A package for Exact Kinetic Data Structures and Sweepline Algorithms[1]

Daniel Russel [a] Menelaos I. Karavelas [b] Leonidas J. Guibas [a]

[a] *Computer Science Department, Stanford University, Stanford, CA 94305, USA;* {drussel,guibas}@cs.stanford.edu

[b] *Department of Applied Mathematics, University of Crete, GR-714 09 Heraklion, Greece, and Institute of Applied and Computational Mathematics, Foundation for Research and Technology - Hellas, P.O. Box 1385, GR-711 10 Heraklion, Greece;* mkaravel@tem.uoc.gr

Abstract

In this paper we present a package for implementing exact kinetic data structures built on objects which move along polynomial trajectories. We discuss how the package design was influenced by various considerations, including extensibility, support for multiple kinetic data structures, access to existing data structures and algorithms in CGAL, as well as debugging. Due to the similarity between the operations involved, the software can also be used to compute arrangements of polynomial objects using a sweepline approach. The package consists of three main parts, the kinetic data structure framework support code, an algebraic kernel which implements the set of algebraic operations required for kinetic data structure processing, and kinetic data structures for Delaunay triangulations in one and two dimensions, and Delaunay and regular triangulations in three dimensions. The models provided for the algebraic kernel support both exact operations and inexact approximations with heuristics to improve numerical stability.

1 Introduction

The past few decades, implementations of geometric algorithms and data structures have reached a high level of maturity. The analysis of predicates for such algorithms has greatly advanced and currently we have predicate implementations which run at speeds approaching that of inexact arithmetic, but which still guarantee exact results. The algorithms for handling geometric objects under motion and, in particular, kinetic data structures have not

[1] A preliminary version of this paper has appeared in [1].

reached the same level of maturity, as compared to their static (i.e., non-moving) counterparts. The implementations have stumbled upon problems of numerical stability and efficiency, and exact implementations have been far from reachable. This paper presents a first step in extending the results of static geometric algorithms to geometric algorithms involving motion. We present a framework for implementing kinetic data structures and provide a well specified interface between the kinetic data structures and the necessary algebraic computations. Our framework includes inexact (but fairly robust) and exact implementations of the necessary underlying algebra and allows for the addition of filtering mechanisms in the near future.

1.1 Kinetic data structures

Kinetic data structures were first introduced by Basch et. al. in 1997 [2]. The idea stems from the observation that most, if not all, computational geometry structures are built using *predicates* — functions on quantities defining the geometric input (e.g., point coordinates), which return a discrete set of values. Many predicates reduce to determining the signs of a number of polynomial expressions on the defining parameters of the primitive objects. For example, to test whether a point lies above or below a plane we compute the dot product of the point with the normal of the plane and subtract the plane's offset along the normal. If the result is positive, the point is above the plane, zero on the plane, negative below. The validity of many combinatorial structures built on top of geometric primitives can be verified by checking a finite number of geometric predicates. These predicates, which collectively certify the correctness of the structure, are called *certificates*. For a Delaunay triangulation in three dimensions, for example, the certificates are one InCircle test per facet of the triangulation, plus a point plane orientation test for each facet or edge of the convex hull.

The kinetic data structures approach is built on top of this view of computational geometry. Let the geometric primitives move by replacing each of their defining quantities with a function of time (generally a polynomial). As time advances, the primitives trace out paths in space called *trajectories*. The values of the polynomial functions of the defining quantities used to evaluate the predicates now also become functions of time. We call these functions *certificate functions*. Typically, a geometric structure is valid when all predicates have a specific non-zero sign. In the kinetic setting, as long as the certificate functions maintain the correct sign, as time varies, the corresponding static predicates do not change values, and the original data structure remains correct. However, if one of the certificate functions changes sign, the original structure must be updated, as well as the set of certificate functions that verify it. We call such occurrences *events*.

Maintaining a kinetic data structure is then a matter of determining which certificate function changes sign next, i.e., determining which certificate function has the first real root that is greater than the current time, and then updating the structure and the set of certificate functions. In addition, the trajectories of primitives are allowed to change at any time, although continuity of the trajectories must be maintained. When a trajectory update occurs for a geometric primitive, all certificates involving that primitive must be updated. We call the collection of kinetic data structures, primitives, event queue and other support structures a *simulation*.

Sweepline algorithms for computing arrangements in d dimensions easily map to kinetic data structures by taking one of the coordinates of the ambient space as the time variable. The kinetic data structure then maintains the arrangement of a set of objects defined by the intersection of a hyperplane of dimension $d-1$ with the objects whose arrangement is being computed.

1.2 C++ terminology

In this section we present some standard C++ terminology used in the sequel of the paper. A reader familiar with C++ and generic programming may skip this section.

- *templates/generic algorithms*: C++ provides *templates* to allow writing of *generic algorithms* (algorithms which can act on any of a variety of types of data) without incurring run-time overhead. The function/class is declared along with a set of template parameters. When someone wants to use the function/class, they must specify the actual types of the template parameters and the compiler automatically generates a custom version of the function/class acting on the specified types. For example, we can write an orientation_2 predicate templated by the number type used to specify the input coordinates. Depending on the input type, we can compute the same predicate, via the same code, using doubles (and hence arrive at an inexact answer) or an exact multiprecision number type like CGAL::Gmpq, or any other number type that fits certain clearly specified requirements.
- *concept/model*: a generic algorithm assumes some requirements on the types which can be used for each template argument. In the orientation_2 predicate example, mentioned in the previous item, the type passed as the template argument must support addition, subtraction, multiplication, copying and sign computation. The set of requirements is called a *concept* and a type which implements these requirements is called a *model*.
- *arithmetic filtering/filtered predicates*: *arithmetic filtering* is a technique that allows predicates to be computed exactly (i.e., without numerical errors) at a speed approaching that of computing inexact certified approxima-

tions using doubles. The technique capitalizes on the fact that, generally, it is only the sign of some number that matters, not its actual value. Computing an approximation of the number, along with a, usually additive, error bound may be sufficient in order to determine the sign of the number, and thus avoid exact computation (i.e., computation using an exact number type). This is what is typically happening when a *filtered predicate* is computed. We first try to evaluate the predicate, i.e., the sign of one or more algebraic expressions, using a fast built-in number type (e.g., double), while keeping track of an upper bound of the occurring numerical error. If the sign of the computed quantity or quantities can be deduced, the value of the predicate is returned, otherwise the evaluation of the predicate is repeated, only this time an exact number type is used, ensuring, via a numerical-error-free computation, that the result will be correct.

- *reference counting*: *reference counting* is a programming technique that allows objects to be shared in a program without worrying about who is responsible for allocating and freeing the memory when we are done using these objects. A shared, reference counted, object has a reference counter, maintaining a count of how many pointers point to it. The pointers are wrapped in handle objects which increment/decrement the reference counter as appropriate and free the object when the count goes to zero.
- *handles*: a *handle* is an object of a class that behaves like a pointer (or is a pointer). A handle class often overloads `operator->` and `operator*`.
- *functors*: a *functor* is simply any object that can be called as if it was a function, and, in particular, an object of a class that defines `operator()`.
- *notifications/proxy objects/listeners*: it is often useful to allow two objects to be connected at run-time and to pass information back and forth. One common model for this communication is to have the object receiving the communication register a *proxy object* (a *listener*) with the object providing the *notifications*.

1.3 Goals and overview

The package presented in this paper aims to provide a solid framework for implementing and studying kinetic data structures. Given such a framework, and a well defined and simple interface between the geometry and the underlying algebra, we can then implement a variety of computational models. Kinetic data structures have been implemented numerous times, for example [3, 4],[5],[6]. However, previous implementations punted on the hard problems involved in exactly comparing failure times and handling degeneracies. The combination of inexact computation and the lack of generality of existing libraries has made implementing kinetic data structures a sometimes painful and frustrating process. Our package addresses both of these issues, providing tools for exactly performing the necessary algebraic operations when the cer-

tificate functions are polynomial functions of time, as well as providing support for easy debugging of kinetic data structures. The framework is implemented so as to allow easy addition of filtering and other techniques for accelerating the process. In addition, the package allows existing CGAL code to operate on moving objects by providing models of the CGAL kernel, which compute predicates on *snapshots* of a simulation — how the scene looks at a particular instant in time. We call *static* the algorithms and data structures defined over non-moving primitives, such as those found in snapshots of a simulation.

Just as static geometric data structures divide the continuous space of all possible inputs (as defined by sets of coordinates) into a discrete set of combinatorial structures, kinetic data structures divide the continuous time domain into a set of disjoint intervals. In each interval the combinatorial structure does not change, so, from the point of view of the combinatorial structure, all times in the interval are equivalent. In typical kinetic data structures event times are only representable by algebraic numbers, rendering computations at event times extremely expensive, requiring a number type such as CORE::Expr [7] or LEDA::real [8, 9]. We can often use the combinatorial invariance of the structure during the interval between events to perform operations at a rational valued time. See Section 4.3 for an example of where this equivalence is exploited.

In our package we provide computational support for geometric primitives defined by polynomial functions of time. The interface between the computational kernel, called the FunctionKernel concept, and the rest of the framework has been kept simple in order to facilitate adding support for other function types than the ones we currently support, as well as using other implementations for calculations on real numbers. We provide implementations of the necessary operations which support exact, filtered and unfiltered operations on real roots of general polynomials (i.e, the polynomials are not required to be square-free), as well as inexact operations which incorporate kinetic data structure-specific heuristics to improve their accuracy and stability. Since our package was written, CORE and LEDA have also provided support for the operations on real numbers necessary for performing kinetic simulations with our package. We provide a wrapper for CORE::Exprs and plan to do so for LEDA::reals. Our implementation of the necessary operations is currently faster and more stable than the CORE implementation (see Table 5 in Section 6).

The kinetic data structures package design is heavily influenced and makes extensive use of CGAL's geometric kernel and traits ideas. Most interestingly, it capitalizes on CGAL's kernel-based structure to allow easy initialization, as well as verification, of snapshots of kinetic data structures, using existing CGAL algorithms and data structures. This illustrates one of the strengths of the kernel based design and is probably the most compelling usage of it so far.

The kinetic data structures package can be used on three different levels:

(1) kinetic data structures, provided along with the framework, can be used and viewed,
(2) new kinetic data structures can be implemented and debugged,
(3) pieces of the package can be modified and replaced in order to customize them for specific applications or to depart from ordinary kinetic data structures.

The last level is generally beyond the scope of this paper, although we will hint at some possible directions for optimizations.

The rest of the paper is structured as follows. In Section 2 we present the considerations that guided our design and a high level summary of our framework. Next, in Section 3, two simple examples illustrate the usage of a couple of the kinetic data structures we supply. In Section 4 we discuss the various pieces of our kinetic data structures framework in detail, followed by a presentation of the FunctionKernel concept in Section 5, which is responsible for computing the failure times of events given their functional description. Finally, in Section 6 we present some, mostly qualitative, results about how our supplied kinetic data structures perform, and in Section 7 we conclude with our plans for future work as well as suggested directions for work in the area of kinetic data structures implementations. The package is quite large, totaling almost 40,000 lines of code, so this paper is by no means meant to provide an exhaustive presentation of all the pieces.

2 Design considerations

The design of the framework has been guided by a number of considerations. Below we mention the most important ones:

- *Lightweight and extensible*: The framework is made from many lightweight components. The components impose minimal extra overhead for unused parts and can be easily replaced if customization/optimization is needed. Modularity and genericity are materialized through the extensive use of C++ templates, which allows for most of the cost of the flexibility to be handled at compile rather than at run-time.
- *Support for multiple kinetic data structures*: Multiple kinetic data structures operating on the same set of geometric primitives must be supported. Unlike their static counterparts, the description of the trajectory of a kinetic primitive can change at any time. Such trajectory changes affect the failure times of all certificates, in all kinetic data structures, that involve the geometric primitive whose trajectory has changed. Our solution is to have a central-

ized repository of geometric primitives to which all kinetic data structures have access, which in turn notifies all registered kinetic data structures in the event of a trajectory change.
- *Support for multiple event types*: The framework must be able to support many different types of events in the same simulation. This is obviously necessary in order to support multiple kinetic data structures, but is also useful for a single kinetic data structure. For example, a Delaunay triangulation has events corresponding to 2-3 flips and 3-2 flips; it is convenient to represent them as different types.
- *Support for and access to existing static data structures*: We provide functionality to aid in the use of existing static data structures in CGAL, by allowing static algorithms to act on snapshots of the running kinetic data structure. Our method of supporting this is to provide a kernel associated with a specific time value, that can be used as a kernel or traits class in CGAL's algorithms and data structures. Our implementation takes advantage of having the primitives stored in a central repository.
- *Support exact and filtered computation*: Our approach is to encapsulate the computation/comparison of event times in the FunctionKernel concept. The interface between this concept and the rest of the kinetic data structures framework is limited to a few functors, allowing reasonable flexibility in the implementation of the algebraic layer. In addition, since filtering internal to the algebraic layer has not, so far, achieved the speed gains we would like, the framework allows filtering to be implemented on a geometric level.
- *Thoroughness*: The common functionality shared by different kinetic data structures should as much as possible be handled by the framework. Different kinetic data structures we have implemented using the framework share only a few lines of code (basically the names of some shared functions). We provide numerous helper classes to facilitate development of new structures.
- *Ease of debugging*: We provide hooks to aid checking the validity of kinetic data structures, as well as checking that the framework is used properly. We also provide a graphical user interface which allows the user to step through the events being processed, to reverse time or to look at the history of the kinetic simulation.

2.1 A high level view of the framework

The package is structured around five main concepts. See Figure 1 for a schematic of how a kinetic data structure interacts with the various parts.

- A kinetic data structures simulation needs some object to keep track of the current time and pending events. The Simulator plays this role. In addition, since the Simulator knows when no events are occurring, it can inform data structures when they can easily verify their correctness. There should be

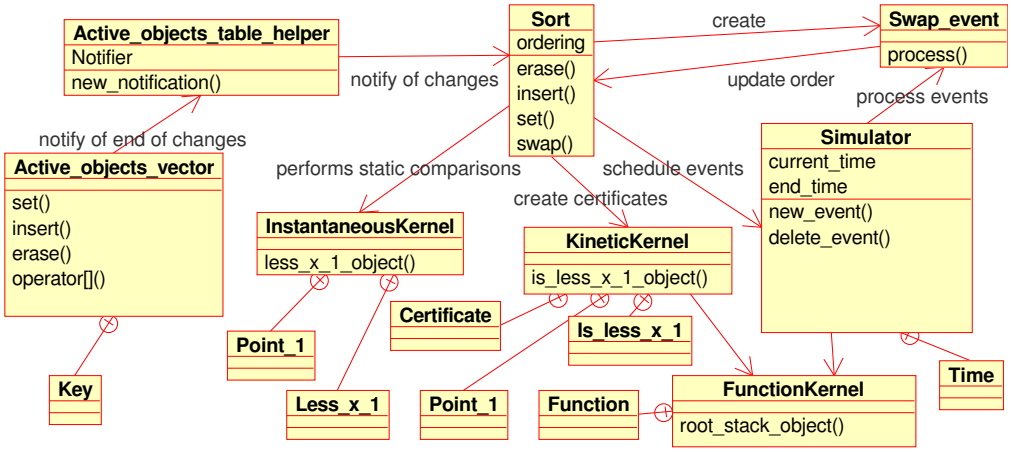

Fig. 1. The figure shows the interaction between the Sort kinetic data structure and the various pieces of our package. Other, more complicated, kinetic data structures will also use the InstantaneousKernel in order to insert/remove geometric primitives and audit themselves. Sort uses the sorting functionality in STL instead.

one instance of a model of this concept per simulation.

- CGAL algorithms are structured around a kernel which provides the interface between the algorithm and the geometric primitives. The KineticKernel concept plays this role for kinetic data structures by defining primitives and generating certificates. For example, under the assumption that motions are polynomial functions of time with rational coefficients, a KineticKernel would define points in 2D (resp. 3D) as a pair (resp. triple) of polynomials defined over time. Certificates, such as the 2D orientation certificate of three points or the 2D incircle certificate for four points, would again be polynomials over time, computed using the polynomial expressions for the coordinates of the moving points involved.
- As was mentioned in the previous section, storing the kinetic primitives in one location allows them to be shared by several kinetic data structures. The ActiveObjectsTable concept defines this repository. It disseminates notifications when trajectories change or primitives are added or deleted. There is generally one instance of a model of this concept per simulation.
- CGAL defines static equivalents of many kinetic data structures. The InstantaneousKernel concept allows kinetic data structures to be built on top of the corresponding static structure and the static algorithms to be used for initialization, modifications and verification. The InstantaneousKernel provides a model of the CGAL kernel which performs predicate computations on the kinetic primitives at a particular instant in time.
- the FunctionKernel concept which defines the computational kernel of our framework. Models of this concept are responsible for representing, generating and manipulating the motion and certificate functions, as well as their real roots. It is this concept that provides the kinetic data structures framework with the necessary algebraic operations for manipulating event times.

The FunctionKernel is discussed in detail in Section 5.

For simplicity, we added an additional concept, that of SimulationTraits, which wraps together a particular set of choices for the above concepts and is responsible for creating instances of each of the models (and handing out handles to instances of the singleton models). This concept encapsulates what is otherwise 10-15 lines of includes, typedefs and declarations. The user now picks the dimension of the ambient space and chooses between exact and inexact computations. The model of SimulationTraits then creates an instance of a Simulator and an ActiveObjectsTable. Handles for these instances as well as instances of the KineticKernel and InstantaneousKernel can be requested from the simulation traits class. See the example in the following section for an illustration of how to use some of this functionality.

Surrounding these central set of concepts are a large number of smaller concepts, the models of which act either as glue between objects or as helper classes. The key smaller concepts will be described along with the appropriate central concepts in the corresponding subsections of Section 4.

3 Using supplied data structures

3.1 A simple example: kinetic sorting

The package ships with kinetic data structures for sorting one-dimensional points (or equivalently, for maintaining one-dimensional Delaunay triangulations), Delaunay triangulations in two and three dimensions, as well as regular triangulations in three dimensions. In addition, there are kinetic data structures to bounce moving points off the walls of a box in two or three dimensions, primarily in order to aid in producing visually pleasing and interesting demos. Regular triangulations in two dimensions will probably be added in short order. Using the provided structures can be as simple as the program in Figure 2, which maintains a sorted list of a set of points as they move from their positions at time zero following polynomial trajectories (which are read from a file).

Several important things happen behind the scenes in this example. First, the ActiveObjectsTable, which holds the moving points, notifies the kinetic data structure that new points have been added to the simulation. This happens when points are read into the table on line 20. Second, the Sort kinetic data structure registers its events with the Simulator by providing, for each event, a time and a proxy object. The sorted list is built incrementally and events are added after the involved points are inserted into the list. When a particular

```
1  #include <CGAL/Kinetic/Exact_simulation_traits_1.h>
2  #include <CGAL/Kinetic/Sort.h>
3  #include <CGAL/Kinetic/Sort_event_log_visitor.h>
4
5  int main() {
6      // choose a simulation traits class that supports exact computations
7      typedef CGAL::Kinetic::Exact_simulation_traits_1 Simulation_traits;
8      // define the KDS we are interested in; here: kinetic 1D sorting
9      typedef CGAL::Kinetic::Sort<Traits,
10                                 CGAL::Kinetic::Sort_event_log_visitor> Sort;
11
12     // create the traits and an instance of kinetic sorting using these traits
13     Simulation_traits tr;
14     Sort sort(tr);
15
16     // read 1D points and their trajectories ; the KDS automatically
17     // builds the initial sorted list incrementally using notifications
18     // from the active objects table to the kinetic data structure
19     std::ifstream in("data/points_1");
20     in >> *tr.active_points_1_table_handle();
21
22     // run the kinetic simulation
23     tr.simulator_handle()->set_current_time(
24         tr.simulator_handle()->end_time()
25     );
26
27     // write everything that happened to the terminal
28     std::copy(sort.visitor().events_begin(), sort.visitor().events_end(),
29               std::ostream_iterator<std::string>(std::cout, "\n"));
30
31     return 0;
32 };
```

Fig. 2. A simple example illustrating the use of a supplied kinetic data structure, namely kinetic sorting. First, on line 7, the SimulationTraits object is chosen (in this case one that supports exact computations). Then the kinetic data structure is defined, using the chosen traits object and a visitor class which logs changes to the sorted list (lines 9–10). Next, instances of the two are created (lines 13–14) and a set of points is read from a file (line 20). The kinetic data structure is automatically notified of the new points by the ActiveObjectsTable and inserts them in the sorted list. Then, the simulator is instructed to process all the events until the end of the simulation (lines 23–25). Finally, a record of what happened is printed to the terminal (lines 28–29).

event occurs (as the events are being processed on line 23), the Simulator calls a function on the event proxy object which in turn updates the kinetic data structure.

The example illustrates how to monitor the supplied data structures as they evolve by using a Visitor object (cf. line 28) — a small class whose methods are called whenever the kinetic data structure changes. Hooks for such visitor concepts are provided for all of the shipped kinetic data structures. In the case of kinetic sorting, the visitor's methods are called every time a new point is inserted in the sorted list, when one is removed, or when two points are swapped in the sorted order.

The visitor concept is quite powerful, allowing us, for example, to implement a data structure for computing and storing two-dimensional arrangements of x-monotone curves on top of the Sort data structure using about 60 lines of code. To achieve this, the arrangement data structure uses an instance of the Sort structure to maintain a y-sorted list of the curves as we sweep along the x-coordinate (used as the time variable). Kinetic events, such as two curves being swapped in sorted order, correspond to the intersection of two curves in the plane and result in a new vertex being created and added to the arrangement graph. Beginnings and ends of curves map to insertions and deletions of points from the simulation, which is also handled by the kinetic data structure framework and Sort. The framework provides event models to insert, change and remove objects from the ActiveObjectsTable.

A note about the example program: The certificates associated with kinetic sorting always involve two moving points (the certificate polynomial is the difference between the polynomials describing the two motions). Since the trajectories are polynomials, each pair of them can have a constant number of intersection points, i.e., each possible certificate of our kinetic simulation can have a constant number of failure times. As a result, there are a finite number of events to be processed and the program is guaranteed to terminate.

3.2 Visualizing kinetic data structures

Figure 3 shows how to use a number of additional features of the framework. First, it shows that two kinetic data structures (kinetic Delaunay and the enclosing box) can coexist on the same set of points without any extra effort. Both interact with the moving points through the ActiveObjectsTable, and never need to directly interact with one another. Second, objects (like qt_w, qt_mps and qt_dt2) are all stored using reference counted handles (Object::-Handle), which allows them to share references to one another without the user having to worry about memory management and order of deletion. For

```
#include <CGAL/Kinetic/Exact_simulation_traits_2.h>
#include <CGAL/Kinetic/Delaunay_triangulation_2.h>
#include <CGAL/Kinetic/Enclosing_box_2.h>
#include <CGAL/Kinetic/IO/Qt_moving_points_2.h>
#include <CGAL/Kinetic/IO/Qt_triangulation_2.h>
#include <CGAL/Kinetic/IO/Qt_widget_2.h>

int main(int argc, char *argv[]) {
    using namespace CGAL::Kinetic;
    typedef Exact_simulation_traits_2 Simulation_traits;
    typedef Delaunay_triangulation_2<Traits> Del_2;
    typedef Enclosing_box_2<Traits> Box_2;
    typedef Qt_widget_2<Traits::Simulator> Qt_widget;
    typedef Qt_moving_points_2<Traits, Qt_gui> Qt_mps;
    typedef Qt_triangulation_2<Del_2, Qt_widget, Qt_mps> Qt_dt2;

    // create a simulation traits and add two KDSs:
    // a kinetic Delaunay triangulation and an enclosing box
    // with x_min = y_min = -10 and x_max = y_max = 10;
    // the moving points bounce against the walls of the enclosing box
    Simulation_traits tr;
    Box_2::Handle box = new Box_2(tr, -10, -10, 10, 10);
    Del_2::Handle kdel = new Del_2(tr);

    // register the simulator, set of moving points and
    // Delaunay triangulation with the kinetic Qt widget
    Qt_widget::Handle qt_w = new Qt_widget(argc, argv, tr.simulator_handle());
    Qt_mps::Handle qt_mps = new Qt_mps(qt_w, tr);
    Qt_dt2::Handle qt_dt2 = new Qt_dt2(kdel, qt_w, qt_mps);

    // read the trajectories of the moving points
    // the simulation traits automatically inserts them in the two KDSs
    // and schedules the appropriate kinetic events; as in the kinetic
    // sorting example this is done with appropriate notifications
    std::ifstream in("data/points_2");
    in >> *tr.active_points_2_table_pointer();

    // run the interactive kinetic simulation
    return qt_w->begin_event_loop();
};
```

Fig. 3. This example shows how to display a kinetic data structure using the provided Qt widget, a screen capture of which is shown in Figure 4.

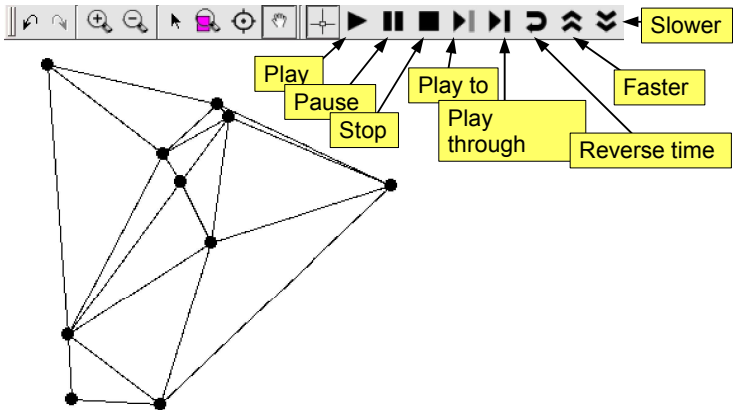

Fig. 4. The figure shows the graphical user interface for controlling two-dimensional kinetic data structures. It is built on top of CGAL::Qt_widget and adds buttons to play, pause the simulation, step through events or run the simulation backwards.

example, the Qt_triangulation_2 object needs a handle to the kinetic triangulation, in order to get the combinatorial structure of the triangulation to display, and a handle to the ActiveObjectsTable to compute the current coordinates of the points.

Finally, the example shows how to use the graphical interface elements provided (see Figure 4). Our package includes Qt widgets [2] for displaying kinetic geometry in two and three dimensions. In addition to being able to play and pause the simulation, the user can step through events one at a time and reverse the simulation to retrace what had happened. The three-dimensional visualization support is based on the Coin library [11].

3.3 A note on Enclosing_box_[23]

The Enclosing_box_[23] data structures provided are fixed boxes within which the moving points are bouncing. In that respect, they are different from the other kinetic data structures (i.e., sorting and triangulations) in that they change the trajectories of the points in the simulation. This introduces a number of additional problems. First, points traveling on polynomial trajectories collide with a plane at a time which is not necessarily representable by a rational number. After the collision, the resulting trajectory can not be represented by a polynomial with rational coefficients, which would require significantly more expensive computations. In addition to the overhead from handling the larger set of possible coefficients, the bit complexity of the coefficients will

[2] The Qt library is a high-performance, cross-platform application development framework. It includes a rich set of collection classes, as well as functionality for, e.g., GUI programming, layout, database programming, networking, XML, internationalization, and OpenGL integration. For details see [10].

increase with each collision, slowly dragging the simulation to a halt (due to limited computational resources, such as memory). As a result, we instead bounce points off of planes close to the desired plane, ensuring that the bit complexity of the coefficients remains constant.

4 The support code

As we have already mentioned, one of the major goals of our package is to support the development of new kinetic data structures. In Section 2 we described at a very high level how we designed our package in order to facilitate this goal. Below we discuss in detail each one of the first four major concepts around which our package is structured: the ActiveObjectsTable, the KineticKernel, the InstantaneousKernel and the Simulator.

The kinetic framework makes use of several types defined by the algebraic layer, the FunctionKernel. First, the algebraic layer defines a number type FT which is used for all calculations (typically a rational number type). Second, it defines a Function which is a univariate function (of time) and which supports and is closed under ring operations. In addition, one can compute the Function's value at a particular time, represented by an FT. Finally, there is a type Root which represents zeros of the Functions and a functor for enumerating the roots given a Function. This interface is kept simple in order to allow changing the implementation of the FunctionKernel without much difficulty — we provide several different implementations each with their own advantages. The FunctionKernel will be described in more detail in Section 5.

4.1 The ActiveObjectsTable

The ActiveObjectsTable stores a set of kinetic primitives. It is needed since several kinetic data structures can be built on top of the same set of primitives and all need to update their certificates when trajectories change. Since trajectories can change at any time, polling of the primitives is impractical and so notifications must be sent.

The table provides centralized storage, which notifies kinetic data structures when new primitives are added, when primitives are removed, or when a trajectory changes using the notification framework. Each primitive is uniquely identified by a Key that can be used to change or remove it. We provide one model of the ActiveObjectsTable concept, called Active_objects_vector which stores the moving primitives in an std::vector.

The table is typically allocated on the heap and accessed through reference counted pointers, since many different classes (e.g., the various kinetic data structures and the instantaneous kernel) need access to it.

A comment about notifications: The kinetic framework uses a common notification framework to provide run-time communication between classes. For the ActiveObjectsTable, notifications are provided when objects are added, removed or changed. To receive such notifications, the kinetic data structure creates a proxy object which implements a standard Listener interface. It then registers this proxy with the table. When a group of changes to the table are finished (changes can be batched for efficiency), the table calls the new_notification(IS_EDITING) method (IS_EDITING is the particular type of notification being passed) on the proxy object. We provide proxy objects which then call insert, remove, or change methods on the kinetic data structure for each modified primitive. Pointers in the notification framework are reference counted appropriately to avoid issues caused by the creation and destruction order of kinetic data structures and the simulator (see Section 4.4).

4.2 The KineticKernel

The KineticKernel is structured very much like static CGAL kernels. It defines a number of primitives, which in the model provided are Point_1, Point_2, Point_3 and Weighted_point_3. The primitives are defined by a set of Cartesian coordinates each of which is a function of time as defined by the Function of the algebraic layer. In addition it defines constructions and certificate generators which act on the primitives. The certificate generators are the direct analog of the non-kinetic predicates. Each certificate generator takes a number of primitives as arguments, but, instead of producing an element from a discrete set, it produces a set of discrete failure times for the certificate. Take for example the 2D orientation test defined over three two-dimensional points. In a static kernel, the orientation test would take three points (six Cartesian coordinates) and return one of three values: `POSITIVE`, `NEGATIVE` or `ZERO`. In the kinetic kernel the 2D orientation predicate takes three moving points (e.g., six polynomial functions of time) and returns a function of time which is the expansion of the standard 3-by-3 determinant expressing the 2D orientation test. The real roots of this polynomial are the failure times of this certificate. These failure times are wrapped in an object of type KineticKernel::Certificate.

A Certificate is a simple object whose primary function is to produce a Time object (the Root of the function kernel) representing the failure time of the certificate. Since, the handling of most certificate failures involves creating a new certificate whose certificate function is the negation of the old certificate function, a Certificate object caches any work that could be useful to isolate

future roots of the certificate function (such as the standard Sturm sequence of the certificate function, if it is a polynomial). To illustrate this further, if you have two one-dimensional points with coordinate functions $p_0(t)$ and $p_1(t)$, the certificate that the first moving point is after the second corresponds to the inequality $p_0(t) - p_1(t) > 0$. When the certificate fails and the two points exchange rank in the sorted list of points, the new certificate is $p_1(t) - p_0(t) > 0$, which is the negated version of the certificate just processed and which has the same roots.

The model of KineticKernel provided with our framework includes the certificate generators necessary for Delaunay triangulations (in 1D, 2D and 3D) and regular triangulations (in 3D). New certificates can be fairly easily added. An example is included in the distributed code.

4.3 The InstantaneousKernel

It is very convenient to be able to use existing static data structures and algorithms code, as provided by CGAL, to act on a kinetic data structure at a particular moment. The InstantaneousKernel is a (currently limited) model of the CGAL kernel which allows predicates to be evaluated as the points appear at a particular instant in time. For example, the kinetic Delaunay triangulation, contains an ordinary CGAL Delaunay triangulation instantiated using an InstantaneousKernel. When a new point needs to be inserted, the kinetic Delaunay just informs the instantaneous kernel of the current time and then calls the CGAL::Delaunay_triangulation_[23]::insert(Point_[23]) method.

Internally, the InstantaneousKernel stores the current time and a handle to the ActiveObjectsTable. The primitive being used (for example Point_3 for 3D Delaunay triangulation) is defined to be a Key in the ActiveObjectsTable (this works since CGAL algorithms never access the coordinates of the primitives directly). When a CGAL static predicate is evaluated, the instantaneous kernel fetches the trajectory from the table, computes the current coordinates of the primitives in question and passes them to the static predicate. For efficiency reasons, the current time used by the instantaneous kernel must be of type FT. This is not a strong limitation in practice as will be discussed in Section 4.4.

4.4 The Simulator

The main role of the Simulator is to maintain the event queue. Kinetic data structures call methods of the Simulator to schedule new events, deschedule old ones and access and change data contained in already scheduled events (the operations on existing events are performed using a key which was returned

when the event was scheduled). For controlling the simulation, methods in the Simulator allow stepping through events, advancing time and even running the simulation backwards (we run the simulation with the time running in the opposite direction).

The kinetic sorting example in Figure 2 shows the basic usage of the Simulator. First, the Simulator is created by the SimulatorTraits. The kinetic data structure gets a handle to the simulator from the simulator traits class and uses the handle to add its events to the simulation. The Simulator is then told to advance time up until the end of the simulation, processing all events along the way.

Each event is represented by a Time and an instance of a model of the Event concept. Events are responsible for taking the appropriate action in order to handle the kinetic event they represent. Specifically, the Event has one key method, Event::process(), that is called when the event occurs (there are a few other optional methods, not discussed here, which aid in handling degeneracies). The body of the process() method typically simply calls a method of the kinetic data structure that created the event; for example in our kinetic sorting example, processing an event means calling the swap method of the kinetic sorting data structure.

In the model of the Simulator concept that we provide, any model of the Event concept can be inserted as an event. This ability implies that events can be mixed at run-time, which is essential when we want to support multiple kinetic data structures operating on the same set of moving geometric primitives. There is some overhead associated with supporting many different event types but it is not large compared to typical event handling costs.

The Time concept is likewise simple. It requires that the objects be comparable (specifically we require that the function CGAL::compare(Time,Time) be defined), and that the FunctionKernel provides a functor which returns a number of type FT in between two non-equal Time values. This capability is used in order to perform computations at times when the combinatorial structure of the kinetic data structure is the same as immediately before or after an event, but the current time can be represented by a much simpler object of type FT. An example use of this capability is auditing.

Internally the Simulator maintains a priority queue containing the scheduled events. The type of the priority queue is a template argument to our Simulator model and, as such, it can be chosen by the user (however, there is a default choice — see below). In our package, we provide two different types of priority queues, a heap and a two-list priority queue [12]. A two-list queue is a queue in which there is a sorted front list, containing all events before some time (and after the current time) and an unsorted back list. The queue tries to maintain a

small number of elements in the front list, leaving most of them in the unsorted main pool. The two-list queue, although seemingly an unconventional choice, is our default queue when using exact computations because it minimizes comparisons involving events that are far in the future. These events are likely to be deleted before they are processed, so extra work done structuring them is wasted. Our experiments have shown that, for example, the two-list queue causes a 20% reduction in running time relative to a binary heap for Delaunay triangulations with degree 3 polynomial motions and 20 points.

The Simulator takes advantage of its knowledge of all currently scheduled events in order to help the kinetic data structures verify their correctness. At any particular instant in time, no kinetic data structure can change combinatorially in the open interval from the last event processed to the next scheduled event. In addition, there are no short-lived degeneracies in the interval, so, in general, the corresponding static structure is non-degenerate (note that a certificate function could be uniformly zero, i.e., the points move in a manner so that they are always degenerate — we have no way of detecting this). If the interval between two consecutive events is non-empty, there is a time in the interval representable by an FT at which the kinetic data structures are non-degenerate. When such a time is found, the Simulator notifies the kinetic data structures, which can then use an InstantaneousKernel coupled with the static CGAL algorithms to verify their correctness. For example, the kinetic Delaunay triangulation implementation can simply call the CGAL::Delaunay_triangulation_3::is_valid() method to check if the data structure is correct. This auditing can typically catch algorithmic or programming errors much closer to the time they arise in the simulation than, for example, using visual inspection. Such easy auditing is one of the powerful advantages of having an exact computational framework, since, as with static data structures, differentiating between errors of implementation and numerical errors, in the context of an inexact framework, is quite tricky.

4.5 Using the pieces of the package

In this section we will explain in detail how a typical kinetic data structure uses the various pieces of the framework. The kinetic data structure of reference will, once again, be our implementation of kinetic sorting (see Figure 2). A schematic of its relationship to the various components is shown in the UML diagram in Figure 1. In this subsection we abuse, for reasons of simplicity of presentation, the concept/model semantics: when we refer to concepts we actually refer to an instance of a model of them.

As with most kinetic data structures, Sort maintains some sort of combinatorial structure (in this case a sorted doubly linked list), each link of which has

a corresponding certificate in the event queue maintained by the simulator. In the case of sorting, there is one certificate maintained for each "edge" between two consecutive elements in the list.

On creation, Sort is passed a copy of the SimulationTraits for this simulation, which it saves for future use. It gets a handle to the ActiveObjectsTable by calling the SimulationTraits::active_points_table_1_handle() method and registers a proxy in order to receive notifications of changes in the point set. It will also use the kinetic and instantaneous kernels from the traits object.

When new points are added to the ActiveObjectsTable, the proxy calls the insert(Point_key) method of the kinetic data structure. The Point_key here is the key which uniquely identifies the newly inserted point both in the ActiveObjectsTable as well as within the scope of the InstantaneousKernel. The data structure then requests a model of the InstantaneousKernel from the SimulationTraits. It sets the time on the instantaneous kernel to the time value gotten from the Simulator::current_time_nt() method. This method returns an FT that is between the previous and next event, as discussed in Section 4.4. An InstantaneousKernel::Less_x_1 predicate and the STL algorithm std::upper_bound() [3] are then used to insert the new point in the sorted list. For each inserted object, the kinetic data structure removes the no longer relevant certificate from the event queue by calling the Simulator::delete_event(Key) function and creates two new certificates using the KineticKernel::Is_less_x_1 certificate functor. The new certificates are inserted in the event queue by calling the Simulator::new_event(Time, Event) method where Event is a proxy object which instructs the Sort kinetic data structure to swap two points when its process() method is called.

Now that the kinetic data structure has been initialized, the Simulator is instructed to process all events. Each time an event occurs, the simulator calls the process() method on the corresponding proxy object. This method typically then calls a function on the corresponding kinetic data structure. In the case of kinetic sorting, it instructs the data structure to swap two points and remake the changed certificates.

The Simulator can periodically instruct the kinetic data structures to audit themselves. A proxy object maps the notification on to an audit() function call in the kinetic data structure. To audit itself, the kinetic data structure builds a list of all the current points and uses std::sort to sort this list with a comparison function gotten from the InstantaneousKernel. This sorted list is compared to the maintained one to verify correctness. This could also have

[3] The STL std::upper_bound() algorithm attempts to find some given element e in an ordered range of elements of the same type as e. More precisely, it returns the last position in the range where e could be inserted without violating the ordering. For more details see [13].

been done by evaluating the InstantaneousKernel predicate for each sorted pair. Since auditing a kinetic data structure typically requires at least linear time, the auditing procedure in between events is deactivated by default. The user can however easily switch it on by defining appropriate CGAL macros.

This general structure of the interaction between the kinetic data structure and the framework is shared by all provided kinetic data structures and has proved itself to go quite far.

5 The FunctionKernel

The FunctionKernel handles all computations involving generating and solving certificate functions. From the point of view of the rest of the kinetic framework, the FunctionKernel is not related to a specific class or classes of functions (such as polynomials or rational functions). Its interface with the upper layers of the framework has been kept very simple intentionally, in order to allow different classes of functions to be handled. Another very important reason for the minimality of the interface is to allow for alternative implementations to those our framework provides, as well as for flexibility in optimizing the techniques used under this layer (for the case of polynomials this refers, for example, to the algorithms used for root isolation). Currently, all implementations provided support polynomials with rational coefficients as the function type. As a result some of the details mentioned in this section only apply to such polynomials.

We provide several models of the function kernel concept, the most efficient uses our algebraic solvers. However, general purpose algebraic packages, such as CORE [7] have become competitively fast and reliable and may replace our implementation of the exact algebra in the near future.

The key types defined by the kernel are

FT a field number type. This type is typically either CGAL::Gmpq or double.

Function the object used to represent a coordinate of a primitive, as well as a certificate function. This type is a polynomial over FT in all current models.

Root a real root of a certificate function. Roots are required to exactly represent all values representable by FT. This type is typically either an isolating interval for a single root of the function coupled with the function, or a double which approximates the root.

The following predicates and constructions are required by the FunctionKernel concept:

(1) comparison of two Roots. This predicate is used to sort the kinetic events so that they are processed in the proper order.
(2) computation of an FT that is between two non-equal Roots. Using this we can determine whether a certificate function is positive or negative immediately before or after a root of another certificate function. This is used for handling degeneracies.
(3) computation of the parity of a root. This predicate determines if a root has even or odd multiplicity. If a root has even multiplicity it needs to be broken into two events (or, possibly, dropped entirely) as it represents a certificate which fails momentarily.
(4) computation of the sign of a Function at an FT. This predicate is used to verify the correctness of kinetic data structures (as discussed below, certificates are required to be positive on creation).
(5) evaluation of a Function at an FT. This predicate is needed for the InstantaneousKernel as well as for displaying kinetic primitives.
(6) construction of a Function g from f such that $g(t) = f(-t)$. This construction is needed in order to allow time to be reversed, which is useful for inspecting and debugging kinetic data structures.

Together these operations are sufficient to allow the correct processing of events, handling of degeneracies, usage of static data structures, run-time error checking as well as run-time verification of the correctness of the kinetic data structure.

We implemented the necessary operations using a variety of algorithms on polynomial functions. The final best candidates we settled on are

- exact computations using unfiltered Sturm sequences to isolate roots
- numerical (inexact) root approximation (as discussed below, certain changes are necessary to the inexact solver in order to make it reliable enough for kinetic data structures), and
- a wrapper around CORE::Expr, which implements the required concepts.

The exact Sturm sequence based model can handle non-square-free polynomials, as well as polynomials with arbitrary field number type coefficients, and is has proved itself to be very robust. We also have models using filtered and unfiltered computations, based on Descartes rule of sign, but they do not offer any performance benefit, nor are they much worse.

All function kernel models provided are structured as two nested kernels, the inner, RationalKernel, which provides rational operations on polynomials — i.e., operations only involving functions and rational numbers as represented by the type FT — and the outer one, from here on just called the PolynomialKernel, which adds operations involving Roots. This two level structure allows the computations involved in constructing Roots to use the rational polynomial

operations in a manner that is oblivious to the details of how these computations are implemented. The rational kernel layer provides a useful general purpose polynomial manipulation library. However, it is being superseded in CGAL by the more general EXACUS library [14].

Example operations provided by the RationalKernel include

- computation of the sign of a polynomial at a rational number
- computation of the Sturm sequence of two polynomials
- computation of a polynomial mapped under various transforms such as $g(x) = x^d f(1/x)$, where d is the degree of f, or $g(x) = f(x - c)$, where c is a number of type FT. These are useful for evaluating Descartes rule of sign, for example.

The PolynomialKernel kernel adds a set of operations on real roots of polynomials to the kernel. These include

- isolating roots of polynomials
- comparing roots of polynomials
- computation of the sign of a polynomial evaluated at a root of another polynomial
- finding a rational number (representable by FT) between two non-equal roots

For exact computations, the primary representation for roots is the now standard choice of a polynomial with an associated isolating interval (an interval containing exactly one distinct root of a polynomial — the root of interest) along with whether the root has odd or even multiplicity and, if needed, the Sturm sequence of the polynomial. Two roots can be compared by first checking if the isolating intervals are disjoint. If they are, then we know the ordering of the respective roots. If not we can subdivide the two intervals, using their endpoints, and repeat. In order to avoid subdividing endlessly, once we subdivide a fixed number of times, we use the Sturm sequence of p and $p'q$ (where p and q are the two polynomials and p' is the derivative of p) to evaluate directly the sign of the second polynomial evaluated at the root of the first (note that this Sturm sequence is applied to a common isolating interval of the roots of interest of both polynomials).

That said, far and away the most expensive aspect of running an exact kinetic data structure is isolating the roots. For example, when maintaining a three-dimensional Delaunay triangulation of 200 points moving on degree four trajectories (i.e., x, y and z are polynomial functions of time of degree four), 45% of the running time of the simulation is spent isolating roots, and approximately 7% is spent refining the isolating intervals for the purposes of comparisons. The remaining portion of the running time accounts for the initialization of the kinetic data structure and the event processing (schedul-

ing/descheduling events and properly updating the data structure).

5.1 FunctionKernels customized for kinetic data structures

There are several modifications we can make to the real root isolation algorithms to optimize them for the case of kinetic data structures. Naively, there is no way to differentiate between a certificate which fails immediately when it is created and one whose function is momentarily zero, but will be valid immediately in the future. In order to handle such degeneracies we ensure that all certificate function generators produce certificate functions which are positive when the corresponding certificates are valid. Then, by looking at the sign of the function at a rational between the current time and the first root of the function after the current time, we can differentiate between a certificate which fails immediately (which is negative in this interval) and one which is simply degenerate (which is positive in the interval). In addition, this allows us, under the assumption that computations are performed exactly, to check the validity of all certificates upon creation.

This assumption is particularly useful when using numerical solvers. Without it there is no reliable way to handle a root of the certificate function which occurs near the current time. Due to numerical errors, the relative ordering may not be correct, so we need to use other cues to tell whether such a root is the certificate having become valid just before the current time (the certificate function is necessarily valid at the current time), or failing shortly in the future. Testing the derivative reliably disambiguates the two cases and is essential in order to get such kinetic data structures to work properly.

Recent work by Milenkovic et al. [15] provides more rigorous analysis for computing arrangements of algebraic curves using approximate polynomial solvers. Their algorithm is equivalent to running a kinetic data structures based sweepline construction with the heuristics mentioned here.

6 Timing results

Kinetic data structures simulations spend much of their time manipulating the certificate functions. These are quite complicated entities compared to static predicates and tend to completely dwarf any strictly combinatorial computations. Figure 5 shows qualitative results of where most of the computation time was spent for kinetic Delaunay triangulations in three dimensions. Generating the certificate functions and solving them took half or more of the total running time even when computing approximate roots. The figures should be

solver	*degree*	*generating*	*solving*	*comparing*
Sturm	1	35%	30%	15%
Sturm	6	20%	50%	10%
numeric	1	25%	30%	0%
numeric	6	45%	35%	0%

Fig. 5. Sample breakdowns of the running times of three-dimensional Delaunay triangulation kinetic data structures. The *degree* column is the degree of the motion (i.e., the degree of the coordinate functions as polynomial functions of time). The *generating* column refers to the cost of creating the certificate functions. The *solving* column is the cost of isolating or numerically approximating roots and *comparing* is how much work goes into ordering the roots. Delaunay triangulations have quite complicated certificate functions (the degree is five times that of the input trajectories), so the certificate generation and root isolation cost is comparatively high. Percentages were measured with `gprof` after compiling with optimizations with `g++` 4.0.2.

degree	*numeric*	*exact*	*CORE*
1	.02	1.3	3.1
2	.02	4.1	5.0
4	.96	8.1	11
8	5.5	51	62
16	35	158	180

Fig. 6. The table shows times per event in milliseconds for the sort kinetic data structure for various degree trajectories. The exact solver does not handle degree one (linear) functions as a special case so these timings could be improved.

taken as rough approximations since the framework is not as heavily optimized yet as most static Delaunay triangulation computations.

Numerical solvers are much cheaper than exact ones for low degree certificates, but the gap narrows as the degree increases. For linear or quadratic certificates, the gap is about a factor of one hundred, which narrows to a factor of five when the degree increases to 16. Figure 6 shows some example timings per event.

As a note, the exact solver and the CORE solver use more or less the same root isolation and root comparison algorithms. The speed difference is due primarily to implementation details, as well as our simpler requirements of only needing to support comparisons of roots rather than field operations on roots, as are supported by CORE::Exprs. CORE has narrowed the gap considerably with

recent releases and is now almost as fast.

7 Future work

Kinetic data structures are expensive compared to rebuilding in many circumstances. For an example involving Delaunay triangulations, see [16]. This is especially true when trying to do exact calculations: filtering is quite well developed for static computations but just in its infancy for polynomial computations. As shown in Section 6 there are several sources of extra expense

(1) generating the certificate polynomials
(2) isolating the roots of the certificate polynomials
(3) comparing the roots of polynomials

Even when inexact computations are used, the first two are significant sources of work, and either of the two can alone surpass the cost of building the structure from scratch once. In the context of updating Delaunay triangulations we now have filtering techniques which remove most of this overhead. These will be covered in an upcoming paper.

An alternative scheme is to integrate the event time comparison into the solver, i.e., the root isolation procedure is solely driven by the comparisons of event times — no preprocessing is performed. Such a scheme was successfully used for high degree polynomials in [5]. We have implemented such a scheme but, so far, the extra overhead outweighs any speed advantages.

Numerical approximations of roots of polynomials are much cheaper to obtain than their exact counterparts and are close to correct most of the time. While producing stable algorithms using only inexact predicates has proved to be an extremely difficult task for static data [17], we have good reason to expect it to be easier for kinetic data structures.

Let us hint on some empirical evidence as to why we expect that building stable algorithms on top of inexact predicates would be feasible in the case of kinetic Delaunay triangulations. First, we can run kinetic Delaunay triangulation simulations on data sets whose static Delaunay triangulation computations fail without exact computations. Each time a certificate fails and a flip occurs, all the neighboring certificates are checked. If any certificate is invalid, action will be immediately taken to correct it (by performing flips). So, unless many certificate failures are missed together, Delaunay flips will attempt to move the structure closer to correct. While we cannot guarantee that this process will converge for classical kinetic Delaunay triangulations [16] (in higher than one dimension), it generally does. If, instead of the normal kinetic Delaunay

data structure, which maintains a triangulation at all times, we implement Shewchuk's star splaying based algorithm [18], we should be able to guarantee that kinetic Delaunay will always converge if not too many certificates are missed.

Second, we have access to good, fast exact static predicates. These allow us to reliably determine if a particular part of a kinetic data structure is correct or not, and, if desired, statically patch it. The problem then becomes ensuring that, with high probability, we detect any missed certificate failures, by, for example, adding more long range certificates.

Acknowledgments

This research was partially supported by the NSF grant ITR-0086013.

References

[1] L. J. Guibas, M. I. Karavelas, D. Russel, A computational framework for handling motion, in: ALENEX '04: Proceedings of the sixth Workshop on Algorithm Engineering and Experiments, 2004, pp. 129–141.

[2] J. Basch, L. Guibas, J. Hershberger, Data structures for mobile data, in: SODA '97: Proceedings of the eighth ACM-SIAM Symposium on Discrete Algorithms, 1997, pp. 747–756.

[3] J. Basch, L. J. Guibas, C. D. Silverstein, L. Zhang, A practical evaluation of kinetic data structures, in: SCG '97: Proceedings of the thirteenth Annual ACM Symposium on Computational geometry, ACM Press, New York, NY, USA, 1997, pp. 388–390.

[4] J. Basch, L. Guibas, C. Silverstein, L. Zhang, The demokin library (1997).
URL `http://julien.b3h.org/demokin/demokin.html`

[5] L. J. Guibas, M. I. Karavelas, Interval methods for kinetic simulations, in: SCG '99: Proceedings of the fifteenth Annual ACM Symposium on Computational Geometry, 1999, pp. 255–264.

[6] D. Kirkpatrick, J. Snoeyink, B. Speckmann, Kinetic collision detection for simple polygons, International Journal of Computational Geometry and Applications 12 (1–2) (2002) 3–27.
URL `http://www.win.tue.nl/~speckman/demos/kcdsp/`

[7] V. Karamcheti, C. Li, I. Pechtchanski, C. Yap, A CORE library for robust numeric and geometric computation, in: SCG '99: Proceedings of the fifteenth Annual ACM Symposium on Computational Geometry, 1999, pp. 351 – 359.
URL `http://www.cs.nyu.edu/exact/core/`

[8] C. Burnikel, K. Mehlhorn, S. Schirra, The LEDA class real number, Tech. Rep. MPI-I-96-1-001, MPI für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany (1996).
URL http://www.mpi-sb.mpg.de/LEDA/

[9] K. Mehlhorn, S. Näher, LEDA: A Platform for Combinatorial and Geometric Computing, Cambridge University Press, Cambridge, UK, 2000.

[10] The Qt library.
URL http://www.trolltech.com/products/qt

[11] The Coin library.
URL http://www.coin3d.org

[12] D. W. Jones, An empirical comparison of priority-queue and event-set implementations, Communications of the ACM 29 (4) (1986) 300–311.

[13] The upper_bound algorithm, Standard Template Library (STL).
URL http://www.sgi.com/tech/stl/upper_bound.html

[14] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, L. Kettner, K. Mehlhorn, J. Reichel, S. Schmitt, E. Schömer, N. Wolpert, EXACUS: Efficient and Exact Algorithms for Curves and Surfaces, in: Proceedings of the thirteenth Annual European Symposium on Algorithms, 2005, pp. 155–166.

[15] V. Milenkovic, E. Sacks, An approximate arrangement algorithm for semi-algebraic curves, in: SCG '06: Proceedings of the twenty-second Annual ACM Symposium on Computational Geometry, 2006, pp. 237–246.

[16] L. Guibas, D. Russel, An empirical comparison of techniques for updating Delaunay triangulations, in: SCG '04: Proceedings of the twentieth Annual ACM Symposium on Computational Geometry, ACM Press, New York, NY, USA, 2004, pp. 170–179.

[17] D. Salesin, J. Stolfi, L. Guibas, Epsilon geometry: building robust algorithms from imprecise computations, in: SCG '89: Proceedings of the fifth Annual ACM Symposium on Computational Geometry, ACM Press, New York, NY, USA, 1989, pp. 208–217.

[18] R. Shewchuk, Star splaying: an algorithm for repairing Delaunay triangulations and convex hulls, in: SCG '05: Proceedings of the twenty-first Annual ACM Symposium on Computational Geometry, ACM Press, New York, NY, USA, 2005, pp. 237–246.