A Computational Basis for Higher-Dimensional Computational Geometry and Applications[1]

Kurt Mehlhorn[2] Michael Müller[3] Stefan Näher[4] Stefan Schirra[2] Michael Seel[2]
Christian Uhrig[2] Joachim Ziegler[2]

Abstract

In this paper we describe and discuss a kernel for higher-dimensional computational geometry and we present its application in the calculation of convex hulls and delaunay triangulations. We introduce the basic data types like points, vectors, directions, hyperplanes, segments, rays, lines, spheres, affine transformations, and operations connecting these types. The description consists of a motivation for the basic class layout as well as topics like layered software design, runtime correctness via checking routines and documentation issues. Finally we shortly describe the usage of the kernel in the application domain.

1 Introduction

A growing community within computer science, academia, and industry tries to transfer the theoretical algorithmic knowledge to practical usable programs. What already happened in other parts of computer science, namely the development of software libraries to speed up program implementation is now also an issue within geometric computing. We are now in a situation where the available computing power, the recent developments concerning exact arithmetic packages, and the identification of reasonable geometric primitives allow us to design a programming toolbox for this purpose.

We describe and discuss a kernel for higher-dimensional computational geometry. We have implemented all basic data types like points, vectors, directions, hyperplanes, segments, rays, lines, spheres, affine transformations, and operations connecting these types. The kernel is structured into three layers.

- arbitrary precision integer and rational arithmetic (classes *integer* and *rational*),
- exact linear algebra (classes *integer_vector* and *integer_matrix*), and
- basic geometric objects (classes *dd_rat_point*, *dd_rat_vector*, *dd_rat_direction*, *dd_rat_hyperplane*, *dd_rat_segment*, *dd_rat_ray*, *dd_rat_line*, *dd_rat_sphere*, and *aff_transformation*)

On top of the kernel we implemented some classical computation tasks in higher-dimensional CG like convex hulls and Delaunay triangulations. To make the kernel a toolbox for a wide user community and to give the whole project some paedagocical value we aimed for the following:

Ease of use — we aimed for a natural and intuitive interface as far as construction of objects, the conversion between objects, and the interaction of the classes and the operations are concerned. The naming scheme tries to achieve a compromise between mnemonics and word length. We followed a clean, complete and adaptable documentation scheme which provides all necessary information for the user of the kernel and at the same time is integrated into the implementation to enforce consistency between implementation and documentation. The information provided by this manual production toolset consists of prototype information, semantic preconditions, helpful implementation details and runtime information.

Functionality — we tried to provide a comprehensive functionality of the objects while avoiding interface bloating. The identification of the set

[1]This work was supported by ESPRIT LTR Project 20244 (ALCOM IT) and ESPRIT Project 21957 (CGAL)

[2]Max-Planck-Institute für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany

[3]RIB Bausoftware GmbH, Vaihinger Str. 151, 70507 Stuttgart

[4]Fachbereich Mathematik und Informatik, Martin-Luther Universität Halle-Wittenberg, 06120 Halle, Germany

Computational Geometry '97 Nice France
Copyright 1997 ACM 0-89791-878-9 97 06 ..$3.50

of primitives for higher level geometric applications was partly a dynamic process influenced by application design (see part 7).

Layered Design — we designed the kernel in a layered fashion for several reasons. First, the functionality of the lower levels is interesting in its own right. In particular, it can be used to realize additional geometric primitives. Second, the fact that the linear algebra layer provides extensive testing and checking routines considerably simplified the development of the geometry layer.

Efficiency — the code is designed to be as fast as possible respecting our primary goal: to develop modular, reusable and maintainable components which are not prone to arithmetic shortcomings like rounding errors. To optimize the runtime behaviour we use the LEDA memory management and a handle-rep scheme to improve memory consumption and to allow identity tests on objects.

The design of our kernel was mainly influenced by three sources: the experiences with the two-dimensional LEDA geometry kernel [MNU95], our experiences with an experimental higher-dimensional kernel [MZ94], and discussions with the group developing the CGAL-kernel [FGK+96].

In this short paper we can only give an overview of the kernel and the applications on top of it. We refer the reader to our website `http://www.mpi-sb.mpg/~seel` for the complete set of manual pages and for the complete documentation of the kernel and to `http://www.mpi-sb.mpg.de/~mehlhorn/Programs.html` for the application packet.

In the following parts we describe the three software layers of the kernel and the application layer, give implementation details, and report about experience with the two software packets.

2 Arithmetic

The bottom layer of our kernel is exact integral computing. We use the LEDA datatypes *integer* and *rational*. Any other bignum package providing the required functionality could be used instead.

The LEDA type *integer* realizes the mathematical type integer. The arithmetic operations $+$, $-$, $*$, $/$, $+=$, $-=$, $-$ (unary), $++$, $--$, the modulus operation (%, % =), bitwise AND (&, &=), bitwise OR (|,|=), the complement operator ($\sim$), the shift operators ($\ll$, $\gg$), the comparison operators $<$, $\leq$, $>$, $\geq$, $==$, $!=$, and the stream operators are all available. These operations never overflow and always yield the exact result. Of course, they may run out of memory.

Integers are essentially implemented by a vector of unsigned longs. The sign and the size are stored in extra variables. Some time critical functions are implemented in Sparc assembler code. The running time of addition is linear and the running time of multiplication is $O(L^{\log 3})$, where L is the length of the operands. (Karazuba-Offman for multiplication)

A LEDA *rational* is essentially a pair of integers. The arithmetic operations $+$, $-$, $*$, $/$, $+=$, $-=$, $*=$, $/=$, $-$ (unary), $++$, $--$ are available on rationals. In addition, there are functions to extract the numerator and denominator, to cancel out the greatest common divisor of numerator and denominator, to compute squares and powers, to round rationals to integers, and many others. LEDA's rational numbers are not necessarily normalized, i.e., numerator and denominator of a rational number may have a common factor. A call *p.normalize( )* normalizes p. This involves a gcd-computation to find the common factor m in numerator and denominator and two divisions to remove the m. Since normalization is a fairly costly process we do not do it automatically.

We have run some runtime tests to compare the LEDA *integers* with the GNU *Integers* included in GNU's C++ library. The tests have been executed on an Ultrasparc 140. We tested the four basic operations $+$, $-$, $*$, $/$ iteratively on a table of k-bit random numbers.

#Bits	#Ops	LEDA | GNU in seconds			
		+		−	
32	10^6	1.1	5.1	1.3	5.5
64	10^6	1.2	5.0	1.3	5.4
500	10^6	1.8	7.3	1.86	7.8
1000	10^6	2.7	10.4	2.8	13.4
		*		/	
32	10^6	1.6	7.5	4.7	26.6
64	10^6	4.2	10.9	24.8	36.6
500	10^6	158.8	272.7	225.7	375.3
1000	10^6	591.6	1063.4	736.5	1281.3

3 Linear Algebra

The second layer of our kernel provides the standard operations of linear algebra encapsulated in the two classes *integer_vector* and *integer_matrix*. These are vectors and matrices with integer entries (= entries of type *integer*). The possible operations on matrices and vectors include

construction mechanisms — we can construct m-vectors, $m \times n$-matrices from a tuple of equidimensional vectors, identity matrices etc.

data access operations — we can access the integral and rational components, the dimensions of matrices and vectors, and the rows and columns of matrices.

arithmetic operations — we provide the inner product, scalar multiplication to vectors and matrices, and vector- and matrix-addition and -multiplication.

Figure 1: The Manual Page of class integer_matrix, an excerpt

1. Definition

An instance of data type *integer_matrix* is a matrix of integer variables. The types integer_matrix and integer_vector together realize many functions of basic linear algebra. All functions on integer matrices compute the exact result, i.e., there is no rounding error. Most functions of linear algebra are *checkable*, i.e., the programs can be asked for a proof that their output is correct.

2. Creation

integer_matrix M(*int* $n = 0$, *int* $m = 0$);

creates an instance M of type integer_matrix, M is initialized to the $n \times m$ - zero matrix.

integer_matrix M(*array<integer_vector>* A);

creates an instance M of type integer_matrix. Let A be an array of m column-vectors of common dimension n. M is initialized to an $n \times m$ matrix with the columns as specified by A.

3. Operations

int	M.dim1( )	returns n, the number of rows of M.
integer_vector&	M.row(*int* i)	returns the i-th row of M (an m-vector). *Precondition*: $0 \leq i \leq n-1$.
integer_matrix	$M + M1$	Addition of two matrices. *Precondition*: M.dim1() = $M1$.dim1() and M.dim2() = $M1$.dim2().
bool	linear_solver(*integer_matrix* M, *integer_vector* b, *integer_vector&* x, *integer&* D, *integer_matrix&* *spanning_vectors*, *integer_vector&* c)	determines the complete solution space of the linear system $M \cdot x = b$. If the system is unsolvable then $c^T \cdot M = 0$ and $c^T \cdot b \neq 0$. If the system is solvable then $(1/D)x$ is a solution, and the columns of *spanning_vectors* are a maximal set of linearly independent solutions to the corresponding homogeneous system. *Precondition*: M.dim1() = b.dim().

4. Implementation

The datatype integer_matrix is implemented by two-dimensional arrays of integers. Operations *determinant*, *inverse*, *linear_solver*, and *rank* use $O(n^3)$ arithmetic operations, *col* takes time $O(n)$, *row*, *dim1*, *dim2*, take constant time, and all other operations take time $O(nm)$. The space requirement is $O(nm)$.

operations based on solving a linear system — we can determine the solution of a linear system $Ax = b$, calculate the determinant, rank, inverse, and independent columns of a matrix

The core operation of the last category is a Gaussian elimination scheme for a non-homogeneous linear system $Ax = b$ as described by J. Edmonds [Edm67]. For a recent reference see the books of A. Schrijver [Sch86, part I] or C. Yap [Yap93, lecture X]. Basically we transform the original matrix into a diagonalized matrix of integral entries which encodes the numerators of the solution vector and accumulate in parallel a common denominator of the numerator entries.

Figure 1 shows an excerpt of the manual page of class *integer_matrix*. The same style is used for all other types of the kernel. Each implemented class is documented by four major sections **Definition**, **Creation**, **Operations**, and **Implementation**. The first part gives an overview of the class specification like intended usage of the class and implementation features which influence this usage like exact arithmetic and proof features. The second and third part describe the user accessible operations of the class like constructors, member operations, and functions which work on the class. The description of each operation gives the semantics of it, describes the result transfer by return values and reference parameters, and states preconditions for the usage of the operation. The fourth part of our manual page completes the description by helpful implementation details and runtime and space bounds.

The cited description of the friend function *linear_solver* shows nicely the proof feature of our core operation. Either a solution x is calculated which can be easily checked by substitution into the linear system $Mx = b$ or a vector c is provided which proves the unsolvability of the system. Of course there's also a selftest incorporated in the code which can be switched on by a compilation flag and thus the testing can be done permanently.

In a second runtime test we compared the LEDA linear algebra module with commercial math packages like *Maple V* and *Mathematica 2.0*. We solved randomly generated non-homogeneous linear systems with *dim* rows and columns and 32 bit entries. The tests were executed on a Sun Sparc 4 with 40 MHz.

dim	LEDA	Maple V R3	Mathematica 2.0
20	1.3 s	15.8 s	31.6 s
30	7.6 s	92.4 s	211.3 s
40	28.4 s	363.7 s	840.3 s
100	32 min	> 1 h	> 1 h

4 Geometric Classes

As mentioned above we provide the geometric classes *dd_rat_point*, *dd_rat_vector*, *dd_rat_direction*, *dd_rat_hyperplane*, *dd_rat_segment*, *dd_rat_ray*, *dd_rat_line*, *dd_rat_sphere*, and *aff_transformation*.

We first give a motivation for our interface design and review briefly the basics of analytical geometry.

We use d to denote the dimension of the ambient space and assume that our space is equipped with a standard Cartesian coordinate system. The basic object within this space is a point p, which we identify with its cartesian coordinate vector $p = (p_0, \ldots, p_{d-1})$, where the p_i, $0 \leq i < d$, are rational numbers. We store a *dd_rat_point* by homogeneous coordinates $(h_0, \ldots, h_d)$ where $p_i = h_i / h_d$ for all i, $0 \leq i < d$, and the h_i's are integer (LEDA type *integer*). The homogenizing coordinate h_d is always positive.

Points, vectors, and directions are closely related but nevertheless clearly distinct types. In order to work out the relationship, it is useful to identify a point with an arrow extending from the origin (= an arbitrary but fixed point) to the point. In this view a point is an arrow attached to the origin. A vector is an arrow that is allowed to float freely in space, more precisely, a vector is an equivalence class of arrows where two arrows are equivalent if one can be moved into the other by a translation of space. Points and vectors can be combined by some arithmetical operations. For two points p and q the difference $p - q$ is a vector (= the equivalence class of arrows containing the arrow extending from q to p) and for a point p and a vector v, $p + v$ is a point.

All operations of linear algebra apply to vectors, i.e., vectors can be stretched and shrunk (by multiplication with a scalar) and inner and cross product applies to them. On the other hand, geometric tests like collinearity or orientation only apply to points. Note that we distinguish the vector type *dd_rat_vector* in this scenario of geometric objects from the data type *integer_vector*, which we use to formulate calculations in our arithmetic linear algebra layer. We cannot identify both because their role and thereby their functionality within their respective code module is quite different and we don't want to have this mixed up.

A direction is also an equivalence class of arrows, where two arrows are equivalent if one can be moved into the other by a translation of space followed by stretching or shrinking. Alternatively, we may view a direction as a point on the unit sphere. In two-dimensional space directions correspond to angles. As in the case of *dd_rat_point* we store *dd_rat_vector*, and *dd_rat_direction*, respectively, as a homogeneous tuple of integers with positive homogenizing component.

The common one-dimensional straight-line objects in d-space like lines, rays and segments (which we allow to be trivial) are implemented in the classes *dd_rat_line*, *dd_rat_ray* and *dd_rat_segment* and determined by a pair of points.

With respect to the user interface we can group together points, vectors, directions, and on the other hand segments, rays, and lines. For the first group there are common operations to access Cartesian and homogeneous coordinates. Conversions within the first group can be made by explicit operations. For the second group there are similar operations to access the coordinates of the determining pair of points.

Oriented hyperplanes in the class *dd_rat_hyperplane* can be used to model halfspaces and affine hulls of $(d-1)$-dimensional point sets. They are internally stored as a $(d+1)$-tuple of integer coefficients. Finally oriented spheres of type *dd_rat_sphere* are helpfull in proximity calculations like Voronoi diagrams or Delaunay triangulations. They are stored as a tuple of $d+1$ *dd_rat_point*s.

For all of our basic geometric types we have affine transformations, which can be used by a call of a common member operation which gets an *aff_transformation*-instance as an argument and delivers a transformed object.

All object classes mentioned use a common handle-rep scheme which is already used in many modules of LEDA. We distinguish between a front-end object which is created by the constructor and a storage object of concrete geometric information which is referenced from the front-end object. The advantages of this scheme emerge in case of frequent copy construction and assignment where only references have to be redirected and no geometric information has to be copied. For large objects this lessens memory consumption and allows us to improve equality checks by testing the reference addresses before a comparison of geometric coordinate information. As in the case of our linear algebra module we use LEDA's improved memory management module which gives us a certain speed-up compared to the standard C++ allocation scheme.

We now take a closer look at two example manual pages, by which we elaborate further on the features of our data types. Figure 2 shows an excerpt from the point class manual page.

The default dimension of all objects is 2. This means that a call to the standard constructor *dd_rat_point*( ) delivers an instance of type *dd_rat_point* for planar geometry. A point in d-dimensional space is constructed by *p*(*integer_vector c*, *integer D*) or *p*(*integer_vector c*). In addition, there are standard initialization operations which allow comfortable creation of objects for the dimensions 2 and 3, which are not shown here. The data access operations allow access to the dimension and to Cartesian and homogeneous coordinates. Operator overloading allows the intuitive calculation of *dd_rat_point* difference, which results in a *dd_rat_vector* and the translation of *dd_rat_points* by adding a *dd_rat_vector*. The orientation predicate and other affine operations use LEDA *array*s as a container type for a tuple of points.

To represent the second group of straight line objects we show a piece of the *dd_rat_line* documentation in figure 3. The basic operations provided on these objects are mainly position checks of points with respect to the objects, like *contains*( ) and intersection calculation between all higher dimensional objects like segments, rays, lines, and hyperplanes.

Figure 2: The Manual Page of class dd_rat_point, an excerpt

1. Definition
An instance of data type *dd_rat_point* is a point with rational coordinates in an arbitrary dimensional space ...

2. Creation
dd_rat_point p(int d = 2); introduces a variable p of type *dd_rat_point* in d-dimensional space.
dd_rat_point p(integer_vector c, integer D);
introduces a variable p of type *dd_rat_point* initialized to the point with homogeneous coordinates $(\pm c_0, \ldots, \pm c_{d-1}, \pm D)$, where d is the dimension of c and the sign chosen is the sign of D.
Precondition: D is non-zero.

3. Operations

int	p.dim( )	returns the dimension of p.
rational	p.coord(*int i*)	returns the i-th cartesian coordinate of p.
integer	p.hcoord(*int i*)	returns the i-th homogeneous coordinate of p.
dd_rat_vector	$p - q$	returns $p - q$. *Precondition*: $p.dim( ) == q.dim( )$.
dd_rat_point	$p +$ *dd_rat_vector v*	returns $p + v$. *Precondition*: $p.dim( ) == v.dim( )$.
int	orientation(*array<dd_rat_point> A*)	determines the orientation of the points in A. This is the sign of the determinant $\begin{vmatrix} 1 & 1 & 1 & 1 \\ A[0] & A[1] & \ldots & A[d] \end{vmatrix}$ where $A[i]$ denotes the cartesian coordinate vector of the i-th point in A. *Precondition*: A consists of $d + 1$ points of dimension d.
bool	contained_in_affine_hull(*array<dd_rat_point> A, dd_rat_point x*)	determines whether x is contained in the affine hull of the points in A.

4. Implementation
Points are implemented by arrays of integers as an item type. All operations like ...

Figure 3: The Manual Page of class dd_rat_line, an excerpt

1. Definition
An instance of data type *dd_rat_line* is an oriented line in d-dimensional Euclidian space ...

2. Creation
dd_rat_line l(int d = 2); introduces a variable l of type *dd_rat_line* and initializes it to some line in d-dimensional space
dd_rat_line l(dd_rat_point p, dd_rat_point q);
introduces a line through p and q and oriented from p to q.
Precondition: p and q are distinct and have the same dimension.

3. Operations

dd_rat_point	l.point1( )	returns a point on l.
dd_rat_direction	l.direction( )	returns the direction of l.
bool	l.contains(*dd_rat_point p*)	returns true if p lies on l and false otherwise.
int	l.intersection(*dd_rat_segment s, dd_rat_point& i1, dd_rat_point& i2*)	returns the intersection set $l \cap s$ by the following means. The return value is one of the constants {*IS_EMPTY, IS_POINT, IS_SEGMENT, IS_RAY, IS_LINE*}. The corresponding set is determined by the two points *i1* and *i2*:

return value	intersection set
IS_EMPTY	empty
IS_POINT	*dd_rat_point(i1)*
IS_SEGMENT	*dd_rat_segment(i1, i2)*
IS_RAY	*dd_rat_ray(i1, i2)*
IS_LINE	*dd_rat_line(i1, i2)*

4. Implementation
Lines are implemented by a pair of points as an item type ...

5 Applications: Convex Hulls and Delaunay Triangulations

The convex hull and the Delaunay triangulation problem are traditionally specified as functions, i.e., given a set of points compute their convex hull or their Delaunay triangulation in some representation. We specify both problems as data types that support insertions and a large variety of query operations. In the case of convex hulls we support navigation through the interior and the boundary of the hull and we support membership and visibility queries. In the case of Delaunay triangulations we support navigation through the triangulation, we support locate[1] and nearest neighbor queries, and we support range queries with spheres and simplices. For two-dimensional convex hulls and Delaunay triangulations we also support an interface to the LEDA graph and window classes [MNU95, MN95]. In this way one can, for example, construct two-dimensional nearest and furthest site Voronoi diagrams and minimum spanning trees, display hulls and Delaunay triangulations.

The next two sections present parts of the specifications of convex hulls and Delaunay triangulations, respectively.

Convex Hulls

An instance C of type *chull* is the convex hull of a multi-set S of points in d-dimensional space. We call S the underlying point set and d or *dim* the dimension of the underlying space. We use *dcur* or *dcurrent* to denote the affine dimension of S. The data type supports incremental construction of hulls.

The closure of the hull is maintained as a simplicial complex, i.e., as a collection of simplices the intersection of any two is a face of both[2]. In the sequel we reserve the word simplex for the simplices of dimension *dcur*. For each simplex there is an item of type *ch_simplex* and for each vertex there is an item of type *ch_vertex*. Each simplex has $1 + dcur$ vertices indexed from 0 to *dcur*; for a simplex s and an index i, $C.vertex(s, i)$ returns the i-th vertex of s. For any simplex s and any index i of s there may or may not be a simplex t opposite to the i-th vertex of s. The function $C.opposite\_simplex(s, i)$ returns t if it exists and returns *nil* otherwise. If t exists then s and t share *dcur* vertices, namely all but the vertex with index i of s and the vertex with index $C.index\_of\_vertex\_in\_opposite\_simplex(s, i)$ of t. Assume that t exists and let $j = C.index\_of\_vertex\_in\_opposite\_simplex(s, i)$. Then $s = C.opposite\_simplex(t, j)$ and $i = C.index\_of\_vertex\_in\_opposite\_simplex(t, j)$. Again, the specification and semantics of all operations connected to *chull* is documented in a manual page shown partly in figure 4.

[1] a locate query finds the simplex of the triangulation containing the query point

[2] The empty set is a facet of every simplex.

Delaunay Triangulations

A instance DT of type *dd_delaunay* is the nearest and furthest site Delaunay triangulation of a set S of points in some d-dimensional space. We call S the underlying point set and d or *dim* the dimension of the underlying space. We use *dcur* or *dcurrent* to denote the affine dimension of S. The data type supports incremental construction of Delaunay triangulations and various kinds of query operations (in particular, nearest and furthest neighbor queries and range queries with spheres and simplices).

A Delaunay triangulation is a simplicial complex. All simplices in the Delaunay triangulation have dimension *dcur*. In the nearest site Delaunay triangulation the circumsphere of any simplex in the triangulation contains no point of S in its interior. In the furthest site Delaunay triangulation the circumsphere of any simplex contains no point of S in its exterior. If the points in S are co-circular then any triangulation of S is a nearest as well as a furthest site Delaunay triangulation of S. If the points in S are not co-circular then no simplex can be a simplex of both triangulations. Accordingly, we view DT as either one or two collection of simplices. If the points in S are co-circular there is just one collection: the set of simplices of some triangulation. If the points in S are not co-circular there are two collections. One collection consists of the simplices of a nearest site Delaunay triangulation and the other collection consists of the simplices of a furthest site Delaunay triangulation.

For each simplex of maximal dimension there is an item of type *dt_simplex* and for each vertex of the triangulation there is an item of type *dt_vertex*. Each simplex has $1 + dcur$ vertices indexed from 0 to *dcur*. For any simplex s and any index i, $DT.vertex\_of(s, i)$ returns the i-th vertex of s. There may or may not be a simplex t opposite to the vertex of s with index i. The function $DT.opposite\_simplex(s, i)$ returns t if it exists and returns *nil* otherwise. If t exists then s and t share *dcur* vertices, namely all but the vertex with index i of s and the vertex with index $DT.index\_of\_vertex\_in\_opposite\_simplex(s, i)$ of t. Assume that $t = DT.opposite\_simplex(s, i)$ exists and let $j = DT.index\_of\_vertex\_in\_opposite\_simplex(s, i)$. Then $s = DT.opposite\_simplex(t, j)$ and $i = DT.index\_of\_vertex\_in\_opposite\_simplex(t, j)$. In general, a vertex belongs to many simplices.

Any simplex of DT belongs either to the nearest or to the furthest site Delaunay triangulation or both. The test $DT.simplex\_of\_nearest(dt\_simplex\ s)$ returns true if s belongs to the nearest site triangulation and the test $DT.simplex\_of\_furthest(dt\_simplex\ s)$ returns true if s belongs to the furthest site triangulation.

Further Implementation Issues

The implementation of type *chull* follows [CMS93] and Delaunay triangulations are reduced to convex

Figure 4: The Manual Page of class chull, an excerpt

1. Definition
An instance C of type *chull* is ...

2. Creation
chull C(*int* $d = 2$); creates an instance C of type *chull*. The dimension of the underlying space is d and S is initialized to the empty point set.

The data type *chull* offers neither copy constructor nor assignment operator.

3. Operations

int	C.dim( )	returns the dimension of ambient space
dd_rat_point	C.associated_point(*ch_vertex* v)	returns the point associated with vertex v.
ch_vertex	C.vertex_of_simplex(*ch_simplex* s, *int* i)	returns the vertex corresponding to the i-th vertex of s. *Precondition*: $0 \leq i \leq dcur$.
dd_rat_hyperplane	C.hyperplane_supporting(*ch_facet* f)	returns a hyperplane supporting facet f. The hyperplane is oriented such that the interior of C is on the negative side of it. *Precondition*: f is a facet of C and $dcur > 1$.
list<ch_facet>	C.all_facets( )	returns a list of all facets of C.

⋮

4. Implementation
The time and space requirement is input dependent. Let C_1, C_2, C_3, ...be the sequence of hulls constructed and for a point x let k_i be the number of facets of C_i that are visible from x and that are not already facets of C_{i-1}. Then the time for inserting x is $O(dim \sum_i k_i)$ and the number of new simplices constructed during the insertion of x is the number of facets of the hull which were not already facets of the hull before the insertion.
The data type *chull* is derived from *regLcomplex*. The space requirement of regular complexes is essentially $12(dim + 2)$ Bytes times the number of simplices plus the space for the points. *chull* needs an additional $8 + (4 + x)dim$ Bytes per simplex where x is the space requirement of the underlying number type and an additional 12 Bytes per point. The total is therefore $(16 + x)dim + 32$ Bytes times the number of simplices plus $28 + x \cdot dim$ Bytes times the number of points.

hulls through the well-known lifting map, see for example [Ede87]. Based on our kernel a class *regLcomplex* was implemented that can represent so-called regular simplicial complexes. A simplicial complex is called regular if all maximal simplices, i.e., simplices that are not a subsimplex of another simplex of the complex, have the same dimension. The class *regLcomplex* provides operations for navigation through the complex and update operations. The class *chull* is derived from *regLcomplex* and the class *dd_delaunay* is derived from *chull*.

The work horse for the query operations on convex hulls and Delaunay triangulations is a method

```
C.visibility_search(dd_rat_point x,
  list<ch_facet>& visible_facets,
  int& location, ch_facet& f);
```

that constructs the list of all x-visible hull facets in *visible_facets*, returns the position of x with respect to the current hull in *location* (-1 for inside, 0 for on the the boundary, and $+1$ for outside) and, if x is contained in the boundary of C, returns a facet incident to x in f.

The membership query and the visible facets query for hulls are easily realized by this method and the nearest neighbor and the range query for Delaunay triangulations use it in an essential way. The nearest neighbor query for Delaunay triangulations lifts the query point (using the lifting map), then determines all visible facets of the hull, and then selects the best vertex by linear search through their vertices[3]. The range query with spheres lifts the sphere (using the lifting map) and then finds all vertices of the hull that lie below the resulting hyperplane[4].

We use program checking [BLR90, MNS+96] in our implementation. In particular,

- the class *regLcomplex* provides a method *RC.check_topology*( ) that partially checks whether *RC* is an abstract simplicial complex[5], and a method *RC.is_Delaunay*(*kind*) that checks whether *RC* is a nearest (*kind* = *nearest*) or furthest (*kind* = *furthest*) site Delaunay triangulation of its vertex set.
- the class *chull* provides a method *C.check*( ) that verifies convex hulls as described in [MNS+96].
- we have algorithms that check whether a graph is a nearest or furthest site Delaunay diagram of its vertex set, whether a graph is a triangulation of its vertex set, and whether a graph is a nearest or furthest site Voronoi diagram.

The representation of convex hulls and Delaunay triangulations in data types *chull* and *dd_delaunay* is simplex-based, i.e., simplices are the main objects and

[3]This method is only efficient in low-dimensional space

[4]The lifing map turns a sphere into a hyperplane.

[5]The method checks whether the neighborhood relationship on simplices is symmetric, whether all vertices of a simplex are distinct, and whether two neighboring simplices share all but one of their vertices. It does not check whether simplices that share all but one of their vertices are actually neighbors in the complex.

lower dimensional faces are only implicitely represented. In two-dimensional space there is an alternative representation which makes the vertices and edges the primary objects and represents simplices (= triangles) implicitely as faces of a planar graph. This is the representation used for two-dimensional Delaunay triangulations (type *delaunay*) in LEDA. If *DT* has type *dd_delaunay*, *DTG* has type *GRAPH<POINT, int>*[6], and *kind* is one of *nearest* or *furthest*, then

```
DT.graphrep(DTG,kind);
```

constructs the graph representation of *DT* in *DTG*. All LEDA graph algorithms can now be applied to *DTG*. For example,

```
compute_voronoi(DTG,VD,kind);
```

will construct the graph representation of the Voronoi diagram in *VD*, and

```
edge_array<rational> dist(DTG);
forall_edges(e,DTG)
  dist[e] = DTG[source(e)].
            sqr_dist(DTG[target(e)]);
list<edge> L =
  MIN_SPANNING_TREE(DTG,dist);
```

will construct in *L* the set of edges comprising a minimum spannning tree of *DTG*.

6 Documentation

The full documentation of the kernel consists of about 240 pages[7] and the documentation of the application layer comprises about 100 pages[8].

Figure 5 shows the implementation of the *orientation*-predicate of the class *dd_rat_point* and one member operation of class *chull*. For each class the documentation and the implementation are collected in a noweb-file[9] and different tools are used to give different views of the noweb-file: the noweb tool **notangle** extracts the code, i.e., the view needed by the C++ compiler, and the LEDA tools **Lman** and **Ldoc** give the manual view and the documentation view, respectively[10].

Figure 5 also illustrates the vertical interaction between the software layers. In case of the *orientation*-predicate the determinant calculation is done by *determinant( )* provided as a friend of *integer_matrix*. Most affine operations on point tuples map to the solution of a corresponding linear system calculated by our linear algebra layer. Notice that precondition checks are formulated as preprocessor macros which can be switched off by the flag **-DLEDA_CHECKING_OFF**.

7 Experimental Experiences

Both programs rely heavily on the fact that the kernel is exact. For example, the insertion routine for convex hulls distinguishes cases according to whether the newly inserted point lies in the affine hull of the points already present or not, and the checking programs would hardly make sense without exact primitives.

In the early stages of program development the checking feature of the kernel was particulary useful. For example, the convex hull program needs to compute the hyperplane defined by a set of points. This can be done by solving a linear system. In the first version of the program we set up the wrong linear system. It was very useful that the linear system solver gives a proof of unsolvability and does not just claim unsolvability. This located the error fairly quickly.

We have used classes *chull* and *dd_delaunay* on problems up to dimension 10. We have also compared it to the *qhull*-program of Barber, Dobkin, and Hudhanpaa[BDH96] and the *hull*-program of Clarkson. The first method computes approximate convex hulls and the latter method computes exact hulls but works only for a limited (albeit large) range of coordinate values. Both methods are significantly faster than ours. This is mostly due to their use of floating point arithmetic. Neither of the algorithms provides the rich functionality that we provide.

8 Availability and Extensions

The whole kernel has been used internally since June 96 and will become part of the next major LEDA release in form of a so called LEDA extension package. The complete specification of the listed data types and the code projects can be obtained via WWW from the LEDA home page **http://www.mpi-sb.mpg.de/LEDA**.

In cooperation with the CGAL-project the code base was extended with number type templatization. This allows runtime comparisons depending on the plugged in arithmetic components. We want to evaluate the use of modular integer data types within the linear algebra packet as well as in different geometric application scenarios. An adapted version of the templatized code will become part of the CGAL-kernel.

9 Conclusions

We described the layered design of a kernel for higher dimensional computational geometry and gave two ex-

[6] In LEDA, *GRAPH<POINT, int>* is the type of graphs where each node has an associated information of type *POINT* and each edge has an associated information of type *int*; the edge information is not used in this code.

[7] see **http://www.mpi-sb.mpg.de/~seel/geo.tar.gz**

[8] see **http://www.mpi-sb.mpg.de/~mehlhorn/Programs.html**

[9] see **http://www.cs.purdue.edu/homes/nr/noweb** for an introduction to noweb

[10] see **http://www.mpi-sb.mpg.de/~mehlhorn/LEDAbook.html** for an introduction to this tools

Figure 5: The implementation style

Orientation

⟨*dd_rat_point.h*⟩≡

```
int orientation(const array<rat_point> & A);
```

Semantics: determines the orientation of the points in A, where A consists of $d+1$ points in d-space. This is the sign of the determinant

$$\begin{vmatrix} 1 & 1 & 1 & 1 \\ A[0] & A[1] & \ldots & A[d] \end{vmatrix}$$

where $A[i]$ denotes the cartesian coordinate vector of the i-th point in A.

We are given an array A of $d+1$ points in d-space and compute their orientation. Multiplying the i-th column of the above matrix by the homogenizing coordinate of $A[i]$ leaves the sign of the determinant unchanged. We set up this matrix and return its determinant. Actually, it is more convenient to transpose it and to make the first row the last. This changes the sign if the number of rows is even, i.e., if d is odd.

⟨*implementing dd_rat_point*⟩+≡

```
int
orientation(const array< dd_rat_point > & A)
{
  TUPLE_DIM_CHECK(A,orientation)

  int al = A.low(); // the lower index start of |A|
  int d = A.high() - al; // A contains d + 1 points

  LEDA_OPT_PRECOND((A[al].dim() == d),"orientation: \
  needs A[].dim() + 1 many input points.")

  integer_matrix M(d+1); // quadratic

  for (int i = 0; i <= d; i++)
  {
    for (int j = 0; j <= d; j++)
      M(i,j) = A[al + i].hcoord(j);
  }

  int row_correction = ( (d % 2 == 0) ? +1 : -1 );
  // we invert the sign if the row number is even i.e. d is odd
  return row_correction * sign_of_determinant(M);
}
```

Facets visible from a point

⟨*public members of class chull*⟩+≡

```
list<ch_facet> facets_visible_from(const dd_rat_point& x);
```

Semantics: returns the list of all facets that are visible from x.
Precondition: x is contained in the affine hull of S.

⟨*Member Impls*⟩+≡

```
list<ch_facet> chull::facets_visible_from(const rat_point& x)
/* returns the list of all facets that are visible from |x|. */
{ list<ch_simplex> visible_simplices;
  int location = -1;                    // intialization is important
  int number_of_visited_simplices = 0;  // irrelevant
  ch_facet f;                           // irrelevant

  visibility_search(origin_simplex, x, visible_simplices,
                    number_of_visited_simplices, location, f);
  return visible_simplices;
}
```

amples for applications. The primary goal was to develop a useful base for geometric application design including checkable correctness, efficiency based on existing software library concepts and a clear and comprehensive documentation scheme.

References

[BDH96] C. Barber, D. Dobkin, and H. Hudhanpaa. The quickhull program for convex hulls. *ACM Transactions on Mathematical Software*, 22:469–483, 1996.

[BLR90] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. In *Proc. 22nd Annual ACM Symp. on Theory of Computing*, pages 73–83, 1990.

[CMS93] Kenneth L. Clarkson, Kurt Mehlhorn, and Raimund Seidel. Four results on randomized incremental constructions. In *Computational geometry : theory and applications*, volume 3, pages 185–212, Amsterdam, 1993. Elsevier.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer, 1987.

[Edm67] J. Edmonds. Systems of distinct representatives and linear algebra. *Journal of Research of the National Bureau of Standards*, 71(B):241–245, 1967.

[FGK+96] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. The CGAL kernel : a basis for geometric computation. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry : Towards Geometric Engineering : Workshop (FCRC-96; WACG-96), Philadelphia, PA, USA, May 27-28, 1996; selected paper*, volume LNCS 1148, pages 191–202S., Berlin, 1996. Springer.

[MN95] K. Mehlhorn and S. Näher. LEDA, a platform for combinatorial and geometric computing. *Communications of the ACM*, 38:96–102, 1995.

[MNS+96] K. Mehlhorn, S. Näher, T. Schilz, S. Schirra, M. Seel, R. Seidel, and Ch. Uhrig. Checking Geometric Programs or Verification of Geometric Structures. In *Proc. of the 12th Annual Symposium on Computational Geometry*, pages 159–165, 1996.

[MNU95] K. Mehlhorn, S. Näher, and C. Uhrig. *The LEDA User Manual*, 1995.

[MZ94] M. Müller and J. Ziegler. An implementation of a convex hull algorithm. Technical Report MPI-I-94-105, Max-Planck-Institut für Informatik, Saarbrücken, 1994.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, Chichester, New York, Brisbane, Toronto, Singapore, 1986.

[Yap93] C. K. Yap. *Fundamental Problems in Algorithmic Algebra*. Princeton University Press, 1993.