THE UNIVERSITY OF CHICAGO

STABLE ALGORITHMS AND KINETIC MESH REFINEMENT

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY

DURU TÜRKOĞLU

CHICAGO, ILLINOIS

MARCH 2012

Anne ve Babama

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

First, I would like to thank my advisor Umut A. Acar; without his efforts and patience this thesis would not have been possible. He introduced me to his line of research, self-adjusting computation, and instilled an alternative mindset for designing dynamic and kinetic algorithms. Also, he taught me how to convey one's ideas to another person and cleared the path for me to learn how to think and write as a scientist. Although I still want to improve myself in this direction, I have learned a lot from him. Outside the academic world, through our conversations and discussions, I have gotten to know him personally and he has been a great friend to me. I am pleased to say I find myself lucky to have met such an advisor during my PhD program.

Next, I would like to thank Benoît Hudson for introducing me to the vast literature of meshing and for always being there when I needed to discuss geometry. Through these discussions, I started discovering the enjoyable aspects of computational geometry in general and meshing to be more specific. Also, I would like to thank Andrew Cotter for being both a collaborator and a friend. Without his efforts, our results would not have been implemented in such a short amount of time.

I would also like to thank Janos Simon, my advisor in the department. He has been very patient in listening to my academic interests and has guided and encouraged me in my decisions. I would like to thank Adam Tauman Kalai and Lázsló Babai for supporting and advising me throughout the initial years of the PhD program; their guidance allowed me to get my master's degree.

My years in Chicago would have been so different without my friends. I would like to thank Özgür especially, in all the stages in the department, from applying to the program to finally earning a doctorate. Also, I would like to thank Aytek, Hari, Soner, Sourav, and countless others for my enjoyable time in Chicago. I would like to thank Baran for understanding and supporting my enthusiasm to complete the PhD program. And I would like to thank Pamela not just for editing my thesis and improving my english, but also for sharing her sincere thoughts and perspectives on life. Outside Chicago, I would like to thank one of my best friends, Levent. For more than twenty years, he has always been supportive of me and he demonstrated the same support while I was

writing this thesis. I am greatly indebted to him. And I would like to thank Giovanna for all the joyful moments and for the support and encouragement she has given me for earning a doctorate.

Finally, I would like to dedicate this thesis to my father, Tuncer, and to my mother, Zafer, and thank them especially, for bringing me to life and helping me build a worldview that tremendously influenced the character I have now. And I would like to thank my sisters, Banu and Ebru, both of whom were like mothers to me when I was a boy. They always assisted me in my endeavors and supported my decisions at every stage of my life.

ABSTRACT

In many applications there has been an increasing interest in computing certain properties of a changing set of input objects where the changes may be of *dynamic* nature in terms of insertions and deletions of an input object, or of *kinetic* nature in terms of continuous motion of these objects. For solving problems that involve dynamic or kinetic modifications to the input, one needs to first solve the *static* version of the same problem where no modifications are allowed, and then develop efficient update algorithms for handling various changes to the input. For developing dynamic and kinetic update algorithms, Acar et al. recently proposed a framework called *self-adjusting computation*. Given a dynamic or kinetic problem, the principal algorithmic technique of the self-adjusting computation framework, called *change propagation*, uses a static solution to the problem to automatically generate a dynamic or a kinetic update algorithm. The efficiency of their update algorithm directly depends on the stability of the static algorithm: a static algorithm is *stable* if its executions with similar inputs produce outputs and intermediate data that are different only by a small fraction. Under this framework, designing an efficient update algorithm can therefore be reduced to designing a stable static algorithm.

Motivated by the self-adjusting computation framework, we follow a stable design approach in this thesis. We first design static algorithms that are stable, and then present update algorithms that are in the form of change propagation and guarantee efficient responses to dynamic and kinetic changes. We apply this approach for solving several open problems in computational geometry. First, we propose a robust motion simulator and experimentally evaluate its effectiveness on kinetically maintaining convex hulls in three dimensions. Then, we consider the *mesh refinement* problem and provide update algorithms that dynamically and kinetically maintain quality meshes.

Mesh refinement is an essential step in many applications in scientific computing, graphics, etc. The idea behind mesh refinement is to break up a physical domain into well-shaped discrete elements, e.g., almost equilateral triangles in two dimensions, so that certain functions defined on the domain may be computed approximately by considering these discrete elements. The refinement

process is carried out by inserting additional *Steiner* points into the given point set, taking care to insert a small number of them. This problem has been studied extensively in the static setting with several recent results achieving fast runtimes. In the dynamic and kinetic settings, however, there has been relatively little progress. In this thesis, we propose efficient solutions in both settings: in the dynamic setting, we design a dynamic algorithm for the closely related problem of well-spaced point sets in arbitrary dimensions; in the kinetic setting, we propose the first kinetic data structure for maintaining quality meshes of continuously moving points on a plane.

The results we present in this thesis demonstrate that the stable design approach not only provides an alternative perspective in designing dynamic and kinetic algorithms, but also transfers the inherent complexity of the update algorithms to the stable design and analysis of a static algorithm. This in turn strengthens the connection between static algorithms and dynamic and kinetic algorithms assisting us to solve several open problems.

CHAPTER 1

INTRODUCTION

Increasingly, many applications in computer science require computing certain properties of an input that changes over time. For example, the information that is available through the internet changes every minute, therefore a search engine must revise its database and respond to user queries with updated content in just a matter of seconds. Another example is the collision-detection problem where one needs to model two moving objects and determine whether the two would hit each other, typically by checking continuously if the two objects intersect. A third example is the kinetic maintenance of an accurate model of the continuously changing atmosphere for understanding the complex processes that affect our climate globally. For such problems, solutions vary depending on the nature of changes that the input goes through. The type of these changes classifies the problems (and their solutions) as *dynamic* or *kinetic*. In a *dynamic* problem, the input is modified by insertions and deletions in a discrete fashion. On the other hand, in a *kinetic* problem, the size of the input does not change, but instead, the objects that constitute the input move. For most such dynamic and kinetic problems, the underlying *static* problem, where the input is not allowed to change, is well understood but the dynamic and kinetic versions are much harder to develop. We need to dedicate special attention, therefore, to solving the nonstatic versions of even those problems whose static versions have well-established results. Towards this goal, over the past three decades, certain algorithmic techniques have been proposed and successfully applied to obtain theoretical guarantees over a range of problems. Our understanding of dynamic and kinetic problems, however, has not reached the level of our understanding of static problems.

In the dynamic setting, the objective is to design algorithms and data structures that can answer certain queries efficiently as the input changes dynamically due to insertions and deletions. In general, given a set of objects, a dynamic solution is composed of a data structure that represents these objects, a construction algorithm that initially creates the data stucture, an update algorithm that modifies the data structure upon a dynamic change to the input, and certain algorithms that

answer the queries that are of interest. The effectiveness of a solution is measured by the space required to store the data structure and by the runtime efficiency of the algorithms proposed. The attempts to solve a given dynamic problem often aim towards designing a data structure that can be maintained efficiently under dynamic changes, usually in the form of a single insertion or a single deletion. In certain cases, handling deletions is much harder than handling insertions (or vice versa), motivating many people to consider solutions for the partial problems. Namely, *incremental* problems consider dynamic changes only in the form of insertions, and *decremental* problems consider dynamic changes only in the form of deletions [26, 31].

In the kinetic setting, the *kinetic data structures* (KDS) framework, proposed by Basch et al. [18, 19, 34], sets forth a unified design scheme for solving kinetic problems and provides four criteria for measuring a solution's effectiveness. A kinetic data structure consists of a data structure that represents the property of interest being computed, and a set of *certificates* that validate the property so long as the outcomes of the certificates remain the same. When the outcome of a certificate changes—a certificate failure, in other words—the data structure updates the computed property and the set of certificates validating the updated property. A KDS is called *local* if each input object participates in a small number of certificates, *compact* if the total number of certificates is small, *responsive* if the data structure can be updated quickly after a certificate failure, and *efficient* if the number of certificate failures over a period of time is not significantly larger than the number of combinatorial changes in the property computed in that period.

Recently, Acar et al. proposed a framework called *self-adjusting computation* that offers an alternative line of thought on algorithms that are required to handle several types of modifications, including dynamic and kinetic [3, 4]. Given an algorithm that solves the static version of a dynamic or kinetic problem, the principal algorithmic technique of this framework, called *change propagation*, generates an update algorithm for the same problem using the given static algorithm. The effectiveness of the update algorithm generated by this approach directly depends on the *stability* of the static algorithm: informally, an algorithm is called *stable* if its executions with two

similar inputs produce outputs and intermediate data that are different only by a small fraction, i.e., the execution remains mostly unchanged. Therefore, using change propagation, the problem of designing an efficient update algorithm can be reduced to the problem of designing a stable construction algorithm.

In this thesis, motivated by the self-adjusting computation framework, we strive for designing stable construction algorithms and develop update algorithms in the form of change propagation for solving certain dynamic and kinetic problems in computational geometry, namely, kinetic convex hulls and dynamic and kinetic mesh refinement.

- We extend to three dimensions a previous result of Acar et al. on kinetically maintaining convex hulls in two dimensions [8]. The result is twofold: first, we propose a robust motion simulator; second, we evaluate the effectiveness of this simulator by designing a kinetic algorithm for maintaining convex hulls in three dimensions (Chapter 2).

- We improve the algorithm proposed by Hudson [43] for dynamically updating meshes, or more specifically, well-spaced point sets. Our algorithm reduces the runtime and memory requirements by a logarithmic factor and it is easier to implement (Chapter 3).

- We propose the first kinetic data structure for solving the problem of kinetic mesh refinement on a plane (Chapter 4).

Our results demonstrate that the stable design approach strengthens the connection between the construction and the update algorithms by transferring the inherent complexity of the dynamic and kinetic update algorithms to the stable design and analysis of a static construction algorithm.

In the rest of this chapter, we introduce dynamic algorithms and data structures, kinetic data structures framework, self-adjusting computation framework, and the mesh refinement problem. Finally, we summarize our results.

1.1 Dynamic Algorithms and Data Structures

Classically, problems in computer science require computing certain properties of a given input. In many applications, however, we need to compute these properties once again after some *dynamic* modifications—insertions and deletions—are applied to the input. For such applications, recomputing these properties "from scratch" may be the best approach we can hope for, but in many instances, it may be possible to take advantage of the already structured data by running a more efficient update algorithm for putting these insertions and deletions into effect. A common example is sorting a given set of numbers, and facilitating insertion and/or deletion of numbers while maintaining a sorted representation of the resulting set. The standard solution of this problem is to maintain a balanced binary tree for representing the sorted list of numbers. The initial run, i.e., the construction, takes $O(n \log n)$ time, and each insertion or deletion takes just $O(\log n)$ time; here n represents the size of the resulting set of numbers, and the update algorithm makes it n times faster to insert or delete a number.

In the example above, we know the objective function ahead of time (sorting). It is possible that we do not know the objective function ahead of time, in which case it would be desirable to answer certain queries as the input undergoes dynamic changes. A simple example is a search engine. The internet goes through dynamic changes as some new sites are built, others cease to exist, links between sites are added and/or removed, and the content of the sites is changed. A successful search engine retrieves the relevant information in seconds as it receives new search queries. For efficiency, any search engine must maintain a current representation of the web and update this representation as the web dynamically changes.

Generalizing these examples, we can formulate a dynamic problem as follows: given a set of input objects, the goal is to design a data structure that represents these objects, a construction algorithm that initially creates the data structure, an update algorithm that modifies the data structure upon a dynamic change to the input, and certain algorithms that answer the queries that are of interest. Moreover, construction and update algorithms must respectively compute and update the

value of any objective function known in advance. To measure the effectiveness of a given dynamic algorithm and data structure, the standard analysis involves considering the space required to store the data structure and the runtime efficiency of the algorithms proposed (construction, update, and query). Since designing efficient dynamic algorithms and data structures has proven to be a difficult task, it is widely accepted to study restricted problems. Some algorithms and data structures called *semidynamic—incremental* or *decremental* to be more specific—support only insertions or only deletions to the input, while algorithms called *fully dynamic* support both kinds of dynamic changes, insertions and deletions. We refer the reader to survey articles [26, 31] for examples of dynamic algorithms and data structures.

For the problems we consider in this thesis, instead of designing specific update algorithms, we design stable construction algorithms and generate update algorithms in the form of change propagation by following the principles of the self-adjusting computation framework. For improved efficiency, we design certain dynamic algorithms and data structures as tools for providing solutions to partial problems, and integrate these tools into the change-propagation mechanism. We provide examples of such dynamic algorithms and data structures in each chapter. In Chapter 2, we use an incremental construction algorithm and when a certificate fails, our kinetic update algorithm (in the form of change propagation) updates this dynamic construction. In Chapter 3, we use a dynamic point location data structure, which internally is not self adjusting. In Chapter 4, we organize the construction of the mesh in levels and when a certificate fails, our kinetic update algorithm (in the form of change propagation) applies a dynamic update algorithm at each level.

1.2 Kinetic Data Structures

In many areas of computer science (e.g., graphics, scientific computing), we must compute with continuously moving objects. For these objects, *kinetic data structures* framework [19, 34] allows efficient computation of their properties as they move. A kinetic data structure (KDS) consists of a data structure that represents the property of interest being computed, and a proof of that property.

The proof is a set of *certificates* that validate the property in such a way that as long as the outcomes of the certificates remain the same, the combinatorial property being computed does not change. To simulate motion, a kinetic data structure is combined with a motion simulator that monitors the times at which certificates *fail*, i.e., the value of certificates change; a certificate failure is also known as a *kinetic event*. Upon a kinetic event, the motion simulator notifies the data structure representing the property. The data structure then updates the computed property and the proof by deleting the certificates that are no longer valid and by inserting new certificates. The performance of a KDS is analyzed according to four properties: compactness, responsiveness, efficiency, and locality.

- *Compactness* requires the number of the certificates to be not much larger than the smallest size of the proof that certifies the property the KDS computes.

- *Responsiveness* requires the data structure to respond to events in a small amount of time; in many instances, we would like the update time to be bounded by a polylogarithmic function in the size of the input.

- *Efficiency* is related to the total number of events processed. In a kinetic simulation, we can categorize events as being *external* or *internal*. An event is *external* if the property computed by the KDS changes with the certificate failure; conversely, an event is *internal* if the certificate failure does not affect the property being computed by the KDS. A kinetic data structure is then called *efficient* if the number of internal events is asymptotically of the same order as, or at most a polylogarithmic factor larger than, the number of external events.

- To handle discrete changes to the data structure such as motion plan changes or dynamic insertions and deletions, *locality* is the criterion that requires each input point to participate in a small number of certificates; generally, this bound is desired to be a polylogarithmic bound.

Since their introduction, many kinetic data structures have been designed and analyzed. See survey articles [15, 35] for references to descriptions of various kinetic data structures. Yet, several difficulties remain in making them effective in practice [15, 38, 62, 61]. Furthermore, many problems, especially in three dimensions, remain essentially open [35]. One set of difficulties stems from the fact that current KDS update mechanisms depend on the assumption that the update is invoked to repair a single certificate failure [15]. This assumption requires a precise ordering of the certificate failure times so that the earliest can always be selected, possibly requiring exact arithmetic. The assumption also makes it particularly difficult to deal with simultaneous events.

In Chapter 2, we propose another approach to advancing time. Our approach is a hybrid between kinetic event-based scheduling and classic fixed-time sampling. The idea is to partition time into a lattice of intervals of fixed size, and identify events only to the resolution of an interval. If many events fall within an interval, they are processed as a batch without regard to their ordering. More specifically, in exact separation, one finds smaller and smaller intervals (e.g., using binary search) until all events fall into separate intervals. In our case, once we reach the lattice interval, we can stop without further separation, thus avoid potentially expensive separation costs.

In Chapter 4, for the kinetic mesh refinement problem, we focus on the construction and the update algorithms rather than the motion simulator, since the most challenging aspects of this problem have been geometric in nature: the selection of Steiner points that fit into the kinetic setting and the guarantee of their well-spacedness. Therefore, we use the original KDS framework; however, since our update algorithm is a change-propagation algorithm, our hybrid approach in Chapter 2 can be applied to support simultaneous updates for motion simulation.

1.3 Self-Adjusting Computation

In many applications that process dynamically changing data, small changes to input data often require small changes to the output. This observation creates the potential for designing efficient dynamic algorithms for updating the output rather than recomputing it from scratch after each

change. As mentioned earlier, designing dynamic algorithms that exploit this potential turns out to be difficult. Nevertheless, efficiency being the most important concern, the same observation also creates an opportunity for a generic approach to designing dynamic algorithms. More specifically, given a static algorithm and a dynamic modification to its input, the same algorithm can be re-executed without having to duplicate parts of the execution by storing certain trace information during the computation. Indeed, Acar et al. have shown that this idea can be used to automate the process of translating a static algorithm into a dynamic one. Their approach, which is based on a combination of dynamic dependence graphs [6] and memoization (function caching), is called *self-adjusting computation* [3]. Self-adjusting computation has been realized by extending the C [39, 41, 40] and ML [13, 52, 51, 50] programming languages.

In self-adjusting computation, the main idea is to generate a computation dependence graph while running an algorithm A on the input data. Each node of this graph stores information representing the code that needs to be re-executed if its current execution becomes invalid when input changes. After any such change, the update mechanism, called *change propagation*, propagates the changes through this graph, updating the parts of the graph and the computation that depend on them, ultimately updating the output. During propagation, memoization enables the update mechanism to identify unchanged portions of the graph, thus avoiding the need to re-execute them and instead spending time only on the portions of the graph that need to be updated. The change-propagation mechanism updates the dependence graph so that it always becomes isomorphic to the dependence graph that would have been generated by running algorithm A on the modified input. Since change propagation virtually transforms the dependence graph before the input modification into the one after, the time taken by change propagation can therefore be stated in terms of the distance between the two dependence graphs. Consequently, the efficiency of the update algorithm depends on how stable the dependence graphs of algorithm A are with respect to input changes [51], making it critically important how dependencies are structured during the execution of the algorithm.

By default, the change-propagation mechanism traces dependencies of an algorithm at the level of data access in terms of reads and writes of a memory cell. With recent advances in self-adjusting computation [7], it is now possible to also trace dependencies at the level of data structures in terms of query and update operations on them. So-called *traceable data types* do this, by directly handling input changes and propagating changes externally only at the interface. Intuitively, a traceable data type can be thought as a dynamic data structure integrated into the self-adjusting computation framework. The use of such specialized dynamic data structures reduces space and time overhead; for some, it improves the runtime of change propagation asymptotically and greatly simplifies the design of stable algorithms.

Besides these advantages, using traceable data types also makes it possible to develop self-adjusting programs for kinetically changing data using the kinetic data structures framework. Simply put, the kinetic data structures framework handles continuous changes to data in a discrete and dynamic fashion through the use of certificates. The certificates, which prove that the computation is valid as long as certificate outcomes do not change, can be considered as traceable data types, allowing change propagation to trace dependencies at the level of certificates. Instead of tracing dependencies at the level of more primitive time data (which continually change and therefore are not suitable for self-adjusting computation), at the level of certificates, self-adjusting programs remain oblivious to time changes. To identify the times when the outcome of a certificate changes, these programs then employ a motion simulator that implements a traceable priority queue for storing certificate failure times. Therefore, self-adjusting programs become capable of kinetizing static algorithms through the use of traceable data types. Because of their inherent flexibility, kinetic self-adjusting programs can also overcome some of the restrictions that currently limit many kinetic data structures, such as processing multiple certificate failures. We discuss the kinetic aspects of self-adjusting computation in more detail in Chapter 2.

Motivated by the advantages of self-adjusting computation framework, we design stable algorithms, i.e., algorithms that have stable dependence graphs, and use the change-propagation

technique to develop update algorithms. In the rest of this section, we discuss change propagation, its runtime analysis, and traceable data types.

1.3.1 Change Propagation

A self-adjusting program, which is based on a static algorithm, when run initially, constructs a trace of its execution in the form of a dynamic dependence graph. This graph stores the function-call tree of the execution, and the data dependencies between the function calls and the data, i.e., the memory accesses performed in each of the function calls. Also, while the trace is being constructed, these function calls get assigned two time stamps indicating the beginning and the end of the call, so that the function calls can be ordered sequentially together with the caller-callee relationship information. Using the dynamic dependence graph and the ordering on the function calls, the self-adjusting computation framework defines a generic update mechanism for updating the execution of the program after some modifications are applied to its input data. This update mechanism is called *change propagation*.

In a nutshell, the change-propagation mechanism propagates input changes throughout the dependence graph, identifies and stores affected function calls in a priority queue, and processes them in execution order. More specifically, it identifies those function calls that read data that has been changed because of propagation, marks them as affected, and re-executes them in increasing time order. When re-executing a function call, the propagation mechanism uses memoization to identify and reuse any unchanged function calls made within the time frame of the function call being re-executed. At the end of re-execution, it removes the function calls that were not reused within the same time frame. This process of re-executing a function call further invalidates some other data; by following the dependencies in the graph, the propagation mechanism identifies other affected function calls and puts them into the priority queue to be re-executed later. Throughout this process, the propagation mechanism updates the dependence graph and the output as necessary; propagation ends when all affected function calls are re-executed.

In this thesis, we design update algorithms in the form of change propagation. However, instead of referring to the description contained in this section, we explain in detail the dependence propagation for each of the problems we consider in their respective chapters.

1.3.2 Trace Stability

Acar [3] provides correctness and runtime analyses of the change propagation mechanism in its most general form. Correctness relies on an important isomorphism property: after change propagation updates the current execution trace of an algorithm, the updated trace becomes isomorphic to the execution trace of the same algorithm with the modified input. Furthermore, this isomorphism property together with memoization form a basis for bounding the update runtime. Intuitively, if the traces of any two executions of a given algorithm with inputs that differ by a single modification are similar, then change propagation should not take too much time. To realize this intuition, self-adjusting computation framework defines a labelling scheme for matching and memoizing function calls that are common to both executions. Using memoization, change propagation mostly needs to remove or execute the unmatched function calls, i.e., the function calls that are not common to both executions. Then, an algorithm is called $O(f(n))$-*stable* if the trace distance between any two executions with inputs that differ by a single modification is bounded by $O(f(n))$, where the trace distance is defined as the time to execute the unmatched function calls. The interesting result is that change propagation takes $O(f(n) \log f(n))$ time, provided that the algorithm is *monotone*.

In order to define the notion of monotonicity, let us first define two function calls in two different executions related with a dynamic input change as *cognates* if their labels and the caller callee relationships are the same. Provided that all function calls in a trace are labelled uniquely, an algorithm is called *monotone*, if all cognates are executed in the same order in any two executions (of the algorithm) that are related with a dynamic input change. During change propagation of monotone algorithms, all cognates, if needed, can be reused via memoization, and memoizing a function call does not remove any cognates from the trace. Therefore, time is spent only on removing and

executing function calls that do not have cognates; this argument supports the runtime proof of change propagation. For further details, we refer the reader to the dissertation of Acar [3].

In this thesis, we do not refer to the proofs summarized in this section but explain in detail the stability analysis for each of the problems we consider.

1.3.3 Traceable Data Types

By default, self-adjusting computation techniques rely on tracing dependencies at the data level by recording the memory operations, both reads and writes. When designing certain data structures, this fine-grained approach can be problematic, because updates to the input may significantly change internals of the data structure, even though the set of changes that propagate to the interface is small. With recent advances in self-adjusting computation [7], we can overcome this problem by extending the tracing of dependencies at the level of arbitrary data types (structures). This extension involves developing traceable versions of these data types, which are called *traceable data types*. For these data types, dependencies must be traced directly, so that the change-propagation mechanism can handle more general dependence tracing. In other words, one can think of traceable data types as dynamic data structures that are integrated into the self-adjusting computation framework.

With traceable data types, i.e., integrated dynamic data structures, the change-propagation mechanism remains insensitive to the specifics of the data structure, and the update algorithm of the data structure itself handles the changes within the data structure. Using such specialized dynamic data structures within the self-adjusting computation framework reduces the number of dependencies that are traced, thus reducing space and time overhead. It can even improve the runtime of change propagation asymptotically because some data structures display suboptimal behavior when made self-adjusting at the memory cell level (e.g., priority queues). In addition to improving performance, using dynamic data structures can also greatly simplify the design of algorithms with stable dependence graphs, since one needs to consider dependencies only between the

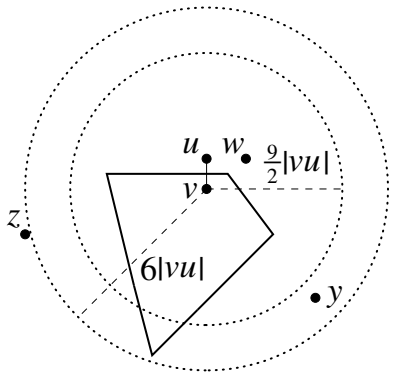
operations on the data structure instead of all the memory accesses inside of it. In summary, integrating certain dynamic data structures and tracing dependencies at varying levels of the algorithm proves useful for efficiency and stability. In this thesis, we employ certain dynamic data structures (mentioned at the end of Section 1.1) in this fashion and integrate them into the change-propagation mechanism.

1.4 Mesh Refinement

Meshing is an essential step in many applications such as physical simulations, surface reconstruction, computer graphics, etc. The idea behind meshing is to break up a physical domain into discrete elements—e.g., triangles in two dimensions; more generally, *simplices* in n-dimensions—so that certain functions defined on the domain may be computed approximately by considering the discrete elements. If the accuracy achieved on a given mesh is not satisfactory, one may want to enhance the quality of the mesh and therefore increase the precision of the computation. The common procedure to enhance mesh quality is to insert additional so-called *Steiner* points into the mesh, taking care to insert as few Steiner points as possible to keep output size small. This procedure is known as *mesh refinement*.

In general, the input to a refinement procedure describes a geometric object (e.g., a plane, a lake) inside a domain, and the output is a refined mesh of the object and the domain. Common practice in mesh refinement sets the domain to be a large enough bounding box so that the domain itself is simple and the applications using the output mesh receive minimal influence from outside the domain. The refinement procedure outputs a mesh by partitioning the domain and the input object into simplices that should be "well shaped" and therefore easy to manipulate. Since the output mesh becomes a part of the input to the applications, it is also important that the size of the output mesh be small enough and the smallest sized simplices in the mesh be large enough. A mesh refinement algorithm, therefore, needs to take these quality criteria into consideration when generating a mesh.

Figure 1.1: Let $\mathcal{M} = \{v, u, w, y, z\}$. The nearest-neighbor distance of v, $\mathrm{NN}_{\mathcal{M}}(v)$, is $|vu|$. The polygon with solid boundary lines depicts the Voronoi cell of v, $\mathrm{Vor}_{\mathcal{M}}(v)$. The vertex v is 6-well-spaced, but not $\frac{9}{2}$-well-spaced.



Since the specific quality criteria for the mesh refinement procedure vary greatly depending on the application (also on the functions to be approximated), instead of considering the diverse set of quality criteria and investigating refinement procedures that are appropriate for different types of applications, we restrict our attention to a specific one, known as *well-spacedness*. In this thesis, we theoretically approach the mesh refinement problem, more specifically, the well-spaced point sets problem, and abstract it out from the applications as a goal on its own. Using stable algorithms, we propose solutions to this problem in dynamic and kinetic settings. In the rest of the section, we introduce the well-spaced point sets problem, summarize our Steiner point selection schemes, and provide a brief survey of mesh refinement.

1.4.1 Well-Spaced Point Sets

Given a domain Ω in $\mathbb{R}^d$, we call a set of points $\mathsf{M} \subset \Omega$ *well-spaced* if for each point $p \in \mathsf{M}$ the ratio of the distance to the farthest point of Ω in the Voronoi cell of p divided by the distance to the nearest neighbor of p in M is small [66]. Well-spaced point sets are strongly related to meshing and triangulation for scientific computing, which require meshes to have certain qualities. In two dimensions, a well-spaced point set induces a Delaunay triangulation with no small angles, which is known to be a good mesh for the finite element method. In higher dimensions, well-spaced point sets can be postprocessed to generate good simplicial meshes [22, 53]. The Voronoi diagram of a well-spaced point set is also immediately useful for the control volume method [55].

Given a domain $\Omega \subset \mathbb{R}^d$, we define the well-spaced point set problem as constructing a well-spaced output $\mathsf{M} \subset \Omega$ that is generated by inserting Steiner points into a given set of input points $\mathsf{N} \subset \Omega$. We define the *geometric spread*, denoted by Δ, to be the ratio of the diameter of N to the distance between the closest pair in N. The geometric spread is a natural measure that yields a $\log \Delta = O(\log |\mathsf{N}|)$ bound when input points are at fixed-point coordinates represented by log-size words. Given N as input, our algorithm constructs a well-spaced output $\mathsf{M} \subset \Omega$ that is a superset of N. We use the term *point* to refer to any point in Ω and the term *vertex* to refer to the input and output points. Consider a vertex set $\mathcal{M} \subset \Omega$. The *nearest-neighbor distance* of v in $\mathcal{M}$, written $\mathrm{NN}_{\mathcal{M}}(v)$, is the distance from v to the nearest other vertex in $\mathcal{M}$. The *Voronoi cell* of v in $\mathcal{M}$, written $\mathrm{Vor}_{\mathcal{M}}(v)$, consists of points $x \in \Omega$ such that for all $u \in \mathcal{M}, |vx| \leq |ux|$. Following Talmor [66], a vertex is *ρ-well-spaced* if the intersection of its Voronoi cell with Ω is contained in the ball of radius $\rho\,\mathrm{NN}_{\mathcal{M}}(v)$ centered at v; $\mathcal{M}$ is ρ-well-spaced if every vertex in $\mathcal{M}$ is ρ-well-spaced. Figure 1.1 illustrates these definitions.

Well-spaced point sets problem, by definition, emphasizes the Voronoi diagram of a given point set, instead of drawing direct attention to the triangles that constitute a mesh. However, Voronoi diagrams are closely related to Delaunay triangulations; in fact, they are the dual graphs of each other provided that the points are in general position. More formally, given an input point set N, the *Delaunay triangulation* of N is defined as a triangulation that satisfies the condition that the circumscribing circle of any triangle of the triangulation does not contain a point of N in its interior. As part of the duality property, every edge $\{u, v\}$ of the Delaunay triangulation corresponds to the relation that the Voronoi cells of u and of v are adjacent; such vertices are also called *Voronoi neighbors*. Furthermore, every triangle $\{u, v, w\}$ of the Delaunay triangulation corresponds to the relation that the Voronoi cells of u, of v, and of w intersect at a single point which is referred to as a *Voronoi node*; this point coincides with the circumcenter of the triangle. Because of this strong connection between Voronoi diagrams and Delaunay triangulations, we sometimes shift the attention of the reader from one setting to the other.

1.4.2 Steiner Point Selection

Regardless of runtime considerations, the fundamental question in mesh refinement is where to insert the Steiner vertices [63]. Traditional solutions place Steiner vertices as far from any other vertex as possible, namely, at the circumcenters of Delaunay simplices (equivalently, at the nodes of the Voronoi diagram) [24, 60, 63, 44]. In two dimensions, Har-Peled and Üngör instead place Steiner vertices close to a vertex, but not too close: at the *off-centers* [42]. This local scheme allows them to build a data structure that can locate the off-centers in constant time. Jampani and Üngör extends this idea to three dimensions using *core disks* [47]. Another approach based on *quadtrees* generates an appropriately refined quadtree over the input points and adds the corners of the quadtree squares as Steiner points [20].

In our proposed algorithms for mesh refinement, we use varyious techniques for picking Steiner vertices. For the dynamic problem, we define a local picking region that includes those Steiner vertices proposed by the approach of Üngör et al. [42, 47]. For the kinetic problem we use a spherical version of the quadtree method to pick Steiner vertices. We explain the details of our Steiner vertex selection schemes in their respective chapters.

1.4.3 History of Mesh Refinement

The mesh refinement problem has been studied since the invention of computers, receiving attention both in practice and in theory. In practice, researchers and engineers have been considering specific types of polygon meshes, e.g., uniform, quadrilateral, hexagonal, and refining or optimizing them mostly using heuristic techniques without regards to theoretical analyses of runtime or mesh quality. Employing heuristic techniques has continued to be the customary approach in many applications to this day, because establishing a theory for assessing mesh quality that is appropriate for various kinds of applications has proven to be difficult, if not impossible. The effectiveness of the meshes used in these applications, therefore, has been analyzed experimentally. On the other hand, focusing on applications that rely on widely used methods such as the finite element

method or the control volume method, theoreticians were able to identify certain quality measures for triangular meshes. Since the late 1980s, starting with Chew [24], several results in theory have achieved guarantees for mesh quality and for fast runtimes. We survey these theoretical results in the rest of this subsection.

The first theoretical result in mesh refinement, due to Chew [24], is motivated by the observation that finite element applications achieve more accurate solutions when the triangles constituting a mesh are closer to being equilateral. He devised an approach to generate an almost uniform triangular mesh in two dimensions and to ensure that all angles in the resulting triangulation are between 30° and 120°. This guarantee on the angles also identified and set the mesh quality criterion for all other theoretical results to follow: a quality mesh should not contain *small angles*, thereby eliminating *large angles* as well. In two dimensions, this criterion directly corresponds to the well-spacedness criterion defined in terms of Voronoi diagrams.

The next milestone result is due to Ruppert [60] for improving the size of the resulting meshes of Chew's algorithm and setting a framework for analyzing the size of a mesh. Observing that the meshes generated by Chew's algorithm are almost uniform and equally refined accross the whole domain, he came up with the idea to refine a mesh only where it is needed so as to avoid unnecessary Steiner point insertions. He adapted Chew's algorithm in this manner and proved that his algorithm outputs a *size-optimal* mesh, that is, the size of the output mesh is within a constant factor of the size of the smallest possible mesh that meets the same quality criterion. He showed that for achieving size-optimality, one can prove that the output mesh is size-conforming. Formally, given an input set N, an output $\mathsf{M} \supset \mathsf{N}$ is called *size-conforming* if there exists a constant c independent of N such that for all $v \in \mathsf{M}$, we have $\mathrm{NN}_{\mathsf{M}}(v) > c \cdot \mathrm{lfs}(v)$, where $\mathrm{lfs}(v)$, the *local feature size* of v, is defined to be the distance from v to the second-nearest vertex in N. These definitions lay the foundations of analyzing the size of a mesh, and in this thesis, we include size-optimality as an integral part of our objective for the mesh refinement problem.

Subsequent research by Chew [25] and Shewchuk [63] extended these results to higher dimen-

sions by bounding the radius/edge ratio in the output, which corresponds to the well-spacedness criterion. However, in dimensions three or higher, well-spacedness criterion does not guarantee that all angles are above some threshold. Even though the Voronoi diagram of a point set is well-spaced, the Delaunay triangulation may include *slivers*, which are almost-flat simplices that have dihedral angles close to 0° and 180°. Some techniques that eliminate slivers have been introduced in the last decade [30, 22, 53]; however, dealing with slivers remain an important challenge in both theory and practice.

In the last decade, several recent results have achieved fast runtimes [42, 44, 64]. The first important efficiency result, due to Spielman, Teng, and Üngör [65], was on parallel mesh refinement. By parallelizing Ruppert's method in two dimensions and Shewchuk's method in three dimensions, they proved that their parallel mesh refinement algorithm takes $O(\log^2 \Delta)$ parallel iterations. The same authors improved their result to $O(\log \Delta)$ parallel iterations by inserting off-centers as Steiner points [64]. This result also led Har-Peled and Üngör to propose and anaylze the first time-optimal Delaunay refinement algorithm in two dimensions [42]. Their algorithm is guaranteed to run in $O(n \log n + m)$ time where n and m are the input and output sizes respectively. Later, Hudson, Miller, and Phillips [44] extended this result to arbitrary dimensions by always maintaining a sparse mesh throughout their algorithm. They proved that their algorithm, called *Sparse Voronoi Refinement*, is guaranteed to run in $O(n \log \Delta + m)$ time.

All of the above results focus on the static setting. In both dynamic and kinetic settings, there has been relatively little progress on solving the mesh "maintenance" problem where one needs to refine as well as coarsen the mesh as the input changes dynamically or kinetically. In the dynamic setting, existing solutions either do not produce size-optimal outputs because they do not coarsen their outputs [23, 59], or they are asymptotically no faster than running a static algorithm from scratch [28, 54]. Similarly, in the kinetic setting, existing solutions either locally refine the current mesh without coarsening it, therefore, failing to maintain size-optimality [21, 56], or they throw away an almost quality mesh and generate a fresh one from scratch, possibly resulting in a whole

new set of triangles [17, 49]. In this thesis, we strive for keeping the current mesh intact as much as possible and replacing only parts of the mesh that need to be refined or coarsened.

1.5 Our Contributions

Using the stable design approach we advocate in this thesis, we offer solutions to the problems of kinetic convex hulls, and of dynamic and kinetic mesh refinement. We present either empirical evidence or theoretical guarantees for these solutions. We summarize our contributions as follows:

- *Kinetic Convex Hulls in 3D*

  We propose a robust motion simulation technique that combines kinetic event-based scheduling and the classic idea of fixed-time sampling. The idea is to divide time into a lattice of fixed-size intervals, and process events at the resolution of an interval. We apply this approach to the problem of kinetic maintenance of convex hulls in 3D—a problem that has been open since the 1990s. We evaluate the effectiveness of our proposal experimentally. Using this approach, we are able to run robust simulations consisting of tens of thousands of points.

- *Dynamic Well-Spaced Point Sets*

  We present a dynamic algorithm for the well-spaced point sets problem in any fixed dimension. Our construction algorithm generates a well-spaced superset in $O(n \log \Delta)$ time, and our dynamic update algorithm allows inserting and/or deleting points into/from the input in $O(\log \Delta)$ time, Δ being the geometric spread. We show that this update algorithm is efficient by proving a lower bound of $\Omega(\log \Delta)$ for a dynamic update. We also show that our algorithms maintain size-optimal outputs: the well-spaced supersets are within a constant factor of the minimum size possible. To the best of our knowledge, this is the first time-optimal and size-optimal algorithm for dynamically maintaining well-spaced point sets.

- *Kinetic Mesh Refinement in 2D*

  We provide a kinetic data structure (KDS) for the planar kinetic mesh refinement problem, which concerns computation of meshes of continuously moving points. Our KDS computes the Delaunay triangulation of a size-optimal well-spaced superset of a set of moving points with algebraic trajectories of constant degree. Our KDS satisfies the following criteria:

  - It is compact, requiring linear space in the size of the output.
  - It is local, involving an input point in $O(\log \Delta)$ certificates.
  - It is responsive, repairing itself in $O(\log \Delta)$ time per event.
  - It is efficient, processing $O(n^2 \log^3 \Delta)$ events in the worst case. This is optimal to within a polylogarithmic factor: we prove a lower bound of $\Omega(n^2 \log \Delta)$ in the worst case.

  In addition to satisfying these criteria, our KDS is also dynamic, responding to point insertions and deletions in $O(\log \Delta)$ time. To the best of our knowledge, this is the first KDS for mesh refinement.

1.5.1 References

The results presented in Chapter 2 are based on the following two articles: "Robust Kinetic Convex Hulls in 3D" published in the proceedings of the sixteenth annual European Symposium on Algorithms, with coauthors Umut A. Acar, Guy E. Blelloch, and Kanat Tangwongsan [9] and "Traceable Data Types for Self-Adjusting Computation" published in the proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation with coauthors Umut A. Acar, Guy E. Blelloch, Ruy Ley-Wild, and Kanat Tangwongsan [7].

The results presented in Chapter 3 are submitted to the journal of Computational Geometry Theory and Applications [10]. These results are based on the following two articles: "An Efficient

Query Structure for Mesh Refinement" published in the proceedings of the twentieth annual Canadian Conference on Computational Geometry coauthored with Benoît Hudson [46], and "Dynamic Well-Spaced Point Sets" published in the proceedings of the twenty-sixth annual Symposium on Computational Geometry with coauthors Umut A. Acar, Andrew Cotter, and Benoît Hudson [11].

The results presented in Chapter 4 are based on the article "Kinetic Mesh Refinement in 2D" published in the proceedings of the twenty-seventh annual Symposium on Computational Geometry with coauthors Umut A. Acar and Benoît Hudson [12].

CHAPTER 2

KINETIC CONVEX HULLS IN 3D

As mentioned in the introductory chapter, kinetic data structures (KDS) [19, 34] is a framework for computing properties of moving objects that has been applied to many problems in computational geometry. A KDS maintains a proof of the computed property via certificates, and it updates the property and its proof when kinetic events happen (when one of the certificates fails). In order to simulate motion, a KDS interacts with a motion simulator that monitors the times of kinetic events. The motion simulator continuously reports the upcoming kinetic events so that the KDS performs the necessary kinetic updates. Although there are no inherent restrictions on motion simulators, since almost all KDS update mechanisms can be invoked to repair only a single certificate failure [15], motion simulators must generally order the certificate failure times so that the earliest failure can always be selected. This requirement enforces exact separation of failure times, which can be costly due to exact arithmetic, and makes it particularly difficult to deal with simultaneous events. In this chapter, using self-adjusting programs, we demonstrate how to process multiple certificate failures simultaneously, and propose another approach to advancing time for robust motion simulation. We then design a kinetic self-adjusting algorithm for maintaining 3D convex hulls and evaluate our motion-simulation approach experimentally.

For motion simulation, our approach is a hybrid of kinetic event-based scheduling and classic fixed-time sampling. The idea is to partition time into a lattice of intervals of fixed size δ, and identify events only to the resolution of an interval. If many events fall within an interval, they are processed as a batch without regard to their ordering. As with kinetic event-based scheduling, we maintain a priority queue, but in our approach, the queue maintains nonempty intervals each possibly with multiple events. To separate events to the resolution of intervals, we use Sturm sequences similar to their use for exact separation of roots of a polynomial [36], but the fixed resolution allows us to stop the process early. More specifically, in exact separation, one finds smaller and smaller intervals (e.g., using binary search) until all events fall into separate intervals.

In our case, once we reach the lattice interval, we stop without further separation. This means that if events are degenerate and happen at the same time, we need not determine this potentially expensive fact.

For 3D kinetic convex hulls, we use a static randomized incremental convex hull algorithm [27, 58] and kinetize it using self-adjusting computation. To ensure that the algorithm responds to kinetic events efficiently, we make some small changes to the standard incremental 3D convex-hull algorithm. This makes progress on the problem of 3D kinetic convex hulls, which was identified in late 1990s [34]. To the best of our knowledge, computing the 3D kinetic convex hulls prior to this work was done by the kinetic Delaunay algorithm of the CGAL package [1], which computes the convex hull as a byproduct of the 3D Delaunay triangulation (of which the convex hull would be a subset). As shown in our experiment, this existing solution generally requires processing many more events than necessary for computing convex hulls.

We present experimental results for the the proposed kinetic 3D convex hull algorithm with the robust motion simulator. Using our implementation, we can run simulations with tens of thousands of moving points in 3D and test their accuracy. We can perform robust motion simulation by processing an average of about two certificate failures per step. The 3D hull algorithm seems to take (poly) logarithmic time on average to respond to a certificate failure as well as to an integrated event—an insertion or deletion that occurs during a motion simulation. We include the code for the math library and the code for the self-adjusting convex hull program in Appendix A.

2.1 Robust Motion Simulation on a Lattice

We propose an approach to robust motion simulation that combines event-based kinetic simulation and the classic idea of fixed-time sampling. The motivation behind the approach is to avoid ordering the roots of polynomials, because it requires high-precision exact arithmetic when the roots are close. To achieve this, we discretize the time axis to form a lattice $\{k \cdot \delta \mid k \in \mathbb{Z}_+\}$ defined by the *precision* parameter δ. We then perform motion simulations at the resolution of the lattice by

processing the certificates that fail within an interval of the lattice simultaneously. This approach requires that the update mechanism used for revising the computed property be able to handle multiple certificate failures at once. We use self-adjusting computation, where computations can respond to any change in their data correctly by means of a generic change propagation algorithm.

Assuming that the coordinates of the trajectories of all points can be expressed as polynomial functions of time, for robust motion simulations, we will need to perform the following operations:

- Compute the signs of a polynomial and of its derivatives at a given lattice point.
- Compute the intervals of the lattice that contain the roots of a polynomial.

In our approach, we assume that the coefficients of the polynomials are integers (up to a scaling factor) and use exact integer arithmetic to compute the signs of the polynomial and its derivatives. For finding the roots, we use a root solver described below.

The Root Solver. Our root solver relies on a procedure, which we call a *Sturm query*, that returns the number of roots of a square-free polynomial that are smaller than a given lattice point. To answer such a query, we compute the Sturm sequence (a.k.a. standard sequence) of the polynomial, which consists of the intermediary polynomials generated by Euclid's algorithm for finding the greatest common divisor (GCD) of the polynomial and its derivative. The answer to the query is the difference in the number of alternations in the signs of the sequence at $-\infty$ and at the query point. Using the Sturm query, we can find the roots of a square-free polynomial by performing a variant of a binary search. We can eliminate the square-free assumption by a known technique that factors the polynomial into square and square-free polynomials.

Motion Simulation. We maintain a priority queue of events (initially empty), and a global simulation time (initially 0). We start by running the static algorithm in the self-adjusting framework. This computes a certificate polynomial for each comparison. For each certificate, we find the

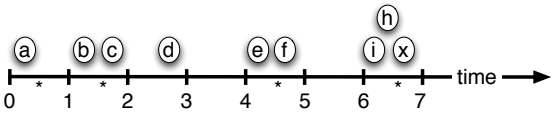


Figure 2.1: The lattice ($\delta = 1$) and the events (certificate failures).

lattice intervals at which the sign of the corresponding polynomial changes, and for each such interval, we insert an event into the priority queue. After the initialization, we simulate motion by advancing the time to the smallest lattice point t such that the lattice interval $[t - \delta, t)$ contains an event. To find the new time t we remove from the priority queue all the events contained in the earliest nonempty interval. We then change the outcome of the removed certificates and perform a change propagation at time t. Change propagation updates the output and the queue by inserting new events and removing invalidated ones. We repeat this process until there is no more certificate failure. Figure 2.1 shows a hypothetical example with $\delta = 1$. We perform change propagation at times $1, 2, 3, 5, 7$. Note that multiple events are propagated simultaneously at time 2 (events b and c), time 5 (events e and f), and time 7 (events h, i and, x).

When performing change propagation at a given time t, we may encounter a polynomial that is zero at t, representing a degeneracy. In this case, we use the derivatives of the polynomial to determine the sign immediately before t. Using this approach, we are able to avoid degeneracies throughout the simulation, as long as the certificate polynomials are not identically zero.

We note that the approach described here is quite different from the approach suggested by Abam et al. [2]. In that approach, root isolation is avoided by allowing certificate failures to be processed out of order. This can lead to incorrect transient results and requires care in the design of the kinetic structures. We do not process certificates out of order but rather as a batch.

2.2 Algorithm

In the kinetic framework based on self-adjusting computation [8], we can use any static algorithm directly. The performance of the approach, however, depends critically on the cost of the change

propagation algorithm when applied after changes are made to input or predicate values. In particular, when invoked, the change-propagation algorithm updates the current trace (sequence of operations together with their data) by removing outdated operations and re-executing parts of the algorithm that cannot be reused from the current trace. The performance of change propagation therefore depends on some form of the edit distance between the execution trace before and after the changes. This edit distance has been formalized in the definition of *trace stability*. In this section, we describe a variant of the randomized incremental convex-hull algorithm, and remark on some of its features that are crucial for stability—i.e., that minimize the number of operations that need to be updated when a certificate fails.

Given an input $S \subseteq \mathbb{R}^3$, the convex hull of S, denoted by *conv*(S), is the smallest convex polyhedron enclosing all points in S. We assign each input point $p \in S$ a random priority and construct the convex hull *conv*(S) incrementally by inserting the input points one by one into the current convex hull in priority order; higher priority points first. During this construction, we say that a face of the hull is *visible* from a point if the point and the convex hull lies on opposite sides of the plane defined by the face. Letting A_i denote the set of i highest-priority points, the next highest priority point p_{i+1} is associated with a set $\Phi(p_{i+1}) \subset$ *conv*(A_i) of faces that is visible from p_{i+1}. We use this set to determine the next convex hull.

More formally, our algorithm takes as input an ordered set of points S with random priorities; $p_1, p_2, \ldots, p_n$ being the list in decreasing priority order, and performs the following steps:

1. Create the initial convex hull *conv*(A_4), where A_4 is the set of four highest-priority points.
2. Pick a *center* point c inside *conv*(A_4).
3. For each next highest priority point p_{i+1}
   (a) Find the associated face f_j in each of the convex hulls *conv*$(A_4), \ldots,$ *conv*$(A_j), \ldots,$ *conv*(A_i) satisfying the condition that the ray $\overrightarrow{cp_{i+1}}$ intersects f_j.

(b) If the associated face f_i of *conv*(A_i) is not visible from p_{i+1} discard p_{i+1}; consequently, set *conv*$(A_{i+1}) =$ *conv*(A_i).

(c) Otherwise, construct *conv*(A_{i+1}) by following the below rip and tent procedures

`Rip` Starting from the associated face f_i of *conv*(A_i), identify the set of visible faces $\Phi(p_{i+1})$ and the set of *horizon edges* which are defined to be the edges incident to both visible and invisible faces.

`Tent` Create a new set $\hat{\Phi}$ of faces each of which consists of the point p_{i+1} and a horizon edge; set *conv*$(A_{i+1}) = \hat{\Phi} \cup$ *conv*$(A_i) \setminus \Phi(p_{i+1})$.

In our implementation (Appendix A), we maintain a "killer" relationship between a point p and a face f to indicate that p takes out f during the rip procedure. This allows us to perform step 3(a) efficiently, by considering the appropriate convex hulls instead of traversing all of them, and by stopping earlier whenever the point p_{i+1} is guaranteed to be inside the current convex hull *conv*(A_i).

Even though the algorithm presented above is fairly standard, certain key elements of this implementation appear to be crucial for stability—without them, the algorithm would be unstable. For stability, we want the edit distance between the traces to be small. Towards this goal, the algorithm should always insert points in the same order—even when new points are added or old points deleted. We ensure this by assigning a random priority to every input point. The use of random priorities makes it easy to handle new points, and obviates the need to explicitly remember the insertion order.

Additionally, we want the insertion of a point p to visit faces of the convex hull in the same order every time. Indeed, the presented algorithm guarantees this by using the following heuristic. The point-to-face assignment with respect to a center point c ensures that the insertion of p always starts excavating at the same face. Furthermore, if p visits a face f in one execution, it will visit the same face f in any other execution provided that f is created as part of some convex hull. Note that the choice of the center point is arbitrary, with the only requirement that the center point has

to lie in the convex hull. Our implementation takes c to be the centroid of the tetrahedron formed by A_4.

2.3 Implementation

Our implementation consists of three main components: 1) the self-adjusting-computation library, 2) the incremental 3D convex-hull algorithm, and 3) the motion simulator. Previous work [5] provided an implementation of the self-adjusting computation library. The library requires that the user adds some notations to their static algorithms to mark what values can change and what needs to be memoized. These notations are used by the system to track the dependences and know when to reuse subcomputations.

In our experiments, we use both the original static 3D convex-hull algorithm and the self-adjusting version with the annotations added. The static version uses exact arithmetic predicates to determine the outcomes of comparisons precisely (we use the static version for checking the robustness of the simulation). The self-adjusting version uses the root solver to find the roots of the polynomial certificates, and inserts them into the event queue of the motion simulator. We implement a motion simulator as described in Section 2.1. Given a precision parameter δ and a bound M_t on the simulation time, the simulator uses an event scheduler to perform a motion simulation on the lattice with precision δ until M_t is reached. We model the points with an initial location traveling at constant speed in a fixed direction. For each coordinate, we use B_ℓ and B_v bits to represent the initial location and the velocity respectively; B_ℓ and B_v can be assigned to arbitrary positive natural numbers.

2.4 Experiments

We describe an experimental evaluation of our kinetic 3D convex-hull algorithm. The evaluation investigates the effectiveness of our approach according to a number of metrics which are respon-

siveness, efficiency, locality, and compactness. Following that, we report timing results for the integrated dynamic and kinetic experiments.

Experimental Setup. All of the experiments were performed on a 2.66Ghz dual-core Xeon machine, with 8 GB of memory, running Ubuntu Linux 7.10. We compiled the applications with the MLton compiler with the option "`-runtime ram-slop 0.75`," directing the run-time system to allocate at most 75% of system memory. Our timings measure the wall-clock time (in seconds).

Input Generation. In our experiments, we pick the initial positions of the points on each axis to fit into 20 bits, i.e., $B_\ell = 20$, and the velocity along each axis to fit into 8 bits, i.e, $B_v = 8$. We pick both the initial locations and the velocities uniformly randomly from the cube $[-1.0, 1.0]^3$. We perform motion simulations on lattice defined by $\delta = 2^{-10}$, with a maximum time of $M_t = 2^{27}$. With this setting, we process an average of about two certificates simultaneously.

Checking for robustness. We check that our algorithm simulates motion robustly by comparing it to our exact static algorithm after each event in the kinetic simulation. When the inputs are large (more than 1000 points), we check the output at randomly selected events (with varying probabilities between 1 and 20%) to save time.

Baseline Comparison. To assess the efficiency of the static version of our algorithm, we compare it to CGAL 3.3's implementation of the incremental convex-hull algorithm. Figure 2.2 shows the timings for our static algorithm and for the CGAL implementation. Inputs to the algorithms are generated by sampling from the same distribution; the reported numbers averaged over three runs. Our implementation is about 30% slower than CGAL's.

We also want to compare our kinetic implementation with an existing kinetic implementation capable of computing 3D convex hulls. Since there is no direct implementation for kinetic 3D convex hulls, we compare our implementation with CGAL's kinetic 3D Delaunay-triangulation implementation, which computes the convex hull as part of the triangulation. Figure 2.3 shows

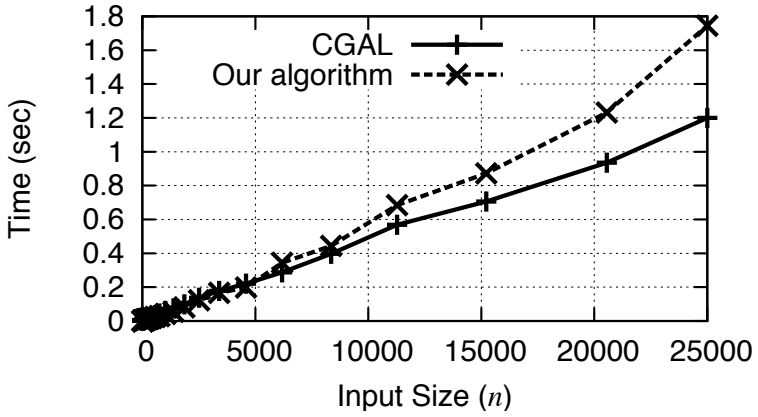


Figure 2.2: Static algorithms compared.

	CGAL		Our Algorithm	
Input Size	# Events	Total Time (s)	# Events	Total Time (s)
22	357	13.42	71	2.66
49	1501	152.41	151	11.80
73	2374	391.31	218	23.42
109	4662	1270.24	316	40.37
163	7842	3552.48	380	70.74
244	15309	12170.08	513	125.16

Figure 2.3: Simulations compared.

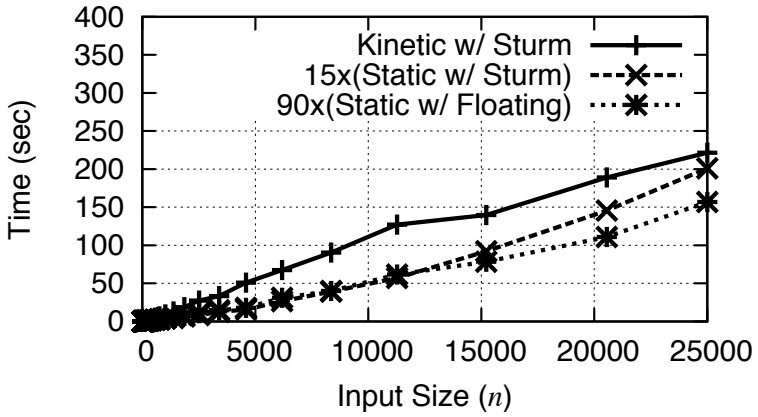


Figure 2.4: Kinetic and static runs.

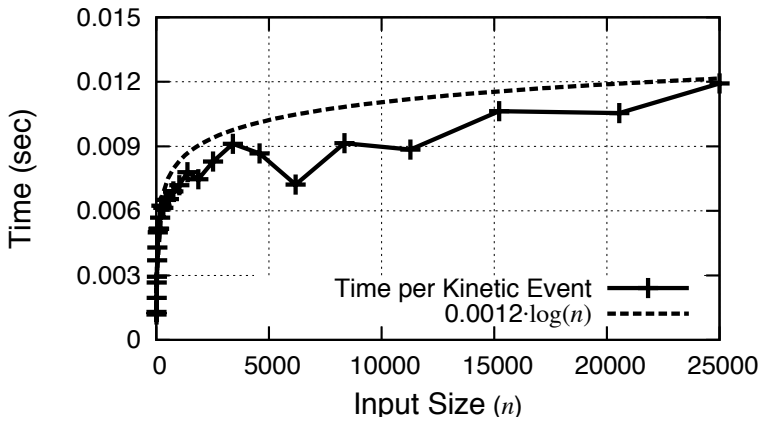


Figure 2.5: Time per kinetic event.

the timings for our algorithm and for CGAL's implementation of kinetic 3D Delaunay (using the `Exact_simulation_traits` traits). These experiments are run until the event queue is empty. As expected, the experiments show that kinetic Delaunay processes many more events than necessary for computing convex hulls.

Kinetic motion simulation. To perform a motion simulation, we first run our kinetic algorithm on the given input at time $t = 0$, which we refer to as the *initial run*. This computes the certificates and inserts them into the priority queue of the motion scheduler. Figure 2.4 illustrates the running time for the initial run of the kinetic algorithm compared to that of our static algorithm which does not create certificates. Timings show a factor of about 15 gap between the kinetic algorithm (using Sturm sequences) and the static algorithm that uses exact arithmetic. The static algorithm runs by a factor of 6 slower when it uses exact arithmetic compared to using floating-point arithmetic. These experiments indicate that the overheads of initializing the kinetic simulations is moderately high: more than an order of magnitude over the static algorithm with exact arithmetic and almost two orders of magnitude over the the static algorithm with floating-point arithmetic. This is due to both the cost of creating certificates and to the overhead of maintaining the dependence structures used by the change propagation algorithm.

After completing the initial run, we are ready to perform the motion simulation. One measure

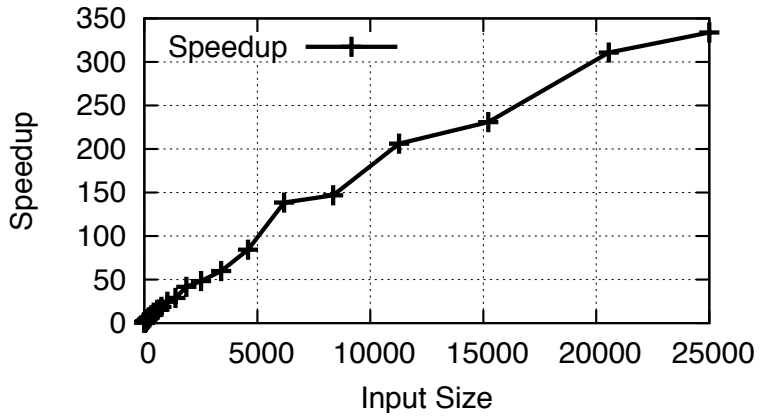


Figure 2.6: Speedup for a kinetic event.

of the effectiveness of the motion simulation is the average time for a kinetic event, calculated as the total time for the simulation divided by the number of events. Figure 2.5 shows the average times for a kinetic event when we use our δ-precision root solver. These averages are for the first $5 \cdot n$ events on an input size of n. The average time per kinetic event appears asymptotically bounded by the logarithm of the input size. A kinetic structure is said to be *responsive* if the cost per kinetic event is small, usually in the worst case. Although our experiments do not indicate responsiveness in the worst case, they do indicate responsiveness in the average case.

One concern with motion simulation with kinetic data structures is that the overhead of computing the roots can exceed the speedup that we may hope to obtain by performing efficient updates. This does *not* appear to be the case in our system. Figure 2.6 shows the speedup for a kinetic event, computed as the time for change propagation divided by the time for a from-scratch execution of the static algorithm using our solver.

In many cases, we also want to be able to insert and remove points or change the motion parameters during the motion simulation. This is naturally supported in our system, because self-adjusting computations can respond to any combination of changes to their data. We perform the following experiment to study the effectiveness of our approach at supporting these *integrated changes*. During the motion simulation, at every event, the motion function of an input point is updated from $r(t)$ to $\frac{3}{4}r(t)$. We update these points in the order they appear in the input, ensuring

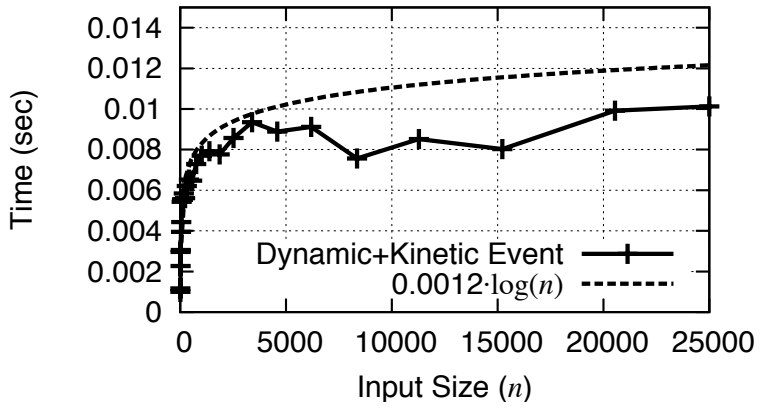


Figure 2.7: Time per integrated event.

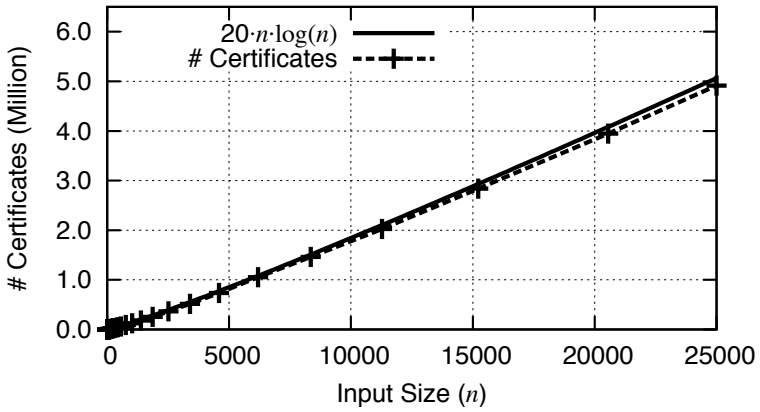


Figure 2.8: Number of certificates.

that every point is updated at least once. From this experiment, we report the average time per integrated event, calculated by dividing the total time to the number of events. Figure 2.7 shows the average time per integrated event for different input sizes. The time per integrated event appears asymptotically bounded by the logarithm of the input size and are similar to those for kinetic events only. A kinetic structure is said to have good *locality* if the number of certificates a point is involved in is small. We note that the time for a dynamic change is directly affected by the number of certificates it is involved in. Again, although our experiments do not indicate good locality in the worst case, they do indicate good locality averaged across points.

Another measure of the effectiveness of a kinetic motion simulation is *compactness*, which is a

measure of the total number of certificates that are live at any time. Since our implementation uses change-propagation to update the computation when a certificate fails, it guarantees that the total number of certificates is equal to the number of certificates created by a from-scratch execution at the current position of the points. Figure 2.8 shows the total number of certificates created by a from-scratch run of the algorithm with the initial positions. The number of certificates appears to be bounded by $O(n \log n)$.

CHAPTER 3

DYNAMIC WELL-SPACED POINT SETS

In this chapter, we present a dynamic algorithm for the well-spaced point set problem. Our algorithm always returns size-optimal outputs and requires worst-case $O(\log \Delta)$ time for an input modification (an insertion or a deletion), Δ being the geometric spread (the ratio of the diameter of the input set to the distance between the closest pair of points in the input). Our update runtime is optimal in the worst case, and our algorithm consumes linear space in the size of the output.

To solve the dynamic problem, we first present an efficient construction algorithm for generating size-optimal, well-spaced supersets (Section 3.2). In addition to the output, the construction algorithm builds a *computation graph* that represents the operations performed during its execution and the dependencies between them. A key property of this algorithm is that it is *stable*: when run with similar inputs, e.g., inputs that differ by one point, it produces similar computation graphs and outputs. We make this property precise by describing a *distance* measure between the computation graphs of two executions and bounding this distance by $O(\log \Delta)$ when inputs differ by a single point (Section 3.3.3). Taking advantage of this bound, we design a change-propagation algorithm that performs dynamic updates in $O(\log \Delta)$ time by identifying the operations that are affected by the modification to the input and deleting or re-executing them as necessary (Section 3.4.1). For the lower bound, we show that there exist inputs and modifications that require $\Omega(\log \Delta)$ Steiner points to be inserted into or deleted from the output (Section 3.4.2).

The efficiency of our dynamic update algorithm directly depends on stability. In order to achieve stability, we use several techniques in the design of our construction algorithm. Generalizing the recently suggested choices of Steiner points [42, 47], we propose an approach for picking Steiner points by making local decisions only, using clipped Voronoi cells. Picking Steiner points locally makes it possible to structure the computation into $\Theta(\log \Delta)$ ranks, inductively ensuring that at the end of each rank the points up to that rank are well spaced [65]. Processing points in rank order alone does not guarantee stability; we further partition points at a given rank into a

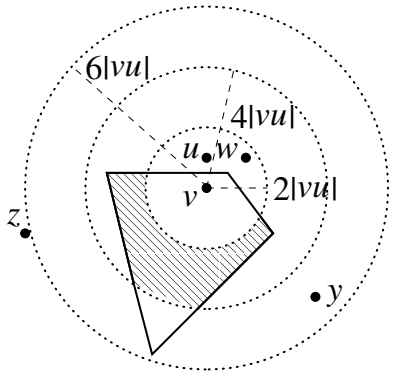


Figure 3.1: Let $\mathcal{M} = \{v, u, w, y, z\}$. The nearest-neighbor distance of v, $\mathrm{NN}_{\mathcal{M}}(v)$, is $|vu|$. The polygon with solid boundary lines depicts the Voronoi cell of v, $\mathrm{Vor}_{\mathcal{M}}(v)$. The shaded region displays the $(2, 4)$ picking region of v, $\mathrm{Vor}^{(2,4)}_{\mathcal{M}}(v)$. Vertices y and z are 4-clipped but not 2-clipped Voronoi neighbors of v.

constant number of color classes such that the points in each color class depend only on the points in the previous color classes. These techniques enable us to process each point only once and help isolate and limit the effects of a modification. Furthermore, our dynamic update algorithm returns an output and a computation graph that are isomorphic to those that would be obtained by re-executing the static algorithm with the modified input (Lemma 3.4.2). Consequently, the output remains both well spaced and size optimal with respect to the modified input (Theorem 3.4.3).

3.1 Steiner Vertices and Spatial Data Structure

We present some definitions used throughout the chapter, describe the technique we use for selecting Steiner vertices, and present a brief overview of the dynamic balanced quadtrees.

3.1.1 Clipped Voronoi Cells

Regardless of runtime considerations, the fundamental question in mesh refinement is about where to insert the Steiner vertices [63]. Traditional solutions place Steiner vertices as far from any other vertex as possible, namely, at the circumcenters of Delaunay simplices (equivalently, at the nodes of the Voronoi diagram). In two dimensions, Har-Peled and Üngör instead place Steiner vertices close to a vertex, but not too close: at the *off-centers* [42]. This local scheme allows them to build a data structure that can locate the off-centers in constant time. More recent work by Jampani and Üngör extends this idea to three dimensions using *core disks* [47]. In this chapter,

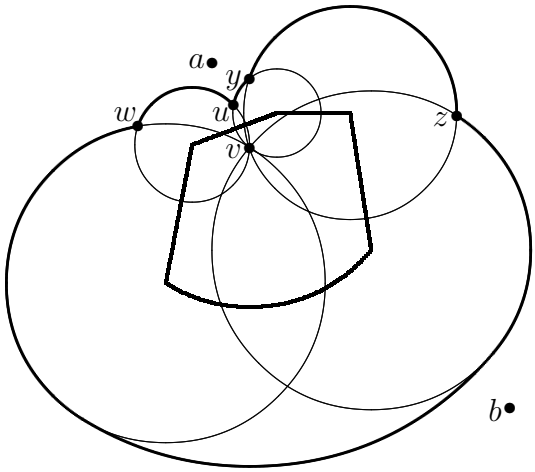


Figure 3.2: $\mathcal{M} = \{a, b, v, u, w, y, z\}$. $\mathrm{NN}_{\mathcal{M}}(v) = |vu|$. The thick boundary depicts the β-clipped Voronoi cell of v, $\mathrm{Vor}^{\beta}_{\mathcal{M}}(v)$, for $\beta = 4$. The thin boundary depicts the certificate region of $\mathrm{Vor}^{\beta}_{\mathcal{M}}(v)$. Vertices a and b are not β-clipped Voronoi neighbors of v since there is no empty certificate ball for a and the empty certificate balls for b have radii exceeding $\beta\,\mathrm{NN}_{\mathcal{M}}(v)$.

we generalize their results, by describing a local picking region that gives us a variety of Steiner vertex choices including the aforementioned ones. The definition of our picking region extends to arbitrary dimensions in a straightforward way.

In order to form the basis for our picking region, we begin by identifying a local neighborhood of a vertex: the *β-clipped Voronoi cell* of v, written $\mathrm{Vor}^{\beta}_{\mathcal{M}}(v)$, is the intersection of $\mathrm{Vor}_{\mathcal{M}}(v)$ with the ball of radius $\beta\,\mathrm{NN}_{\mathcal{M}}(v)$ centered at v. For any $\beta > \rho$, we define the (ρ, β) *picking region* of v, written $\mathrm{Vor}^{(\rho,\beta)}_{\mathcal{M}}(v)$, as $\mathrm{Vor}^{\beta}_{\mathcal{M}}(v) \setminus \mathrm{Vor}^{\rho}_{\mathcal{M}}(v)$, the region of the Voronoi cell bounded by concentric balls of radius $\rho\,\mathrm{NN}_{\mathcal{M}}(v)$ and $\beta\,\mathrm{NN}_{\mathcal{M}}(v)$. Note that v is ρ-well-spaced if and only if the (ρ, β) picking region is empty, i.e., $\mathrm{Vor}^{\rho}_{\mathcal{M}}(v) = \mathrm{Vor}^{\beta}_{\mathcal{M}}(v) = \mathrm{Vor}_{\mathcal{M}}(v)$. Figure 3.1 illustrates these definitions.

In order to correctly compute the β-clipped Voronoi cell $\mathrm{Vor}^{\beta}_{\mathcal{M}}(v)$ of a vertex v, an algorithm must certify that v is the closest vertex to any point inside $\mathrm{Vor}^{\beta}_{\mathcal{M}}(v)$ and that any point on the boundary of $\mathrm{Vor}^{\beta}_{\mathcal{M}}(v)$ is either equidistant to v and another vertex or at the clipping distance $\beta\,\mathrm{NN}_{\mathcal{M}}(v)$ from v. Defining a vertex u to be a *β-clipped Voronoi neighbor* of v if the β-clipped Voronoi cell of v contains a point equidistant from v and u, we reduce the problem of computing the β-clipped Voronoi cell of v to computing the set of β-clipped Voronoi neighbors of v. Simply,

given the set of β-clipped Voronoi neighbors of v, one can construct its β-clipped Voronoi cell efficiently. In order to ensure that the computation of the set of β-clipped Voronoi neighbors is correct, an algorithm needs to certify that for any $x \in \mathrm{Vor}^{\beta}_{\mathcal{M}}(v)$ the ball centered at x with radius $|vx|$ is empty of vertices. We call these balls *certificate balls* and define the *certificate region* of $\mathrm{Vor}^{\beta}_{\mathcal{M}}(v)$ as their union. Figure 3.2 illustrates these definitions.

3.1.2 Dynamic Balanced Quadtrees

To permit the efficient calculation of such things as nearest neighbors and clipped Voronoi cells, we use a point location data structure based on the balanced quadtrees of Bern, Eppstein, and Gilbert [20]. We extend their balanced quadtrees to arbitrary (d) dimensions and dynamize them by describing an update algorithm for inserting/deleting an input vertex in $O(\log \Delta)$ time. We use the term *quadtree* to mean 2^d-tree and the term *quadtree node* to mean d-hypercube. Our well-spaced point set algorithms treat the quadtree data structure almost as a black box; they use only the leaves of the quadtree, which we refer to as *(quadtree) squares*. Our quadtrees are minimal among those that satisfy the *crowding* and the *grading* properties defined as follows:

- Crowding: every quadtree square (leaf node) contains at most one vertex, and if it does, none of its neighbors contains a vertex.

- Grading: all neighbors of any internal node must exist in the quadtree.

Here, we define the *neighbors* of a quadtree node to be the same size nodes in each of the $3^d - 1$ cardinal and diagonal directions. To support fast traversal and access, a quadtree node keeps pointers to its parent, children, and neighbors. Additionally, every square contains a pointer to an input vertex it may contain, and a list of Steiner vertices. We define two squares to be *adjacent* if their intersection contains at least one point. Given any two adjacent squares, the grading condition ensures that either they are neighbors or one of them is a neighbor of the parent of the other.

For computing the clipped Voronoi cells, our quadtree data structure supports the function `QTClippedVoronoi`, details of which we explain in Section 3.2.4. Briefly, given a vertex v and a parameter $\beta > 1$, this function returns the set of β-clipped Voronoi neighbors of v. Taking advantage of the careful scheduling of the operations in the construction algorithm, we prove that `QTClippedVoronoi` runs in $O(1)$ time. For efficiently locating and inserting Steiner vertices, our quadtree data structure supports the `QTInsertSteiner` function. Given a Steiner vertex w that is inside the (ρ, β) picking region of a vertex v, this function finds the quadtree square that contains w by a traversal starting from the square of v towards w. Since w is inside the β-clipped Voronoi cell of v, this function runs in $O(1)$ time.

For constructing the quadtree, we provide the function `QTBuild`. For inserting and/or deleting an input vertex $\hat{v}$ into/from a quadtree, we provide the functions `QTInsertInput` and `QTDeleteInput`. Given the original quadtree $\mathcal{Q}$ and the vertex $\hat{v}$, these two functions return the updated quadtree $\mathcal{Q}'$ and the set of obsolete squares of $\mathcal{Q}$. For insertions, the obsolete squares are those that become internal nodes in $\mathcal{Q}'$; for deletions, they are the deleted leaf nodes of $\mathcal{Q}$.

In the rest of this section, we explain the construction and the dynamic modification functions and prove some results which will be useful in the analysis of the dynamic well-spaced point set algorithms. `QTBuild` function iteratively inserts each input vertex into an empty quadtree using `QTInsertInput`. Given an input vertex $\hat{v}$ to be inserted, `QTInsertInput` first determines the square that contains $\hat{v}$ by performing a top-down traversal of the quadtree. If this square already contains an input vertex, it then splits this square and descends into the child containing $\hat{v}$, repeating as necessary. Finally, it inserts $\hat{v}$ into the resulting (currently empty) square, and in order to restore the quadtree, it imposes the crowding and the grading conditions. For crowding, the algorithm simply enumerates every square which could possibly be crowded by $\hat{v}$, check if it is crowded, and if so, split it. The following lemma follows immediately from the definition of the crowding property; it implies that there are only a constant number of squares at each level, and hence $O(\log \Delta)$ overall, which need to be checked:

Lemma 3.1.1 *During a call to* `QTInsertInput` *to insert* $\hat{v}$*, if a quadtree node is split due to crowding, then either that node or one of its neighbors contains* $\hat{v}$*.*

The algorithm imposes the grading condition similarly: by enumerating all squares which could possibly be split due to grading in response to the insertion of $\hat{v}$, checking the grading condition, and performing splits as necessary. The following lemma (very similar to Lemma 1 of [57]), when combined with Lemma 3.1.1, implies that there are only $O(\log \Delta)$ squares which need to be checked:

Lemma 3.1.2 *During a call to* `QTInsertInput`*, if a quadtree node is split due to grading, then a descendent of one of its neighbors must have been split due to crowding.*

Proof: For this result, suppose that we enforce the grading condition after each crowding split (the resulting quadtree would be exactly the same). Assuming that s is a quadtree node at depth ℓ split due to crowding, let s' be a quadtree node at depth $\lambda < \ell$ which must be split due to grading in response to s being split. We claim that the Chebyshev distance (L_∞-metric) from s to s' (minimum among the pair of points in s and s') is bounded by $2^{-\lambda} - 2^{-(\ell-1)}$. Then, assuming s'' to be the ancestor of s at depth λ, the distance from s'' to s' is also bounded by $2^{-\lambda} - 2^{-(\ell-1)}$. Therefore s' and s'' must be neighbors and this completes our proof.

We prove our claim by induction on $\lambda < \ell$. For the base case, $\lambda = \ell - 1$, the only nodes which may be split are the nodes that are adjacent to s, and they are at 0 distance. Assuming our claim for some $\lambda < \ell$, let σ be a quadtree node at depth $\lambda - 1$ which must be split. By the definition of the grading property, it follows that a node σ' at depth λ and adjacent to σ must have been split and become an internal node. By the inductive hypothesis, σ' is within $2^{-\lambda} - 2^{-(\ell-1)}$ distance of s. Since, the maximum distance between any pair of points in σ' is bounded by $2^{-\lambda}$, using the triangle inequality, we show that σ is within $2^{-(\lambda-1)} - 2^{-(\ell-1)}$ distance of s. This concludes our claim. ■

We have shown that the entire `QTInsertInput` function may be implemented in $O(\log \Delta)$ time. Next, we prove a result that characterizes the squares which became internal nodes during

the call to `QTInsertInput`:

Lemma 3.1.3 *For any square $s \in \mathcal{Q}$ that is returned by `QTInsertInput`$(\mathcal{Q}, \hat{v})$, we have $|s\hat{v}| \in O(|s|)$.*

Proof: Lemmas 3.1.1 and 3.1.2 prove that every split square s is at most a neighbor's neighbor of the quadtree node containing $\hat{v}$ at the same depth as s. Hence, the distance between $\hat{v}$ and s satisfies $|s\hat{v}| \leq 2\sqrt{d}|s|$. ■

The `QTDeleteInput` function essentially performs the same steps as the `QTInsertInput` function, in reverse. First, descending through the quadtree, it locates the quadtree square containing $\hat{v}$ and deletes it from the square. Next, motivated by Lemmas 3.1.1 and 3.1.2, it checks all ancestors of this square, their neighbors and neighbors' neighbors, all in a bottom-up fashion, merging them if they are no longer crowded, and do not need to be split due to grading. An analogue of Lemma 3.1.3 holds for `QTDeleteInput`, and the proof follows similarly.

Lemma 3.1.4 *For any square $s \in \mathcal{Q}$ that is returned by `QTDeleteInput`$(\mathcal{Q}, \hat{v})$, we have $|s\hat{v}| \in O(|s|)$.*

Finally, we prove a lemma demonstrating that the quadtree data structure we describe in this section can be used to approximate the local feature size of the points up to a constant factor. By definition of lfs, this implies an approximation on the nearest-neighbor distances of the input vertices.

Lemma 3.1.5 *Given an input N, let $\mathcal{Q}$ be the minimum quadtree that represents N satisfying the crowding and the grading conditions, and let $s \in \mathcal{Q}$ be a square and p be a point in s. We have $\mathrm{lfs}(p) \in \Theta(|s|)$; also, if $p \in \mathsf{N}$, then $\mathrm{lfs}(p) > |s|$.*

Proof: If $p \in \mathsf{N}$ or there exists an input vertex in s, then by the crowding condition, the neighbors of s do not contain a vertex. This implies that there are no other input vertices in a

ball of radius $|s|$ around p, i.e., $\mathrm{lfs}(p) > |s|$. Otherwise, let $v \in s' \neq s$ be the vertex nearest p. If s' is not adjacent to s, since all the adjacent squares have size at least $|s|/2$ by the grading condition, we have $\mathrm{lfs}(p) \geq \mathrm{NN}_{\mathsf{N}}(p) > |s|/2$. If s' is adjacent to s, then by the Lipschitz condition $\mathrm{lfs}(p) + |pv| \geq \mathrm{lfs}(v)$, and $\mathrm{lfs}(v) > |s'|$ from the analysis above. Therefore, if $|pv| > |s'|/2$ then $\mathrm{lfs}(p) \geq \mathrm{NN}_{\mathsf{N}}(p) = |pv| > |s|/4$; otherwise, the same lower bound still holds, $\mathrm{lfs}(p) \geq \mathrm{lfs}(v) - |pv| > |s'|/2 \geq |s|/4$.

For the upper bound, we use the minimality of the quadtree, that σ, the parent of s, must have been split because of one of the two conditions. If σ is split because of crowding then there exist two vertices within $2\sqrt{d}|\sigma|$ distance of p. If σ is split due to grading, this split must have been caused by a crowding split due to vertices $v, u \neq p$. Lemmas 3.1.1 and 3.1.2 prove that σ is at most a neighbor's neighbor of the quadtree node containing v, also u. This implies that there exist two vertices within $3\sqrt{d}|\sigma|$ distance of p. In either case, $\mathrm{lfs}(p) < O(|s|)$. ∎

3.2 Construction Algorithm

For any given $\beta > \rho > 1$, consider *filling* a vertex, which we define as inserting Steiner vertices in the (ρ, β) picking region of that vertex until it becomes ρ-well-spaced. Given an input set of vertices, we can construct a ρ-well-spaced superset of it by repeatedly filling vertices until the resulting superset becomes ρ-well-spaced. Although correct, this basic algorithm is not efficient because vertices may need to be filled multiple times. More specifically, a Steiner vertex inserted while filling a vertex may become the nearest neighbor of an already filled vertex making it lose its ρ-well-spacedness and requiring it to be filled more than once. This algorithm is not stable either; that is, it can generate very different outputs when run on similar inputs, because the presence or absence of a single vertex can affect the choice of many subsequent Steiner vertices. To address these problems and achieve efficiency and stability, we refine the basic algorithm by specifying an order in which vertices are filled.

3.2.1 Ranks

In order to ensure efficiency and avoid filling vertices multiple times, we fill vertices in increasing order according to their nearest-neighbor distances. Before we explain the details of this schedule, we discuss the intuition behind it by pointing out several facts about our Steiner vertex selection scheme. Given a vertex set $\mathcal{M}$, consider filling a vertex $v \in \mathcal{M}$ that is not ρ-well-spaced. Let w be a Steiner vertex inserted while filling v. The first fact is that w is in the (ρ, β) picking region of v.

Fact 1 *The Steiner vertex w is in $\mathrm{Vor}^{(\rho,\beta)}_{\mathcal{M}}(v)$. That is, $\forall u \in \mathcal{M}, |wv| \leq |wu|$ and $\rho\,\mathrm{NN}_{\mathcal{M}}(v) \leq |wv| < \beta\,\mathrm{NN}_{\mathcal{M}}(v)$.*

Since v is the nearest neighbor of w, i.e., $|wv| = \mathrm{NN}_{\mathcal{M}}(w)$, this fact implies that $\mathrm{NN}_{\mathcal{M}}(w) \geq \rho\,\mathrm{NN}_{\mathcal{M}}(v)$. Now, let us suppose that the vertices whose nearest neighbors are at distance less than α are all ρ-well-spaced. Since v is not ρ-well-spaced in $\mathcal{M}$, we have $\mathrm{NN}_{\mathcal{M}}(v) \geq \alpha$. Then, we infer the following fact.

Fact 2 *For any given $\alpha > 0$ if every vertex $u \in \mathcal{M}$ with $\mathrm{NN}_{\mathcal{M}}(u) < \alpha$ is ρ-well-spaced then $\mathrm{NN}_{\mathcal{M}}(w) \geq \rho\alpha$.*

This fact implies that the Steiner vertices that will be inserted into $\mathcal{M}$ are all at least $\rho\alpha$ away from any vertex in $\mathcal{M}$. Therefore, inserting a Steiner vertex does not change the nearest neighbors and hence the well-spacedness of the vertices whose nearest neighbors are at distance less than $\rho\alpha$. Motivated by this property, we define the *rank* of a vertex $v \in \mathcal{M}$ as the logarithm in base ρ of its nearest neighbor distance, i.e., $\lfloor \log_\rho \mathrm{NN}_{\mathcal{M}}(v) \rfloor$ and fill the vertices in a single pass using the rank order. With this partial ordering, for example, the vertices with nearest neighbor distances in $[\rho^r, \rho^{r+1})$ would be at rank r. Note that for any $\rho > 1$, this partial order has only a logarithmic number of ranks, $O(\log \Delta)$ in particular. As we prove in Lemma 3.3.9, filling vertices in rank order guarantees that filling each vertex takes $O(1)$ time, yielding an efficient construction algorithm. However, these refinements are not enough to ensure stability.

3.2.2 Colors

In order to achieve stability, we take advantage of the locality of our Steiner vertex selection scheme and geometrically partition the vertices at each rank into a constant number of *color* classes and fill them in color order so that vertices of the same color class can be filled independently. More specifically, we say that two vertices at the same rank are *independent* if at least one of them is not ρ-well-spaced and the certificate region of the β-clipped Voronoi cell of any of them does not intersect the (ρ, β) picking region of the other. Intuitively, two vertices are independent if the Steiner vertices inserted while filling one of them do not alter the picking region of the other. We identify independent vertices by using a *coloring scheme* that partitions the space based on a *coloring parameter* κ, and a real valued function $\ell(r)$ defined on ranks. At each rank r, we partition the space into d-dimensional hypercubes or r*-tiles* with side length $\ell(r)$. We color r-tiles such that they are colored periodically in each dimension with period κ, using κ^d colors in total. A vertex v has color $c \in \{0, 1, \ldots, \kappa^d - 1\}$ if it lies in a c colored r-tile. Figure 3.3 illustrates a coloring scheme. By choosing $\ell(r)$ small enough and κ large enough, we prove that two vertices at the same rank are independent if they have the same color (Lemma 3.3.12). Therefore, at a given rank, filling vertices in color order restricts the dependencies between vertices: filling a vertex may affect only the unprocessed vertices of different colors. During a dynamic update, this makes it possible to re-fill a vertex without affecting other independent vertices at the same rank and color.

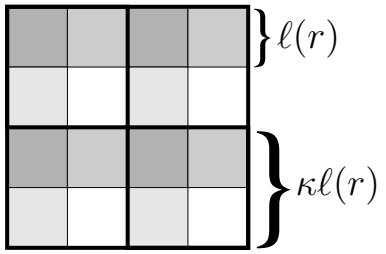


Figure 3.3: Illustration of a coloring scheme in 2D. The coloring parameter κ is 2 and there are 4 colors in total.

3.2.3 Algorithm

The efficiency and stability of our algorithm critically relies on filling vertices in rank and color order. In order to fill a vertex v that is currently at rank r, the algorithm schedules a *fill* operation

acting on v at rank r. However, since the rank of a vertex depends on its nearest neighbor and since that can change as Steiner vertices are inserted, we need to update the ranks of the vertices dynamically. In order to ensure that the ranks of the vertices are up-to-date, in our algorithm, we use another type of operation called *dispatch*. For each vertex v, our algorithm creates a single dispatch operation acting on v. This operation computes the rank of v, updates the ranks of β-clipped Voronoi neighbors of v, and schedules new fill operations as necessary. In order to ensure timely execution of dispatch operations, the algorithm assigns ranks to dispatch operations as well and executes both types of operations in rank order with dispatches having precedence over fills at the same rank. For an input vertex v, the algorithm assigns the logarithm in base ρ of the side length of the quadtree square that contains v as the rank of the dispatch operation acting on v. As we prove in Lemma 3.1.5 this rank is $O(1)$ ranks below the actual rank of v. For a Steiner vertex w, at the time of its insertion, since we know that the nearest neighbor of w is the vertex being filled, the fill operation that inserts w easily computes the current rank of w and schedules a dispatch operation acting on w at its current rank.

Using dispatch operations, the algorithm guarantees that for each vertex there exists a fill operation acting on it at its most up-to-date rank (Lemma 3.3.1). When executed, a fill operation makes the vertex it acts on ρ-well-spaced; subsequent fill operations terminate immediately without inserting Steiner vertices. Instead of creating a single fill operation per vertex and updating its rank as the ranks of the vertices change, we prefer the approach of recording and executing multiple fill operations acting on a single vertex because it simplifies the analysis by making the dependencies between the operations explicit.

Figure 3.4 shows the pseudo-code of our algorithm `StableWS`. The algorithm starts by constructing a quadtree $\mathcal{Q}$ and stores it for use in dynamic updates. It then constructs a ρ-well-spaced output by performing dispatch and fill operations that it enqueues in Ω. When enqueueing a dispatch or a fill operation acting on a vertex w, the algorithm computes the rank r_w of w by using its current nearest neighbor distance. (For dispatch operations acting on input vertices, it uses an

```
Dimension: d, Parameters: ρ, β, κ, ℓ(r)

StableWS (N) =
  Q ← QTBuild(N)
  for each v ∈ N
    apxnnv ← |square of v|
    Enqueue (v, D, apxnnv, v_(0), Ω)
  for r = min rank in Ω to ⌊log_ρ √d⌋
    for each v_(r,D) ∈ Ω
      Dispatch(v_(r,D), Ω)
    for c = 0 to κ^d − 1
      for each v_(r,F,c) ∈ Ω
        Fill(v_(r,F,c), Ω)
  return Q

Dispatch (v_(t), Ω) =
  CV ← QTClippedVoronoi (v_(t), β)
  nnv ← min {|vu| : u ∈ CV}
  Enqueue (v, F, nnv, v_(t), Ω)
  for each w ∈ CV
    Enqueue (w, F, |wv|, v_(t), Ω)

Fill (v_(t), Ω) =
  CV ← QTClippedVoronoi (v_(t), β)
  while v is not ρ-well-spaced
    Pick w ∈ Vor^(ρ,β)(v)
    QTInsertSteiner (v_(t), w)
    CGInsertEdge (v_(t) ⟶ square of w)
    Enqueue (w, D, |wv|, v_(t), Ω)
    CV ← CV ∪ {w}

Enqueue (w, flag, nnw, v_(t), Ω) =
  r_w ← ⌊log_ρ nnw⌋, c_w ← Color(w, r_w, κ)
  if flag = D then t_w ← (r_w, D)
  else t_w ← (r_w, F, c_w)
  if t_w > t then
    if ∄ edge · ⟶ w_(t_w) then
      Ω ← Ω ∪ {w_(t_w)}
    CGInsertEdge (v_(t) ⟶ w_(t_w))

Color (v, r, κ) =
  for i = 1 to d do c_i ← ⌊v_i/ℓ(r)⌋ mod κ
  return c = c_1 c_2 ... c_d in base κ
```

Figure 3.4: The pseudo-code of the stable construction algorithm.

approximation.) Then, computing the color c_w of w at this rank, it determines the *time* of this operation, which is comprised of the rank r_w, a flag indicating a dispatch (D) or a fill (F), and the color c_w only if the operation is a fill operation. In the pseudo-code, we represent this operation with the vertex w itself and its time t_w in the subscript ($w_{(t_w)}$). The algorithm executes the operations in Ω in time order, first by rank then by operation type and then by color order (for fills): it performs the dispatch operations before the fill operations, ordering fill operations at the same rank by color. For brevity, we define time $t = 0$ to be the beginning of time, when the input vertices are enqueued for dispatch operations but before any of them are performed, and define time $t = \infty$ to be the end of the algorithm. We write M_t to refer to the output at time t, e.g., M_0 is the input, N, and M_∞ is the output, M. For readability, we use t instead of M_t in the subscript, e.g., NN_t instead of $\mathrm{NN}_{\mathsf{M}_t}$.

To support efficient dynamic updates, the construction algorithm builds a computation graph of all executed operations and various dependencies between them. The *computation graph* $G = (V, E)$ consists of nodes, $V = \Sigma \cup \Omega$, comprised of the set of quadtree squares (Σ) and the

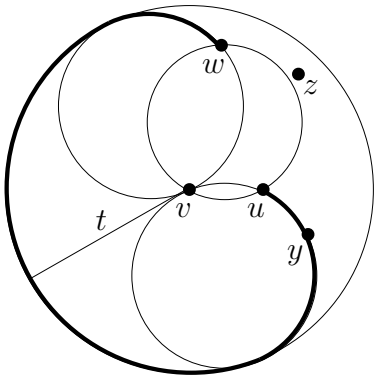


Figure 3.5: Illustration of the distance function δ^v. In this example, $CV_t = \{u, w\}$ and the thick curve is the set of points with distance $\delta^v_{CV_t}(x) = t$, e.g., y is at distance t. Since y is guaranteed to be a Voronoi neighbor, the algorithm inserts y into CV_t. There is no empty ball that touches both v and z, so $\delta^v_{CV_t}(z) = \infty$.

set of operations (Ω), and directed edges representing various dependencies between operations and squares. Consider executing an operation represented by $v_{(t)}$. If the `QTClippedVoronoi` function executed by $v_{(t)}$ reads a square s, it records this dependency by the edge $s \longrightarrow v_{(t)}$ (Section 3.2.4). If $v_{(t)}$ calls `Enqueue` to schedule an operation *op* acting on w into Ω, the `Enqueue` function first computes the rank and color of w and determines the time t_w of $op = w_{(t_w)}$. If $t_w > t$, `Enqueue` records this dependency by the edge $v_{(t)} \longrightarrow w_{(t_w)}$. In order to avoid duplicate operations in Ω, it schedules $w_{(t_w)}$ into Ω only if there is no edge from another operation towards $w_{(t_w)}$. Finally, to account for the dependencies that arise by inserting Steiner vertices, for each Steiner vertex w that $v_{(t)}$ inserts, the algorithm inserts the edge $v_{(t)} \longrightarrow s$, where s is the square that contains w. For the analysis, we tag each edge with the time of the operation that creates it, in the examples above, this time is t.

3.2.4 Computing Clipped Voronoi Cells

We represent the β-clipped Voronoi cell of a vertex v using the set of β-clipped Voronoi neighbors of v. To determine this set, CV, we perform a scan starting at v and proceeding along a circular frontier that moves away from v up to a maximum distance $t_{\max} = \beta \, \mathrm{NN}_{\mathcal{M}}(v)$; at time t the frontier is the boundary of the ball of radius t. When the scan reaches a vertex u, we determine whether u is a Voronoi neighbor of v or not. If it is, we add u to CV, otherwise, we discard it. This way, throughout the scan, we maintain the set CV, which consists of the β-clipped Voronoi neigh-

bors of v within the current scan distance. For efficiency, we need to ensure that the scan does not exceed the certificate region. Therefore, we use a distance function that differs from the Euclidean one. For any point x, we define the distance $\delta^v_{\mathcal{M}}(x)$ as the diameter of the smallest certificate ball that includes v and x on its boundary. If no such ball exists then we define $\delta^v_{\mathcal{M}}(x) = \infty$. By using this distance function, we eliminate the need to check whether a vertex that the scan reaches is a Voronoi neighbor or not, because the empty ball that defines its distance to v is a certificate ball for x. Figure 3.5 illustrates an example.

It is not clear how to compute the distance $\delta^v_{\mathcal{M}}(x)$ exactly without a prohibitive overhead. However, we can relax the requirement that the balls be empty of vertices in $\mathcal{M}$. Instead, using CV_t, which we define as the set of β-clipped Voronoi neighbors of v up to time t, we compute $\delta^v_{CV_t}(x)$, the diameter of the smallest ball with v and x on its boundary that includes no vertex of CV_t in its interior. As we advance time, this distance function does not decrease because adding new vertices into CV can only make it harder for a ball to be empty. Therefore, $\delta^v_{CV_t}(x) \leq \delta^v_{\mathcal{M}}(x)$ for all t. We also prove that at time t, when the scan reaches a vertex x, the distance to x is accurately computed, that $\delta^v_{CV_t}(x) = \delta^v_{\mathcal{M}}(x) = t$ (Lemma 3.2.1). A corollary of this result is that $CV_{t_{\max}} = CV$ and that the scan visits the certificate region of $\mathrm{Vor}^{\beta}_{\mathcal{M}}(v)$ because all points in the certificate region have distance less than $t_{\max}$.

Lemma 3.2.1 *During the computation of* $\mathrm{Vor}^{\beta}_{\mathcal{M}}(v)$*, when the scan reaches time* t*, for any vertex* u *satisfying* $\delta^v_{CV_t}(u) \leq t$*, we have* $\delta^v_{CV_t}(u) = \delta^v_{\mathcal{M}}(u)$.

Proof: Pick any vertex u satisfying $\delta^v_{CV_t}(u) \leq t$, i.e., there exists a ball of diameter $\delta^v_{CV_t}(u)$ touching both v and u and containing no vertex of CV_t in its interior. By the monotonicity of the distance function, we know that $\delta^v_{CV_t}(u) \leq \delta^v_{\mathcal{M}}(u)$. We want to show that $\delta^v_{CV_t}(u) < \delta^v_{\mathcal{M}}(u)$ is not possible by proving that there is no vertex of $\mathcal{M} \setminus CV_t$ inside this ball. Towards a contradiction, assume that there exists one, say w. Then, there exists a smaller ball for w, i.e., $\delta^v_{CV_t}(w) < \delta^v_{CV_t}(u)$. Again, by monotonicity, we have $\delta^v_{CV_{t'}}(w) \leq \delta^v_{CV_t}(w)$ for all $t' < t$. Therefore, w must have been

```
QTClippedVoronoi (v_(t), β) =
  t_max ← ∞, E ← {square of v}, CV ← ∅
  while ∃p ∈ E w/ δ^v_CV(p) < t_max
    p ← arg min_{p∈E} δ^v_CV(p)
    if p is a vertex then
      if CV = ∅ then t_max ← β|vp|
      CV ← CV ∪ {p}
    else (p is a square)
      CGInsertEdge (p ⟶ v_(t))
      E ← E ∪ {vertices of p}
      E ← E ∪ {squares adjacent to p}
  return CV
```

For computing the distance $\delta^v_{CV}(p)$

$$\begin{aligned} \text{minimize} \quad & |cv| \\ \text{subject to} \quad & |cv| = |cp| \\ & |cv| \leq |cu| \ \forall u \in CV \\ \text{distance } \delta^v_{CV}(p) &= 2|cv| \end{aligned}$$

Figure 3.6: Pseudo-code for computing clipped Voronoi cells.

discovered at some time $t' < t$ and inserted into $CV_{t'} \subset CV_t$. This contradicts our assumption that w is a vertex in $\mathcal{M} \setminus CV_t$. ■

Figure 3.6 shows the pseudo-code for the scan we describe above. The algorithm discretizes the scan using quadtree squares. Starting at the square of v, it explores outward from v using a queue E of events — reaching vertices and squares. Upon reaching a vertex, it updates CV, and upon reaching a square, it enqueues the vertices that the square contains and the unvisited squares it is adjacent to. To compute $\delta^v_{CV_t}(p)$, the algorithm finds a point c that is the center of a certificate ball of minimum radius using a convex program. For a quadtree square, the distance is the minimum distance to any point p in the square. This corresponds to letting p to be free variables in the above program and adding a box constraint ($2d$ linear constraints) on the coordinates of p. This leaves us with a quadratic program rather than a convex one, but it remains a program with $O(1)$ variables and constraints.

3.3 Analysis

3.3.1 Output Quality and Size

This section includes the proofs of the quality of the output of our algorithm, i.e., M is ρ-well-spaced and size-optimal. Lemma 3.3.3 proves size-optimality by showing that M is size-conforming. For ρ-well-spacedness, the first two lemmas prove that our algorithm fills vertices in an order such

that after filling a vertex the key invariant is satisfied—the vertex becomes and remains ρ-well-spaced. Therefore, our algorithm incrementally progresses towards a ρ-well-spaced output. In these two lemmas, let $\mathcal{M}$ be the set of vertices in the output at the beginning of rank r.

Lemma 3.3.1 *At the beginning of rank r, assume that every vertex $u \in \mathcal{M}$ with $\mathrm{NN}_{\mathcal{M}}(u) < \rho^r$ is ρ-well-spaced. Then, for every vertex $w \in \mathcal{M}$ with $\mathrm{NN}_{\mathcal{M}}(w) \in [\rho^r, \rho^{r+1})$, there exists a fill operation that acts on w at rank r.*

Proof: Consider a vertex $w \in \mathcal{M}$, let u be its nearest neighbor in $\mathcal{M}$, and assume that $\rho^r \leq |wu| < \rho^{r+1}$. Let $w_{(r_w,\mathtt{D})}$ and $u_{(r_u,\mathtt{D})}$ be the dispatch operations that act on w and u respectively. If $r_w \leq r$ and u is in the output at the beginning of rank r_w then $w_{(r_w,\mathtt{D})}$ schedules a fill operation that acts on w at rank r. Alternatively, $u_{(r_u,\mathtt{D})}$ schedules such a fill operation if $r_u \leq r$ and w is a β-clipped Voronoi neighbor of u at the beginning of rank r_u. We prove that one of the two conditions holds.

Analyzing the vertices w and u, whether they are both input vertices or one of them is a Steiner vertex inserted when the other one was in the output, in two of the three cases, we prove that the first condition holds. In the first case, if both w and u are input vertices then by Lemma 3.1.5, $r_w \leq r$. In the second case, in which w is a Steiner vertex and u is in the output when w is being inserted, consider the vertex v that creates w. By Fact 1, we know that $|wv| \leq |wu|$, which implies that $r_w \leq r$.

We prove that the second condition holds in the remaining case, in which u is a Steiner vertex and w is already in the output when u is being inserted. Similar to the previous case, we deduce that $r_u \leq r$. Since u is the nearest neighbor of w in $\mathcal{M}$, w is a Voronoi neighbor of u in $\mathcal{M}'$, where $\mathcal{M}' \subset \mathcal{M}$ is the output at the beginning of rank r_u. If u is ρ-well-spaced in $\mathcal{M}$ then $|wu| \leq 2\rho\,\mathrm{NN}_{\mathcal{M}}(u) < 2\beta\,\mathrm{NN}_{\mathcal{M}'}(u)$. Otherwise, the assumption of the lemma implies $\rho^r \leq \mathrm{NN}_{\mathcal{M}}(u)$. Since $|wu| < \rho^{r+1}$, we get $|wu| < \rho\,\mathrm{NN}_{\mathcal{M}}(u) < 2\beta\,\mathrm{NN}_{\mathcal{M}'}(u)$. Either way, w is a β-clipped Voronoi neighbor of u in $\mathcal{M}'$. ∎

Lemma 3.3.2 (Progress) *At the beginning of rank r, every vertex $u \in \mathcal{M}$ with $\mathrm{NN}_{\mathcal{M}}(u) < \rho^r$ is ρ-well-spaced.*

Proof: We use induction. At the minimum rank, there are no vertices with smaller nearest-neighbor distance, so the claim is trivially true. Assume that the lemma holds up to rank r, that is, every vertex $u \in \mathcal{M}$ with $\mathrm{NN}_{\mathcal{M}}(u) < \rho^r$ is ρ-well-spaced. For rank $r+1$, let $\mathcal{M}' \supset \mathcal{M}$ be the set of vertices in the output at the beginning of rank $r+1$ and consider a vertex $w \in \mathcal{M}'$ with $\mathrm{NN}_{\mathcal{M}'}(w) < \rho^{r+1}$. We claim that $w \in \mathcal{M}$; towards a contradiction, assume that $w \in \mathcal{M}' \setminus \mathcal{M}$. Then, w is a Steiner vertex inserted at rank r. Repeatedly applying Fact 2 for each (Steiner) vertex in $\mathcal{M}' \setminus \mathcal{M}$, we see that the nearest neighbors of these Steiner vertices are at distance $\geq \rho^{r+1}$; in particular, $\mathrm{NN}_{\mathcal{M}'}(w) \geq \rho^{r+1}$. This is a contradiction to our criteria $\mathrm{NN}_{\mathcal{M}'}(w) < \rho^{r+1}$, thus, $w \in \mathcal{M}$. Furthermore, $\mathrm{NN}_{\mathcal{M}}(w) < \rho^{r+1}$ for similar reasons. If $\mathrm{NN}_{\mathcal{M}}(w) < \rho^r$ then by our induction hypothesis w is ρ-well-spaced. Otherwise, if $\rho^r \leq \mathrm{NN}_{\mathcal{M}}(w) < \rho^{r+1}$, by Lemma 3.3.1, there exists a fill operation that acts on w at rank r. After executing that operation, w becomes ρ-well-spaced. Finally, Fact 2 implies that w remains ρ-well-spaced. ∎

Lemma 3.3.3 *The output M is size-conforming and size-optimal with respect to N.*

Proof: We use induction over the order in which the algorithm inserts Steiner vertices and show that there exists a constant c such that for every $v \in \mathsf{M}$, $c\,\mathrm{NN}_{\mathsf{M}}(v) \geq \mathrm{lfs}(v)$, thereby proving that M is size-conforming. In the base case, every vertex is an input vertex and the nearest neighbor of an input vertex is exactly the local feature size. For the inductive case, assume that there exists a constant c such that, for every $v \in \mathcal{M}$, we have $c\,\mathrm{NN}_{\mathcal{M}}(v) \geq \mathrm{lfs}(v)$. Furthermore, assume that v inserts a Steiner vertex w and the new output is $\mathcal{M}' = \mathcal{M} \cup \{w\}$. We analyze the inductive claim for w and for any vertex $u \in \mathcal{M}$ separately. For w, by Fact 1 we know that $|wv| \geq \rho\,\mathrm{NN}_{\mathcal{M}}(v)$ and $\mathrm{NN}_{\mathcal{M}'}(w) = |wv|$. By the triangle inequality, lfs satisfies the Lipschitz condition: $\mathrm{lfs}(v) + |wv| \geq \mathrm{lfs}(w)$. By the inductive hypothesis, $c\,\mathrm{NN}_{\mathcal{M}}(v) \geq \mathrm{lfs}(v)$. Therefore, we have $(\frac{c}{\rho} + 1)|wv| = (\frac{c}{\rho} + 1)\,\mathrm{NN}_{\mathcal{M}'}(w) \geq \mathrm{lfs}(w)$.

For any vertex $u \in \mathcal{M}$, if $\mathrm{NN}_{\mathcal{M}}(u) = \mathrm{NN}_{\mathcal{M}'}(u)$ then the claim holds trivially. Otherwise, assume that $\mathrm{NN}_{\mathcal{M}}(u) > \mathrm{NN}_{\mathcal{M}'}(u) = |wu|$. By the Lipschitz condition, we have $|wu| + \mathrm{lfs}(w) \geq \mathrm{lfs}(u)$ and by Fact 1 we know $|wu| \geq |wv|$. Combining these by the bound we obtained for $\mathrm{lfs}(w)$, we get $(\frac{c}{\rho} + 2)|wu| = (\frac{c}{\rho} + 2)\,\mathrm{NN}_{\mathcal{M}'}(u) \geq \mathrm{lfs}(u)$. Solving for $c \geq \frac{c}{\rho} + 2$, we conclude that any $c \geq \frac{2\rho}{\rho - 1}$ suffices to prove the inductive step. Therefore, M is size-conforming and hence size-optimal [60]. ∎

Theorem 3.3.4 *`StableWS` constructs a size-optimal ρ-well-spaced superset M of its input N.*

Proof: The property that M is ρ-well-spaced follows from the Progress Lemma and the fact that `StableWS` iterates over all ranks. Lemma 3.3.3 proves the size bound. ∎

3.3.2 Runtime

We analyze the running time of our static algorithm and rely on two lemmas that are useful in the analysis of our dynamic algorithm. The first lemma (Lemma 3.3.5) proves that throughout the algorithm, the nearest-neighbor distance of a vertex v changes only by a constant factor. The second lemma (Lemma 3.3.6) proves that all operations acting on v have rank $\lfloor \log_\rho \mathrm{NN}_\infty(v) \rfloor \pm O(1)$; none are scheduled too early nor too late.

Lemma 3.3.5 *Let t be the time at which v is created ($t = 0$ for input vertices). Then, $\mathrm{NN}_t(v) \in \Theta(\mathrm{NN}_\infty(v))$.*

Proof: As time progresses, more vertices are added, so the nearest neighbor distance can only shrink: $\mathrm{NN}_t(v) \geq \mathrm{NN}_\infty(v)$. For the upper bound, we analyze input vertices and Steiner vertices separately. By definition, an input vertex v has $\mathrm{lfs}(v) = \mathrm{NN}_0(v)$. The algorithm is size-conforming (Lemma 3.3.3), so $\mathrm{NN}_0(v) = \mathrm{lfs}(v) \in O(\mathrm{NN}_\infty(v))$. For a Steiner vertex w that is created at time $t = (r, \mathtt{F}, c)$, Fact 1 implies that $\rho^{r+1} \leq \mathrm{NN}_t(w) \leq \beta\rho^{r+1}$. For any other Steiner vertex u that is created later, the same fact implies that $\rho^{r+1} \leq |uw|$ which means $\rho^{r+1} \leq \mathrm{NN}_\infty(w)$. Therefore, $\mathrm{NN}_t(w) \leq \beta\rho^{r+1} \leq \beta\,\mathrm{NN}_\infty(w)$. ∎

Lemma 3.3.6 *If an operation at rank r acts on v then* $\mathrm{NN}_\infty(v) \in \Theta(\rho^r)$.

Proof: Consider an operation $v_{(t_v)}$ at rank r. Lemma 3.3.5 implies that it suffices to prove $\mathrm{NN}_t(v) \in \Theta(\rho^r)$, where t is the time $v_{(t_v)}$ is created. If $v_{(t_v)}$ is a dispatch operation and v is an input vertex then $t = 0$ and our algorithm uses the size of the square that contains v to approximate $\mathrm{NN}_0(v)$ and Lemma 3.1.5 implies that $\mathrm{NN}_0(v) \in \Theta(\rho^r)$. If $v_{(t_v)}$ is a dispatch operation and v is a Steiner vertex then we know that v is created at time t by a fill operation acting on a vertex u and that $r = \lfloor \log_\rho |vu| \rfloor$. Since v is picked from the Voronoi cell of u, $|vu| = \mathrm{NN}_t(v)$, thus $\mathrm{NN}_t(v) \in \Theta(\rho^r)$. If $v_{(t_v)}$ is a fill operation created by the dispatch operation acting on v, then we know that the rank is computed exactly, i.e., $r = \lfloor \log_\rho \mathrm{NN}_t(v) \rfloor$. The last case is that $v_{(t_v)}$ is a fill operation created by a dispatch operation op acting on another vertex u at rank r'. We know that op assigns the rank of v to be $r = \left\lfloor \log_\rho |vu| \right\rfloor$. Since $\mathrm{NN}_t(v) \leq |vu|$, we get $\mathrm{NN}_t(v) < \rho^{r+1}$, thus, the upper bound holds. For the lower bound, since $|vu| \geq \rho^r$, it suffices to show that $\mathrm{NN}_t(v) \in \Omega(|vu|)$. If $\mathrm{NN}_t(v) \geq \rho^{r'}$, we show that $\rho^{r'} \in \Omega(|vu|)$ by applying the result from above for op, that $\mathrm{NN}_t(u) \in \Theta(\rho^{r'})$, and by using the fact that v is a β-clipped Voronoi neighbor of u at time t, that $2\beta\,\mathrm{NN}_t(u) \geq |vu|$. Otherwise, if $\mathrm{NN}_t(v) < \rho^{r'}$ then by the Progress Lemma, v is ρ-well-spaced at time t. Since v and u are Voronoi neighbors at time t, this implies that u is a ρ-clipped Voronoi neighbor of v. Therefore, $2\rho\,\mathrm{NN}_t(v) \geq |vu|$ and we prove in all cases that $\mathrm{NN}_\infty \in \Theta(\rho^r)$. ∎

Lemma 3.3.7 *At the beginning of rank r, any point p inside an empty ball of radius ρ^r satisfies* $\mathrm{lfs}(p) \in \Omega(\rho^r)$.

Proof: Let c be the center of an empty ball of radius ρ^r, i.e., $\mathrm{NN}_t(c) \geq \rho^r$, where $t = (r, \mathtt{D})$. Given this ball, let p be a point inside it. For some constant ϵ whose value we will set later, if $\mathrm{NN}_t(p) \geq \epsilon\rho^r$ then our proof is done. Otherwise, if $\mathrm{NN}_t(p) < \epsilon\rho^r$, let q be the point at distance $\rho^r/2$ away from c on the ray from c to p, and let $u \in \mathsf{M}_t$ be the vertex nearest q. Then, we claim that $\mathrm{NN}_t(u) > \rho^r/2\rho$ and for some small enough ϵ that u is the vertex nearest p as well. Using the Lipschitz condition, we get $\mathrm{lfs}(p) + |pu| \geq \mathrm{lfs}(u) \geq \mathrm{NN}_t(u)$. Then, our claims imply $\mathrm{lfs}(p) > \rho^r/2\rho - |pu| > (1/2\rho - \epsilon)\rho^r$ and consequently prove our lemma statement.

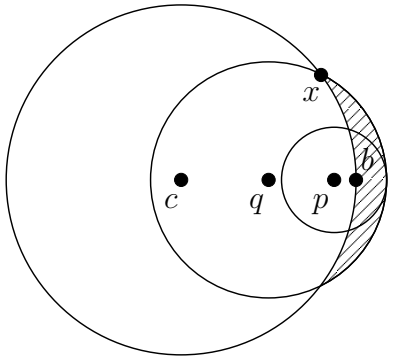


Figure 3.7: Illustration of the proof of Lemma 3.3.7. There is an empty ball centered at c of radius ρ^r and p is a point inside this ball. The nearest neighbor of p is within $\epsilon\rho^r$ distance (the small ball). The point q, $\rho^r/2$ away from c on the ray from c to p, has its nearest neighbor within $(1/2+\epsilon)\rho^r$ distance (the midsize ball), inside the shaded region. The point x in the shaded region is one of the farthest away from b. The lemma is proven by showing that the shaded region can be made small enough.

Our first claim is trivially true if $\mathrm{NN}_t(u) \geq \rho^r$; otherwise, u must be ρ-well-spaced by Lemma 3.3.2, which implies that q, being a point inside the Voronoi cell of u, is within $\rho\,\mathrm{NN}_t(u)$ distance of u. In other words, $\mathrm{NN}_t(u) > |qu|/\rho$. Since u is outside the empty ball, $|qu| \geq \rho^r/2$, therefore, we prove our first claim.

For proving the second claim, we first observe that u, the nearest neighbor of q, lies inside the ball of radius $(1/2+\epsilon)\rho^r$ centered at q because there is a vertex inside the ball of radius $\epsilon\rho^r$ centered at p. Since the ball of radius ρ^r centered at c is empty, the crescent defined by this empty ball and the ball centered at q contains u. We will show that this region, shaded in Figure 3.7, is contained in a ball of diameter $\rho^r/2\rho$. Then, using the fact that the nearest neighbor of u is at least $\rho^r/2\rho$ far away from u, we prove that u is the only vertex in this region and therefore the vertex nearest p. In order to bound the diameter, for appropriate ϵ, we show that any point of the shaded region is within $\rho^r/4\rho$ distance of the point b, that is located ρ^r away from c on the ray from c to p: observe that any point x on the intersection of the ball centered at c of radius ρ^r and the ball centered at q of radius $(1/2+\epsilon)\rho^r$ is the farthest away from b. Since xq is the median of the side cb of the triangle cbx, using Apollonius' theorem, one can get $|xb|^2 = 2\epsilon(\epsilon+1)\rho^{2r}$. Solving for $|xb| \leq \rho^r/4\rho$, we see that any $\epsilon \leq \sqrt{1/4 + 1/32\rho^2} - 1/2$ suffices to prove our claim and therefore our lemma. ∎

Theorem 3.3.8 *Computing the β-clipped Voronoi cell of a vertex takes $O(1)$ time.*

Proof: When the `QTClippedVoronoi` function visits a quadtree square, that square either intersects the certificate region, or it is a neighbor of a square that intersects the certificate region. Let r be the rank at which the function is called and s be a square that intersects the certificate region. By Lemma 3.3.7, for any point $p \in s$, that is inside the certificate region, $\mathrm{lfs}(p) \in \Omega(\rho^r)$. Furthermore, by Lemma 3.1.5, $\mathrm{lfs}(p) \in \Theta(|s|)$, which implies that s covers a volume of $\Omega(\rho^r)$. By Lemmas 3.3.5 and 3.3.6, the certificate region of the β-clipped Voronoi cell of v is within a ball of radius $O(\rho^r)$. This implies that there are only $O(1)$ squares that intersect the certificate region. Thanks to the grading condition, the squares have bounded number of squares adjacent to them, therefore `QTClippedVoronoi` visits only $O(1)$ squares. The function also does work iterating over the vertices each square contains. By Lemma 3.3.3, a vertex u has a nearest neighbor no closer than $\Omega(\mathrm{lfs}(u))$; meanwhile, again by Lemma 3.1.5, the quadtree square that contains u has side length $O(\mathrm{lfs}(u))$. Hence, each square contains only a constant number of vertices and the total work is $O(1)$. ■

Lemma 3.3.9 *Dispatch and fill operations run in $O(1)$ time.*

Proof: The main costs of an operation $v_{(t)}$ are the β-clipped Voronoi cell computations and the loops. Theorem 3.3.8 shows that Steiner vertex insertions and the clipped Voronoi cell computations take constant time. This implies that there are a constant number of β-clipped Voronoi neighbors of v. If $v_{(t)}$ is a dispatch operation, it iterates over each of them, this takes constant time. If $v_{(t)}$ is a fill operation, it has a loop that inserts Steiner vertices until v is ρ-well-spaced. For each inserted Steiner vertex w, Fact 1 implies $\mathrm{NN}_t(w) \geq \rho\,\mathrm{NN}_t(v)$. Thus, we can associate non-overlapping empty balls of radius $\rho\,\mathrm{NN}_t(v)/2$ around every Steiner vertex. Since the Steiner vertices are in a ball of radius $\beta\,\mathrm{NN}_t(v)$ around v, a packing argument shows that $v_{(t)}$ inserts a constant number of Steiner vertices. This concludes that the operation represented by $v_{(t)}$ runs in constant time. ■

Lemma 3.3.10 *For every vertex $v \in \mathsf{M}$, there are $O(1)$ operations that act on v.*

Proof: By Lemma 3.3.6, any operation acting on v has rank $\lfloor \log_\rho \mathrm{NN}_\infty(v) \rfloor \pm O(1)$. Therefore, if we can bound the number of operations acting on v at each rank by a constant, our claim will hold. There is only one dispatch operation for each vertex, so we only need to count the fill operations scheduled by other dispatch operations. Fix r and consider a dispatch operation acting on u at time $t' = (r', \mathtt{D})$ scheduling a fill operation acting on v at rank r. Then, v is β-clipped Voronoi neighbor of u, in other words, $|uv| \leq 2\beta \,\mathrm{NN}_{t'}(u)$. The fact that the fill operation is scheduled for rank r implies $\rho^r \leq |uv| < \rho^{r+1}$. Considering the dispatch operation, Lemmas 3.3.5 and 3.3.6 show that $\mathrm{NN}_{t'}(u) = O(\rho^{r'})$. These facts imply $\rho^r = O(\rho^{r'})$. Again by Lemma 3.3.6, we know that there exists an empty ball around u with radius $\Omega(\rho^{r'})$ which is $\Omega(\rho^r)$ by the previous assertion. We already know that $|uv| < \rho^{r+1}$, therefore, a packing argument proves our claim. ■

Theorem 3.3.11 *`StableWS` runs in $O(n \log \Delta)$ time.*

Proof: As shown in Section 3.1.2, building the quadtree takes $O(n \log \Delta)$ time. By Lemmas 3.3.9 and 3.3.10, the rest of the algorithm takes $O(m)$ time, where $m = |\mathsf{M}|$. The total runtime is $O(n \log \Delta + m)$. That $m \in O(n \log \Delta)$ follows from our dynamic bounds. ■

3.3.3 Dynamic Stability

We call two inputs N and N' *related* if they differ by one vertex, i.e., N' can be obtained from N by inserting or deleting a vertex. To analyze the stability of the algorithm `StableWS`, we define a notion of distance between two executions with related inputs. We prove that this distance is bounded by $O(\log \Delta)$ in the worst-case, where Δ is the larger geometric spread of the inputs N and N' (Lemma 3.3.16).

As described in Section 3.2, `StableWS`(N) constructs a computation graph $G = (V, E)$ by building quadtree squares Σ and a set of operations Ω. The set of nodes V is $\Sigma \cup \Omega$; the edges E represent the dependencies in the computation. For another input set N' which is related to N, consider running `StableWS`(N') and creating $G' = (V', E')$, Σ', and Ω' similarly. We

define a recursive *matching* between the nodes of the two executions: two operations $v_{(t)} \in \Omega$ and $v'_{(t')} \in \Omega'$ match, if $v = v'$, $t = t'$, and either the times $t = t' = 0$ or there exist matching operations $w_{(\tau)} \in \Omega$ and $w'_{(\tau')} \in \Omega'$ such that the computation graphs G and G' include the edges $w_{(\tau)} \longrightarrow v_{(t)}$ and $w'_{(\tau')} \longrightarrow v'_{(t')}$ respectively; and, two squares $s \in \Sigma$ and $s' \in \Sigma'$ match if s and s' have the same corner points. We denote this matching by $\mu : V' \to V$, where $\mu = \{(v'_{(t')}, v_{(t)}) \mid v'_{(t')}$ and $v_{(t)}$ match$\}$. We denote the domain and the range of μ by $\mathsf{dom}(\mu)$ and $\mathsf{range}(\mu)$. Using this matching, we define $\mu' = \mu \cup \{(u, u) \mid u \in V' \setminus \mathsf{dom}(\mu)\}$ to be a total function defined on the nodes V' of G'. We combine the computation graphs in a *union graph* $G^{\cup} = (V \cup \mu'(V'), E \cup \mu'(E'))$, where $\mu'(E') = \{(\mu'(u), \mu'(v)) \mid (u, v) \in E'\}$. Intuitively, the union graph injects G' into G under the guidance of μ by extending G with the unmatched nodes of G', unifying the matched nodes, and adding the edges of G' while redirecting them to the matched nodes appropriately. In order to capture the dependencies between two operations, we define a path in the union graph to be a dependency path if the times of the edges on the path do not decrease. Lemma 3.3.12 allows us to refine this definition: a path $(x_0, x_1, \ldots, x_k)$ is a *dependency path* if the times of the edges $x_0 \longrightarrow x_1,\ x_1 \longrightarrow x_2, \cdots, x_{h-1} \longrightarrow x_h$ increase monotonically.

Lemma 3.3.12 *Consider coloring parameters $\ell(r)$ and κ that satisfy the inequalities $\ell(r) < \rho^r/\sqrt{d}$ and $\kappa > 1 + 3\beta\rho^{r+1}/\ell(r)$. Then, any two fill operations at the same rank are independent if the vertices they act on have the same color.*

Proof: Consider two operations $v_{(t)}$ and $u_{(t)}$, where $t = (r, \mathtt{F}, c)$. Let $\mathcal{M}$ be the set of vertices in the output at the beginning of rank r. If both v and u are ρ-well-spaced in $\mathcal{M}$ then $v_{(t)}$ and $u_{(t)}$ are independent. Otherwise, if v is not ρ-well-spaced the Progress Lemma implies that $\mathrm{NN}_{\mathcal{M}}(v) \geq \rho^r$. Since $\ell(r) < \rho^r/\sqrt{d}$, the diameter of an r-tile is less than ρ^r, and thus v and u cannot be in the same r-tile. Since v and u have the same color, v and u are far apart, more precisely, $|vu| \geq (\kappa - 1)\ell(r) > 3\beta\rho^{r+1}$. The fact that our construction algorithm creates these operations implies $\mathrm{NN}_{\mathcal{M}}(v), \mathrm{NN}_{\mathcal{M}}(u) < \rho^{r+1}$. Then, the (ρ, β) picking regions and the certificate regions of the β-clipped Voronoi cells of v and u are inside balls of radii $\beta\rho^{r+1}$ and $2\beta\rho^{r+1}$ around these

vertices respectively. Using the triangle inequality, we know that that the (ρ, β) picking region of one of them does not intersect the certificate region of the β-clipped Voronoi cell of the other; therefore, $v_{(t)}$ and $u_{(t)}$ are independent. ∎

We partition the nodes of the union graph $G^{\cup} = (V^{\cup}, E^{\cup})$ into several categories. The nodes $V^{-} = V \setminus \mathsf{range}(\mu)$ are called *obsolete* (squares Σ^{-}, operations Ω^{-}); these are the nodes of G that have no matching pairs in G'. The nodes $V^{+} = V' \setminus \mathsf{dom}(\mu)$ are called *fresh* (squares Σ^{+}, operations Ω^{+}); these are the nodes of G' that have no matching pairs in G. Furthermore, we call a square $s \in V^{\cup}$ *inconsistent* if it is fresh or obsolete, or if it contains the vertex $\hat{v}$ of the symmetric difference of N and N'. We define an operation $v_{(t)} \in \mathsf{range}(\mu)$ to be *inconsistent* if it is reachable from an inconsistent square via a dependency path. We represent inconsistent nodes with $V^{\times}$ (squares $\Sigma^{\times}$, operations $\Omega^{\times}$). We define the *distance* between the executions with related inputs N and N' to be the number of obsolete, fresh, or inconsistent operations of the union graph, i.e., $|\Omega^{-} \cup \Omega^{+} \cup \Omega^{\times}|$.

Lemma 3.3.13 *For every operation in $\Omega^{-} \cup \Omega^{+} \cup \Omega^{\times}$, there exists a dependency path from a square in $\Sigma^{\times}$.*

Proof: By definition, an inconsistent operation can be reachable via a dependency path from $\Sigma^{\times}$. For unmatched operations, assume towards a contradiction that there exist an operation in $\Omega^{-} \cup \Omega^{+}$ that is not reachable from $\Sigma^{\times}$. Let $v_{(t)}$ be the earliest of such operations. Let us assume that $v_{(t)}$ represents a dispatch operation, and that v is an input vertex. Since $v_{(t)}$ does not depend on an inconsistent square, it does not read one. Therefore, v is in $\mathsf{N} \cap \mathsf{N}'$ and lies in identical squares in both executions, which implies that its nearest neighbor approximation is the same in both executions. Hence, there exists an operation $v_{(t)}$ in the other execution as well. The definition of μ matches these operations because in both computation graphs contain the edge $v_{(0)} \longrightarrow v_{(t)}$. For the remaining cases, there exists an edge $w_{(\tau)} \longrightarrow v_{(t)}$ for some unmatched or inconsistent operation $w_{(\tau)}$ with $\tau < t$. By the minimality of $v_{(t)}$, $w_{(\tau)}$ can be reached via a dependency path from a square in $\Sigma^{\times}$. Extending that path to $v_{(t)}$ proves the contradiction. ∎

As proven in the previous section, the function `QTClippedVoronoi` satisfies the following locality property: for a given input N, a size-conforming set of vertices $\mathcal{M} \supset \mathsf{N}$, and a square s read by `QTClippedVoronoi`, for all $x \in s$, $|vx| \in O(\mathrm{NN}_{\mathcal{M}}(v))$. This property allows us to relate the operations on a dependency path geometrically.

Lemma 3.3.14 *Consider two operations $w_{(\tau)}$ and $v_{(t)}$ in $G^{\cup}$. If there exists a dependency path from $w_{(\tau)}$ to $v_{(t)}$ and rank of t is r, then $|vw| \in O(\rho^r)$.*

Proof: First, we show that for any edge in $G^{\cup}$, the distance between its nodes is short. We define the distance between a square and an operation to be the distance from the vertex of the operation to the farthest point in the square, and the distance between two operations to be the distance between their vertices. Consider an edge $e \in E$ with time t_e whose rank is r_e. The edge e consists of an operation $u_{(t_e)} \in \Omega$ and either a square s that $u_{(t_e)}$ accesses (reads/writes) or another operation $u'_{(t')}$ that it schedules. Using the locality result stated above, we bound the distance between $u_{(t_e)}$ and s by $O(\mathrm{NN}_{t_e}(u))$. Also, $u'_{(t')}$ is within the same distance. Lemmas 3.3.5 and 3.3.6 bound $\mathrm{NN}_{t_e}(u)$ by $O(\rho^{r_e})$; thus, the distance between the nodes of e is at most $\alpha\rho^{r_e}$, where α is a constant in the big-Oh notation. The same analysis applies for any edge $e' \in E^{\cup}$.

By definition of dependency paths, the times of the edges on a dependency path from $w_{(\tau)}$ to $v_{(t)}$ monotonically increase. Assuming that the rank of τ is r', there can be at most κ^d edges for each rank between r' and r. Therefore, in the worst case, the distance between v and w is bounded by $\sum_{i=r'}^{r} \kappa^d \alpha \rho^i = \alpha\kappa^d \frac{\rho^{r+1}-\rho^{r'}}{\rho-1} < \alpha\kappa^d \frac{\rho^{r+1}}{\rho-1}$. Consequently, $|vw| \in O(\rho^r)$. ∎

In order to bound the distance between the executions with inputs N and N' which generate outputs M and M', we focus on the vertices rather than the operations. We define a vertex to be *affected* if there exists an obsolete, a fresh, or an inconsistent operation of it. Since there is a constant number of operations acting on a given vertex (Lemma 3.3.10), the number of affected vertices measures the distance asymptotically. We define the sets of affected vertices in both executions: $\widehat{\mathsf{M}} = \{v \mid v_{(t)} \in \Omega^{-} \cup \Omega^{\times}\}$ and $\widehat{\mathsf{M}}' = \{v \mid v_{(t)} \in \Omega^{+} \cup \Omega^{\times}\}$. The next two lemmas bound the number of affected vertices.

Lemma 3.3.15 *For any vertex* $v \in \widehat{\mathsf{M}}$, $|v\hat{v}| \in O(\mathrm{NN}_{\mathsf{M}}(v))$ *and for any* $v \in \widehat{\mathsf{M}}'$, $|v\hat{v}| \in O(\mathrm{NN}_{\mathsf{M}'}(v))$.

Proof: We prove the lemma for $v \in \widehat{\mathsf{M}}$; symmetric arguments apply for $\widehat{\mathsf{M}}'$. By definition of $\widehat{\mathsf{M}}$, there exists an operation $v_{(t_v)} \in \Omega^- \cup \Omega^\times$ at rank r. Lemma 3.3.13 suggests that there exists a dependency path from a square $s \in \Sigma^\times$ to $v_{(t_v)}$. Let $s \longrightarrow u_{(t_u)}$ be the first edge on this path, where the rank of t_u is r_u. By Lemma 3.3.14, we know that $|vu| \in O(\rho^r)$. By the fact that the operation that $u_{(t_u)}$ represents reads s, we know $|us|$ is in $O(\rho^{r_u})$ and by lemmas 3.1.3 and 3.1.4 the quadtree functions `QTInsertInput` and `QTDeleteInput` guarantee that $|s\hat{v}| \in O(|s|)$ which is in $O(\rho^{r_u})$ as well. Using the triangle inequality and the fact that $r_u \leq r$, we bound $|v\hat{v}|$ by $O(\rho^r)$. It only remains to prove that there is a ball around v of radius $\Omega(\rho^r)$ empty of vertices of M. Lemma 3.3.6 proves precisely this. ■

Lemma 3.3.16 (Distance) *The distance between two executions with related inputs is bounded by* $O(\log \Delta)$.

Proof: The distance is asymptotically bounded by the sizes of the affected sets of vertices $|\widehat{\mathsf{M}}|$ and $|\widehat{\mathsf{M}}'|$. Consider the vertices $v \in \widehat{\mathsf{M}}$ with $|v\hat{v}| \in [2^i, 2^{i+1})$. By Lemma 3.3.15, we can assign non-overlapping empty balls of radius $\Omega(2^i)$ to them. Therefore, there is a constant number of such vertices for any i. At most $O(\log \Delta)$ values of i cover $\widehat{\mathsf{M}}$, so $|\widehat{\mathsf{M}}| \in O(\log \Delta)$. Similar arguments apply to $\widehat{\mathsf{M}}'$. ■

3.4 Dynamic Update

We describe an algorithm for dynamically updating the output of `StableWS` when the input is modified by insertion/deletion of a vertex, prove it correct (Lemma 3.4.2) and efficient (Theorem 3.4.3). Furthermore, we prove a lower bound for the update runtime by showing that there exists examples for which inserting/deleting an input vertex causes $\Omega(\log \Delta)$ Steiner vertices to be inserted/deleted (Theorem 3.4.5).

```
Global queues:  Ω^⊖, Ω^⊕, Ω^⊗

PropagateWS (Σ^−, v̂) =
  for each s ∈ Σ^− ∪ {square of v̂}
    MarkReaders(s, 0)
    for each input vertex v ≠ v̂ ∈ s
      Undo(v_(0))
      apxnnv ← |square of v|
      Enqueue(v, D, apxnnv, v_(0), Ω^⊕)
  r_min ← min rank in Ω^⊖ ∪ Ω^⊕ ∪ Ω^⊗
  for r = r_min to ⌊log_ρ √d⌋
    for each v_(r,D) ∈ Ω^⊖ ∪ Ω^⊗
      Undo(v_(r,D))
    for each v_(r,D) ∈ Ω^⊕ ∪ Ω^⊗
      Dispatch(v_(r,D), Ω^⊕)
    for c = 0 to κ^d − 1
      for each v_(r,F,c) ∈ Ω^⊖ ∪ Ω^⊗
        Undo(v_(r,F,c))
      for each v_(r,F,c) ∈ Ω^⊕ ∪ Ω^⊗
        Fill(v_(r,F,c), Ω^⊕)
        for each Steiner w inserted by v
          MarkReaders(square of w, (r, F, c))

MarkReaders (s, t) =
  for each edge s ⟶ v_(t_v)
    if t_v > t then Ω^⊗ ← Ω^⊗ ∪ {v_(t_v)}

Insert (Q, v̂) =
  (Q′, Σ^−) ← QTInsertInput(Q, v̂)
  appnnv̂ ← |square of v̂|
  Enqueue(v̂, D, apxnnv̂, v̂_(0), Ω^⊕)
  PropagateWS(Σ^−, v̂)
  return Q′

Delete (Q, v̂) =
  (Q′, Σ^−) ← QTDeleteInput(Q, v̂)
  Undo(v̂_(0))
  PropagateWS(Σ^−, v̂)
  return Q′

Undo (v_(t)) =
  for each edge s ⟶ v_(t)
    CGDeleteEdge(s ⟶ v_(t))
  for each edge v_(t) ⟶ w_(t_w)
    CGDeleteEdge(v_(t) ⟶ w_(t_w))
    if ∄ edge · ⟶ w_(t_w) then
      Ω^⊖ ← Ω^⊖ ∪ {w_(t_w)}
    if t = (r, F, c) then
      s_w ← square of w
      MarkReaders(s_w, t)
      CGDeleteEdge(v_(t) ⟶ s_w)
  if v_(t) ∈ Ω^⊖ then
    Ω^⊗ ← Ω^⊗ \ {v_(t)}
```

Figure 3.8: The pseudo-code of the dynamic algorithm.

3.4.1 Update Algorithm

Our dynamic update algorithm is a change-propagation algorithm. Given the input modification, the update algorithm re-executes the actions of the stable algorithm for the part of the computation affected by the modification and undoes the part of the computation that becomes obsolete. More precisely, the algorithm maintains distinct set of operations for removal $\Omega^{\ominus}$ (obsolete operations), for execution $\Omega^{\oplus}$ (fresh operations), and for re-execution $\Omega^{\otimes}$ (inconsistent operations), which contain operations representing the operations that become obsolete, that need to be executed, and that become inconsistent respectively. The inconsistent operations are updated by deleting their old versions and executing them again, which may now perform actions different than before. The algorithm removes and executes operations in the same order as the stable algorithm. It uses

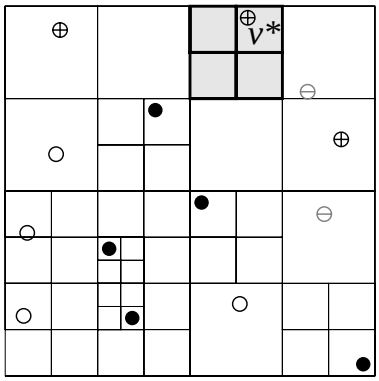


Figure 3.9: Dynamic update after insertion of $\hat{v}$. Solid vertices are input (N), vertices marked $+$ are inserted, vertices marked $-$ are deleted. Gray squares are inconsistent. The four smaller gray squares are fresh; they replace the bigger obsolete square.

the `Undo` function to remove obsolete operations and the `Dispatch` and `Fill` functions of the stable algorithm for executing fresh operations.

Figure 3.8 shows the pseudo-code for the `Insert` and `Delete` functions for inserting and deleting a vertex $\hat{v}$ into and from the input, and the `PropagateWS` function for dynamic updates. Given $\hat{v}$, `Insert`/`Delete` updates the quadtree, determines the set of inconsistent squares $\Sigma^{\otimes}$, and initializes the fresh/obsolete set by creating a dispatch operation or by marking the old dispatch operation acting on $\hat{v}$. Both functions then call `PropagateWS`.

The `PropagateWS` function starts by updating the operation sets by finding the input vertices that are contained in the inconsistent squares, deleting their dispatch operations, and creating new dispatch operations for them. It also initializes the inconsistent operation set, as `MarkReaders` marks inconsistent all operations that read an inconsistent square. The algorithm then proceeds in time order, first undoing the obsolete and inconsistent operations and then performing the fresh and inconsistent operations by calling `Dispatch` and `Fill` (Figure 3.4). The `Undo` function undoes the work of obsolete and fresh operations by removing the Steiner vertices they insert (if any) and by removing the edges incident to them (dependencies) from the computation graph. While removing these edges, it also marks for removal the operations that lose their last incoming edge from other operations—these operations would not be created in a fresh execution with the modified input. The `Undo` function also calls the `MarkReaders` function to expand the set of inconsistent operations as the set of vertices in a square changes due to the removal of Steiner

vertices. After `Undo` finishes its work and as the algorithm executes fresh fill operations, it calls the `MarkReaders` function for a similar reason: to update the set of inconsistent operations due to the insertion of fresh Steiner vertices.

As their notation suggests, the obsolete, fresh, and inconsistent operations used by the algorithm are related to those defined in the stability analysis; the following lemma makes the relationship between them precise.

Lemma 3.4.1 *The set of operations processed in the dynamic update algorithm, $\Omega^{\ominus} \cup \Omega^{\oplus} \cup \Omega^{\otimes}$, is a subset of the set of obsolete, fresh, and inconsistent operations, $\Omega^{-} \cup \Omega^{+} \cup \Omega^{\times}$.*

Proof: Let $A = \Omega^{\ominus} \cup \Omega^{\oplus} \cup \Omega^{\otimes}$ and $B = \Omega^{-} \cup \Omega^{+} \cup \Omega^{\times}$. Towards a contradiction, assume that $A \not\subset B$ and let $v_{(t)}$ be the earliest operation in $A \setminus B$. If $v_{(t)} \in \Omega^{\ominus}$ then either $v_{(t)}$ is a dispatch operation acting on an input vertex or there is an edge from another operation $w_{(\tau)} \in \Omega^{\ominus} \cup \Omega^{\otimes}$ towards $v_{(t)}$. In the first case, $v_{(t)}$ depends on a square in $\Sigma^{\times}$, which implies $v_{(t)} \in B$. In the second case, by minimality of t, since $\tau < t$, $w_{(\tau)} \in B$. Lemma 3.3.13 implies that $w_{(\tau)}$ is reachable from a square in $\Sigma^{\times}$ by a dependency path, therefore, $v_{(t)}$ is also reachable using dependency paths. Then $v_{(t)}$ must be in B, either because it is inconsistent or because there it has no matching operation. Similar arguments show that $v_{(t)} \in \Omega^{\oplus}$ implies $v_{(t)} \in B$. Therefore $v_{(t)}$ must be in $\Omega^{\otimes}$, i.e., $v_{(t)}$ reads a square s for which the algorithm calls the function `MarkReaders` (s, t') with a time $t' < t$. If $s \in \Sigma^{\otimes}$ then clearly $v_{(t)} \in B$; otherwise, there is another operation $v'_{(t')}$ that writes into s. Again, by minimality of $v_{(t)}$, $v'_{(t')} \in B$ and by Lemma 3.3.13 there exists a dependency path from $v'_{(t')}$ to $v_{(t)}$ which puts $v_{(t)}$ in B. Contradiction. ∎

When completed, `PropagateWS` updates the output to $\tilde{\mathsf{M}}$ and the computation graph to $\tilde{G}$ as if `StableWS` is run from-scratch with N' as input, computing M' and G'.

Lemma 3.4.2 (Isomorphism) *The output sets $\tilde{\mathsf{M}}$ and M' are equal and there exists an isomorphism $\phi : \tilde{G} \to G'$ that preserves the vertex and time of each operation.*

Proof: We prove equality of the output and build ϕ by induction on time. Define the following sets of operations: $\Omega^{\ominus}_{t} = \{v_{(\tau)} \in \Omega^{\ominus} \mid v_{(\tau)}$ is created at time $< t\}$ based on their creation times.

($\Omega_0^{\ominus}$ is the set of dispatch operations acting on input vertices). Also, define a similar assemblage for the $^{\oplus}$, $^{\otimes}$, and $'$ sets. Let $\tilde{G}_t$ be the subgraph of $\tilde{G}$ induced by the nodes $\tilde{\Omega}_t \cup \tilde{\Sigma}$ excluding the edges with time $\geq t$; the excluded edges are related to the execution of operations at time $\geq t$. Define G'_t similarly and let $\tilde{\mathsf{M}}_t$ be the updated set of vertices obtained by removing and inserting vertices until time t, just before the executing operations at time t.

Initially, $\tilde{\mathsf{M}}_0 = \mathsf{M}'_0 = \mathsf{N}'$ and $\tilde{\Sigma} = \Sigma'$. Therefore, there exists an isomorphism $\phi_0 : \tilde{G}_0 \to G'_0$. Assume the inductive hypothesis at time t, that $\tilde{\mathsf{M}}_t = \mathsf{M}'_t$ and that we have an isomorphism $\phi_t : \tilde{G}_t \to G'_t$. Pick $op \in \tilde{\Omega}_t$ with time t and let $op' = \phi_t(op)$. We aim to prove that op and op' execute in the same way. Because our functions are all deterministic, it suffices to show that op and op' read the same data. There are three cases: op is either in $\Omega_t^{\oplus}$, or in $\Omega_t^{\otimes}$, or otherwise op is an operation that has not been modified.

Assume that op is in $\Omega_t^{\oplus}$. We know $\tilde{\Sigma} = \Sigma'$, therefore, op and op' traverse the same quadtree structure in their execution. For a vertex v that op reads, v cannot be in $\mathsf{M}_t^{\ominus}$ because the vertices in $\mathsf{M}_t^{\ominus}$ are removed at time $< t$. Thus, op reads only the vertices in $\tilde{\mathsf{M}}_t = \mathsf{M}'_t$, in other words op reads the same data as op' does. The case that $op \in \Omega_t^{\otimes}$ is similar, because the re-execution of the inconsistent operations follow the same rules. In the remaining case, op is not modified. Consider a square s that op accesses. Because the update algorithm did not schedule op for re-execution, we know that s is not in Σ^{-}. Furthermore, for the same reason, s does not contain a vertex in $\mathsf{M}_t^{\ominus} \cup \mathsf{M}_t^{\oplus}$. Therefore, op only reads vertices in $\mathsf{M}'_t \cap \mathsf{M}_t$; op reads the same data as op' does. Hence, in all cases, op and op' execute similarly.

We have a natural correspondence between the operations that op and op' create and the Steiner vertices they insert (in any). Therefore, $\tilde{\mathsf{M}}_{t+1} = \mathsf{M}'_{t+1}$. Furthermore, because op and op' read and write the same squares the edges incident to these operations have natural correspondences as well. Extending ϕ_t to ϕ_{t+1} by adding these correspondences completes proof of the inductive step. ∎

Theorem 3.4.3 *The* `Insert` *and* `Delete` *functions modify the output in* $O(\log \Delta)$ *time and maintain a* ρ*-well-spaced output of optimal-size with respect to the updated input.*

Proof: By Lemma 3.4.2, we know that the output is the same as what would have been generated by executing from scratch `StableWS` with the new input, therefore, Theorem 3.3.4 applies. As discussed in Section 3.1.2, the quadtree can be updated in $O(\log \Delta)$ time. Also, Lemma 3.4.1 relates the runtime of the update algorithm to the distance between the executions with the old and new inputs. Finally, Lemma 3.3.16 bounds the runtime of `PropagateWS` as desired. ■

3.4.2 Lower Bound

We present a lower bound proving that any algorithm which explicitly maintains a well-spaced superset requires $\Omega(\log \Delta)$ time per dynamic update. Consider dynamically inserting a new point very close to an existing input vertex. Even the optimal dynamic algorithm is forced to insert geometrically growing rings of new Steiner vertices around the dynamically inserted vertex. We prove that we can iterate this process using a gadget. This shows that our algorithm is worst-case optimal compared to all other explicit algorithms, even in an amortized setting.

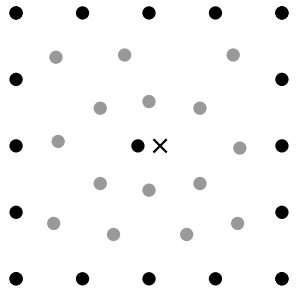


Figure 3.10: Inserting x creates $\Omega(\log \Delta)$ fresh Steiner vertices.

We define a gadget (see Figure 3.10) consisting of points in the hypercube $[0, k^{-1/d}]^d$. Consider two vertices at distance $1/\Delta$ from each other in the middle of the box; let one of them be the *dynamic vertex* x which will be inserted later. Also, consider a grid of $O(1)$ vertices on each of the faces of the hypercube, chosen according to the scheme of Hudson [43, p.79]. The input N consists of tiling $[0, 1]^d$ with the gadgets, $k^{1/d}$ for each dimension, without any dynamic vertex. The dynamic modification sequence consists of inserting k dynamic vertices, one for each gadget.

Lemma 3.4.4 *Inserting the dynamic vertex to a single gadget requires inserting* $\Omega(\log \Delta)$ *Steiner vertices.*

Proof: Let N be the input before adding the dynamic vertex x. Any size-optimal output M of N has $O(1)$ Steiner vertices inside the gadget box. Consider inserting x and let $\mathsf{N}' = \mathsf{N} \cup \{x\}$ and

$\delta = \mathrm{NN}_{\mathsf{N}'}(x)$. Draw the segment from x to the farthest point in $\mathrm{Vor}_{\mathsf{N}'}(x)$. This segment has length at least $\ell = \frac{1}{4} - \frac{\delta}{2}$. Consider the Voronoi diagram of a ρ-well-spaced superset M' of N' and consider the Voronoi cells that this segment cuts. Let $v_1, v_2, \ldots$ be the vertices of those Voronoi cells, in order. We know that the vertices in M' are ρ-well-spaced, therefore, $|v_1 x| \leq 2\rho\,\mathrm{NN}_{\mathsf{N}'}(x) = 2\rho\delta$. Also, the nearest neighbour distance of v_1 is at most $|v_1 x|$. We can use the same argument to get $|v_1 v_2| \leq 2\rho |v_1 x|$ and repeat. In other words, distance from x grows only geometrically as we walk down the segment: covering the distance ℓ requires $\Omega(\log 1/\delta) = \Omega(\log \Delta)$ many Steiner vertices. This implies that M differs from M' in at least $O(\log \Delta)$ vertices. ■

Theorem 3.4.5 (Lower Bound) *There exists an initial input and a set of n dynamic insertions that forces any algorithm to insert $\Omega(n \log \Delta)$ new Steiner vertices.*

Proof: In the above scheme, let $k = n$. Then, we would like to prove that inserting n dynamic vertices requires inserting $\Omega(n \log \Delta)$ Steiner vertices. We refer to a technique of inserting vertices to the hypercube faces [43]. It was developed precisely to make sure that certain algorithms need not add vertices outside the hypercube when making the interior ρ-well-spaced. Contrapositively, adding vertices outside a gadget does not help make the gadget, with its dynamic vertex, be ρ-well-spaced. Thus the prior lemma applies to each gadget individually, showing that the final ρ-well-spaced superset must contain at least $\Omega(n \log \Delta)$ Steiner vertices, for a carefully selected ρ. Since there exists a constant $\rho > 1$ such that the original input of n gadgets is ρ-well-spaced, the initial output must be of size $O(n)$. This completes our proof. ■

CHAPTER 4

KINETIC MESH REFINEMENT IN 2D

In this chapter, our main focus is the case in which points have *kinetics*: each point has an associated velocity function, and we maintain a quality mesh at all times. We develop our solution in the kinetic data structures framework, KDS in short [19, 34]. Previous work proposes KDSs for the related problem of kinetic triangulations, without inserting Steiner points. Regardless of insertion of Steiner points, maintaining a Delaunay triangulation has proven to be a difficult task: the most efficient kinetic Delaunay triangulation schemes [32, 37] are a linear factor less efficient than optimal. Recent research therefore considers maintaining triangulations that are not Delaunay. Indeed, in two dimensions, some results achieve efficiency [16, 48]. Furthermore, Agarwal et al. [14] show how to maintain the *stable Delaunay graph*, namely the subset of the Delaunay triangulation that has large minimum angles. In this work, by inserting Steiner points, we efficiently maintain the Delaunay triangulation, i.e., the full Delaunay graph, of the output point set, and ensure that all triangles have large minimum angles.

To see the challenges in providing an effective kinetic data structure for meshing, it helps to consider the techniques that work well for the static version of the problem. One approach, based on balanced quadtrees, generates an appropriately refined quadtree over the input points and adds the corners of the quadtree squares as Steiner points [20]. Because the quadtree is fixed, this approach can generate a large number of events. For example, if the input contains two close points that move along parallel linear trajectories preserving the distance, say ϵ, between them, then the quadtree may need to be restructured every time the points leave their quadtree cell, which is only $\Theta(\epsilon)$ distance. In other words, a quadtree approach cannot be *efficient*. Another approach computes Voronoi diagrams and inserts the corners of the Voronoi cells (equivalently, the Delaunay circumcenters) as Steiner points [44]. The main difficulty in kinetizing the algorithms that follow this approach is that the position function of a Steiner point depends on three points, some of which may themselves be Steiner points. The description length of the position function can thereby build

up to be polynomial in n. Since computing just one event time could take polynomial time, such a structure cannot be *responsive*.

In this chapter, we provide an effective kinetic data structure for computing meshes of a dynamically changing set of points that move along algebraic trajectories of constant degree. Our KDS yields triangulations of size-optimal well-spaced point sets (Section 4.4). We analyze the responsiveness, efficiency, locality, and compactness of our data structure as functions of the input size and the geometric spread (the ratio of diameter to closest pair). Since the spread changes as time evolves, we define Δ to be the ratio of the maximum diameter of the input at any time, to the minimum distance between closest pair of input points at any time. If the spread is polynomially bounded by the input size, our data structure yields bounds all in the problem size with logarithmic factors. Our KDS guarantees the following properties:

Responsiveness. A certificate failure requires $O(\log \Delta)$ update time (Theorem 4.3.7).

Locality. A point participates in $O(\log \Delta)$ certificates (Lemma 4.4.1).

Compactness. The total number of certificates is $O(m)$, where m is the output size and $m \in O(n \log \Delta)$ (Lemma 4.4.1).

Efficiency. The number of events is $O(n^2 \log^3 \Delta)$ which is within a $O(\log^2 \Delta)$ factor of the optimal (Lemma 4.4.4).

Dynamic updates. A point insertion or deletion requires $O(\log \Delta)$ update time (Theorem 4.3.7).

At a high level, our solution is in essence a balanced quadtree method, replacing the quadtree with a variant of the deformable spanners of Gao, Guibas, and Nguyen [33]. Regarding Steiner points, our solution hinges on a technique for determining their position and their motion plans (Section 4.1). Our KDS consists of a construction algorithm (Section 4.2) that computes a quality mesh of the input, and an update algorithm (Section 4.3) that enables kinetic motion simulation and dynamic changes. Given a set of input points, the construction algorithm first builds a *well-spaced* superset of the input, organizing the computation in $O(\log \Delta)$ levels. It then computes the

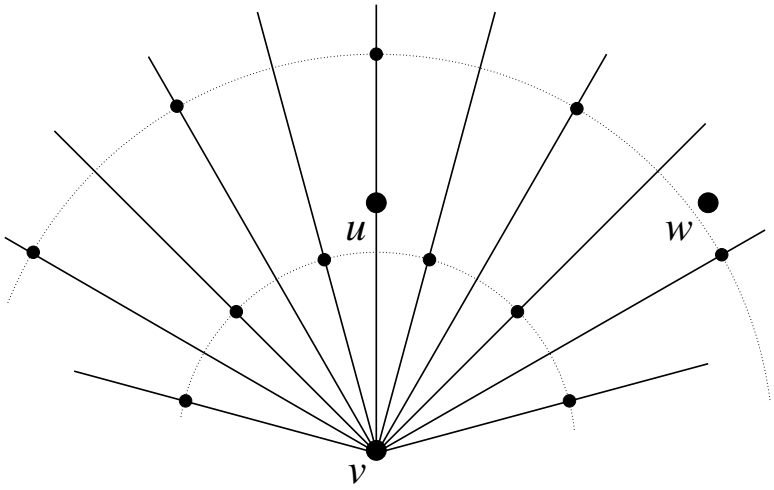


Figure 4.1: The satellites of an input point v in relation to the input points u and w. The first two orbits of v and some rays illustrates the definition of the location of v's satellites: intersections of odd rays with orbits at odd ranks and of even rays with orbits at even ranks form satellites shown in smaller dots.

Delaunay triangulation of the well-spaced set by performing local operations only, yielding a triangulation with no small angles. Upon a kinetic event or a dynamic modification to the input set, the update algorithm repairs the well-spaced superset and its Delaunay triangulation, by propagating the changes through the construction algorithm. Taking advantage of the organization in $O(\log \Delta)$ levels, our update algorithm repairs each level by performing amortized constant work per level.

4.1 Steiner Vertices and Spatial Data Structure

We present some definitions used throughout the chapter, describe the technique we use for selecting Steiner vertices, and present a brief overview of the deformable spanners.

4.1.1 Satellites

The key problem in mesh refinement is determining Steiner vertices, i.e., where should they be inserted and how should they move. To solve this problem, we propose a local template-based approach. This approach specifies for each input vertex v, a *nucleus*, an infinite number of *satellites*

that may be inserted as Steiner vertices. A nucleus and its satellites form a well-spaced point set and the satellites move together with their nucleus: the position functions of the satellites are the same as their nucleus' position function plus a fixed translation. To ensure a size-conforming output, we set the spacing between the satellites proportional to the distance to their nucleus, i.e., for each satellite, the distance to the nearest other satellite is within a constant multiplicative factor of the distance to their nucleus.

For the planar case, we use a particular template (illustrated in Figure 4.1) defined by a fixed set of rays emanating from a nucleus and their intersections with concentric circles of geometrically increasing radii. Consider 24 rays leaving each input vertex at angles $\theta, 2\theta, \ldots, 24\theta$, where $\theta = \pi/12$. Also, consider concentric circular orbits of radius 2^ℓ around every input vertex where $\ell \in \mathbb{Z}$ is the *rank* of these orbits. Defining odd (even) rays to be rays at angles that are odd (even) multiples of θ, we choose certain translations to define the satellites: the intersections of odd rays with orbits at odd ranks and the intersections of even rays with orbits at even ranks. In this template, a nucleus, a rank, and a ray with the same parity as that of the rank defines a unique satellite. Intuitively, this is a discrete polar coordinate system, where the nucleus defines the origin, the rank defines the radius (exponentially), and the ray defines the polar angle. In the rest of the chapter, we use the term ℓ-satellite to refer to a satellite on an orbit at rank ℓ.

4.1.2 Deformable Spanners

Our algorithm uses the kinetic deformable spanners of Gao, Guibas, and Nguyen as a point location data structure [33]. In our algorithms, in order to generate a quality mesh, we insert certain satellites into the deformable spanner data structure. Taking advantage of the location of the satellites, we achieve accuracy and efficiency by extending the deformable spanners with exact nearest neighbor queries and more efficient vertex insertion procedures. In this section, we briefly overview the original deformable spanners data structure; we explain the extension in the next section. In the most general form, given a parameter $\varepsilon > 0$, the deformable spanner $(1 + \varepsilon)$-approximates the

Euclidean distance between the vertices. Throughout the chapter, we use the spanner with $\varepsilon = 1$, guaranteeing 2-approximation. A spanner represents a hierarchical discretization of a vertex set at geometrically increasing scales. Given a vertex set M and any $s > 0$, a *discretization* of M at scale s is a subset $\mathsf{M}' \subseteq \mathsf{M}$ of vertices which satisfy the following two conditions:

- The minimum distance between any two vertices of M' is at least s.
- The set of balls $\{B(v, s) \mid v \in \mathsf{M}'\}$ cover M.

Note that there can be multiple discretizations of a vertex set. A spanner is based on a hierarchy of discretizations $\mathsf{M} = \mathsf{M}_\lambda \supseteq \mathsf{M}_{\lambda+1} \supseteq \cdots \supseteq \mathsf{M}_\Lambda$ that satisfy the following properties:

- λ is the minimum integer such that the condition $2^{\lambda-1} \leq \delta < 2^\lambda$ is satisfied, where δ is the closest pair distance in M.
- For each $\ell \in \{\lambda + 1, \lambda + 2, \ldots, \Lambda\}$, M_ℓ is a discretization of $\mathsf{M}_{\ell-1}$ at scale 2^ℓ.
- M_Λ is the only singleton in the hierarchy.

We call M_ℓ a discretization at *rank* ℓ. Note that this definition coincides with the rank definition that is used to describe the satellites—both definitions provide a logarithmic scale with base 2. One could use different bases; for simplicity, we choose base 2 for both. We refer to the vertices of M_ℓ as *discrete centers* at rank ℓ, or shortly as ℓ*-centers*. A spanner connects discrete centers at the same and at the consecutive ranks with neighbor, parent, and child pointers. Specifically, two ℓ-centers are ℓ*-neighbors* if the distance between them is at most $c \cdot 2^\ell$, where $c = 4 + 16/\varepsilon = 20$. If v is both an ℓ-center and an $(\ell + 1)$-center, v is its own parent/child. Otherwise, an $(\ell + 1)$-center w whose distance to v is at most $2^{\ell+1}$ is designated to be the *parent* of v; and v to be a child of w at rank ℓ. For a vertex v, we define its *maximum rank*, written Λ_v, to be the highest rank where v is a discrete center. We define its *minimum rank*, written λ_v, to be the lowest rank ℓ where v is a discrete center with at least one ℓ-neighbor. Now, we briefly explain the construction of the

spanner using a somewhat different presentation than the one used by Gao et al. [33]. Also, we describe their certificates and summarize their update algorithm.

Construction. We start by assigning an arbitrary input vertex v to be the root of the spanner S and set the maximum rank Λ of S to $\max_{w \in \mathsf{N}} \lceil \lg |vw| \rceil$, i.e., we set $\mathsf{S}_\Lambda = \{v\}$. Furthermore, we temporarily set v to be the parent of every other input vertex. In a top-down pass, at each rank ℓ, we greedily determine the set of ℓ-centers (S_ℓ) as follows: initializing $\mathsf{S}_\ell = \mathsf{S}_{\ell+1}$ and performing a linear pass on the set of remaining vertices, $\mathsf{N} \setminus \mathsf{S}_{\ell+1}$, we insert a vertex w into S_ℓ if the ball $B(w, 2^\ell)$ does not contain an ℓ-center. This can be done by checking the cousins (parent's neighbors' children) of w. Otherwise, if w is 2^ℓ-close to an ℓ-center v, we update the parent of w to be v. We also insert neighbor edges between any two ℓ-centers within $c \cdot 2^\ell$ distance, again using cousins. We stop at rank $\ell = \lambda$, when $\mathsf{S}_\ell = \mathsf{N}$.

Certificates. To certify the spanner, Gao et al. describe four kinds of certificates:

- *Parent-child* certificates for certifying that $|vu| \leq 2^{\ell+1}$ for an ℓ-center v and its parent u at rank $\ell + 1$.
- *Separation* certificates for certifying that $|vu| > 2^\ell$ for ℓ-neighbors v and u.
- *Edge* certificates for certifying that $|vu| \leq c \cdot 2^\ell$ for ℓ-neighbors v and u.
- *Potential-neighbor* certificates for certifying that $|vu| > c \cdot 2^\ell$ for two ℓ-centers v and u, whose parents are $(\ell + 1)$-neighbors.

Maintenance. When some certificates fail, we update the discretizations in a top-down pass. Assuming that the spanner at ranks above ℓ is updated, we fix the spanner at rank ℓ. First, we check if any of the parent-child certificates at rank ℓ has failed. If an ℓ-center v is no longer a child of an $(\ell + 1)$-center w, we check to see if there is another $(\ell + 1)$-center, an $(\ell + 1)$-neighbor of w, that is $2^{\ell+1}$-close to v. If there is, we assign that vertex as the new parent of v, if not we *promote* v to rank $\ell + 1$ and repeat the promotion procedure until we assign a new parent to v. In promoting to a higher rank $\ell' > \ell + 1$, we use the rank ℓ' ancestor of v for locating v, e.g., parent of w for

```
Construct a spanner S for N

∀v ∈ N, activation rank ρ_v ← λ_v
for ℓ = λ to Λ + 4
  ∀v ∈ N, v is active if
    ρ_v = ℓ or ∃ a converted (ℓ − 1)-satellite of v
  for each ℓ-satellite s of each active v ∈ N
    if BallEmpty(S, s, 2^{ℓ−2}) then Convert(S, s, ℓ)
  for each Steiner vertex s ∈ S_ℓ
    TryToPromote(S_{ℓ+1}, s)

for each rank ℓ
  for each s ∈ N with ρ_s = ℓ or ℓ-satellite s ∈ M \ N
    V ← ∪_{ℓ'∈{ℓ−6,...,ℓ−3}} ℓ'-neighbors of s
    IdentifyVoronoiNeighbors(s, V)
```

Figure 4.2: The pseudo-code for the construction

$\ell' = \ell + 2$. Next, if a separation certificate has failed, i.e., an ℓ-center v becomes a child of an $(\ell + 1)$-center, we remove v from ranks $\ell + 1$ and above. This leaves the children of v at ranks $\ell + 1$ and above without a parent. We promote these children as necessary, by applying the above procedure. In repairing both types of certificate failures, we insert and remove neighbor edges in order to maintain a valid spanner structure. In addition, we update the neighbor edges, if any of the edge or potential-neighbor certificates has failed. During maintenance, besides updating the spanner, we update the set of certificates as necessary in order to certify the updated spanner.

4.2 Construction Algorithm

Our construction algorithm builds a mesh in three stages (Figure 4.2). First, it constructs a spanner for the input vertices by running the construction algorithm described in Section 4.1.2; second, it constructs a well-spaced superset M of the input by inserting certain satellites into the spanner; third, it constructs the Delaunay triangulation of M.

In the second stage, our algorithm iterates over ranks bottom-up and determines a set of *active* input vertices and applies a *conversion* and a *promotion* process at each rank. In the conversion

process, it selects certain satellites of active input vertices and inserts them into the spanner as Steiner vertices. In the promotion process, it inserts certain Steiner vertices into the discretization at the next rank in order to represent the current superset correctly at the next rank.

As part of determining the active status of input vertices, our algorithm starts the second stage by assigning each input vertex an *activation rank*, which is defined as the minimum rank in the initial spanner. At each rank ℓ, our algorithm then determines the active input vertices—an input vertex is *active* at rank ℓ if its activation rank is ℓ or it was active at rank $\ell-1$ and one of its $(\ell-1)$-satellites was converted to a Steiner vertex. It continues by applying the conversion process on the ℓ-satellites of active input vertices. Specifically, iterating through each ℓ-satellite s of each active vertex, our algorithm converts s if and only if s has an empty *certificate ball*, which is defined as the ball $B(s, 2^{\ell-2})$. When converting an ℓ-satellite s, our algorithm inserts s into the spanner hierarchy beginning at S_{λ_s} and promotes s to the highest possible rank up to rank ℓ. Once the algorithm is done with the conversion process, it updates the discretization at the next rank ($\mathsf{S}_{\ell+1}$) and prepares the spanner for the next iteration by promoting certain ℓ-centers. In this promotion process, by calling `TryToPromote` (details below) for each Steiner vertex $s \in \mathsf{S}_\ell$, the algorithm determines whether there is an $(\ell+1)$-center $2^{\ell+1}$ close to s. If there is one, the algorithm assigns it to be the parent of s. Otherwise, the algorithm inserts s into $\mathsf{S}_{\ell+1}$, i.e., promotes s to rank $\ell+1$, and determines the $(\ell+1)$-neighbors of s.

In the third stage, the algorithm identifies the Voronoi neighbors of the vertices. For each Steiner vertex s at rank ℓ and each input vertex with activation rank ℓ, it first generates a candidate set consisting of the spanner neighbors of s at ranks $\{\ell-6, \ldots, \ell-3\}$. It then computes the Voronoi cell of s in this candidate set in $O(1)$ time; this computation yields the true Voronoi cell of s in the well-spaced superset M (Lemma 4.2.9). Then, the complete Voronoi diagram yields the Delaunay triangulation of M.

4.2.1 Analysis

In the rest of this section, we prove the correctness and the $O(n \log \Delta)$ runtime bound of our algorithm. For realizing the run-time bound, we focus on efficiently locating any ℓ-satellite s in the spanner using its nucleus v. We extend the spanner data structure with $O(1)$ time functions, `BallEmpty`, `Convert`, and `TryToPromote`, details of which we describe in the next two paragraphs. Taking advantage of these $O(1)$ time functions, we bound the total runtime converting satellites to Steiner vertices by $O(n \log \Delta)$ as there are $O(n \log \Delta)$ many satellites to consider. Then, we bound the runtime of the discretization step by $O(n \log \Delta)$, by proving that there are $O(n)$ ℓ-centers in the spanner at each rank ℓ (Lemma 4.2.11) and $O(\log \Delta)$ ranks. In the third stage, since the computation of the Voronoi cell of each vertex in M takes $O(1)$ time, we achieve the desired runtime bound.

In order to efficiently locate an ℓ-satellite s of an active vertex v, we define the notion of ℓ-sites. An *ℓ-site* of s is an input vertex $w \in \mathsf{S}_\ell$ (at rank ℓ) that satisfies the condition $|sw| \leq c \cdot 2^{\ell-2}$. By Lemma 4.2.1, the rank ℓ ancestor of v, say w, is an ℓ-site of s. A naive approach locates w in $O(\ell - \rho_v)$ time by walking up the parent chain of v. If the activation rank of v is ℓ, this is efficient, however, it might be as costly as $O(\log \Delta)$ if ℓ is significantly higher than the activation rank ρ_v. In this case, the rank ℓ ancestor, say w', of the last converted satellite u of v can be located in $O(1)$ time (Lemma 4.2.10) and by modifying the arguments of Lemma 4.2.1 slightly, one can prove that w' is $c \cdot 2^{\ell-2}$ close to s. Once locating an ℓ-center (w or w') that is $c \cdot 2^{\ell-2}$ close to s, all ℓ-sites of s can be located in $O(1)$ time by checking its ℓ-neighbors.

We describe the $O(1)$ time implementations of `BallEmpty`, `Convert`, and `TryToPromote` using ℓ-sites. `BallEmpty` first finds all ℓ-sites of s, then computes the ℓ-neighbors of s in $O(1)$ time by checking the ℓ-neighbors of the cousins of an ℓ-site of s (Lemma 4.2.2). It then computes the ℓ'-neighbors of s for ranks $\ell' = \ell - 1$ down to $\ell' = \ell - 6$ by checking the ℓ'-neighbors of the children of the $(\ell' + 1)$-neighbors of s. It determines whether the certificate ball of s is empty or not by checking whether there is a neighbor of s within $2^{\ell-2}$ distance of s or not (Lemma 4.2.4).

If there are no neighbors, the algorithm converts s by calling `Convert`; otherwise, it discards s. `Convert` inserts s into the spanner at its minimum rank λ_s and promotes it up to rank ℓ. At the end of each rank, `TryToPromote` promotes certain Steiner vertices to the next rank. For a given ℓ-center s, it uses an $(\ell+1)$-site u of s, the parent of an ℓ-site of s to be precise, to determine if an $(\ell+1)$-center is close enough to be the parent of s. It checks all $(\ell+1)$-neighbors of u; if there is an $(\ell+1)$-center within $2^{\ell+1}$ distance of s, it assigns that vertex to be the parent of s. If no such $(\ell+1)$-center exists, it includes s into $\mathsf{S}_{\ell+1}$ as an $(\ell+1)$-center. Also, it locates all $(\ell+1)$-neighbors of s by checking $(\ell+1)$-neighbors of cousins of u.

Lemma 4.2.1 *For any given $\ell' \geq \ell$, the rank ℓ' ancestor of the nucleus of an ℓ-satellite s is an ℓ'-site of s.*

Proof: Let v be the nucleus of s and w be the rank ℓ' ancestor of v. Since $|sv| = 2^{\ell}$ and $|vw| \leq 2^{\ell'+1}$, by the triangle inequality, we have $|sw| < 2^{\ell'+2} < c \cdot 2^{\ell'-2}$. ∎

Lemma 4.2.2 *Given an ℓ-satellite s, for some rank $\ell' \geq \ell$, let u be an ℓ'-site and w be an ℓ'-neighbor of s. Then, w is either a cousin of u (u's parent's neighbors' child) or an ℓ'-neighbor of one of the cousins of u.*

Proof: Let v be the nucleus of s, u' be the parent of u and therefore an $(\ell'+1)$-site of s. If w is an input vertex, let w' be its parent, otherwise, let w' be one of its $(\ell'+1)$-sites. In both cases, $|ww'| \leq c \cdot 2^{\ell'-1}$. Our claim is proven if we can show that u' and w' are $(\ell'+1)$-neighbors. Since w is an ℓ'-neighbor of s, $|sw| \leq c \cdot 2^{\ell'}$, and since u' is an $(\ell'+1)$-site of s, $|su'| \leq c \cdot 2^{\ell'-1}$. Using the triangle inequality, we have $|u'w'| \leq c \cdot 2^{\ell'+1}$, that is, u' and w' are $(\ell'+1)$-neighbors. ∎

The correctness proof of the construction algorithm relies on some technical lemmas. In our main lemma, we prove that our algorithm progresses towards a well-spaced superset incrementally (Lemma 4.2.3). In order to prove this lemma, we show that the bottom-up processing order over the ranks ensure that querying only certain neighbors of a satellite is enough to determine whether the certificate ball of that satellite is empty or not (Lemma 4.2.4). Furthermore, converting, again

in a bottom-up order, only the satellites with empty certificate balls guarantees that their certificate balls remain empty—we never convert a satellite that lies in the certificate ball of a converted satellite (Lemma 4.2.6). After proving well-spacedness, we prove that the output is size-optimal as well (Lemma 4.2.8).

Lemma 4.2.3 *At the end of rank ℓ, all input vertices with activation ranks $\leq \ell$ and all satellites converted at ranks $< \ell$ are $\frac{9}{2}$-well-spaced.*

For the base case, at the beginning of rank λ, there are no active input vertices and no converted satellites, therefore the claim is trivially true. For the inductive hypothesis, we assume that at the end of rank $\ell-1$, all input vertices with activation ranks $\ell-1$ and below and all satellites converted at ranks $\ell-2$ and below are $\frac{9}{2}$-well-spaced. We break down the inductive proof into several steps and conclude it later.

Lemma 4.2.4 *The certificate ball of an ℓ-satellite s is empty of input vertices and earlier converted satellites iff it does not contain an ℓ'-neighbor of s at ranks $\ell' \in \{\ell-6, \ldots, \ell-3\}$.*

Proof: The only if part of the proof is trivial; if the certificate ball of s contains a neighbor, clearly, it is not empty. For the if part, for any rank $\ell' \geq \ell-2$ and any ℓ'-neighbor u of s, $|su| > 2^{\ell-2}$ because u and s are both ℓ'-centers. Therefore u cannot be inside the certificate ball of s. For the case that s has an ℓ'-neighbor u for some rank $\ell' \leq \ell-7$, we have $|su| \leq c \cdot 2^{\ell'}$. Using the triangle inequality, the rank $\ell-6$ ancestor, say u', of u satisfies $|su'| \leq c \cdot 2^{\ell'} + |uu'| < c \cdot 2^{\ell-7} + 2^{\ell-5} < 2^{\ell-2} < c \cdot 2^{\ell-6}$. Therefore u' being an $(\ell-6)$-neighbor of s would be inside the certificate ball of s. Now, given the premise of the lemma, assume towards a contradiction that a vertex not neighboring s lies inside the certificate ball and let v be the nearest one.

Since the spanner has a 2-approximation guarantee, s has a neighbor within distance $2|sv| \leq 2^{\ell-1} < c \cdot 2^{\ell-5}$, thus, s has an $(\ell-5)$-neighbor. Since v is not a neighbor of s, v cannot be among the children of $(\ell-5)$-neighbors of s. More specifically, v is not an $(\ell-6)$-center; if it were, its parent would have been an $(\ell-5)$-neighbor of s. Therefore, since v is not an $(\ell-6)$-center, we have

$\mathrm{NN}(v) < 2^{\ell-6}$. Hence, for some $\ell' < \ell$, v is either an ℓ'-satellite or an input vertex with activation rank ℓ'. By the inductive hypothesis, v is $\frac{9}{2}$-well-spaced. Since v is the exact nearest neighbor of s, s lies in the Voronoi cell of v. Therefore, we have $|sv| \leq \frac{9}{2}\,\mathrm{NN}(v) < \frac{9}{2} \cdot 2^{\ell-6} < 2^{\ell-3}$. Let w be the rank $\ell - 5$ ancestor of v, then $|vw| < 2^{\ell-4}$, and using the triangle inequality, we get $|sw| < 2^{\ell-2} < c \cdot 2^{\ell-5}$. Being that close, w must be an $(\ell - 5)$-neighbor of s. This is a contradiction to our premise because w is in the certificate ball of s. ■

In order to better explain our construction algorithm, we relate the problem of determining which satellites to be converted to the maximal independent subset problem. At a given rank ℓ, before converting any satellites, consider the set of ℓ-satellites whose certificate balls are empty. Let the *proximity graph* at rank ℓ be the undirected graph on this set with edges connecting two ℓ-satellites if and only if they are within distance $2^{\ell-2}$ from each other. Then we prove:

Lemma 4.2.5 *At each rank ℓ, the algorithm converts a maximal independent subset of the proximity graph.*

Proof: First, we prove independence. Pick any satellite s that is converted at rank ℓ. By Lemma 4.2.4, since we never convert a satellite before ensuring that its certificate ball is empty, s cannot be $2^{\ell-2}$ close to any existing vertex prior to its conversion. Similarly, the algorithm cannot convert an ℓ-satellite that is $2^{\ell-2}$ close to s. Thus, in the proximity graph, none of the satellites adjacent to s are converted. For maximality, consider an ℓ-satellite s, none of whose neighbors in the proximity graph is converted. This implies that s has an empty certificate ball, therefore s would be converted when the algorithm tries to insert s. ■

Using the above lemma, we state a corollary that is useful in our analysis: the certificate ball of each ℓ-satellite s of each active input vertex contains a vertex, either the converted vertex s or a vertex that prohibits the conversion of s.

Lemma 4.2.6 *For any rank $\ell' \leq \ell$, there is an empty ball of radius $2^{\ell'-2}$ around any converted ℓ'-satellite and around any input vertex with activation rank ℓ'.*

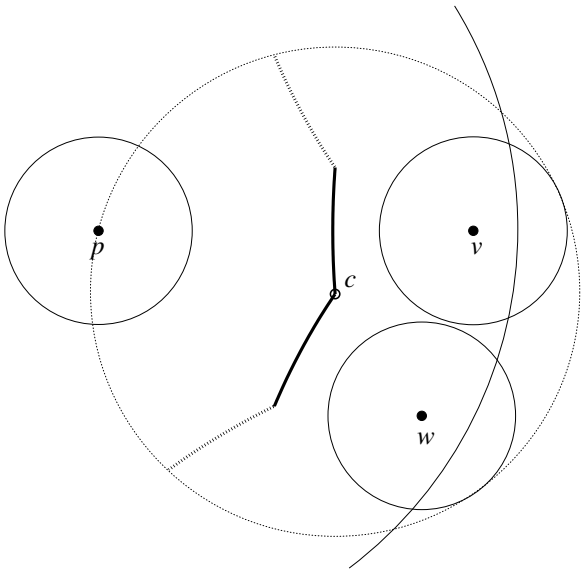


Figure 4.3: The points v and w are ℓ-satellites of an input vertex p. Each of the hyperbolic thick curves depicts the locus of the points whose distance to an ℓ-satellite is $2^{\ell-2}$ less than its distance to p. The Voronoi cell of p is a subset of the weighted Voronoi cell of p defined by these hyperbolic curves.

Proof: First, consider a converted ℓ'-satellite s. Using Lemma 4.2.4, we know that at rank ℓ', the certificate ball of s (of radius $2^{\ell'-2}$) is empty. Subsequent conversions require empty certificate balls of radius at least $2^{\ell'-2}$, thus, the certificate ball of s remains empty. Now, consider an input vertex v with activation rank ℓ'. We know that all other input vertices are at least $2^{\ell'}$ far away. Therefore, by the triangle inequality, for $k < \ell'$, all k-satellites of other input vertices are at least $2^{\ell'} - 2^k \geq 2^{\ell'-1}$ far away from v. Moreover, for $k \geq \ell'$, all converted k-satellites have empty balls of radius $2^{k-2} \geq 2^{\ell'-2}$, hence, our result follows. ■

We prove the inductive step of Lemma 4.2.3, that all input vertices with activation rank ℓ and Steiner vertices converted at rank $\ell - 1$ are $\frac{9}{2}$-well-spaced. Figure 4.3 displays an input vertex p with activation rank ℓ. By the corollary to Lemma 4.2.5, there is a vertex inside the certificate ball of every ℓ-satellite of p. We know that the vertex q inside the certificate ball of a given ℓ-satellite, say v, is within $2^{\ell-2}$ distance of v. Considering the locus of the points whose distance to v is $2^{\ell-2}$ less than its distance to p, the collection of these hyperbolas defines a weighted Voronoi cell of p. Observe that none of the points that lies in the half that contains v can be in the Voronoi cell of p. Consequently, the weighted Voronoi cell bounds the Voronoi cell of p. The extreme points of this region are the intersections of two hyperbolas, which correspond to the circumcenters of the circles that are tangent to p and to two certificate balls on the outside. One can show that

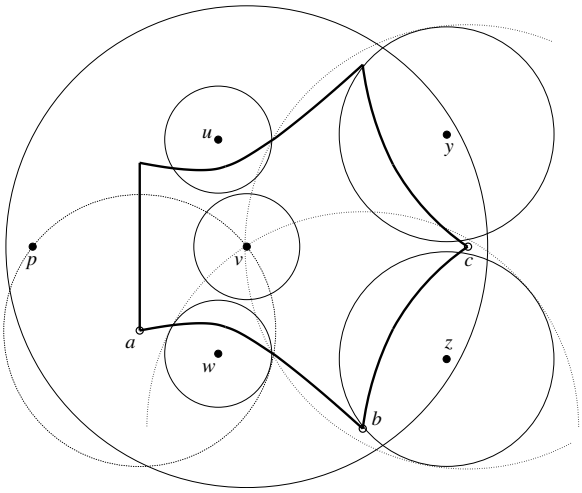


Figure 4.4: Illustration of the $\frac{9}{2}$-well-spacedness of a converted $(\ell-1)$-satellite v of an input vertex p. The weighted Voronoi cell defined by the thick hyperbolic curves bound the Voronoi cell of v.

these circumcenters (e.g., c in Figure 4.3) lie within $9 \cdot 2^{\ell-3}$ distance of p. Then, the proof of $\frac{9}{2}$-well-spacedness follows from Lemma 4.2.6, that the nearest neighbor of p is at least at $2^{\ell-2}$ distance.

In order to prove the $\frac{9}{2}$-well-spacedness of the Steiner vertices, we illustrate an example in Figure 4.4 displaying an input vertex p, its $(\ell-1)$-satellites u, v, w, its ℓ-satellites y and z, and their certificate balls. Assuming that v is converted to a Steiner vertex, we prove that its Voronoi cell lies within the big ball $B(v, 9 \cdot 2^{\ell-4})$ displayed in the figure. Intuitively the proof considers the output vertices in each of the certificate balls of u, w, y, and z. We prove that p and these four vertices bound the Voronoi cell of v as desired. Once again, we consider the weighted Voronoi cell of v, which upper bounds the actual Voronoi cell of v, where each of the satellites u, w, y, and z has weight equal to the radius of its certificate ball. Then, we prove that the corners of the weighted Voronoi cell, or the circumcenters, e.g., a, b, c, lie within $9 \cdot 2^{\ell-4}$ distance of v. Again Lemma 4.2.6 states that the nearest neighbor of v is at least at $2^{\ell-3}$ distance; this implies the $\frac{9}{2}$-well-spacedness of v.

Lemma 4.2.7 *All Steiner vertices converted at the final rank are $\frac{9}{2}$-well-spaced.*

Proof: Given an $(\Lambda+4)$-satellite s of an input vertex v, we know $|sv| = 2^{\Lambda+4}$. Since diameter D is less than $2^{\Lambda+2}$, only one of the satellites on the same ray as that of s can be inserted. We prove

that for each of the 12 rays, the $(\Lambda + 4)$-satellite of exactly one of the active vertices is converted to a Steiner vertex: any satellite s' that is converted to a Steiner vertex at rank $\Lambda + 3$ is at distance $2^{\Lambda+3}$ away from its nucleus v' and by the triangle inequality, $|ss'| \geq |sv| - (|s'v'| + |v'v|) \geq 2^{\Lambda+4} - (2^{\Lambda+3} + D) > 2^{\Lambda+2}$; the certificate ball of s is empty of output vertices. Then arguments similar to those in Lemma 4.2.3 proves well-spacedness. ∎

Lemma 4.2.8 M *is size-conforming with respect to* N.

Proof: For any $v \in \mathsf{M}$, we show $\mathrm{NN}_{\mathsf{M}}(v) \in \Omega(\mathrm{lfs}(v))$. First, we analyze the vertices by ranks and upper bound their local feature sizes. If v is an input vertex with activation rank ℓ, we have $\mathrm{lfs}(v) = \mathrm{NN}_{\mathsf{N}}(v) \in O(2^{\ell})$. If v is an ℓ-satellite with nucleus u, by the Lipschitz condition, we have $\mathrm{lfs}(v) \leq \mathrm{lfs}(u) + |uv|$. Since the activation rank of u is at most ℓ, we have $\mathrm{lfs}(u) \in O(2^{\ell})$, and since $|uv| = 2^{\ell}$, we deduce that $\mathrm{lfs}(v) \in O(2^{\ell})$. In both cases, by Lemma 4.2.6, we know that $\mathrm{NN}_{\mathsf{M}}(v) \geq 2^{\ell-2}$, therefore, our result follows. ∎

After the algorithm constructs a $\frac{9}{2}$-well-spaced size-optimal superset M of the input N, it constructs the Delaunay triangulation of M. Let $s \in \mathsf{M}$ be either an input vertex with activation rank ℓ or a Steiner vertex converted at rank ℓ. Using Lemma 4.2.9 below, the algorithm generates a candidate set by iterating through each neighbor of s at ranks $\{\ell - 6, \ldots, \ell - 3\}$ and checking if it lies within $9 \cdot 2^{\ell-2}$ distance of s. With this candidate set (of constant size), the algorithm applies halfspace tests to determine exactly which of these candidates are indeed Voronoi neighbors of s.

Lemma 4.2.9 *Let s be a converted ℓ-satellite or an input vertex with activation rank ℓ. If v is a Voronoi neighbor of s in M, v and s are ℓ'-neighbors at a rank $\ell' \in \{\ell - 6, \ldots, \ell - 3\}$.*

Proof: By Lemma 4.2.6, we know that there is an empty ball of radius $2^{\ell-2}$ around s. Since s is $\frac{9}{2}$-well-spaced, the Voronoi cell of s must be confined in a ball of radius $\frac{9}{2} \cdot 2^{\ell-2}$, which implies $|vs| \leq 9 \cdot 2^{\ell-2}$. Because of the 2-approximation guarantee, both v and s have neighbors in the spanner within $18 \cdot 2^{\ell-2}$ distance. Thus, the minimum ranks of v and s are less than $\ell - 1$. Because of the empty ball, s is an $(\ell - 2)$-center and a discrete center at a lower rank ℓ' as long as s has

an ℓ'-neighbor. Since $9 \cdot 2^{\ell-2} \geq |vs| > 2^{\ell-2}$, there is some $k \in \{\ell-6, \ldots, \ell-3\}$ such that $c \cdot 2^k \geq |vs| > c \cdot 2^{k-1}$. Then the Voronoi cell of v includes a point at least $c \cdot 2^{k-2}$ far from v. Since v is $\frac{9}{2}$-well-spaced, there is an empty ball of radius at least $\frac{2}{9} \cdot c \cdot 2^{k-2} > 2^k$ around v. Therefore, v is a k-center and consequently a k-neighbor of s. ∎

Lemma 4.2.10 *Consider a converted ℓ-satellite s. The minimum rank of s, λ_s, satisfies the condition $\ell - 6 \leq \lambda_s \leq \ell - 3$.*

Proof: By definition of λ_s, s has a λ_s-neighbor u within distance $c \cdot 2^{\lambda_s}$. Since s is converted to a Steiner vertex, its certificate ball must be empty. Thus, $2^{\ell-2} < |su| < c \cdot 2^{\lambda_s}$, that is, $2^\ell/4c < 2^{\lambda_s}$. This implies $\ell - 7 < \lambda_s$. For the upper bound, observe that the nucleus of s is 2^ℓ away from s. Again, by definition of λ_s, we know that any λ_s-neighbor of s is at least $c \cdot 2^{\lambda_s - 1}$ away from s. Gao et al. proves that the nearest of those λ_s-neighbors provides a 2-approximation for the exact nearest neighbor of s [33]. Since the nucleus of s is at 2^ℓ distance, we have $c \cdot 2^{\lambda_s - 1} \leq 2 \cdot 2^\ell$, that is, $2^{\lambda_s} \leq \frac{4}{c} \cdot 2^\ell$. We conclude, $\lambda_s < \ell - 2$. ∎

Lemma 4.2.11 *For any rank ℓ, the number of ℓ-centers in the spanner is bounded by $O(n)$, i.e., $|\mathsf{S}_\ell| \in O(n)$.*

Proof: By Lemma 4.2.10, if an ℓ'-satellite s of an input vertex v is converted to a Steiner vertex, its minimum rank in the spanner is at least $\ell' - O(1)$. In particular, for s to be an ℓ-center, ℓ' must be bounded by $\ell + O(1)$, in other words, $|sv|$ must be bounded by $O(2^\ell)$. Since two ℓ-centers are at least 2^ℓ far apart, a packing argument shows that there can be at most $O(1)$ satellites of v that are ℓ-centers. The proof follows from the fact that there are n input vertices. ∎

Theorem 4.2.12 *Our algorithm constructs a $\frac{9}{2}$-well-spaced, size-optimal superset M of the input N, and computes the Delaunay triangulation of M in $O(n \log \Delta)$ time.*

Proof: The quality proof follows from the Lemmas 4.2.3 and 4.2.7. Lemma 4.2.8 shows that the output is size-conforming, which is sufficient to prove size optimality [60]. Finally, Lemma 4.2.9

proves that the algorithm correctly computes the Voronoi diagram, i.e., the Delaunay triangulation of the superset M. Runtime proof follows from previous discussions based on Lemmas 4.2.10 and 4.2.11. ■

4.3 Dynamic and Kinetic Maintenance

We present an update algorithm that, given a dynamic modification or a kinetic event, updates the set of Steiner vertices and the corresponding Delaunay triangulation so that the output remains to be a quality, size-optimal triangulation. The update algorithm is a change-propagation algorithm: it maintains a set of affected satellites for each rank and repairs each rank consecutively. The algorithm therefore is trivially $O(\log \Delta)$-stable at the level of abstraction of the ranks, i.e., it has to repair only $O(\log \Delta)$ ranks to update the output. To bound the total update runtime, we show that there are $O(1)$ affected satellites at each rank (Lemma 4.3.5) and that repairing each rank requires amortized $O(1)$ time, yielding our final bound on responsiveness (Theorem 4.3.7).

As required by the KDS framework, we certify both the spanner structure and our extension of it. For the spanner, we use the same set of certificates (parent-child, edge, separation, and potential neighbor) used by Gao et al. [33]. For the extension, we generate our own certificates. We certify the conversion decisions for each ℓ-satellite s considered.

- If the certificate ball of s is not empty, we generate a *ball-not-empty* certificate to certify that a vertex $v \in \mathsf{M}$ lies inside $B(s, 2^{\ell-2})$.
- Otherwise, we generate a *ball-empty* certificate to certify that $v \notin B(s, 2^{\ell-2})$ for each ℓ'-neighbor v of s for $\ell' < \ell$.

For certifying the Delaunay triangulation, we observe by Lemma 4.2.9 that, for each converted ℓ-satellite s, neighbors of s at ranks $\{\ell - 6, \ldots, \ell - 3\}$ constitute its potential Voronoi neighbors. We certify the halfspace tests performed for determining the Voronoi cell of s considering this candidate set. These discussions allow us to state:

```
Update the spanner S restricted to N
for each rank ℓ, F_ℓ = P_ℓ = ∅
for each affected input vertex v
  ρ'_v ← ρ_v, ρ_v ← λ_v (in S restricted to N)
  if ρ_v ≠ ρ'_v then
    for each rank ℓ between ρ_v and ρ'_v (inclusive)
      F_ℓ ← F_ℓ ∪ {ℓ-satellites of v}
for each violated ball-empty/ball-not-empty certificate
  F_ℓ ← F_ℓ ∪ {s}, where s is the ℓ-satellite involved
for each affected Steiner vertex s
  P_ℓ ← P_ℓ ∪ {s}, where ℓ is the rank of affection

for each rank ℓ with F_ℓ ∪ P_ℓ ≠ ∅
  for each s ∈ F_ℓ ∩ S Remove(S, s)
  for each s ∈ F_ℓ
    v ← nucleus of s
    if (ρ_v = ℓ or ∃ a converted (ℓ−1)-satellite of v)
      and BallEmpty(S, s, 2^{ℓ−2}) then Convert(S, s, ℓ)
  for each nucleus v of each satellite s ∈ F_ℓ
    if an ℓ− or (ℓ+1)-satellite of v is converted then
      F_{ℓ+1} ← F_{ℓ+1} ∪ {(ℓ+1)-satellites of v}
  S_{ℓ+1} ← UpdatePromotion(F_ℓ ∪ P_ℓ)

for each rank ℓ and each affected s ∈ N with ρ_s = ℓ
                or each ℓ-satellite s ∈ M ∩ (F_ℓ ∪ P_ℓ)
  V ← ∪_{ℓ'∈{ℓ−6,...,ℓ−3}} ℓ'-neighbors of s
  IdentifyVoronoiNeighbors(s, V)
```

Figure 4.5: Psuedo-code for the update algorithm.

Lemma 4.3.1 *The certificates generated by our construction algorithm certifies that the output is the Delaunay triangulation of a size-optimal and well-spaced superset of the given input set.*

Upon a certificate failure or a dynamic modification, the update algorithm whose pseudo-code is shown in Figure 4.5 updates the output and the set of certificates. Following the structure of the construction algorithm, in three stages, it repairs the proximity graph and updates the maximal independent subset of satellites chosen at each rank. It keeps two sets of vertices at each rank for affected satellites. The first set, $\mathcal{F}$, tracks the satellites that may be required to be removed or converted to Steiner vertices, we call them *fully affected*. The second set, $\mathcal{P}$, tracks the satellites which are previously converted and whose ball-empty certificates remain unaffected. These satel-

lites, which we call *partially affected*, are not required to be removed; however, we may need to promote or demote these vertices in the spanner.

In the first stage, the update algorithm updates the spanner data structure restricted to the input vertices using the update procedure described in Section 4.1.2. It updates the diameter and the activation ranks of the input vertices if needed. Each of these updates may partially/fully affect certain satellites. The algorithm then initializes $\mathcal{F}_\ell$ and $\mathcal{P}_\ell$ lists. For each input vertex whose activation rank has changed, it marks all satellites at ranks between the previous and current activation ranks as fully affected as these satellites may need to be removed or considered for conversion. Also, for Steiner vertices, if any of the ball-empty or ball-not-empty certificates has failed, it marks the corresponding satellite fully affected. It marks all other satellites related to failed certificates as partially affected at the rank at which the certificate is defined.

In the second stage, the update algorithm iterates through each rank, updating the structure in three phases: *remove*, *convert*, and *repair*. In the remove phase, it removes the fully affected satellites from the spanner by calling `Remove`, which removes a satellite s in a bottom-up pass starting from its minimum rank λ_s until its maximum rank Λ_s by removing it from the neighbors, parent, and child lists. This function runs in $O(\Lambda_s - \lambda_s)$ time. In the convert phase, similar to the construction algorithm, the update algorithm tries to convert all fully affected satellites (including the ones that are removed earlier) to Steiner vertices, provided that their nuclei are active and their certificate balls are empty. Finally, in the repair phase, the update algorithm repairs the discretization at the next rank by updating the promotion decisions for the affected Steiner vertices using `UpdatePromotion`, which performs the discretization step only on the affected Steiner vertices. This function takes linear time in the number of affected Steiner vertices, which we prove to be of constant size. After the update algorithm is done with the second stage, it updates the Delaunay triangulation in the third stage by computing the up-to-date Voronoi neighbors of the affected vertices.

If an update affects a ball-empty/ball-not-empty certificate of an ℓ-satellite s, it enqueues s into

the $\mathcal{F}_\ell$ list. The only exception to this rule is that the update algorithm never enqueues a satellite into $\mathcal{F}$ lists if its ball-empty certificate is affected during a call to `Remove`. If an update affects an edge/potential-neighbor certificate of an Steiner vertex s at rank ℓ, the algorithm enqueues s into the $\mathcal{P}_\ell$ list. Similarly, if an update affects a parent-child/separation certificate of two vertices at consecutive/same ranks, it enqueues these vertices in $\mathcal{P}_\ell$ ($\mathcal{P}_{\ell+1}$) list. Finally, if an update affects a Voronoi certificate, it enqueues the ℓ-satellite for which the Voronoi cell is being computed to $\mathcal{P}_\ell$ list. Based on our update algorithm, we state following lemma without a proof.

Lemma 4.3.2 *After the iteration at rank ℓ, the spanner contains a maximal independent subset of the satellites in the up-to-date proximity graph at rank ℓ.*

Lemma 4.3.3 *At rank ℓ, the number of active vertices in a ball of radius $O(2^\ell)$ is bounded by a constant.*

Proof: A vertex v inside the given ball may be active for two reasons. Either its activation rank is ℓ or one of its satellites at rank $\ell-1$ is converted. In the first case, we bound the nearest neighbor distance of v by $\Omega(2^\ell)$. Thus, a packing argument bounds the number of vertices that fall into the first case by $O(1)$. In the second case, let s be a converted $(\ell-1)$-satellite of v. Then, there is an empty ball around s with radius $\Omega(2^\ell)$. At a fixed rank and ray, all satellites are shifted versions of input vertices. Therefore, if the $(\ell-1)$-satellite on the same ray (same polar angle) of another vertex u is converted, we have $|uv| \in \Omega(2^\ell)$. Similar packing arguments bound the number of such vertices by $O(1)$ for each ray r. Since there is a constant number of rays, the result follows. ■

4.3.1 Responsiveness Analysis

In order to bound the runtime of our updates, we define the *focus* of the dynamic update or the kinetic event as one of the input vertices and prove that all modifications take place around the focus. For a dynamic modification the focus is the vertex being inserted or deleted. For a kinetic event, consider the vertices/satellites involved in the certificate failure. Representing the satellites

with their nuclei and input vertices by themselves, the focus is any of the two representations of the points involved in the certificate.

Lemma 4.3.4 *Consider a kinetic event or a dynamic modification and let p be the focus. For any rank ℓ, any satellite in $\mathcal{F}_\ell$ or $\mathcal{P}_\ell$ lists is $O(2^\ell)$ away from p.*

Proof: We use induction over the order in which the algorithm inserts satellites into these lists. For the base case, consider the affected lists at the end of the first stage. Fix rank ℓ and consider a satellite s satisfying the premises. Construction and the update algorithms of Gao et al. perform local operations at each rank. More specifically, given the focus p, if the update removes or inserts an edge in the spanner at rank ℓ, the vertices incident to the edge are within $O(2^\ell)$ distance of p. Therefore, if s is enqueued into these lists while repairing the initial spanner structure then we have $|sp| < O(2^\ell)$. Similar to the base case, we prove that if applying an update on a satellite s_1 affects another satellite s_2 and s_2 is either fully affected at or partially affected at rank ℓ', then we have $|s_1 s_2| < O(2^{\ell'})$. Geometrically relating the affected satellites defines a dependence path from each affected satellite to the focus p. Except for a constant overhead, all the effects grow in ranks, therefore, using the triangle inequality on any of the paths from p to s, we can bound the distance between p and s by a geometric series with constantly many repetitions of the terms. Using the fact that a geometric series is dominated by its last term, and the fact that the last term is $O(2^\ell)$, we conclude our result. ■

Lemma 4.3.5 *For any rank ℓ, there are $O(1)$ satellites in $\mathcal{F}_\ell$ and $\mathcal{P}_\ell$ lists.*

Proof: Let s be an affected satellite, by Lemma 4.3.4, s lies inside a ball of radius $O(2^\ell)$ around the focus p. We analyze the two cases, $s \in \mathcal{F}_\ell$ and $s \in \mathcal{P}_\ell$ and prove that in each case there is a constant number of such satellites. For the first case, assume that $s \in \mathcal{F}_\ell$, i.e., s is an ℓ-satellite of an active vertex. The nucleus lies inside a slightly larger ball around p of radius $O(2^\ell)$. Lemma 4.3.3 proves that there are only $O(1)$ many vertices; since a vertex has $O(1)$ satellites at a given rank, we conclude the first case. For the second case, assume that $s \in \mathcal{P}_\ell$, i.e., s is an ℓ-center. Therefore,

s is at least 2^ℓ far from another ℓ-center. Since s is $O(2^\ell)$ close to p, a packing argument bounds the number of such vertices by $O(1)$ as well. ■

Lemma 4.3.6 *The total runtime of* `Remove` *calls made by the update algorithm is* $O(\log \Delta)$.

Proof: By Lemma 4.3.4, for each rank ℓ, all of the fully affected satellites are within $O(2^\ell)$ distance of the focus p. The runtime required to remove these affected satellites is

$$T < \sum_{\ell=\lambda}^{\Lambda} \sum_{s \in \mathcal{F}_\ell} \alpha \cdot (\Lambda_s - \lambda_s)$$

where λ and Λ are the minimum and the maximum rank of the spanner and α is the constant in the big-Oh notation hidden in the runtime of the `Remove` function. Using Lemma 4.2.10 to bound λ_s, we rewrite $T < O(\log \Delta) + \alpha \cdot \sum_{\ell=\lambda}^{\Lambda} \sum_{s \in \mathcal{F}_\ell} (\Lambda_s - \ell)$. Rearranging, we get

$$T < O(\log \Delta) + \alpha \cdot \sum_{\ell=\lambda}^{\Lambda} \sum_{\ell'=\ell}^{\Lambda} \left| \{ s \mid s \in \mathcal{F}_\ell, \Lambda_s > \ell' \} \right|$$

$$T < O(\log \Delta) + \alpha \cdot \sum_{\ell'=\lambda}^{\Lambda} \sum_{\ell=\lambda}^{\ell'} \left| \{ s \mid s \in \mathcal{F}_\ell, \Lambda_s > \ell' \} \right|$$

For any given ℓ', we claim that $\sum_{\ell=\lambda}^{\ell'} \left| \{ s \mid s \in \mathcal{F}_\ell, \Lambda_s > \ell' \} \right|$ is bounded by $O(1)$: for any $\ell \leq \ell'$, the Steiner vertices in $\mathcal{F}_\ell$ are within $O(2^{\ell'})$ distance of the focus p and any vertex with $\Lambda_s > \ell'$ is an ℓ'-center; each of them has an empty ball of radius $2^{\ell'}$. Then, a packing argument proves our claim and we get $T = O(\log \Delta)$. ■

Theorem 4.3.7 *Given a kinetic event or a dynamic modification, the update algorithm repairs the spanner structure and the well-spaced, size-optimal superset in* $O(\log \Delta)$ *time.*

Proof: At each rank, ensuring that the output contains a maximal independent subset proves well-spacedness (Lemmas 4.2.3, 4.2.7, and 4.3.2). Since we update the minimum ranks of affected

input vertices, we can apply Lemma 4.2.8 to ensure size-optimality as well. Lemma 4.3.5 bounds the total number of affected satellites by $O(\log \Delta)$. Processing each of the affected satellites takes $O(1)$ time except for removal. Lemma 4.3.6 bounds the total runtime of the removal of the satellites by $O(\log \Delta)$. Thus, our result follows. ■

4.4 Quality of the KDS

In order to prove the efficiency of our kinetic data structure we show that our KDS is responsive, local, compact, and efficient. We proved responsiveness in the previous section, that when a certificate fails, the KDS can be updated quickly, in $O(\log \Delta)$ time. In this section, we prove that our KDS is compact and local, i.e., it has near linear total number of certificates, and each input vertex participates in a logarithmic number of certificates. Then, we prove that our KDS is efficient, i.e., there are not too many certificate failures compared to the number of combinatorial changes required in the worst case.

Lemma 4.4.1 *Every input vertex participates in $O(\log \Delta)$ certificates. In total, our kinetic data structure maintains $O(m)$ certificates, where m is $|\mathsf{M}|$.*

Proof: Consider an input vertex v and fix a rank ℓ. In Lemma 4.2.11, we prove that v has $O(1)$ converted satellites that are ℓ-centers in the final spanner. Since there are $O(\log \Delta)$ ranks and since each of the ℓ-centers is associated with a constant number of certificates, each input vertex, through its satellites, participates in $O(\log \Delta)$ certificates. For the total number of certificates, we already know that the resulting spanner has $O(m)$ neighbor edges and that there are $O(m)$ ball-empty certificates. We are left to prove that there are $O(m)$ ball-not-empty certificates. For each satellite that is not converted, we charge its ball-not-empty certificate to an inserted satellite at the previous rank or if no such satellite exists to its nucleus. Using this approach, every Steiner vertex or input vertex is charged at most $O(1)$ ball-not-empty certificates. There are $O(m)$ vertices in the output, therefore, the result follows. ■

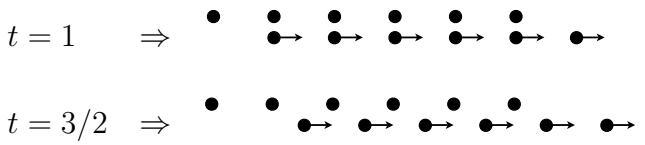


Figure 4.6: Consider a horizontal line of $2k$ evenly-spaced vertices: $(0, 0)$, $(1, 0)$, $\ldots (2k, 0)$, and a second line ϵ above the first line: $(0, \epsilon)$, $(1, \epsilon)$, $\ldots (2k, \epsilon)$. Assign a fixed velocity vector $(1, 0)$ to the vertices of the lower line. The upper line does not move.

We know that each input vertex may introduce $O(\log \Delta)$ many satellites, a total of $O(n \log \Delta)$. Most of our certificates are distances between pairs of vertices and there are $O(\log \Delta)$ different distances we consider, therefore, our algorithm processes a total of at most $O(n^2 \log^3 \Delta)$ events through these types of certificate failures. For certifying the Delaunay triangulation, by the above analysis, we know that the candidate Voronoi neighbor set of a given Steiner vertex changes $O(n \log^2 \Delta)$ many times. There are $O(1)$ halfspace tests performed to determine the exact Voronoi cell, hence, our algorithm processes a total of at most $O(n^2 \log^3 \Delta)$ events altogether. We show that there exist examples for which maintaining a size-optimal well-spaced point set requires $\Omega(n^2 \log \Delta)$ combinatorial changes. Specifically, consider the example shown in Figure 4.6 with $n = 4k$ points and $\epsilon = 1/k$ and let time evolve from $t = 0$ to k. The diameter is $\Theta(n)$ and the minimum pairwise distance oscillates between $\Theta(1)$ at half-integer times and $\Theta(\epsilon)$ at integer times. The spread is therefore $\Delta \in \Theta(n/\epsilon)$, which implies $\log \Delta \in \Theta(\log \epsilon^{-1})$.

Lemma 4.4.2 *At integer times, a well-spaced superset requires* $\Omega(n \log \Delta)$ *Steiner vertices to be inserted.*

Proof: Ruppert proves [60] that even the smallest well-spaced superset of a given input has $\int_\Omega \mathrm{lfs}^{-d}(x)dx$ vertices, where Ω denotes the domain. To bound the integral, for each pair (u_i, v_i) of ϵ-close vertices, take their midpoint p_i. We define a set of non-overlapping balls $B(p_i, 1/2)$ for each of the at least k pairs of ϵ-close vertices. The integral over all of Ω is lower-bounded by the

sums of the integrals over each ball:

$$\int_\Omega \mathrm{lfs}^{-d}(x)dx \geq \sum_{i=1}^{k} \int_{B(p_i,1/2)} \mathrm{lfs}^{-d}(x)dx$$

At the midpoint, $\mathrm{lfs}(p_i) = \epsilon/2$. Because lfs is 1-Lipschitz, $\mathrm{lfs}(x) \leq |p_i x| + \epsilon/2$. Then the integral over each ball is at least $\Omega(\log \epsilon^{-1})$. Since $k = n/4$, the sum is $\Omega(n \log \epsilon^{-1})$. ∎

Lemma 4.4.3 *At half-integer times, inserting $O(n)$ Steiner vertices is sufficient for the lower bound example of Figure 4.6.*

Proof: This proof requires some results from the field of curve reconstruction. We refer to the book by Dey [29] for an introduction. The key result we establish is that the input vertices form a good sample of a smooth curve: it is sufficiently dense so that no point of the curve is too far from a vertex, but sufficiently sparse so that no two vertices are too close to each other. Density and sparsity are both relative to the distance from the curve to its medial axis. We can fit a sinusoidal curve through the vertices as they are arranged at half-integer times; the curve has amplitude $\epsilon/2$. Any point on the curve is at distance less than $1/2$ from an input vertex. Meanwhile, the medial axis of the curve is at distance $\frac{1}{8\epsilon} + \frac{\epsilon}{2} = \Theta(1/\epsilon)$ from the curve. Thus the vertices form an $O(\epsilon)$-dense sample of the curve. Vertices are at distance $\sqrt{\epsilon^2 + 1/4} \approx 1/2$ from each other, thus they form an $\Omega(\epsilon)$-sparse sample. Hudson, Miller, Phillips and Sheehy [45] show that such a vertex set has a well-spaced superset of size $O(n)$. ∎

Lemma 4.4.4 *The lower bound example of Figure 4.6 requires $\Omega(n^2 \log \Delta)$ Steiner vertex insertions and deletions.*

Proof: At integer times, the input requires $\Omega(n \log \Delta)$ Steiner vertices to be made well-spaced. At half-integer times, the input requires that there be no more than $O(n)$ Steiner vertices to be size-optimal. Thus, any algorithm that maintains a size-optimal well-spaced superset must add and subsequently remove $\Theta(n \log \Delta)$ vertices for every unit of time, a total of $\Omega(n^2 \log \Delta)$ such changes. ∎

CHAPTER 5

CONCLUDING REMARKS

By applying the stable design approach in developing dynamic and kinetic algorithms, we solve several open problems in computational geometry. The stable design approach requires designing a *construction algorithm* that performs a computation in such a way that when the algorithm is executed on similar inputs, the executions themselves are similar, i.e., remain stable. Stable construction algorithms make it possible for a generic change-propagation algorithm to update the execution and the output when the input data changes dynamically due to insertions and deletions or kinetically due to continuous motion. Since the change-propagation algorithm is fully generic, it can adapt the output under any changes to the input data, making it possible to process simultaneous events that arise during motion simulation.

The stability approach to designing dynamic and kinetic algorithms is motivated by recent advances on self-adjusting computation, which make it possible to translate a given construction algorithm into a dynamic or a kinetic update algorithm. By using this approach, we gain a broader view of the given problem; instead of considering the details of the update algorithm itself, we focus on the construction algorithm and its stability analysis. Since we focus on the stability of the construction algorithm for dynamic and kinetic update algorithms, we pay closer attention to the maintenance problem while organizing the construction rather than delaying it (partially) only after the construction is completed. In the remaining of this chapter, we provide further details of our approach by summarizing our conclusions for each problem we consider in this thesis.

Chapter 2 presents a technique for robust motion simulation based on a hybrid of kinetic-event scheduling and fixed-time sampling. The idea behind this technique is to partition the time line into a lattice of intervals, perform motion simulation at the resolution of an interval, and process events in the same interval together, regardless of their relative order. To separate roots to the resolution of intervals, we use Sturm sequences in a way similar to their use for exact separation of roots in previous work; the fixed resolution, however, allows us to stop the process early.

The approach critically relies on self-adjusting computation, which enables processing multiple events simultaneously. Although robustness issues motivate the hybrid technique using kinetic-event scheduling and fixed-time sampling, this technique may also help in situations where explicit motion prediction is difficult [15].

We apply the approach to kinetic convex hulls in 3D by kinetizing a version of the incremental convex-hull algorithm via self-adjusting computation. We implement the motion simulator and the algorithm and perform an experimental evaluation. Our experiments show that our algorithm is effective in practice: we are able to run efficient robust simulations involving thousands of points. Our experiments also indicate that the data structure can respond to a kinetic event, as well as to an integrated dynamic change (an insertion/deletion during motion simulation), in logarithmic time in the size of the input. To the best of our knowledge, this is the first implementation of kinetic 3D convex hulls that can guarantee robustness for reasonably large input sizes.

Chapter 3 presents a dynamic algorithm for computing a well-spaced point set of a dynamically changing set of input points. Our algorithm is efficient, finds an optimal-size output, consumes linear space, and responds to dynamic modifications in worst-case optimal time. The underlying technique behind these results is a stable algorithm for computing well-spaced point sets whose executions can be represented with computation graphs that remain similar when the input sets themselves are similar. Our dynamic update algorithm takes advantage of stability and efficiently updates the output by propagating the input modification through the computation graph. To the best of our knowledge, this is the first time- and size-optimal algorithm for dynamically maintaining well-spaced point sets.

To assess the practicality of our approach we refer to a prototype implementation [11]. Our experiments show that the algorithm can be implemented efficiently such that it delivers performance consistent with our theoretical bounds. We expect a well-polished implementation will provide static performance comparable to the state of the art, and dynamic performance orders of magnitude faster.

Chapter 4 presents a kinetic data structure for mesh refinement that computes the Delaunay triangulation of a size-optimal well-spaced superset of a set of moving points in the plane. Our KDS is compact, responsive, local, and efficient: it requires linear space in the size of the output; it repairs itself in logarithmic time; every point is involved in a logarithmic number of certificates; and the number of events is within a polylogarithmic factor of optimal. Also, our KDS is dynamic, responding to point insertions and deletions in logarithmic time. To the best of our knowledge, this is the first KDS for mesh refinement.

Our approach, inspired by self-adjusting computation techniques, critically relies on deformable spanners [33]. In fact, one can decribe our KDS as a balanced (deformable) spanner. Similar to the balanced quadtrees, it extends the deformable spanner with such additional points that in the resulting superset, all neighbors of a point in the spanner have empty balls of similar sizes. Also, because we take a self-adjusting approach, our KDS can handle simultaneous certificate failures, making it effective in practice.

Our result applies only to the planar case, though it is very promising for arbitrary-dimension extension. The two missing pieces are these: (1) definition of satellites in higher dimensions, which will let us kinetically maintain well-spacing; (2) kinetization of a method to convert a well-spaced point cloud into a quality mesh (e.g. [22]), since the direct correspondence between well-spacing and Delaunay mesh quality applies only in two dimensions.

Our bounds in Chapters 3 and 4 depend on the spread Δ, which *a priori* has no relationship to the input size n. However, if the points form an ϵ-net of a manifold, then the spread is at worst linear in n. This is because in an ϵ-net, no point of the manifold is farther than ϵ from an input point, which bounds the diameter by $O(n\epsilon)$, and no two input points are at distance $o(\epsilon)$ from each other, which bounds the closest pair by $\Omega(\epsilon)$. Furthermore, in an ϵ-net, the output size is in $O(n)$ [45]. In other words, if points are a sample taken from a moving manifold, then our meshes have linear size, our update algorithms take $O(\log n)$ time, and our KDS is efficient in the usual sense of being within a $\mathrm{polylog}(n)$ factor of optimal.

REFERENCES

[1] CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.

[2] M. A. Abam, P. K. Agarwal, M. de Berg, and H. Yu. Out-of-order event processing in kinetic data structures. In *14th Annual European Symposium on Algorithms*, volume 4168, pages 624–635. Springer, 2006.

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

[4] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Transactions on Programming Languages and Systems*, 32(1):3:1–3:53, 2009.

[5] U. A. Acar, G. E. Blelloch, M. Blume, and K. Tangwongsan. An experimental analysis of self-adjusting computation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[6] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Transactions on Programming Languages and Systems*, 28(6):990–1034, 2006.

[7] U. A. Acar, G. E. Blelloch, R. Ley-Wild, K. Tangwongsan, and D. Türkoğlu. Traceable data types for self-adjusting computation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[8] U. A. Acar, G. E. Blelloch, and K. 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.

[9] U. A. Acar, G. E. Blelloch, K. Tangwongsan, and D. Türkoğlu. Robust Kinetic Convex Hulls in 3D. In *16th Annual European Symposium on Algorithms*, 2008.

[10] U. A. Acar, A. Cotter, B. Hudson, and D. Türkoğlu. Dynamic well-spaced point sets. *Submitted to Computational Geometry: Theory and Applications*.

[11] U. A. Acar, A. Cotter, B. Hudson, and D. Türkoğlu. Dynamic well-spaced point sets. In *SCG '10: Proceedings of the 26th Annual Symposium on Computational Geometry*, 2010.

[12] U. A. Acar, B. Hudson, and D. Türkoğlu. Kinetic mesh refinement in 2d. In *SCG '11: Proceedings of the 27th Annual Symposium on Computational Geometry*, 2011.

[13] U. A. Acar and R. Ley-Wild. Self-adjusting computation with Delta ML. *Lecture Notes in Computer Science*, 5832/2009:1–38, 2009.

[14] P. K. Agarwal, J. Gao, L. J. Guibas, H. Kaplan, V. Koltun, N. Rubin, and M. Sharir. Kinetic stable delaunay graphs. In *SCG '10: Proceedings of the 26th Annual Symposium on Computational Geometry*, 2010.

[15] P. K. Agarwal, L. J. Guibas, H. Edelsbrunner, J. Erickson, M. Isard, S. Har-Peled, J. Hershberger, C. Jensen, L. Kavraki, P. Koehl, M. Lin, D. Manocha, D. Metaxas, B. Mirtich, D. Mount, S. Muthukrishnan, D. Pai, E. Sacks, J. Snoeyink, S. Suri, and O. Wolefson. Algorithmic issues in modeling motion. *ACM Comput. Surv.*, 34(4):550–572, 2002.

[16] P. K. Agarwal, Y. Wang, and H. Yu. A two-dimensional kinetic triangulation with near-quadratic topological changes. *Discrete & Computational Geometry*, 36(4):573–592, 2006.

[17] A. W. Bargteil, C. Wojtan, J. K. Hodgins, and G. Turk. A finite element method for animating large viscoplastic flow. *ACM Trans. Graph.*, 26, July 2007.

[18] J. Basch. *Kinetic Data Structures*. PhD thesis, Department of Computer Science, Stanford University, June 1999.

[19] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31(1):1–28, 1999.

[20] M. Bern, D. Eppstein, and J. R. Gilbert. Provably Good Mesh Generation. *Journal of Computer and System Sciences*, 48(3):384–409, 1994.

[21] D. Cardoze, A. Cunha, G. L. Miller, T. Phillips, and N. Walkington. A bézier-based approach to unstructured moving meshes. In *Proceedings of the twentieth annual symposium on Computational geometry*, SCG '04, pages 310–319, New York, NY, USA, 2004. ACM.

[22] S.-W. Cheng, T. K. Dey, H. Edelsbrunner, M. A. Facello, and S.-H. Teng. Sliver Exudation. *Journal of the ACM*, 47(5):883–904, 2000.

[23] N. Chentanez, R. Alterovitz, D. Ritchie, L. Cho, K. K. Hauser, K. Goldberg, J. R. Shewchuk, and J. F. O'Brien. Interactive simulation of surgical needle insertion and steering. In *Proceedings of ACM SIGGRAPH*, Aug 2009.

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

[25] L. P. Chew. Guaranteed-quality delaunay meshing in 3d. In *SCG '97: Proceedings of the Thirteenth Annual Symposium on Computational Geometry*, pages 391–393, 1997.

[26] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE, Special issue on Computational Geometry*, 80(9):1412–1434, 1992.

[27] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry,II. *Discrete and Computational Geometry*, 4(1):387–421, 1989.

[28] N. Coll, M. Guerrieri, and J. A. Sellarès. Mesh modification under local domain changes. In *15th International Meshing Roundtable*, pages 39–56, 2006.

[29] T. K. Dey. *Curve and Surface Reconstruction*. Cambridge University Press, 2006.

[30] H. Edelsbrunner, X.-Y. Li, G. L. Miller, A. Stathopoulos, D. Talmor, S.-H. Teng, A. Üngör, and N. Walkington. Smoothing and cleaning up slivers. In *STOC*, pages 273–277, 2000.

[31] D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 8. CRC Press, 1999.

[32] J.-J. Fu and R. C. T. Lee. Voronoi diagrams of moving points in the plane. In *FST and TC 10: Proceedings of the tenth conference on Foundations of software technology and theoretical computer science*, pages 238–254, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[33] J. Gao, L. J. Guibas, and A. Nguyen. Deformable spanners and applications. *Computational Geometry: Theory and Applications*, 35(1):2–19, 2006.

[34] L. J. Guibas. Kinetic data structures: a state of the art report. In *WAFR '98: Proceedings of the third workshop on the algorithmic foundations of robotics on Robotics : the algorithmic perspective*, pages 191–209, Natick, MA, USA, 1998. A. K. Peters, Ltd.

[35] L. J. Guibas. Modeling motion. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 1117–1134. Chapman and Hall/CRC, 2nd edition, 2004.

[36] L. J. Guibas and M. I. Karavelas. Interval methods for kinetic simulations. In *SCG '99: Proceedings of the fifteenth annual symposium on Computational geometry*, pages 255–264. ACM Press, 1999.

[37] L. J. Guibas, J. S. B. Mitchell, and T. Roos. Voronoi diagrams of moving points in the plane. In *17th International Workshop Graph-Theoretic Concepts Computer Science*, pages 113–209. Springer-Verlag, Inc., 1992.

[38] L. J. Guibas and D. Russel. An empirical comparison of techniques for updating delaunay triangulat ions. In *SCG '04: Proceedings of the twentieth annual symposium on Computat ional geometry*, pages 170–179, New York, NY, USA, 2004. ACM Press.

[39] M. Hammer and U. A. Acar. Memory management for self-adjusting computation. In *International Symposium on Memory Management*, pages 51–60, 2008.

[40] M. Hammer, G. Neis, Y. Chen, and U. A. Acar. Self-adjusting stack machines. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[41] M. A. Hammer, U. A. Acar, and Y. Chen. CEAL: A C-based language for self-adjusting computation. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2009.

[42] S. Har-Peled and A. Üngör. A time-optimal Delaunay refinement algorithm in two dimensions. In *Proceedings of the 21st Annual Symposium on Computational Geometry*, pages 228–236, 2005.

[43] B. Hudson. *Dynamic Mesh Refinement*. PhD thesis, School of Computer Science, Carnegie Mellon University, Dec. 2007. Available as Technical Report CMU-CS-07-162.

[44] B. Hudson, G. L. Miller, and T. Phillips. Sparse Voronoi Refinement. In *15th International Meshing Roundtable*, pages 339–356, 2006. Long version in Carnegie Mellon University Technical Report CMU-CS-06-132.

[45] B. Hudson, G. L. Miller, T. Phillips, and D. R. Sheehy. Size complexity of volume meshes vs. surface meshes. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, 2009.

[46] B. Hudson and D. Türkoğlu. An efficient query structure for mesh refinement. In *Canadian Conference on Computational Geometry*, 2008.

[47] R. Jampani and A. Üngör. Construction of sparse well-spaced point sets for quality tetrahedralizations. In *16th International Meshing Roundtable*, pages 63–80, 2007.

[48] H. Kaplan, N. Rubin, and M. Sharir. A kinetic triangulation scheme for moving points in the plane. In *SCG '10: Proceedings of the 26th Annual Symposium on Computational Geometry*, 2010.

[49] B. M. Klingner, B. E. Feldman, N. Chentanez, and J. F. O'Brien. Fluid animation with dynamic meshes. *ACM Trans. Graph.*, 25:820–825, July 2006.

[50] R. Ley-Wild. *Programmable Self-Adjusting Computation*. PhD thesis, Computer Science Department, Carnegie Mellon University, Oct. 2010.

[51] R. Ley-Wild, U. A. Acar, and M. Fluet. A cost semantics for self-adjusting computation. In *Proceedings of the 26th Annual ACM Symposium on Principles of Programming Languages*, 2009.

[52] R. Ley-Wild, M. Fluet, and U. A. Acar. Compiling self-adjusting programs with continuations. In *Int'l Conference on Functional Programming*, 2008.

[53] X.-Y. Li and S.-H. Teng. Generating well-shaped Delaunay meshes in 3D. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 28–37, 2001.

[54] X.-Y. Li, S.-H. Teng, and A. Üngör. Simultaneous refinement and coarsening for adaptive meshing. *Engineering with Computers*, 15(3):280–291, 1999.

[55] G. L. Miller, D. Talmor, S.-H. Teng, N. Walkington, and H. Wang. Control Volume Meshes Using Sphere Packing: Generation, Refinement and Coarsening. In *15th International Meshing Roundtable*, pages 47–61, 1996.

[56] N. Molino, Z. Bao, and R. Fedkiw. A virtual node algorithm for changing mesh topology during simulation. *ACM Trans. Graph.*, 23:385–392, Aug. 2004.

[57] D. Moore. The cost of balancing generalized quadtrees. In *SMA '95: Proceedings of the third ACM symposium on Solid modeling and applications*, pages 305–312, New York, NY, USA, 1995. ACM.

[58] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[59] H.-W. Nienhuys and A. F. van der Stappen. A Delaunay approach to interactive cutting in triangulated surfaces. In *Fifth International Workshop on Algorithmic Foundations of Robotics*, 2004.

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

[61] D. Russel. *Kinetic Data Structures in Practice*. PhD thesis, Department of Computer Science, Stanford University, Mar. 2007.

[62] D. Russel, M. I. Karavelas, and L. J. Guibas. A package for exact kinetic data structures and sweepline algorithms. *Computational Geometry Theory and Applications*, 38(1-2):111–127, 2007.

[63] J. R. Shewchuk. Tetrahedral mesh generation by Delaunay refinement. pages 86–95, 1998.

[64] D. Spielman, S.-h. Teng, and A. Üngör. Parallel delaunay refinement with off-centers. In *Euro-Par 2004 Parallel Processing*, volume 3149, pages 812–819. Springer Berlin / Heidelberg, 2004.

[65] D. Spielman, S.-H. Teng, and A. Üngör. Parallel Delaunay refinement: Algorithms and analyses. *IJCGA*, 17:1–30, 2007.

[66] D. Talmor. *Well-Spaced Points for Numerical Methods*. PhD thesis, Carnegie Mellon University, August 1997. Available as Technical Report CMU-CS-97-164.

APPENDIX A

IMPLEMENTATION

A.1 Math Library

We provide the implementations of the constituents of the math library, called *number*, *polynomial*, and *geometry*. Our librares are written in ΔML.

A.1.1 Numbers

In the numbers library, we implement the basic algebraic structure ring, and extend this structure for supporting exact arithmetic. We design a binary number representation with a variable radix point and implement the following two versions of binary numbers: one using IntInf structure so that the computations are exact, and one using floating point numbers (Real structure) so that we can compare the speed difference. Also, we provide an interval structure that will be helpful in implementing Sturm sequences.

```
signature RING =
sig
   type t

   (*** Zero element for addition and multiplication ***)
   val zero : t
   val one : t

   (*** Arithmetic ***)
   val + : t * t -> t
   val - : t * t -> t
   val ~ : t -> t
   val * : t * t -> t

   (*** Equality ***)
   val == : t * t -> bool
   val != : t * t -> bool

   (*** String ***)
   val toString : t -> string
   val fmtString : t -> string

   (*** Self adjusting stuff ***)
   val eq : t Eq.t
   val hash : t Hash.t
   val key : t Key.t

end (* RING *)
```

```
signature BINARY_NUMBER =
sig
   include RING

   (*** Comparison ***)
   val compare : t * t -> order
   val < : t * t -> bool
   val <= : t * t -> bool
   val > : t * t -> bool
   val >= : t * t -> bool

   (*** Min/Max using comparisons ***)
   val min : t * t -> t
   val max : t * t -> t

   (*** Extended arithmetic ***)
   val sign : t -> int
   val abs : t -> t
   (*** Generalized multiplication by 2 (division for negative values) ***)
   val shift : t * int -> t
   (*** Division where the result is truncated to an integer. ***)
   val div : t * t -> t
   (*** Modulus ***)
   val mod : t * t -> t
   (*** Greatest common divisor where each quotient is an integer. ***)
   val gcd : t * t -> t

   (*** Conversion ***)
   val fromReal : real -> t
   val toReal : t -> real (* by truncation *)
   val fromInt : int -> t
   val toInt : t -> int   (* by truncation *)

   (*** Restriction ***)
   (*** Second arguments specify the number of bits to keep after the
    *** radix point. Formally, truncate(n, r) = trunc(n * 2^r) / 2^r,
    *** where trunc is the function that truncates to an integer.
    *** Other functions are defined similarly. ***)
                               (*** Let n = 1.01, r = 1.***)
   val truncate : t * word -> t (*** truncate (n, r) = 1.0
                                     truncate (~n, r) = ~1.0 ***)
   val floor : t * word -> t    (*** floor (n, r) = 1.0
                                     floor (~n, r) = ~1.1 ***)
   val ceiling : t * word -> t  (*** ceiling (n, r) = 1.1
                                     ceiling (~n, r) = ~1.0 ***)

end (* BINARY_NUMBER *)



signature INTERVAL =
sig
   structure Number : BINARY_NUMBER

   (*** Boundary type used in representing intervals ***)
   datatype b = POS_INFTY | NEG_INFTY | OPEN of Number.t | CLOSED of Number.t

   (*** t is a tuple of boundaries ***)
   type t = b * b

   (*** Conversion ***)
   val fromReal : real * word * bool * bool -> t
   val toNumber : t -> Number.t * Number.t

   (*** Equality ***)
   val == : t * t -> bool
```

```
   val != : t * t -> bool

   (*** Comparison ***)
   val compare : t * t -> order
   val < : t * t -> bool
   val <= : t * t -> bool
   val > : t * t -> bool
   val >= : t * t -> bool

   (*** Operation ***)
   val length : t -> Number.t
   val intersect : t * t -> t option

   (*** String ***)
   val toString : t -> string

end



structure ExactFloatingPoint : BINARY_NUMBER =
struct
   structure Base = IntInf

   (*** Representation of y / (2^r) is (y, r): IntInf.int * word (r >= 0) ***)
   type t = Base.int * word

   (*** Zeroes of addition and multiplication ***)
   val baseZero = Base.fromInt 0
   val baseOne = Base.fromInt 1
   val powerZero = Word.fromInt 0

   val zero = (baseZero, powerZero)
   val one = (baseOne, powerZero)

   (*** Representation of a tuple using the same floating point location ***)
   fun tuple ((x, p), (y, q)) =
      if Word.>(p, q) then ((x, Base.<<(y, Word.-(p, q))), p)
      else ((Base.<<(x, Word.-(q, p)), y), q)

   fun baseTuple (n, m) = #1 (tuple (n, m))

   (*** Arithmetic ***)
   fun op + (n, m) = let val (xy, p) = tuple(n, m) in (Base.+ xy, p) end
   fun op - (n, m) = let val (xy, p) = tuple(n, m) in (Base.- xy, p) end
   fun op ~ (x, p) = (Base.~ x, p)
   fun op * ((x, p), (y, q)) = (Base.*(x, y), Word.+(p, q))

   fun shift ((x, p), q) =
      let
         val r = Int.-((Word.toInt p), q)
      in
         if Int.>=(r, 0) then (x, Word.fromInt r)
         else (Base.<<(x, Word.fromInt(Int.~ r)), powerZero)
      end

   (*** Equality ***)
   val op == = (op =) o baseTuple
   val op != = (op <>) o baseTuple

   (*** Extended arithmetic ***)
   fun sign (x, p) = Base.sign x
   fun abs (x, p) = (Base.abs x, p)

   fun op div (m, n) =
      let val (xy, p) = tuple(m, n) in (Base.div xy, powerZero) end
```

```
fun op mod (m, n) = let val (xy, p) = tuple(m, n) in (Base.mod xy, p) end
fun gcd (m, n) = if ==(n, zero) then abs(m) else gcd(n, m mod n)

(*** Comparison ***)
val compare = Base.compare o baseTuple
val op < = Base.< o baseTuple
val op <= = Base.<= o baseTuple
val op > = Base.> o baseTuple
val op >= = Base.>= o baseTuple

(*** Min/Max ***)
fun min (n, m) = if (n <= m) then n else m
fun max (n, m) = if (n >= m) then n else m

(*** Restriction ***)
fun truncate (n as (x, p), q) =
   if Word.<=(p, q) then n else (Base.~>>(x, Word.-(p, q)), q)

fun allZeroes (x, p) =
   (Base.andb(Base.-(Base.<<(baseOne, p), baseOne), x) = baseZero)

fun floor (n as (x, p), q) =
   if Word.<=(p, q) then n else
      let
         val diff = Word.-(p, q)
         val y = Base.~>>(x, diff)
      in
         if Base.>=(x, baseZero) orelse allZeroes(x, diff) then (y, q)
         else (Base.-(y, baseOne), q)
      end

fun ceiling (n as (x, p), q) =
   if Word.<=(p, q) then n else
      let
         val diff = Word.-(p, q)
         val y = Base.~>>(x, diff)
      in
         if Base.<=(x, baseZero) orelse allZeroes(x, diff) then (y, q)
         else (Base.+(y, baseOne), q)
      end

(*** Conversion ***)
fun fromReal r =
   if Real.==(r, 0.0) then zero else (* ML has a weird problem *)
   let
      val {exp = p, man = m} = Real.toManExp r
      val x = Real.toLargeInt IEEEReal.TO_ZERO
             (Real.fromManExp {exp = (Real.precision), man = m})

      fun search (s0, s2) =
         if Int.-(s2, s0) = 1 then s0 else
            let
               val s1 = Int.div(Int.+(s0, s2), 2)
            in
               if not (allZeroes(x, Word.fromInt s1)) then search(s0, s1)
               else search(s1, s2)
            end

      val p = Int.-(Real.precision, p)
      val s = search(0, Real.precision)
      val q = Int.-(p, s)
   in
      if Int.>=(q, 0) then (Base.~>>(x, Word.fromInt s), Word.fromInt q)
      else if Int.>=(p, 0) then (Base.~>>(x, Word.fromInt p), powerZero)
      else (Base.<<(x, Word.fromInt(Int.~ p)), powerZero)
   end
```

```
   fun toReal (x, p) =
      if (x = baseZero) then 0.0 else (* ML has a weird problem *)
      Real.fromManExp {exp = Int.~(Word.toInt p), man = Real.fromLargeInt x}

   fun fromInt i = (Base.fromInt i, powerZero)

   fun toInt n = Base.toInt(#1 (truncate(n, powerZero)))

   (*** String ***)
   fun toString (i,p) = Real.toString(toReal(i,p))
   fun fmtString (i,p) =
      let val str = Real.fmt (StringCvt.FIX NONE) (toReal(i,p))
      in String.map (fn #"~" => #"-" | c => c) str end

   (*** Self adjusting stuff ***)
   val eq = Eq.default
   val hash = Hash.default
   val key = Key.default

end (* ExactFloatingPoint *)



structure ApproximateReal : BINARY_NUMBER =
struct
   open Real

   type t = real

   (*** Zeroes of addition and multiplication ***)
   val zero = 0.0
   val one = 1.0

   (*** Conversion ***)
   fun fromReal r = r
   fun toReal r = r
   val toInt = (toInt IEEEReal.TO_ZERO)

   (*** Extended arithmetic ***)
   fun shift (x, s) = if Real.==(x, zero) then zero else
      let
         val {exp, man} = toManExp x
      in
         fromManExp {exp = Int.+(exp, s), man = man}
      end

   val op div = realTrunc o (op /)
   fun op mod (x, y) = x - (x div y) * y
   fun gcd (x, y) = if ==(y, zero) then abs(x) else gcd(y, x mod y)

   (*** Restriction ***)
   fun restrictionAt (f : t -> t) (x : t, p : word) =
      let
         val s = Word.toInt p
      in
         shift(f (shift(x, s)), Int.~ s)
      end

   val truncate = restrictionAt realTrunc

   val floor = restrictionAt realFloor

   val ceiling = restrictionAt realCeil
```

```
   (*** String ***)
   fun fmtString x =
      let val str = fmt (StringCvt.FIX NONE) x
      in String.map (fn #"~" => #"-" | c => c) str end

   (*** Self adjusting stuff ***)
   val eq = Eq.real
   val hash = Hash.real
   val key = Key.real

end (* ApproximateReal *)



functor Interval (structure Number : BINARY_NUMBER) : INTERVAL =
struct
   structure Number = Number
   structure N = Number

   (*** Boundary type used in representing intervals ***)
   datatype b = POS_INFTY | NEG_INFTY | OPEN of N.t | CLOSED of N.t

   (*** Interval type consists of a tuple of boundaries ***)
   type t = b * b


   (*** Conversion ***)
   fun fromReal (x, n, leftOpen, rightOpen) =
      let
         val delta = N.shift(N.one, ~(Word.toInt n))
         val x = N.fromReal x
         val y = N.truncate(x, n)
      in
         (if leftOpen then OPEN (if N.<(y, x) then y else N.-(y, delta))
          else CLOSED (if N.<=(y, x) then y else N.-(y, delta)),
          if rightOpen then OPEN (if N.>(y, x) then y else N.+(y, delta))
          else CLOSED (if N.>=(y, x) then y else N.+(y, delta)))
      end

   fun toNumber (OPEN x, OPEN y) = (x, y)
     | toNumber (OPEN x, CLOSED y) = (x, y)
     | toNumber (CLOSED x, OPEN y) = (x, y)
     | toNumber (CLOSED x, CLOSED y) = (x, y)
     | toNumber _ = raise Fail "Unbounded interval!"


   (*** Boundary equality and comparisons ***)
   fun equal (OPEN x, OPEN y) = N.==(x, y)
     | equal (CLOSED x, CLOSED y) = N.==(x, y)
     | equal (NEG_INFTY, NEG_INFTY) = true
     | equal (POS_INFTY, POS_INFTY) = true
     | equal _ = false

   fun inequal (x, y) = not (equal (x, y))

   (*** [(..., x)] ?< [(y, ...)] ***)
   fun less (OPEN x, OPEN y) = N.<=(x, y)
     | less (OPEN x, CLOSED y) = N.<=(x, y)
     | less (CLOSED x, OPEN y) = N.<=(x, y)
     | less (CLOSED x, CLOSED y) = N.<(x, y)
     | less _ = false

   (*** [(x, ...)] ?<= [(y, ...)] ***)
   fun lessEq (OPEN x, OPEN y) = N.<=(x, y)
     | lessEq (OPEN x, CLOSED y) = N.<(x, y)
     | lessEq (CLOSED x, OPEN y) = N.<=(x, y)
```

```
      | lessEq (CLOSED x, CLOSED y) = N.<=(x, y)
      | lessEq (NEG_INFTY, _) = true
      | lessEq _ = false

   (*** [(..., x)] ?<= [(..., y)] ***)
   fun greaterEq (OPEN x, OPEN y) = N.>=(x, y)
     | greaterEq (OPEN x, CLOSED y) = N.>(x, y)
     | greaterEq (CLOSED x, OPEN y) = N.>=(x, y)
     | greaterEq (CLOSED x, CLOSED y) = N.>=(x, y)
     | greaterEq (POS_INFTY, _) = true
     | greaterEq _ = false


   (*** Equality ***)
   fun op == ((x0, x1) : t, (y0, y1) : t) = equal(x0, y0) andalso equal(x1, y1)

   fun op != (x : t, y : t) = not (==(x, y))


   (*** Comparison ***)
   fun op < ((_, r) : t, (l, _) : t) = less(r, l)

   fun op <= ((r0, _) : t, (r1, _) : t) = lessEq(r0, r1)

   fun op > ((l, _) : t, (_, r) : t) = less(r, l)

   fun op >= ((_, l0) : t, (_, l1) : t) = greaterEq(l0, l1)

   fun compare (interval : t, interval' : t) =
      if interval < interval' then LESS
      else if interval > interval' then GREATER
      else EQUAL


   (*** Operations ***)
   fun length (interval : t) =
      let
         val (l, r) = toNumber interval
      in
         N.-(r, l)
      end

   fun intersect (interval as (l, r) : t, interval' as (l', r') : t) =
      let
         val left = if interval <= interval' then l' else l
         val right = if interval >= interval' then r' else r
      in
         if less(right, left) then NONE else SOME(left, right)
      end


   (*** String ***)
   fun toString ((l, r) : t) =
      (case l of
         OPEN x => "(" ^ (N.toString x)
       | CLOSED x => "[" ^ (N.toString x)
       | NEG_INFTY => "(-infty"
       | POS_INFTY => raise Fail "Left boundary is positive infinity")
      ^ ", " ^
      (case r of
         OPEN x => (N.toString x) ^ ")"
       | CLOSED x => (N.toString x) ^ "]"
       | POS_INFTY => "+infty)"
       | NEG_INFTY => raise Fail "Right boundary is negative infinity")

end (* Interval *)
```

A.1.2 Polynomials

In the polynomials library, we implement the polynomial ring structure and an extended certificate polynomial structure for generating certificates as described in the kinetic data structures framework. We use Sturm sequences to identify the roots of a polynomial at a given resolution.

```
signature POLYNOMIAL_RING =
sig
   include RING

   structure Coefficient : BINARY_NUMBER

   type c = Coefficient.t

   (*** Operations ***)
   val scale : t * c -> t

   (*** Extended arithmetic ***)
   (*** Given two polynomials p(x) and q(x), this function computes ***
    *** (c, qt(x), rm(x)) such that c * p(x) = qt(x) * q(x) + rm(x) ***)
   val divMod : t * t -> c * t * t

   (*** Information ***)
   val degree : t -> int
   val evaluate : t -> c -> c

   (*** Construction/Conversion ***)
   val fromList : c list -> t
   val toList : t -> c list

end (* POLYNOMIAL_RING *)



signature CERTIFICATE_POLYNOMIAL =
sig
   include POLYNOMIAL_RING

   (*** A polynomial should not be constant zero! ***)
   exception ZeroPolynomial

   (*** Operations ***)
   val derivative : t -> t

   (*** Evaluation ***)
   val evaluateSignRight : t -> Coefficient.t -> bool
   val evaluateSignLeft : t -> Coefficient.t -> bool

end (* CERTIFICATE_POLYNOMIAL *)



(** Structure for a polynomial (ring) whose coefficients are from a given
 ** Coefficient set. The polynomial a_n x^n + ... + a_1 x + a_0 is stored
 ** as the list of Coefficients [a_0, a_1, ..., a_n]. **)

functor PolynomialRing (structure Coefficient : BINARY_NUMBER)
                       : POLYNOMIAL_RING =
```

```
struct
   structure Coefficient = Coefficient
   structure C = Coefficient

   type c = C.t
   type t = c list

   (*** Zeroes of addition and multiplication ***)
   val zero = [C.zero]
   val one = [C.one]


   (*** IMPORTANT: Input must be in reverse order. ***)
   fun normalize nil = zero
     | normalize p = if (C.==(hd p, C.zero)) then normalize (tl p) else p


   (*** (Arithmetic) Operations ***)
   fun op + (p : t, q : t) =
      let
         fun plus (p, q, revSum) =
            case (p, q) of
               (a::tailP, b::tailQ) => plus(tailP, tailQ, (C.+(a, b))::revSum)
             | (nil, nil) => normalize revSum
             | (_, nil) => List.revAppend(p, revSum)
             | (nil, _) => List.revAppend(q, revSum)
      in
         List.rev (plus (p, q, nil))
      end

   fun op - (p : t, q : t) =
      let
         fun minus (p, q, revSub) =
            case (p, q) of
               (a::tailP, b::tailQ) => minus(tailP, tailQ, (C.-(a, b))::revSub)
             | (nil, nil) => normalize revSub
             | (_, nil) => List.revAppend(p, revSub)
             | (nil, _) => List.foldl (fn (a, sub) => (C.~(a))::sub) revSub q
      in
         List.rev (minus (p, q, nil))
      end

   fun op ~ (p : t) = List.map (C.~) p

   fun op * (p : t, q : t) =
      let
         fun termMult (a, p) = List.map (fn b => C.*(a, b)) p
         fun termAdd (p, q) =
            case (p, q) of
               (a::tailP, b::tailQ) => (C.+(a, b))::termAdd(tailP, tailQ)
             | (_, nil) => p
             | (nil, _) => q

         fun mult (p, q, sum) =
            case p of
               a::tail =>
                  let
                     val newSum = termAdd(sum, termMult(a, q))
                  in
                     (hd newSum)::(mult (tail, q, tl newSum))
                  end
             | nil => sum
      in
         mult (p, q, nil)
      end
```

```
fun scale (p : t, c : C.t) = List.map (fn a => C.*(a, c)) p


(*** Equality ***)
fun op == (a::tailP, b::tailQ) = (C.==(a, b)) andalso (==(tailP, tailQ))
  | op == (nil, b::tailQ) = (C.==(b, C.zero)) andalso (==(nil, tailQ))
  | op == (a::tailP, nil) = (C.==(a, C.zero)) andalso (==(tailP, nil))
  | op == (nil, nil) = true

fun op != (p : t, q : t) = not (==(p, q))


(*** Information ***)
fun degree p = Int.-(List.length p, 1)
fun evaluate p x = List.foldr (fn (a, s) => C.+(a, C.*(x, s))) (C.zero) p


(*** Extended arithmetic ***)
(*** Given two polynomials p(x) and q(x), this function computes ***
 *** (c, qt(x), rm(x)) such that c * p(x) = qt(x) * q(x) + rm(x) ***)
fun divMod (p : t, q : t) =
   let
      fun weightedSub (nil, _, _, _) = nil
        | weightedSub (p, nil, wP, _) = List.map (fn a => C.*(a, wP)) p
        | weightedSub (a::p, b::q, wP, wQ) =
         (C.-(C.*(a, wP), C.*(b, wQ)))::(weightedSub(p, q, wP, wQ))

      val diff = Int.-(degree p, degree q)
      val (p, q) = (List.rev p, List.rev q)

      fun divMod (p, diff) =
         if diff < 0 then (C.one, nil, p) else
            let
               val (a, b) = (hd p, hd q)
               val d = C.gcd(a, b)
               val (multP, multQ) = (C.div(b, d), C.div(a, d))
               val sub = tl (weightedSub(p, q, multP, multQ))
               val (c, quotient, remainder) = divMod(sub, Int.-(diff, 1))
            in
               (C.*(c, multP), (C.*(c, multQ))::quotient, remainder)
            end

      val (c, qt, rm) = divMod(p, diff)
   in
      (c, List.rev qt, List.rev rm)
   end


(*** Conversions ***)

(*** The polynomial a_n x^n + ... + a_1 x + a_0 is given as
 *** [a_n, ..., a_1, a_0] and stored as [a_0, a_1, ..., a_n] ***)
fun fromList c = List.rev (normalize c)

fun toList c = List.rev c

(*** Input to toString has the internal order, but output is reverse ***)
fun toString p =
   let
      fun polyToString (nil, _, str) = str
        | polyToString (a::tail, degree, str) =
          let
             val term = C.toString a
             val str = if degree > 1 then
                         (term ^ " x^" ^ (Int.toString degree) ^ " + " ^ str)
                       else if degree = 1 then (term ^ " x + " ^ str)
```

```
                        else term
            in
               polyToString (tail, Int.+(degree, 1), str)
            end
      in
         polyToString (p, 0, "")
      end

   fun fmtString p =
      let
         fun polyFmtString (nil, _, str) = str
           | polyFmtString (a::tail, degree, str) =
            let
               val term = C.fmtString a
               val str = if degree > 1 then
                           (term ^ " * x ** " ^ (Int.toString degree) ^ " + " ^ str)
                         else if degree = 1 then (term ^ " * x + " ^ str)
                         else term
            in
               polyFmtString (tail, Int.+(degree, 1), str)
            end
      in
         polyFmtString (p, 0, "")
      end


   (*** Self adjusting stuff ***)
   val eq = Eq.list C.eq
   val hash = Hash.list C.hash
   val key = Key.list C.key

end (* PolynomialRing *)



functor CertificatePolynomial (structure Coefficient : BINARY_NUMBER)
                           : CERTIFICATE_POLYNOMIAL =
struct
   structure P = PolynomialRing (structure Coefficient = Coefficient)
   open P

   structure C = Coefficient

   exception ZeroPolynomial


   (*** Operations ***)
   fun derivative p =
      let
         fun prime (nil, _) = nil
           | prime (a::tail, degree) =
            let
               val primeTail = prime(tail, C.+(degree, C.one))
            in
               if C.==(degree, C.zero) then primeTail
               else (C.*(degree, a))::primeTail
            end
      in
         case (prime (p, C.zero)) of nil => zero | p' => p'
      end

   (** If evaluation is zero at x, find the sign of the gradient **)
   fun gradientSign (p, x) =
      let
         fun gradientSign p =
            case C.sign(evaluate p x) of
```

```
               ~1 => false
             | 1 => true
             | _ => gradientSign(derivative p)
      in
         gradientSign p
      end

   (*** Information ***)
   fun evaluateSignRight p x =
      if P.==(p, zero) then raise ZeroPolynomial else (gradientSign(p, x))

   fun evaluateSignLeft p x =
      let
         fun pMinusX (nil, _) = nil
           | pMinusX (a::tail, degree) =
            (if (Int.rem(degree, 2) = 1) then (C.~ a) else a)::
            (pMinusX (tail, Int.+(degree, 1)))
      in
         if P.==(p, zero) then raise ZeroPolynomial else
            case C.sign(evaluate p x) of
               ~1 => false
             | 1 => true
             | _ => gradientSign(pMinusX(derivative p, 1), C.~ x)
      end

end (* CertificatePolynomial *)



signature STURM_SEQUENCE =
sig
   structure Polynomial : CERTIFICATE_POLYNOMIAL

   (*** Sturm sequence ***)
   type s

   (*** Sturm sequence with sigma values at minus and plus infinity ***)
   type t = {sequence : s, left : int, right : int}

   (*** Information ***)
   val squareFreePolynomial : s -> Polynomial.t
   val sigma : s * Polynomial.c -> int

   (*** Construction ***)
   val makeSequence : Polynomial.t -> t

end (* STURM_SEQUENCE *)



signature ROOTS =
sig
   structure Polynomial : CERTIFICATE_POLYNOMIAL
   structure Interval : INTERVAL where type Number.t = Polynomial.c

   (*** t is an interval with necessary information for further refinement ***)
   type t

   (*** Conversion ***)
   val boundaries : t -> Interval.t
   val signs : t -> bool * bool
   (*val numberOfRoots : t -> int*)

   (*** Construction/Modification ***)
   (*** Given a polynomial p and a maximum value m, createInterval returns ***
```

```
    *** three optional intervals (-infty, -m], (-m, +m], and (+m, +infty). ***
    *** NONE implies that there is no root in the corresponding interval.  ***)
   val createInterval : Polynomial.t * Polynomial.c ->
                        t option * t option * t option

   (*** Given an interval and a point x (guaranteed to be) inside the       ***
    *** interval, splitInterval computes the necessary information to be    ***
    *** stored in the left and right intervals and returns these intervals. ***)
   val splitInterval : t * Polynomial.c -> t option * t option

   (*** Similarly, triSplitInterval trisects the given interval at l and r. ***)
   val triSplitInterval : t * Polynomial.c * Polynomial.c ->
                       t option * t option * t option

   (*** Given an interval, a guess for a root and a precision, verifyRoot ***
    *** checks whether the lattice interval containing the guess indeed   ***
    *** has a root. Left of the root, the root interval and right of the  ***
    *** root are returned.                                                ***)
   val verifyRoot : t * real * word -> t option * t option * t option

   (*** Given an interval and a precision, iterate refines the interval ***
    *** using the root finding algorithm for one iteration.             ***)
   val iterate : t * word -> t option * t option

end (* ROOTS *)



signature ROOT_SOLVER =
sig
   structure Roots : ROOTS

   (*** Given a polynomial p, a maximum value m, and the significant number ***
    *** of bits n after the radix point, this function enumerates all real  ***
    *** roots x of p satisfying |x| < m. Approximation error is <= 2^-n.    ***)
   val allRealRoots : Roots.Polynomial.t * Roots.Polynomial.c * word ->
                      Roots.t list

end (* ROOT_SOLVER *)



functor SturmSequence (structure Polynomial : CERTIFICATE_POLYNOMIAL)
                      : STURM_SEQUENCE =
struct
   structure Polynomial = Polynomial
   structure P = Polynomial
   structure C = P.Coefficient

   (*** Sequence type is a list of polynomials ***)
   type s = P.t list

   (*** Main type is a Sturm sequence accompanied with computed values of ***
    *** sigma at left and right boundaries. (default = (-infty, +infty))  ***)
   type t = {sequence : s, left : int, right : int}


   (*** Information ***)

   (*** Given a sequence of polynomials, {p_i}, this function counts ***
    *** the number of sign changes in f(p_1), f(p_2), ...            ***)
   fun signChanges (sequence : s, f : P.t -> int) =
      let
         fun countChanges (p, (count, sign)) =
            case f(p) of
```

```
           ~1 => (if sign = 1 then count + 1 else count, ~1)
         | 1 => (if sign = ~1 then count + 1 else count, 1)
         | _ => (count, sign)
   in
      #1 (List.foldl countChanges (0, 0) sequence)
   end

(*** Sigma function of a Sturm sequence ***)
fun sigma (sequence : s, x : P.c) =
   signChanges(sequence, fn p => C.sign(P.evaluate p x))

(*** Returns the square-free polynomial of the original polynomial. ***)
fun squareFreePolynomial (sequence : s) = hd sequence


(*** Construction ***)

(*** Given a polynomial p(x) this function computes gcd(p(x), p'(x)) ***
 *** and the chain of polynomials p(x), p'(x), -rem(p(x)/p'(x)), ... ***)
fun makeChain (p : P.t) =
   let
      fun chain (p, q) =
         if P.==(q, P.zero) then [p] else
            let
               val (c, _, r) = P.divMod(p, q)
            in
               p::(chain(q, if C.sign(c) = 1 then P.~ r else r))
            end
   in
      chain(p, P.derivative p)
   end

(*** Optimized Sturm sequence computation for quadratic polynomials ***)
fun quadraticSequence (p : P.t) =
   let
      fun coefficients [a, b, c] = (a, b, c)
        | coefficients _ = raise Fail "Not quadratic"
      val (a, b, c) = coefficients(P.toList p)
      (*** Computing the sign of the discriminant ***)
      val delta = C.sign(C.-(C.*(b, b), C.*(C.fromInt(4), C.*(a, c))))
   in
      (*** delta < 0 => There are no real roots of p(x) ***)
      if delta < 0 then {sequence = nil, left = 0, right = 0} else
         let
            val pA = P.fromList [a]
            val p' = P.derivative p
         in
            (*** delta = 0 => Double root, it is the root of p'(x) ***)
            if delta = 0 then {sequence = [p', pA], left = 1, right = 0}
            (*** delta > 0 => Two distinct real roots ***)
            else {sequence = [p, p', pA], left = 2, right = 0}
         end
   end

(*** Optimized Sturm sequence computation for cubic polynomials ***)
fun cubicSequence (p : P.t) =
   let
      fun coefficients [a, b, c, d] = (a, b, c, d)
        | coefficients _ = raise Fail "Not cubic"
      val (a, b, c, d) = coefficients(P.toList p)

      val pA = P.fromList [C.fromInt(C.sign(a))]
      val p' = P.derivative p
      val two = C.fromInt(2)
      val three = C.fromInt(3)
      val nine = C.fromInt(9)
```

```
         (*** Remainder of division of (9a p(x)) by p'(x) is -(mx + n) ***)
         val m = C.*(two, C.-(C.*(b, b), C.*(three, C.*(a, c))))
         val n = C.-(C.*(b, c), C.*(nine, C.*(a, d)))
         val q = P.*(pA, P.fromList[m, n])
      in
         case C.sign(m) of
            0 => (case C.sign(n) of
              ~1 => {sequence = [p, p', P.~ pA], left = 2, right = 1}
            | 1 => {sequence = [p, p', pA], left = 1, right = 0}
            | _ => {sequence = [P.fromList[C.*(three, a), b], pA], left = 1,
                                                                   right = 0})
          | 1 =>
            let
               val temp = C.*(two, C.-(C.*(c, c), C.*(three, C.*(b, d))))
               val delta = C.sign(C.-(C.*(m, temp), C.*(n, n)))
            in
               case delta of
                ~1 => {sequence = [p, p', q, P.~ pA], left = 2, right = 1}
               | 1 => {sequence = [p, p', q, pA], left = 3, right = 0}
                      (*** TODO: compute this directly! ***)
               | _ => quadraticSequence(#2 (P.divMod(p, q)))
            end
          (*** sgn(delta) = ~1 ***)
          | _ => {sequence = [p, p', q, P.~ pA], left = 2, right = 1}
      end


   (*** Assuming p(x) = r_1(x) * r_2^2(x) * r_3^3(x) * ..., for square-free ***
    *** polynomials r_1(x), r_2(x), r_3(x), ..., this function computes the ***
    *** Sturm sequence of the square-free polynomial p(x) / gcd(p(x),p'(x)) ***
    *** which is equal to r_1(x) * r_2(x) * r_3(x) * ...                    ***)
   fun makeSequence (p : P.t) =
      case P.degree p of
         0 => {sequence = nil, left = 0, right = 0}
       | 1 => {sequence = [p, P.derivative p], left = 1, right = 0}
       | 2 => quadraticSequence p
       | 3 => cubicSequence p
       | _ =>
         let
            (*** chain is the sequence of polynomials consisting of the   ***
             *** remainders in the computation of d(x) = gcd(p(x), p'(x)) ***)
            val chain = makeChain p
            val d = List.last chain
            (*** If gcd is a (non-zero) constant, chain is a valid sequence ***)
            val sequence = if (P.degree d) = 0 then chain
            (*** Otherwise, use the sequence generated by p(x)/d(x) ***)
                           else makeChain(#2 (P.divMod(p, d)))

            fun signAtPlusInfty p = C.sign(hd (P.toList p))

            fun signAtMinusInfty p = (signAtPlusInfty p) *
                                     (if (P.degree p) mod 2 = 1 then ~1 else 1)
         in
            {sequence = sequence,
             left = signChanges(sequence, signAtMinusInfty),
             right = signChanges(sequence, signAtPlusInfty)}
         end

end (* SturmSequence *)



functor RootsBisection (structure Polynomial : CERTIFICATE_POLYNOMIAL
                        val IgnoreDoubleRoots : bool) : ROOTS =
```

```
struct
   structure Polynomial = Polynomial
   structure P = Polynomial
   structure S = SturmSequence(structure Polynomial = P)

   structure C = P.Coefficient
   structure Interval = Interval(structure Number = C)
   structure I = Interval


   (*** The polynomial with sign values at each end (boundary + epsilon) ***)
   type poly = {polynomial : P.t, left : bool, right : bool}

   (*** Root type contains a basic interval and an optional Sturm sequence ***)
   type t = I.t * poly * S.t option


   (*** Conversion ***)
   fun boundaries (interval : t) = #1 interval

   fun signs ((_, {left, right, ...}, _) : t) = (left, right)

   fun numberOfRoots ((_, _, NONE) : t) = 1
     | numberOfRoots ((_, _, SOME{left, right, ...}) : t) = left - right


   (*** Construction ***)
   (*** This function decides whether to store the Sturm sequence, ***
    *** as well as whether to store the interval at all or not.    ***)
   fun intervalSingleRoot (interval : I.t, info : poly, seq : S.t) =
      if (#left info) = (#right info) then
         if IgnoreDoubleRoots then NONE else SOME(interval, info, SOME(seq))
      else SOME(interval, info, NONE)

   (*** This function computes sigma values and signs at -infty and +infty ***
    *** and returns the interval (-infty, +infty), if there exists a root. ***)
   fun infiniteInterval (p : P.t) =
      let
         val sequence = S.makeSequence p
         val numberOfRoots = (#left sequence) - (#right sequence)
      in
         if numberOfRoots = 0 then NONE else
            let
               val interval = (I.NEG_INFTY, I.POS_INFTY)
               val signRight = C.>(hd(P.toList p), C.zero)
               val signLeft = signRight <> ((P.degree p) mod 2 = 1)
               val info = {polynomial = p, left = signLeft, right = signRight}
            in
               if numberOfRoots = 1 then
                  intervalSingleRoot(interval, info, sequence)
               else SOME(interval, info, SOME(sequence))
            end
      end

   (*** This function splits a given interval into two at x. ***)
   fun splitInterval ((interval, info, sequence) : t, x : P.c) =
      let
         val leftInt = (#1 interval, I.CLOSED x)
         val rightInt = (I.OPEN x, #2 interval)

         val {polynomial = p, left, right} = info
         val middle = P.evaluateSignRight p x

         val leftInfo = {polynomial = p, left = left, right = middle}
         val rightInfo = {polynomial = p, left = middle, right = right}
      in
```

```
         case sequence of
            NONE => if middle=left then (NONE, SOME(rightInt, rightInfo, NONE))
                    else (SOME(leftInt, leftInfo, NONE), NONE)
          | SOME {sequence = seq, left, right} =>
            let
               val middle = S.sigma(seq, x)
               val leftSeq = {sequence = seq, left = left, right = middle}
               val rightSeq = {sequence = seq, left = middle, right = right}
               val leftRoots = left - middle
               val rightRoots = middle - right
            in
               if leftRoots = 0 then
                  (NONE, SOME(rightInt, rightInfo, SOME rightSeq))
               else if rightRoots = 0 then
                  (SOME(leftInt, leftInfo, SOME leftSeq), NONE)
               else
                  (if leftRoots > 1 then SOME(leftInt, leftInfo, SOME leftSeq)
                   else intervalSingleRoot(leftInt, leftInfo, leftSeq)
                  ,if rightRoots > 1 then SOME(rightInt,rightInfo,SOME rightSeq)
                   else intervalSingleRoot(rightInt, rightInfo, rightSeq))
            end
      end

   fun triSplitInterval (interval : t, l : P.c, r : P.c) =
      case splitInterval(interval, r) of
         (NONE, right) => (NONE, NONE, right)
       | (SOME midLeft, right) =>
         (case splitInterval(midLeft, l) of
            (NONE, mid) => (NONE, mid, right)
          | (left, mid) => (left, mid, right))

   fun createInterval (p : P.t, m : P.c) =
      case infiniteInterval p of
         NONE => (NONE, NONE, NONE)
       | SOME interval => triSplitInterval(interval, C.~ m, m)

   fun verifyRoot (interval : t, root : real, precision : word) =
      let
         val rootInt = I.fromReal(root, precision, true, false)
      in
         case I.intersect(boundaries interval, rootInt) of
            NONE => (NONE, NONE, SOME interval)
          | SOME(I.OPEN l, I.CLOSED r) => triSplitInterval(interval, l, r)
          | _ => raise Fail "Expecting half open boundary (x, y]."
      end

   fun iterate (roots : t, precision : word) =
      case boundaries roots of
         (I.OPEN l, I.CLOSED r) =>
            splitInterval(roots, C.truncate(C.shift(C.+(l, r), ~1), precision))
       | _ => raise Fail "Expecting half open boundary (x, y]."

end (* RootsBisection *)



functor RootSolver (structure Roots : ROOTS) : ROOT_SOLVER =
struct
   structure Roots = Roots
   structure R = Roots
   structure P = R.Polynomial
   structure C = P.Coefficient

   structure AC = ApproximateReal
   structure AP = CertificatePolynomial(structure Coefficient = AC)
```

```
(*** Optimized root solvers for polynomials of degree < 4. ***)
fun linearSolver [a, b] = [~b / a]
  | linearSolver _ = raise Fail "Not linear"

fun quadraticSolver [a, b, c] =
   let
      val determinant = b * b - 4.0 * a * c
   in
      case Real.compare(determinant, 0.0) of
         LESS => []
       | EQUAL => [~0.5 * b / a]
       | GREATER =>
         let
            val temp = ~0.5 * (b + (Real.Math.sqrt determinant))
            val root = temp / a
            val root' = c / temp
         in
            if root > root' then [root', root] else [root, root']
         end
   end
  | quadraticSolver _ = raise Fail "Not quadratic"

fun cubicSolver [a, b, c, d] =
   let
      (*** a x^3 + b x^2 + c x + d ==> x^3 + a x^2 + b x + c ***)
      val (a, b, c) = (b / a, c / a, d / a)

      (*** x = t - a/3 ==> x^3 + a x^2 + b x + c = t^3 + p t + q ***)
      val lambda = a / 3.0
      val P = a * a - 3.0 * b (*** P = -3p ***)
      val Q = (2.0 * a * a - 9.0 * b) * a + 27.0 * c (*** Q = 27q ***)

      val roots =
         if Real.==(P, 0.0) then (*** One real root ***)
            if Q > 0.0 then [~1.0 * Real.Math.pow(Q / 27.0, 1.0 / 3.0)]
            else [Real.Math.pow(~Q / 27.0, 1.0 / 3.0)]
         else let (*** Delta = (Q2 - FourP3)/2916 ***)
            val FourP3 = 4.0 * P * P * P (*** -108 p^3 ***)
            val Q2 = Q * Q (*** 729 q^2 ***)
         in case Real.compare(Q2, FourP3) of
            LESS => (*** Delta < 0, three real roots ***)
            let
               val theta = Real.Math.acos(Q / Real.Math.sqrt(FourP3))
               val m = Real.Math.sqrt(P) / ~1.5
               val r0 = m * Real.Math.cos(theta / 3.0)
               val r1 = m * Real.Math.cos((theta - 2.0 * Real.Math.pi) / 3.0)
               val r2 = m * Real.Math.cos((theta + 2.0 * Real.Math.pi) / 3.0)
            in
               [r0, r1, r2]
            end
          | EQUAL => (*** Delta = 0, two real roots ***)
            let
               val sqrt = Real.Math.sqrt(P / 9.0)
            in
               if Q > 0.0 then [~2.0 * sqrt, sqrt] else [~sqrt, 2.0 * sqrt]
            end
          | GREATER => (*** Delta > 0, one real root ***)
            let
               val temp = Real.abs(Q) + Real.Math.sqrt(Q2 - FourP3)
               val A = Real.Math.pow(temp / 54.0, 1.0 / 3.0)
               val root = A + P / 9.0 / A
            in
               if Q > 0.0 then [~root] else [root]
            end
         end
   in
```

```
         List.map (fn r => r - lambda) roots
      end
    | cubicSolver _ = raise Fail "Not cubic"


   fun approximateRoots (p : P.t) =
      let
         val coefficients = List.map C.toReal (P.toList p)
      in
         case P.degree p of
            1 => (linearSolver coefficients)
          | 2 => (quadraticSolver coefficients)
          | 3 => (cubicSolver coefficients)
          | _ => nil
            (*let
               val p = AP.fromList(List.map AC.fromReal coefficients)
            in end*)
      end

   fun allRealRoots (p : P.t, m : P.c, precision : word) =
      let
         val delta = C.shift(C.one, ~(Word.toInt precision))

         fun findRoots NONE = []
           | findRoots (SOME interval) =
            if C.>(R.Interval.length(R.boundaries interval), delta) then
               let
                  val (left, right) = R.iterate(interval, precision)
               in
                  (findRoots left) @ (findRoots right)
               end
            else [interval]

         fun verify (NONE, _) = []
           | verify (interval, []) = findRoots interval
           | verify (SOME interval, root::roots) =
            let
               val (left, mid, right) = R.verifyRoot(interval, root, precision)
            in
               (findRoots left) @ (findRoots mid) @ (verify(right, roots))
            end

         val (left, interval, right) = R.createInterval(p, m)
      in
         verify(interval, approximateRoots p)
      end

end (* RootSolver *)
```

A.1.3 Geometry

In the geometry library, we assume every coordinate to be a polynomial, either a constant polynomial or any single variable polynomial of degree ≥ 1. Using this coordinate structure we design the basics such as points and simplices, and construct the setup for applications in two or three dimensions.

```
signature COORDINATE =
sig
   (*** Coordinates can be any function that implements RING ***)
   include RING

   (*** Coordinate functions' domain (also range) is Domain ***)
   structure Domain : BINARY_NUMBER

   (*** Roots of a certificate fail inside an interval ***)
   structure Interval : INTERVAL where type Number.t = Domain.t

   (*** Result type for evaluating coordinates ***)
   type d = Domain.t

   (*** Certificates should not be derived from constant zero functions ***)
   exception ZeroFunction

   (*** The origin for evalCertificate, useful in Kinetic testing ***)
   val origin : d ref

   (*** Constant Function ***)
   val constant : d -> t

   (*** Evaluation ***)
   val evaluate : t -> d -> d


   (*** Dynamic Predicates ***)

   (*** Sign evaluation without creating a certificate (no roots computed) ***
    *** First parameter is the default result for zero evaluation          ***)
   val evalCertificate : bool option -> t -> bool

   (*** Comparison functions (no roots computed) ***)
   val compare : t * t -> order
   val < : t * t -> bool
   val <= : t * t -> bool
   val > : t * t -> bool
   val >= : t * t -> bool


   (*** Kinetic Certificate Creation ***)

   (*** Creating a certificate: word * word are the precision parameters   ***
                               bool option is the default result for zero ***)
   val makeCertificate : word * word -> bool option -> t ->
                         (*** The certificate is represented as a discrete  ***
                          *** boolean function (undefined in the intervals) ***)
                         bool * (Interval.t * bool) list

end (* COORDINATE *)



functor FixedCoordinate (structure Domain : BINARY_NUMBER) :
        COORDINATE where type t = Domain.t =
struct
   structure Domain = Domain
   structure Interval = Interval(structure Number = Domain)
   open Domain

   type d = Domain.t

   exception ZeroFunction

   (*** Defining origin, just to comply with the signature ***)
```

```
   val origin = ref Domain.zero

   (*** Constant Function ***)
   fun constant n = n

   (*** Evaluation ***)
   fun evaluate n = fn _ => n

   (*** Certificate evaluation ***)
   fun evalCertificate zeroEval n =
      case Domain.compare(n, Domain.zero) of
         EQUAL => (case zeroEval of
                     NONE => raise ZeroFunction
                   | SOME sign => sign)
       | GREATER => true
       | LESS => false

   (*** Certificate creation ***)
   fun makeCertificate _ zeroEval n = (evalCertificate zeroEval n, [])

end (* FixedCoordinate *)



functor PolynomialCoordinate (structure RootSolver : ROOT_SOLVER
                              val UseLeftGradient : bool) :
        COORDINATE where type t = RootSolver.Roots.Polynomial.t =
struct
   structure Roots = RootSolver.Roots
   structure Domain = Roots.Polynomial.Coefficient
   structure Interval = Roots.Interval

   open Roots.Polynomial

   type d = Domain.t

   exception ZeroFunction

   (*** Defining origin to be used in evalCertificate ***)
   val origin = ref Domain.zero

   fun constant n = fromList [n]

   (*** Certificate evaluation ***)
   fun evalInclude x p = Domain.>=(evaluate p x, Domain.zero)
   fun evalExclude x p = Domain.>(evaluate p x, Domain.zero)
   fun evalLeft x p = evaluateSignLeft p x
   fun evalRight x p = evaluateSignRight p x

   fun evalCertificate zeroEval =
      (case zeroEval of
         SOME true => evalInclude
       | SOME false => evalExclude
       | NONE => if UseLeftGradient then evalLeft else evalRight) (!origin)

   (*** Comparison Functions ***)
   fun compare (p, q) =
      let
         val diff = evaluate (p - q) (!origin)
      in
         if Domain.==(diff, Domain.zero) then EQUAL
         else if Domain.>(diff, Domain.zero) then GREATER else LESS
      end

   fun op > (p, q) = evalCertificate (SOME false) (p - q)
   fun op >= (p, q) = evalCertificate (SOME true) (p - q)
```

```
   fun op < (p, q) = evalCertificate (SOME false) (q - p)
   fun op <= (p, q) = evalCertificate (SOME true) (q - p)

   (*** Certificate creation ***)
   (*** Left: p(x) actually means p(x - epsilon) for infinitesimally
              small epsilon. In other words, the sign of the polynomial
              does not change in the interval (x - epsilon, x).
              Therefore, roots lie in the intervals of the form [x0, x1) ***)
   fun leftCertificate (_ : word * word) (_ : bool option) p =
      (evaluateSignLeft p Domain.zero, [] : (Interval.t * bool) list)

   (*** Right: Similarly, p(x) actually means p(x + epsilon).
               Therefore, roots lie in the intervals of the form (x0, x1] ***)
   fun rightCertificate (whole, fraction) zeroEval p =
      let
         val maximum = Domain.shift(Domain.one, Word.toInt whole)
         val roots = RootSolver.allRealRoots(p, maximum, fraction)
         val sign = if List.null roots then (evalCertificate zeroEval p)
                                       else #1(Roots.signs(hd roots))
         fun format r = (Roots.boundaries r, #2 (Roots.signs r))
      in
         (sign, List.map format roots)
      end

   val makeCertificate = if UseLeftGradient then leftCertificate
                                            else rightCertificate

   fun toString p = Domain.toString(evaluate p (!origin))

end (* PolynomialCoordinate *)
```

After implementing coordinates, now we define the tuple, simplex, point, and geometry signatures and structures.

```
signature TUPLE =
sig
   (*** Any d-dimensional tuple (d-tuple) type ***)
   type 'a tuple

   val dimension : int

   val unitTuple : unit tuple

   val map : ('a -> 'b) -> 'a tuple -> 'b tuple
   val fold : ('a * 'a -> 'a) -> 'a tuple -> 'a
   val all : ('a -> bool) -> 'a tuple -> bool
   val exists : ('a -> bool) -> 'a tuple -> bool

   val mapPair : ('a * 'b -> 'c) -> 'a tuple * 'b tuple -> 'c tuple
   val allPair : ('a * 'b -> bool) -> 'a tuple * 'b tuple -> bool
   val existsPair : ('a * 'b -> bool) -> 'a tuple * 'b tuple -> bool

   (*** String ***)
   val fmt : string * string * string -> ('a -> string) -> 'a tuple -> string
   val mapToString : ('a -> string) -> 'a tuple -> string

   (*** Self adjusting stuff ***)
   val eqTuple : 'a Eq.t -> 'a tuple Eq.t
   val hashTuple : 'a Hash.t -> 'a tuple Hash.t
   val keyTuple : 'a Key.t -> 'a tuple Key.t

end (* TUPLE *)
```

```
signature SIMPLEX =
sig
   include TUPLE

   (*** Base type is a coordinate (for points) or a point ***)
   type base

   (*** Main type is a d-tuple of the base type ***)
   type t = base tuple

   (*** Coordinate type ***)
   type c

   (*** Arithmetic ***)
   val + : t * t -> t
   val - : t * t -> t
   val ~ : t -> t
   val scale : t * c -> t

   (*** Equality ***)
   val == : t * t -> bool
   val != : t * t -> bool

   (*** String ***)
   val toString : t -> string
   val fmtString : t -> string

   (*** Self adjusting stuff ***)
   val eq : t Eq.t
   val hash : t Hash.t
   val key : t Key.t

end (* SIMPLEX *)



signature POINT =
sig
   (*** Main type is a d-tuple of coordinates ***)
   include SIMPLEX sharing type base = c

   (*** Arithmetic ***)
   val dot : t * t -> c
   val squareDistance : t * t -> c

end (* POINT *)



signature GEOMETRY =
sig
   include TUPLE

   structure Coordinate : COORDINATE

   structure Vertex : POINT where type c = Coordinate.t
                            where type 'a tuple = 'a tuple

   type input = real tuple

   type pair = Coordinate.t * Coordinate.t
   type ball = Vertex.t * Coordinate.t
   type circle = Vertex.t * Vertex.t tuple

   val origin : Vertex.t
```

```
   val NotImplemented : exn

   (*** Input generators ***)
   val randomInSquarePoint : real * real * Random.rand -> input
   val randomInCirclePoint : real * real * Random.rand -> input

   (*** Geometric functions ***)
   val normal : Vertex.t tuple -> Vertex.t
   val centerWithOrigin : Vertex.t tuple -> Vertex.t * Coordinate.t
   val center : circle -> Vertex.t * Coordinate.t

   (*** Certificates ***)
   val halfSpaceTest : Vertex.t tuple * Vertex.t -> Coordinate.t

   (*** Dynamic predicates and functions ***)
   val squareDistanceVertexRectangle : Vertex.t * pair tuple -> Coordinate.t

end (* GEOMETRY *)



structure Pair : TUPLE =
struct
   type 'a tuple = 'a * 'a

   val dimension = 2

   val unitTuple = ((), ())

   fun map f (x, y) = (f x, f y)
   fun fold f (x, y) = f (x, y)
   fun all f (x, y) = (f x andalso f y)
   fun exists f (x, y) = (f x orelse f y)

   fun mapPair f ((px, py), (qx, qy)) = (f (px, qx), f (py, qy))
   fun allPair f ((px, py), (qx, qy)) = (f (px, qx) andalso f (py, qy))
   fun existsPair f ((px, py), (qx, qy)) = (f (px, qx) orelse f (py, qy))

   (*** String ***)
   fun fmt (left, mid, right) str (x, y) = left ^ str x ^ mid ^ str y ^ right
   fun mapToString str = fmt ("(", ", ", ")") str

   (*** Self adjusting stuff ***)
   fun eqTuple eq = Eq.tuple2 (eq, eq)
   fun hashTuple hash = Hash.tuple2 (hash, hash)
   fun keyTuple key = Key.tuple2 (key, key)

end (* Pair *)



structure Triple : TUPLE =
struct
   type 'a tuple = 'a * 'a * 'a

   val dimension = 3

   val unitTuple = ((), (), ())

   fun map f (x, y, z) = (f x, f y, f z)
   fun fold f (x, y, z) = f(x, f(y, z))
   fun all f (x, y, z) = (f x andalso f y andalso f z)
   fun exists f (x, y, z) = (f x orelse f y orelse f z)
```

```
   fun mapPair f ((px, py, pz), (qx, qy, qz)) = (f(px,qx), f(py,qy), f(pz,qz))
   fun allPair f ((px, py, pz), (qx, qy, qz)) = (f(px,qx) andalso f(py,qy)
                                                 andalso f(pz,qz))
   fun existsPair f ((px, py, pz), (qx, qy, qz)) = (f(px,qx) orelse f(py,qy)
                                                    orelse f(pz,qz))

   (*** String ***)
   fun fmt (left, mid, right) str tpl =
      left ^ fold (fn (s, s') => s ^ mid ^ s') (map str tpl) ^ right
   fun mapToString str = fmt ("(", ", ", ")") str

   (*** Self adjusting stuff ***)
   fun eqTuple eq = Eq.tuple3 (eq, eq, eq)
   fun hashTuple hash = Hash.tuple3 (hash, hash, hash)
   fun keyTuple key = Key.tuple3 (key, key, key)

end (* Triple *)



functor Simplex (structure Point : POINT
                 structure Tuple : TUPLE) : SIMPLEX =
struct
   structure P = Point

   open Tuple

   type c = P.c
   type base = P.t
   type t = base tuple

   (*** Arithmetic ***)
   val op + = mapPair P.+
   val op - = mapPair P.-
   val op ~ = map P.~
   fun scale (s, c) = map (fn p => P.scale(p, c)) s

   (*** Equality ***)
   val op == = allPair P.==
   val op != = existsPair P.!=

   (*** String ***)
   val toString =
      fmt (Int.toString(Int.-(dimension,1)) ^ "Simplex[", ", ", "]") P.toString
   fun fmtString s = fold (op ^) (map P.fmtString s) ^ "\n"

   (*** Self adjusting stuff ***)
   val eq = eqTuple P.eq
   val hash = hashTuple P.hash
   val key = keyTuple P.key

end (* Simplex *)



functor Point (structure Coordinate : COORDINATE
               structure Tuple : TUPLE) : POINT =
struct
   structure Coordinate = Coordinate
   structure C = Coordinate

   open Tuple

   type c = Coordinate.t
   type base = c
```

```
   type t = base tuple

   (*** Arithmetic ***)
   val op + = mapPair C.+
   val op - = mapPair C.-
   val op ~ = map C.~
   fun scale (p, c) = map (fn x => C.*(x, c)) p
   val dot = (fold C.+) o (mapPair C.*)
   fun squareDistance (p, q) = let val d = p - q in dot (d, d) end

   (*** Equality ***)
   val op == = allPair C.==
   val op != = existsPair C.!=

   (*** String ***)
   val toString = mapToString C.toString
   val fmtString = fmt ("", " ", "\n") C.fmtString

   (*** Self adjusting stuff ***)
   val eq = eqTuple C.eq
   val hash = hashTuple C.hash
   val key = keyTuple C.key

end (* Point *)



functor Geometry (structure Coordinate : COORDINATE
                  structure Tuple : TUPLE) : GEOMETRY =
struct
   structure Coordinate = Coordinate
   structure Vertex = Point (structure Coordinate = Coordinate
                             structure Tuple = Tuple)

   structure C = Coordinate
   structure V = Vertex

   open Tuple

   type input = real tuple

   type pair = C.t * C.t
   type ball = V.t * C.t
   type circle = V.t * V.t tuple

   val origin = map (fn () => C.zero) unitTuple

   val NotImplemented = Fail ("This function is not implemented for " ^
                    Int.toString dimension ^ "D!")

   (*** Input generation ***)
   (*** Generates a random point in Square(O; max) / Square(O; max*alpha) ***)
   fun randomInSquarePoint (max, alpha, state) =
      let
         (*** Random Cartesian coordinates ***)
         fun randomCoordinate () = max * (alpha + (1.0 - alpha) *
                                                  (Random.randReal state))
      in
         map randomCoordinate unitTuple
      end

   (*** Generates a random point in Ball(O; r) / B(O; r * alpha) ***)
   fun randomInCirclePoint (r, alpha, state) = raise NotImplemented

   (*** Geometric functions ***)
   fun normal vertices = raise NotImplemented
```

```
   fun centerWithOrigin vertices = raise NotImplemented

   (*** WARNING : This function must be implemented in the extended      ***
    ***           structure since centerWithOrigin is dimension specific ***)
   fun center (v, vertices) =
      let
         val verticesShifted = map (fn u => V.-(u, v)) vertices
         val (origin, scale) = centerWithOrigin verticesShifted
      in
          (V.+(origin, V.scale(v, scale)), scale)
      end


   (*** Return value > 0 <==> p is on the positive half of the space ***)
   fun halfSpaceTest (vertices, p) = raise NotImplemented


   (*** Dynamic predicates and functions ***)
   (*** Distance from vertex to rectangle ***)
   fun squareDistanceVertexRectangle (vertex, pairs) =
      let
         fun distanceToInterval (x, (a, b)) =
            if C.>(a, x) then
               let val dist = C.-(a, x)
               in C.*(dist, dist) end
            else if C.>(x, b) then
               let val dist = C.-(x, b)
               in C.*(dist, dist) end
            else C.zero
      in
         fold C.+ (mapPair distanceToInterval (vertex, pairs))
      end

end (* Geometry *)
```

Finally, we implement the two and three dimensional geometry structures.

```
signature GEOMETRY_2D =
sig
   include GEOMETRY

   structure Edge : SIMPLEX where type c = Coordinate.t
                            where type base = Vertex.t
                            where type 'a tuple = 'a Pair.tuple

   (*** Certificates ***)
   val atLeftOf : Vertex.t * Vertex.t -> Coordinate.t
   val atRightOf : Vertex.t * Vertex.t -> Coordinate.t
   val largerProjection : Vertex.t -> Vertex.t * Vertex.t -> Coordinate.t

end (* GEOMETRY_2D *)



functor Geometry2D (structure Coordinate : COORDINATE) : GEOMETRY_2D =
struct
   structure Geometry = Geometry (structure Coordinate = Coordinate
                                  structure Tuple = Pair)
   open Geometry

   structure Edge = Simplex (structure Point = Vertex
                             structure Tuple = Pair)
```

```
   structure C = Coordinate
   structure V = Vertex
   structure E = Edge

   (*** Input generation ***)
   (*** Generates a random point in Ball(O; r) / B(O; r * alpha) ***)
   fun randomInCirclePoint (r, alpha, state) =
      let
         (*** Random cylindirical coordinates ***)
         val theta = 2.0 * Real.Math.pi * (Random.randReal state)
         val r = r * (alpha + (1.0 - alpha) * (Random.randReal state))

         (*** Convert to Cartesian coordinates ***)
         val x = r * (Real.Math.cos theta)
         val y = r * (Real.Math.sin theta)
      in
         (x, y)
      end

   (*** Geometric functions ***)
   fun normal (((x1, y1), (x2, y2)) : E.t) : V.t = (C.-(y1, y2), C.-(x2, x1))

   fun centerWithOrigin (v1 as (x1, y1), v2 as (x2, y2)) =
      let
         val norm1 = V.dot(v1, v1)
         val norm2 = V.dot(v2, v2)
         val x = C.-(C.*(y2, norm1), C.*(y1, norm2))
         val y = C.-(C.*(x1, norm2), C.*(x2, norm1))
         val two = C.constant(C.Domain.fromInt 2)
         val scale = C.*(two, C.-(C.*(x1, y2), C.*(y1, x2)))
      in
         ((x, y), scale)
      end

   fun center (v, vertices) =
      let
         val verticesShifted = map (fn u => V.-(u, v)) vertices
         val (origin, scale) = centerWithOrigin verticesShifted
      in
         (V.+(origin, V.scale(v, scale)), scale)
      end


   (*** Certificates ***)
   (*** Return value > 0 <==> u is at the left of v ***)
   fun atLeftOf ((xu, _) : V.t, (xv, _) : V.t) : C.t = C.-(xv, xu)

   (*** Return value > 0 <==> u is at the right of v ***)
   fun atRightOf ((xu, _) : V.t, (xv, _) : V.t) : C.t = C.-(xu, xv)

   (*** Return value > 0 <==> p is on the positive half of the line uv ***)
   fun halfSpaceTest ((u, v) : E.t, p : V.t) : C.t =
      let
         val (x1, y1) = V.-(u, p)
         val (x2, y2) = V.-(v, p)
      in
         C.-(C.*(x1, y2), C.*(x2, y1))
      end

   (*** Return value > 0 <==> u.vector > v.vector ***)
   fun largerProjection vector (u : V.t, v : V.t) : C.t =
      C.-(V.dot(u, vector), V.dot(v, vector))

end (* Geometry2D *)
```

```
signature GEOMETRY_3D =
sig
   include GEOMETRY

   structure Edge : SIMPLEX where type c = Coordinate.t
                            where type base = Vertex.t
                            where type 'a tuple = 'a Pair.tuple

   structure Face : SIMPLEX where type c = Coordinate.t
                            where type base = Vertex.t
                            where type 'a tuple = 'a Triple.tuple

end (* GEOMETRY_3D *)



functor Geometry3D (structure Coordinate : COORDINATE) : GEOMETRY_3D =
struct
   structure Geometry = Geometry (structure Coordinate = Coordinate
                                  structure Tuple = Triple)
   open Geometry

   structure Edge = Simplex (structure Point = Vertex
                             structure Tuple = Pair)
   structure Face = Simplex (structure Point = Vertex
                             structure Tuple = Triple)

   structure C = Coordinate
   structure V = Vertex
   structure E = Edge
   structure F = Face


   (*** Return value > 0 <==> p is on the positive half of the plane uvw ***)
   fun halfSpaceTest ((u, v, w) : F.t, p : V.t) : C.t =
      let
         val (x1, y1, z1) = V.-(u, p)
         val (x2, y2, z2) = V.-(v, p)
         val (x3, y3, z3) = V.-(w, p)

         val mz = C.*(z3, C.-(C.*(x1, y2), C.*(x2, y1)))
         val my = C.*(y3, C.-(C.*(z1, x2), C.*(z2, x1)))
         val mx = C.*(x3, C.-(C.*(y1, z2), C.*(y2, z1)))
      in
         C.+(C.+(mx, my), mz)
      end

end (* Geometry3D *)
```

A.2 3D Convex Hulls

We provide the ΔML code for the self-adjusting application signature and the application settings signatures. We then implement two functors for generating dynamic or kinetic versions of a given self-adjusting application. Using these settings, we implement our self-adjusting convex hull algorithm in ΔML. In the code, we use the term *adaptive* as a synonym for the term self adjusting.

Our implementation yields a dynamic algorithm when compiled with the dynamic settings, and a kinetic algorithm when compiled with the kinetic settings. Furthermore, we use an added feature of the ΔML library that allows any self-adjusting program to be compiled into a static algorithm. Therefore, our self-adjusting convex hull program can be compiled into three different versions: static, dynamic, and kinetic. In our experiments, we use the static version to check the correctness of the dynamic and kinetic versions.

A.2.1 Self-Adjusting Applications

We provide the generic adaptive application code and the generic adaptive dynamic and kinetic settings code. Since the nonadaptive version is generated automatically, we do not include the code for the nonadaptive version. However, one can think of the automatically generated nonadaptive code to be the same adaptive code where all ΔML specific annotations are dropped, hence, disallowing any tracing to be done during the execution.

```
signature APPLICATION_ADAPTIVE =
sig
   (*** Structure for application settings ***)
   structure Settings : APPLICATION_SETTINGS_ADAPTIVE

   (*** Input type ***)
   type it

   (*** Output type ***)
   type ot

   (*** Application main function ***)
   val run : it -$> ot

   val propagate : unit -> unit

end (* APPLICATION_ADAPTIVE *)



signature APPLICATION_SETTINGS_ADAPTIVE =
sig
   (*** Hash structure ***)
   structure Hash : HASH_ADAPTIVE

   (*** Key structure ***)
   structure Key : KEY_ADAPTIVE

   (*** Base structure for representing moving/fixed points or numbers. ***)
   structure Coordinate : COORDINATE
```

```
   (*** Evaluation function for geometric primitives (certificates) ***)
   val evaluate : bool option -> Coordinate.t -$> bool

   (*** Time for kinetic simulation ***)
   val nextFailure : unit -> Coordinate.d option
   val prevFailure : unit -> Coordinate.d option

   val getTime : unit -$> Coordinate.d
   val setTime : Coordinate.d -> unit

   val incTime : int -> unit
   val decTime : int -> unit

   val resetStats : unit -> unit
   val statsToString: unit -> string
   val printStats: unit -> unit

   val propagate : unit -> unit
   val changed : bool Adaptive.Box.t

   val printOutput : bool ref

end (* APPLICATION_SETTINGS_ADAPTIVE *)



functor MkDynamicSettingsAdaptive (structure Hash : HASH_ADAPTIVE
                                   structure Key : KEY_ADAPTIVE
                                   structure Coordinate : COORDINATE) :
        APPLICATION_SETTINGS_ADAPTIVE =
struct
   structure Hash = Hash
   structure Key = Key
   structure Coordinate = Coordinate

   fun evaluate zeroEval = afn c => Coordinate.evalCertificate zeroEval c

   fun nextFailure () = NONE
   fun prevFailure () = NONE

   afun getTime () = !Coordinate.origin
   fun setTime t = Coordinate.origin := t

   fun incTime c = ()
   fun decTime c = ()

   fun resetStats () = ()
   fun statsToString () = "No Application Specific Statistics"
   fun printStats () = print (statsToString ())

   val changed = Adaptive.Box.Meta.new true
   fun propagate () = (Adaptive.Box.Meta.change(changed, false);
                       Adaptive.Box.Meta.change(changed, true);
                       Adaptive.Meta.propagate(); ())

   val printOutput = ref false

end (* MkDynamicSettingsAdaptive *)



functor MkKineticSettings (structure Hash : HASH_ADAPTIVE
                           structure Key : KEY_ADAPTIVE
                           structure Coordinate : COORDINATE
```

```
                          val bits : word * word) :
       APPLICATION_SETTINGS_ADAPTIVE =
struct
   structure Hash = Hash
   structure Key = Key
   structure Coordinate = Coordinate

   structure Time = Coordinate.Domain

   structure MBox =
   struct
      structure IBox = Adaptive.IBox
      structure MBox = Adaptive.RetroModular

      datatype ('a,'b) t = T of {ibox: 'a IBox.t, mbox: ('a,'b) MBox.t}

      val eq = fn (T{mbox=x1,...}, T{mbox=x2,...}) => MBox.eq (x1,x2)
      val hash = fn T{mbox=x,...} => MBox.hash x
      val key = {eq=eq, hash=hash}

      structure Meta =
      struct
         val toString = fn f => fn T{mbox,...} => MBox.Meta.toString f mbox
         val new = fn (cmp,v) => T {ibox=IBox.Meta.new v, mbox=MBox.Meta.new (cmp,v)}
         val next = fn T{mbox,...} => MBox.Meta.next mbox
         val prev = fn T{mbox,...} => MBox.Meta.prev mbox
         val change = fn (T{ibox,mbox},v) =>
                         (IBox.Meta.change (ibox,v)
                        ; MBox.Meta.change (mbox,v))
         val deref = fn T{mbox,...} => MBox.Meta.deref mbox
         structure Stats =
         struct
            val reset = fn T{mbox,...} => MBox.Meta.Stats.reset mbox
            val get = fn T{mbox,...} => MBox.Meta.Stats.get mbox
         end
      end

      val get = afn T{ibox,...} => IBox.get $ ibox
      val map = afn (T{mbox,...},bounds) => MBox.map $ (mbox, bounds)
   end

   val epsilon = Time.shift(Time.one, ~(Word.toInt(#2 bits)))
   val time = MBox.Meta.new (Time.compare, Time.zero)
   val () = MBox.Meta.Stats.reset time

   fun evaluate zeroEval = afn c =>
      let
         val (signLeft, roots) =
            Coordinate.makeCertificate bits zeroEval c

         fun signRight (i, b) = (#2 (Coordinate.Interval.toNumber i), b)
      in
         MBox.map $ (time, Boundary.Discrete.Bounds
                           {first = signLeft, above = List.map signRight roots})
      end

   afun getTime () = MBox.get $ time

   fun nextFailure () = MBox.Meta.next time
   fun prevFailure () =
      case MBox.Meta.prev time of
         NONE => NONE
       | SOME t => SOME(Time.-(t, epsilon))

   val printOutput = ref false
```

```
   fun setTimePerPropagateStats t =
      if !printOutput then
         (MBox.Meta.Stats.reset time;
          Coordinate.origin := t;
          MBox.Meta.change(time, t))
      else
         let
            val {mapReadersInsert = i, mapReadersRemove = r, ...} =
                MBox.Meta.Stats.get time

            val () = print ("Certificates Deleted = " ^ Int.toString r ^ "\n")
            val () = print ("Certificates Inserted = " ^ Int.toString i ^"\n")
            val () = print ("Time = " ^ (Time.toString t) ^ "\n")

            val () = MBox.Meta.Stats.reset time
            val () = Coordinate.origin := t
            val () = MBox.Meta.change(time, t)

            val {mapReadersWakeUp = w, ...} = MBox.Meta.Stats.get time
            val () = print ("Certificates Changed = " ^ Int.toString w ^ "\n")
         in
            ()
         end

   fun setTimeGlobalStats t =
      if !printOutput then
         (Coordinate.origin := t;
          MBox.Meta.change(time, t))
      else
         let
            (*val () = print ("Time = " ^ (Time.toString t) ^ "\n")*)
            val () = Coordinate.origin := t
            val () = MBox.Meta.change(time, t)
         in
            ()
         end

   val setTime = setTimeGlobalStats

   fun incTime c = setTime(Time.+(MBox.Meta.deref time,
                                  Time.*(Time.fromInt c, epsilon)))
   fun decTime c = incTime(~c)

   fun resetStats () = MBox.Meta.Stats.reset time

   fun statsToString () =
     let
       val {mapReadersInsert = i,
            mapReadersRemove = r,
            mapReadersWakeUp = w, ...} = MBox.Meta.Stats.get time
       val () = resetStats()
     in
        ("Certificates Deleted: " ^ Int.toString r ^ "\n" ^
         "Certificates Inserted: " ^ Int.toString i ^ "\n" ^
         "Certificates Woken up: " ^ Int.toString w)
     end
   fun printStats () =
       print ("Application-Specific Statistics: " ^ statsToString () ^ "\n")

   val changed = Adaptive.Box.Meta.new true
   fun propagate () = (Adaptive.Box.Meta.change(changed, false);
                       Adaptive.Box.Meta.change(changed, true);
                       Adaptive.Meta.propagate(); ())

end (* MkKineticSettings *)
```

A.2.2 Incremental 3D Convex Hull Algorithm

First, we provide the code of the extended geometry which includes the geometric data structures for the convex hull algorithm. Then, we include the adaptive incremental convex hull algorithm in three dimensions. The code follows the pseudo-code described in Chapter 2.

```
signature EXTENDED_GEOMETRY_3D_ADAPTIVE =
sig
   structure Primitives : GEOMETRY_3D

   type vertex
   type edge = vertex * vertex
   type face

   val vertexID : vertex -> int
   val edgeID : edge -> int * int
   val faceID : face -> int * int * int

   (*** Constructors ***)
   val mkNewVertex : unit -$> int * Primitives.Vertex.t -$> vertex
   val mkNewFace : unit -$> vertex * edge -$> face

   (*** Conversions ***)
   val vertexData : vertex -> Primitives.Vertex.t
   val edgeData : edge -> Primitives.Edge.t
   val faceData : face -> Primitives.Face.t
   val facePyramidSides : face * Primitives.Vertex.t ->
                          Primitives.Face.t * Primitives.Face.t

   (*** Modifiable fields ***)
   val getVertexFaces : vertex -$> face ListEqAdaptive.t
   val getVertexLastFace : vertex -$> (int * face) option
   val getFaceNeighbors : face -$> (face * edge) * (face * edge) * (face * edge)
   val getFaceKiller : face -$> vertex option

   val setVertexFaces : vertex * face ListEqAdaptive.t -$> unit
   val setVertexLastFace : vertex * (int * face) -$> unit
   val setFaceNeighbor : face * edge * face -$> unit
   val setFaceKiller : face * vertex -$> unit

   (*** String ***)
   val vertexToIDString : vertex -> string
   val edgeToIDString : edge -> string
   val faceToIDString : face -> string

   val vertexToString : vertex -> string
   val edgeToString : edge -> string
   val faceToString : face -> string

end (* EXTENDED_GEOMETRY_3D_ADAPTIVE *)



signature IHULL3D_ADAPTIVE =
sig
   include APPLICATION_ADAPTIVE

   structure Geometry : EXTENDED_GEOMETRY_3D_ADAPTIVE
              where type Primitives.Coordinate.t = Settings.Coordinate.t
end
```

```
(*********************************************************************
 ** Description: 3 dimensional geometric structures
 **
 **   Edges and faces are oriented; the representation is canonical
 **   for the faces: (0, 1, 2) represents (1, 2, 0) and (2, 0, 1). The
 **   vertex with the smallest ID is the first one. Faces (0, 1, 2)
 **   and (0, 2, 1) are the same face with opposite orientations. The
 **   edges of the face (0, 1, 2) are (0, 1), (1, 2), and (2, 0).
 *********************************************************************)
functor MkExtendedGeometry3DAdaptive (structure Primitives : GEOMETRY_3D) :
        EXTENDED_GEOMETRY_3D_ADAPTIVE =
struct
   structure Primitives = Primitives
   structure Box = Adaptive.Box
   structure IBox = Adaptive.IBox
   structure List = ListEqAdaptive
   structure Key = KeyDfltAdaptive

   datatype vertex = Vertex of {id : int,
                                data : Primitives.Vertex.t,
                                faces : face List.t IBox.t,
                                lastFace : (int * face) option IBox.t}
   and face = Face of {vertices : (vertex * vertex * vertex),
                       neighbors : (face option IBox.t *
                                    face option IBox.t * face option IBox.t),
                       killer : vertex option IBox.t}

   type edge = vertex * vertex

   fun vertexID (Vertex {id, ...}) = id

   fun edgeID (u, v) = (vertexID u, vertexID v)

   fun faceID (Face {vertices = (u, v, w), ...}) =
      (vertexID u, vertexID v, vertexID w)

   (*** Constructors ***)
   afun mkNewVertex () =
      let
         val putMFaces = #putM (IBox.PutMemoTable.mkPut $ ())
         val putMLast = #putM (IBox.PutMemoTable.mkPut $ ())
   afun newVertex (id, p) =
      (Vertex {id = id,
               data = p,
               faces = putMFaces $ (id, Box.put $ (List.NIL)),
               lastFace = putMLast $ (id, NONE)})
      in
        newVertex
      end

   afun mkNewFace () =
      let
         val putMuv = #putM (IBox.PutMemoTable.mkPut $ ())
         val putMvw = #putM (IBox.PutMemoTable.mkPut $ ())
         val putMwu = #putM (IBox.PutMemoTable.mkPut $ ())
         val putMkiller = #putM (IBox.PutMemoTable.mkPut $ ())

         fun sortVertices (u, (v, w)) =
            if (vertexID u) > (vertexID v) then
               (if (vertexID u) > (vertexID w) then (u, v, w) else (w, u, v))
            else
               (if (vertexID v) > (vertexID w) then (v, w, u) else (w, u, v))

         afun newFace (vtx, edg) =
            let
               val (u, v, w) = sortVertices (vtx, edg)
```

```
            val key = (vertexID u, vertexID v, vertexID w)
            val neighborUV = putMuv $ (key, NONE)
            val neighborVW = putMvw $ (key, NONE)
            val neighborWU = putMwu $ (key, NONE)
            val killer = putMkiller $ (key, NONE)
         in
            Face {vertices = (u, v, w),
                  neighbors = (neighborUV, neighborVW, neighborWU),
                  killer = killer}
         end
   in
      newFace
   end


(*** Conversions ***)
fun vertexData (Vertex {data, ...}) = data

fun edgeData (u, v) = (vertexData u, vertexData v)

fun faceData (Face {vertices = (u, v, w), ...}) =
   (vertexData u, vertexData v, vertexData w)

fun facePyramidSides (Face {vertices = (u, v, w), ...}, c) =
   ((vertexData w, vertexData u, c), (vertexData u, vertexData v, c))


(*** Modifiable Fields ***)
afun getVertexFaces (Vertex {faces, ...}) = IBox.get $ faces

afun getVertexLastFace (Vertex {lastFace, ...}) = IBox.get $ lastFace

afun getFaceNeighbors (Face f) =
   let
      val {vertices = (u, v, w), neighbors = (nuv, nvw, nwu), ...} = f
      val nbrUV = Option.valOf(IBox.get $ nuv)
      val nbrVW = Option.valOf(IBox.get $ nvw)
      val nbrWU = Option.valOf(IBox.get $ nwu)
   in
      ((nbrUV, (u, v)), (nbrVW, (v, w)), (nbrWU, (w, u)))
   end

afun getFaceKiller (Face {killer, ...}) = IBox.get $ killer

afun setVertexFaces (Vertex {faces, ...}, fList) = IBox.set $ (faces, fList)

afun setVertexLastFace (Vertex {lastFace, ...}, (id, f)) =
   IBox.set $ (lastFace, SOME(id, f))

afun setFaceNeighbor (f, e, nbr) =
   let
      val (idA, idB) = edgeID e
      val except = Fail "Face does not have the specified edge!"

      afun setNeighbor (Face f, nbr) =
         let
            val {vertices = (u, v, w), neighbors = (nuv, nvw, nwu), ...} = f
            val (idU, idV, idW) = (vertexID u, vertexID v, vertexID w)
         in
            if idA = idU then
               if idB = idV then IBox.set $ (nuv, SOME nbr)
               else if idB = idW then IBox.set $ (nwu, SOME nbr)
               else raise except
            else if idA = idV then
               if idB = idU then IBox.set $ (nuv, SOME nbr)
               else if idB = idW then IBox.set $ (nvw, SOME nbr)
```

```
                  else raise except
               else if idA = idW then
                  if idB = idU then IBox.set $ (nwu, SOME nbr)
                  else if idB = idV then IBox.set $ (nvw, SOME nbr)
                  else raise except
               else raise except
            end
      in
         (setNeighbor $ (f, nbr) ; setNeighbor $ (nbr, f))
      end

   afun setFaceKiller (Face {killer, ...}, k) = IBox.set $ (killer, SOME k)


   (*** String ***)
   fun vertexToIDString v = Int.toString (vertexID v)

   fun edgeToIDString (u, v) =
      "(" ^ (vertexToIDString u) ^ ", " ^ (vertexToIDString v) ^ ")"

   fun faceToIDString (Face {vertices = (u, v, w), ...}) =
      "(" ^ (vertexToIDString u) ^ ", " ^ (vertexToIDString v) ^ ", " ^
            (vertexToIDString w) ^ ")"

   fun vertexToString v = Primitives.Vertex.toString (vertexData v)

   fun edgeToString e = Primitives.Edge.toString (edgeData e)

   fun faceToString f = Primitives.Face.toString (faceData f)

end (* MkExtendedGeometry3DAdaptive *)
```

We present the code for the incremental hull algorithm below. As described in Chapter 2, this algorithm is a randomized algorithm where each point is inserted into the current hull in a predetermined random order. Each point p is traversed through the previously constructed convex hulls (`findAssociatedFace`) to figure out if it lies outside the current convex hull or not. If so, a new hull is created by removing the faces (`rip`) that are visible to the current point p, and by creating new faces (`tent`) to replace the removed faces. If not, the point p is simply discarded. When all the points are processed, the algorithm outputs the final convex hull of all input points.

```
functor MkIHull3DAdaptive (structure Settings : APPLICATION_SETTINGS_ADAPTIVE) :
        IHULL3D_ADAPTIVE =
struct
   structure Settings = Settings
   open Settings

   structure C = Coordinate
   structure Primitives = Geometry3D (structure Coordinate = C)
   structure Geometry = MkExtendedGeometry3DAdaptive
                           (structure Primitives = Primitives)

   structure List = ListEqAdaptive
   open Adaptive

   type it = (int * Primitives.Vertex.t) List.t
```

```
   type ot = it
   val origin = (C.zero, C.zero, C.zero)
   val evaluate = evaluate NONE

   fun app f = afn l =>
      let
         mfun apply x = f $ x
         and loop l =
            case Box.get $ l of
               List.NIL => ()
             | List.CONS(x, tail) => let val () = apply $ x
                                     in loop $ tail end
      in
         loop $ l
      end

   fun map key f = afn l =>
      let
         val {putM = putMap, ...} = Box.PutMemoTable.mkPut $ ()

         mfun convert x = f $ x
         and loop (k, l) =
            case Box.get $ l of
               List.NIL => putMap $ (k, List.NIL)
             | List.CONS(x, tail) =>
               let val x' = convert $ x
                   val tail' = loop $ (SOME(key x), tail)
               in putMap $ (k, List.CONS(x', tail')) end
      in
         loop $ (NONE, l)
      end

   fun keyBorder (_, e) = Geometry.edgeID e

val run = afn vertices =>
let
   val newFace = Geometry.mkNewFace $ ()
   val newVertex = Geometry.mkNewVertex $ ()

   afun tent (p, boundary) =
      let
         val idP = Geometry.vertexID p

         afun processEdgeVertex (q, f) =
            case Geometry.getVertexLastFace $ q of
               NONE => Geometry.setVertexLastFace $ (q, (idP, f))
             | SOME (id, nbr) =>
               if id = idP then Geometry.setFaceNeighbor $ (f, (p, q), nbr)
               else Geometry.setVertexLastFace $ (q, (idP, f))

         afun createFace (nbr, e) =
            let
               val f = newFace $ (p, e)
               val () = Geometry.setFaceNeighbor $ (f, e, nbr)
               val () = processEdgeVertex $ (#1 e, f)
               val () = processEdgeVertex $ (#2 e, f)
            in
               f
            end

         val faces = map keyBorder createFace $ boundary
         val () = Geometry.setVertexFaces $ (p, faces)
      in
         ()
      end
```

```
mfun rip (p, f) =
   let
      val dataP = Geometry.vertexData p

      afun ripFace (f, boundary) =
         let
            val () = Geometry.setFaceKiller $ (f, p)

            val (nbr1, nbr2, nbr3) = Geometry.getFaceNeighbors $ f
            val boundary = ripLoop $ (nbr1, boundary)
            val boundary = ripLoop $ (nbr2, boundary)
            val boundary = ripLoop $ (nbr3, boundary)
         in
            boundary
         end

      and ripLoop ((f, e), boundary) =
         case Geometry.getFaceKiller $ f of
            SOME _ => boundary
          | NONE =>
            let
               val dataFace = Geometry.faceData f
               val conflict = Primitives.halfSpaceTest(dataFace, dataP)
            in
               if evaluate $ conflict then
                  ripFace $ (f, boundary)
               else
                  Box.put $ (List.CONS((f, e), boundary))
            end

      val {putM = putML, ...} = Box.PutMemoTable.mkPut $ ()

      afun sort boundary =
         let
            (*** returns true iff border b2 < border b1 ***)
            fun lessThan b1 = afn b2 =>
               let val ((u1, v1), (u2, v2)) = (keyBorder b1, keyBorder b2)
               in u2 < u1 orelse (u2 = u1 andalso v2 < v1) end

            fun borderEq (b1, b2) = (keyBorder b1) = (keyBorder b2)
            fun borderHash b = Hash.tuple2 (Hash.int, Hash.int) (keyBorder b)
            val borderKey = {eq = borderEq, hash = borderHash}

            afun quicksort (key, boundary, acc) =
               case Box.get $ boundary of
                  List.NIL => putML $ (key, acc)
                | List.CONS(border, tail) =>
                  let
                     val split = List.partition borderKey (lessThan border)
                     val (less, greater) = split $ tail
                     val keyGrAcc = SOME (keyBorder border)
                     val grAcc = quicksort $ (keyGrAcc, greater, acc)
                  in
                     quicksort $ (key, less, List.CONS(border, grAcc))
                  end
         in
            quicksort $ (NONE, boundary, List.NIL)
         end


      val f = Box.get $ f
      val dataFace = Geometry.faceData f
      val conflict = Primitives.halfSpaceTest(dataFace, dataP)
   in
      if evaluate $ conflict then
```

```
         sort $ (ripFace $ (f, Box.put $ List.NIL))
      else
         putML $ (NONE, List.NIL)
   end


mfun findAssociatedFace (p, q) =
   let
      val {putM = putReturn, ...} = Box.PutMemoTable.mkPut $ ()
      val idP = Geometry.vertexID p
      val dataP = Geometry.vertexData p

      mfun search faces =
         case Box.get $ faces of
            List.NIL => raise Fail "No faces"
          | List.CONS (f, tail) =>
               let
                  val (side1, side2) = Geometry.facePyramidSides(f, origin)
                  val conflict1 = Primitives.halfSpaceTest(side1, dataP)
                  val conflict2 = Primitives.halfSpaceTest(side2, dataP)
               in
                  if (evaluate $ conflict1) andalso
                     (evaluate $ conflict2) then
                     f
                  else
                     search $ tail
               end

      mfun findFace q =
         let
            val assocFace = search $ (Geometry.getVertexFaces $ q)
            val dataFace = Geometry.faceData assocFace
            val conflict = Primitives.halfSpaceTest(dataFace, dataP)
         in
            if evaluate $ conflict then
               case Geometry.getFaceKiller $ assocFace of
                  NONE => putReturn $ (idP, assocFace)
                | SOME k => findFace $ k
            else
               putReturn $ (idP, assocFace)
         end
   in
      findFace $ q
   end


afun incrementalHull ((u, v), q, vlist) =
   let
      val f1 = newFace $ (q, (u, v))
      val f2 = newFace $ (q, (v, u))
      val () = Geometry.setFaceNeighbor $ (f1, (u, v), f2)
      val () = Geometry.setFaceNeighbor $ (f1, (q, u), f2)
      val () = Geometry.setFaceNeighbor $ (f1, (q, v), f2)
      val faces = Box.put $ (List.CONS(f2, Box.put $ List.NIL))
      val faces = Box.put $ (List.CONS(f1, faces))
      val () = Geometry.setVertexFaces $ (q, faces)

      afun insertVertex p =
         let
            val associatedFace = findAssociatedFace $ (p, q)
            val boundary = rip $ (p, associatedFace)
            val () = tent $ (p, boundary)
         in
            ()
         end
   in
```

```
      app insertVertex $ vlist
   end


afun writeOutput ((u, v), q, vlist, hull) =
   let
      val t = getTime $ ()

      fun fixTime p =
         Primitives.Vertex.map (fn c => C.constant(C.evaluate c t)) p

      val fmtString = Primitives.Vertex.fmtString

      afun writeVertex p = print(fmtString(fixTime(Geometry.vertexData p)))
      afun writeOutVertex (_, p) = print(fmtString(fixTime p))

      afun writeHull p =
         let
            afun writeFace face =
               if Option.isSome(Geometry.getFaceKiller $ face) then () else
               let val (u, v, w) = Primitives.map fixTime
                                                  (Geometry.faceData face)
               in
                  print(fmtString u ^ fmtString v ^ "\n" ^
                        fmtString w ^ fmtString w ^ "\n\n")
               end

            val faces = Geometry.getVertexFaces $ p
         in
            app writeFace $ faces
         end

      val tStr = "t = " ^ C.Domain.fmtString t
      val () = print("# BEGIN INPUT @ " ^ tStr ^ "\n")
      val () = app writeVertex $ vlist
      val () = writeVertex $ q
      val () = writeVertex $ v
      val () = writeVertex $ u
      val () = print("# END INPUT @ " ^ tStr ^ "\n")

      val () = print("# BEGIN OUTPUT @ " ^ tStr ^ "\n")
      val () = app writeOutVertex $ hull
      val () = print("# END OUTPUT @ " ^ tStr ^ "\n")

      val () = print("# BEGIN HULL @ " ^ tStr ^ "\n")
      val () = app writeHull $ vlist
      val () = writeHull $ q
      val () = print("# END HULL @ " ^ tStr ^ "\n")
   in
      ()
   end


afun output ((u, v), q, vlist) =
   let
      afun insideVertex (p, default)  =
         case Geometry.getVertexLastFace $ p of
            NONE => default
          | SOME (id, f) => Option.isSome(Geometry.getFaceKiller $ f)

      afun checkVertex p =
         case Box.get $ (Geometry.getVertexFaces $ p) of
            List.NIL => true
          | List.CONS(f, _) =>
            insideVertex $ (p, Option.isSome(Geometry.getFaceKiller $ f))
```

```
         val {putM = putML, ...} = Box.PutMemoTable.mkPut $ ()

         afun write (p, out) =
            let
               val vtx = (Geometry.vertexID p, Geometry.vertexData p)
            in
               putML $ (SOME (#1 vtx), List.CONS(vtx, out))
            end

         mfun filter vertices =
            case Box.get $ vertices of
               List.NIL => putML $ (NONE, List.NIL)
             | List.CONS(p, tail) =>
               let
                  val tail = filter $ tail
               in
                  if checkVertex $ p then tail else write $ (p, tail)
               end

         val hull = filter $ vlist
         val hull = if checkVertex $ q then hull else write $ (q, hull)
         val hull = if insideVertex $ (v,false) then hull else write $ (v, hull)
         val hull = if insideVertex $ (u,false) then hull else write $ (u, hull)
      in
         hull
      end

   val notEnough = Fail "Not enough vertices!"

   afun run vertices =
      let
         val {putM = putInput, ...} = Box.PutMemoTable.mkPut $ ()

         mfun initialVertices () =
            case Box.get $ vertices of
               List.NIL => raise notEnough
             | List.CONS (u as (idU, _), tail) =>
               case Box.get $ tail of
                  List.NIL => raise notEnough
                | List.CONS (v as (idV, _), tail) =>
                  case Box.get $ tail of
                     List.NIL => raise notEnough
                   | List.CONS (q as (idQ, _), vlist) =>
                     ((u, v), q, putInput $ ((idU, idV, idQ), Box.get $ vlist))

         val (((idU, u), (idV, v)), (idQ, q), vlist) = initialVertices $ ()
         val sum = Primitives.Vertex.+(Primitives.Vertex.+(u, v), q)
         fun transform v = Primitives.Vertex.+(Primitives.Vertex.+(v, v),
                                               Primitives.Vertex.-(v, sum))
         afun convert (id, v) = newVertex $ (id, transform v)

         val e = (convert $ (idU, u), convert $ (idV, v))
         val q = convert $ (idQ, q)
         val vlist = map (fn (id, _) => id) convert $ vlist

         val () = incrementalHull $ (e, q, vlist)
         val hull = output $ (e, q, vlist)
         val () = if !printOutput then writeOutput $ (e, q, vlist, hull)
                                  else ()
      in
         hull
      end

in run $ vertices end

end (* MkIHull3DAdaptive *)
```