

Lecture 5

A First Geometric Kernel

This lecture will be quite different from the preceding one. There will be no definitions and theorem; this lecture will be about software design. We will address two issues: how to package basic geometric objects into a geometric kernel and how to make use of approximate arithmetic in an exact kernel. We will also study the efficiency of such a kernel. We will see that generic programming techniques support a clean separation between algorithms and basic objects through the introduction of kernel without sacrificing efficiency.

5.1 A Kernel

A *kernel* comprises basic geometric objects and operations on these objects. It reveals nothing about the representation of the objects. A *model* of the kernel is a concrete implementation of the objects in the kernel. Algorithms are formulated in terms of the kernel and can be instantiated with any model of the kernel.

The most basic kernel offers only one kind of object, namely points in the plane, and a small collection of operations on them, e.g., the orientation function of three points, lexicographic comparison of points, and access to the Cartesian coordinates of a point. Depending on the programming language, it may also have to provide additional functions. For example, C++ requires constructors and an assignment operator. In pseudo-code (we will see the C++ formulation in the next section) we might write:

```
concept basic_kernel {
    object:    point_2d;
    operations: NT x_coordinate();
               NT y_coordinate();
               ops required by the language

               int orientation(point_2d,point_2d,point_2d);
}
```

In programming language parlance (TODO: is this correct, or is it only C++ parlance), a kernel is a *concept*. A concept is a collection of objects, operation on these objects, and a set of requirements. In our example, the requirements are that the orientation-function actually computes the orientation of its arguments and that the access function return the Cartesian coordinates. We might also require that these functions run in constant time.

You have seen the notion of a concept in your math-courses. For example, a vector space is a concept. It comprises a ring F (another concept), a set V , a special element $0 \in V$, and two operations $+$ and \cdot . Addition realizes a commutative group with neutral element 0 . And multiplication by a scalar takes a field element k and a vector $v \in V$ and yields a vector $k \cdot v$ such that $0 \cdot v = 0$, $1 \cdot v = v$, $(k_1 + k_2) \cdot v = k_1 \cdot v + k_2 \cdot v$, $(k_1 k_2) \cdot v = k_1 \cdot (k_2 \cdot v)$, $k \cdot (v + w) = k \cdot v + k \cdot w$. A model of this concept is any concrete vector space, e.g., $F = \mathbb{R}$ and $V = \mathbb{R}^d$. Addition of vectors and multiplication by a scalar is component-wise. The notions of linear-independence and basis are defined for vector-spaces. The theorem that all bases of a vector space have the same cardinality is proved generally for vector spaces. Of course, the theorem then holds for any concrete vector space.

The role of a concept in programming is exactly the same, except that we do not prove theorems but write algorithms. We write algorithms in terms of concepts and the algorithm will then run for any model of the concept. For example, we could formulate our convex hull algorithm from Lecture ?? as follows:

```
algorithm convex_hull based on concept linear_kernel {
// the algorithm as in Lecture XXX using the names in the kernel;
point_2d p;    // declaration of a point p
...
}
```

5.2 Concrete Kernels

We discuss models of the basic kernel. We have many choices. We may present points by their Cartesian coordinates or by their homogeneous coordinates or as the intersection of two lines or We discuss the first choice and ask the reader to work out the second choice in the exercises.

In the Cartesian model, a point has two data members x and y , the access functions `x_coord` and `y_coord` return x and y , respectively, and orientation is implemented by formula XXX from Lecture ?. The Cartesian coordinates come from a number type `NT` which supports exact computations of signs. We have seen three such types in Lecture ?: arbitrary precision integers, rational numbers, and arbitrary precision floating point numbers without rounding.

```
model Cartesian_Points of concept basic_kernel {
struct point_2d { NT x,y;
real x_coord() { return x; }
real y_coord() { return y; }
}
int orientation(point_2d p, point_2d q, point_2d r){ return sign ..... ; }
}
```

Exercise 0.1: Formulate a model of the basic kernel, in which points are represented by their homogeneous coordinates. \diamond

An Unusual Kernel: To see the flexibility of the approach, we give another example. The example may seem weird, but is actually inspired by reduction of Delaunay triangulations to lower convex hulls in one higher dimension. We will see this reduction in Lecture ?.

We are interested only in points on the parabola $y = x^2$. So a point has a single data member, its x -coordinate. The y -coordinate is computed as the square of the x -coordinate. Orientation can be computed

simpler than in the general case. Assume that p lies left of q . Then p, q, r form a right turn, if r lies between p and q .

```
model parabola_points of concept basic_kernel {
struct point_2d{ NT x;
int x_coord(){ return x; }
int y_coord(){ return x^2; }
}
int orientation(point_2d p, point_2d q, point_2d r)
{ if (p.x_coord() < q.x_coord() < r.x_coord) return -1;
  ....
}
}
```

5.3 C++ Formulation*

We use pseudocode to introduce the notions concept and model. In C++ , the formulation is as follows.

```
//! a simple cartesian kernel for points (and operations on them)
template < class NT >
class Cartesian_kernel {

public:

    // GEOMETRIC TYPES (ref-counted ones would be better)

    //! Type of Point (with cartesian x- and y-coordinates)
    class Point {

    public:

        //! default constructor constructs origin
        Point() :
            m_x(0), m_y(0) // assumes that NT is constructible from SmallIntConstant
        {}

        //! constructor from two given coordinates
        Point(NT x, NT y) :
            m_x(x), m_y(y) // assumes that NT is copy-constructible
        {}

        //! returns x-coordinate of point
        NT x() const { return m_x; }

        //! returns y-coordinate of point
```

```

    NT y() const { return m_y; }

private:

    //! x-coordinate of point
    NT m_x;

    //! y-coordinate of point
    NT m_y;

};

// GEOMETRIC OPERATIONS (as functions)

int orientation(const Point& p, const Point& q, const Point& r) const {

    NT det = (q.x() - p.x()) * (r.y() - p.y()) -
        (q.y() - p.y()) * (r.x() - p.x());

    if (det < 0) { // assumes that NT has operator<(int)
        return -1;
    } else if (det > 0) { // assumes that NT has operator>(int)
        return 1;
    }

    return 0;

}

};

```

C++purists would probably criticize the code above on two accounts. Identifiers for template parameters should not be used as types. It is advised to use `NT_` as parameter and to declare a public type `typedef NT_ NT` subsequently. It is also recommended to implement predicates and constructions 'functors' and to use an enumeration type instead of 'int' as the result type of the orientation function.

```

//! a simple kernel for points on a parabola (and operations on them)
template < class NT >
class Parabolic_kernel {

public:

    // GEOMETRIC TYPES (ref-counted ones would be encouraged)

    class Point {

```

```

public:

    //! default constructor constructs origin
    Point() :
        m_x(0) // assumes that NT is constructible from SmallIntConstant
    {}

    //! constructor from one given coordinate
    Point(NT x) :
        m_x(x) // assumes that NT is copy-constructible
    {}

    //! returns x-coordinate of point
    NT x() const { return m_x; }

    //! returns y-coordinate of point
    NT y() const { return m_x * m_x; }

private:

    //! x-coordinate of point
    NT m_x;

};

// GEOMETRIC OPERATIONS (as functions)

int orientation(const Point& p, const Point& q, const Point& r) const {
    std::cerr << "Parabolic Orientation not complete!" << std::endl;

    if (p.x() < q.x()) { // assumes that NT has operator<(int)
        if (q.x() < r.x()) {
            return -1;
        }
    }
    // else
    return 0;
}
};

```

Next comes the convex hull algorithm. We give only a stub.

```

//! class stub for convex hull
template < class Kernel >
class Convex_hull {

```

```

public:

    //! the kernel's point type
    typedef typename Kernel::Point Point;

    template < class InputIterator, class OutputIterator >
    OutputIterator operator()(InputIterator begin, InputIterator end,
                             OutputIterator result) {

        /* CONVEX HULL algorithm for points in [begin,end) */

        InputIterator it = begin;

        while (it != end) {

            Point p = *it;

            // do process p

            // next
            it++;
        }

        return result;
    }
};

```

and finally the main program.

```

#include <iostream>
#include <list>

#include "KMCartesian_kernel.h"
#include "KMParabolic_kernel.h"
#include "KMConvex_hull.h"

template < class NT >
void cartesian() {

    typedef Cartesian_kernel< NT > Kernel;

    typedef typename Kernel::Point Point;

```

```

// construct some points
Point o;
Point p1(-1,1); // requires NT to be ConstructibleFromSmallInt
Point p2(-5,5);
Point pl(-2,3);
Point pr(-4,1);

// orientation of points
Kernel kernel;
std::cout << "Orientation(o,p1,p2) = " << kernel.orientation(o,p1,p2) << std::endl;
std::cout << "Orientation(o,p1,pl) = " << kernel.orientation(o,p1,pl) << std::endl;
std::cout << "Orientation(o,p1,pr) = " << kernel.orientation(o,p1,pr) << std::endl;

std::list< Point > input;
input.push_back(o);
input.push_back(p1);
input.push_back(p2);
input.push_back(pl);
input.push_back(pr);

// Convex hull
typedef Convex_hull< Kernel > CH;

std::list< Point > hull;
CH ch;
ch(input.begin(), input.end(), std::back_inserter(hull));

}

template < class NT >
void parabolic() {

    typedef Parabolic_kernel< NT > Kernel;

    typedef typename Kernel::Point Point;

    // construct some points
    Point o;
    Point p1(1); // requires NT to be ConstructibleFromSmallInt
    Point p2(5);
    Point pl(2);
    Point pr(4);

    // orientation of points

```

```

Kernel kernel;
std::cout << "Orientation(o,p1,p2) = " << kernel.orientation(o,p1,p2) << std::endl;
std::cout << "Orientation(o,p1,p1) = " << kernel.orientation(o,p1,p1) << std::endl;
std::cout << "Orientation(o,p1,pr) = " << kernel.orientation(o,p1,pr) << std::endl;

std::list< Point > input;
input.push_back(o);
input.push_back(p1);
input.push_back(p2);
input.push_back(pl);
input.push_back(pr);

// Convex hull
typedef Convex_hull< Kernel > CH;

std::list< Point > hull;
CH ch;
ch(input.begin(), input.end(), std::back_inserter(hull));

}

int main() {

    std::cout << "CARTESIAN with 'int'" << std::endl;
    cartesian< int >();
    std::cout << std::endl;

    std::cout << "CARTESIAN with 'unsigned int' - evil, because of '-1' in input" << std::endl;
    cartesian< unsigned int >();
    std::cout << std::endl;

    std::cout << "CARTESIAN with 'double'" << std::endl;
    cartesian< double >();
    std::cout << std::endl;

    std::cout << "PARABOLIC with 'int'" << std::endl;
    parabolic< int >();
    std::cout << std::endl;

    std::cout << "PARABOLIC with 'double'" << std::endl;
    parabolic< double >();
    std::cout << std::endl;

}

```


Exercise 0.2: Redo the above in the programming language of your choice. ◇

5.4 A Floating Point Filter

Exact arithmetic is much slower than hardware floating point arithmetic. However, floating point arithmetic is only approximate and we have seen in Lecture ?? that a naive use of floating point arithmetic can lead to disaster. In Lecture ?? we learned how to estimate the error errors in floating point computations. We will now put this knowledge to use. We will obtain an exact kernel that is also efficient. We will give experimental evidence in the next section and theoretical analysis in Lecture ??.

The idea is to preface the evaluation of any expression (here the expression defining the orientation predicate) by an evaluation with floating point arithmetic. We also compute a bound on the roundoff error. If the absolute value of the float value is larger than the bound on the roundoff error, we return the sign of the float value. Otherwise, we evaluate the expression with exact arithmetic. This scheme is called a *floating point filter*. The following code realizes this strategy for the orientation predicate.

```
int orientation(point_2d p, point_2d q, point_2d r){
NT px = p.xcoord(), py = p.ycoord(), qx = q.xcoord(), .... ;
// evaluation in floating point arithmetic
float pxd = fl(px), pyd = fl(py), qxd = fl(qx), .....;
float Etilde = (qxd - pxd)*(ryd - pyd) - (qyd - pyd)*(rxd - pxd);
float apxd = abs(pxd), apyd = abs(pyd), aqxd = abs(qxd), ....;
float mes = (aqxd + apxd)*(aryd + apyd) + (aqyd + apyd)*(arxd + apxd);
if ( abs(Etilde) > 7 * uu * mes ) return (sign Etilde);
// exact evaluation
NT E = (qx - px)*(ry - py) - (qy - py)*(rx - px);
return sign E;
}
```

According to Lemma ??, this implementation is correct.

Exercise 0.3: Formulate a floating point filter for points represented by their homogeneous coordinates. ◇

5.5 Performance of the Floating Point Filter

We study the performance of the floating point filter under two aspects. How often is it necessary to resort to exact computation and how much do we save in running time? This section is based on [7, Section 9.7.4].

[[TODO: repeat the experiments and make them available on the companion page of the book.]]

Table 5.1 sheds light on the first question. The following experiment was performed. First, a set S of n random points with 52 bit Cartesian coordinates either on the unit circle or in the unit square was generated. A random point in the unit square is generated by choosing its coordinates as follows: Generate a random integer $i \in [0, 2^{52} - 1..]$ and then set the coordinate to $i/2^{52}$. The generation of points on the unit circle is the topic of Section 5.6. Then the Cartesian coordinates were truncated to d bits for different values of d , i.e., a point p with Cartesian coordinates (p_x, p_y) was turned into a point p' with Cartesian coordinates $(\lfloor 2^d p_x \rfloor, \lfloor 2^d p_y \rfloor)$. Let S' be the resulting set of points. The Delaunay triangulation of S' was constructed . Explain Al

d	N	Compare			Orientation			Side of circle		
		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 5.1: 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 5.1 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. The following exper-

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 5.2: 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.

iment continues the preceding experiment. The computation of the Delaunay diagram was performed in three different ways:

- naive use of floating point arithmetic: the truncated Cartesian coordinates were stored as double precision floating point numbers and Delaunay diagram algorithm was run with double precision arithmetic.
- exact integer arithmetic with a floating point filter.
- exact integer arithmetic without the floating point filter turned off.

. Table 5.2 summarizes the outcome. Let us first look at 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 5.1) 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 kernel is not guaranteed to give the correct result. In fact, it produced an incorrect result in one of the experiments (indicated by a *).

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

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 5.3: 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.

the exact points is faster than the computation with the rounded points. The explanation can be found in Table 5.1. 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. skip

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.

[[The remainder of this section is obsolete. The discussion is superseded by the the work on controlled perturbation. We should add an experiment where the points are all on a small segment of the circle.]]

We were very surprised when we first saw Table 5.2. 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.

Exercise 0.4: Repeat the experiments of this section for points that lie on a segment. Predict the outcome of the experiment before making it. ◇

We generated Table 5.3 to gain more insight¹. It gives more detailed information for d ranging from 43 to 52. For our difficult inputs the floating point computation fails when d is 46 or larger and for our

¹While 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.

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 interesting² 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 circle³. We use points a', b', c' , and d' with integer Cartesian coordinates $\lfloor a_x D \rfloor, \lfloor a_y D \rfloor, \dots$. 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 ???. The value of this determinant is a homogeneous fourth degree polynomial $p(a_x, a_y, \dots)$. We need to determine the sign of $p(a'_x, a'_y, \dots)$. Let us relate $p(a_x, a_y, \dots)$ and $p(a'_x, a'_y, \dots)$.

We have

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

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

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

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

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

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 $p(a'_x, a'_y, \dots)$ 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 $p(a'_x, a'_y, \dots)$ with floating point arithmetic. By Theorem ??, 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 ?? shows that $g \leq 2^8$. Thus we might expect that the floating point evaluation of $p(a'_x, a'_y, \dots)$ 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 5.3.

²We all know from our physics classes that the important experiments are the ones that require a new explanation.

³In 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.

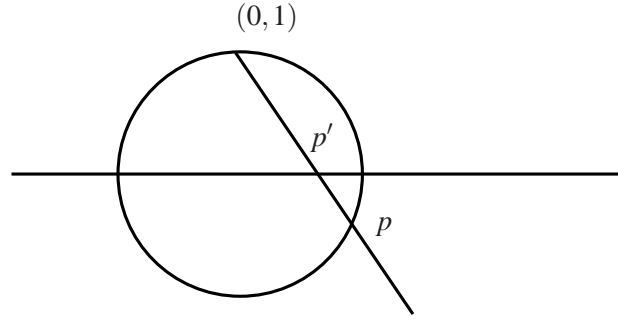


Figure 5.1: Point $p = (p_x, p_y)$ lies on the unit circle, point $p' = (a, 0)$ lies on the x -axis, and points $(0, 1)$, p , and p' lie on a common line.

5.6 Points on a Circle

A point on the unit circle has Cartesian coordinates $(\cos \alpha, \sin \alpha)$, where $0 \leq \alpha < 2\pi$. In general, sines and cosines are non-rational numbers, e.g., $\cos \pi/4 = \sqrt{2}/2$. In this section, we will show how to find a dense set of points with rational Cartesian coordinates on the unit circle. For any α and any $\varepsilon > 0$, we will show how to find a triple (a, b, w) of integral homogeneous coordinates such that

$$a^2 + b^2 = w^2 \quad \text{and} \quad |\alpha - \alpha'| \leq \varepsilon \quad \text{where} \quad \cos \alpha' = a/w \quad \text{and} \quad \sin \alpha' = b/w.$$

A triple (a, b, w) of integers with $a^2 + b^2 = w^2$ is called a Pythagorean triple.

LEMMA 1. *For any rational point $p = (p_x, p_y)$ on the unit circle there is a rational a and integers n and m such that*

$$(p_x, p_y) = \left(\frac{2a}{a^2 + 1}, \frac{a^2 - 1}{a^2 + 1} \right) = \left(\frac{2mn}{n^2 + m^2}, \frac{n^2 - m^2}{n^2 + m^2} \right)$$

Proof. Stereographic projection is a one-to-one correspondence between the points on the unit circle and the points on x -axis, see Figure 5.1. If $p = (p_x, p_y)$ lies on the unit circle, $p' = (a, 0)$ lies on the x -axis, and $(0, 1)$, p and p' lie on a common line, then

$$a = \frac{p_x}{1 - p_y} \quad \text{and} \quad p_x = \frac{2a}{a^2 + 1}, \quad p_y = \frac{a^2 - 1}{a^2 + 1}$$

as a simple computation shows. Thus, if p has rational coordinates, p' has rational coordinates, and if p' has rational coordinates, p has rational coordinates. We conclude that every rational point on the unit circle has coordinates $p_x = 2a/(a^2 + 1)$ and $p_y = (a^2 - 1)/(a^2 + 1)$ for some rational a . Let $a = n/m$. Then

$$p_x = \frac{2(n/m)}{(n/m)^2 + 1} = \frac{2nm}{n^2 + m^2} \quad \text{and} \quad p_y = \frac{a^2 - 1}{a^2 + 1} = \frac{(n/m)^2 - 1}{(n/m)^2 + 1} = \frac{n^2 - m^2}{n^2 + m^2}.$$

□

Exercise 0.5: Why can there be no Pythagorean triple (a, b, c) with a and b odd?

◇

If we would not insist on a being rational, we could simply choose a such that

$$\cos \alpha = \frac{2a}{a^2 + 1} \quad \text{or} \quad a = \frac{1}{\cos \alpha} \pm \sqrt{\frac{1}{\cos^2 \alpha} - 1}.$$

The two choices for a correspond to the two possible values for $\sin \alpha$. However, we want a to be rational. An obvious way to obtain a rational approximation with error at most 2^{-s} is as follows. We compute a floating point approximation \tilde{a} of a with error at most 2^{-s} as shown in Section ??; \tilde{a} is the desired rational approximation. The fraction obtained in this way has a numerator and denominator of s bits.

One can usually obtain better approximations with fewer bits as we discuss next. The less mathematically inclined reader may proceed directly to the end of the section. We first show that there is always a good rational approximation with small denominator and then show how to compute such an approximation.

THEOREM 2 (Dirichlet, 1842). *For any real x and any positive ε there is a rational number p/q such that*

$$q \leq \frac{1}{\varepsilon} \quad \text{and} \quad \left| x - \frac{p}{q} \right| < \frac{\varepsilon}{q}.$$

Proof. If $\varepsilon \geq 1$, we simply take $p = \lfloor x \rfloor$ and $q = 1$. So assume $\varepsilon < 1$. Let $M = \lfloor 1/\varepsilon \rfloor$ and consider the numbers. For each i , $0 \leq i \leq M$, let f_i be the fractional part of ix , i.e., $f_i = ix - \lfloor ix \rfloor$. The fractional parts lie between 0 and 1 and hence there are distinct i and j such that $|f_j - f_i| \leq 1/M$. Assume $j > i$. Then

$$|(j-i)x - (\lfloor jx \rfloor - \lfloor ix \rfloor)| = |f_j - f_i| \leq \frac{1}{M}$$

and hence

$$\left| x - \frac{\lfloor jx \rfloor - \lfloor ix \rfloor}{j-i} \right| \leq \frac{1}{(j-i)M} \leq \frac{\varepsilon}{j-i}.$$

Set $q = j - i$ and $p = \lfloor jx \rfloor - \lfloor ix \rfloor$. □

The standard technique for approximating a real by a rational is to compute its continued fraction expansion. For an $x \in \mathbb{R}_{\geq 0}$, define a sequence x_0, x_1, x_2, \dots of reals and a sequence a_0, a_1, a_2, \dots of integers as follows.

$$\begin{array}{ll} x_0 = x & a_0 = \lfloor x_0 \rfloor \\ x_1 = \frac{1}{x_0 - a_0} & a_1 = \lfloor x_1 \rfloor \\ x_2 = \frac{1}{x_1 - a_1} & a_2 = \lfloor x_2 \rfloor \\ \vdots & \vdots \end{array}$$

If some x_i is integral, the sequence ends with a_i . Otherwise, the sequence is infinite. Clearly, $x_i - a_i < 1$ for all i . If $a_i = x_i$, the sequence ends, if $a_i < x_i$, $x_{i+1} > 1$ and hence $a_{i+1} \geq 1$. We call $[a_0; a_1, a_2, \dots]$ the *continued fraction expansion* of x . We will next derive some properties of this expansion. Observe first that $x_i = a_i + 1/x_{i+1}$ whenever x_{i+1} is defined and hence

$$x = x_0 = a_0 + \frac{1}{x_1} = a_0 + \frac{1}{a_1 + \frac{1}{x_2}} = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{x_3}}} = \dots$$

A finite continued fraction defines a rational number. The converse is also true as we will see below. The continued fraction $[a_0; a_1, \dots, a_n]$ is a rational number. We call it the n -th convergent of x . The convergents of a continued fraction have many nice properties.

LEMMA 3. *Let $x \in \mathbb{R}_{\geq 0}$ and let $[a_0; a_1, a_2, \dots]$ be the continued fraction expansion of x . Define $p_{-2} = 0$, $q_{-2} = 1$, $p_{-1} = 1$, $q_{-1} = 0$, and*

$$p_n = a_n p_{n-1} + p_{n-2} \quad \text{and} \quad q_n = a_n q_{n-1} + q_{n-2} \quad \text{for } n \geq 0.$$

Then

1. $\frac{p_n}{q_n} = [a_0; a_1, \dots, a_n]$ is the n -th convergent of x .
2. $p_n q_{n-1} - p_{n-1} q_n = (-1)^{n+1}$ for $n \geq -1$.
3. $\left| \frac{p_{n+1}}{q_{n+1}} - \frac{p_n}{q_n} \right| = \frac{1}{q_n q_{n+1}}$ for $n \geq 0$.
4. $q_n \geq (3/2)^{n-1}$ for $n \geq 0$.
5. $\frac{p_{-2}}{q_{-2}} < \frac{p_0}{q_0} < \frac{p_2}{q_2} < \dots \leq x \leq \dots < \frac{p_3}{q_3} < \frac{p_1}{q_1} < \frac{p_{-1}}{q_{-1}}$.
6. The $n+2$ -th convergent is closer to the $n+1$ -th convergent than to the n -th convergent.
7. $x - p_n/q_n$ is strictly decreasing in n .

Proof. Let z be variable. Define

$$M_n(z) = [a_0; a_1, \dots, a_n + z].$$

We will show that

$$M_n(z) = \frac{p_n + p_{n-1}z}{q_n + q_{n-1}z}$$

by induction on n . For $n = 0$, we have

$$M_0(z) = a_0 + z = \frac{p_0 + p_{-1}z}{q_0 + q_{-1}z}.$$

For $n+1 \geq 1$, we have

$$M_{n+1}(z) = M_n\left(\frac{1}{a_{n+1} + z}\right) = \frac{p_n + p_{n-1} \frac{1}{a_{n+1} + z}}{q_n + q_{n-1} \frac{1}{a_{n+1} + z}} = \frac{a_{n+1} p_n + p_{n-1} + p_n z}{a_{n+1} q_n + q_{n-1} + q_n z} = \frac{p_{n+1} + p_n z}{q_{n+1} + q_n z}.$$

$M_n(0)$ is the n -th convergent of x . Thus $[a_0; a_1, \dots, a_n] = p_n/q_n$. This proves (1).

We turn to (2). Observe first that $p_{-1}q_{-2} - p_{-2}q_{-1} = 1 = (-1)^0$. For $n \geq 0$, we have

$$\begin{aligned} p_n q_{n-1} - p_{n-1} q_n &= (a_n p_{n-1} + p_{n-2}) q_{n-1} - p_{n-1} (a_n q_{n-1} + q_{n-2}) \\ &= p_{n-2} q_{n-1} - p_{n-1} q_{n-2} = (-1) \cdot (-1)^n = (-1)^{n+1}. \end{aligned}$$

(3) follows from a simple calculation.

$$\left| \frac{p_{n+1}}{q_{n+1}} - \frac{p_n}{q_n} \right| = \frac{|p_{n+1} q_n - p_n q_{n+1}|}{q_n q_{n+1}} = \frac{1}{q_n q_{n+1}}.$$

(4) is a simple induction. $q_0 = 1 \geq (3/2)^{-1}$ and $q_1 = a_1 \geq (3/2)^0$ and for $n \geq 2$,

$$q_n = a_n q_{n-1} + q_{n-2} \geq (3/2)^{n-2} + (3/2)^{n-3} = (3/2)^{n-3} (3/2 + 1) = (3/2)^{n-3} 5/2 \geq (3/2)^{n-1}.$$

We turn to (5). Assume inductively that $p_n/q_n \leq x \leq p_{n-1}/q_{n-1}$ for even n . This is certainly true for $n = -2$. $M_n(z)$ is an increasing function of z , $M_n(0) = p_n/q_n$, $M_n(\infty) = p_{n-1}/q_{n-1}$, and $M_n(1/x_{n+1}) = x$. Now $a_{n+1} = \lfloor x_{n+1} \rfloor$ and hence $1/a_{n+1} \geq 1/x_{n+1}$. Thus that $x \leq p_{n+1}/q_{n+1} = M_n(1/a_{n+1}) < p_{n-1}/q_{n-1}$. A similar argument shows $p_n/q_n < p_{n+2}/q_{n+2} \leq x$.

For (6), we consider the case of even n . We have

$$\frac{p_{n+2}}{q_{n+2}} - \frac{p_n}{q_n} = \frac{p_{n+1}}{q_{n+1}} - \frac{p_n}{q_n} - \left(\frac{p_{n+1}}{q_{n+1}} - \frac{p_{n+2}}{q_{n+2}} \right) = \frac{1}{q_n q_{n+1}} - \frac{1}{q_{n+1} q_{n+2}} > \frac{1}{q_{n+1} q_{n+2}} = \frac{p_{n+1}}{q_{n+1}} - \frac{p_{n+2}}{q_{n+2}},$$

where the inequality follows from $q_{n+2} \geq q_{n+1} + q_n > 2q_n$. The proof for odd n is similar.

(7) is an easy consequence of (6). Consider an even n . Then $p_n/q_n < p_{n+2}/q_{n+2} \leq x \leq p_{n+1}/q_{n+1}$ and

$$\frac{p_{n+1}}{q_{n+1}} - x \leq \frac{p_{n+1}}{q_{n+1}} - \frac{p_{n+2}}{q_{n+2}} \leq \frac{p_{n+2}}{q_{n+2}} - \frac{p_n}{q_n} \leq x - \frac{p_n}{q_n}.$$

□

The convergents p_n/q_n are in lowest terms, because otherwise we could not have $p_n q_{n-1} - p_{n-1} q_n = (-1)^{n+1}$. The even convergents converge to x from below and the odd convergents converge to x from above. We have

$$\frac{p_n}{q_n} = \frac{p_0}{q_0} + \sum_{1 \leq i \leq n} \frac{p_i}{q_i} - \frac{p_{i-1}}{q_{i-1}} = a_0 + \sum_{1 \leq i \leq n} \frac{(-1)^{i+1}}{q_i q_{i-1}}.$$

Thus $x = a_0 + \sum_{i \geq 1} (-1)^{i+1} / (q_i q_{i-1})$.

LEMMA 4. Let $x \in \mathbb{R}_{\geq 0}$ and let $[a_0; a_1, a_2, \dots]$ be the continued fraction expansion of x . The convergents are optimal approximation of x in the following sense: Assume $q < q_n$. Then

$$\left| x - \frac{p}{q} \right| \geq \left| x - \frac{p_n}{q_n} \right|$$

for all p . The continued fraction expansion is finite if and only if x is rational.

Proof. Let n be minimal such that $q_n > q$. The convergents p_{n-1}/q_{n-1} and p_n/q_n bracket x and have distance $1/(q_{n-1} q_n)$ from each other. This is smaller than $1/q_{n-1} q$. If p/q is closer to x than p_n/q_n then the distance of p/q to either p_{n-1}/q_{n-1} or p_n/q_n must be smaller than the distance between these points. However,

$$\min\left(\left|\frac{p}{q} - \frac{p_n}{q_n}\right|, \left|\frac{p}{q} - \frac{p_{n-1}}{q_{n-1}}\right|\right) \geq \min\left(\frac{1}{q q_n}, \frac{1}{q q_{n-1}}\right) = \frac{1}{q q_{n-1}}$$

and hence p/q cannot lie closer to x than p_n/q_n .

If the fraction is finite, x is rational. So assume x is rational, say $x = p/q$. If the expansion is infinite, there is a convergent p_n/q_n with $q_n > q$. Then p_n/q_n is closer to x than p/q . This is a contradiction. □

It is now clear how to proceed. We compute an approximation of

$$a = \frac{1}{\cos \alpha} \pm \sqrt{\frac{1}{\cos^2 \alpha} - 1}$$

using floating point arithmetic (of sufficient precision) and then compute a rational approximation of a of sufficient precision.

Exercise 0.6: Give more details on how to compute a rational approximation of a with error at most ε . \diamond

Exercise 0.7: Extend the previous exercise and show how to guarantee an approximation of $\cos \alpha$ with error at most ε (an approximation of α with error at most ε). \diamond

5.7 Notes

Generic programming,

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 [4, 1, 2]. None of the methods is available in LEDA.

Specialized Arithmetics: The orientation predicate for points with integral homogeneous coordinates.

$$\text{sign}(pw \cdot qw \cdot rw) \cdot \text{sign}(pw \cdot (qx \cdot ry - qy \cdot rx) - qw \cdot (px \cdot ry - py \cdot rx) + rw \cdot (px \cdot qy - py \cdot qx)).$$

If the coordinates are less than 2^L , the value of the orientation expression is at most $3 \cdot 2^{3L+1}$. With 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 [6, 8].

Section 5.6 is based on [3].

[[The following should go to the lectures on perturbation.]] 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 ??) 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 [5] 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.

Bibliography

- [1] F. Avnaim, J.-D. Boissonnat, O. Devillers, and F. Preparata. Evaluating signs of determinants with floating point arithmetic. *Algorithmica*, 17(2):111–132, 1997.
- [2] H. Brönnimann, I. Emiris, V. Pan, and S. Pion. Computing exact geometric predicates using modular arithmetic with single precision. In *Proceedings of 13th Annual ACM Symposium on Computational Geometry (SCG'97)*, pages 174–182, 1997.
- [3] J. Canny, B. Donald, and G. Ressler. A rational rotation method for robust geometric algorithms. In A.-S. ACM-SIGGRAPH, editor, *Proceedings of the 8th Annual ACM Symposium on Computational Geometry (SCG '92)*, pages 251–260, 1992.
- [4] K. L. Clarkson. Safe and effective determinant evaluation. *IEEE Foundations of Computer Sci.*, 33:387–395, 1992.
- [5] O. Devillers and F. Preparata. A probabilistic analysis of the power of arithmetic filters. *Discrete & Computational Geometry*, 20:523–547, 1998.
- [6] S. Fortune and C. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15:223–248, 1996. preliminary version in ACM Conference on Computational Geometry 1993.
- [7] K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
- [8] J. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18:305–363, 1997.