Contents

8
The Geometry Kernels

A geometry kernel offers basic geometric objects, such as points, lines, segments, rays, planes, circles, ..., and geometric primitives operating on these objects, e.g, the computation of the area of the triangle defined by three points and the computation of the intersection of two lines.

LEDA offers geometric kernels for plane geometry, for three-dimensional geometry, and for geometry in higher dimensional space. We discuss the kernels for two- and three-dimensional geometry in the first eight sections. The kernel for higher dimensional geometry will be discussed in Section 8.9.

The two- and three-dimensional kernels come in two kinds: the rational kernel and the floating point kernel. Write one of

```
#include <LEDA/rat_kernel.h>
#include <LEDA/float_kernel.h>

#include <LEDA/d3_rat_kernel.h>
#include <LEDA/d3_float_kernel.h>
```

to select a kernel. The kernels for two-dimensional geometry provide points, lines, segments, rays, vectors, circles, polygons, generalized polygons, and affine transformations. We use the type names *point*, *line*, *segment*, *ray*, *vector*, *circle*, *polygon*, *gen_polygon*, and *transform* for the corresponding classes of the floating point kernel and the names *rat_point*, *rat_line*, *rat_segment*, *rat_ray*, *rat_vector*, *rat_circle*, *rat_polygon*, *rat_gen_polygon*, and *rat_transform* for the corresponding classes of the rational kernel. If the distinction between rational and floating point kernel is immaterial, we use capital letters: POINT, LINE, SEGMENT, ... The three-dimensional kernels provide lines and planes.

The header files above simply collect the header files of all relevant classes into one. For example,

⟨*rat_kernel.h*⟩≡

```
#include <LEDA/rational.h>
#include <LEDA/rat_point.h>
#include <LEDA/random_rat_point.h>
#include <LEDA/rat_segment.h>
#include <LEDA/rat_ray.h>
#include <LEDA/rat_line.h>
#include <LEDA/rat_circle.h>
#include <LEDA/rat_vector.h>
#include <LEDA/rat_polygon.h>
#include <LEDA/rat_gen_polygon.h>
```

It is important to understand the difference between the rational and the floating point kernel.

In the rational kernel the Cartesian coordinates of points are rational numbers (in the sense of mathematics) and the geometric primitives are exact, i.e., always give the correct result.

In the floating point kernel the Cartesian coordinates of points are double precision floating point numbers and the geometric primitives are approximate, i.e, they usually give the correct result but there is no guarantee. The use of the floating point kernel is therefore not without risk.

Why do we have the floating point kernel at all? There are several reasons: (1) the outside world, e.g., the graphics systems used to visualize the results of geometric computations, wants floating point numbers, (2) we started with the floating point kernel, and (3) floating point computation is faster than computation with rational numbers. The last sentence requires further explanation. First, floating point computation is unreliable and hence the cost of efficiency is a reliability problem. The dangers of floating point arithmetic in geometric computations are discussed in Section 8.6. Second, the overhead of exact computation is surprisingly small due to our extensive use of so-called floating point filters. Our experiments show that the cost of exact arithmetic is never more than a factor of three in running time and usually much smaller. The efficient realization of exact geometric computation and floating point filters are discussed in Section 8.7.

In our own work we do program development exclusively with the rational kernel. Only when a program is stable, we might consider switching to the floating point kernel. We switch only if the use of the rational kernel does not give the desired performance. A switch to the floating point kernel should always be accompanied by a careful analysis of its limits, see Section 8.8.

This chapter is organized as follows: the first two sections deal with geometric objects and geometric predicates, respectively. Every user of LEDA geometry should read them. The next three sections treat special topics: affine transformations, generators for geometric objects, and writing kernel independent code. They may be skipped on first reading. We then have three sections on arithmetic. We first discuss the danger of using floating point arithmetic as an implementation of mathematics' real numbers, then describe the efficient implementation of exact geometric predicates in the rational kernel, and finally comment on the safe use of the floating point kernel. The last three sections give a glimpse at the higher-dimensional kernel, briefly review the history of geometry in LEDA, and discuss the relation between LEDA and CGAL.

8.1 Basics

We discuss points, segments, lines, rays, vectors, and circles.

Cartesian and Homogeneous Coordinates: We assume that the ambient space is equipped with the standard Cartesian coordinate system and specify points by their Cartesian coordinates. For a point p in the plane the functions

```
p.xcoord();
p.ycoord();
```

return the x- and y-coordinate of p, respectively. Of course, the z-coordinate of a point in space is returned by *p.zcoord*( ). The Cartesian coordinates of a *point* are of type *double* and the Cartesian coordinates of a *rat_point* are of type *rational*. We use RAT_TYPE as the generic name, i.e., RAT_TYPE stands for *double* when the floating point kernel is used and stands for *rational* when the rational kernel is used.

Points are stored by their Cartesian coordinates. For *rat_points* it is more efficient to store them by their homogeneous coordinates, i.e., to use the same denominator for the x- and the y-coordinate. The homogeneous coordinates of a point in the plane are a triple (x, y, w) with $w \neq 0$; here w is called the homogenizing coordinate. The Cartesian coordinates of a point with homogeneous coordinates (x, y, w) are $(x/w, y/w)$. Observe that the homogeneous coordinates of a point are not unique. Two triples that are multiples of each other specify the same point. The homogeneous coordinates of a point p in the plane are returned by

```
p.X();
p.Y();
p.W();
```

respectively. The homogeneous coordinates of a *rat_point* are of type *integer*. Do *points* also have homogeneous coordinates? Yes, for compatibility with *rat_points* they do. The homogenizing coordinate of a *point* is the constant 1.0 and the X- and Y-coordinate is simply the corresponding Cartesian coordinate. Thus the homogeneous coordinates of a *point* are of type *double*. We use INT_TYPE to denote the type of the homogeneous coordinates[1], i.e., INT_TYPE stands for *integer* when the rational kernel is used, and stands for *double* when the floating point kernel is used.

We said above that homogeneous coordinates are not unique. We guarantee, however, that all accesses to the homogeneous coordinates of a point return the same value. We do

[1] We chose RAT_TYPE and INT_TYPE as the names for the types of the Cartesian and the homogeneous coordinates because we prefer the rational kernel.

not guarantee, however, that these values are the homogeneous coordinates specified in the constructor for the point. The constructor may simplify the representation by cancelling out common factors. Moreover, we always store a positive value for the homogenizing coordinate.

In mathematical context we also use x_p and y_p for the Cartesian coordinates of a point p and X_p, Y_p, and W_p for the homogeneous coordinates.

Construction: Points are constructed by either specifying their Cartesian or their homogeneous coordinates. Thus

```
point     p(0.2,0.8);
point     q(1,4,5);
rat_point r(1,4,5);
rat_point s(rational(1,5),rational(4,5));
```

are four different ways of defining a point with coordinates $(1/5, 4/5)$. In the first constructor we have defined a *point* by specifying its Cartesian coordinates, in the second constructor we have specified a *point* by giving a triple of doubles[2], in the third constructor we have specified a *rat_point* by a triple of *integers*, and in the fourth constructor we have specified a *rat_point* by a pair of rational numbers.

The generic form of the constructor is

```
POINT p(RAT_TYPE x, RAT_TYPE y)
```

for the construction from Cartesian coordinates, and

```
POINT p(INT_TYPE X, INT_TYPE Y, INT_TYPE W = 1)
```

for the construction from homogeneous coordinates. The default constructor

```
POINT p;
```

constructs the origin. It is bad programming style to exploit this fact. We recommend writing

```
POINT p(0,0);
```

to construct the origin.

We turn to segments, lines, and rays. A segment is constructed by specifying its two endpoints. Thus

```
segment     s(point p, point q);
rat_segment s(rat_point p, rat_point q);
```

define a *segment* and a *rat_segment*, respectively. The second point may also be specified by a vector which defines the relative position of the second point with respect to the first point. The generic forms are

```
SEGMENT s(POINT p, POINT q);
SEGMENT s(POINT p, VECTOR v);    // same as s(p,p+v)
```

[2] The Cartesian coordinates are obtained by performing the floating point divisions $1/5$ and $4/5$.

The defining points of a segment can be accessed by

```
s.source();
s.target();
```

Lines and rays are also defined by two points or by a point and a vector.

```
LINE l(POINT p, POINT q);
LINE l(POINT p, VECTOR v);  // same as l(p,p+v)

RAY r(POINT p, POINT q);
RAY r(POINT p, VECTOR v);  // same as r(p,p+v)
```

Of course, the two defining points must not be equal and the vector must not be the zero-vector.

The default constructors

```
SEGMENT s;
LINE    l;
RAY     r;
```

introduce variables of the appropriate type. They are initialized to some object of the type (the manual even specifies which), but it is bad programming style to rely on this fact.

Vectors can be specified by either their Cartesian or their homogeneous coordinates.

```
vector     v(double x, double y);
rat_vector v(rational x, rational x);
rat_vector v(integer X, integer Y, integer W = 1);
```

Observe that the analogy between *vectors* and *rat_vectors* is not complete. There is no way to define a two-dimensional *vector* by a triple of doubles. The reason is that *vectors* and *rat_vectors* exist for arbitrary dimensions and that

```
vector     v(double x, double y, double z);
```

constructs a three-dimensional vector. The default constructor defines the zero vector.

Circles can be constructed in many ways. We describe two:

```
CIRCLE C(POINT a, POINT b, POINT c);
CIRCLE C(POINT a, POINT b);
```

define a circle passing through points a, b, and c, and a circle with center a and passing through b respectively. If $a = b$ in the second constructor, the circle has radius zero.

Some triples of points are unsuitable for defining a circle. A triple is *admissible* if $|\{p_1, p_2, p_3\}| \neq 2$. Assume now that p_1, p_2, p_3 are admissible. If $|\{p_1, p_2, p_3\}| = 1$, they define the circle with center p_1 and radius zero. If p_1, p_2, and p_3 are collinear, C is a straight line passing through them and the center of C is undefined. If p_1, p_2, and p_3 are not collinear, C is the circle passing through them.

Affine transformations are discussed in Section 8.3 and polygons and generalized polygons are discussed in Section 9.8.

Points and Vectors: Points and vectors are related but clearly distinct geometric objects. In order to work out the relationship between points and vectors it is useful to identify a point with an arrow extending from the origin to the point. In this view a point is an arrow attached to the origin. A vector is an arrow which is allowed to float freely in space[3].

Points and vectors can be combined by arithmetical operations: for two points p and q the difference $p - q$ is a vector[4] and for a point p and a vector v, $p + v$ is a point.

For two vectors v and w their sum $v + w$ and their difference $v - w$ are also vectors. However, it does not make sense to add two points. The unary operator $-$ reverses a vector.

The coordinates of a vector v are accessed by

```
RAT_TYPE v.coord(int i);  // i-th Cartesian coordinate
RAT_TYPE v[int i];        // i-th Cartesian coordinate
INT_TYPE v.hcoord(int i); // i-th homogeneous coordinate
```

For a vector v in d-space the Cartesian coordinates are indexed from 0 to $d - 1$ and the homogeneous coordinates are indexed from 0 to d. The homogenizing coordinate has index d. The homogenizing coordinate of a *vector* is the constant 1. In two-dimensional space the Cartesian and homogeneous coordinates can also be accessed by *xcoord*( ), *ycoord*( ), X( ), Y( ), and W( ), respectively.

Vectors may be stretched or shrunk. If v is a vector and r has INT_TYPE or RAT_TYPE then

```
r * v;
v / r;
```

compute the vectors whose Cartesian coordinates are multiplied by r and divided by r, respectively.

If v and w are vectors then

```
v * w
```

returns the scalar product of v and w. This is the component-wise product of the Cartesian coordinates and has RAT_TYPE.

The scalar product of a vector with itself yields the squared length of the vector. Instead of writing $v * v$ one can also write

```
v.sqr_length();
```

Handle Types, Identity and Equality: All geometric types are so-called handle types or independent item types, see Sections 2.2 and 2.2.2, i.e., an object of any geometric type is a (smart) pointer to a representation object. For example, a *rat_point* is a pointer to a *rat_point_rep* and a *segment* is a pointer to a *segment_rep*. The objects of the representation class contain the defining information about the geometric object and possibly additional information for internal use.

[3] 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.

[4] More precisely, it is the equivalence class of arrows containing the arrow extending from p to q.

We give more details for *rat_points*. The classes *rat_point* and *rat_point_rep* are derived from *handle_base* and *handle_rep*, respectively. The class *handle_base* contains a data member *PTR*, which is a pointer to a *handle_rep*. In *rat_point* we have a private member function *ptr* which casts this pointer to a pointer to a *rat_point_rep*. The class *handle_rep* is discussed in Section 13.7. A *rat_point_rep* contains the homogeneous coordinates of a point (three *integers*), floating point approximations of the homogeneous coordinates (three *doubles*) and the id-number of the point. The floating point approximations of the homogeneous coordinates are used in the floating point filter and will be discussed in Section 8.7. The id-number is used as the hash key in maps and hashing arrays. Any two *point_reps* have distinct id-numbers.

```
class rat_point_rep  : public handle_rep {
  integer x,  y,  w;
  double  xd, yd, wd;
  unsigned long id;
};
class rat_point  : public handle_base {
  rat_point_rep* ptr() const { return (rat_point_rep*)PTR; }
};
```

We distinguish between identical and equal objects. Two points p and q are *identical* (function *identical*(p, q)) if they point to the same *point_rep*, and two points p and q are *equal* (binary operator $==$) if they agree as geometric objects, i.e., have the same Cartesian coordinates.

The assignment statement and the copy constructor preserve identity, i.e., are realized by pointer assignment.

```
POINT p(0,0);
POINT q(0,0);
POINT r = p;
identical(p,q); // evaluates to false
p == q;         // evaluates to true
identical(p,r); // evaluates to true
p == r;         // evaluates to true
```

Linear Orders: There are several linear orders defined on points.

- *cmp_x* compares points by their x-coordinate.
- *cmp_y* compares points by their y-coordinate.
- *cmp_xy* compares points by their x-coordinates. Points with equal x-coordinate are compared by their y-coordinate.
- *cmp_yx* compares points by their y-coordinates. Points with equal y-coordinate are compared by their x-coordinate.
- *cmp* is the same as *cmp_xy*. It is the default order for points.

Associating Information with Geometric Objects: Points, lines, segments, rays, and circles have id-numbers and hence *maps* and *h_arrays* can be defined for them. Observe that *maps* and *h_arrays* associate information with representation objects, i.e, only identical objects share their information. For example,

```
map<POINT,int> color;
POINT p(0,0); color[p] = 0;
POINT q(0,0); color[q] = 1;
POINT r = p;
cout << color[p] << color[q] << color[r];  // outputs 010
```

For points we can also use dictionaries and dictionary arrays to associate information (for the other geometric types this requires the definition of a compare function). In dictionaries and dictionary arrays equal objects share their information. For example,

```
d_array<POINT,int> color;
POINT p(0,0); color[p] = 0;
POINT q(0,0); color[q] = 1;
POINT r = p;
cout << color[p] << color[q] << color[r];  // outputs 111
```

Observe that p and q are equal and hence the assignment to $color[q]$ also changes the color of p.

Dictionary arrays are useful for removing multiple occurrences of equal objects. For example, if L is a list of points, then

```
d_array<POINT,bool> first_occurrence(true);
list_item it;
forall_items(it,L)
{ if ( !first_occurrence[ L[it] ] )
    L.del_item(it);
  else
    first_occurrence[ L[it] ] = false;
}
```

removes all but the first occurrence of every point from L. What will this program do when a *map* is used instead of a *d_array*?

Converting between the Rational and the Floating Point Kernel: Floating point objects can be converted to rational objects and rational objects can be converted to floating point objects. We illustrate conversion for points.

If p is a *point* or *rat_point* then

```
point p.to_point();
```

returns a *point*. If p is a point the call is equivalent to the call of the copy constructor, and if p is a *rat_point*, the Cartesian coordinates of the point returned are floating point approximations of the Cartesian coordinates of p.

The conversion from rational objects to floating point objects needs to be used whenever an object is to be displayed in a window. For example, if W is a *window* and p is a POINT, then

```
W << p.to_point();
```

draws p in W. The output statement above could be written even more elegantly as $W \ll p$ if the class *rat_point* provided a conversion operator to *point*. We opted for the less elegant code since the use of conversion operators can lead to unexpected side effects.

Both point classes have a constructor

```
POINT(const point& p, int prec = 0);
```

If POINT is *rat_point* and *prec* is positive the constructor is equivalent to

```
rat_point(integer(p.xcoord() * P), integer(p.ycoord() * P), P),
```

where $P = 2^{prec}$, i.e., the Cartesian coordinates of p are approximated as rational numbers with denominator P. If *prec* is non-positive, the value of *prec* is chosen such that there is no loss of precision in the conversion.

When POINT is *point* and *prec* is positive, the point constructed has Cartesian coordinates $(\lfloor P * x \rfloor / P, \lfloor P * x \rfloor / P)$, where $p = (x, y)$ and $P = 2^{prec}$. If *prec* is non-positive, the new point has coordinates x and y.

Immutability: All geometric objects are *immutable*. There are no operations that change a geometric object, there are only operations to generate new geometric objects from already existing ones. For example, the operation

```
p.translate(1,1);
```

returns a point which is obtained from p by translating it by the vector $(1, 1)$; it does not change the coordinates of the point p. Of course, the translated point may be assigned to p:

```
p = p.translate(1,1);
```

Input and Output: Geometric objects can be written on files and read from files. For example, if p is a POINT then

```
cout << p;
cin  >> p;
```

writes p on standard output, and reads p from standard input, respectively. The input operators $\gg$ are designed such that output written by $\ll$ can be read by $\gg$.

Graphical input and output is very important for geometric objects. The *window* class knows how to draw geometric objects and supports the construction of geometric objects by mouse input. The simplest way to draw a geometric object is to use the operator $\ll$, for example,

```
W << p.to_point();   // W << p can be used if p is a point
W << s.to_segment(); // W << s can be used if s is a segment
W << r.to_ray();     // W << r can be used if r is a ray
W << l.to_line();    // W << l can be used if l is a line
W << C.to_circle();  // W << C can be used if C is a circle
W << P.to_polygon(); // W << P can be used if P is a polygon
```

If more control is needed, e.g, concerning the color or whether a circle should be drawn as a disk, the *draw* functions need to be used. For example,

```
W.draw_segment(s,red);          // draws s in red
W.draw_disk(C,blue);            // draws a blue filled circle
W.draw_filled_polygon(P,green); // draws a filled green polygon
```

Observe that s, C, and P must be floating point objects. Rational objects must be converted to floating point objects first. For example,

```
W.draw_filled_polygon(P.to_polygon(),green);
```

has to be used to draw a filled *rat_polygon*. Observe that the call will also work for *polygons*.

Why did we not overload the *draw*-functions such that they also work for rational objects? The reason is that this would have required to include the header files of the rational kernel into the header file of the window class. The header file of *window* is very large already and we wanted to avoid a further increase in size.

We come to mouse input. The operator $\gg$ can be used to read a point, segment, line, ray, circle, or polygon. For example,

```
W >> p;  // p is a point
W >> s;  // s is a segment
```

read a point and a segment, respectively. The reading operations are blocking and wait for mouse clicks. A point is constructed by a single click of the left mouse button, and a segment, line, ray, and circle is constructed by two clicks of the left mouse button.

What happens when a mouse button different from the left mouse button is clicked? Windows have an internal state in the same way as C++ input streams do. The state indicates whether there is more input to read or not. The state is initially true and is set to false by a click of the right mouse button (this is similar to ending stream input by the "eof" character). If an input statement is used in the test of a conditional, an object of type *window* is automatically converted to a boolean whose value is the internal state. For example,

```
list<point> L;
point p;
while ( W >> p ) L.append(p);
```

reads a sequence of points from W. Every click of the left mouse button inputs a point and a click of the right mouse button terminates the sequence. The three lines above are essentially the implementation of the input operator for polygons.

In window.h the input operator $\gg$ is only defined for the floating point objects. If you want to use them for rational objects you must include the header file rat_window.h. For example,

```
#include <LEDA/rat_window.h>

rat_point p;
while (W >> p)  W << p.to_point();
```

reads a sequence of *rat_points* and echos them in W.

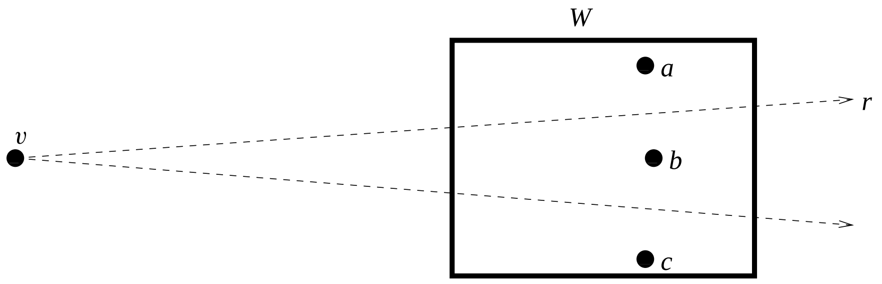

Figure 8.1 The Voronoi vertex v is the center of the circle passing through the points a, b, and c. The three points lie in the window W (indicated as a solid frame) but v lies far outside W. It is a bad strategy to draw the ray r as a ray starting in v and having direction orthogonal to the direction from a to b. A slight error in the computation of the coordinates of v due to round-off may change the appearance of r in W dramatically.

Input and Output: A Warning: As already mentioned, the *window* class offers functions to draw lines, rays, and segments, and many other geometric objects. For example,

```
W.draw_segment(point p, point q);
W.draw_ray(point p, point q);
```

will draw the segment with endpoints p and q and the ray with start point p passing through q, respectively. These functions have the desired effect if the points p and q lie in a rectangle whose side lengths are about 1000 times the side lengths of W. If one of the points lies further away from W, the use of these functions is ill-advised.

Consider the following situation. We are given three points a, b, and c in a window W and want to display their Voronoi diagram. Voronoi diagrams are discussed in Section 9.5. Except when the points lie on a common line, the Voronoi diagram will consist of a single vertex v from which three rays emanate. The Voronoi vertex is the center of the circle passing through the three points. When the three points lie almost on a line, v will lie far outside W, see Figure 8.1. Each ray is part of the perpendicular bisector of two sites. It is natural to draw the ray which is part of the perpendicular bisector of a and b by the following piece of code:

```
POINT v = CIRCLE(a,b,c).center();
VECTOR vec = b - a;
POINT ray_point = v + vec.rotate90();
W.draw_ray(v.to_point(),ray_point.to_point());
```

The drawing produced by this program will be a disappointment, if a, b, and c lie sufficiently close to a common line, since the conversion of v and *ray_point* to points of the floating point kernel (note that this conversion cannot be avoided since the windows class knows only floating point objects) will incur rounding error. Moving either v or *ray_point* slightly has a dramatic effect on the appearance of r in W.

We recommend using a different strategy to draw rays and segments whose defining points may lie far outside W. In this situation the underlying line l is frequently known by other means. In our example, l is the perpendicular bisector of the points a and b.

```
LINE l = p_bisector(a,b);
```

The defining elements of l lie in W and are hence known with high precision. The window class offers functions

```
W.draw_segment(point p, point q, line l, color c);
W.draw_ray(point p, point q, line l, color);
```

that draw the part of the line l between p and q, respectively, the part of l on the ray with source p and second point q. Of course, p and q must lie on l or at least close to it. We give the implementation of the second function.

If p is contained in W we simply draw the ray with source p and second point q. If p lies outside the window we clip the line l on W and call the resulting segment s. The segment s has the property that its source preceeds its target in the lexicographic order of points; equality is possible. We draw s either if p is smaller than the source of s and q is larger than p, or if p is larger than the target of s and q is smaller than p, or if p lies lexicographically between the source and the target of s. The latter case cannot happen mathematically, but it can happen numerically, if p lies close to either the source or the target of s but not exactly on l.

```
void window::draw_ray(point p, point q, line l, color col)
{
  if ( contains(p) ) { draw_ray(p,q,col); return; }
  segment s;
  point llc(xmin(),ymin()); // left lower corner
  point rrc(xmax(),ymax()); // right upper corner
  if ( !l.clip(llc,rrc,s) ) return;
  if ( compare(p,s.source()) < 0 && compare(p,q) < 0 ||
       compare(s.target(),p) < 0 && compare(q,p) < 0 ||
       compare(s.source(),p) <= 0 && compare(p,s.target()) <= 0 )
    draw_segment(s,col);
}
```

We will see an application of the refined drawing functions in Section 9.10.

Exercises for 8.1

1 Write a program that allows to input points in a graphics window and colors the points randomly red and blue.

2 Write a program that allows to input points in a graphics window and always highlights a pair of points with smallest distance. For two points p and q, $p.sqr\_dist(q)$ computes the squared distance between p and q.

3 Write a program that removes duplicates from a list of segments.

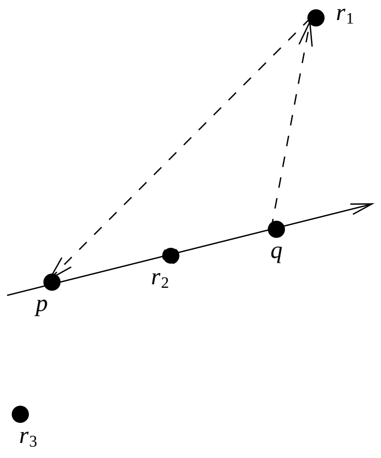

Figure 8.2 *orientation*$(p, q, r_1) = 1$, *orientation*$(p, q, r_2) = 0$, and *orientation*$(p, q, r_3) = -1$. The triangle $\triangle(p, q, r_1)$ is shown dashed.

8.2 Geometric Primitives

We discuss some of the geometric primitives available in LEDA, in particular, the orientation function and its variants, lengths and distances, angles, and intersections.

8.2.1 *The Orientation Function in the Plane*

The *orientation function* is probably the most useful geometric primitive. Let p, q, and r be three points in the plane. The tuple (p, q, r) is said to have *positive orientation* if p and q are distinct and r lies to the left of the oriented line passing through p and q and oriented from p to q, the tuple is said to have *negative orientation* if r lies to the right of the line, and the tuple is said to have *orientation zero* if the three points are collinear, see Figure 8.2. An alternative way to define positive orientation is to say that p, q, and r form a counter-clockwise oriented triangle. The function

```
int orientation(POINT p, POINT q, POINT r)
```

computes the orientation of the triple (p, q, r). It returns $+1$ in the case of positive orientation, -1 in the case of negative orientation, and 0 in the case of zero orientation. There are also predicates that test for special cases.

```
bool leftturn(p,q,r);  // same as orientation(p,q,r) >  0
bool rightturn(p,q,r); // same as orientation(p,q,r) <  0
bool collinear(p,q,r); // same as orientation(p,q,r) == 0
```

We next derive a determinant formula for the orientation function. For points p, q, and r we use $\triangle(p, q, r)$ to denote the triangle with vertices p, q, and r. We define the *signed area* of the triangle $\triangle(p, q, r)$ as its area times the orientation of the triple (p, q, r).

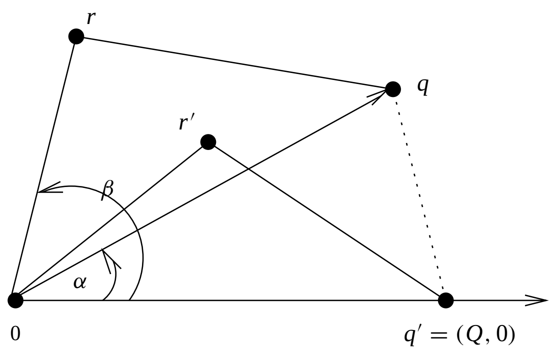

Figure 8.3 Proof of Lemma 1.

Lemma 1 *Let p, q, and r be points in the plane.*
(a) The signed area of the triangle $\triangle(p, q, r)$ is given by

$$\frac{1}{2}\begin{vmatrix} 1 & 1 & 1 \\ x_p & x_q & x_r \\ y_p & y_q & y_r \end{vmatrix}$$

(b) The orientation of (p, q, r) is equal to the sign of the determinant above.

Proof Part(b) follows immediately from part (a) and the definition of signed area. So we only need to show part (a). We do so in two steps. We first verify the formula for the case that p is the origin and then extend it to arbitrary p. So let us assume that p is equal to the origin. We need to show that the signed area A of $\triangle(p, q, r)$ is equal to $(x_q y_r - x_r y_q)/2$.

Let α be the angle between the positive x-axis and the ray Oq and let Q be the length of the segment Oq, cf. Figure 8.3. Then $\cos\alpha = x_q/Q$ and $\sin\alpha = y_q/Q$. Rotating the triangle $\triangle(O, q, r)$ by $-\alpha$ degrees about the origin yields a triangle $\triangle(O, q', r')$ with $q' = (Q, 0)$ and the same signed area. Thus, $A = Q \cdot y_{r'}/2$.

Next observe that $y_r' = R\sin(\beta - \alpha)$, where R is the length of the segment Or and β is the angle between the positive x-axis and the ray Or. Since $\sin(\beta - \alpha) = \sin\beta\cos\alpha - \cos\beta\sin\alpha$ and $R\cos\beta = x_r$ and $R\sin\beta = y_r$ we conclude that

$$\begin{aligned} A &= Q \cdot y_{r'}/2 = Q \cdot R \cdot \sin(\beta - \alpha)/2 \\ &= (Q\cos\alpha \cdot R\sin\beta - Q\sin\alpha \cdot R\cos\beta)/2 = (x_q y_r - x_r y_q)/2. \end{aligned}$$

This verifies the formula in the case where p is the origin.

Assume next that p is different from the origin. Translating p into the origin yields the triangle $\triangle(O, q', r')$ with $q' = q - p$ and $r' = r - p$[5] . On the other hand subtracting the

[5] Strictly speaking, we would have to write $q' = 0 + (q - p)$ and similarly for r'.

first column from the other two columns of the determinant yields

$$\begin{vmatrix} 1 & 1 & 1 \\ x_p & x_q & x_r \\ y_p & y_q & y_r \end{vmatrix} = \begin{vmatrix} 1 & 0 & 0 \\ x_p & x_q - x_p & x_r - x_p \\ y_q & y_q - y_p & y_r - y_p \end{vmatrix} = \begin{vmatrix} x_{q'} & x_{r'} \\ y_{q'} & y_{r'} \end{vmatrix}$$

which by the above is twice the area of the translated triangle. □

Part (b) of the lemma above is the implementation of the orientation function.

8.2.2 *The Orientation Function in Higher-Dimensional Space*

We define the orientation function for an arbitrary dimensional space and derive a determinant formula for it. Less mathematically inclined readers may skip the proofs of the lemmas to follow.

Let $(p_0, p_1, \ldots, p_d)$ be a $d+1$-tuple of points in d-dimensional space. Their orientation is zero if the points lie in a common hyperplane. If they do not, their orientation is either positive or negative as determined by the following rules:

- Let o be the origin and let e_i for i, $0 \le i < d$, be the endpoint of the i-th coordinate vector of d-dimensional space. The tuple $(o, e_0, \ldots, e_{d-1})$ has positive orientation.
- Two tuples $(p_0, p_1, \ldots, p_d)$ and $(q_0, q_1, \ldots, q_d)$ have the same orientation if the affine map that maps p_i into q_i for i, $0 \le i \le d$, has positive determinant.

Lemma 2 *Let $(p_0, p_1, \ldots, p_d)$ be a $d+1$-tuple of points in d-dimensional space. Then*

$$\mathit{orientation}(p_0, p_1, \ldots, p_d) = \mathit{sign}\det\begin{pmatrix} 1 & \cdots & 1 \\ p_0 & \cdots & p_d \end{pmatrix},$$

where the i-th column of the determinant consists of a 1 followed by the vector of Cartesian coordinates of p_i for all i, $0 \le i \le d$.

Proof Observe first that the points $p_0, \ldots, p_d$ have orientation zero iff they lie in a common hyperplane which is true iff the homogeneous linear system

$$\sum_{0 \le i \le d} \lambda_i = 0$$

$$\sum_{0 \le i \le d} \lambda_i p_{i,l} = 0\,,\ 0 \le l \le d-1$$

in variables $\lambda_0, \lambda_1, \ldots, \lambda_d$ has a non-trivial solution. The determinant above is the determinant of this system. We conclude that $\mathit{orientation}(p_0, \ldots, p_d) = 0$ iff the sign of the determinant above is zero.

Assume next that $\mathit{orientation}(p_0, p_1, \ldots, p_d) \ne 0$. The affine transformation that maps $(o, e_0, \ldots, e_{d-1})$ into $(p_0, p_1, \ldots, p_d)$ is given by $x \mapsto p_0 + P \cdot x$ where P has columns $p_1 - p_0, p_2 - p_0, \ldots, p_d - p_0$. Thus

$$\det P \quad = \quad \det\begin{pmatrix} p_1 - p_0 & p_2 - p_0 & \cdots & p_d - p_0 \end{pmatrix}.$$

Adding an additional first row and first column to this determinant with the first entry in the new row equal to one and all other entries in the new row equal to zero does not change the value of the determinant (develop the determinant according to the new row). Therefore

$$\begin{aligned}\det P &= \det\begin{pmatrix} p_1 - p_0 & p_2 - p_0 & \cdots & p_d - p_0 \end{pmatrix}\\ &= \det\begin{pmatrix} 1 & 0 & \cdots & 0 \\ p_0 & p_1 - p_0 & \cdots & p_d - p_0 \end{pmatrix} = \det\begin{pmatrix} 1 & 1 & \cdots & 1 \\ p_0 & p_1 & \cdots & p_d \end{pmatrix},\end{aligned}$$

where the last equality follows from adding the first column to all other columns. We conclude that $(p_0, p_1, \ldots, p_d)$ has the same orientation as $(o, e_0, \ldots, e_{d-1})$ if and only if the determinant above is positive. □

The lemma above generalizes Lemma 1. Observe that both lemmas give the same formula for points in the plane.

We have already given an intuitive definition of orientation in the plane: three points (p_0, p_1, p_2) in the plane have orientation zero if they are collinear, have positive orientation if they form a counter-clockwise oriented triangle, and have negative orientation if they form a clockwise oriented triangle.

In three-dimensional space there is also an intuitive definition. Four points (p_0, p_1, p_2, p_3) in three-dimensional space have orientation zero if they are coplanar, have positive orientation if they form a right-handed system, and have negative orientation if they form a left-handed system. We need to explain the terms right- and left-handed system. Imagine that you place the base of your thumb at point p_0 and let the thumb (index finger, middle finger) point to p_1, p_2, and p_3, respectively. Only one of your hands will work and this determines the handedness of the system. For four three-dimensional points p, q, r, and s

```
int orientation(p,q,r,s);
```

computes their orientation.

An alternative definition of orientation in three-dimensional space is to say that the four-tuple (p_0, p_1, p_2, p_3) has positive orientation if p_3 sees (p_0, p_1, p_2) in counter-clockwise orientation. The last sentence connects orientation in three-dimensional space with orientation in two-dimensional space. The next lemma generalizes this connection to higher dimensions.

Lemma 3 *Let $(p'_0, p'_1, \ldots, p'_{d-1})$ be a d-tuple of points in $(d-1)$-dimensional space with positive orientation and let $(p_0, p_1, \ldots, p_d)$ be a $d+1$-tuple of points in d-dimensional space such that p_i projects into p'_i for i, $1 \le i < d$, i.e., the Cartesian coordinate vector of p'_i is the Cartesian coordinate vector of p_i with the last entry removed. Let h be the hyperplane spanned by $p_0, \ldots, p_{d-1}$. Then $(p_0, p_1, \ldots, p_d)$ has positive orientation if p_d lies above h, has orientation zero if p_d lies on h, and has negative orientation if p_d lies below h.*

Proof Let q be the projection of p_d into h. Then $p_d = q + c \cdot e_{d-1}$ where e_{d-1} is the

$(d-1)$-th coordinate vector and c is positive if p_d lies above h, is zero if p_d lies on h, and is negative if p_d lies below h. Moreover there are $\lambda_0, \lambda_1, \ldots, \lambda_{d-1}$ such that

$$\sum_{0 \le i \le d-1} \lambda_i = 1,$$

and

$$\sum_{0 \le i \le d-1} \lambda_i p_i = q.$$

Thus

$$\begin{aligned}
\det\begin{pmatrix} 1 & 1 & \cdots & 1 \\ p_0 & p_1 & \cdots & p_d \end{pmatrix} &= \det\begin{pmatrix} 1 & 1 & \cdots & 1 & 1 \\ p_0 & p_1 & \cdots & p_{d-1} & q + c \cdot e_{d-1} \end{pmatrix} \\
&= \det\begin{pmatrix} 1 & 1 & \cdots & 1 & 0 \\ p_0 & p_1 & \cdots & p_{d-1} & c \cdot e_{d-1} \end{pmatrix} \\
&= c \cdot \det\begin{pmatrix} 1 & 1 & \cdots & 1 \\ p'_0 & p'_1 & \cdots & p'_{d-1} \end{pmatrix},
\end{aligned}$$

where the second equality follows from subtracting the λ_i-th multiple of the i-th column from the last column for i, $0 \le i < d$, and the last equality follows by expanding the determinant according to the last column. Observe that the last column has only one non-zero entry and that this entry is in the last row. □

In the plane we connected the orientation of a triple (p, q, r) to the signed area of the triangle defined by the points. A similar connection holds in higher-dimensional space. The signed area of the simplex with vertices $p_0, p_1, \ldots, p_d$ is equal to $\frac{1}{d!}$ times the determinant defined by the points.

8.2.3 *Sidedness*

Many geometric objects, such as lines and circles in the plane, planes and spheres in three-dimensional space, and more generally hyperplanes and hyperspheres in d-dimensional space, partition ambient space into two parts. We designate one of the parts as positive and one as negative. The function

```
int O.side_of(x);
```

where O is a geometric object and x is a point in ambient space returns a positive number (zero, a negative number, respectively) if x lies in the positive part (lies on O, lies in the negative part, respectively). Examples are

```
int l.side_of(x);     // l is a line
int C.side_of(x);     // C is a circle
int P.side_of(x);     // P is a polygon
```

What is the positive subspace with respect to a line or circle or hyperplane? We use the orientation function for points to formulate general rules:

- For a hyperplane h in d-space defined by points $p_0,\ p_1,\ \ldots,\ p_{d-1}$ (in this order) the positive subspace consists of all points p_d such that $(p_0,\ p_1,\ \ldots,\ p_d)$ has positive orientation. Thus *line*(p, q).*side_of*(x) is the same as *orientation*(p, q, x), if p and q are distinct.
- For a hypersphere S in d-space defined by points $p_0,\ p_1,\ \ldots,\ p_d$ (in this order) the positive subspace consists of the interior of the sphere if $(p_0,\ p_1,\ \ldots,\ p_d)$ is positively oriented and consists of the exterior of the sphere otherwise. The same rule applies to simplices.

In two-dimensional space the following alternative rule is also worth remembering. Two points defining a line and three points defining a circle impose a sense of direction on the line or circle respectively (from the first point to the second point in the case of a line, and from the first point through the second point to the third point in the case of a circle). *The positive subspace is the region to the left of the object.*

Let p, q, and r be points in the plane. We may want to inquire about the position of a point x with respect to *circle*(p, q, r). We could write *circle*(p, q, r).*side_of*(x). Since this test incurs overhead for the construction of a circle we also have an alternative syntactic format that avoids this overhead and also gives an answer in the case where the p, q, and r do not define a circle.

```
int side_of_circle(p,q,r,x);
```

returns $+1$ if x is to the left of the oriented circle through p, q, and r, returns -1 if x is to the right of the oriented circle through p, q, and r, and returns 0 if either $|\{p, q, r\}| \le 2$ or x lies on the oriented circle through p, q, and r. We give some more explanations.

Three points p, q, and r that are not collinear define a unique circle passing through them. We give this circle an orientation by insisting that p, q, and r occur in this order on the circle. Consider now a fourth point x. It is either left of, on, or right of the oriented circle through p, q, and r. Note that left corresponds to inside if the circle is counter-clockwise oriented and to outside otherwise, see Figure 8.4. The case that the points p, q, and r are collinear deserves special attention. If the three points are not pairwise distinct then the *which_side* function returns zero. If they are pairwise distinct then we orient the line passing through them such that the order of the points along the line is a circular permutation of (p, q, r), i.e., either (p, q, r) or (q, r, p) or (r, p, q), and use again $+1$ for the left side and -1 for the right side of the line.

Circles, spheres, triangles, simplices, simple polygons, and many other geometric objects partition ambient space into a bounded and an unbounded region. Since there is no standard convention in mathematics that connects boundedness and unboundedness with positive and negative respectively, we have an enumeration type for the outcome of the *region_of* function.

```
enum region_kind { BOUNDED_REGION, ON_REGION, UNBOUNDED_REGION };
region_kind O.region_of(x);            // the generic form
region_kind C.region_of(x);            // C is a circle
```

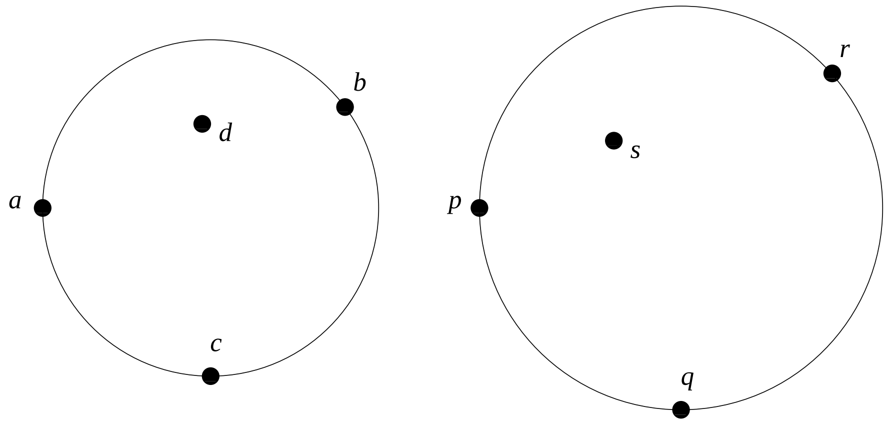

Figure 8.4 The sides of a circle: d lies on the negative side of the circle defined by points a, b, and c, and s lies on the positive side of the circle defined by points p, q, and r.

Frequently, one only wants to test for one of the outcomes. We have appropriate predicates.

```
bool O.inside(x);      // O.region_of(x) == bounded_region
bool O.on_boundary(x); // O.region_of(x) == on_region
bool O.outside(x);     // O.region_of(x) == unbounded_region
```

8.2.4 *Length and Distance*

If p and q are POINTs and l is a LINE,

```
RAT_TYPE p.sqr_dist(q);
RAT_TYPE l.sqr_dist(q);
```

compute the square of the distance between q and p or l, respectively.

In the rational kernel there are no functions to compute distances, in the floating point kernel there are, but think twice before using them. Why?

The distance between two points p and q is equal to $((x_p - x_q)^2 + (y_p - y_q)^2)^{1/2}$ and is hence, in general, not a rational number. The squared distance is a rational number and hence the rational kernel provides only functions to compute squared distances. The floating point kernel uses the *sqrt* function from the standard math-library to compute distances.

We find that the computation of distances is rarely needed. Consider the following problem. Let p and q be points. We want to define the circle centered at p whose radius is ρ times the distance between p and q. This is best written as

```
CIRCLE C(p, p + rho * (q - p));
```

Observe that $q - p$ is the vector from p to q and hence $rho * (q - p)$ is a vector whose length is ρ times the distance between p and q.

The distances between p and q and r, respectively, can be compared by

```
int p.cmp_dist(q,r); // same as cmp(p.sqr_dist(q),p.sqr_dist(r));
```

This is more efficient than computing the two squared distances and comparing them.

8.2.5 ***Angles***

There is no type angle in either the rational or the floating point kernel. There are, however, a number of functions related to angles. In particular, two vectors v_1 and v_2 can be compared by the angle which they form with the positive x-axis. For a vector v let $\alpha(v)$ be the angle by which the positive x-axis has to be turned counter-clockwise until it aligns with v. The zero vector defines the angle zero.

```
int compare_by_angle(VECTOR v1, VECTOR v2);
```

returns $cmp(\alpha(v1), \alpha(v2))$.

We describe the implementation. If one of the vectors is the zero vector the comparison is easily made. If both vectors are zero, they are equal, and if only one is zero, it is the smaller. So assume that both vectors are non-zero. We say that a non-zero vector (x, y) belongs to the upper half-plane if either $y > 0$ or $y = 0$ and $x > 0$, and we say that it belongs to the lower half-plane otherwise. Let *upper1* and *upper2* be the half-planes to which our vectors belong (the value is $+1$ for a vector in the upper half-plane and -1 for a vector in the lower half-plane). If the two vectors belong to distinct half-planes, the vector in the upper half-plane is smaller and hence we may return the sign of *upper2* $-$ *upper1*. If the two vectors lie in the same half-plane, v_1 precedes v_2 iff the triangle $(O, O+v_1, O+v_2)$ is counter-clockwise oriented iff the orientation of $(O, O+v_1, O+v_2)$ is positive iff its signed area is positive. The signed area is the length of the cross-product of v_1 and v_2, i.e., $x_1y_2 - x_2y_1$. We may therefore return $-sign(x_1y_2 - x_2y_1)$.

Rational vectors are stored by their homogeneous coordinates. Since the ordering of angles does not depend on the length of vectors and since the homogenizing coordinate is guaranteed to be non-negative, we may ignore it.

$\langle$*_angle_order.c*$\rangle{+}{\equiv}$

```
int compare_by_angle(const rat_vector& v1, const rat_vector& v2)
{ const integer& x1 = v1.hcoord(0);
  const integer& y1 = v1.hcoord(1);

  const integer& x2 = v2.hcoord(0);
  const integer& y2 = v2.hcoord(1);

  if ( x1 == 0 && y1 == 0 ) return ( x2 == 0 && y2 == 0 ? 0 : -1);
  if ( x2 == 0 && y2 == 0 ) return 1;

  // both vectors are non-zero

  int sy1 = sign(y1); int sy2 = sign(y2);

  int upper1 = ( sy1 != 0 ? sy1 : sign(x1) );
  int upper2 = ( sy2 != 0 ? sy2 : sign(x2) );

  if ( upper1 == upper2 ) return sign(x2*y1 - x1*y2);

  return sign(upper2 - upper1);
}
```

8.2.6 *Intersections*

There are functions to compute the intersections between lines, rays, and segments. For example, if l is a LINE and s is a SEGMENT then

```
bool l.intersection(s, p);
```

returns *true* if l and s have a single point in common and returns *false* otherwise. In the latter case, the unique point of intersection is assigned to p.

Exercises for 8.2

1 Write a function *circum_center* that takes three points p, q, and r and returns the center of the circle passing through p, q, and r. The three points are assumed to be non-collinear.

2 Use the left-turn predicate to write a function that tests whether four points p, q, r, and s in the plane form a convex quadrilateral.

3 Modify the test from the previous exercise such that it decides whether the four points form a counter-clockwise oriented convex quadrangle.

4 Let p, q, r, and s be four points in three space not lying in a plane. Position your left or right hand such that p coincides with the base of your thumb, and q, r, and s coincide with the tips of your thumb, index finger, and middle finger, respectively. Convince yourself that only one of the two hands will work and relate the choice of hand to the orientation of the four points.

8.3 Affine Transformations

An affine transformation T of the plane is specified by a 3×3 matrix T with $T_{2,0} = T_{2,1} = 0$ and $T_{2,2} \neq 0$. It maps the point p with homogeneous coordinate vector (p_x, p_y, p_w) to the point $T \cdot p$. Transformations are called *transform* in the floating point kernel and are called *rat_transform* in the rational kernel. We use TRANSFORM as the generic name.

```
TRANSFORM T;
TRANSFORM T1(M);
```

declares T as the identity transform and declares *T1* as the transform with transformation matrix M. M must be a 3×3 *matrix* in the floating point kernel and a 3×3 *integer_matrix* in the rational kernel. Functional notation is used to apply an affine transformation to a geometric object. For example,

```
p = T(q);    // p and q are points
P = T(Q);    // P and Q are polygons
v = T(w);    // v and w are vectors
C = T(D);    // C and D are circles; T must be rigid
```

The norm of an affine transformation T is defined as

$$|T| = (T_{0,0}T_{1,1} - T_{0,1}T_{1,0})/T_{2,2}^2.$$

A transformation is called *rigid* iff its norm has absolute value one.

```
RAT_TYPE T.norm();
```

returns the norm of T.

If T and *T1* are transformations then

```
T(T1);
```

is the transformation obtained by first applying *T1* and then T.

Translations, rotations, and reflections are special cases of affine transformations.

A matrix of the form

$$\begin{pmatrix} w & 0 & x \\ 0 & w & y \\ 0 & 0 & w \end{pmatrix}$$

realizes a translation by the vector $(x/w, y/w)$ and a matrix of the form

$$\begin{pmatrix} a & -b & 0 \\ b & a & 0 \\ 0 & 0 & w \end{pmatrix}$$

where $a^2 + b^2 = w^2$ realizes a rotation by the angle α about the origin, where $\cos\alpha = a/w$ and $\sin\alpha = b/w$. Rotations are in counter-clockwise direction.

It is inconvenient to specify transformations by their transformation matrix. We have several functions that construct transformations. Observe that these functions are not constructors but functions that return transformations. For example

```
TRANSFORM T = translation(const INT_TYPE& dx, const INT_TYPE& dy,
                          const INT_TYPE& dw);
TRANSFORM T = translation(const RAT_TYPE& dx, const RAT_TYPE& dy);
```

construct translations by the vector $(dx/dw, dy/dw)$ and the vector (dx, dy), respectively.

```
TRANSFORM T = reflection(const POINT& q, const POINT& r);
TRANSFORM T = reflection(const POINT& q);
```

construct the reflection across the straight line passing through q and r and the reflection across the point q, respectively.

```
TRANSFORM T = rotation90(const POINT & q);
TRANSFORM T = rotation(const POINT& q, double alpha, double eps);
```

construct rotations about the point q. In the first case the rotation is by $\pi/4$ and in the second case the rotation is approximately by α. ϵ is a tolerance parameter.

We show the implementations of the last two functions. Rotation by $\pi/4$ is achieved by the rotation matrix

$$\begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

and rotation about an arbitrary point q is achieved by first translating by the vector $O - q$, rotating about the origin, and finally translating back by the vector $q - 0$.

⟨*rotation*⟩≡
```
static TRANSFORM rotation90_origin(const POINT& q)
{
  INT_MATRIX M(3,3);
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      M(i,j) = 0 ;
  M(0,1) = -1;   M(1,0) = +1;
  M(2,2) = 1;
  return TRANSFORM(M);
}
TRANSFORM rotation90(const POINT& q)
{
  TRANSFORM R = rotation90_origin(q);
  TRANSFORM T0 = translation(-q.X(),-q.Y(),q.W());
  TRANSFORM T1 = translation( q.X(), q.Y(),q.W());
  TRANSFORM T = T1(R(T0));
  T.simplify();
  return T;
}
```

Observe that we have given the function *rotation90_origin* an artificial argument of type *POINT* so that we can use the same code for both kernels. In the piece of code above, we declared *rotation90_origin* static, as it is an auxiliary function that should not be visible outside the file _transform.c.

We come to the rotation by an arbitrary angle α. We only show how to construct the transformation matrix for the rotation about the origin. We construct a point p on the unit circle and in direction α (this is a member function of CIRCLE) and then use the coordinates of p as the sine and cosine of α.

⟨*rotation*⟩+≡
```
static TRANSFORM rotation_origin(const POINT& q,
                                 double alpha, double eps)
{ POINT origin(0,0);
  POINT X(1,0);
  CIRCLE C(origin,X); // unit circle centered at origin
  POINT p = C.point_on_circle(alpha,eps);
  INT_MATRIX M(3,3);
  M(0,2) = M(1,2) = M(2,0) = M(2,1) = 0;
  M(0,0) = M(1,1) = p.X() ;
  M(0,1) = -p.Y();   M(1,0) = p.Y();
  M(2,2) = p.W();
  return TRANSFORM(M);
}
```

It remains to explain the function *point_on_circle*. In the floating point kernel we use the sine and cosine function from the math-library to construct p; *eps* plays no role in this construction. In the rational kernel we use the method described in [CDR92] to find integers a, b, and w and an angle α'such that

$$\begin{aligned} a^2 + b^2 &= w^2 \\ \cos\alpha' &= a/w \\ \sin\alpha' &= b/w \\ |\alpha' - \alpha| &\le \epsilon. \end{aligned}$$

General affine transformations are a fairly recent addition to our geometry kernels. In earlier versions we had only functions for special affine transformations. They were member functions of the geometric classes. For example,

```
p.translate(RAT_TYPE dx,RAT_TYPE dy);
```

returns the point $p + v$ where $v = (dx, dy)$.

Transformations are a good tool to generate difficult inputs for geometric algorithms. In Section 9.8.4 we perform the following experiment. We first construct a regular n-gon P, $n = 20000$, with its vertices on the unit circle. We then construct $Q = T(P)$ where T is a rotation by $2\pi/(nm)$ and m is a large integer, e.g., $m = 10^9$. We finally compute the union of P and Q.

Exercises for 8.3

1 Implement the function that composes two transformations.
2 Implement the function that applies a transformation to a point.
3 Implement the function that applies a transformation to a vector. This is different from the solution to the previous exercise.
4 Implement the function that constructs the transformation matrix for reflection at a point.
5 Implement the function that constructs the transformation matrix for reflection at a line.

8.4 Generators for Geometric Objects

There is a frequent need to generate geometric objects, random or otherwise. We describe generators for random points in the plane and generators for polygons. There are also generators for random points in space.

Generators for Random Points: We have generators for random points in squares, in discs, near circles, and on circles. For each generator there is a version that generates a single point and a version that generates a list of points.

```
random_point_in_square(POINT& p, int maxc);
random_points_in_square(int n, int maxc, list<POINT>& L);
```

generate a random point with integer coordinates in the range $[-maxc\,..\,+maxc]$ and a list of n such points, respectively.

```
random_point_in_unit_square(POINT& p, int D = (1<<30) - 1 );
random_points_in_unit_square(int n, int D, list<POINT>& L);
random_points_in_unit_square(int n, list<POINT>& L);
```

generate a point in the unit square, i.e., a point whose coordinates are of the form i/D for a random integer i, $0 \leq i \leq D$, n such points, and n such points with the default value of D, respectively.

For the remaining generators we only give the form that generates a single point.

```
random_point_in_disc(POINT& p, int R);
random_point_in_unit_disc(POINT& p, int D = (1<<30) - 1);
```

generate a random point with integer coordinates in the disc with radius R and a random point with coordinates of the form i/D for integer i in the unit disc, respectively.

```
random_point_near_circle(POINT& p, int R);
random_point_near_unit_circle(POINT& p, int D = (1<<30) - 1);
```

generate a random point with integer coordinates near the circle with radius R and a random point with coordinates of the form i/D for integer i near the unit circle, respectively.

The latter function is implemented as follows. We generate a random double x in the unit interval, set $\phi = 2\pi x$, and construct the point $(\lfloor D\cos\phi\rfloor, \lfloor D\sin\phi\rfloor, D)$.

```
void random_point_near_unit_circle(POINT& p, int D)
{ double a;
  Rand_Source >> a;
  double phi = 2*a*LEDA_PI;

  int x = int(D*cos(phi));
  int y = int(D*sin(phi));
  p = POINT(x,y,D);
}
```

With the rational kernel we can also generate points that lie *exactly* on a circle.

```
random_point_on_circle(POINT& p, int R, int C = 1000000);
random_point_on_unit_circle(POINT& p, int C = 1000000);
```

constructs a point on the circle with radius R and on the unit circle, respectively. This assumes that the rational kernel is used. In both cases the point is chosen at random from a set of at least C candidates. With the floating point kernel the function is equivalent to the *near_circle* and the *near_unit_circle* function with $D = 1.0/C$, respectively.

The implementation of *random_point_on_unit_circle* with the rational kernel is as follows:

```
void random_point_on_unit_circle(rat_point& p, int C)
{ rat_point origin(0,0);
  rat_circle Circ(origin,origin + rat_vector::unit(1));
  double a; Rand_Source >> a;
```

```
    double eps = 1.0/(2*C);
    p = Circ.point_on_circle(2*LEDA_PI*a,eps);
  }
```

where the function *point_on_circle* is as described at the end of Section 8.3.

The last two generators are much slower than all other generators when the rational kernel is used. We have therefore generated files of 50000 random points (with $C = 10^6$). They are available as:

LEDAROOT/data/geo/rat_points_unit_circle_random_50000.ex
LEDAROOT/data/geo/points_unit_circle_random_50000.ex

Generating Polygons: We have two generators for polygons.

```
POLYGON P = reg_n_gon(int n, CIRCLE C, double epsilon);
POLYGON P = n_gon(    int n, CIRCLE C, double epsilon);
```

The first generator generates a nearly regular n-gon. The i-th point is generated by the call *C.point_on_circle*$(2\pi i/n, \mathit{epsilon})$. With the rational kernel the vertices of the n-gon are guaranteed to lie on the circle, with the floating point kernel they are only guaranteed to lie near C.

The second generator generates a (nearly) regular n-gon whose vertices lie near the circle C. For the floating point kernel the function is equivalent to the function above. For the rational kernel the function first generates an n-gon with floating point arithmetic and then converts the resulting *polygon* to a *rat_polygon*.

8.5 Writing Kernel Independent Code

We use the C++ precompilation mechanism to write code that is independent of the kernel. Recall that the kernels are designed such that all functions that are available in a rational kernel are also available in the corresponding floating point kernel.

The only difference between the rational kernel and the floating point kernel is the interpretation of the generic names POINT, SEGMENT, LINE, ... In order to give the generic names the interpretation required in a particular kernel one of the files must be included:

```
#include <LEDA/rat_kernel_names.h>
#include <LEDA/float_kernel_names.h>
#include <LEDA/d3_rat_kernel_names.h>
#include <LEDA/d3_kernel_names.h>
```

Every one of these files consists of a sequence of define-statements which define the generic names for the corresponding kernel. For example,

```
// part of rat_kernel_names.h

#define KERNEL        RAT_KERNEL
#define INT_TYPE      integer
#define RAT_TYPE      rational
```

```
#define VECTOR        rat_vector
#define POINT         rat_point
#define SEGMENT       rat_segment
#define TRANSFORM     rat_transform
```

We also have files that undefine all names used in a kernel. They are:

```
#include <LEDA/kernel_names_undef.h>
#include <LEDA/d3_kernel_names_undef.h>
```

Suppose now that we want to write a program that is supposed to work for both two-dimensional kernels. We write a generic version of the program using only the generic names and then derive the two specialized versions from it. For example,

⟨*FOO.c*⟩≡

```
main(){

window W; W.display();

POINT p;
while ( W >> p) W << p.to_point();
}
```

⟨*rat_foo_test.c*⟩≡

```
#include <LEDA/rat_point.h>
#include <LEDA/window.h>
#include <LEDA/rat_window.h> // lets W >> p work for rat_points
#include <LEDA/rat_kernel_names.h>
⟨FOO.c⟩
#include <LEDA/kernel_names_undef.h>
```

⟨*foo_test.c*⟩≡

```
#include <LEDA/point.h>
#include <LEDA/window.h>

#include <LEDA/float_kernel_names.h>
⟨FOO.c⟩
#include <LEDA/kernel_names_undef.h>
```

The header file window.h is included in both specializations and it is hence tempting to write

⟨*BAD_FOO.c*⟩≡

```
#include <LEDA/window.h>

main(){

window W; W.display();

POINT p;
while ( W >> p) W << p.to_point();
}
```

This will lead to a disaster. Never include a file in a piece of code that is subject to renaming, except if you are absolutely sure that the renaming mechanism is not used in the included file. Window.h includes the entire floating point kernel which in turn includes files like transform.h. The latter file uses the renaming mechanism.

Why did we undefine all names at the end of foo_test.c and rat_foo_test.c? We found that it helps to guard against the error pointed out in the preceding paragraph. If foo_test.c is included in a file that uses the renaming mechanism the compiler will generate a message that certain names are undefined. For example

```
#include <LEDA/rat_kernel_names.h>
#include "rat_foo_test.c"
POINT p;  // POINT is undefined here
```

We use the renaming mechanism just described for all source files in src/plane_alg and for some source files in src/plane. We also use the mechanism for the header files for polygons, generalized polygons, transformations, point sets, and generation of random points. In these cases the generic header files are stored in incl/LEDA/generic.

Sometimes, a small part of the code is specific to a particular kernel. We use conditional compilation in this situation. For example,

```
// an error was just discovered
#if ( KERNEL == FLOAT_KERNEL )
cerr << "Please move to the rational kernel.";
#else
cerr << "Please report this error.";
#endif
```

The conversion functions between floating point objects and rational objects form a more substantial example. In the case of POLYGONs we have:

```
// part of POLYGON.h
POLYGON(const POLYGON& P) : handle_base(P) {} // copy constructor
#if ( KERNEL == RAT_KERNEL )
rat_polygon(const polygon& Q, int prec = 0);
#endif
#if ( KERNEL == FLOAT_KERNEL )
polygon(const polygon& Q, int prec);
#endif
polygon     to_polygon() const;
```

The first declaration defines the copy constructor for both instantiations and the last declaration defines the conversion function to *polygons* for both instantiations. The middle declaration is conditional. In class *rat_polygon* we also have the constructors

```
rat_polygon(const polygon&, int);
rat_polygon(const polygon&);
```

and in class *polygon* we also have the constructor

```
polygon(const polygon&, int prec);
```

It is important that *prec* is not an optional argument in the latter case as this would clash with the copy constructor.

We summarize: the pre-compilation mechanism of C++ allows us to write kernel independent code. Files that use the renaming mechanism must never be included in a piece of code that is subject to renaming.

8.6 The Dangers of Floating Point Arithmetic

We give two examples for the dangers of floating point arithmetic in geometric computation. Both examples show that floating point geometric objects can exhibit bizarre behavior that deviates widely from the behavior predicted by mathematics. We will see more examples in the chapter on geometry algorithms.

8.6.1 *Convex Hulls*

The first example was suggested by Stefan Schirra. Consider the following piece of code. We define a segment s and construct a set L of points consisting of the endpoints of L and the intersections between s and some number of random lines.

⟨*float_hull_test*⟩≡

```
point p0(-LEDA_PI, -LEDA_PI);
point p1(+LEDA_PI, +LEDA_PI);
segment s(p0,p1);
list<point> L; L.append(p0); L.append(p1);
for (int i = 0; i < 10000; i++)
{ double ax, ay;
  rand_int >> ax; rand_int >> ay; point p(ax*LEDA_PI, ay*LEDA_PI);
  rand_int >> ax; rand_int >> ay; point q(ax*LEDA_PI, ay*LEDA_PI);
  line l(p,q); point r;
  if ( l.intersection(s,r) ) L.append(r);
}
list<point> CH = CONVEX_HULL(L);
```

We then compute the convex hull of L, see Section 9.1. Since all points in L lie on s, the convex hull should have exactly two vertices. Figure 8.5 shows the output of a sample run of the program. The convex hull has more than two vertices, contrary to what mathematics tells us. The explanation is simple. When the intersection between s and a line l is computed with the floating point kernel, the point of intersection does not necessarily lie on s but only near s.

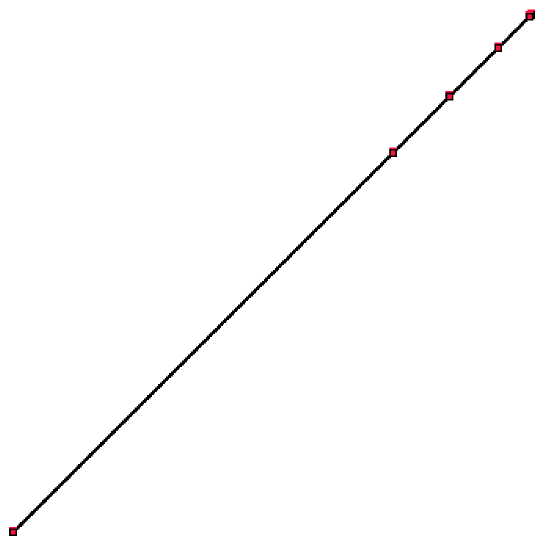

Figure 8.5 The convex hull of points contained in a common line segment computed with the floating point kernel. The hull has five vertices although there should be only two.

8.6.2 *Braided Lines (Verzopfte Geraden)*

The second example was suggested by Lyle Ramshaw who also coined the name braided lines (verzopfte Geraden in German) for it. Consider the lines

$$l_1 : \ y = 9833 \cdot x/9454 \quad \text{and} \quad l_2 : \ y = 9366 \cdot x/9005.$$

Both lines pass through the origin and the slope of l_1 is slightly larger than the slope of l_2. At $x = 9454 \cdot 9005$ we have $y_1 = 9833 \cdot 9005 = 9366 \cdot 9454 + 1 = y_2 + 1$.

The following program runs through multiples of 0.001 between 0 and 1 and computes the corresponding y-values y_1 and y_2. It compares the two y-values and, if the outcome of the comparison is different than in the previous iteration, prints x together with the current outcome.

⟨*braided_lines_test.c*⟩≡
```
#include <stream.h>
main(){
cout.precision(12);
float delta = 0.001;
int last_comp = -1;
float a = 9833, b = 9454, c = 9366, d = 9005;
for (float x = 0; x < 0.1; x = x + delta)
{ float y1 = a*x/b;   // l1 is steeper
  float y2 = c*x/d;
  int comp = (y1 < y2? -1 : (y1 == y2? 0 : +1));
```

```
    if (comp != last_comp)
    { cout <<"\n" << x << ": ";
      if (comp == -1) cout << "l1 is below l2";
      if (comp ==  0) cout << "l1 intersects l2";
      if (comp == +1) cout << "l1 is above l2";
    }
    last_comp = comp;
  }
  cout <<"\n\n";
  }
```

Clearly, we should expect the program to print

```
  0.000: l1 intersects l2
  0.001: l1 is above l2
```

Well, the first few lines of the actual output are[6] :

```
  0: l1 intersects l2
  0.00300000002608: l1 is above l2
  0.00400000018999: l1 intersects l2
  0.0050000003539: l1 is above l2
  0.00800000037998: l1 intersects l2
  0.00900000054389: l1 is below l2
  0.0100000007078: l1 is above l2
  0.0110000008717: l1 intersects l2
  0.0120000010356: l1 is above l2
  0.0130000011995: l1 intersects l2
  0.0140000013635: l1 is above l2
  0.0150000015274: l1 is below l2
  0.01600000076: l1 intersects l2
  0.0180000010878: l1 is below l2
  0.0190000012517: l1 intersects l2
```

We conclude that the lines intersect many times, contrary to what mathematics teaches us.

What went wrong? The type *float* consists of only a finite number of values and hence a line is really a step function as shown in Figure 8.6. The width of the steps of our two lines l_1 and l_2 are distinct and hence the lines intersect.

8.6.3 ***Overcoming the Dangers of Floating Point Arithmetic***

The examples above show that the implementation of geometric algorithms may be a difficult task. How can we overcome the difficulties?

The first approach sticks with inexact arithmetic but uses it more carefully. The papers [Mil88, Mil89a, Mil89b, FM91, LM90, GSS93, GSS89] develop algorithms for line

[6] This output is produced on the first author's workstation. If the program is run on the same author's notebook, it produces the correct result. The explanation for this behavior is that on the notebook double precision arithmetic is used to implement floats. According to the C++ standard floats must not offer more precision than doubles; they are not required to provide less.

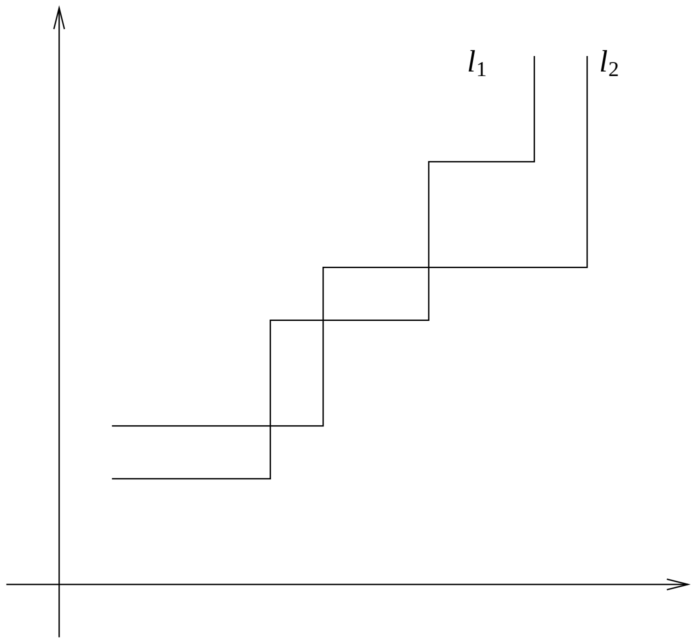

Figure 8.6 Lines as step functions and their multiple intersections.

arrangements, intersections, convex hulls, and Voronoi diagrams based on imprecise primitives. We suggest that the reader has a look at at least one of these papers in order to appreciate the ingenuity needed to overcome the shortcomings of floating point arithmetic. We were afraid of the required ingenuity and therefore did not adopt this approach for LEDA.

The alternative approach is to switch to exact arithmetic. This approach was pioneered by Karasick, Lieber, and Nackman [KLN91]. They discussed the computation of Delaunay diagrams by exact rational arithmetic. The use of exact arithmetic overcomes the correctness problems associated with floating point arithmetic, however, at the cost of a much increased running time. Fortune and van Wyk [FvW96] showed that the use of floating point filters can give exact geometric computation at low cost. We adapted their ideas[7] to the LEDA system [MN94b, MN94a]. Floating point filters are the topic of the next section.

Exercises for 8.6

1 Give a version of the intertwined lines for *double* arithmetic.

2 Play with the voronoi demo (in xlman) and try to find examples where it works incorrectly when run with the floating point kernel. Try to explain what goes wrong.

[7] The conference version of their paper appeared in 1993.

8.7 Floating Point Filters

Floating point filters apply to the evaluation of geometric predicates as used in the conditionals of geometric programs. For example,

```
switch ( orientation(a,b,c) )
{ case -1: // negative orientation
  case  0: // collinear points
  case +1: // positive orientation
}
```

Evaluating a geometric predicate is tantamount to determining the sign of an arithmetic expression. For example, the test above is equivalent to

```
switch (sign((ax*bw-bx*aw)*(ay*cw-cy*aw)-(ay*bw-by*aw)*(ax*cw-cx*aw)))
{ case -1: //
  case  0: //
  case +1: //
}
```

where *ax*, *ay*, *aw* denote the homogeneous coordinates of point *a* and similarly for the points *b* and *c*. The homogeneous coordinates of a *rat_point* are *integers* and hence evaluating the conditional involves ten multiplications and four additions of *integers*. Unfortunately, *integer* arithmetic is considerably more expensive than floating point arithmetic and hence we might expect to pay a tremendous price for exact computation.

The observation that paves the way for floating point filters is that we only want to know the sign of the arithmetic expression but not its value. It is frequently possible to determine the sign of an expression with floating point arithmetic although it is impossible to determine its value with floating point arithmetic.

In order to compute the sign of an expression[8] E, a floating point filter computes an approximation $\tilde{E}$ of E using floating point arithmetic and also a bound B on the maximal difference between $\tilde{E}$ and the (unknown) exact value E, i.e.,

$$|E - \tilde{E}| \le B,$$

or ,

$$\tilde{E} - B \le E \le \tilde{E} + B.$$

Thus:

- if $\tilde{E} > B$ then $E > 0$,
- if $\tilde{E} < -B$ then $E < 0$,
- if neither of the above, $B < 1$ and E and $\tilde{E}$ are integral then $E = 0$.

For the third item observe that if neither of the first two cases applies then $|\tilde{E}| \le B$. If $\tilde{E}$ is integral and $B < 1$ this implies $\tilde{E} = 0$. If E is integral this implies further that $E = 0$.

In order to derive a specific floating point filter one has to:

[8] We use E in the usual double meaning: it denotes an expression and also the value of the expression.

E	$\tilde{E}$	mes_E	ind_E
a, integer	$fl(a)$	$\|fl(a)\|$	1
a, float integer	$fl(a)$	$\|fl(a)\|$	0
$A + B$	$\tilde{A} \oplus \tilde{B}$	$mes_A \oplus mes_B$	$1 + \max(ind_A, ind_B) \cdot \delta$
$A - B$	$\tilde{A} \ominus \tilde{B}$	$mes_A \oplus mes_B$	$1 + \max(ind_A, ind_B) \cdot \delta$
$A \cdot B$	$\tilde{A} \odot \tilde{B}$	$mes_A \odot mes_B$	$1 + ind_A \cdot \delta + ind_B \cdot \delta^2$

Table 8.1 The recursive definition of mes_E and ind_E. The first column contains the case distinction according to the syntactic structure of E, the second column contains the rule for computing $\tilde{E}$ and the third and fourth columns contain the rules for computing mes_E and ind_E; $\oplus$ and $\odot$ denote the floating point implementations of addition and multiplication. We use the abbreviations $\delta = 1 + 2^{-53}$ and $fl(a) = a.to\_double(\,)$. For the entry in the last row and last column one may assume $ind_B \le ind_A$.

- specify how the approximation $\tilde{E}$ is computed,
- specify how the bound B is computed, and
- prove that $|E - \tilde{E}| \le B$ holds.

In the next section we will describe a variant of the floating point filter used in the rational kernel. In later sections we comment on other filters, we discuss an expression compiler for the automatic generation of floating point filters, and we give theoretical and experimental evidence for the efficacy and efficiency of floating point filters.

8.7.1 *A Floating Point Filter*

We discuss a variant of the filter used in the rational kernel. The filter described here is slightly stronger that the one described in [MN94b, MN94a]. In the current kernel you will find a mixture of both filters. The filter works for expressions with integer operands and operations addition, subtraction, and multiplication. An extension to expressions with real operands and the additional operations division and square root was later devised in [Bur96, Fun97, BFS98].

The approximation $\tilde{E}$ is simply the value obtained by evaluating E with double precision floating point arithmetic.

The bound B is computed according to the rules given in Table 8.1. This table contains the recursive definitions of the index ind_E and the measure mes_E of an expression E; B is defined as

$$B = 2^{-53} \cdot ind_E \cdot mes_E .$$

Before we prove that $\tilde{E}$ and B have the property required for a floating point filter, we apply the filter to the orientation predicate. We obtain:

```
// convert arguments to double
double axd = ax.to_double(), ayd = ay.to_double();
// and similarly for the other coordinates
// evaluate E with floating point arithmetic
double E_tilde = (axd*bwd - bxd*awd) * (ayd*cwd - cyd*awd) -
                 (ayd*bwd - byd*awd) * (axd*cwd - cxd*awd);
// compute mes by replacing all arguments by their absolute
// values and by replacing - by + in E.
double axd = fabs(axd), ayd = fabs(ayd);
// and similarly for the other coordinates
double mes = (axd*bwd + bxd*awd) * (ayd*cwd + cyd*awd) +
                 (ayd*bwd + byd*awd) * (axd*cwd + cxd*awd);
double ind = 11.0;  // see below
double B   =  ind * mes * eps;  // eps = 2^{-53}.
if ( E_tilde >  B ) return  1;
if ( E_tilde < -B ) return -1;
if ( B < 1)         return  0;
// resort to integer arithmetic
return sign((ax*bw-bx*aw)*(ay*cw-cy*aw)-(ay*bw-by*aw)*(ax*cw-cx*aw));
```

Some comments on this program are in order.

(1) How did we compute the index? We have:
The index of an integer a is $s_1 = 1$;
The index of an expression of the form $a \cdot a$ is $s_2 = 1 + s_1(\delta + \delta^2) \approx 3$.
The index of an expression of the form $a \cdot a + a \cdot a$ is $s_3 = 1 + s_2\delta \approx 4$.
The index of an expression of the form $(a \cdot a + a \cdot a) \cdot (a \cdot a + a \cdot a)$ is $s_4 = 1 + s_3(\delta + \delta^2) \approx 9$.
The index of the orientation predicate is $s_5 = 1 + s_4\delta \approx 10$.
s_5 is slightly larger than 10 and certainly less than 11. We may therefore use 11 as the index of the expression predicate. This overestimate of ind_E will also cover any rounding error in the computation of B. Note that we defined B as $2^{-53} \cdot ind_E \cdot mes_E$ but compute $2^{-53} \odot ind_E \odot mes_E$, where $\odot$ denotes floating point multiplication.

(2) The computation of $\tilde{E}$ starts with the conversion of the homogeneous coordinates of a, b, and c from *integer* to *double*. In the rational kernel we make this conversion when the points are constructed. In this way the conversion is made only once for each *rat_point* and not every time a predicate is evaluated for a *rat_point*.

(3) The computation of mes_E involves the same number of arithmetic operations as the computation of $\tilde{E}$. The computation of B requires, in addition, to take the absolute values of the arguments and to multiply ind_E, mes_E, and 2^{-53}. The number of operations to compute B is therefore at least the number of operations to compute $\tilde{E}$. The actual time required to compute $\tilde{E}$ and B is usually less than twice the time to compute $\tilde{E}$ alone (see Section 8.7.4 for some measurements), since modern micro-processors have highly effective floating point units with multiple pipelined arithmetic units and since the cost of arithmetic is small once the data is in the processing unit.

(4) Our expressions have integer operands and operations $+$, $-$, and $\cdot$. Hence E and $\tilde{E}$ are integral.

We will next prove that Table 8.1 indeed defines a valid bound B. We need to review some properties of the IEEE floating point standard [Gol90, Gol91, IEE87].

A floating point number consists of a sign s, a mantissa m, and an exponent e. In double format s has one bit, m consists of fifty-two bits $m_1, \ldots, m_{52}$, and e consists of the remaining eleven bits of a double word. The number represented by the triple (s, m, e) is defined as follows:

- e is interpreted as an integer in $[0 \mathinner{..} 2^{11} - 1] = [0 \mathinner{..} 2047]$.
- If $m_1 = \ldots = m_{52} = 0$ and $e = 0$ then the number is +0 or -0 depending on s.
- If $1 \le e \le 2046$ then the number is $s \cdot (1 + \sum_{1 \le i \le 52} m_i 2^{-i}) \cdot 2^{e-1023}$.
- If some m_i is non-zero and $e = 0$ then the number is $s \cdot \sum_{1 \le i \le 52} m_i 2^{-i} 2^{-1023}$. This is a so-called denormalized number.
- If all m_i are zero and $e = 2047$ then the number is $+\infty$ or $-\infty$ depending on s.
- In all other cases the triple represents NaN (= not a number).

The largest positive double (except for ∞) is `MAXDOUBLE` $= (2 - 2^{-52}) \cdot 2^{1023}$ and the smallest positive double is `MINDOUBLE` $= 2^{-52} \cdot 2^{-1023}$.

In this section we are interested in *floating point integers*, i.e., integers that can be represented as floating point numbers. The set of floating point integers consists of:

- the number zero,
- all integers of the form $s \cdot (1 + \sum_{1 \le i \le 52} m_i 2^{-i}) \cdot 2^e$ with $0 \le e \le 1023$ (we must have $m_i = 0$ for $i > e$),
- the numbers $+\infty$ and $-\infty$.

We call an integer *representable* if $|a| \le 2 \cdot 2^{1023}$. For a representable integer a, let $fl(a)$ be a floating point number nearest to a. For a non-representable integer let $fl(a) = \pm\infty$ depending on the sign of a.

Floating point arithmetic incurs rounding error. It is therefore important to distinguish between the mathematical operations addition, subtraction, multiplication and their floating point implementations. We use $+$, $-$, and $\cdot$ for the exact operations and $\oplus$, $\ominus$, and $\odot$ for their floating point implementations.

We need the following facts:

(a) If a is an integer then

$$|a - fl(a)| \le 2^{-53} \cdot |fl(a)|,$$

where *eps* $= 2^{-53}$ is called the *machine precision*. If a is a non-representable integer (including $\pm\infty$) then $fl(a) = \infty$ and the claim is true. So, assume that a is representable. The floating point approximation of a is obtained by "rounding" in the 53-rd bit. More precisely, if $|a| < 2^{53}$ then $fl(a) = a$ and if $|a| \geq 2^{53}$ and a has the binary representation

$$a = s \cdot \sum_{0 \leq i \leq L} m_i \cdot 2^{L-i}$$

with $m_0 = 1$ and $L \geq 53$, then

$$fl(a) = s \cdot (\sum_{0 \leq i \leq 52} m_i \cdot 2^{L-i} + \delta \cdot 2^{L-52}),$$

where $\delta \in \{0, 1\}$ is chosen such that the better approximation of a is obtained. Clearly, $|a - fl(a)| \leq 2^{L-52}/2$ and $|fl(a)| \geq 2^L$. Thus, $|a - fl(a)| \leq 2^{-53} \cdot |fl(a)|$.

We want to remark that the assumption that a is integer is crucial for claim (a). If $|a| \leq$ *MinDouble*$/2$, the best floating point approximation of a is zero. Thus, there is no bound on the error $|a - fl(a)|$ in terms of $fl(a)$. Life is easier for integers.

(b) If a is an integer then $fl(a)$ is a floating point integer.

(c) If f_1 and f_2 are floating point integers, op $\in \{+, -, \cdot\}$, $f = f_1$ op f_2, and $\widetilde{\text{op}}$ is the floating point implementation of op, then

$$f_1 \widetilde{\text{op}} f_2 = fl(f),$$

i.e., the floating point operation returns a floating point integer closest to f. There is no need to argue here. It is an "axiom" of the IEEE standard that every arithmetic operation is implemented with the least possible error.

(d) Under the same hypothesis as in the preceding item:

$$|f_1 \widetilde{\text{op}} f_2 - f_1 \text{ op } f_2| \leq 2^{-53} |f_1 \widetilde{op} f_2|.$$

Let $\tilde{f} = f_1 \widetilde{\text{op}} f_2$ and $f = f_1$ op f_2. Then $\tilde{f} = fl(f)$ by (c) and hence $|\tilde{f} - f| \leq 2^{-53}|\tilde{f}|$ by part (a).

(e) If f is an *integer* then *a.to_double*( ) returns $fl(a)$. That is the way we implemented the function *to_double*.

(f) Floating point arithmetic is monotone, i.e., if $a_1 \leq a_2$ and $b_1 \leq b_2$ then $a_1 \oplus a_2 \leq b_1 \oplus b_2$ and if $0 \leq a_1 \leq a_2$ and $0 \leq b_1 \leq b_2$ then $a_1 \odot a_2 \leq b_1 \odot b_2$.

(g) Multiplication by a power of two incurs no rounding error, i.e., if a is a power of two and b is a floating point integer such that $2a$ and $a \cdot b$ are representable, then $a \oplus a = 2 \cdot a$ and $a \odot b = a \cdot b$.

Theorem 1 *If mes_E and ind_E are computed according to Table 8.1 then $|\tilde{E}| \leq mes_E$ and*

$$\begin{aligned} |\tilde{E} - E| &\leq 2^{-53} \cdot ind_E \cdot mes_E \\ &\leq 2^{-53} \odot ind_E \odot mes_E \odot (1 + 2^{-52}). \end{aligned}$$

Proof We use induction on the structure of the expression E. The claim $|\tilde{E}| \le mes_E$ follows immediately from the monotonicity of floating point arithmetic. For the other claims we have to work slightly harder. We first prove

$$|\tilde{E} - E| \le 2^{-53} \cdot ind_E \cdot mes_E.$$

Assume first that E is an integer a. Then

$$|a - fl(a)| \le 2^{-53} \cdot |fl(a)|$$

by item (a) and the claim is certainly true. If a is a floating point integer then $fl(a) = a$ and hence the index can be set to zero for floating point integers.

We come to the induction step. Let A and B be the two subexpressions of E and let $\tilde{A}$ and $\tilde{B}$ be their floating point values. Then

$$\begin{array}{rcl} |\tilde{A}| & \le & mes_A \\ |\tilde{A} - A| & \le & 2^{-53} \cdot ind_A \cdot mes_A \\ |\tilde{B}| & \le & mes_B \\ |\tilde{B} - B| & \le & 2^{-53} \cdot ind_B \cdot mes_B \end{array}$$

by induction hypothesis.

We now make a case distinction according to the operation combining A and B.

Assume $E = A + B$. Then

$$|\tilde{E} - E| = |\tilde{A} \oplus \tilde{B} - (A + B)| \le |\tilde{A} \oplus \tilde{B} - (\tilde{A} + \tilde{B})| + |\tilde{A} - A| + |\tilde{B} - B|.$$

Item (d) with $f_1 = \tilde{A}$ and $f_2 = \tilde{B}$ implies that the first term is bounded by $2^{-53}|\tilde{A} \oplus \tilde{B}|$ and monotonicity of floating point arithmetic implies that

$$|\tilde{A} \oplus \tilde{B}| \le mes_A \oplus mes_B = mes_E.$$

For the other two terms we use the induction hypothesis to conclude

$$\begin{array}{rcl} |\tilde{A} - A| + |\tilde{B} - B| & \le & 2^{-53} \cdot (ind_A \cdot mes_A + ind_B \cdot mes_B) \\ & \le & 2^{-53} \cdot \max(ind_A, ind_B) \cdot (mes_A + mes_B) \\ & \le & 2^{-53} \cdot \max(ind_A, ind_B) \cdot (1 + 2^{-53}) \cdot (mes_A \oplus mes_B) \\ & = & 2^{-53} \cdot \max(ind_A, ind_B) \cdot (1 + 2^{-53}) \cdot mes_E. \end{array}$$

Putting the two bounds together completes the induction step for the case of an addition. The argument for subtractions is completely analogous.

We turn to multiplications, $E = A \cdot B$. We have

$$|\tilde{E} - E| = |\tilde{A} \odot \tilde{B} - A \cdot B| \le |\tilde{A} \odot \tilde{B} - \tilde{A} \cdot \tilde{B}| + |\tilde{A} \cdot \tilde{B} - A \cdot \tilde{B}| + |A \cdot \tilde{B} - A \cdot B|.$$

Item (d) with $f_1 = \tilde{A}$ and $f_2 = \tilde{B}$ implies that the first term is bounded by $2^{-53}|\tilde{A} \odot \tilde{B}|$ and monotonicity of floating point arithmetic implies that

$$|\tilde{A} \odot \tilde{B}| \le mes_A \odot mes_B = mes_E.$$

For the second term we use the induction hypothesis to conclude

$$\begin{aligned} |\tilde{A} \cdot \tilde{B} - A \cdot \tilde{B}| &= |\tilde{A} - A| \cdot |\tilde{B}| \\ &\le 2^{-53} \cdot ind_A \cdot mes_A \cdot mes_B \\ &\le 2^{-53} \cdot ind_A \cdot (1 + 2^{-53}) \cdot (mes_A \odot mes_B) \\ &= 2^{-53} \cdot ind_A \cdot (1 + 2^{-53}) \cdot mes_E, \end{aligned}$$

and for the third term we conclude analogously

$$\begin{aligned} |A \cdot \tilde{B} - A \cdot B| &= |A| \cdot |\tilde{B} - B| \\ &\le (1 + 2^{-53}) \cdot |\tilde{A}| \cdot 2^{-53} \cdot ind_B \cdot mes_B \\ &\le (1 + 2^{-53}) \cdot mes_A \cdot 2^{-53} \cdot ind_B \cdot mes_B \\ &\le 2^{-53} \cdot ind_B \cdot (1 + 2^{-53})^2 (mes_A \odot mes_B) \\ &= 2^{-53} \cdot ind_B \cdot (1 + 2^{-53})^2 \cdot mes_E. \end{aligned}$$

Putting the three bounds together completes the induction step for the case of a multiplication.

It remains to prove the inequality

$$2^{-53} \cdot ind_E \cdot mes_E \le 2^{-53} \odot ind_E \odot mes_E \odot (1 + 2^{-52}).$$

It follows from

$$ind_E \cdot mes_E \le (ind_E \odot mes_E) \cdot (1 + 2^{-53}) \le ind_E \odot mes_E \odot (1 + 2^{-52})$$

and the fact that the multiplication by 2^{-53} incurs no rounding error. □

8.7.2 *Alternative Filters*

We discuss the filter originally (and still mostly) used in the kernel, static and dynamic filters, special methods for determinants, and specialized arithmetics.

The Filter Used Originally in the Kernel: In our original filter we computed ind_E and mes_E according to Table 8.2. In this table we also define a quantity P_E. P_E is a power of two with $|E| \le P_E$, $|\tilde{E}| \le P_E$, and $P_E \le mes_E$. The bound $B(E)$ is defined as

$$B = 2^{-53} \odot ind_E \odot mes_E.$$

In order to see that this bound is correct one proves that

$$|E - \tilde{E}| \le 2^{-53} \cdot ind_E \cdot P_E \quad \text{and} \quad P_E \le mes_E$$

and observes that

$$2^{-53} \cdot ind_E \cdot P_E = 2^{-53} \odot ind_E \odot P_E \le 2^{-53} \odot ind_E \odot mes_E,$$

since 2^{-53} and P_E are powers of two and since floating point arithmetic is monotonic.

The inequality

$$|E - \tilde{E}| \le 2^{-53} \cdot ind_E \cdot P_E$$

is again shown by induction on the structure of E. The base case is obvious. The induction steps are as follows.

In the case of an addition we have

$$\begin{aligned} |E - \tilde{E}| &= |\tilde{A} \oplus \tilde{B} - (A + B)| \;=\; |\tilde{A} \oplus \tilde{B} - (\tilde{A} + \tilde{B})| + |\tilde{A} - A| + |\tilde{B} - B| \\ &\le 2^{-53}(|\tilde{A} \oplus \tilde{B}| + ind_A P_A + ind_B P_B) \\ &\le 2^{-53}(P_A \oplus P_B + (ind_A + ind_B)\max(P_A, P_B)) \\ &\le 2^{-53}(1 + (ind_A + ind_B)/2) \cdot 2 \cdot \max(P_A, P_B)), \end{aligned}$$

where the last inequality follows from

$$\begin{aligned} P_A \oplus P_B &\le \max(P_A, P_B) \oplus \max(P_A, P_B) \\ &= \max(P_A, P_B) + \max(P_A, P_B) \;=\; 2 \cdot \max(P_A, P_B). \end{aligned}$$

In the case of multiplication we have

$$\begin{aligned} |E - \tilde{E}| &= |\tilde{A} \odot \tilde{B} - \tilde{A} \cdot \tilde{B}| + |\tilde{A}| \cdot |\tilde{B} - B| + |B| \cdot |\tilde{A} - A| \\ &\le 2^{-53}(|\tilde{A} \odot \tilde{B}| + |\tilde{A}||\tilde{B} - B| + |B||\tilde{A} - A|) \\ &\le 2^{-53}(P_A \odot P_B + P_A \cdot ind_B \cdot P_B + P_B \cdot ind_A \cdot P_A) \\ &\le 2^{-53}(1 + ind_A + ind_B) \cdot P_A \cdot P_B. \end{aligned}$$

The inequality $P_E \le mes_E$ is also shown by induction on the structure of E. We leave the induction step to the reader. For the basis of the induction we observe that $2^{\log|a|} \le 2 \cdot fl(a) = mes_a$ for an integer a.

This concludes the proof that Table 8.2 defines a filter.

For the orientation predicate Table 8.2 gives an index of 5 and a measure of $8 \cdot M$, where M is the measure according to Table 8.1. Thus $B = 40 \cdot M$. Table 8.1 gives $B = 11 \cdot M$, which is significantly better.

Static Filters: Fortune and van Wyk [FvW96] invented the idea of a floating point filter. They proposed a static filter in which B is precomputed completely. Assume that it is known apriori that $|a| \le 2^L$ for all integer arguments of an expression E. Then $mes_a \le 2^L$ for all arguments a and we may *precompute* mes_E by replacing mes_a by 2^L for all arguments a. This yields $B = 2^{-53} \cdot 11 \cdot 2^{4L+3}$ with Table 8.1. The filter of Fortune and van Wyk is called *static* because B is precomputed entirely. In contrast, the filter used in the rational kernel precomputes ind_E but computes mes_E on the fly. Such a filter may be called *semi-dynamic*.

Static filters are faster than semi-dynamic filters, but they are less precise and they are less convenient to use. For example, they cannot be used at all in an on-line algorithm, where no apriori bound on the size of the arguments is known. We decided against static filters because of their less convenient use.

E	$\tilde{E}$	P_E	mes_E	ind_E
a, integer	$fl(a)$	$2^{\lceil \log \lvert a \rvert \rceil}$	$2\lvert fl(a)\rvert$	1
a, float integer	$fl(a)$	$2^{\lceil \log \lvert a \rvert \rceil}$	$2\lvert fl(a)\rvert$	0
$A+B$	$\tilde{A} \oplus \tilde{B}$	$2\max(P_A, P_B)$	$2(mes_A \oplus mes_B)$	$1 + (ind_A + ind_B)/2$
$A-B$	$\tilde{A} \ominus \tilde{B}$	$2\max(P_A, P_B)$	$2(mes_A \oplus mes_B)$	$1 + (ind_A + ind_B)/2$
$A \cdot B$	$\tilde{A} \odot \tilde{B}$	$P_A P_B$	$mes_A \odot mes_B$	$1 + ind_A + ind_B$

Table 8.2 The recursive definition of mes_E and ind_E in the original filter. P_E is a power of two with $|E| \le P_E$, $|\tilde{E}| \le P_E$, and $P_E \le mes_E$; it is only needed for the correctness proof of the filter. We set $2^{\lceil \log 0 \rceil} = 0$.

Dynamic Filters: Consider the expression

$$E = (a+b) - a$$

when a and b are float integers and $a \gg b$. The semi-dynamic filter of Section 8.7.1 assumes that the error in the subtraction may be as large as

$$2^{-53} mes_E \approx 2^{-53}(2a+b).$$

However, the actual error is approximately

$$2^{-53} \cdot \tilde{E} \approx 2^{-53} \cdot b,$$

which is much smaller.

Dynamic filters attempt to exploit this differency by estimating the round-off error more carefully. They use the formulae

$$\begin{aligned} |\tilde{A} \oplus \tilde{B} - (A+B)| &\le |\tilde{A} \oplus \tilde{B} - (\tilde{A} + \tilde{B})| + |\tilde{A} - A| + |\tilde{B} - B| \\ &\le 2^{-53}|\tilde{A} \oplus \tilde{B}| + |\tilde{A} - A| + |\tilde{B} - B| \end{aligned}$$

and

$$\begin{aligned} |\tilde{A} \odot \tilde{B} - A \cdot B| &= |\tilde{A} \odot \tilde{B} - \tilde{A} \cdot \tilde{B} + \tilde{A} \cdot \tilde{B} - A \cdot \tilde{B} + A \cdot \tilde{B} - A \cdot B| \\ &\le 2^{-53}|\tilde{A} \odot \tilde{B}| + |\tilde{A} - A| \cdot |\tilde{B}| + |A||\tilde{B} - B| \end{aligned}$$

to recursively compute a bound on the error. More precisely, in the case of an addition the error err_E for the expression E is computed as

$$err_e = (2^{-53} \odot |\tilde{E}| \oplus err_A \oplus err_B) \odot (1 + 2^{-51}),$$

where the multiplication by $1 + 2^{-51}$ accounts for the error in the computation of the error bound. We leave it to the reader to derive the corresponding formula for multiplication.

Dynamic filters are more costly but also more precise than semi-dynamic filters. Observe that the computation of err_E in the case of an addition requires two additions and two multiplications. The computation of mes_E requires only one addition. We concluded from our experiments in [MN94b] that the additional cost is not warranted for the rational kernel.

We do use dynamic filters in the number type *real*, see Section 4.4, since the cost of exact computation is very high for *reals* and hence a higher computation time for the filter is justified.

Determinants: Many geometric predicates, e.g., the orientation and the insphere predicates, are naturally formulated as the sign of a determinant. The efficient computation of the signs of determinants has therefore received special attention [Cla92, ABDP97, BEPP97]. None of the methods is available in LEDA.

Specialized Arithmetics: Consider again the orientation predicate

```
sign((ax*bw-bx*aw)*(ay*cw-cy*aw) - (ay*bw-by*aw)*(ax*cw-cx*aw) )
```

and assume that it is known that the absolute value of all arguments is less than 2^L. The arguments are assumed to be integer. It is then easy to compute an apriori bound on the maximal number of binary digits required for any of the intermediate results. We have:
The integer a requires L bits;
An expression of the form $a \cdot a$ requires $2L$ bits.
An expression of the form $a \cdot a + a \cdot a$ requires $2L + 1$ bits.
An expression of the form $(a \cdot a + a \cdot a) \cdot (a \cdot a + a \cdot a)$ requires $4L + 2$ bits.
The orientation predicate requires at most $4L + 3$ bits.
Given this knowledge one could try to optimize the arithmetic, i.e., instead of using a general purpose package for the computation with arbitrary precision integers (such as the class *integer*) one could design integer arithmetic optimized for a particular bit length. This avenue is taken in [FvW96, She97].

8.7.3 *Expression Compilers*

The incorporation of the floating point filter into the rational kernels was a tedious task; it was done to a large extent by Ulrike Bartuschka. For each predicate she had to derive manually the formulae for ind_E and mes_E. For example, the code for the orientation test contains the following comment:

```
-----------------------------------------------------------------------
 ERROR BOUNDS
-----------------------------------------------------------------------
 mes(E) = 2*(mes(aybw-byaw)*mes(axcw-cxaw) + mes(axbw-bxaw)*mes(aycw-cyaw))
        = 2*(4*(fabs(aybw)+fabs(byaw)) * (fabs(axcw)+fabs(cxaw)) +
             4*(fabs(axbw)+fabs(bxaw)) * (fabs(aycw)+fabs(cyaw)))
        = 8*((fabs(aybw)+fabs(byaw)) * (fabs(axcw)+fabs(cxaw)) +
             (fabs(axbw)+fabs(bxaw)) * (fabs(aycw)+fabs(cyaw)))

 ind(E) = ((ind(aybw-byaw) + ind(axcw-cxaw) +0.5) +
           (ind(axbw-bxaw) + ind(aycw-cyaw) +0.5) + 1 ) / 2
```

```
       = (4.5 + 4.5 + 1) / 2  =  5

 eps(E) = ind(E) * mes(E) * eps0
        = 40 * ((fabs(aybw)+fabs(byaw))*(fabs(axcw)-fabs(cxaw)) +
               (fabs(axbw)-fabs(bxaw))*(fabs(aycw)-fabs(cyaw))) * eps0;
---------------------------------------------------------------------------
```

Already Fortune and Wyk [FvW96] observed that the generation of the filters can be automated. Stefan Funke [Fun97, BFS98] adopted the idea for LEDA and generalized it to a larger class of expressions and number types. His expression compiler generates floating point filters automatically from suitably decorated expressions. For example, in order to generate a filter for the orientation predicate one writes

```
int orientation(const rat_point& a, const rat_point& b,
                                    const rat_point& c)
{ int res_sign;
BEGIN_PREDICATE
{
DECLARE_ATTRIBUTES integer_type FOR a.X() a.Y() a.W() b.X()
                               b.Y() b.W() c.X() c.Y() c.W();
  integer AX=a.X(); integer AY=a.Y(); integer AW=a.W();
  integer BX=b.X(); integer BY=b.Y(); integer BW=b.W();
  integer CX=c.X(); integer CY=c.Y(); integer CW=c.W();
  integer D= (AX*BW-BX*AW) * (AY*CW-CY*AW) -
           (AY*BW-BY*AW) * (AX*CW-CX*AW);
  res_sign=sign(D);
}
END_PREDICATE
  return res_sign;
}
```

The expression compiler produces a (very lengthy) program of the following form.

```
int orientation(const rat_point& a, const rat_point& b,
                                    const rat_point& c)
{ int res_sign;
{
  /* a floating point evaluation of the predicate which assigns
     one of -1, 0, +1, NO_IDEA to res_sign   */
  if (res_sign == NO_IDEA)
  { /* exact evaluation of predicate with result in res_sign */
  }
}
 return res_sign;
}
```

The expression compiler is available as an LEP.

8.7.4 *Efficacy and Efficiency of Filters*

We discuss the efficacy and the efficiency of floating point filters. Efficacy refers to the percentage of tests, for which the filter is able to deduce the sign of the test, and efficiency refers to the cost of the evaluation of the filter and the relationship of this cost to the cost of a computation with integers.

A floating point filter for an expression E computes an approximation $\tilde{E}$ of E and a bound B for the maximal difference between the approximation and the exact value. The following lemma is trivial but useful.

Lemma 4 *If E and $\tilde{E}$ are integral and $B < 1$ then sign$(\tilde{E}) =$ sign(E).*

Under what conditions can we claim that $B < 1$ without actually computing it? Consider the orientation predicate for points with integer homogeneous coordinates $(x, y, 1)$ with $|x|, |y| \leq 2^L$. We assume that L is small enough such that the coordinates are floating point integers. The orientation predicate for points a, b, and c is given by the expression

```
E = (AX - BX) * (AY - CY) - (AY - BY) * (AX - CX)
```

and hence $B \leq 8 \cdot 2^{-53} \cdot 2^{2L+3}$ according to Theorem 1; the index of the expression is 7 when computed with $\delta = 1$. We rounded up to 8 to account for the fact that $\delta = 1 + 2^{-53}$.

We have $8 \cdot 2^{-53} \cdot 2^{2L+3} < 1$ iff $3 - 53 + 2L + 3 < 0$ iff $L < 47/2$. We conclude that double precision floating point arithmetic is guaranteed to give the correct result if the x- and y-coordinates are at most 2^{23}.

What happens if L is larger? The floating point computation is able to deduce the sign of E if $|\tilde{E}| > B$. Since E is twice the signed area (see Lemma 8.2.1) of the triangle with vertices (a, b, c), the floating point computation is able to deduce the correct sign for any triple of points which span a triangle whose area is at least $8 \cdot 2^{-53} \cdot 2^{2L+3}/2$. Devillers and Preparata [DP96] have shown that for a random triple of points and for L going to infinity, the probability that the area of the spanned triangle is at least $8 \cdot 2^{-53} \cdot 2^{2L+3}/2$ goes to one. Thus for large L and for triples of random points, the floating point computation will almost always be able to deduce the sign of E and exact computation will be rarely needed.

Observe that the result cited in the previous paragraph depends crucially on the fact that the points are chosen randomly. In an actual computation orientation tests will not be performed for random triples of points even if the input consists of random points. It is therefore not clear what the result says about actual computations.

The class *rat_point* has a static member function *print_statistics* which gives information about the efficacy of its floating point filter. The call

```
rat_point::print_statistics();
```

prints a statistic of the following form:

```
compare:           167 / 44330   (0.38 %)
orientation:        71 / 48975   (0.14 %)
side of circle:   3194 / 22317   (14.31 %)
```

The statistic states for each of the functions *compare*, *orientation*, and *side_of_circle* how many times it was evaluated and how many times the filter failed and an exact computation was necessary. In this particular execution, 22317 side of circle tests were performed out of which 3194 required exact computation. This amounts to 14.31 percent.

Table 8.3 shows the results of a more substantial experiment. The table was generated by the program below. We first generate a list *L0* of *n* random points either on the unit circle or in the unit square. We then construct a list *L1* of points whose homogeneous coordinates are *d* bit binary numbers for different values of *d* by truncating the Cartesian coordinates to *d* bits; for $d = 60$ no truncation takes place (this is indicated by the infinity-sign in Table 8.3. We construct the Delaunay diagram for the points in *L1*.

⟨*produce efficacy of filter table*⟩≡

```
int n = 10000;

list<rat_point> L0;

for (int k = 0; k < 2; k++)
{ if ( k == 0 ) random_points_on_unit_circle(n,L0);
  else          random_points_in_unit_square(n,L0);

  for (int d = 8; d <= 60; d += d < 12 ? 2 : 10)
  { list<rat_point> L1;
    rat_point p;
    I.write_table("\n");
    if ( d <= 50 )
    { double D = ldexp(1,d);
      forall(p,L0) L1.append(rat_point(integer(p.xcoordD()*D),
                                       integer(p.ycoordD()*D),1));
      I.write_table("",d);
    }
    else
    { L1 = L0;
      I.write_table("$ \\infty $");
    }
    ⟨reset counters to zero⟩

    GRAPH<rat_point,int>  DT;
    DELAUNAY_TRIANG(L1,DT);

    ⟨write a line of the table⟩

  }
  I.write_table(" \\hline");
}
```

For each experiment we generate one line in Table 8.3. The class *rat_point* has static data members that keep a count of the number of compare, orientation, and side of circle tests performed and also of the number of tests where the filter fails. Before each experiment we set the counters to zero. After each experiment we print a line of the table.

⟨*reset counters to zero*⟩≡

```
rat_point::cmp_count = 0;
rat_point::exact_cmp_count = 0;

rat_point::orient_count = 0;
rat_point::exact_orient_count = 0;

rat_point::soc_count = 0;
rat_point::exact_soc_count = 0;
```

		Compare			Orientation			Side of circle		
d	N	number	exact	%	number	exact	%	number	exact	%
8	1883	157814	0	0.00	19909	0	0.00	7242	0	0.00
10	5298	187379	0	0.00	58263	0	0.00	20736	5743	27.70
12	8383	216679	0	0.00	89307	0	0.00	35931	24693	68.72
22	9999	230556	0	0.00	98899	0	0.00	46410	42454	91.48
32	9999	231656	0	0.00	90664	137	0.15	40003	39797	99.49
42	9999	231665	0	0.00	91205	152	0.17	40083	40083	100.00
∞	9999	231665	125	0.05	44279	87	0.20	13082	13082	100.00
8	9267	230060	0	0.00	130431	0	0.00	64176	0	0.00
10	9953	236690	0	0.00	147814	0	0.00	77409	136	0.18
12	9996	236661	0	0.00	149233	0	0.00	78693	105	0.13
22	10000	235727	0	0.00	149057	0	0.00	78695	113	0.14
32	10000	235729	0	0.00	149059	0	0.00	78695	115	0.15
42	10000	235729	0	0.00	149059	0	0.00	78695	115	0.15
∞	10000	235729	574	0.24	149059	0	0.00	78695	115	0.15

Table 8.3 Efficacy of floating point filter: The top part contains the results for random points on the unit circle and the lower part contains the results for random points in the unit square. In each case we generated 10000 points. The first column shows the precision (= number of binary places) used for the homogeneous coordinates of the points, the second column contains the number of distinct points in the input. The other columns contain the number of tests, the number of exact tests, and the percentage of exact tests performed for the compare, the orientation, and the side of circle primitive.

Table 8.3 confirms the theoretical considerations from the beginning of the section. For each test there is a value of d below which the floating point computation is able to decide all tests. For the orientation test this value of d is somewhere between 22 and 32 (we argued above that the value is $47/2$) and for the side of circle test the value is somewhere between 8 and 10 (we ask the reader in the exercises to compute the exact value). Also, the percentage of the tests, where the filter fails, is essentially an increasing function of d.

The compare, orientation, and side of circle functions seem to be tests of increasing difficulty. This is easily explained. The compare function decides the sign of a linear function of the Cartesian coordinates of two points, the orientation function decides the sign of a quadratic function of the Cartesian coordinates of three points, and the side of circle function decides the sign of a polynomial of degree four in the Cartesian coordinates of four points. The larger the degree of the polynomial of the test, the larger the arithmetic demand of the test.

Among the two sets of inputs, the random points on the unit circle are much more difficult than the random points in the unit square, in particular, for the side of circle test. Again this is easily explained.

For the side of circle test, four almost co-circular points or four exactly co-circular points are the most difficult input, and for sufficiently large d the situation that $|\tilde{E}| \leq B$ and $B > 1$ arises frequently. Points on (or near) the unit circle cause no particular difficulty for the compare and the orientation function. Points on (or near) a segment would prove to be difficult for the orientation test.

For random points in the unit square the filter is highly effective for all three tests; the filter fails only for a very small percentage of the tests.

We turn to the question of how much a filter saves with respect to running time. Table 8.4 was produced by the following program.

⟨*produce efficiency of filter table*⟩≡

```
forall(p,L1) Lf.append(p.to_point());

GRAPH<rat_point,int>  DT;
GRAPH<rat_point,int>  DT_no_filter;
GRAPH<    point,int>  DT_FK;

float T = used_time();
DELAUNAY_TRIANG(Lf,DT_FK);
I.write_table(" & ", used_time(T));
```

⟨*efficiency table: check correctness of float computation*⟩

```
used_time(T);  // to set the timer
DELAUNAY_TRIANG(L1,DT);
I.write_table(" & ", used_time(T));

rat_point::use_filter = 0;
DELAUNAY_TRIANG(L1,DT_no_filter);
I.write_table(" & ", used_time(T));
rat_point::use_filter = 1;
```

We generated the same list *L1* of *rat_points* as above. We then converted each *rat_point* to a *point* to obtain a list *Lf* of *points*. Finally, we computed the Delaunay triangulation in three different ways: first with the floating point kernel, then with the rational kernel, and finally with the rational kernel without its floating point filter. The class *rat_point* has a static variable *use_filter* which controls the use of the floating point filter.

Table 8.4 has to be interpreted with care. Let us first inspect the individual columns.

The running time with the floating point kernel does not increase with the precision of the input. Observe, that for $d < 22$ and points on the unit circle, the input contains a significant fraction of multiple points (see the second column of Table 8.3) and hence the first three lines really refer to simpler problem instances. For $d \geq 22$ and points on the unit circle and for $d \geq 10$ and points in the unit square the input contains almost no multiple points and the running times are independent of the precision. The computation with the floating point

d	Float kernel	Rational kernel	RK without filter
8	0.73	1.12	4.35
10	1.3	2.43	7.8
12	1.85	5.09	11.18
22	2.17	7.93	14.4
32	2.02	7.79	13.29
42	2.01	8.32	15.46
∞	2*	5.09	9.19
8	2.58	3.59	16.33
10	2.8	3.98	18.36
12	2.83	4.04	18.63
22	2.82	4.02	20.51
32	2.86	3.96	20.77
42	2.83	4.01	26.02
∞	2.83	3.99	33.2

Table 8.4 Efficiency of the floating point filter: The top part contains the results for random points on the unit circle and the lower part contains the results for random points in the unit square. The first column shows the precision (= number of binary places) used for the Cartesian coordinates of the points. The other columns show the running time with the floating point filter, with the rational kernel with the floating point filter, and with the rational kernel without its floating point filter. A star in the second column indicates that the computation with the floating point kernel produced an incorrect result. geometry kernels!running time

kernel is not guaranteed to give the correct result. In fact, it produced an incorrect result in one of the experiments (indicated by a *). We come back to this point below.

The running time with the rational kernel and no filter increases sharply as a function of the precision. This is due to the fact that larger precision means larger integers and hence larger computation time for the integer arithmetic. We see one exception in the table. For points on the unit circle the computation on the exact points is faster than the computation with the rounded points. The explanation can be found in Table 8.3. The number of tests performed is much smaller for exact inputs than for rounded inputs. Observe, that for points that lie exactly on a circle any triangulation is Delaunay.

The running time for the rational kernel (with the filter) increases only slightly for the second set of inputs and increases more pronouncedly for the points on the unit circle. This is to be expected because the filter fails more often for the points on the unit circle.

Let us next compare columns.

The comparison between the last two columns shows the efficiency gained by the floating point filter. The gains are impressive, in particular, for the easier set of inputs. For random points in the unit square, the computation without the filter is between five and almost ten times slower. For random points on a unit circle the gain is less impressive, but still substantial. The running time without the filter is between two and five times higher than with the filter.

The comparison between the second and the third column shows what we might gain by further improving our filter technology. For our easier set of inputs the computation with the rational kernel is about 50% slower than the computation with the floating point kernel. This increase in running time stems from the computation of the error bound B in the filter. For our harder set of inputs the difference between the rational kernel and the floating point kernel is more pronounced. This is to be expected since the rational kernel resorts to exact computation more frequently for the harder inputs. The floating point kernel produced the incorrect result in one of the experiments.

We used the following piece of code to check the correctness of the computation with the floating point kernel. We make a copy *DT_FK1* of the graph computed with the floating point kernel, in which every *point* is converted to a *rat_point*. This conversion is without loss of precision. We then check whether the copy is a Delaunay triangulation; the check is discussed in Section 9.4.3. The check is executed with the rational kernel and is therefore exact.

⟨*efficiency table: check correctness of float computation*⟩≡

```
GRAPH<rat_point,int> DT_FK1;
node v; edge e;
node_array<node> copy_of(DT_FK);
forall_nodes(v,DT_FK) copy_of[v] = DT_FK1.new_node(rat_point(DT_FK[v]));
forall_nodes(v,DT_FK)
  forall_adj_edges(e,v)
    DT_FK1.new_edge(copy_of[v],copy_of[DT_FK.target(e)],DT_FK[e]);
DT_FK1.make_map();
if ( !Is_Delaunay_Triangulation(DT_FK1,NEAREST) ) I.write_table("$^*$");
```

We were very surprised when we first saw Table 8.4. We expected that the floating point computation would fail more often, not only when the full 52 bits are used to represent Cartesian coordinates of points. After all, the rational kernel resorts to integer arithmetic most of the time already for much smaller coordinate length and the difficult set of inputs.

We generated Table 8.5 to gain more insight[9]. It gives more detailed information for d ranging from 43 to 52. For our difficult inputs the floating point computation fails when d

[9] While writing this section, our work was very much guided by experiments. We had a theory of what floating point filters can do. Based on this theory we had certain expectations about the behavior of filters. We made experiments to confirm our intuition. In some cases the experiments contradicted our intuition and we had to revise the theory.

d	43	44	45	46	47	48	49	50	51	52
diff	C	C	C	F	F	F	F	F	F	F
easy	C	C	C	C	C	C	C	C	C	C

Table 8.5 Correctness of floating point computation: A detailed view for d ranging from 43 to 52. The second row corresponds to points on the unit circle and the last row corresponds to points in the unit square. A "C" indicates that the computation produced the correct result and a "F" indicates that a incorrect result was produced.

is 46 or larger and for our easy inputs it never fails. For $d < 45$ and both sets of inputs it produces the correct result. Our theoretical considerations give a guarantee only for $d < 10$.

In the remainder of this section we try to explain this discrepancy. We find the explanation interesting[10] but do not know at present whether it has any consequences for the design of floating point filters.

Let $D = 2^d$ and consider four points a, b, c, and d on the unit circle[11]. We use points a', b', c', and d' with integer Cartesian coordinates $\lfloor a_x D \rfloor$, $\lfloor a_y D \rfloor$, $\ldots$. The side of circle function is the sign of the determinant

$$\begin{vmatrix} 1 & 1 & 1 & 1 \\ a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_x^2 + a_y^2 & b_x^2 + b_y^2 & c_x^2 + c_y^2 & d_x^2 + d_y^2 \end{vmatrix}$$

as will be shown in Section 9.9. The value of this determinant is a homogeneous fourth degree polynomial $p(a_x, a_y, \ldots)$. We need to determine the sign of $p(a'_x, a'_y, \ldots)$. Let us relate $p(a_x, a_y, \ldots)$ and $p(a'_x, a'_y, \ldots)$.

We have

$$a'_x = \lfloor a_x D \rfloor = a_x D + \delta_{a_x},$$

where $-1 < \delta_{a_x} \le 0$, and analogous equalities hold for the other coordinates. Thus

$$\begin{aligned} p(a'_x, a'_y, \ldots) &= p(a_x D + \delta_{a_x}, a_y D + \delta_{a_y}, \ldots) \\ &= p(a_x D, a_y D, \ldots) \ + \ q_3(a_x D, \delta_{a_x}, a_y D, \delta_{a_y}, \ldots) \\ &\quad + \ q_2(a_x D, \delta_{a_x}, a_y D, \delta_{a_y}, \ldots) \ + \ q_1(a_x D, \delta_{a_x}, a_y D, \delta_{a_y}, \ldots) \\ &\quad + \ q_0(a_x D, \delta_{a_x}, a_y D, \delta_{a_y}, \ldots), \end{aligned}$$

where q_i has degree i in the $a_x D$, $a_y D$, $\ldots$ and degree $4 - i$ in the δ_{a_x}, δ_{a_y}, $\ldots$. Since the four points a, b, c, and d are co-circular, we have

$$p(a_x D, a_y D, \ldots) = D^4 p(a_x, a_y, \ldots) = 0.$$

[10] We all know from our physics classes that the important experiments are the ones that require a new explanation.
[11] In the final round of proof-reading we noticed that we use d with two meanings. In the sequel d is a point, except in the final sentence of the section.

Up to this point our argumentation was rigorous. From now on we give only plausibility arguments. Since the values $a_x D$ may be as large as D and since the values δ_{a_x} are smaller than one, the sign of $p(a'_x, a'_y, \ldots)$ is likely to be determined by the sign of q_3. Since q_3 is a third degree polynomial in the $a_x D$ we might expect its value to be about $f \cdot D^3$ for some constant f. The constant f is smaller than one but not much smaller. Expansion of the side of circle determinant shows that the coefficient of δ_{a_x} in q_3 is equal to

$$\begin{vmatrix} 1 & 1 & 1 \\ b_y D & c_y D & d_y D \\ (b_x^2 + b_y^2) \cdot D^2 & (c_x^2 + c_y^2) \cdot D^2 & (d_x^2 + d_y^2) \cdot D^2 \end{vmatrix} = D^3(c_y - a_y - b_y),$$

where we used the fact that $p_x^2 + p_y^2 = 1$ for a point p on the unit circle. We conclude that f has the same order as the y-coordinate of a random point on the unit circle and hence $f \approx 1/2$.

We evaluate $p(a'_x, a'_y, \ldots)$ with floating point arithmetic. By Theorem 1, the maximal error in the computation of p is $g \cdot D^4 \cdot 2^{-53}$ for some constant g; the actual error will be less. The argument in the proof of Lemma 5 shows that $g \le 2^8$. Thus we might expect that the floating point evaluation of $p(a'_x, a'_y, \ldots)$ gives the correct sign as long as $g \cdot D^4 \cdot 2^{-53} < f \cdot D^3$ or $d < 53 - \log g + \log f \approx 53 - 8 - 1 = 44$. This agrees quite well with Table 8.5.

8.7.5 *Conclusion*

We discussed the floating point filter in the rational kernel. We have seen that floating point filters give an exact implementation of geometric primitives at a reasonable cost.

Exercises for 8.7

1 The side of circle predicate determines for a four tuple (a, b, c, d) of points, whether d lies to the left, on, or to the right of the circle defined by the first three points. Derive a formula for the side of circle predicate for points given by Cartesian coordinates and for points given by homogeneous coordinates.

2 (Continuation) Derive a filter for both versions of the side of circle predicate according to Tables 8.1 and 8.2. Compare your results with the implementation of the side of circle predicate for *rat_points*.

3 Dynamic Filter: Derive a formula to compute err_E from $\tilde{E}$, err_A, and err_B for $E = A \cdot B$.

4 In ⟨*produce efficacy of filter table*⟩ we generated points by truncating the Cartesian coordinates to D bits, i.e., we generated *rat_points* by

```
rat_point(integer(p.xcoordD()*D),integer(p.ycoordD()*D),1).
```

What will change if we generate the points by

```
rat_point(integer(p.xcoordD()*D),integer(p.ycoordD()*D),D).
```

instead? Predict and then experiment.

5 Produce tables similar to Tables 8.3 and 8.4 for points that lie on a segment. Predict the outcome of the experiment before making it.

8.8 Safe Use of the Floating Point Kernel

The discussion of floating point filters in the previous section paves the way for a safe use of the floating point kernel. The following statement is trivial but nevertheless important.

It is safe to use the floating point kernel if it is guaranteed to give the correct result.

Lemma 4 gives a sufficient condition for the correctness of a floating point computation. If all arguments of an expression are integers, if the expression is a polynomial, i.e., uses only operations addition, subtraction, and multiplication, and if $B < 1$ then the evaluation with floating point arithmetic gives the correct sign of the expression. We have seen in Section 8.7.4 that the condition $B < 1$ is guaranteed if the arguments of the expression are sufficiently small; of course, the meaning of sufficiently small depends on the test. The following lemma gives information.

Lemma 5 *Assume that all points have integer Cartesian coordinates whose absolute value is less than* 2^L. *Then the floating point kernel correctly evaluates the compare function if* $L \le 50$, *correctly evaluates the orientation function if* $L \le 24$, *and correctly evaluates the side of circle function if* $L \le 11$.

Proof We give the proof for the side of circle function. Let a, b, c and d be points. We use ax and ay to denote the Cartesian coordinates of a and similarly for the other points.

The side of circle function is the sign of the determinant

$$\begin{vmatrix} 1 & 1 & 1 & 1 \\ ax & bx & cx & dx \\ ay & by & cy & dy \\ ax^2 + ay^2 & bx^2 + by^2 & cx^2 + cy^2 & dx^2 + dy^2 \end{vmatrix}$$

as will be shown in Section 9.9.

If a is equal to the origin the determinant above reduces to a 3×3 determinant. If a is not equal to the origin, we may shift a into the origin without changing the side of circle function. Shifting a into the origin replaces any point p by the point $O + (p - a)$.

This leads to the following program to compute the side of circle function. In this program we indicate the bit length of all intermediate results as comments.

```
int side_of_circle(const point& a, const point& b, const point& c,
                                                 const point& d)
{  // comments indicate bit lengths of values if coordinates have
   // at most L bits.
   double ax = a.xcoord();    // L bits
   double ay = a.ycoord();

   double bx = b.xcoord() - ax;  // L + 1 bits
   double by = b.ycoord() - ay;
   double bw = bx*bx + by*by;    // 2L + 3 bits

   double cx = c.xcoord() - ax;  // L + 1 bits
   double cy = c.ycoord() - ay;
   double cw = cx*cx + cy*cy;    // 2L + 3 bits
```

```
    double D1 = cy*bw - by*cw;  // 2L + 3 + L + 1 + 1 = 3L + 5 bits
    double D2 = bx*cw - cx*bw;  // 3L + 5 bits
    double D3 = by*cx - bx*cy;  // 2L + 3
    double dx = d.xcoord() - ax;  // L + 1 bits
    double dy = d.ycoord() - ay;
    double D  = D1*dx  + D2*dy + D3*(dx*dx + dy*dy);
                                // 3L + 5 + L + 1 + 2 = 4L + 8 bits
    if (D != 0)
       return (D > 0) ? 1 : -1;
    else
       return 0;
 }
```

The comments show that the maximal number of bits required for the determinant D is $4L + 8$. Thus D can be represented provided that $4L + 8 \leq 53$; observe that the mantissa of a double precision floating point number consists of 53 bits m_0, m_1, ..., m_{52}, of which the bit m_0 is not stored, since it is always 1 (except if the number is zero or underflow occurred).

□

The computation of, for example, Delaunay diagrams uses only the compare, orientation, and side of circle functions applied to input points and hence is safe as long as all input points have integer Cartesian coordinates whose absolute value is less than $2^{11} = 2048$.

If the coordinates of the inputs come from a larger range, it is frequently possible to round the input coordinates to a smaller precision without affecting the meaning of the computation, for example, if the coordinates come from a physical measurement whose precision is limited.

The following function *truncate* is useful in this situation. It takes a list *L0* of points and an integer *prec* and returns a list L of points. If all points in *L0* are equal to the origin, L is equal to *L0*. So assume otherwise and let M be the smallest power of two larger than the absolute value of all coordinates of all points in *L0*, and let $P = 2^{prec}$. For each point $p = (x, y)$ the point $(\lfloor (x/M) \cdot P \rfloor \cdot (M/P), \lfloor (y/M) \cdot P \rfloor \cdot (M/P))$ is added to L. Observe that x/M (and similarly y/M) is less than 1 and hence $(x/M) \cdot P$ is less than 2^{prec}. The multiplication by M/P (which is a power of two) moves the binary point for all points in the same way. Thus the theorem above applies to the modified points (with $L = prec$).

The implementation is simple. We first determine the maximum absolute value of any coordinate. If it is zero we are done. Otherwise, we set M to the smallest power of two larger than any absolute value. This is easily done using the functions *frexp* and *ldexp* from the math-library. Recall that *frexp*$(M, *exp)$ assigns to *exp* the exponent of the smallest power of two larger than M and that *ldexp*$(1, k)$ returns 2^k.

$\langle$*_truncate.c*$\rangle+\equiv$

```
list<point> truncate(const list<point>& L0, int prec)
{ double M = 0;
  point p;
  forall(p,L0)
    M = leda_max(M,leda_max(fabs(p.xcoord()),fabs(p.ycoord())));
    if ( M == 0 ) return L0;

    int exp;
    frexp(M,&exp);     // 2^(exp - 1) <= max < 2^exp
    M = ldexp(1,exp);  // round max to next power of two

    double C =     ldexp(1,prec - exp);  // P/M
    double C_inv = ldexp(1,exp - prec); // M/P

    list<point> L;
    forall(p,L0) L.append(point(floor(p.xcoord() * C)*C_inv,
                                floor(p.ycoord() * C)*C_inv));

    return L;
  }
```

There is also a version of truncate which operates on a list of *rat_points*. It simply converts every *rat_point* p to a point by calling *p.to_point*( ), then applies the function above to the resulting list of points, and finally converts every *point* q in the resulting list to a *rat_point* by calling the constructor *rat_point*(q).

8.9 A Glimpse at the Higher-Dimensional Kernel

The higher-dimensional kernel provides points, vectors, directions, hyperplanes, segments, lines, affine transformations, and operations connecting these types in d-dimensional Euclidean space for arbitrary finite d. Points have rational coordinates, hyperplanes have rational coefficients, and analogous statements hold for the other types. All geometric primitives are exact, since they are implemented using rational arithmetic. The computational basis for the kernel is provided by the classes integer, integer vector, and integer matrix discussed in Chapter 4. We refer the reader to [MMN$^+$98] for details. The higher-dimensional kernel is available as an LEP and was developed as part of the CGAL project.

8.10 History

The geometric part of LEDA evolved slowly and not without pain. We started with plane geometry in 1991. We introduced classes point, line, and segment and some algorithms operating on them, e.g., line segment intersection, Voronoi diagram construction, and convex hull construction. The programs provided in 1991 were not robust; on some inputs they failed by either delivering a wrong result or by crashing. The non-robustness of our original implementations was mainly due to three reasons:

- The programs were only designed to handle so-called non-degenerate inputs, e.g., the line segment intersection program assumed that no two input segments overlapped and the convex hull program assumed that the first three points were not collinear.

- Floating point arithmetic was used as the underlying arithmetic. We have seen in Section 8.6 that floating point arithmetic can lead to bizarre behavior of geometric objects.
- We had no checkers for geometric objects and hence were limited in our ability to test our algorithms.

Based on the bad experiences made by us and many others, we and others laid the theoretical foundations for correct and efficient implementations of geometric algorithms [FvW96, For96, CDR92, Yap93, Cla92, MN94b, BMS94a, BMS94b, BFS98, BRMS97, MNS+96, OLPT97, BR96, YD95, Sch, BEPP97].

Starting in 1994 we reimplemented the geometric classes and algorithms and simultaneously extended them considerably. We introduced the rational kernel with its built-in floating point filter, we redesigned all geometric algorithms and freed them from the assumption of non-degenerate inputs, and we added many new algorithms and checkers.

8.11 **LEDA and CGAL**

In 1997 the geometry effort of LEDA became part of project CGAL (= Constructing a Geometry Algorithms Library), a research project carried out by ETH Zürich, Freie Universität Berlin, INRIA Sophia Antipolis, Martin-Luther Universität Halle-Wittenberg, Max-Planck-Institut für Informatik and Universität des Saarlandes, RISC Linz, Tel-Aviv University, and Universiteit Utrecht, and funded by the European Union. The project was coordinated by Mark Overmars from Utrecht and ran for twenty-four months. The successor project is called GALIA and will be coordinated by the Max-Planck-Institut.

One of the goals of the projects is to build a comprehensive library for computational geometry called CGAL (Computational Geometry Algorithms Library). CGAL [CGA] goes much beyond LEDA geometry. The distinctive features of CGAL are:

- A geometry kernel [FGK+96] that can be instantiated with any number type. In LEDA we only have a floating point kernel and a rational kernel. It would be a non-trivial task to build a kernel based on the number type *real*. In CGAL this is easily possible.
- Geometric algorithms that are decoupled from the geometry kernel and can be used with any geometry kernel. Observe that LEDA's geometric algorithms are tied to the LEDA kernels and also to LEDA's graphs and data structures. CGAL achieves the new flexibility by the use of so-called *generic programming*. In this paradigm the kernel and the data structures are specified as template arguments of any geometric algorithm. The algorithm can then be instantiated with different kernels and data structures.
- A large variety of geometric data structures and algorithms which will go beyond what is offered by LEDA.

- An open architecture that makes it easy to import modules from other libraries.

The development of CGAL will not make LEDA geometry obsolete. The systems can be used side by side and both systems offer functionality which the other system does not have.

Bibliography

[ABDP97] F. Avnaim, J.-D. Boissonnat, O. Devillers, and F.P. Preparata. Evaluating signs of determinants with floating point arithmetic. *Algorithmica*, 17(2):111–132, 1997.

[BEPP97] Hervé Brönnimann, Ioannis Emiris, Victor Pan, and Sylvain Pion. Computing exact geometric predicates using modular arithmetic with single precision. In *Proceedings of the Symp. on Computational Geometry*, pages 174–182, 1997.

[BFS98] C. Burnikel, S. Funke, and M. Seel. Exact arithmetic using cascaded computation. In *Proc. of the ACM Symp. on Computational Geometry*, 1998.

[BMS94a] Ch. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. SODA 94*, pages 16–23, 1994.

[BMS94b] Ch. Burnikel, K. Mehlhorn, and St. Schirra. How to compute the Voronoi diagram of line segments: Theoretical and experimental results. In *LNCS*, volume 855, pages 227–239. Springer-Verlag Berlin/New York, 1994. Proceedings of ESA'94.

[BR96] Raja P. K. Banerjee and Jarek R. Rossignac. Topologically exact evaluation of polyhedra defined in CSG with loose primitives. *Computer Graphics Forum*, 15(4):205–217, 1996. ISSN 0167-7055.

[BRMS97] Ch. Burnikel, R.Fleischer, K. Mehlhorn, and S. Schirra. A strong and easily computable separation bound for arithmetic expressions involving square roots(ps). In *Proc. SODA 97*, pages 702–709, 1997.

[Bur96] Chr. Burnikel. *Exact Computation of Voronoi Diagrams and Line Segment Intersections*. PhD thesis, Max-Planck-Insitut für Informatik, Saarbrücken, April 1996.

[CDR92] J. Canny, B. Donald, and G. Ressler. A rational rotation method for robust geometric algorithms. In ACM-SIGACT ACM-SIGGRAPH, editor, *Proceedings of the 8th Annual Symposium on Computational Geometry (SCG '92)*, pages 251–260, Berlin, FRG, June 1992. ACM Press.

[CGA] CGAL (computational geometry algorithms library). `http://www.cs.ruu.nl/CGAL`.

[Cla92] K. L. Clarkson. Safe and effective determinant evaluation. In *31st IEEE FOCS*, pages 387–395, 1992.

[DP96] O. Devillers and F. Preparata. A probabilistic analysis of the power of arithmetic filters. Technical Report CS-96-27, Brown University - Department of Computer Science, September 1996.

[FGK$^+$96] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL Kernel: A basis for geometric computation. In *Workshop on Applied Computational Geometry (WACG96)*, LNCS, 1996.

[FM91] S. Fortune and V. Milenkovic. Numerical stability of algorithms for line arrangements. In ACM-SIGACT ACM-SIGGRAPH, editor, *Proceedings of the 7th Annual Symposium on Computational Geometry (SCG '91)*, pages 334–341, North Conway, NH, USA, June 1991.

ACM Press.

[For96] Fortune. Robustness issues in geometric algorithms. In *WACG: 1st Workshop on Applied Computational Geometry: Towards Geometric Engineering, WACG*. LNCS, 1996.

[Fun97] S. Funke. Exact arithmetic using cascaded computation. Master's thesis, Max-Planck-Insitut für Informatik, Saarbrücken, April 1997.

[FvW96] S. Fortune and C. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. Graph.*, 15:223–248, 1996.

[Gol90] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1990.

[Gol91] David Goldberg. Corrigendum: "What every computer scientist should know about floating-point arithmetic". *ACM Computing Surveys*, 23(3):413–413, September 1991.

[GSS89] L. Guibas, D. Salesin, and J. Stolfi. Epsilon geometry: building robust algorithms from imprecise computations. In *Proceedings of the fifth annual Symposium on Computational Geometry: Saarbrucken, West Germany, June 5–7, 1989*, pages 208–217, New York, NY 10036, USA, 1989. ACM Press.

[GSS93] Leonidas J. Guibas, David Salesin, and Jorge Stolfi. Constructing strongly convex approximate hulls with inaccurate primitives. *Algorithmica*, 9:534–560, 1993.

[IEE87] IEEE standard 754-1985 for binary floating-point arithmetic, 1987.

[KLN91] Michael Karasick, Derek Lieber, and Lee R. Nackman. Efficient Delaunay Triangulation Using Rational Arithmetic. *ACM Transactions on Graphics*, 10(1):71–91, January 1991.

[LM90] Z. Li and V. Milenkovic. Constructing strongly convex hulls using exact or rounded arithmetic. In ACM-SIGACT ACM-SIGGRAPH, editor, *Proceedings of the 6th Annual Symposium on Computational Geometry (SCG '90)*, pages 235–243, Berkeley, CA, June 1990. ACM Press.

[Mil88] V. J. Milenkovic. *Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic*. PhD thesis, Carnegie Mellon University, July 1988.

[Mil89a] V. Milenkovic. Calculating approximate curve arrangements using rounded arithmetic. In Kurt Mehlhorn, editor, *Proceedings of the 5th Annual Symposium on Computational Geometry (SCG '89)*, pages 197–207, Saarbrücken, FRG, June 1989. ACM Press.

[Mil89b] Victor Milenkovic. Double precision geometry: A general technique for calculating line and segment intersections using rounded arithmetic. In *30th Annual Symposium on Foundations of Computer Science*, pages 500–505, Research Triangle Park, North Carolina, 30 October–1 November 1989. IEEE.

[MMN$^+$98] K. Mehlhorn, Müller, S. Näher, S. Schirra, M. Seel, C. Uhrig, and J. Ziegler. A computational basis for higher-dimensional computational geometry and its applications. *Computational Geometry: Theory and Applications*, 10:289–303, 1998. http://www.mpi-sb.mpg.de/˜seel.

[MN94a] K. Mehlhorn and S. Näher. Implementation of a sweep line algorithm for the straight line segment intersection problem. Technical Report MPI-I-94-160, Max-Planck-Institut für Informatik,Saarbrücken, 1994.

[MN94b] K. Mehlhorn and S. Näher. The implementation of geometric algorithms. In *13th World Computer Congress IFIP94*, volume 1, pages 223–231. Elsevier Science B.V. North-Holland, Amsterdam, 1994.

[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.

[OLPT97] O.Devillers, G. Liotta, F.P. Preparata, and R. Tamassia. Checking the convexity of polytopes and the planarity of subdivisions. Technical report, Center for Geometric Computing, Department of Computer Science, Brown University, 1997.

[Sch] S. Schirra. Robustness and precision issues in geometric computation. to appear, prelimiary version avaiable as MPI report.

[She97] J.R. Shewchuk. Adaptive precision flaoting-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18:305–363, 1997.

[Yap93] C.K. Yap. Towards Exact Geometric Computation. In *CCCG5*, pages 405–419, 1993.

[YD95] C.K. Yap and T. Dube. The Exact Computation Paradigm. In *Computing in Euclidean Geometry II*. World Scientific Press, 1995.

Index