# Robust Kinetic Convex Hulls in 3D*

Umut A. Acar[1], Guy E. Blelloch[2], Kanat Tangwongsan[2], and Duru Türkoğlu[3]

[1] Toyota Technological Institute at Chicago (TTI-C)
[2] Carnegie Mellon University
[3] University of Chicago

**Abstract.** Kinetic data structures provide a framework for computing combinatorial properties of continuously moving objects. Although kinetic data structures for many problems have been proposed, some difficulties remain in devising and implementing them, especially robustly. One set of difficulties stems from the required update mechanisms used for processing certificate failures—devising efficient update mechanisms can be difficult, especially for sophisticated problems such as those in 3D. Another set of difficulties arises due to the strong assumption in the framework that the update mechanism is invoked with a single event. This assumption requires ordering the events precisely, which is generally expensive. This assumption also makes it difficult to deal with simultaneous events that arise due to degeneracies or due to intrinsic properties of the kinetized algorithms. In this paper, we apply advances on self-adjusting computation to provide 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 the approach to the problem of kinetic maintenance of convex hulls in 3D, a problem that has been open since 90s. We evaluate the effectiveness of the proposal experimentally. Using the approach, we are able to run simulations consisting of tens of thousands of points robustly and efficiently.

## 1 Introduction

In many areas of computer science (e.g., graphics, scientific computing), we must compute with continuously moving objects. For these objects, kinetic data structures [BGH99] is a framework for computing 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* or comparisons 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., change value. When a certificate fails, the motion simulator

* Acar, Blelloch, and Tangwongsan are supported in part by a gift from Intel.

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. To determine the time at which the certificates fail, it is typically assumed that the points move along polynomial trajectories of time. When a comparison is performed, the polynomial that represents the comparison is calculated; the roots of this polynomial at which the sign of the polynomial changes becomes the failure times of the computed certificate.

Since their introduction [BGH99], many kinetic data structures have been designed and analyzed. We refer the reader to survey articles [AGE+02,Gui04] for references to various kinetic data structures, but many problems, especially in three-dimensions, remain essentially open [Gui04]. Furthermore, several difficulties remain in making them effective in practice [AGE+02,GR04,RKG07,Rus07]. One set of difficulties stems from the fact that current KDS update mechanisms strongly depend on the assumption that the update is invoked to repair a single certificate failure [AGE+02]. This assumption requires a precise ordering of the roots of the polynomials so that the earliest can always be selected, possibly requiring exact arithmetic. The assumption also makes it particularly difficult to deal with simultaneous events. Such events can arise naturally due to degeneracies in the data, or due to the intrinsic properties of the kinetized algorithm[4].

Another set of difficulties concerns the implementation of the algorithms. In the standard scheme, the data structures need to keep track of what needs to be updated on a certificate failure, and properly propagate those changes. This can lead to quite complicated and error-prone code. Furthermore, the scheme makes no provision for composing code—there is no simple way, for example, to use one kinetic algorithm as a subroutine for another. Together, this makes it difficult to implement sophisticated algorithms.

Recent work [ABTV06] proposed an alternative approach for kinetizing algorithms using *self-adjusting computation* [ABH+04,Aca05,ABBT06,AH06]. The idea is that one implements a static algorithm for the problem, and then runs it under a general-purpose interpreter that keeps track of dependences in the code (e.g., some piece of code depends on the value of a certain variable or on the outcome of a certain test). Now when the variable or test outcome changes, the code that depends on it is re-run, in turn possibly invalidating old code and updates, and making new updates. The algorithm that propagates these changes is called a *change propagation* algorithm and it is guaranteed to return the output to the same state as if the static algorithm was run directly on the modified input. The efficiency of the approach for a particular static algorithm and class of input/test changes can be analyzed using *trace stability*, which can be thought as an edit distance between computations represented as sequences of operations [ABH+04].

[4] For example, the standard incremental 3D convex hull algorithm can perform a plane-side-test between a face and a point twice, once when deleting a face and once when identifying the conflict between a point and the face.

The approach can make it significantly simpler to implement kinetic algorithms for a number of reasons: only the static algorithms need to be implemented[5]; algorithms are trivial to compose as static algorithms compose in the normal way; and simultaneous update of multiple certificates are possible because the change propagation algorithm can handle any number of changes. Acar et al. [ABTV06] used the ability to process multiple updates to help deal with numerical inaccuracy. The observation was that if the roots can be limited to an interval in time (e.g. using interval arithmetic), then one need only identify a position in time not covered by any root. It is then safe to move the simulation forward to that position and simultaneously process all certificates before it. Although the approach using floating-point number arithmetic worked for 2D examples in that paper, it has proved to be more difficult to find such positions in time for problems in three dimensions.

In this paper, we propose another approach to advancing time for robust motion simulation and apply it to a 3D convex hull algorithm. We then evaluate the approach experimentally. The 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 $\delta$, and only identify events to the resolution of an interval. If many roots 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 non-empty intervals each possibly with multiple events. To separate roots to the resolution of intervals, we use Sturm sequences in a similar way as used for exact separation of roots [GK99], 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 roots fall into separate intervals. In our case, once we reach the lattice interval, we can stop without further separation. This means that if events are degenerate and happen at the same time, for example, we need not determine this potentially expensive fact.

For kinetic 3D convex hulls, we use a static randomized incremental convex hull algorithm [CS89,BDH96,MR95] 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 kinetic 3D convex hulls, which was identified in late 1990s [Gui98]. To the best of our knowledge, currently the best way to compute the 3D kinetic convex hulls is to use the kinetic Delaunay algorithm of the CGAL package [Boa07], 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

[5] In the current system, some annotations are needed to mark changeable values.

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 an integrated event—an insertion or deletion that occurs during a motion simulation.

## 2 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 $\delta$. 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. In this paper, we use self-adjusting computation, where computations can respond to any change in their data correctly by means of a generic change propagation algorithm. The correctness of change propagation has been proven elsewhere, sometimes by providing machine-checked proofs [ABD07,AAB08].

For robust motion simulations, we will need to perform the following operations:

- Compute the signs of a polynomial and 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 the 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.[6] We can eliminate the square-free assumption by a known technique that factors the polynomial into square and square-free polynomials.

[6] In practice, we start with an approximation computed by floating-point arithmetic.

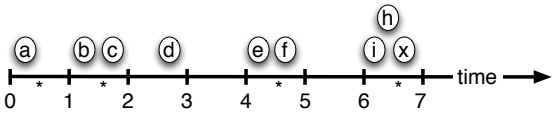


**Fig. 1.** The lattice ($\delta = 1$) and the events (certificate failures).

*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 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 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 Ali Abam et al. [AAdBY06]. 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.

## 3 Algorithm

In the kinetic framework based on self-adjusting computation [ABTV06], 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* [ABH$^+$04]. In this section, we describe a variant of the randomized incremental convex-hull

algorithm [CS89,BDH96,MR95], 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 $S \subseteq \mathbb{R}^3$, the convex hull of $S$, denoted by $conv(S)$, is the smallest convex polyhedron enclosing all points in $S$. During the execution of the algorithm on input $S$, each face $f$ of the convex hull will be associated with a set $\Sigma(f) \subset S$ of points (possibly empty). Each input point $p$ will be given a real number $\pi(p) \in [0, 1]$, called its *priority*. Each face $f$ will have the priority $\pi(f) := \min\{\pi(p) : p \in \Sigma(f)\}$. We say that a face of the hull is *visible* from a point if the point is outside the plane defined by the face.

The algorithm takes as input a set of points $S = \{p_1, p_2, \ldots, p_n\}$, and performs the following steps:

1. Assign to each $p_i$ a random *priority* $\pi(p_i) \in [0, 1]$.
2. Initialize $H := conv(A_4)$, where $A_4$ is the set of four highest-priority points.
3. Pick a *center* point $c$ inside the convex body $H$.
4. For each $f \in H$, set $\Sigma(f) := \{p \in S \setminus H :$ the ray $\overrightarrow{cp}$ penetrates $f\}$.
5. While $\exists f \in H$ such that $\Sigma(f) \neq \emptyset$:
   (a) Choose the face $f^*$ with the highest priority, and let $p^* \in \Sigma(f)$ be the point with the highest priority.
   (b) Delete all faces on $H$ visible from $p^*$. This creates a cavity in the convex hull whose boundary is defined by *horizon edges* that are incident to both deleted and live faces.
   (c) Update $H$ by creating new faces each of which consists of $p^*$ and a horizon edge to fill up the cavity. Set $\Sigma(f) := \{p^* \in S \setminus H :$ the ray $\overrightarrow{cp^*}$ penetrates $f\}$ for each new faces $f$.

In our implementation, we maintain a priority queue of faces ordered by priorities of the faces. We also store at each face the point in $\Sigma(f)$ with priority $\pi(f)$. This allows us to perform step 5(a) efficiently.

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.

For better stability, we also want the insertion of a point $p$ to visit faces of the convex hull in the same order every time. While the presented algorithm cannot guarantee this, we use the following heuristic to enhance stability. 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, increasing the likelihood that the faces are visited in the same order. 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 $A_4$.

## 4 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 [ABBT06] 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. Given a precision parameter $\delta$ 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 $\delta$ 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 $B_\ell$ and $B_v$ bits to represent the initial location and the velocity respectively; $B_\ell$ and $B_v$ can be assigned to arbitrary positive natural numbers.

## 5 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 proposed in the previous work [BGH99], i.e., responsiveness, 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 [MLt,Wee06] 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.

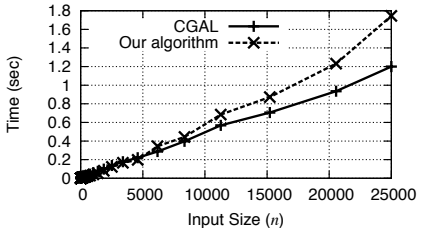


**Fig. 2.** Static algorithms compared.

| | 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 |

**Fig. 3.** Simulations compared.

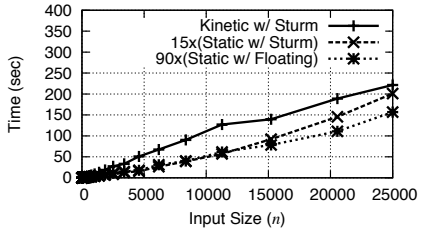


**Fig. 4.** Kinetic and static runs.

*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 shows the timings for our static algorithm and for the CGAL implementation with the `Homogeneous<double>` kernel. 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. Implementation details or our use of a high-level, garbage-collected language may be causing this difference.

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 3 shows 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 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 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 5 shows the average times for a kinetic event when we use our $\delta$-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 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 $\boldsymbol{r}(t)$ to $\frac{3}{4}\boldsymbol{r}(t)$. We update these points in the order they appear in the input, ensuring that every point is updated at least once. From this exper-

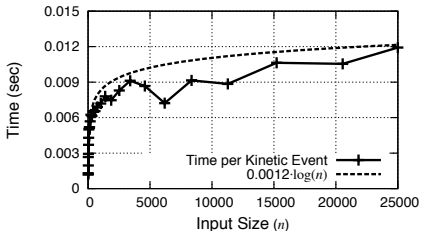


**Fig. 5.** Time per kinetic event.

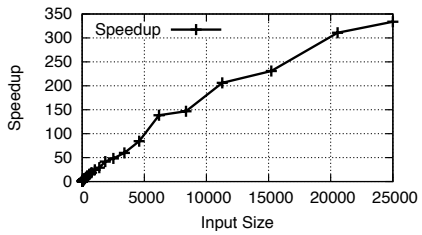


**Fig. 6.** Speedup for a kinetic event.

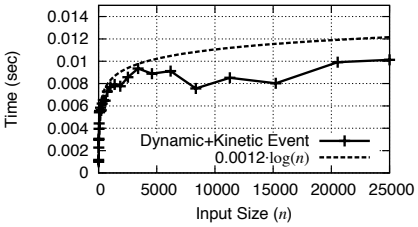


**Fig. 7.** Time per integrated event.

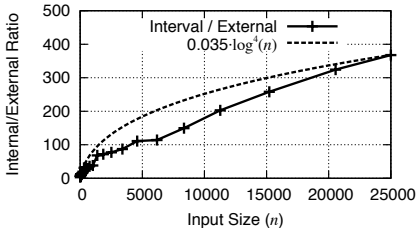


**Fig. 8.** Interval/external events.

iment, we report the average time per integrated event, calculated by dividing the total time to the number of events. Figure 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.

In a kinetic simulation, we say that an event is *internal* if it does not cause the output to change. Similarly, we say that an event is *external* if it causes the output to change. A kinetic algorithm is said to be *efficient* if the ratio of interval events to external events is small. Figure 8 shows this ratio in complete simulations with out algorithm. The ratio can be reasonably large but appears to grow sublinearly.

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 9 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 $O(n \log n)$.

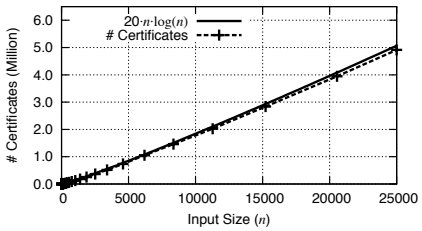


**Fig. 9.** Number of certificates.

## 6 Conclusion

We present a technique for robust motion simulation based on a hybrid of kinetic event scheduling and fixed-time sampling. The idea behind the approach is to partition the time line into a lattice of intervals and perform motion simulation at the resolution of an interval by processing the events in the same interval altogether, regardless of their relative order. To separate roots to the resolution

of intervals, we use Sturm sequences in a similar way as used for exact separation of roots in previous work, but the fixed resolution allows us to stop the process early. The approach critically relies on self-adjusting computation, which enables processing multiple events simultaneously. Although the hybrid technique using kinetic-event-scheduling and fixed-time sampling was primarily motivated by robustness issues, it may also be helpful in situations where explicit motion prediction is difficult [AGE+02].

We apply the approach to the problem of 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 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 and efficiency for reasonably large input sizes.

## References


[AAB08] Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative self-adjusting computation. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2008.

[AAdBY06] Mohammad Ali Abam, Pankaj K. Agarwal, Mark de Berg, and Hai Yu. Out-of-order event processing in kinetic data structures. In *European Symposium on Algorithms. Lecture Notes in Computer Science*, volume 4168, pages 624–635. Springer, 2006.

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

[ABD07] Umut A. Acar, Matthias Blume, and Jacob Donham. A consistent semantics of self-adjusting computation. In *Proceedings of the 16th Annual European Symposium on Programming (ESOP)*, 2007.

[ABH+04] Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Maverick Woo. Dynamizing static algorithms with applications to dynamic trees and history independence. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004.

[ABTV06] Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Jorge L. Vittes. Kinetic Algorithms via Self-Adjusting Computation. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA)*, pages 636–647, September 2006.

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

[AGE+02] Pankaj K. Agarwal, Leonidas J. Guibas, Herbert Edelsbrunner, Jeff Erickson, Michael Isard, Sariel Har-Peled, John Hershberger, Christian Jensen, Lydia Kavraki, Patrice Koehl, Ming Lin, Dinesh Manocha, Dimitris

Metaxas, Brian Mirtich, David Mount, S. Muthukrishnan, Dinesh Pai, Elisha Sacks, Jack Snoeyink, Subhash Suri, and Ouri Wolefson. Algorithmic issues in modeling motion. *ACM Comput. Surv.*, 34(4):550–572, 2002.

[AH06] Umut A. Acar and Benoît Hudson. Optimal-time dynamic mesh refinement: preliminary results. In *Proceedings of the 16th Annual Fall Workshop on Computational Geometry*, 2006.

[BDH96] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The Quickhull Algorithm for Convex Hulls. *ACM Trans. Math. Softw.*, 22(4):469–483, 1996.

[BGH99] Julien Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31(1):1–28, 1999.

[Boa07] CGAL Editorial Board. *CGAL User and Reference Manual*, 3.3 edition, 2007.

[CS89] Kenneth L. Clarkson and Peter W. Shor. Applications of random sampling in computational geometry,II. *Discrete and Computational Geometry*, 4(1):387–421, 1989.

[GK99] Leonidas J. Guibas and Menelaos 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.

[GR04] Leonidas Guibas and Daniel Russel. An empirical comparison of techniques for updating delaunay triangulations. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 170–179, New York, NY, USA, 2004. ACM Press.

[Gui98] Leonidas 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*, pages 191–209, 1998.

[Gui04] L. 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.

[MLt] MLton. http://mlton.org/.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[RKG07] Daniel Russel, Menelaos I. Karavelas, and Leonidas J. Guibas. A package for exact kinetic data structures and sweepline algorithms. *Comput. Geom. Theory Appl.*, 38(1-2):111–127, 2007.

[Rus07] Daniel Ruseel. *Kinetic Data Structures in Practice*. PhD thesis, Department of Computer Science, Stanford University, March 2007.

[Wee06] Stephen Weeks. Whole-program compilation in mlton. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 1–1, New York, NY, USA, 2006. ACM.