 6.2.4 shows that when a vertex is added, its nearest neighbour is not much closer than local feature size allows. The standard inductive argument shows that this remains the case at the end of the algorithm. The arguments of Lemma 5.2.2 and Lemma 5.2.3 are agnostic of the types of features, so they apply for PLC inputs as well as for just features. Together, this implies that the mesh is size-conforming.

The lack of any gap of radius $\rho\,\mathrm{NN}(v)$ around any vertex implies that there is no Delaunay simplex in the output that has bad quality. The lack of any gap that denotes the encroachment of a boundary implies that each feature appears as a union of Delaunay facets, and thus the mesh respects the input. ■

6.3 Efficient algorithm

The key to efficient operation is the trick, again, of bucketing a series of COMPLETE and DISENCROACH tasks, along with COMPUTENN for input vertices whose nearest neighbour we do not exactly know. Work with key in $[l, \beta l)$ is processed essentially in parallel. It is critical that a vertex only spawn work in a later bucket, or at least that it only spawn a constant amount of work in the current or prior bucket. The bucketing constant β is determined by the proofs to ensure that this "growth" property holds. See Figure 6.3.

The same example we saw in the chapter on meshing with segments requires me again to define a DISENCROACH operation for efficient operation. The goal is to make sure that encroachment of one input feature on another does not spawn vertices with smaller nearest neighbour than the current bucket. The disencroachment procedure ensures this by making sure that areas with small lfs are not encroached by other features. If an area with large lfs is encroached, then the new vertex has a large nearest neighbour, so there is no problem. See Figure 6.4.

The DISENCROACH operation is size-conforming: if the ball b touches a vertex u that does not lie on f, then inserting x corresponds to an "encroach" completion on u, which we have already analyzed. Otherwise, the ball has radius $\mathrm{lfs}(x)$, so after yielding, the new vertex has $\mathrm{NN}(y) \geq \bar{k}^{d-2}\,\mathrm{lfs}(x)$. By Lipschitz, $\mathrm{lfs}(y) \leq \mathrm{lfs}(x) + ||xy|| \leq (2 - \bar{k}^{d-2})\,\mathrm{lfs}(x)$. Therefore, $\mathrm{lfs}(y) \leq \frac{2-\bar{k}^{d-2}}{\bar{k}^{d-2}}\,\mathrm{NN}(y)$.

Lemma 6.3.1 (DISENCROACH disencroaches.) *Set the priority for the* DISENCROACH*(c) operation to* $\frac{|c|}{2\eta}$ *for a given constant* η*. While processing a bucket* $[l, \beta l)$*, for any* x *that lies on a feature* f *and* $\mathrm{lfs}(x) < \eta l$*, then* x *has a vertex* u *on* f *such that* $||ux|| < \mathrm{lfs}(x)$ *where the inequality is strict.*

Proof: The claim is that if $\mathrm{lfs}(x) < \eta l$, then c was disencroached; contrapositively, if c has not yet been disencroached, then $\mathrm{lfs}(x) \geq \eta l$. We are processing a bucket $[l, \beta l)$, so for c to be unprocessed, $|c| \geq \frac{l}{\mathrm{key}}$. Given that c contains input, $\mathrm{lfs}(x) \geq |c|/2 \geq \frac{l}{2\mathrm{key}}$. We want to ensure that $\mathrm{lfs}(x) \geq \eta l$. Then it suffices to set the key such that $\frac{l}{2\mathrm{key}} > \eta l$. ■

6.3.1 Dependency paths are short

Lemma 6.3.2 (Quality insertions grow) *If* v *has a gap ball* $B(x, r)$ *with radius* $r > \rho\,\mathrm{NN}(v)$*, then after yielding to* y*,* $\mathrm{NN}(y) \geq \beta\,\mathrm{NN}(v)$*.*

Proof: Lemma 6.2.1 shows that $\mathrm{NN}(y) \geq \bar{k}^{d-1} r \geq \bar{k}^{d-1} \rho\,\mathrm{NN}(v)$. So long as $\boxed{\beta \leq \bar{k}^{d-1}\rho}$, the lemma is proved. ■

Lemma 6.3.3 (Encroachment from features) *Let* v *be a vertex with containing dimension less than* d*. Assume* v *has a gap ball* $B(x, r)$ *with* x *on a feature disjoint from the*

CHOOSESTEINERS($\mathcal{P}$, $\mathcal{X}$, QT, X, ρ)

P: map storing the correspondence between quadtree cells and vertices
S: map storing the correspondence between quadtree cells and segments
Q: work queue, bucketed with factor β
for each cell c in QT **do**
    If c contains a point p, enqueue a COMPUTENN(p) event with priority $|c|/2$
    If c contains a feature f, enqueue a DISENCROACH(c) event with priority $\frac{|c|}{2\eta}$
end for
while Q not empty **do**
    $w \leftarrow \text{POP}(Q)$
    if w is a COMPUTENN(p) event **then**
        Grow a ball $b = B(p, r)$ until it touches a disjoint segment or mesh vertex
        Add a COMPLETE(p) event with priority r
    else if w is a DISENCROACH(s, c) event **then**
        Disencroach s over c
        For each new vertex v (if any), INSERT(v)
    else if w is a COMPLETE(u) event **then**
        Complete u
        For each new vertex v (if any), INSERT(v)
    end if
end while

INSERT(v)

Add a COMPLETE(v) event with priority $\mathrm{NN}(v)$
for each v' in the link of v **do**
    Add a COMPLETE(v') event with priority $||vv'||$
end for

Figure 6.3: Algorithm to complete a mesh such that it is quality, size-conforming, and respects input points and segments. The map X does not change during the algorithm, since no new features are added. The map P changes as new vertices are added. To ensure short dependency paths, use gridding and colouring as described in Chapter 4. Set the constants $\beta = \rho/2^{d-1}$ and $\eta > \rho^2/2^d$ as described in the text. The parameter ρ must be strictly larger than 2^{d-1}.

DISENCROACH(c, f)

for each $x \in c \cap f$ **do**
  Grow a ball $b = B(x, r)$ until either $r = \mathrm{lfs}(x)$, or b touches a vertex
  If b touches a vertex u, and u lies on f, go on.
  Otherwise, insert x, possibly yielding to y.
end for

Figure 6.4: The disencroach operation, taking into account PLC features. The chief difference from the eponymous operation on segments is the need to possibly yield instead of inserting x.

feature on which v lies. Let y be the vertex that is inserted after yielding from x. Then

$$\mathrm{NN}(y) \geq \beta\, \mathrm{NN}(v)$$

Proof: On the one hand, we are processing v, so $\mathrm{NN}(v) < \beta l$. On the other, the distance from v to x establishes local feature size at x: $||vx|| \geq \mathrm{lfs}(x)$. The ball $B(x, ||vx||)$ is empty, so x must be in a cell not yet disencroached: $\mathrm{lfs}(x) \geq \eta l$. Finally, $\mathrm{NN}(y) \geq \bar{k}^{d-2}||vx|| \geq \bar{k}^{d-2}\eta l$. Setting η such that $\boxed{\eta > \frac{\beta^2}{\bar{k}^{d-2}}}$ provides the bound. Given the free variable, β is not constrained by this case. ■

Lemma 6.3.4 (Encroachment from space) *Let v be a vertex with containing dimension d, and v was inserted by a vertex u. Assume v has a gap ball $B(x, r)$ with x on a feature. Let y be the vertex that is inserted after yielding from x. Then*

$$\mathrm{NN}(y) \geq \beta\, \mathrm{NN}(u)$$

Proof: When u inserted v, it was because of a gap of radius $r_v \geq \rho\, \mathrm{NN}(u)$, since only quality can cause the creation of vertices in space. Given that v did not yield, $\mathrm{NN}(v) \geq kr_v$. Later, when v tried to insert x, it was the center of a gap of radius $r_x \geq \mathrm{NN}(v)$. If x yielded to y, then $\mathrm{NN}(y) \geq \bar{k}^{d-2}r_x$. Thus, $\mathrm{NN}(y) \geq \bar{k}^{d-2}k\rho\, \mathrm{NN}(u)$.

Assuming $\boxed{\beta \leq \bar{k}^{d-2}k\rho}$ the lemma is proved. ■

The proofs show that with $\beta = \min(\bar{k}^{d-1}\rho, \bar{k}^{d-2}k\rho)$ and $\eta > \frac{\beta^2}{\bar{k}^{d-2}}$, work only grows. At $k = 1/2, \beta = \rho/2^{d-1}$ and $\eta > \rho^2/2^d$. This constrains $\rho > 2^{d-1}$ as was already required for the algorithm to terminate.

6.4 The main result of the thesis

Theorem 6.4.1 *Given a PLC input $\mathcal{X}$ in fixed dimension d, with all input angles non-acute, and a user-desired Voronoi quality bound $\rho > 2^{d-1}$, the algorithms* BUILDQT *and* CHOOSESTEINERS *run in $O(n \lg L/s + m)$ time to produce a quality, size-conforming (and hence optimal-size) mesh whose Delaunay respects $\mathcal{X}$.*

Furthermore, upon adding or removing one of the PLC features $f \in \mathcal{X}$, which when present has m_f vertices on it, the algorithms respond in time $O(m_f \lg L/s)$.

Proof: Every event on the work queue can be processed in constant time: they all involve a series of $O(1)$ range queries, where each range query has radius $\Theta(\mathrm{lfs}(v))$ for COMPUTENN or COMPLETE events, or $\Theta(|c|)$ for DISENCROACH events. Lemma 4.3.3 shows that each query can therefore be completed in constant time. This bounds the total runtime.

Dynamic stability can be established by colouring the jobs on the work queue. After colouring jobs of size $[l, \beta l)$ at most a constant number of times, we are assured that no more will be created. Any vertex v is blamed for at most $O(1)$ vertices inserted in the same bucket as the bucket in which v was processed, or in any single later bucket. There are $O(\lg L/s)$ buckets, which proves the stability bound. ∎

Chapter 7

Closing Remarks

In this thesis, I implemented code for one meshing algorithm, and developed a new meshing algorithm. Future work includes implementing code for the new meshing algorithm, and developing new new meshing algorithms.

I see as the principal theoretical advance of this thesis the techniques for analyzing dependencies in a broad class of meshing algorithms. Beyond the dynamic algorithm I proposed, this almost immediately yields a parallel algorithm with logarithmic depth. The bounding box argument of Section 4.4 can be used to prove independence of a simplex from any activity in the mesh at some practical constant distance away, which will surely be useful in out-of-core and distributed meshing. Solving the kinetic problem also involves dependency tracking, of a sort closely related to those in the dynamic problem.

I have already discussed future goals of the present implementation of SVR. Summarizing, they are to make the implementation more robust in the face of slivers and illegal input, and to speed up the program by reducing the work it performs and by parallelizing it. The new dynamically-stable algorithm should also find its way onto a processor, since it should allow substantial speedup. In terms of remaining theoretical advances, none of the algorithms I mention properly handle input angles less than orthogonal. This is the most important problem blocking wider adoption of provable refinement algorithms at the moment: the algorithms are only proven to work over too small a class of inputs. Natively meshing curved surfaces is another important needed advance, though less critically so than small angles.

On a smaller scale of improvement, and nearer-term, it is extremely likely that the sizing arguments developed to prove that bounding boxes are not very expensive [HMP07a] can be adapted to reduce the multiplicative logarithmic factor in the response time from

$O(k \lg L/s)$ to an additive $O(k + \lg L/s)$. This is a purely theoretical question: no matter the true answer, the algorithm will run at the same speed. More importantly, currently the dynamic mesher uses $O(n \lg L/s + m)$ space to track dependencies for change propagation. It seems likely that this could be reduced to $O(m)$: On point-cloud input, the dynamic BUILDQT can reduce the size usage by storing only a count in each crowded cell of the number of points in the cell. When a new point is added to the cell, it does not affect whether it will be split, which means that we need not know the old points' coordinates. When an old point is removed, we can update the count. Anywhere that the count goes to one, the remaining point can be found by looking down one level in the quadtree. Performing similar optimizations should be still possible in the face of features and, hopefully, during CHOOSESTEINERS. Another near-term theoretical advance would be to reduce the requirement on the user-requestable quality bound from 2^{d-1} to $\sqrt{2}^{d-1}$. It is not entirely clear if this is purely a proof-theoretic problem, or one requiring some updates to the algorithm.

It is my hope that the new meshing algorithm will prove to allow better point location strategies than standard circumcenter refinement. Historically, each Steiner point was considered independent of any future points to be added. In the framework of my algorithms, we surround a vertex with new Steiner points, which suggests using some technique to optimize the placement of all the points at once. Particularly I hope that this can be used to eliminate slivers more effectively than in a one-at-a-time framework, since on choosing the points to insert, we can easily ensure that they do not form slivers among each other. Another use of the freedom my algorithm allows in point placement is, as Üngör and his students have been doing, to find positions for points that in practice allow the user to request much better radius/edge ratio than can be proved. It may even be possible to prove some results on these lines. Finally, in the dynamic algorithm, dependencies can *a priori* cross the Voronoi cell of the vertex being completed. It is likely that in practice, it should be possible to independently choose Steiner points on opposite sides of the vertex and thereby reduce the propagation during dynamic changes (albeit not provably).

New capabilities in meshing will open up new capabilities in scientific computing. Parallel, out of core, and distributed meshing will let engineers generate unstructured meshes even on supercomputers, temporarily slaking their thirst for additional cycles. More importantly, I believe that asymptotically faster generation of unstructured meshes should allow commodity hardware to solve substantially harder problems than has been possible in the past. Meshing is a major cost in many problems of interest to the community. Dynamically-stable meshing is of clear applicability to problems where the domain changes essentially discontinuously; here, I have shown how to remesh in logarithmic rather than merely near-linear time. Graphics and scientific computing are both highly interested in solving fluid-structure interaction problems, which are most naturally approached in a Lagrangian framework. Provable techniques based on the work here should enable kinetic meshing to avoid the problems of tangling and mesh quality loss that are currently the bane of moving mesh approaches.

Bibliography

[AAD07] Nina Amenta, Dominique Attali, and Olivier Devillers. Complexity of Delaunay triangulation for points on lower-dimensional polyhedra. In *SODA*, 2007.

[ABBT06] Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation. In *ACM-SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[ABT06] Umut A. Acar, Guy E. Blelloch, and Kanat Tangwongsan. Kinetic algorithms via self-adjusting computation. In *European Symposium on Algorithms*, 2006. See also CMU Computer Science Department Technical Report CMU-CS-06-115.

[Aca05] Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.

[AH06] Umut A. Acar and Benoît Hudson. Optimal-time dynamic mesh refinement: preliminary results. In *Fall Workshop on Computational Geometry*, Northampton, Mass., 2006.

[AHMP07] Umut A. Acar, Benoît Hudson, Gary L. Miller, and Todd Phillips. SVR: Practical engineering of a fast 3D meshing algorithm. In *International Meshing Roundtable*, pages 45–62, 2007.

[BA76] Ivo Babuška and A. K. Aziz. On the Angle Condition in the Finite Element Method. *SIAM Journal on Numerical Analysis*, 13(2):214–226, April 1976.

[BBCK05] Daniel K. Blandford, Guy E. Blelloch, David E. Cardoze, and Clemens Kadow. Compact Representations of Simplicial Meshes in Two and Three Dimensions. *International Journal of Computational Geometry and Applications*, 15(1):3–24, February 2005.

[BEG94] Marshall Bern, David Eppstein, and John R. Gilbert. Provably Good Mesh Generation. *Journal of Computer and System Sciences*, 48(3):384–409, June 1994.

[BET99] Marshall W. Bern, David Eppstein, and Shang-Hua Teng. Parallel construction of quadtrees and quality triangulations. *International Journal of Computational Geometry and Applications*, 9(6):517–532, 1999.

[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.

[BOG02] Charles Boivin and Carl F. Ollivier-Gooch. Guaranteed-quality triangular mesh generation for domains with curved boundaries. *International Journal for Numerical Methods in Engineering*, 55 (10):1185–1213, 2002.

[Bri93] E. Brisson. Representing geometric structures in d dimensions: Topology and order. *Discrete and Computational Geometry*, 9:387–426, 1993.

[BWHT07] Adam W. Bargteil, Chris Wojtan, Jessica K. Hodgins, and Greg Turk. A finite element method for animating large viscoplastic flow. *ACM Trans. Graph.*, 26(3), 2007.

[CCM+04] David Cardoze, Alexandre Cunha, Gary L. Miller, Todd Phillips, and Noel Walkington. A bezier-based approach to unstructured moving meshes. In *Symposium on Computational Geometry*, pages 71–80, 2004.

[CDE+00] Siu-Wing Cheng, Tamal Krishna Dey, Herbert Edelsbrunner, Michael A. Facello, and Shang-Hua Teng. Sliver Exudation. *Journal of the ACM*, 47(5):883–904, September 2000.

[CDL07] Siu-Wing Cheng, Tamal K. Dey, and Joshua A. Levine. A practical Delaunay meshing algorithm for a large class of domains. In *International Meshing Roundtable*, pages 477–494, 2007.

[CDR07] Siu-Wing Cheng, Tamal K. Dey, and Edgar A. Ramos. Delaunay refinement for piecewise smooth complexes. In *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1096–1105, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[CGS06] Narcis Coll, Marité Guerrieri, and J. Antoni Sellarès. Mesh modification under local domain changes. In *15th International Meshing Roundtable*, pages 39–56, 2006.

[Che87] L. Paul Chew. Constrained Delauany triangulation. In *in Proc. ACM Symposium on Comp. Geometry*, pages 213–222, 1987.

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

[Che97] L. Paul Chew. Guaranteed-Quality Delaunay Meshing in 3D. In *Proceedings of the Thirteenth Annual Symposium on Computational Geometry*, pages 391–393, Nice, France, June 1997. Association for Computing Machinery.

[Del34] Boris Nikolaevich Delaunay. Sur la Sphère Vide. *Izvestia Akademia Nauk SSSR, VII Seria, Otdelenie Matematicheskii i Estestvennyka Nauk*, 7:793–800, 1934.

[EGS05] David Eppstein, Michael T. Goodrich, and Jonathan Zheng Sun. The skip quadtree: a simple dynamic data structure for multidimensional data. In *21st Symposium on Computational Geometry*, pages 296–305, 2005.

[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 *STOC*, pages 273–277, Portland, Oregon, 2000.

[Gui98] Leonidas J. Guibas. Kinetic Data Structures—A State of the Art Report. In *Proceedings of the Third Workshop on Algorithmic Foundations of Robotics*, pages 191–209, Houston, Texas, 1998.

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

[HMP07a] Benoît Hudson, Gary L. Miller, and Todd Phillips. Bounding the cost of bounding boxes in mesh generation. In submission, 2007.

[HMP07b] Benoît Hudson, Gary L. Miller, and Todd Phillips. Sparse Parallel Delaunay Refinement. In *19th ACM Symposium on Parallelism in Algorithms and Architectures*, 2007.

[HPÜ05] Sariel Har-Peled and Alper Üngör. A time-optimal Delaunay refinement algorithm in two dimensions. In *21st Symposium on Computational Geometry*, pages 228–236, 2005.

[KCP06] Milind Kulkarni, L. Paul Chew, and Keshav Pingali. Using transactions in delaunay mesh generation. In *Workshop on Transactional Memory Workloads*, 2006.

[KFCO06] Bryan M. Klingner, Bryan E. Feldman, Nuttapong Chentanez, and James F. O'Brien. Fluid animation with dynamic meshes. In *Proceedings of ACM SIGGRAPH 2006*, August 2006.

[Kir83] David G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.

[Lab06] François Labelle. Sliver Removal by Lattice Refinement. In *Proceedings of the Twenty-Second Annual Symposium on Computational Geometry*. Association for Computing Machinery, June 2006.

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

[LL85] D. T. Lee and A. K. Lin. Generalized Delaunay triangulation for planar graphs. *Discrete and Computational Geometry*, 1985.

[LS07] François Labelle and Jonathan Richard Shewchuk. Isosurface stuffing: Fast tetrahedral meshes with good dihedral angles. In *ACM Transactions on Graphics*, 2007.

[LT01] Xiang-Yang Li and Shang-Hua Teng. Generating well-shaped Delaunay meshes in 3D. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 28–37. ACM Press, 2001.

[LTU99a] Xiang-Yang Li, Shang-Hua Teng, and Alper Üngör. Biting: Advancing front meets sphere packing. *International Journal of Numerical Methods in Engineering*, 1999.

[LTU99b] Xiang-Yang Li, Shang-Hua Teng, and Alper Üngör. Simultaneous refinement and coarsening for adaptive meshing. *Engineering with Computers*, 15(3):280–291, September 1999.

[MBF04] Neil Molino, Zhaosheng Bao, and Ron Fedkiw. A virtual node algorithm for changing mesh topology during simulation. In *SIGGRAPH*, 2004.

[McM70] Peter McMullen. The maximum numbers of faces of a convex polytope. *Mathematika*, 17:179–184, 1970.

[Mic97] Daniele Micciancio. Oblivious data structures: applications to cryptography. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 456–464, 1997.

[Mil04] Gary L. Miller. A time-efficient Delaunay refinement algorithm. In *Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 400–409, New Orleans, 2004.

[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.

[MTT+96] Gary L. Miller, Dafna Talmor, Shang-Hua Teng, Noel Walkington, and Han Wang. Control Volume Meshes Using Sphere Packing: Generation, Refinement and Coarsening. In *Fifth International Meshing Roundtable*, pages 47–61, Pittsburgh, Pennsylvania, October 1996.

[MTT99] Gary L. Miller, Dafna Talmor, and Shang-Hua Teng. Optimal coarsening of unstructured meshes. *Journal of Algorithms*, 31(1):29–65, Apr 1999.

[MTTW95] Gary L. Miller, Dafna Talmor, Shang-Hua Teng, and Noel Walkington. A Delaunay based numerical method for three dimensions: generation, formulation, and partition. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 683–692, Las Vegas, May 1995. ACM.

[MV00] Scott A. Mitchell and Stephen A. Vavasis. Quality mesh generation in higher dimensions. *SIAM J. Comput.*, 29(4):1334–1370 (electronic), 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.

[NvdS04] Han-Wen Nienhuys and A. Frank van der Stappen. A Delaunay approach to interactive cutting in triangulated surfaces. In *Fifth International Workshop on Algorithmic Foundations of Robotics*, 2004.

[Rup95] Jim Ruppert. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, 18(3):548–585, 1995. Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA) (Austin, TX, 1993).

[She96] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, Berlin, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[She97a] Jonathan Richard Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry*, 18(3):305–363, October 1997.

[She97b] Jonathan Richard Shewchuk. *Delaunay Refinement Mesh Generation*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1997. Available as Technical Report CMU-CS-97-137.

[She98a] Jonathan Richard Shewchuk. A Condition Guaranteeing the Existence of Higher-Dimensional Constrained Delaunay Triangulations. In *Proceedings of the Fourteenth Annual Symposium on Computational Geometry*, pages 76–85, Minneapolis, Minnesota, June 1998. Association for Computing Machinery.

[She98b] 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.

[She02] Jonathan Richard Shewchuk. What Is a Good Linear Element? Interpolation, Conditioning, and Quality Measures. In *Eleventh International Meshing Roundtable*, pages 115–126, Ithaca, New York, September 2002. Sandia National Laboratories.

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

[STÜ04] Daniel Spielman, Shang-Hua Teng, and Alper Üngör. Parallel Delaunay refinement with off-centers. In *EUROPAR*, 2004.

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

[Tal97] Dafna Talmor. *Well-Spaced Points for Numerical Methods*. PhD thesis, Carnegie Mellon University, Pittsburgh, August 1997. CMU CS Tech Report CMU-CS-97-164.

[Üng04] Alper Üngör. Off-centers: A new type of Steiner point for computing size-optimal quality-guaranteed Delaunay triangulations. In *LATIN*, pages 152–161, 2004.