A computational framework for handling motion

Leonidas Guibas* Menelaos I. Karavelas† Daniel Russel‡

Abstract

We present a framework for implementing geometric algorithms involving motion. It is written in C++ and modeled after and makes extensive use of CGAL (Computational Geometry Algorithms Library) [4]. The framework allows easy implementation of kinetic data structure style geometric algorithms—ones in which the combinatorial structure changes only at discrete times corresponding to roots of functions of the motions of the primitives. This paper discusses the architecture of the framework and how to use it. We also briefly present a polynomial package we wrote, that supports exact and filtered comparisons of real roots of polynomials and is extensively used in the framework. We plan to include our framework in the next release of CGAL.

1 Introduction

Motion is ubiquitous in the world around us and is a feature of many problems of interest to computational geometers. While projects such as CGAL [15] have provided an excellent software foundation for implementing static geometric algorithms (where nothing moves), there is no similar foundation for algorithms involving motion. In this paper we present such a framework for algorithms that fit the constraints of kinetic data structures. Kinetic data structures were introduced by Basch et. al. in '97 [1, 12]. They exploit the combinatorial nature of most geometric data structures—the combinatorial structure remains invariant under some motions of the underlying geometric primitives and, when the structure does need to change, it does so at discrete times and in a limited manner. Algorithms that fit within the kinetic data structures framework have been found for a number of geometric constructs of interest, including Delaunay and regular triangulations in two and three dimensions and various types of clustering.

Computational geometry is built on the idea of *predicates*—functions of the description of geometric primitives which return a discrete set of values. Many of the predicates reduce to determining the sign of an algebraic expression of the representation (i.e. coordinates of points) of the geometric primitives. 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 proved by checking a finite number of predicates of the primitives. These predicates are called *certificates*. For example, a three-dimensional convex hull is proved to be correct by checking, for each face, that all points are below the outward facing plane supporting it.

The kinetic data structures framework is built on top of this view of computational geometry. Let the geometric primitives move by replacing each of their coordinates with a function of time. As time advances, the primitives now trace paths in space called *trajectories*. The values of the certificates which proved the correctness of the static structure now become functions of time, called the *certificate functions*. As long as these functions have the correct value, the original structure is still correct. However, if one of the certificate functions changes value, the original structure must be updated and some new set of certificate functions computed. We call such occurrences *events*.

Maintaining a kinetic data structure is then a matter of determining which certificate function changes value next (typically this amounts to determining which certificate function has the first root after the current time) and then updating the structure and certificate functions.

The CGAL project [9, 4] provides a solid basis for performing exact and efficient geometric computations as well as a large library of algorithms and data structures. A key idea they use is that of a computational kernel, an object which defines primitives, methods to create instances of primitives, and functors[1] which act on the primitives. CGAL defines a geometric kernel [15], which provides constant complexity geometric objects and predicates and constructions acting on them. The algorithms use methods provided by the kernel to access and modify the geometric primitives, so the actual representation need never be revealed. As a result, the implementation of the kernel primitives and predicates can be replaced, allowing different types of computation (such as fixed precision or exact) and different types of primitive representation (such as Cartesian or homoge-

*Stanford University, guibas@cs.stanford.edu
†University of Notre Dame, mkaravel@cse.nd.edu
‡Stanford University, drussel@graphics.stanford.edu

[1] Functors are C++ classes that provide one or more operator() methods.

neous coordinates) to be used. The library uses C++ templates to implement the generic algorithms which results in little or no run time inefficiency.

CGAL provides support for exact computation (as opposed to fixed precision computation with double-type numbers which can result in numerical errors), as do some other libraries, such as CORE [24] and LEDA [3]. Exact computation can greatly simplify algorithms as they no longer need to worry about numerical errors and can properly handle degeneracy. However, it can be painfully slow at run time. *Floating point filters* [10, 21] were developed to address this slowness when computing predicates. The key observation is that fixed precision computation often produces the correct answer. For example, in our point-plane orientation test from above, if the dot product of the point coordinates and the plane normal is very much larger than the plane offset, then the point is above the plane, even if the exact value of the dot product is in some doubt. Floating point filters formalize this observation by computing some additive upper bound on the error. Then, if the magnitude of value computed for the predicate is larger than that upper bound, the sign of the computed value is guaranteed to be correct, and the predicate value is known. However, if the computed value is not large enough, the calculation must be repeated using an exact number type. A variety of techniques have been developed to compute the error bounds. One of the easiest to use and tightest general purpose techniques is interval arithmetic [18, 2].

In this paper we present a framework for implementing kinetic data structures. We also provide a standalone library for manipulating and comparing real roots of polynomials using fixed precision and exact arithmetic and which can use floating point filters to accelerate the computations. We have used the framework for investigating kinetic data structure based techniques for updating Delaunay triangulations [14].

The framework depends on CGAL for a small number of low level classes and functions, mostly concerning evaluating and manipulating number types and performing filtered computations. In addition, it provides models of the CGAL geometric kernel to enable usage of static geometric data structures and algorithms on moving primitives.

Note on terminology: We adopt the terminology used by the C++ Standard Template Library and talk about concepts and models. A *concept* is a set of functionalities that any class which conforms to that concept is expected to implement. A class is a *model* of a concept if it implements that functionality. Concepts will be denoted using ThisStyle.

2 Design Considerations

There were a number of important considerations which guided our design. They are

- *Runtime efficiency*: There should be little or no penalty for using our framework compared to implementing your own more specialized and tightly integrated components. We use templates to make our code generic and modular. This allows most of the cost of the flexibility of the framework to be handled at compile time by the compiler rather than at runtime, and allows components to be easily replaced if further optimization is needed.

- *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, for example when a collision occurs. While such trajectory changes can not change the current state of a kinetic data structure since the trajectories are required to be C^0-continuous, they do affect the time when certificates fail. As a result there must be a central repository for the kinetic primitives which provides signals to the kinetic data structures when trajectories change. This repository, which is discussed in Section 3.6 can be easily omitted if there is no need for its extra functionality. A less obvious issue raised by having multiple kinetic data structures is that events from different kinetic data structures must be able to be stored together and have their times compared. The ramifications of this concern are discussed in Section 3.5.

- *Support for existing static data structures*: We provide functionality to aid in the use of existing static data structures, especially ones implemented using CGAL, by allowing static algorithms to act on snapshots of the running kinetic data structure. Our method of supporting this is discussed in Section 3.4.

- *Support exact and filtered computation*: Our polynomial solvers support exact root comparison and other operations necessary for exact kinetic data structures. In addition we support filtered computation throughout the framework. The effects of these requirements are discussed in Sections 3.2 and 3.3.

- *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 only share around 10 lines of code. An example of such a kinetic data structure is included in the appendix in Figure 2.

- *Extensibility and modularity*: The framework should be made of many lightweight components which the user can easily replace or extend. All components are tied together using templates so replacing any one model with another model of the same concept will not require any changes to the framework.
- *Ease of optimization*: We explicitly supported many common optimizations. The easy extensibility of the framework makes it easy to modify components if the existing structure is not flexible enough.
- *Ease of debugging of kinetic data structures*: We provide hooks to aid checking the validity of kinetic data structures as well as checking that the framework is used properly. These checks are discussed in Section 3.5. We also provide a graphical user interface which allows the user to step through the events being processed and to reverse time and to look at the history.

3 Architecture

3.1 Overview The framework is divided in to five main concepts as shown in Figure 1. They are:

- FunctionKernel: a computational kernel for representing and manipulating functions and their roots.
- KineticKernel: a class which defines kinetic geometric primitives and predicates acting on them.
- MovingObjectTable: a container which stores kinetic geometric primitives and provides notifications when their trajectories change.
- InstantaneousKernel: a model of the CGAL kernel concept which allows static algorithms to act on a snapshot of the kinetic data structure.
- Simulator: a class that maintains the concept of time and a priority queue of the events.

In a typical scenario using the framework, a Simulator and MovingObjectTable are created and a number of geometric primitives (e.g. points) are added to the MovingObjectTable. Then a kinetic data structure, for example a two dimensional kinetic Delaunay triangulation, is initialized and passed pointers to the Simulator and MovingObjectTable. The kinetic Delaunay triangulation extracts the trajectories of the points from the MovingObjectTable and the current time from the Simulator. It then uses an instance of an InstantaneousKernel to enable a static algorithm to compute the Delaunay triangulation of the points at the current time. An instance of a KineticKernel is used to compute the in_circle certificate function for each edge of the initial Delaunay triangulation. The kinetic data structure requests that the Simulator solve each certificate function and schedule an appropriate event. The Simulator uses the FunctionKernel to compute and compare the roots of the certificate functions.

Initialization is now complete and the kinetic data structure can be run. Running consists of the Simulator finding the next event and processing it until there are no more events. Here, processing an event involves flipping an edge of the Delaunay triangulation and computing five new event times. The processing occurs via a callback from an object representing an event to the kinetic Delaunay data structure.

If the trajectory of a moving point changes, for example it bounces off a wall, then the MovingObjectTable notifies the kinetic Delaunay data structure. The kinetic Delaunay data structure then updates all the certificates of edges adjacent to faces containing the updated point and reschedules those events with the Simulator.

A more detailed example will be discussed in Section 4. We will next discuss the principle concepts.

3.2 The polynomial package: The FunctionKernel, solvers and roots. The FunctionKernel is a computational kernel for manipulating and solving univariate equations. The polynomial package provides several models of the FunctionKernel all of which act on polynomials. The FunctionKernel defines three primitives: the Function, the Solver and the ConstructedFunction. The two nested primitives shown in Figure 1 are NT the number type used for storage and Root the representation type for roots. The kernel defines a number of operations acting on the primitives such as translating zero, counting the number of roots in an interval, evaluating the sign of a function at a root of another, finding a rational number between two roots, and enumerating the roots of a function in an interval. Our models additionally provide a number of polynomial specific operations such as computing Sturm sequences and Bézier representations of polynomials.

The Function concept (a polynomial in our models) supports all the expected ring operations, i.e., Functions can be added, subtracted and multiplied. The ConstructedFunction wraps the information necessary to construct a polynomial from other polynomials. The distinction between Functions and ConstructedFunctions is necessary in order to support filtering, discussed below. If no filtering is used, a ConstructedFunction is an opaque wrapper around a Function.

The Root type supports comparisons with other Roots and with constants and a few other basic operations such as generation of an isolating interval for the root, negation, and computation of its multiplicity. We plan to extend the Root to support full field operations, but have not done so yet.

The most important attributes differentiating our various models of the FunctionKernel are the type of solver used and the how the resulting root is represented. We currently provide five different solver types

- *Eigenvalue*: a solver which computes roots using fixed precision computation of the eigenvalues of a matrix.
- *Descartes*: a set of solvers which use Descartes rule [20] of signs to isolate roots in intervals.
- *Sturm*: a set of solvers which use Sturm sequences [25] to isolate roots. An earlier use of Sturm sequences in the context of kinetic data structures was published in [13].
- *Bézier*: a solver which used a Bézier curve based representation of the polynomial to perform root isolation [17].
- *CORE*: a solver which wraps the CORE Expr type [24].

We also provide a FunctionKernel specialized to handle linear functions, which can avoid much of the overhead associated with manipulating polynomials. We also plan to provide specialized function kernels for small degree polynomials using the techniques presented in [7, 8, 16]. All of the solvers except for the *Eigenvalue* solver can perform exact computations when using an exact number type and produce roots which support exact operations.

The simplest way to try to represent a root of a polynomial is explicitly, using some provided number type. This is used by numerical solvers (such as the our *Eigenvalue* solver), which represent roots using a double, and the CORE-based solver which represents the root using CORE's Expr type. However, since roots are not always rational numbers, this technique is limited to either approximating the root (the former case) or depending on an expensive real number type (the latter case).

An alternative is to represent roots using an isolating interval along with the polynomial being solved. Such intervals can be computed using Sturm sequences, Descartes rule of signs or the theory of Bézier curves. When two isolating intervals are compared, we subdivide the intervals if they overlap in order to attempt to separate the two roots. This subdivision can be performed (for simple polynomials) by checking the sign of the original polynomial at the midpoint of the interval. However, subdivision will continue infinitely if two equal roots are compared. To avoid an infinite loop, when the intervals get too small, we fall back on a Sturm sequence based technique, which allows us to exactly compute the sign of one polynomial at the root of another. This allows us to handle all root comparisons exactly.

We have variants of the Sturm sequence based solver and the Descartes rule of sign based solver, that perform filtered computations. Unfortunately, in a kinetic data structure, the functions being solved are certificate functions which are generated from the coordinate functions of the geometric primitives. If the coordinate functions are stored using a fixed precision type, then computing the certificate function naively will result in the solver being passed an inexact function, ending all hopes of exact comparisons. Alternatively, the certificate function generation could be done using an exact type, but this technique would be excessively expensive as fixed precision calculations are often sufficient. This means that in the kinetic data structures setting, the filtered root computation must have access to a way of generating the certificate function.

To solve this problem we introduce the concept of a FunctionGenerator. This is a functor which takes a desired number type as a parameter and generates a function. The computations necessary to generate the function are performed using the number type passed. The FunctionGenerator gets wrapped by a ConstructedFunction and passed to the solver. The filtered solvers can first request that the certificate function generation be performed using an interval arithmetic type, so the error bounds are computed for each coefficient of the certificate polynomial. The solver then attempts to isolate a root. In general, the root isolation computation involves determining the signs of modified versions of the generated polynomial. If some of the sign values cannot be determined (because the resulting interval includes zero), the solver requests generation of the certificate polynomial using an exact number type and repeats the calculations using the exact representation.

In a kinetic data structures situation, we are only interested in roots which occur after the last event processed. In addition, there is often an end time beyond which the trajectories are known not to be valid, or of no interest for the simulation. These two times define an interval containing all the roots of interest. The Bézier, Descartes and Sturm based solvers all act on intervals and so can capitalize on this extra information. The Simulator, described in Section 3.5, keeps track of these two time bounds and makes sure the correct values are passed to all instances of the solvers.

All of the solvers except for the CORE-based solver correctly handle non-square free polynomials. All exact solvers handle roots which are arbitrarily close together and roots which are too large to be represented by doubles, although the presence of any of these issues slows down computations, since filtering is no longer effective and we have to resort to exact computation. A qualitative comparison of the performance of our solvers can be found in Table 1. We plan to describe the

Solver:	Low degree	Wilkinson	Mignotte	Small Intervals	Non-simple
Eigenvalue	0.5	12	400	15	8
Filtered Descartes	18	230	9k	30	160
Descartes (double)	5	90	–	7	44
Descartes	32	2k	240k	150	750
Sturm	81	2.5k	12k	2.9k	780
Filtered Sturm	28	99	9k	55	130
CORE	291	126k	114k	2.5k	–
Bézier	143	19k	2M	29	180

Table 1: The time taken to isolate a root of various classes of polynomials is shown for each of the solvers. "Low degree" are several degree six or lower polynomials with various bounding intervals and numbers of roots. "Wilkinson" is a degree 15 Wilkinson polynomial which has 15 evenly spaced roots. "Mignotte" is a polynomial of degree 50 with two roots that are very close together. "Small Intervals" is a degree nine polynomial solved over several comparatively small parts of the real number line. "Non-simple" are non-square free polynomials. All of the solvers except the *Eigenvalue* and the *Descartes (double)* produce exact results. When roots are only needed from a small fraction of the real line, (the "High degree" test case), interval-based solvers are actually quite competitive with the numeric solvers, although the comparison of the resulting roots will be more expensive. Note the large running time on the Mignotte polynomials since the solvers have to fall back on a slower computation technique to separate the close roots. We believe the comparatively large running times of the Bézier based solver reflect the relative immaturity of our implementation, rather than a fundamental slowness of the method. The *Eigenvalue* solver is based on the GNU Scientific Library [11] and the ATLAS linear algebra package [23]. Times are in μs in a Pentium 4 running at 2.8GHz. "k" stands for thousands and "M" for millions of μs.

polynomial package in more detail in a later paper.

3.3 Kinetic primitives and predicates: the KineticKernel. The KineticKernel is the kinetic analog of the CGAL Kernel. It defines constant complexity kinetic geometric primitives, kinetic predicates acting on them and constructions from them. The short example in Section 3.1 uses the KineticKernel to compute the in_circle certificate functions. We currently provide two models which define two and three dimensional moving weighted and unweighted points and the predicates necessary for Delaunay triangulations and regular triangulations. The Function concept discussed in Section 3.2 takes the place of the ring concept of the CGAL Kernel and is the storage type for coordinates of kinetic primitives. As in CGAL, algorithms request predicate functors from the kernel and then apply these functors to kinetic primitives. There is no imperative programming interface provided at the moment.

In principle, kinetic predicates return univariate functions, so they should return a Function. However, as was discussed in Section 3.2, in order to support floating point filters, the input to a solver is a model of ConstructedFunction. As a result, when a predicate functor is applied no predicate calculations are done. Instead, a model of FunctionGenerator is created that stores the arguments of the predicate and can perform the necessary predicate calculations when requested. We provide helper classes to aid users in adding their own predicates to a KineticKernel.

It is important to note that the KineticKernel does not have any notion of finding roots of polynomials, of performing operations at the roots of polynomials, or of static geometric concepts. The first and second are restricted to the FunctionKernel (Section 3.2) and the Simulator (Section 3.5). The third is handled by the InstantaneousKernel which is discussed in the next section.

3.4 Connecting the kinetic and the static worlds: the InstantaneousKernel. There are many well implemented static geometric algorithms and predicates in CGAL. These can be used to initialize, test and modify kinetic data structures by acting on "snapshots" of the changing data structure. A model of the InstantaneousKernel concept is a model of the CGAL Kernel which allows existing CGAL algorithms to be used on such snapshots. For example, as mentioned in Section 3.1, with the InstantaneousKernel we can use the CGAL Delaunay triangulation package to initialize a kinetic Delaunay data structure. We can also use the InstantaneousKernel and static Delaunay triangulation data structure to manage insertion of new points into and deletion of points from a kinetic Delaunay triangulation. The InstantaneousKernel model redefines the geometric primitives expected by the CGAL algorithm to be their kinetic counterparts (or, in practice, handles to them). When the algorithm wants to compute a predicate on some geometric primitives, the InstantaneousKernel first computes the static representation of the kinetic primitives, and then uses these to compute the static predicate.

We are able to use this technique due to a couple of important features of the CGAL architecture. First

of all, the kernel is stored as an object in CGAL data structures, so it can have state (for the InstantaneousKernel the important state is the current time). Secondly, predicates are not global functions, instead they are functors that the algorithm requests from the kernel. This means that they too, can have internal state, namely a pointer to the InstantaneousKernel and this state can be set correctly when they are created. Then, when an algorithm tries to compute a predicate, the predicate functor asks the InstantaneousKernel to convert its input (handles to kinetic geometric primitives) into static primitives and can then use a predicate from a static CGAL kernel to properly compute the predicate value.

The pointer from the InstantaneousKernel predicate to the InstantaneousKernel object is unfortunate, but necessary. Some CGAL algorithms request all the predicate functors they need from the kernel at initialization and store those functors internally. Since a given CGAL object (i.e. a Delaunay triangulation) must be able to be used at several snapshots of time, there must be a way to easily update time for all the predicates, necessitating shared data. Fortunately, predicates are not copied around too much, so reference counting the shared data is not expensive.

Note that the time value used by our InstantaneousKernel model must be represented by a number type, meaning that it cannot currently be a model of Root. This somewhat limits the use of the InstantaneousKernel. We use it primarily for initialization and verification, neither of which need to occur at roots of functions. Some techniques for addressing verification will be discussed in the next section.

Let us conclude the discussion of the InstantaneousKernel concept by noting that we do not require that models of the static kernels used by the InstantaneousKernel be CGAL Kernels, but rather that they conform with the CGAL Kernel concept. The user has the ability to provide his/her own kernel models and may or may not use CGAL.

3.5 Tracking time: the Simulator. Running a kinetic data structure consists of repeatedly figuring out when the next event occurs and processing it. This is the job of the Simulator. It handles all event scheduling, descheduling and processing and provides objects which can be used to determine when certificate functions become invalid. Since events occur at the roots of certificate functions, the Root type defined by a FunctionKernel is used to represent time by the Simulator. In the example in Section 3.1 the kinetic Delaunay data structure requests that the Simulator determine when in_circle certificate functions become invalid and schedules events with the Simulator. The Simulator also makes sure the appropriate callbacks to the kinetic Delaunay data structure are made when certificates become invalid.

Our model of the Simulator is parameterized by a FunctionKernel and a priority queue. The former allows the solver and root type to be changed, so numeric, exact or filtered exact computation models can be used. The priority queue is by default a queue which uses an interface with virtual functions to access the events, allowing different kinetic data structures to use a single queue. It can be replaced by a queue specialized for a particular kinetic data structure if desired.

The Root concept is quite limited in which operations it supports—it effectively only supports comparisons. Roots cannot be used in computations or as the time value in an InstantaneousKernel. As a result, we take a somewhat more topological view of time.

Two times, t_0 and t_1, are considered *topologically equivalent* if no roots occur in the interval $[t_0, t_1]$. The lack of separating roots means that the function has the same sign over the interval. This idea can be extended to a set of kinetic data structures. When a simulation is running, if the time of the last event which occurred, t_{last}, and the time of the next event, t_{next}, are not equal, then the current combinatorial structures of all of the kinetic data structures are valid over the entire interval (t_{last}, t_{next}). In addition there is a rational value of time, t_r, which is topologically equivalent to all times in the interval. Computations can be performed at t_r since it can be easily represented. This flexibility is used extensively in the Simulator.

When such a t_r exists, the kinetic data structures are all guaranteed to be valid and non-degenerate and so can be easily verified. The Simulator can notify the kinetic data structures when this occurs and they can then use an InstantaneousKernel to perform self-verification.

We can also use this idea to check the correctness of individual certificates upon construction. We define a certificate to be invalid when the certificate function is negative. As a result it is an error, and a common sign of a bug in a kinetic data structure, to construct a certificate function whose value is negative at the time of construction. Unfortunately, the time of construction is generally a root and this check cannot be performed easily. However, we can find a time topologically equivalent to the current time for that function (or discover if no such time exists) and evaluate the function at that time. This is still a very expensive operation, but faster than the alternative of using a real number type.

In addition, in order to properly handle two events occurring simultaneously, the Simulator must check if the certificate function being solved is zero at the current time. If it is zero, and negative immediately

afterwords, then the certificate fails immediately. This can be checked in a similar manner.

Even roots of polynomials (where the polynomial touches zero but does not become negative) can generally be discarded without any work being done since they represent a momentary degeneracy. However, at an even root, the kinetic data structure is degenerate, and as a result is not easily verifiable. Since, kinetic data structures are generally written only to handle odd roots, when verification is being performed as above, each even root must be divided into two odd roots before being returned. These cases are handled properly by our Simulator and solvers.

3.6 Coordinating many kinetic data structures: the MovingObjectTable**.** A framework for kinetic data structures needs to have support for easily updating the trajectories of kinetic primitives, such as when a collision occurs. This requirement is in contrast to static geometric data structures where the geometric primitives never change and their representations are often stored internally to the data structure.

In the simple example presented in Section 3.1, the kinetic Delaunay triangulation queries the MovingObjectTable for all the moving points on initialization. Later, when the simulation is running, the MovingObjectTable notifies the kinetic Delaunay data structure whenever a point's trajectory changes.

The MovingObjectTable allows multiple kinetic data structures to access a set of kinetic geometric primitives of a particular type and alerts the kinetic data structures when a new primitive is added, one is removed, or a primitive's trajectory changes. Our model of the MovingObjectTable is actually a generic container that provides notification when an editing session ends (for efficiency, changes are batched together). There is no internal functionality specific to kinetic data structures or to a particular type of primitive. The user must specify what type of kinetic primitive a particular instance of the MovingObjectTable model will contain through a template argument (in the architecture diagram Figure 1, a three dimensional moving point is used as an example). This type is exposed as the Object type shown in the figure. To access an object, a Key which uniquely identifies the object within this container and which has a type specific to this container type is used.

The MovingObjectTable uses a notification system which will be briefly explained in Section 3.7 to notify interested kinetic data structures when a set of changes to the primitives is completed. The kinetic data structures must then request the keys of the new, changed and deleted objects from the MovingObjectTable and handle them accordingly. We provide helper classes to handle a number of common scenarios such as a kinetic data structure which is incremental and can handle the changes of a single object at a time (as is done in the example, Figure 2 in the appendix), or a kinetic data structure which will rebuild all certificates any time any objects are changed (which can be more efficient when many objects change at once). The user can easily add other policies as needed.

The MovingObjectTable model provided will not meet the needs of all users as there are many more specialized scenarios where a more optimized handling of updates will be needed. The general structure of the MovingObjectTable model can be extended to handle many such cases. For example, if moving polygons are used, then some kinetic data structures will want to access each polygon as an object, where as others will only need to access the individual points. This extra capability can be added without forcing any changes to existing (point based) kinetic data structures by adding methods to return modified polygons in addition to those which return changed points.

When trajectory changes happen at rational time values, the MovingObjectTable can check that the trajectories are C^0-continuous. Unfortunately, the situation is much more complicated for changes which occur at roots. Such trajectories cannot be exactly represented in our framework at this time.

The MovingObjectTable only knows about one type of kinetic primitive and has no concept of time, other kinetic primitives or predicates. When a kinetic data structure handles an insertion, for example, it must query the Simulator for an appropriate time value and generate primitives using the KineticKernel. A more detailed discussion of how to use the MovingObjectTable appears in Section 4.

3.7 Miscellaneous: graphical display, notification and reference management. We provide a number of different classes to facilitate graphical display and manipulation of kinetic data structures. There are two and three dimensional user interfaces based on the Qt [19] and Coin [6] libraries, respectively. We provide support for displaying two and three dimensional weighted and unweighted point sets and two and three dimensional CGAL triangulations. Other types can be easily added.

A number of objects need to maintain pointers to other independent objects. For example, each kinetic data structure must have access to the Simulator so that it can schedule and deschedule events. These pointers are all reference counted in order to guarantee that they are always valid. We provide a standard reference counting pointer and object base to facilitate this [5].

Runtime events must be passed from the MovingObjectTable and the Simulator to the kinetic data structures. These are passed using a simple, stan-

dardized notification interface. To use notifications, an object registers a proxy object with the MovingObjectTable or Simulator. This proxy has a method new_notification which is called when some state of the notifying object changes and is passed a label corresponding to the state that changed. For convenience in implementing simple kinetic data structures, we provide glue code which converts these notifications into function calls—i.e., the glue code converts the MovingObjectTable notification that a new object has been added, into the function call new_object on the kinetic data structure. The glue code is used in the example in Figure 2. The base class for the notification objects manages the registration and unregistration to guard against invalid pointers and circular dependencies. This notification model is described in [5].

4 Implementing a kinetic data structure

4.1 Sort_kds overview. Figure 2, located in the appendix, depicts a complete kinetic data structure, Sort_kds, implemented using our framework. The data structure being maintained is very simple: a list of the geometric objects in the simulation sorted by their x coordinate. However, it touches upon the most important parts of the framework. For a simple kinetic data structure like this, much of the code is shared with other kinetic data structures. We provide a base class that implements much of this shared functionality. However, we do not use it here in order to better illustrate the various parts of our framework.

Like most kinetic data structures the maintained data has two parts (in this case stored separately)

- the combinatorial structure being maintained, in this case in the list objects_ declared at the end of the class.
- the mapping between connections in the combinatorial structure and pending events. In this case the connections are pairs of adjacent objects in the sorted list. The mapping is stored in the map certificates_ using the key of the first point in the pair. When a pair is destroyed (because the objects are no longer adjacent), the event key stored in the mapping is used to deschedule the corresponding event.

As is characteristic of many kinetic data structures, Sort_kds defines a class Event, which stores the information for a single event, and has six main methods. The methods are:

- new_object: a point has been added to the simulation and must be added to the data structure.
- change_object: an point has changed its trajectory and the two certificates involving it must be updated
- delete_object: an point has been removed from the simulation. It must be removed from the data structure, the events involving it descheduled and a new event created.
- swap: an event has occurred and two objects are about to become out of order in the list and so must be exchanged.
- rebuild_certificate: for some reason, a predicate corresponding to a particular piece of the combinatorial structure is no longer valid or the action that was going to be taken in response to its failure is no longer correct. Update the predicate appropriately. This method is only called from within the kinetic data structure.
- validate_at: check that the combinatorial structure is valid at the given time.

The first three methods are called in response to notifications from the MovingObjectTable. The fourth method is called by Event objects. The last method is called in response to a notification from the Simulator.

4.2 Sort_kds in detail. On initialization the Sort_kds registers for notifications with a MovingObjectTable and a Simulator. It receives notifications through two proxy objects, mot_listener_ and sim_listener_, which implement the notification interface and call functions on the kinetic data structure when appropriate. We provide standard proxy objects, Moving_object_table_listener_helper and Simulator_kds_listener_, which are used, but implementers of kinetic data structures are free to implement their own versions of these simple classes. The MovingObjectTable proxy calls the new_object, delete_object and change_object methods of the kinetic data structure when appropriate. The Simulator proxy calls the validate_at method when there is a rational time value at which verification can be performed. See Section 3.5 for an explanation of when this occurs.

The proxy objects store the (reference counted) pointers to the MovingObjectTable and Simulator objects for later use. The Simulator pointer is used by the kinetic data structure to request the current time and schedule and deschedule events. The MovingObjectTable pointer is used to access the actual coordinates of the kinetic objects. Once initialization is completed, the behavior of the kinetic data structure is entirely event driven.

The first thing that will occur is the addition of a point to the MovingObjectTable which results in the new_object method being called. This method is passed a Key which uniquely identifies a point in the MovingObjectTable. The Sort_kds makes use of the InstantaneousKernel to properly handle the insertion by using a InstantaneousKernel-provided functor which compares the x coordinates of two objects

at the current instant of time. This functor is then passed to the STL [22] library function upper_bound which returns the location in the sorted list of the point before which the new point should be inserted to maintain a sorted order. The point is inserted and the new pairs created (the new point and the objects before and after it) must have certificates created for them and events scheduled. The rebuild_certificate function is called to handle updating the certificates. The rebuild_certificate function will also deschedule any previous certificates when necessary.

Note that this implementation assumes that new_object is only called at instants when there is a rational time topologically equivalent to the current root. The current_time_nt call made to the Simulator will fail otherwise–i.e. when two events occur simultaneously, a degeneracy. The easiest way to handle this is to postpone insertion until a non-degenerate rational time exists or to only insert objects at rational times. We ignore that issue in the example since handling it is somewhat situation dependent.

The rebuild_certificate function updates the certificate associated with a passed pair to make sure it is correct. It first checks if there is a previous event corresponding to the pair which needs to be descheduled, and if so requests that the Simulator deschedule it. Then a Solver is requested from the Simulator, passing in the ConstructedFunction created by the Kinetic-Kernel's Less_x_2 predicate applied to the pair of objects in question. Then an Event is created to exchange the two objects and scheduled in the Simulator at for that time. Note that the certificate function may not have any roots after the current time. In that case, the solver will return Root::infinity (this is a special value of the Root type representing $+\infty$). The Simulator detects this and will not schedule the associated event, but will instead return a placeholder Event_key.

The Event is in charge of alerting the Sort_kds that it needs to be updated when a particular certificate failure occurs. Typically event classes are very simple, effectively just storing a pointer to the kinetic data structure and an identifier for the combinatorial piece which needs to be updated in addition to the time when the update must occur. This certificate also stores a copy of the Solver for reasons which will be discussed in the next paragraph. In order to be handled by the Simulator, the Event class must have the following methods

- time() which returns the time at which the event occurs and
- set_processed(bool) which is called with the value true when the event occurs.

In addition, in order to ease debugging, it must be able to be output to an std::ostream.

The swap method is the update method in the Sort_kds. When a pair of objects is swapped, three old pairs of points are destroyed and replaced by three new pairs. Calls to rebuild_certificate handle the updating of the certificates between a point of the swapped pair and its outside neighbors in the list. The pair that has just been exchanged should be dealt with differently for optimal efficiency. The predicate function corresponding to the new ordering of the swapped pair is the negation of that for the old ordering (i.e. $x_k(t)-x_j(t)$ as opposed to $x_j(t)-x_k(t)$), and so has the same roots. As a result, the old Solver can be used to find the next root, saving a great deal of time. In addition, the event which is currently being processed does not need to be descheduled as it is deleted by the Simulator. Notice that the update method does not make any reference to time. This is necessary to properly support degeneracies, since few or no exact calculations can be made without a topologically equivalent rational time, which might not exist. The new_object method is mostly used for initialization and so can be assumed to occur at a non-degenerate time, the same assumption is less easily made about an event.

As described in Section 3.5, the Simulator can periodically send out notifications that there is a rational time at which all the kinetic data structures are non-degenerate and can be easily verified. The validate_at method is called in response to such a notification. Validation consists of using the InstantaneousKernel to check that each pair in the list is ordered correctly.

The remaining two methods, change_object and delete_object are only necessary if the the kinetic data structure wishes to support dynamic trajectory changes and removals. These methods are called by the mot_listener_ helper when appropriate.

That is all it takes to implement a kinetic data structure which is exact, supports dynamic insertions and deletions of objects, allows points to change motions on the fly, and allows a variety of solvers and motion types to be used without modifications.

5 Conclusions and future work

Our framework does not provide a mechanism for exactly updating the motions of objects at event times, for example bouncing a ball when it collides with a wall. Providing this functionality efficiently is non-trivial since, in general, the time of an event, t_e, is a Root which is not a rational number. The trajectory after the bounce is a polynomial in $t-t_e$ and hence will not have rational coefficients. One approach would be to represent the polynomial coefficients using a number type that represents real algebraic numbers (such as CORE Expr or an extended version of our Root type) and write solvers that handle this. While our solvers currently support this functionality (except for the

CORE based one), it is extremely slow and the bit complexity of the coefficients will rapidly increase with the number of trajectory modifications.

In many circumstances it is not necessary to know the new trajectory exactly, as long as the approximations preserve the continuity of the trajectory and do not violate any predicates. An alternative approach is then to find a polynomial with rational coefficients of some bounded bit complexity which is close to the exact new trajectory. Ensuring that the new trajectory does not violate any predicates can be slightly tricky, as can ensuring continuity. We have not worked out all the ramifications of this approach and whether it can be made fully general.

A third alternative would be to allow fuzzy motions—motions represented by polynomials whose coefficients are refinable intervals, for example, whose accuracy will depend on how accurately we need to know the motion. A root of such a polynomial cannot be known exactly and indeed may not exist at all, complicating matters. How to consistently process such events to give a generally meaningful and approximately correct simulation needs to be explored.

We are investigating extending filtering into more areas of the framework. For example, currently, the InstantaneousKernel must compute the static coordinates of the objects requested using an exact number type and then pass this exact representation to a static predicate. If the static predicate uses filtering, it will then convert the exact representation into an interval representation, and attempt to perform the predicate computation. In many cases this will be enough and the exact representation will never need to be used as is. A better alternative would be to initially generate an interval representation of the static objects and attempt the interval based predicate calculation. Only when that fails, compute the exact representation. CGAL provides support for all the necessary operations.

Acknowledgments This research was partly supported by the NSF grants ITR-0086013 and CCR-020448.

References

[1] J. Basch, L. Guibas, and J. Hershberger. Data structures for mobile data. In *Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 747–756, 1997.

[2] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Mathematics*, 109:25–47, 2001.

[3] C. Burnikel, J. Konemann, K. Mehlhorn, S. Naher, S. Schirra, and C. Uhrig. Exact geometric computation in LEDA. In *Proc. 11th Annual ACM Symposium on Computational Geometry*, pages 18–19, 1995.

[4]

[5] D. Cheriton. *CS249 Course Reader*. 2003.

[6]

[7] I. Z. Emiris and E. P. Tsigaridas. Comparison of fourth-degree algebraic numbers and applications to geometric predicates. Technical Report ECG-TR-302206-03, INRIA Sophia-Antipolis, 2003.

[8] I. Z. Emiris and E. P. Tsigaridas. Methods to compare real roots of polynomials of small degree. Technical Report ECG-TR-242200-01, INRIA Sophia-Antipolis, 2003.

[9] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL a computational geometry algorithms library. *Software– Practice and Experience*, 30(11):1167–1202, 2000.

[10] S. Fortune and C. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics (TOG)*, 15(3):223–248, 1996.

[11]

[12] L. Guibas. Kinetic data structures: A state of the art report. In *Proc. 3rd Workshop on Algorithmic Foundations of Robotics*, 1998.

[13] L. Guibas and M. Karavelas. Interval methods for kinetic simulations. In *Proc. 15th Annual ACM Symposium on Computational Geometry*, pages 255–264, 1999.

[14] L. Guibas and D. Russel. An empirical comparison of techniques for updating delaunay triangulations. Manuscript, 2004.

[15] S. Hert, M. Hoffman, L. Kettner, S. Pion, and M. Seel. An adaptable and extensible geometry kernel. In *Algorithm Engineering: 5th International Workshop, WAE 2001*, pages 79–91. Springer-Verlag Heidelberg, 2001.

[16] M. I. Karavelas and I. Z. Emiris. Root comparison techniques applied to computing the additively weighted voronoi diagram. In *Proc. 14th ACM-SIAM Symposium on Discrete Algorithms*, pages 320–329, 2003.

[17] B. Mourrain, M. Vrahatis, , and J.-C. Yakoubsohn. On the complexity of isolating real roots and computing with certainty the topological degree. *J. of Complexity*, 18(2):612–640, 2002.

[18] S. Pion. Interval arithmetic: an efficient implementation and an application to computational geometry. In *Proc. of the Workshop on Applications of Interval Analysis to Systems and Control with special emphasis on recent advances in Modal Interval Analysis MISC'99*, pages 99–109, 1999.

[19]

[20] F. Rouillier and P. Zimmerman. Efficient isolation of a polynomial real roots. Technical Report RR-4113, INRIA, February 2001.

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

[22]

[23] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS

project. *Parallel Computing*, 27(1–2):3–35, 2001.

[24] C. Yap. A new number core for robust numerical and geometric libraries. In *Proc. 3rd CGC Workshop on Geometric Computing*, 1998.

[25] C. Yap. *Fundamental problems of algorithmic algebra*. Oxford, 2000.

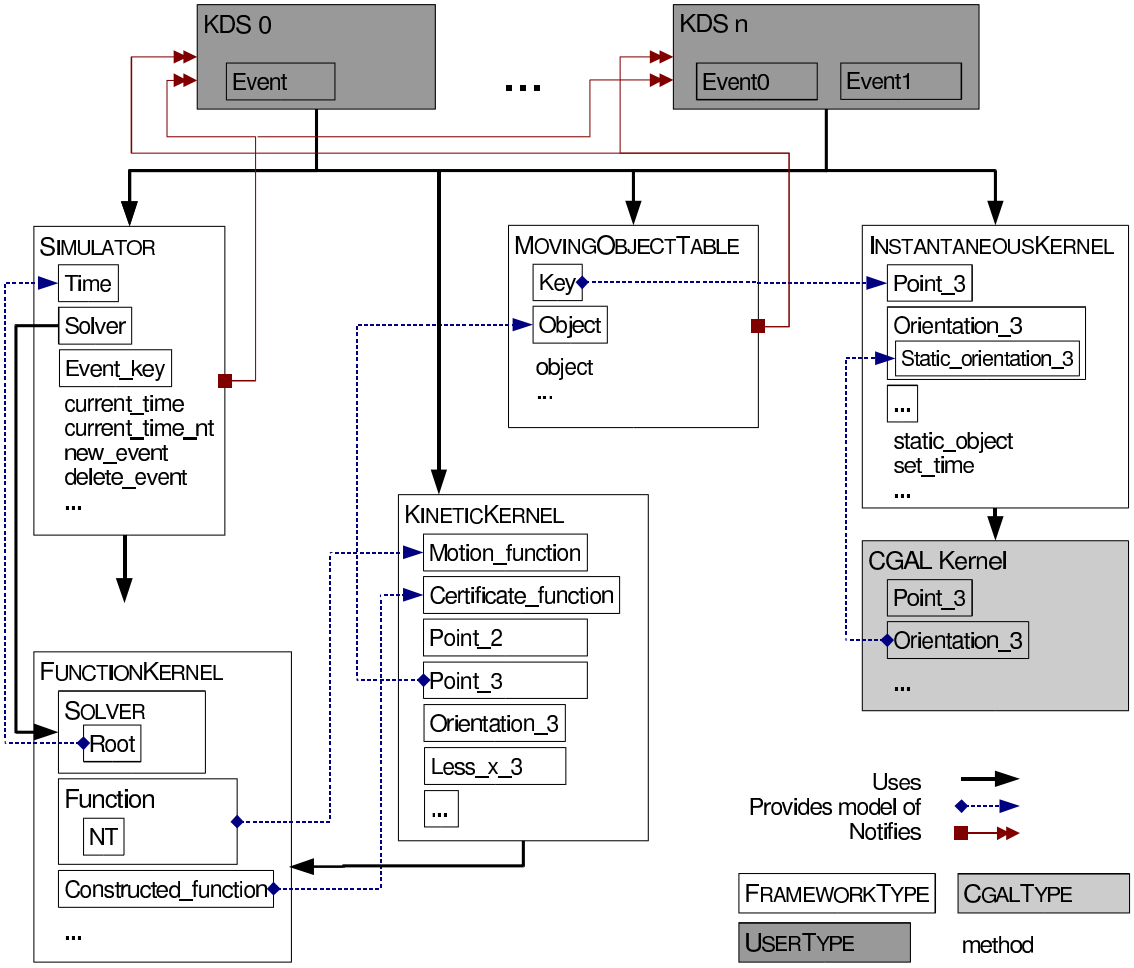

Figure 1: *Framework architecture:* Each large white box represents a main concept, the sub boxes their contained concepts, and regular text their methods. A "Uses" arrow means that a model of concept will generally use methods from (and therefore should take as a template parameter) the target of the arrow. A "Provides model of" arrow means that the source model provides an implementation of the destination concept through a `typedef`. Finally, a "Notifies" arrow means that the class notifies the other class of events using a standardized notification interface. See Section 3 for a description of each of the main concepts.

```
template <class Time, class Sort, class Id, class Solver>
class Swap_event;

// The template arguments are the KineticKernel, the Simulator
// and the MovingObjectTable.
template <class KK, class Sim, class MOT> class Sort_kds:
  // for ref counted pointers
  public CGAL::Ref_counted_base<Sort_kds< KK, Sim, MOT> > {
  typedef Sort_kds<KK, Sim, MOT> This;
  // The way the Simulator represents time.
  typedef typename Sim::Time Time;
  // A label for a moving primitive in the MovingObjectTable
  typedef typename MOT::Key Object_key;
  // A label for a certificate so it can be descheduled.
  typedef typename Sim::Event_key Event_key;
  // To shorten the names. Use the default choice for the static kernel.
  typedef typename CGAL::KDS::
  Cartesian_instantaneous_kernel<MOT> Instantaneous_kernel;
  // this is used to identify pairs of objects in the list
  typedef typename std::list<Object_key>::iterator iterator;
  typedef Swap_event<Time,This,iterator,typename Sim::Solver> Event;
  // Redirects the Simulator notifications to function calls
  typedef typename CGAL::KDS::
  Simulator_kds_listener<typename Sim::Listener,
                         This> Sim_listener;
  // Redirects the MovingObjectTable notifications to function calls
  typedef typename CGAL::KDS::
  Moving_object_table_listener_helper<typename MOT::Listener,
                                      This> MOT_listener;
public:
  typedef CGAL::Ref_counted_pointer<This> Pointer;

  // Register this KDS with the MovingObjectTable and the Simulator
  Sort_kds(typename Sim::Pointer sim, typename MOT::Pointer mot,
           const KK &kk=KK()): sim_listener_(sim, this),
                               mot_listener_(mot, this),
                               kernel_(kk), kernel_i_(mot) {}

  /* Insert k and update the affected certificates. std::upper_bound
     returns the first place where an item can be inserted in a sorted
     list. Called by the MOT_listener.*/
  void new_object(Object_key k) {
    kernel_i_.set_time(simulator()->current_time_nt());
    iterator it = std::upper_bound(sorted_.begin(), sorted_.end(),
                                   k,kernel_i_.less_x_2_object());
    sorted_.insert(it, k);
    rebuild_certificate(--it); rebuild_certificate(--it);
  }

  /* Rebuild the certificate for the pair of points *it and *(++it).
     If there is a previous certificate there, deschedule it.*/
  void rebuild_certificate(const iterator it) {
    if (it == sorted_.end()) return;
    if (events_.find(*it) != events_.end()) {
      simulator()->delete_event(events_[*it]); events_.erase(*it);
    }
    if (next(it)== sorted_.end()) return;
    typename KK::Less_x_2 less=kernel_.less_x_2_object();
    typename Sim::Solver s
      = simulator()->solver_object(less(object(*(it)),
                                        object(*next(it))));
    Time ft= s.next_time_negative();
    // the Simulator will detect if the failure time is at infinity
    events_[*it]= simulator()->new_event(Event(ft, it, Pointer(this),s));
  }

  /* Swap the pair of objects with *it as the first element. The old
     solver is used to compute the next root between the two points
     being swapped. This method is called by an Event object.*/
  void swap(iterator it, typename Sim::Solver &s) {
    events_.erase(*it);
    simulator()->delete_event(events_[*next(it)]);
    events_.erase(*next(it));
    std::swap(*it, *next(it));
    rebuild_certificate(next(it));
    Time ft= s.next_time_negative();
    events_[*it]= simulator()->new_event(Event(ft, it, this,s));
    rebuild_certificate(--it);
  }

  /* Verify the structure by checking that the current coordinates are
     properly sorted for time t. This function is called by the Sim_listener.*
  void validate_at(typename Sim::NT t) const {
    kernel_i_.set_time(t);
    typename Instantaneous_kernel::Less_x_2 less= kernel_i_.less_x_2_o
    for (typename std::list<Object_key>::const_iterator it
           = sorted_.begin(); *it != sorted_.back(); ++it){
      assert(!less(*it,*next(it)));
    }
  }

  /* Update the certificates adjacent to object k. This method is called by
     the MOT_listener. std::equal_range finds all items equal
     to a key in a sorted list (there can only be one).*/
  void change_object(Object_key k) {
    iterator it =  std::equal_range(sorted_.begin(), sorted_.end(),k).first;
    rebuild_certificate(it); rebuild_certificate(--it);
  }

  /* Remove object k and destroy 2 certificates and create one new one.
     This function is called by the MOT_listener.*/
  void delete_object(Object_key k) {
    iterator it =  std::equal_range(sorted_.begin(), sorted_.end(),k).first;
    sorted_.erase(it--);
    rebuild_certificate(it);
    simulator()->delete_event(events_[*it]);
    events_.erase(*it);
  }
  template <class It> static It next(It it){ return ++it;}
  typename MOT::Object object(Object_key k) const {
    return mot_listener_.notifier()->object(k);
  }
  Sim* simulator() {return sim_listener_.notifier();}

  Sim_listener sim_listener_;
  MOT_listener mot_listener_;
  // The points in sorted order
  std::list<Object_key> sorted_;
  // events_[k] is the certificates between k and the object after it
  std::map<Object_key, Event_key > events_;
  KK kernel_;
  Instantaneous_kernel kernel_i_;
};

/* It needs to implement the time() and process() functions and
   operator<< */
template <class Time, class Sort, class Id, class Solver>
class Swap_event {
public:
  Swap_event(const Time &t, Id o, typename Sort::Pointer sorter,
             const Solver &s): left_object_(o), sorter_(sorter), s_(s), t_(t){}
  void set_processed(bool tf){
    if (tf==true) sorter_->swap(left_object_, s_);
  }
  const Time &time() const {return t_;}
  Id left_object_; typename Sort::Pointer sorter_; Solver s_; Time t_;
};
template <class T, class S, class I, class SS>
std::ostream &operator<<(std::ostream &out,
                         const Swap_event<T,S,I,SS> &ev){
  return out << "swap " << *ev.left_object_ << " at " << ev.t_;
}
```

Figure 2: *A simple kinetic data structure:* it maintains a list of points sorted by their x coordinate. The code is complete and works as printed. Insertions and deletions are in linear time due to the lack of an exposed binary tree class in STL or CGAL. Support for graphical display is skipped due to lack of space.