Lectures on Reliable Geometric Computing

November 2, 2009

Contents

Lecture 1

Introduction

We give examples of geometric computing tasks and an outline of the course.

1.1 Geometric Computing

Geometric computing refers to computation with geometric objects such as points, lines, hyperplanes, disks, curves, surfaces, and solids.. These objects live in an ambient space. In this book, ambient space will be mainly two- and three-dimensional Euclidean space. Geometric computing is ubiquitous. We illustrate its richness by way of examples. add

Computer-aided Design: Computer-aided design is about the construction of geometric objects. Starting from a ground set of geometric objects, e.g., half-planes, circles, ellipsoids in the plane or cubes, spheres, cylinders, tori, one constructs complex shapes by applying geometric operations to previously constructed objects. Figures 1.1 and 1.2 show examples in two and three dimensions, respectively. Figure **??** shows a more complex example.

Robotics: A central task of robotics is the planning of collision-free paths. Consider a simple situation; the goal is to move a disk-like robot amongst polygonal obstacles in the plane, see Figure **??**. The Voronoi diagram of the obstacles is an appropriate data structure for the task. It consists of all points of maximal clearance from the obstacles; a disk grown at a point of the Voronoi diagram hits two or more obstacles simultaneously. The diagram represents paths for maximal safety. In order to move a disk from a point A to a point B, we first move it from A to a point on the Voronoi diagram, then along the Voronoi diagram, and finally from the Voronoi diagram to B.

Graphics: A 3D scanner is a device that analyzes a real-world object or environment to collect data on its shape and possibly its appearance (i.e. color). In its simplest form it returns a set of points on the surface of the object, see Figure 1.4. The geometric computing task is then to construct a digital three dimensional model of the object from the collected data. The task arises in the production of movies and video games. Other common applications of this technology include industrial design, orthotics and prosthetics, reverse engineering and prototyping, quality control/inspection and documentation of cultural artifacts.

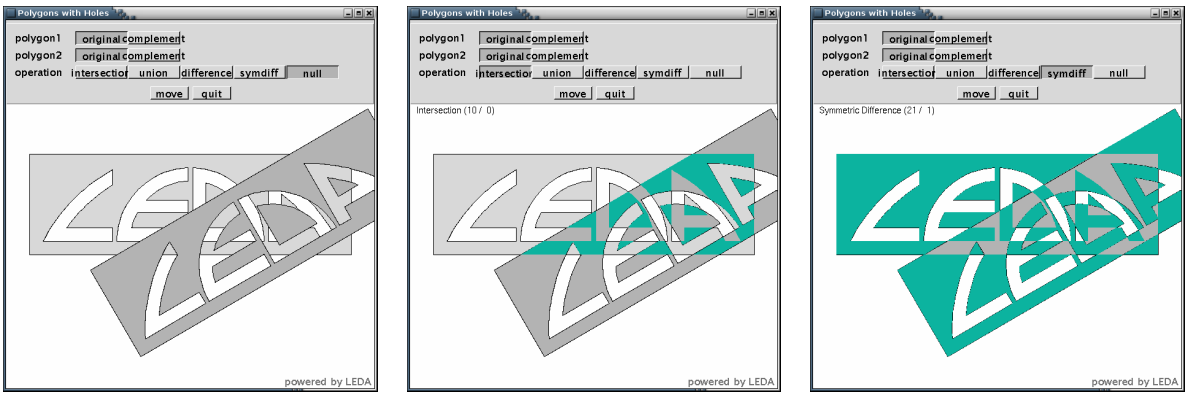


Figure 1.1: The left part shows two polygonal regions with holes (in light and dark grey). The middle part shows the intersection of these regions and the right part shows the symmetric difference. The figure was produced with the LEDA demo polygon_logo [45, 43].

Linear Programming: Linear programming is concerned with the optimization (maximization or minimization) of a linear function subject to linear constraints:

$$\text{maximize} \quad c^T x \quad \text{subject to} \quad Ax \leq b,$$

where x is a vector of n variables, $c \in \mathbb{R}^n$ defines the objective function, $A \in \mathbb{R}^{m \times n}$ is a $m \times n$ real matrix and $b \in \mathbb{R}^m$ is a real vector. Each row a_i of A and the corresponding entry b_i of b defines a linear inequality $a_i x \leq b$. Geometrically, the set of x satisfying this inequality form a halfspace in $\mathbb{R}^n$. The set of x satisfying all constraints $Ax \leq b$ is the intersection of halfspaces, i.e., a convex polyhedron P in $\mathbb{R}^n$. Figure 1.5 shows an example. The aim of linear programming is to find a point $x \in P$ that maximizes $c^T x$. The maximum is attained at a vertex of P that is maximal in direction c.

Mathematics: Algebraic curves and algebraic surfaces are an important topic in mathematics. An algebraic curve is the zero set of a polynomial $p(x,y)$ in two variables and an algebraic surface is the zero set of a polynomial $p(x,y,z)$ in three variables. In applications of algebraic curves and surfaces, it is important to visualize them. Figure 1.6 shows some examples.

More Examples: continue definition with pictures, give examples, examples should come form computational geometry, but also from fields outside CS, e.g.,

- medicine: reconstruction of artery system in brain from NMR-images
- searching for patterns in astronomy
- have a look at Danny Halperin's page: he has nice examples with pictures. Also he taught a course on applied computational geometry.
- GIS: map overlay, map simplification, map labelling,
- examples from the book of Overmars

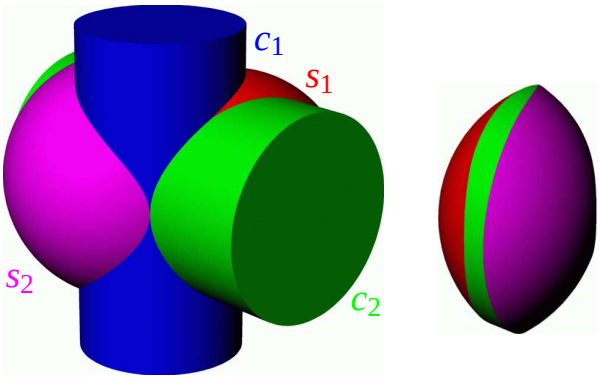


Figure 1.2: The left part shows four solids: two cylinders and two spheres. The right part shows their intersection. The surface of the intersection composed of faces (surface patches stemming from one of the solids), vertices (intersection curves between two input solids) and vertices (points in common to three or more input solids). The picture was produced with the CAD-software Rhino3D.

1.2 Preview of the Course

Now that we have developed an intuition for geometric computations, we are ready for an overview of the course. We will discuss the subject along three axes.

1. Geometric Algorithms

2. Geometric Objects and Predicates

3. Applications.

A geometric algorithm takes geometric objects and produces new geometric objects from them, e.g., it produces the convex hull of a set of points, or a surface interpolating a point cloud, or the intersection of a set of solids, or a path for a robot amidst obstacles. The algorithms operate on geometric objects, query these objects through geometric functions or predicates, and construct new objects through geometric constructors. For example, an algorithm may wish to know the location of point relative to a circle defined by three other points (side-of-circle predicate) or construct a point as the intersection of two curves. We will build on knowledge from discrete mathematics, geometry, combinatorial algorithms, and data structures.

How do we represent geometric objects? Primitive geometric objects such as points, lines, hyperplanes, curves, amd surfaces are represented by their coordinates or their equations. A point may be specified by its Cartesian coordinates and a line through its line equation. The curve shown in Figure **??** is the zero set of the polynomial ..... Predicates and constructors are then functions of these coordinates and equations. We will build on knowledge from analytical geometry, numerical analysis, and algebra. More complex geometric objects are composed of primitive geometric objects and hence we will data structures for these compositions.

Applications are ubiquitous as we seen in the preceding section. We will discuss some of them so that our readers see the full picture.

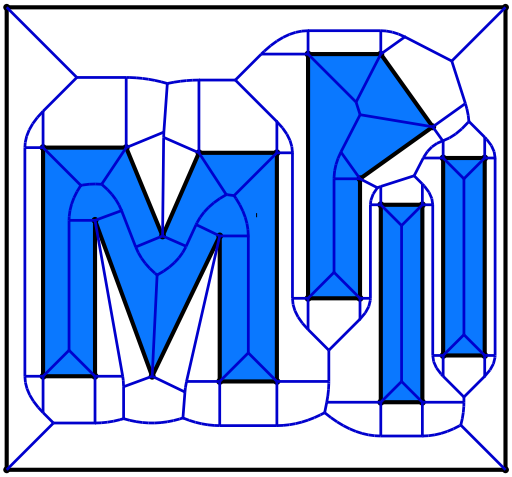


Figure 1.3: Robot motion planning: The figure shows four polygons (the letters M, P, I, and I) enclosed in a square frame. The space between the polygons and the space inside the polygon in partitioned into cells by the Voronoi diagram of the polygons. Imagine to grow a disk centered in an arbitrary point of the plane. In general, the first collision of the growing disk with one of the polygons will be with a single polygon. The Voronoi diagram consists of all points, where this first collision involves two or more polygons. The Voronoi diagram (also called Medial axes) comprises the points of maximum clearance from the disks. The figure was created by Michael Seel [50].

Lecture II: We will start with a simple geometric problem, the computation of the convex hull of a finite set of points in the plane. We will see several algorithms for solving the problem based on different computational paradigm: incremental computation, sweep, and divide-and-conquer. We will formulate the algorithms in terms of geometric predicates. The primitive required for the convex hull problem is the orientation predicate for three points. Given three points p, q, and r in the plane, the predicate tells whether the points form a left turn, are collinear, or form a right turn; see Figure **??** for an illustration. The triple (p,q,r) is a left turn if $p \neq q$ and r lies to the left of the line passing through p and q and oriented from p to q.

Lecture III: Points are usually represented by their Euclidean coordinates. We derive an analytical formula that expresses the orientation of three points in terms of their coordinates. We will see that

$$\mathsf{Orientation}(p,q,r) = \mathrm{sign}(\det \begin{bmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{bmatrix}),$$

where p_x and p_y are the x- and y-coordinate of p, respectively. The sign is $+1$ if (p,q,r) form a left turn, is 0 if they are collinear, and is -1 if they form a right turn.

Point coordinates are real numbers as are the parameters defining other geometric objects, e.g., the coordinates of the center and the radius of a disk. Therefore the natural model of computation for geometric computing is the *Real-RAM*. It is a random access machine with the additional capability of handling real numbers. Of course, the operations on real numbers follow the laws of mathematics. The Real-RAM model is also used successfully in numerical analysis.

t

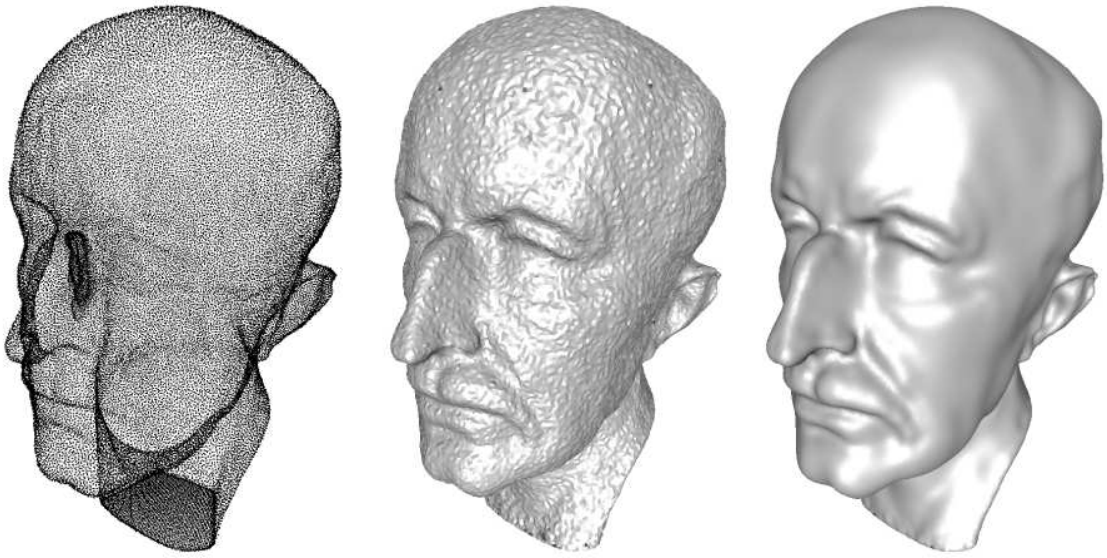


Figure 1.4: The left part of the picture shows a point cloud obtained from a 3D-scan of a bust of Max Planck. The middle part and right part show reconstructions of the object (non-smoothed and smoothed). The reconstruction is by Tamal Dey, University of Ohio.

Real computers do not come with real arithmetic. They provide only floating point arithmetic and bounded integer arithmetic. We will study the effect of floating point arithmetic on geometry. We will first see the effect on the orientation predicate (see Figure **??**) and then the effect on our convex hull algorithm (see Figure **??**). *The former effect will be surprising, the latter disastrous.*

We continue to give more examples of geometric programs (academic and commercial) that break on some inputs. Why is it that

Lecture IV:

Lecture V:

Lecture VI:

Lecture VII:

Lecture VIII:

Lecture IX:

Lecture X:

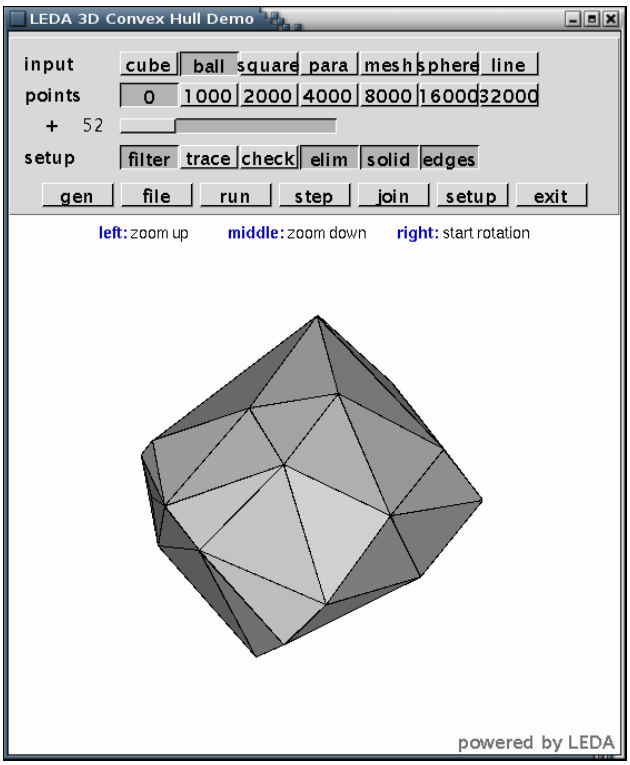


Figure 1.5: A convex polyhedron in three dimensional space. It was generated as the convex hull of a set of points (using the LEDA demo 3d-hull [45, 43]). Alternatively, it could be constructed as the intersection of the halfspaces corresponding to the faces of the polyhedron. Linear programming finds the extreme vertex in the direction of the objective function.

Lecture XI:

Lecture XII:

Lecture XIII:

1.3 Historical Notes

1.4 Implementation Notes

CGAL [23], LEDA [43, 45], and CORE [39] are designed according to the principles put forward in this course. They package much of the content of the course.

Also point to other resources.

Figure 1.6: A figure of an algebraic curve (from Pavel's gallery, a close-up view of a singularity, a figure of a triangulated surface. Show the equations, either in the text or in the caption.

1.5 Exercises

Exercise 0.1: Collect three further examples of geometric computing and document them on the wiki-page of the course. ♢

Exercise 0.2: ♢

Lecture 2

A First Algorithm: Planar Convex Hulls

We will start with a simple geometric problem, the computation of the convex hull of a finite set of points in the plane. We will formulate a basic algorithm that constructs the planar hull in quadratic time. It accesses the input points through a single predicate, the orientation predicate for three points. We will see how this predicate can be realized by a simple formula in the point coordinates. Next we discuss two techniques for improving the running time to $O(n \log n)$, where n is the number of input points. Collinear points require special care in convex hull algorithms and hence we call them a degeneracy. Finally, the algorithm would lead directly to an implementation if we had a Real-RAM to our disposal.

2.1 The Convex Hull Problem

A set is called *convex* if for any two points p and q in the set the entire line segment pq is contained in the set, see Figure 2.1. The *convex hull* $\operatorname{conv} S$ of a set S of points is the smallest (with respect to set inclusion) convex set containing S, see Figure 2.1. A point $p \in S$ is called an *extreme point* of S if there is a closed halfspace containing S such that p is the only point in S that lies in the boundary of the halfspace.

From now on we restrict our discussion to the plane. We define the convex hull problem as the problem of computing the extreme points of a finite set of points as a cyclically ordered list of point, see Figure 2.1. The cyclic order is the counter-clockwise order in which the extreme points appear on the hull.

2.2 A First Algorithm

The simplest method for constructing the convex hull works iteratively. We start with the convex hull of the first three points; we assume for simplicity that the first three points of S are not collinear and come back to this assumption in Section 2.5. For every point, we first determine whether it lies outside the current hull or not. If it is contained in the current hull, we do nothing. Otherwise, the point is an extreme point of the new hull and we update the hull by constructing the tangents from the new point to the old hull, see Figure 2.2a.

How can we determine whether a point r is contained in the current hull? Recall that the current hull is represented by its cyclic list of extreme points in counter-clockwise order, say $(v_0, v_1, \ldots, v_{k-1}, v_k = v_0)$. Consider a pair (v_i, v_{i+1}) of consecutive extreme points. Any point in the current hull lies on or to the left of the oriented line $\ell(v_i, v_{i+1})$ and every point to the right of $\ell(v_i, v_{i+1})$ lies outside the current hull, see Figure 2.2b. The geometric predicate of locating a point with respect to an oriented line is so important that we give it a name.

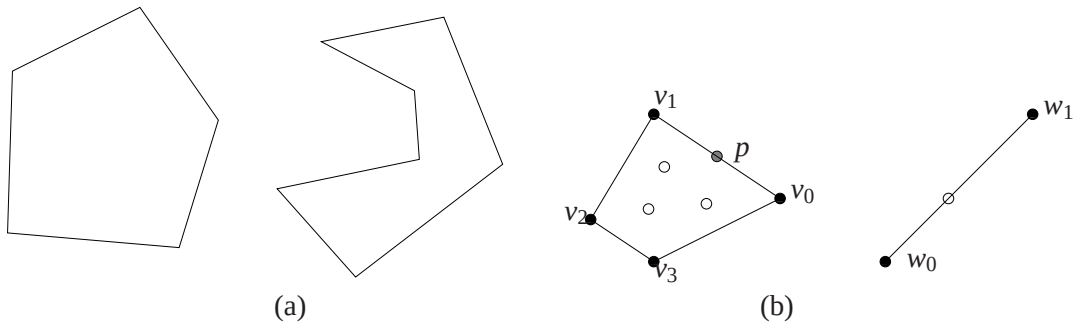


Figure 2.1: (a) shows a convex and a non-convex set, (b) shows two point sets and their convex hulls. The extreme points v_0, v_1, v_2, and v_3, respectively w_0 and w_1 are highlighted as solid disks. The point p lies on the boundary of the hull, but is not an extreme point. The cyclic clockwise list of extreme points is v_0, v_1, v_2, v_3 and w_0, w_1 (or any cycle shifts thereof), respectively.

Definition: Let p, q, and r be points in the plane (see Figure 2.3). If $p \neq q$, let $\ell(p,q)$ be the line passing through p and q and oriented from p to q. Then

$$\mathsf{Orientation}(p,q,r) = \begin{cases} +1 & \text{if } p \neq q \text{ and } r \text{ lies to the left of } \ell(p,q) \\ 0 & \text{if } p = q \text{ or } p \neq q \text{ and } r \text{ lies on } \ell(p,q) \\ -1 & \text{if } p \neq q \text{ and } r \text{ lies to the right of } \ell(p,q). \end{cases}$$

If $\mathsf{Orientation}(p,q,r) = +1$ (-1), we say that (p,q,r) form a left (right) turn, if $\mathsf{Orientation}(p,q,r) = +1$, the points are collinear. We next specialize to the convex hull problem. Assume that v_i and v_{i+1} are consecutive extreme points in the counter-clockwise order of extreme points. If r lies to the right of $\ell(v_i, v_{i+1})$, we also say that r *sees* the (counter-clockwise) hull edge $v_i v_{i+1}$ and that this hull edge is *visible* from r.

THEOREM 1. *A point r lies outside* $\operatorname{conv} S$ *if and only if it can see at least one edge of* $\operatorname{conv} S$.

Proof. If r can see a hull edge, it is clearly outside $\operatorname{conv} S$. Assume next that $r \notin \operatorname{conv} S$ and let z be the point in $\operatorname{conv} S$ closest to r. If r lies in the interior of some hull edge then r can see this edge. So assume that z is an extreme point of S, say $z = v_i$. Then r sees at least one of the two hull edges incident to v_i. □

We now know how to check whether a new point r lies outside the current hull. We simply check whether it can see some hull edge. We will see more efficient methods in Section **??**. We next turn to the update step. We need the notion of *weak visibility*. If r lies to the right of or on $\ell(v_i, v_{i+1})$, we say that r *weekly sees* the hull segment $v_i v_{i+1}$ and that this segment is *weakly visible* from r.

THEOREM 2. *Let $(v_0, v_1, \ldots, v_{k-1})$ be the sequence of extreme points of* $\operatorname{conv} S$ *in counter-clockwise order and assume that $r \notin \operatorname{conv} S$. The hull edges weakly visible from r form a contiguous subsequence and so do the edges that are not weakly visible.*

If (v_i, v_{i+1}), …, (v_{j-1}, v_j) is the subsequence of weakly visible edges, the updated hull is obtained by replacing the subsequence $(v_{i+1}, \ldots, v_{j-1})$ by r. The subsequence $(v_i, \ldots, v_j)$ is taken in the circular sense, i.e., if $i > j$ then the subsequence is $(v_i, \ldots, v_{k-1}, v_0, \ldots, v_j)$.

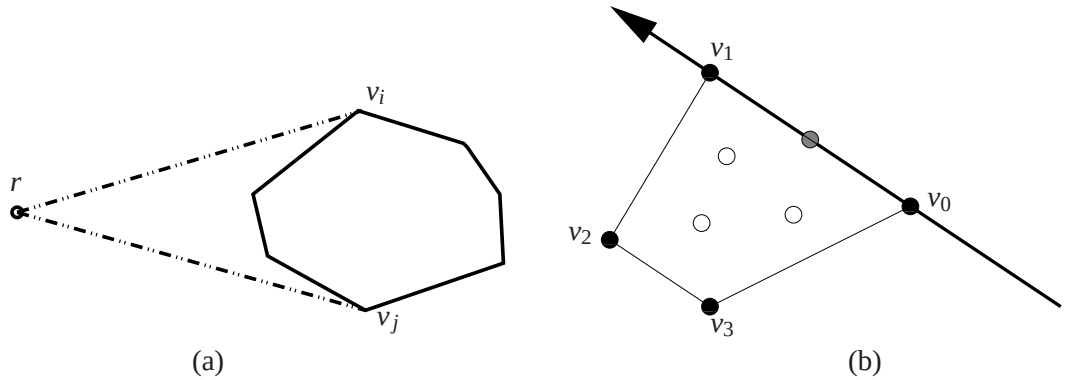


Figure 2.2: In (a) the current hull is shown as a polygon whose boundary is indicated by solid segments. The point r lies outside the current hull. The tangents from r to the current hull touch the hull in vertices v_i and v_j. The boundary of the new hull consists of the segment rv_j, followed by the part of the old hull from v_j to v_i, followed by the segment $v_i r$.
In (b) the oriented line $\ell(v_0, v_1)$ is highlighted. Every point to the right of this line lies outside the current hull.

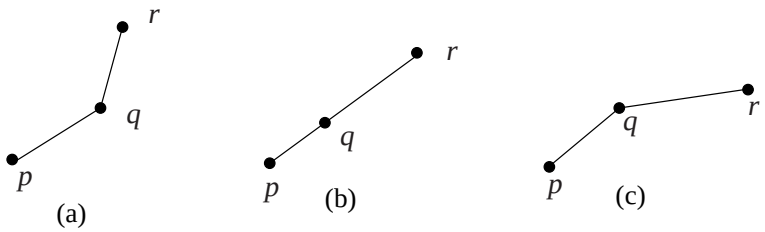


Figure 2.3: (a) shows a left turn, (b) shows collinear points, and (c) shows a right turn.

Proof. Consider the tangents t_1 and t_2 from r to $\operatorname{conv} S$. A tangent t_i intersects the boundary of $\operatorname{conv} S$ either in a single vertex or in an edge. In either case, let z_i be the tangent point of maximal distance. Then one of the hull edges incident to z_i is weakly visible from r and one is not. Moreover, z_1 and z_2 split the boundary of $\operatorname{conv} S$ into two chains. In one chain all edges are weakly visible from r, and in the other chain, no edge is weakly visible from r. The boundary of $\operatorname{conv}(S \cup r)$ consists of the chain of edges that are not weakly visible from r plus the two segments $z_1 r$ and $r z_2$. □

Theorems 1 and 2 lead to the incremental convex hull algorithm shown as Algorithm 1. We still need to explain how we find all edges weakly visible from r and how we update L. Starting from the visible edge e, we move counter-clockwise along the boundary until a non-weakly-visible edge is encountered. Similarly, we move clockwise from e until a non-weakly-visible edge is encountered.

How to update the list L? We can delete the vertices in $(v_{i+1}, \ldots, v_{j-1})$ after all visible edges are found, as suggested in the above sketch ("the off-line strategy") or we can delete them concurrently with the search for weakly visible edges ("the on-line strategy").

We have now almost completed the description of our first geometric algorithm. We still need to discuss the implementation of the orientation predicate. We will see in the next section that the orientation predicate

Algorithm 1 Incremental Convex Hull Algorithm

initialize L to a counter-clockwise triangle (a,b,c) with $a,b,c \in S$. Remove a,b,c from S.
for all $r \in S$ **do**
  if there is an edge e visible from r **then**
    determine the sequence $((v_i,v_{i+1}),(v_{i+1},v_{i+2})\ldots,(v_{j-1},v_j))$ of edges that are weakly visible from r.
    replace the subsequence $(v_{i+1},\ldots,v_{j-1})$ in L by r.
  end if
end for

can be formulates as a simple arithmetic expression in point coordinates and hence orientation of three points can be determined in constant time.

Algorithm 1 computes the convex hull of n points in $O(n^2)$ time. For any point r, we check all edges of the current hull for visibility and maybe weak visibility. We also remove zero or more points from the current hull. Thus any point is processed in $O(n)$ time. The bound of $O(n^2)$ follows. In Section 2.4 we will improve the running time to $O(n\log n)$.

2.3 The Orientation Predicate

LEMMA 3. *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 2.4. 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 $r' = r-p$[1] . On the other hand subtracting the first column from the other two columns

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

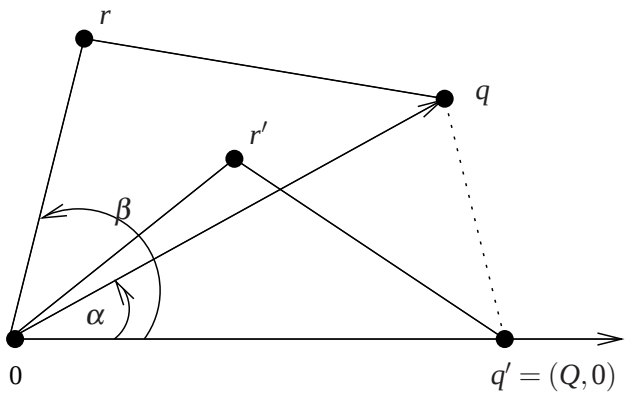


Figure 2.4: Proof of Lemma 3.

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 analytical formula for the orientation predicate:

$$\mathsf{Orientation}(p,q,r) = \mathrm{sign}(\det \begin{bmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{bmatrix}) = \mathrm{sign}((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x)). \quad (1)$$

We have $\mathsf{Orientation}(p,q,r) = +1$ (resp., -1, 0) iff the polyline (p,q,r) represents a left turn (resp., right turn, collinearity). Interchanging two points in the triple changes the sign of the orientation.

We will frequently represent points by homogeneous coordinates. Consider a point p with Cartesian coordinates p_x and p_y. The homogeneous coordinates of p are any triply (px, py, pw) such that $p_x = px/pw$ and $p_y = py/pw$. Homogeneous coordinates are not unique; multiplication by a non-zero factor does not change the point represented. If the Cartesian coordinates are rationals, we may choose the homogeneous coordinates to be integral. In this situation, pw is a common denominator for x- and y-coordinate. In homogeneous coordinates, we have

$$\begin{aligned} \mathsf{Orientation}(p,q,r) &= \mathrm{sign}(\det \begin{bmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{bmatrix}) = \mathrm{sign}(\det \begin{bmatrix} 1 & px/pw & py/pw \\ 1 & qx/qw & qy/qw \\ 1 & rx/rw & ry/rw \end{bmatrix}) \\ &= \mathrm{sign}(pw \cdot qw \cdot rw) \cdot \mathrm{sign}(\det \begin{bmatrix} pw & px & py \\ qw & qx & qy \\ rw & rx & ry \end{bmatrix}) \\ &= \mathrm{sign}(pw \cdot qw \cdot rw) \cdot \mathrm{sign}(pw \cdot (qx \cdot ry - qy \cdot rx) - qw \cdot (px \cdot ry - py \cdot rx) + rw \cdot (px \cdot qy - py \cdot qx)). \end{aligned}$$

2.4 Efficiency

Our incremental algorithm for convex hulls runs in $O(n^2)$ time on an input of n points. We show how to improve the running time to $O(n \log n)$. We first observe that the cost of updating the hull is $O(n)$, once it is known whether the new point r sees some edge.

Indeed, if r sees no edge, the old hull is the new hull and the cost of the update is zero. So assume that r sees some edge e of the current hull. We walk from e in both directions as long as edges are weakly visible. The cost of the walk is $O(1+x)$, where x is the number of edges weakly visible from r. We then delete x edges from the convex hull and add two new edges. We charge $O(1)$ to the update and to each edge removed. Since any edge can be removed only once and since at most $2n$ edges are every constructed, the total charge for the update is $O(n)$.

We next describe two techniques for finding a first visible edge or to decide that there is none.

2.4.1 A Sweep Algorithm

We simplify the search for a visible edge by processing the points in lexicographic order. A point p precedes a point q in lexicographic order if either p has the smaller x-coordinate or the x-coordinates are the same and p has the smaller y-coordinate. Sorting points according to lexicographic order takes $O(n \log n)$ time.

The advantage of processing the points in lexicographic order is twofold: First, any point is outside the convex hull of the preceding points, and second, one of the edges incident to the lexicographically largest vertex is visible from the next point. Thus the search for a visible hull edge is trivial and takes $O(1)$ time.

THEOREM 4. *The sweep hull algorithm constructs the convex hull of n points in the plane in $O(n \log n)$ time.*

2.4.2 Incremental Construction

edit? Boissonnec?

We describe an alternative method for speeding up the search for a visible hull edge. The idea is to maintain the *history of the construction*. Again, we start with the counter-clockwise triangle formed by the first three points. The algorithm maintains the current hull as a cyclically linked list of edges and also keeps all edges that ever belonged to a hull. Every edge that is not on the current hull anymore points to the two edges that replaced it. More precisely, assume that S is the set of points already seen and that p is a point outside the current hull $CH(S)$. There is a chain C of edges of the boundary of $CH(S)$ that do not belong to the boundary of $CH(S \cup p)$. The chain is replaced by the two tangents from p to the previous hull. All edges in C are made to point to the two new edges, see Figure 2.5.

We are now ready to deal with the insertion of a point p. We proceed in two steps. We first determine whether p is outside the current hull and then update the hull (if p is outside).

In order to find out whether p lies outside the current hull, we walk through the history of hulls; see algorithm 2. We first determine whether p can see one of the edges of the initial triangle. If it can see no edge of the initial triangle, p lies inside the current hull and we are done. So assume that p can see an edge of the initial triangle, say e. If e is an edge of the current hull, p lies outside the current hull and e is a visible hull edge. If e is not an edge of the current hull, let r_0 and r_1 be the two edges that replaced e when $CH(S)$ was enlarged to $CH(S \cup q)$. p is outside $CH(S \cup q)$ if it sees either r_0 or r_1, see Figure 2.6. If p sees neither r_0 nor r_1, we stop. Otherwise, we set e to a visible edge among r_0 and r_1 and continue in the same fashion. In this way, the search either stops or finds a hull edge visible from p. Once we have found such an edge, we continue as in the basic algorithm.

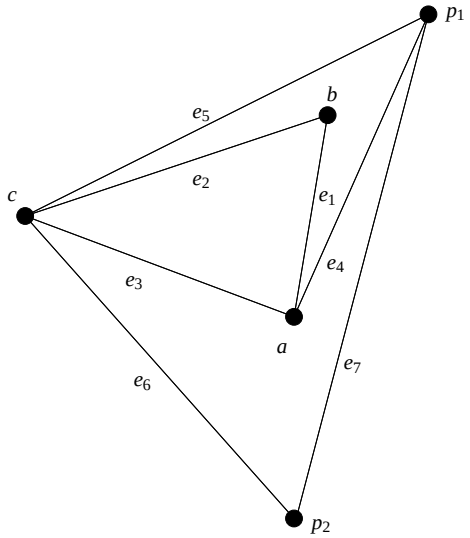


Figure 2.5: The initial convex hull consists of the points a, b, and c. When point p_1 is added the edges e_1 and e_2 are deleted from the hull and the edges e_4 and e_5 are added, and when p_2 is added to the hull the edges e_3 and e_4 are deleted from the hull and the edges e_6 and e_7 are added. The boundary of the current hull consists of edges e_7, e_5, and e_6 in counter-clockwise order. Every edge ever deleted from the hull points to the two edges that replaced it, e.g., e_3 and e_4 point to e_6 and e_7.

What is the running time of the incremental construction of convex hulls? The worst case running time is $O(n^2)$ since the time to insert a point is $O(n)$. The time to insert a point is $O(n)$ since there are at most $2(k+1)$ edges after the insertion of k points and since every edge is looked at at most once in the insertion process.

The best case running time is $O(n)$. An example for the best case is when the points a, b, and c span the hull.

2.4.3 Randomized Incremental Construction*

The average case running time is $O(n \log n)$ as we will show next. What are we averaging over? We consider a fixed but arbitrary set S of n points and average over the $n!$ possible insertion orders. The following theorem is a special case of the by now famous *probabilistic analysis of incremental constructions* started by Clarkson and Shor [13]. The books [48, 5, 47, 16] contain detailed presentations of the method. The reader may skip the proof of Theorem 5. Why do we include a proof at all given the fact that the method is already well treated in textbooks? We give a proof because the cited references prove the theorem only for points in general position. We want to do without the general position assumption in this book.

THEOREM 5. *The average running time of the incremental construction method for convex hulls is $O(n \log n)$.*

Proof. We assume for simplicity that the points in S are pairwise distinct. The theorem is true without this assumption; however, the notation required in the proof is more clumsy.

The running time of the algorithm is linear iff all points in S are collinear. So let us assume that S contains three points that are not collinear. In this case we will first construct a triangle and then insert the

Algorithm 2 The search for a visible edge of the current hull.

```
let e be a visible edge of the initial triangle; stop, if e does not exist.
while (true) do
  if e is an edge of the current hull then
    stop("e is a visible hull edge");
  end if
  if one of the replacement edges r_0 or r_1 is visible then
    let e be a visible replacement edge;
  else
    stop("p lies inside current hull");
  end if
end while
```

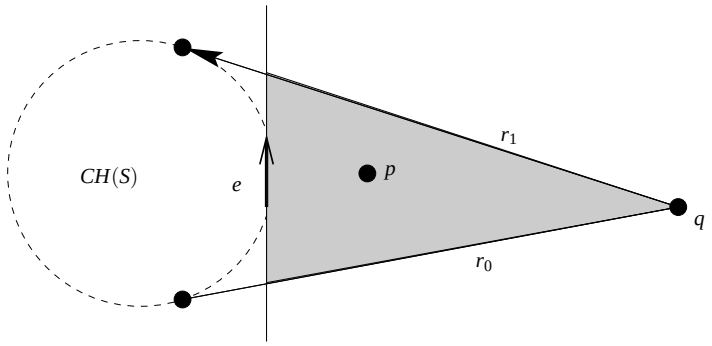


Figure 2.6: e is a (counter-clockwise) edge of the current hull and p lies to the right of it; e is replaced by r_0 and r_1 when the point q is added. If p lies neither to the right of r_0 nor to the right of r_1 then p lies in the shaded region and hence in $CH(S \cup q)$.

remaining points. Let p be one of the remaining points. When p is inserted, we first determine the position of p with respect to the initial triangle (time $O(1)$), then search for a hull edge e visible by p, and finally update the hull. The time to update the hull is $O(1)$ plus some bounded amount of time for each edge that is removed from the hull. We conclude that the total time (= time summed over all insertions) spent outside the search for a visible hull edge is $O(n)$.

In the search for a visible hull edge we perform tests $\mathit{rightturn}(x,y,p)$ where x and y are previously inserted points. We call a test *successful* if it returns true and observe that in each iteration of the while-loop at most two rightturn tests are performed and that in all iterations except the last at least one rightturn test is successful. It therefore suffices to bound the number of successful rightturn tests.

What characterizes hull edges? An oriented segment xy is a CCW hull edge if there is no point in $z \in S$ that weakly sees xy, i.e., either lies in the right halfplane of $\ell(x,y)$ or lies on the line $\ell(x,y)$ but not on the segment xy. For an ordered pair (x,y) of distinct points in S we use $K_{x,y}$ to denote the set of points z in S such that $\mathit{rightturn}(x,y,z)$ is true plus[2] the set of points on the line through (x,y) but not between x and y, see Figure 2.7. Every point $z \in K_{x,y}$ is a witness for xy not being a CCW hull edge. We use k_{xy} to denote the

[2]The set to be defined next is empty if S is in general position. The probabilistic analysis of incremental constructions usually assumes general position. We do not want to assume it here and hence have to modify the proof somewhat.

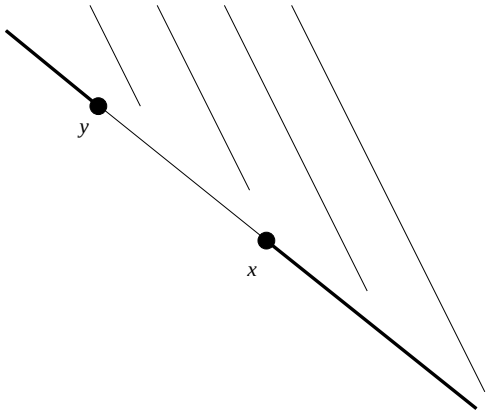


Figure 2.7: $K_{x,y}$ consists of all points in the shaded region plus the two solid rays.

cardinality of $K_{x,y}$, F_k to denote the set of pairs (x,y) with $k_{xy} = k$, $F_{\leq k}$ to denote the set of pairs (x,y) with $k_{xy} \leq k$, and f_k and $f_{\leq k}$ to denote the cardinalities of F_k and $F_{\leq k}$, respectively. We have

LEMMA 6. *The average number A of successful rightturn tests is bounded by* $\sum_{k\geq 1} 2 f_{\leq k}/k^2$.

Proof. Consider a pair (x,y) with $k_{xy} = k$. If some point in $K_{x,y}$ is inserted before both x and y are inserted then (x,y) is never constructed as a hull edge and hence no rightturn tests $(x,y,-)$ are performed. However, if x and y are inserted before all points in $K_{x,y}$ then up to k successful rightturn tests (x,y,z) are performed.

The probability that x and y are inserted before all points in $K_{x,y}$ is

$$2!k!/(k+2)!$$

since there are $(k+2)!$ permutations of $k+2$ points out of which $2!k!$ have x and y as their first two elements. Thus the expected number of successful rightturn tests (x,y,z) is bounded by

$$2!k!/(k+2)! \cdot k = 2 \cdot k/(k+1)(k+2) < 2/(k+1).$$

The argument above applies to any pair (x,y) and hence the average number of successful rightturn tests is bounded by

$$\sum_{k\geq 1} 2 f_k/(k+1).$$

We next write $f_k = f_{\leq k} - f_{\leq k-1}$ and obtain

$$\begin{aligned} A &\leq \sum_{k\geq 1} 2(f_{\leq k} - f_{\leq k-1})/(k+1) = \sum_{k\geq 1} 2 f_{\leq k}(1/(k+1) - 1/(k+2)) \\ &= \sum_{k\geq 1} 2 f_{\leq k}/((k+1)(k+2)). \end{aligned}$$

□

It remains to bound $f_{\leq k}$. We use random sampling to derive a bound.

LEMMA 7. $f_{\leq k} \leq 2e^2 n \cdot k$ *for all* k, $1 \leq k \leq n$.

Proof. There are only n^2 pairs of points of S and hence we always have $f_{\leq k} \leq n^2$. Thus, the claim is certainly true for $n \leq 10$ or $k \geq n/4$.

So assume that $n \geq 10$ and $k \leq n/4$ and let R be a random subset of S of size r. We will fix r later. Clearly, the convex hull of R consists of at most r edges. On the other hand, if for some $(x,y) \in F_{\leq k}$, x and y are in R but none of the points in $K_{x,y}$ is in R, then (x,y) will be an edge of the convex hull of R. The probability of this event is

$$\frac{\binom{n-i-2}{r-2}}{\binom{n}{r}} \geq \frac{\binom{n-k-2}{r-2}}{\binom{n}{r}},$$

where $i = k_{x,y} \leq k$. Observe that the event occurs if x and y are chosen and the remaining $r-2$ points in R are chosen from $S \setminus \{x,y\} \setminus K_{x,y}$. The expected number of edges of the convex hull of R is therefore at least

$$f_{\leq k} \cdot \frac{\binom{n-k-2}{r-2}}{\binom{n}{r}}.$$

Since the number of edges is at most r we have

$$f_{\leq k} \cdot \binom{n-k-2}{r-2} / \binom{n}{r} \leq r$$

or

$$f_{\leq k} \leq r \cdot \binom{n}{r} / \binom{n-k-2}{r-2} = r \cdot \frac{n(n-1)}{r(r-1)} \cdot \frac{[n-2]_{r-2}}{[n-k-2]_{r-2}},$$

where $[n]_i = n(n-1)\cdots(n-i+1)$. Next observe that

$$\begin{aligned} \frac{[n-2]_{r-2}}{[n-k-2]_{r-2}} &\leq \frac{[n]_r}{[n-k]_r} = \prod_{i=0}^{r-1} \frac{n-i}{n-k-i} = \prod_{i=0}^{r-1} \left(1 + \frac{k}{n-k-i}\right) \\ &= \exp\left(\sum_{i=0}^{r-1} \ln(1 + k/(n-k-i))\right) \leq \exp\left(rk/(n-k-r)\right), \end{aligned}$$

where the last inequality follows from $\ln(1+x) \leq x$ for $x \geq 0$ and the fact that $k/(n-k-i) \leq k/(n-k-r)$ for $0 \leq i \leq r-1$. Setting $r = n/(2k)$ and using the fact that $n-k-r \geq n/4$ for $k \leq n/4$ and $n \geq 10$, we obtain

$$f_{\leq k} \leq e^2 n^2/r = 2e^2 nk.$$

□

Putting our two lemmas together completes the proof of Theorem 5

$$A \leq 4e^2 \sum_{k \geq 1} nk/k^2 = O(n \log n).$$

□

There are two important situations when the assumptions of the theorem above are satisfied:

- When the points in S are generated according to a probability distribution for points in the plane.
- When the points are randomly permuted before the incremental construction process is started. We then speak about a *randomized incremental construction.*

2.5 Degeneracy

We assumed that the first three points in the input span a proper triangle. How can we remove this assumption?

In an off-line setting, i.e., all points are available at program start, we scan over the points once. Let p be the first point. We scan until we find a point q that is different from p. If all input points are equal to p, the convex hull is equal to the set consisting only of p. So assume we have two distinct points p and q. We continue scanning until we find a point r that is not collinear to p and q. If there is no such point, the convex hull is contained in the line passing through p and q and we simply need to find the two extreme points on the line. If there is such a point, we have found the initial triangle.

In an on-line setting, we have to work slightly harder. We initialize the hull to $\{p\}$. As long as input points are equal to p, there is nothing to do. As soon, as we encounter a point q different from p, we know that the hull is at least one-dimensional. The current hull is the line segment pq. As long as input points are collinear to p and q, the hull stays a segment and we update it accordingly. Once an input point r that is not collinear with p and q comes along, we know that the hull is two-dimensional and we switch to the algorithm discussed in the preceding sections.

If no three points are collinear, the assumption is trivially satisfied. Also, there is no need to distinguish between visible and weakly visible edges as there are no edges that are weakly visible but not visible. Collinear points make the formulation of convex hull algorithms more complex and therefore we call them a *degenerate configuration* for the convex hull problem.

Geometric algorithms are frequently formulated under the *non-degeneracy assumption* or *general position assumption*: The input contains no degenerate configuration. In Lecture **??** we will study perturbation as a general technique for ensuring general position.

2.6 Arbitrary Dimension

either in the text or as a remark in historial and implementation notes.

2.7 The Real-RAM

We have an algorithm for planar convex hulls. *Do we have an implementation, i.e., is it straight-forward to convert the discussion into a running program in a popular programming language? The answer is No.*

In the formulation of the algorithm we have tacitly assumed the *Real-RAM* model of computation. A Real-RAM is a random access machine with the capability of handling real numbers. Of course, the operations on real numbers follow the laws of mathematics. The Real-RAM model is the natural computing model for geometric computing and numerical analysis. After all geometric objects are usually specified by real parameters: point coordinates are reals, the radius of a circle is a real, plane coefficients are reals, and so on.

Unfortunately, one cannot buy a Real-RAM. Real computers do not come with real arithmetic. They provide only floating point arithmetic and bounded integer arithmetic. We will study the effect of floating point arithmetic on geometry in the next lecture. We will see that we are far from an implementation.

In Lectures **??** to **??** we will then discuss the efficient realization of a Real-RAM to the extent needed by the convex hull algorithm and any other geometric algorithm that deal only with linear objects.

2.8 Historical Notes

The sweep hull algorithm was proposed by Andrew [1]; it refines an earlier algorithm of Graham [34].
randomized incremental algorithm [13]. Dimension jumps first in

2.9 Implementation Notes

2.10 Exercises

Lecture 3

A First Implementation

We come to the implementation of our convex hull algorithm. There is one choice to be made. *How do we realize real arithmetic?* We make the obvious choice. We use what computers offer us: floating point arithmetic, i.e.,

Implementation of a Real RAM = RAM + double precision floating point arithmetic.

Double precision floating point arithmetic is governed by the IEEE standard 754-1985 [32, **?**]). Modern processors implement this standard and programming languages provide it under names such as "double" (C++), "XXX" (Java), TODO.. Floating point arithmetic is the workhorse for numerical computations. TODO Double precision floating point numbers have the form

$$\pm\, m\, 2^e$$

where $m = 1.m_1m_2\ldots m_{52}$, $m_i \in \{0,1\}$, is the mantissa in binary and e is the exponent satisfying $-1023 < e < 1024$.[1] We discuss floating point arithmetic in detail in Lecture **??**. At this point it suffices to know that *arithmetic in a floating point system is approximate and not exact.* The result of any floating point arithmetic operation is the exact result of the operation rounded to the nearest double (with ties broken using some fixed rule). For example, in a decimal floating point system with a mantissa of two places, we have

$$0.36 \cdot 0.11 = 0.40$$

since the exact result 0.0396 is rounded to the approximate result 0.040.

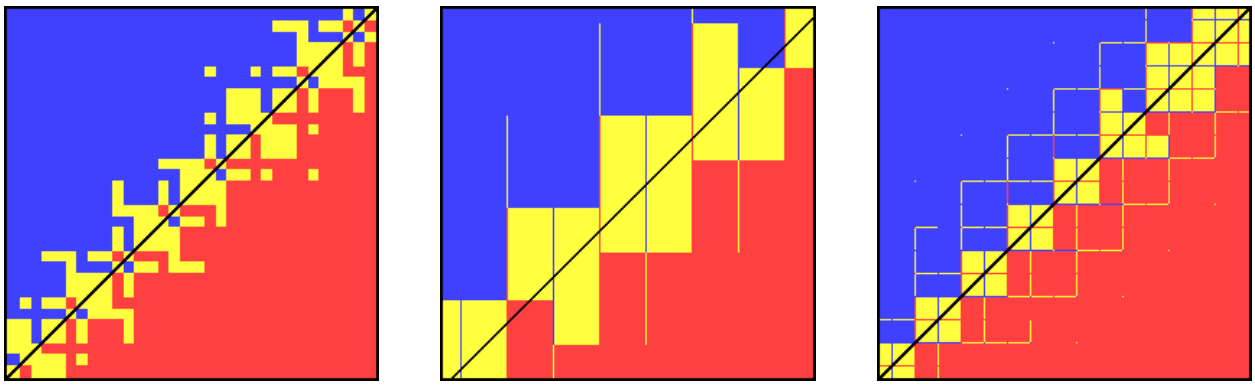
We will see in this lecture that floating point arithmetic is a pure substitute for real arithmetic and that the floating point implementation of our algorithm can produce very strange results. We hope that, after seeing these examples, our students look forward to the solution techniques that we present in later lectures. The core of a C++ implementation of our algorithm is given in Section 3.2. The full code can be found in the companion web page [2] of article [42] on which this lecture is based.

3.1 The Geometry of Float-Orient

Our convex hull algorithms uses the orientation predicate for three points. In the last lecture we derived the following formula for the orientation predicate. For three points $p = (p_x, p_y)$, $q = (q_x, q_y)$, and $r = (r_x, r_y)$

[1] We ignore here so called *denormalized* numbers that play no role in our experiments and arguments.

[2] `http://www.mpi-inf.mpg.de/departments/d1/ClassroomExamples/`



p : $\begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}$ q : $\begin{pmatrix} 12 \\ 12 \end{pmatrix}$ r : $\begin{pmatrix} 24 \\ 24 \end{pmatrix}$

(a)

$\begin{pmatrix} 0.50000000000002531 \\ 0.5000000000000171 \end{pmatrix}$ $\begin{pmatrix} 17.300000000000001 \\ 17.300000000000001 \end{pmatrix}$ $\begin{pmatrix} 24.00000000000005 \\ 24.000000000000517765 \end{pmatrix}$

(b)

$\begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}$ $\begin{pmatrix} 8.8000000000000007 \\ 8.8000000000000007 \end{pmatrix}$ $\begin{pmatrix} 12.1 \\ 12.1 \end{pmatrix}$

(c)

Figure 3.1: The weird geometry of the float-orientation predicate: The figure shows the results of *float_orient*$(p_x + Xu_x, p_y + Yu_y, q, r)$ for $0 \le X, Y \le 255$, where $u_x = u_y = 2^{-53}$ is the increment between adjacent floating-point numbers in the considered range. The result is color coded: Yellow (red, blue, resp.) pixels represent collinear (negative, positive, resp.) orientation. The line through q and r is shown in black.

in the plane let

$$\mathsf{Orientation}(p,q,r) = \mathit{sign}((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x)). \tag{1}$$

We have $\mathsf{Orientation}(p,q,r) = +1$ (resp., -1, 0) iff the polyline (p,q,r) represents a left turn (resp., right turn, collinearity). *When the orientation predicate is implemented in this way and evaluated with floating-point arithmetic, we call it* *float_orient*(p,q,r) *to distinguish it from the ideal predicate.*

What is the geometry of *float_orient*, i.e., which triples of points are classified as left-turns, right-turns, or collinear? The following type of experiment addresses the question: We choose three points p, q, and r and then compute *float_orient*(p',q,r) for points p' in the floating-point neighborhood of p. More precisely, let u_x be the increment between adjacent floating-point numbers in the range right of p_x; for example, $u_x = 2^{-53}$ if $p_x = \frac{1}{2}$ and $u_x = 4 \cdot 2^{-53}$ if $p_x = 2 = 4 \cdot \frac{1}{2}$. Analogously, we define u_y. We consider

$$\mathit{float\_orient}((p_x + Xu_x, p_y + Yu_y), q, r)$$

for $0 \le X, Y \le 255$. We visualize the resulting 256×256 array of signs as a 256×256 grid of colored pixels: A yellow (red, blue) pixel represents collinear (negative, positive, respectively) orientation. In the figures in this section we also indicate an approximation of the exact line through q and r in black.

Figure 3.1(a) shows the result of our first experiment: We use the line defined by the points $q = (12,12)$ and $r = (24,24)$ and query it near $p = (0.5,0.5)$. We urge the reader to pause for a moment and to sketch what he/she expects to see. The authors expected to see a yellow band around the diagonal with nearly straight boundaries. Even for points with such simple coordinates the geometry of *float_orient* is quite weird: the set of yellow points (= the points classified as on the line) does not resemble a straight line and

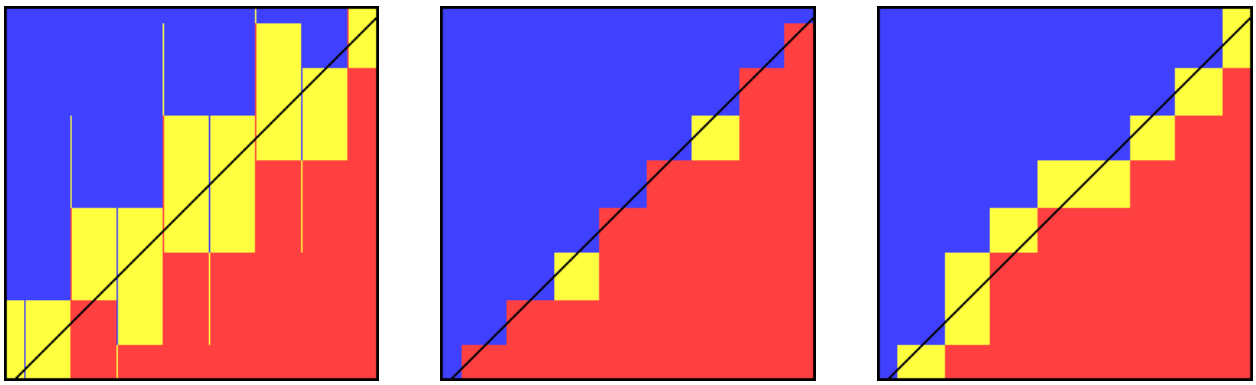


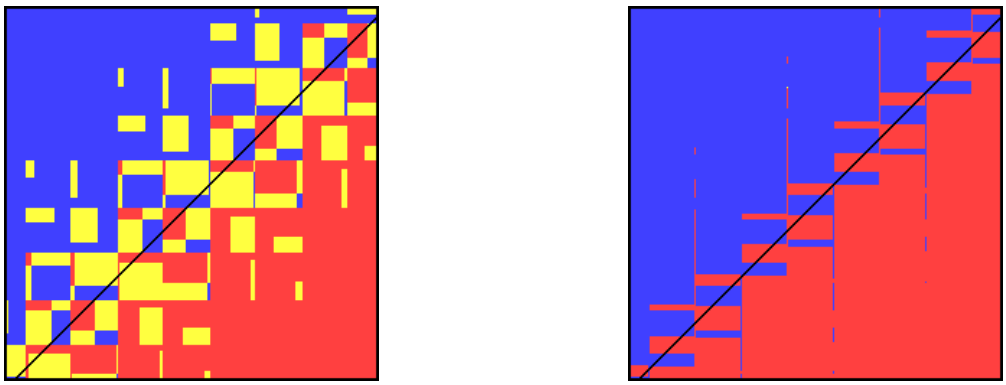
Figure 3.2: We repeat the example from Figure 3.1(b) and show the result for all three distinct choices for the pivot; namely p on the left, q in the middle, and r on the right. All figures exhibit sign reversal.

the sets of red or blue points do not resemble half-spaces. We even have points that change the side of the line, i.e., are lying left of the line and being classified as right of the line and vice versa.

In Figures 3.1(b) and (c) we have given our base points coordinates with more bits of precision by adding some digits behind the binary point. This enhances the cancellation effects in the evaluation of *float_orient* and leads to even more striking pictures. In (b), the red region looks like a step function at first sight. Note however, it is not monotone, has yellow rays extending into it, and red lines extruding from it. The yellow region (= collinear-region) forms blocks along the line. Strangely enough, these blocks are separated by blue and red lines. Finally, many points change sides. In Figure (c), we have yellow blocks of varying sizes along the diagonal, thin yellow and partly red lines extending into the blue region (similarly for the red region), red points (the left upper corners of the yellow structures extending into the blue region) deep inside the blue region, and isolated yellow points almost 100 units away from the diagonal.

All diagrams in Figure 3.1 exhibit block structure. We now explain why: We focus on one dimension, i.e., assume we keep Y fixed and vary only X. We evaluate $\mathit{float\_orient}((p_x + Xu_x, p_y + Yu_y), q, r)$ for $0 \leq X \leq 255$, where $u_x = u_y$ is the increment between adjacent floating-point numbers in the considered range. Recall that $\mathsf{Orientation}(p,q,r) = \mathit{sign}((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x))$. We incur round-off errors in the additions/subtractions and also in the multiplications. Consider first one of the differences, say $q_x - p_x$. In (a), we have $q_x = 12$ and $p_x \approx 0.5$. Since 12 has four binary digits, we lose the last four bits of X in the subtraction, in other words, the result of the subtraction $q_x - p_x$ is constant for 2^4 consecutive values of X. Because of rounding to nearest, the intervals of constant value are $[8,23]$, $[24,39]$, $[40,55]$ .... Similarly, the floating-point result of $r_x - p_x$ is constant for 2^5 consecutive values of X. Because of rounding to nearest, the intervals of constant value are $[16,47]$, $[48,69]$, ... Overlaying the two progressions gives intervals $[16,23]$, $[24,39]$, $[40,47]$, $[48,55]$, ... and this explains the structure we see in the rows of (a). We see short blocks of length 8, 16, 24, ... in (a). In (b) and (c), the situation is somewhat more complicated. It is again true that we have intervals for X, where the results of the subtractions are constant. However, since q and r have more complex coordinates, the relative shifts of these intervals are different and hence we see narrow and broad features.

Exercise 0.3: Download the code from the web page of the course and perform your own experiments. ♢



(a) **(b)**

Figure 3.3: Examples of the impact of extended double arithmetic. We repeat the example from Figure 3.1(b) with different implementations of the orientation test: **(a)** We evaluate $(q_x - p_x)(r_y - p_y)$ and $(q_y - p_y)(r_x - p_x)$ in extended double arithmetic, convert their values to double precision, and compare them. **(b)** We evaluate $sign((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x))$ in extended double arithmetic. For both experiments, we used $u_x = u_y = 2^{-53}$, the same as for the regular double precision examples in Figure 3.1. Note that there are no collinearities (yellow points) reported in **(b)**.

Choice of a Pivot Point: The orientation predicate is the sign of a three-by-three determinant and this determinant may be evaluated in different ways. In *float_orient* as defined above we use the point p as the *pivot*, i.e., we subtract the row representing the point p from the other rows and reduce the problem to the evaluation of a two-by-two determinant. Similarly, we may choose one of the other points as the pivot. Figure 3.2 displays the effect of the different choices of the pivot point on the example of Figure 3.1(b). The choice of the pivot makes a difference, but nonetheless the geometry remains non-trivial and sign reversals happen for all three choices. We will see in Lecture **??** that the center point w.r.t. the x-coordinate (or the y-coordinate) is the best choice for the pivot. However, no choice of pivot can avoid all sign errors.

Extended Double Precision: Some architectures, for example, Intel Pentium processors, offer IEEE extended double precision with a 64 bit mantissa in an 80 bit representation. Does this additional precision help? Not really, as the examples in Figure 3.3 suggest. One might argue that the number of misclassified points decreases, but the geometry of *float_orient* remains fractured and exploitable for failures similar to those that we develop below for double precision arithmetic.

3.2 Implementation of the Convex Hull Algorithm

We describe our `C++` reference implementation of our simple incremental algorithm. We give the details necessary to reproduce our results, for example, the exact parameter order in the predicate calls, but we omit details of the startup phase when we search for the initial three non-collinear points and the circular list data structure. We offer the full working source code based on CGAL [24], all the point data sets, and the images from the analysis on our companion web page `http://www.mpi-inf.mpg.de/~kettner/proj/NonRobust/` for reference.

We use our own plain conventional `C++` point type. Worth mentioning are equality comparison and lexicographic order used to find extreme points among collinear points in the startup phase.

```
struct Point { double x, y; };
```

The orientation test returns $+1$ if the points p, q, and r make a left turn, it returns zero if they are collinear, and it returns -1 if they form a right turn. We implement the orientation test as explained above with p as pivot point. Not shown here, but we make sure that all intermediate results are represented as 64 bit doubles and not as 80 bit extended doubles as it might happen, e.g., on Intel platforms.

```
int orientation( Point p, Point q, Point r) {
    return sign((q.x-p.x) * (r.y-p.y) - (q.y-p.y) * (r.x-p.x));
}
```

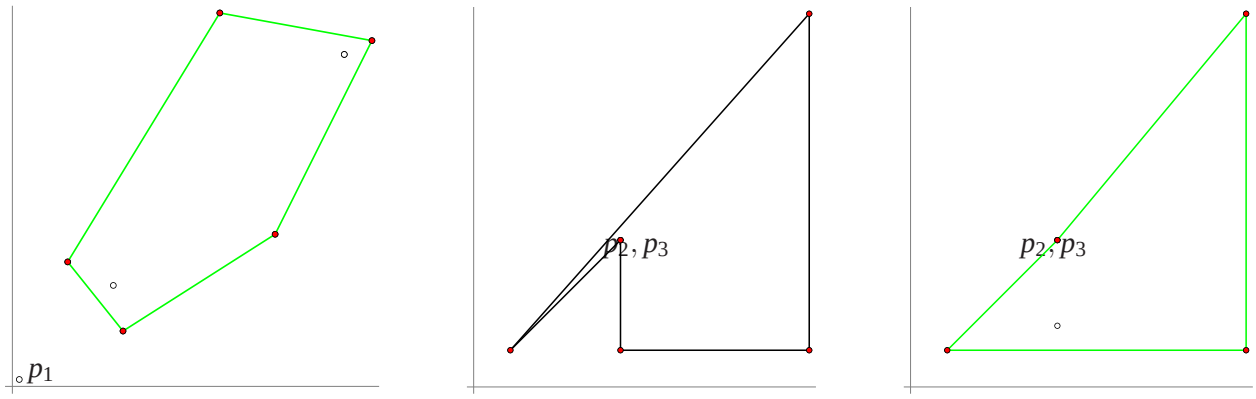
For the initial three non-collinear points we scan the input sequence and maintain its convex hull of up to two extreme points until we run out of input points or we find a third extreme point for the convex hull. From there on we scan the remaining points in our main `convex_hull` function as shown below.

The circular list used in our implementation is self explaining in its use. We assume a Standard Template Library (STL) compliant interface and extend it with circulators, a concept similar to STL iterators that allow the circular traversal in the list without any past-the-end position using the increment and decrement operators. In addition, we assume a function that can remove a range in the list specified by two non-identical circulator positions.

Our main `convex_hull` function shown below has a conventional iterator-based interface like other STL algorithms. It computes the extreme points in counterclockwise order of the 2d convex hull of the points in the iterator range `[first,last)`. It uses internally the circular list `hull` to store the current extreme points and copies this list to the `result` output iterator at the end of the function. It also returns the modified `result` iterator.

```
template <typename ForwardIter, typename OutputIter>
OutputIter incr_convex_hull( ForwardIter first, ForwardIter last,
                             OutputIter result)
{
    typedef std::iterator_traits<ForwardIter>    Iterator_traits;
    typedef typename Iterator_traits::value_type Point;
    typedef Circular_list<Point>                 Hull;
    typedef typename Hull::circulator            Circulator;

    Hull hull; // extreme points in counterclockwise (ccw) orientation
    // first the degenerate cases until we have a proper triangle
    first = find_first_triangle( first, last, hull);
    while ( first != last) {
        Point p = *first;
        // find visible edge in circular list of vertices of current hull
        Circulator c_source = hull.circulator_begin();
        Circulator c_dest = c_source;
        do {
            c_source = c_dest++;
            if ( orientation( *c_source, *c_dest, p) < 0) {
                // found visible edge, find ccw tangent
                Circulator c_succ = c_dest++;
                while ( orientation( *c_succ, *c_dest, p) <= 0)
                    c_succ = c_dest++;
```



(a) (b) (c)

Figure 3.4: Results of a convex hull algorithm using double-precision floating-point arithmetic with the coordinate axes drawn to give the reader a frame of reference. The implementation makes gross mistakes: In (a), the clearly extreme point p_1 is left out. In (b), the convex hull has a large concave corner and a (non-visible) self intersection. In (c), the convex hull has a clearly visible concave chain (and no self-intersection).

```
                // find cw tangent
                Circulator c_pred = c_source--;
                while ( orientation( *c_source, *c_pred, p) <= 0)
                    c_pred = c_source--;
                // c˙source is the first point visible, c˙succ the last
                if ( ++c_pred != c_succ)
                    hull.circular_remove( c_pred, c_succ);
                hull.insert( c_succ, p);
                break; // we processed all visible edges
            }
        } while ( c_source != hull.circulator_begin());
        ++first;
    }
    return std::copy( hull.begin(), hull.end(), result);
}
```

3.3 The Impact on the Convex Hull Algorithm

Let us next see the impact of approximate arithmetic on our convex hull algorithm. Figure 3.4 shows point sets (we give the numerical coordinates of the points below) and the respective convex hulls computed by the floating-point implementation of our algorithm. In each case the input points are indicated by small circles, the computed convex hull polygon is shown in green, and the alleged extreme points are shown as filled red circles. The examples show that the implementation may make gross mistakes. It may leave out points that are clearly extreme, it may compute polygons that are clearly non-convex, and it may even run forever.

We discuss in detail the output shown in Figure 3.4(b). We consider the points below. For improved readability, we will write numerical data in decimals. Such decimal values, when read into the machine, are internally represented by the nearest double. We have made sure that our data can be safely converted in this manner, i.e., conversion to binary and back to decimal is the identity operation. However, the C++ standard library does not provide sufficient guarantees and we offer additionally the binary data in little-endian format on the accompanying web page.

$$\begin{array}{lcl} p_1 & = & (24.00000000000005,\ \ 24.000000000000053) \\ p_2 & = & (24.0,\ \ 6.0\ ) \\ p_3 & = & (54.85,\ \ 6.0\ ) \\ p_4 & = & (54.850000000000357,\ 61.000000000000121) \\ p_5 & = & (24.000000000000068,\ 24.000000000000071) \\ p_6 & = & (\ 6.0,\ \ 6.0\ ). \end{array}$$

After the insertion of p_1 to p_4, we have the convex hull (p_1, p_2, p_3, p_4). This is correct. Point p_5 lies inside the convex hull of the first four points; but *float_orient*$(p_4, p_1, p_5) < 0$. Thus p_5 is inserted between p_4 and p_1 and we obtain $(p_1, p_2, p_3, p_4, p_5)$. However, this error is not visible yet to the eye, see Figure 3.5(a).

The point p_6 sees the edges (p_4, p_5) and (p_1, p_2), but does not see the edge (p_5, p_1). All of this is correctly determined by *float_orient*. Consider now the insertion process for point p_6. Depending on where we start the search for a visible edge, we will either find the edge (p_4, p_5) or the edge (p_1, p_2). In the former case, we insert p_6 between p_4 and p_5 and obtain the polygon shown in (b). It is visibly non-convex and has a self-intersection. In the latter case, we insert p_6 between p_1 and p_2 and obtain the polygon shown in (c). It is visibly non-convex.

Of course, in a deterministic implementation, we will see only one of the errors, namely (b). This is because in our sample implementation as given in the appendix, we have $L = (p_2, p_3, p_4, p_1)$, and hence the search for a visible edge starts at edge (p_2, p_3). In order to produce (c) with our implementation we replace the point p_2 by the point $p_2' = (24.0, 10.0)$. Then p_6 sees (p_2', p_3) and identifies (p_1, p_2', p_3) as the chain of visible edges and hence constructs (c).

3.4 Further Examples*

We give further examples for large effects of seemingly small errors. We give sequences p_1, p_2, p_3, ... of points such that the first three points form a counter-clockwise triangle (and *float_orient* correctly discovers this) and such that the insertion of some later point leads the algorithm astray (in the computations with *float_orient*). We also discuss how we arrived at the examples. All our examples involve nearly or truly collinear points; we will see in Lecture **??** that sufficiently non-collinear points do not cause any problems. Does this make the examples unrealistic? We believe not. Many point sets contain nearly collinear points or truly collinear points, which become nearly collinear by conversion to floating-point representation.

An extreme point is overlooked: Consider the set of points below. Figure 3.4(a) and 3.6(a) show the computed convex hull; a point that is clearly extreme is left out of the hull.

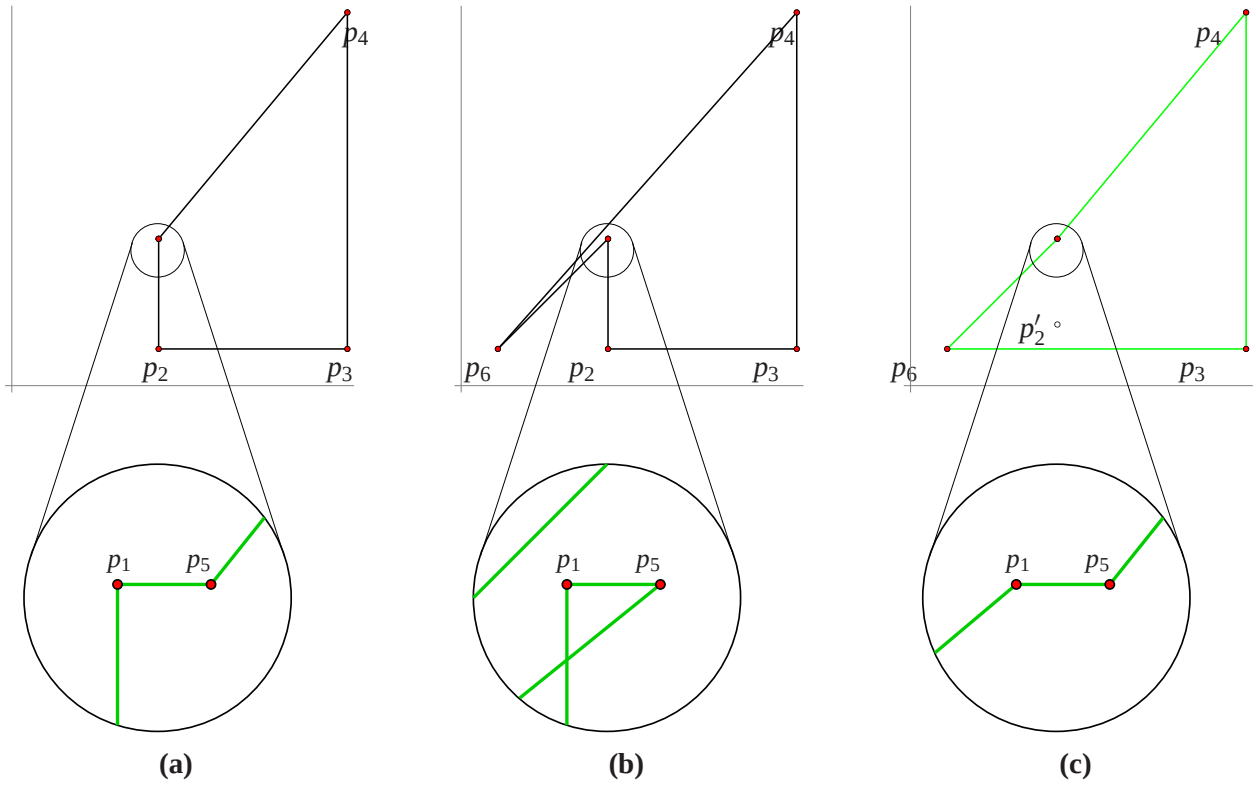


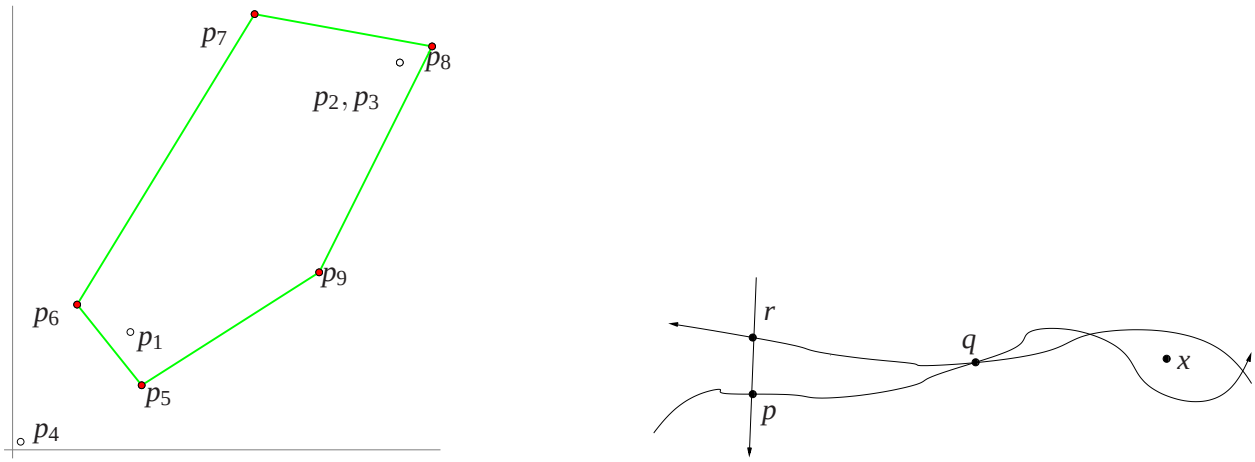
Figure 3.5: **(a)** The hull constructed after processing points p_1 to p_5. Points p_1 and p_5 lie close to each other and are indistinguishable in the upper figure. The magnified schematic view below shows that we have a concave corner at p_5. The point p_6 sees the edges (p_1, p_2) and (p_4, p_5), but does **not** see the edge (p_5, p_1). One of the former edges will be chosen by the algorithm as the chain of edges visible from p_6. Depending on the choice, we obtain the hulls shown in **(b)** or **(c)**. In **(b)**, (p_4, p_5) is found as the visible edge, and in **(c)**, (p_1, p_2) is found. We refer the reader to the text for further explanations. The figures show the coordinate axes to give the reader a frame of reference.

$$
\begin{array}{lcl}
p_1 & = & (\ 7.3000000000000194,\quad 7.3000000000000167\ ) \\
p_2 & = & (24.000000000000068,\quad 24.000000000000071\ ) \\
p_3 & = & (24.00000000000005,\quad 24.000000000000053\ ) \\
p_4 & = & (\ 0.50000000000001621,\ 0.50000000000001243) \\
p_5 & = & (\ 8,\ 4) \quad p_6 = (\ 4,\ 9) \quad p_7 = (15,27) \\
p_8 & = & (26,25) \quad p_9 = (19,11)
\end{array}
$$

$$
\begin{array}{l}
\mathit{float\_orient}(p_1, p_2, p_3) > 0 \\
\mathit{float\_orient}(p_1, p_2, p_4) > 0 \\
\mathit{float\_orient}(p_2, p_3, p_4) > 0 \\
\mathit{float\_orient}(p_3, p_1, p_4) > 0 \ (??)
\end{array}
$$

What went wrong? Let us look at the first four points. They lie almost on the line $y = x$, and *float_orient* gives the results shown above. Only the last evaluation is wrong, indicated by "(??)". Geometrically, these four evaluations say that p_4 sees no edge of the triangle (p_1, p_2, p_3). Figure 3.6(b) gives a schematic view of this impossible situation. The points $p_5, \ldots, p_9$ are then correctly identified as extreme points and are added to the hull. However, the algorithm never recovers from the error made when considering p_4 and the result of the computation differs drastically from the correct hull.

We next explain how we arrived at the instance above. Intuition told us that an example (if it exists at all) would be a triangle with two almost parallel sides and with a query point near the wedge defined by



(a) **(b)**

Figure 3.6: **(a)** The case of an overlooked extreme point: The point p_4 in the lower left corner is left out of the hull. **(b)** Schematic view indicating the impossible situation of a point outside the current hull and seeing no edge of the hull: x lies to the left of all sides of the triangle (p,q,r).

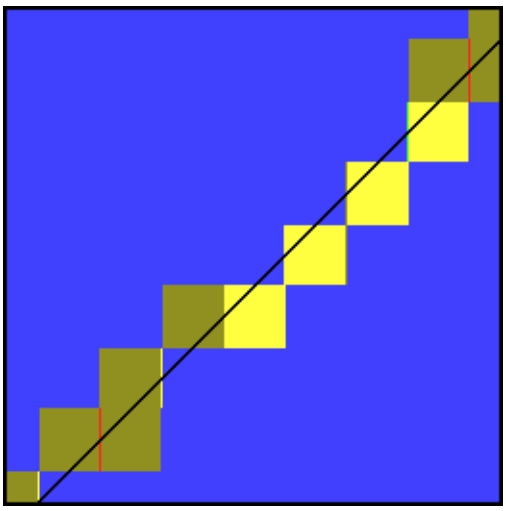
the two nearly parallel edges. In view of Figure 3.1 such a point might be mis-classified with respect to one of the edges and hence would be unable to see any edge of the triangle. So we started with the points used in Figure 3.1(b), i.e., $p_1 \approx (17,17)$, $p_2 \approx (24,24) \approx p_3$, where we moved p_2 slightly to the right so as to guarantee that we obtain a counter-clockwise triangle. We then probed the edges incident to p_1 with points p_4 in and near the wedge formed by these edges. Figure 3.7(a) visualizes the outcomes of the two relevant orientation tests. Each red pixel corresponds to a point that sees no edge. The example obtained in this way was not completely satisfactory, since some orientation tests on the initial triangle (p_1,p_2,p_3) were evaluating to zero.

We perturbed the example further, aided by visualizing *float_orient*(p_1,p_2,p_3), until we found the example shown in (b). The final example has the nice property that all possible *float_orient* tests on the first three points are correct. So this example is independent from any conceivable initialization an algorithm could use to create the first valid triangle. Figure 3.7(b) shows the outcomes of the two orientations tests for our final example.

A point outside the current hull sees all edges of the convex hull: Intuition told us that an example (if it exists) would consist of a triangle with one angle close to π and hence three almost parallel sides. Where should one place the query point? We first placed it in the extension of the three parallel sides and quite a distance away from the triangle. This did not work. The choice that worked is to place the point near one of the sides so that it could see two of the sides and "float-see" the third. Figure 3.8 illustrates this choice. A concrete example follows:

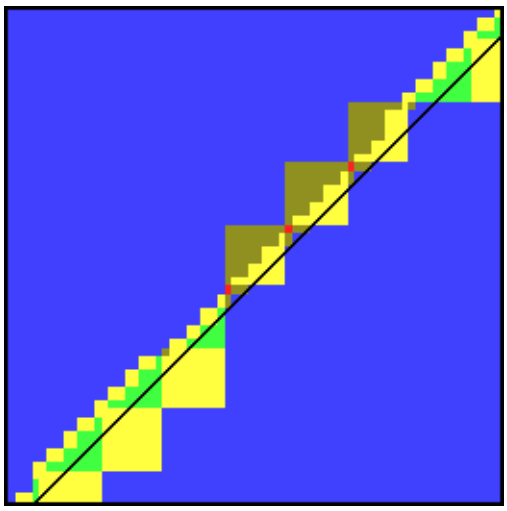
$$
\begin{array}{lcll}
p_1 & = & (\ 200.0, & 49.200000000000003) \\
p_2 & = & (\ 100.0, & 49.600000000000001) \\
p_3 & = & (-233.33333333333334, & 50.93333333333333\ ) \\
p_4 & = & (\ 166.66666666666669, & 49.333333333333336)
\end{array}
\qquad
\begin{array}{l}
\textit{float\_orient}(p_1,p_2,p_3) > 0 \\
\textit{float\_orient}(p_1,p_2,p_4) < 0 \\
\textit{float\_orient}(p_2,p_3,p_4) < 0 \\
\textit{float\_orient}(p_3,p_1,p_4) < 0\ (??)
\end{array}
$$

The first three points form a counter-clockwise oriented triangle and according to *float_orient*, the algorithm believes that p_4 can see all edges of the triangle. What will our algorithm do? It depends on the



p_1 : $(17.300000000000001, 17.300000000000001)$
p_2 : $(24.000000000000068, 24.000000000000071)$
p_3 : $(24.00000000000005, 24.000000000000053)$
p_4 : $(0.50000000000000711, 0.5)$

(a)



$(7.3000000000000194, 7.3000000000000167)$
$(24.000000000000068, 24.000000000000071)$
$(24.00000000000005, 24.000000000000053)$
$(0.50000000000000355, 0.5)$

(b)

Figure 3.7: The points (p_1, p_2, p_3) form a counter-clockwise triangle and we are interested in the classification of points $(x(p_4) + Xu_x, y(p_4) + Yu_y)$ with respect to the edges (p_1, p_2) and (p_3, p_1) incident to p_1. The extensions of these edges are indistinguishable in the pictures and are drawn as a single black line. The red points do not "float-see" either one of the edges. These are the points we were looking for. The points collinear with one of the edges are ocher, those collinear with both edges are yellow, those classified as seeing one but not the other edge are blue, and those seeing both edges are green. **(a)** Example starting from points in Figure 3.1. **(b)** Example that achieves "invariance" with respect to permutation of the first three points.

implementation details. If the algorithm first searches for an invisible edge, it will search forever and never terminate. If it deletes points on-line from L it will crash or compute nonsense depending on the details of the implementation of L.

3.5 Non-Continuous Functions

Why can our convex hull algorithm produce outputs that are grossly incorrect? The reason is the use of approximate arithmetic for computing non-continuous functions.

Three points are collinear or form a left or a right turn. This discontinuity is clearly visible in the analytical formula for the orientation function:

$$\mathsf{Orientation}(p,q,r) = \mathit{sign}((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x)).$$

It is the sign of a real numbers; the sign function is a step function and hence non-continuous.

Geometric algorithms are based on the laws of geometry; e.g., a point lies outside a convex polygon if and only if it can see one of its edges. Float-see is an incorrect implementation of "see" and hence points

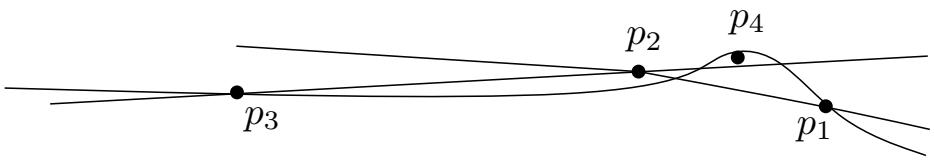


Figure 3.8: Schematic view of a point seeing all hull edges: The point p_4 sees all edges of the triangle (p_1, p_2, p_3).

are misclassified. Of course, only nearly collinear points are misclassified. So why doesn't our algorithm compute polygons that are close to the true hull? There are at least two reasons, why we should not expect this to be the case. First, a point far away from a convex polygon may be classified as lying inside the polygon (see Figure 3.6(a)). Second, a misclassified point may create a slightly non-convex polygon. This small error is amplified by later insertions (see Figure 3.4(b)).

Not only our primitive is non-continuous, the higher level geometric tasks are also tantamount to non-continuous functions. In the convex hull problem, we ask for the set of extreme points. This set is a non-continuous function of the input. For example, if a point that lies of an edge of the convex hull moves to the outside of the hull, the set increases in size. Figure 1.2 provides another example. Observe that the blue cylinder does not contribute to the output. However, as a result of shrinking it ever so slightly, a blue spot will appear in the center of the front side of the result. Since the result of the computation is a data structure that records the origin of each surface patch of the output, the output is again a non-continuous function of the input. Figure 1.2 was produced with the CAD-software Rhine3D. We asked the system to compute

$$(((s_1 \cap s_2) \cap c_2) \cap c_1).$$

If, the task is specified as

$$(((c_1 \cap c_2) \cap s_1) \cap s_2,$$

the software returns an error.

3.6 Geometric Computing vs. Numerical Analysis

We contrast geometric computing and numerical analysis. Algorithms in numerical analysis are also developed for the Real-RAM model of computation. The standard implementation of real numbers is floating point arithmetic. Numerical analysts are well aware of the pitfalls of floating point computation [**?**]. Forsythe's paper and many numerical analysis textbooks, see for example [17, page 9], contain instructive examples of how popular algorithms, e.g., Gaussian elimination, can fail when used with floating point arithmetic. These examples have played a guiding role in the development of robust numerical methods.

Many numerical algorithms are self-correcting, i.e., an error made at some time of the computation is remedied at a later time. In contrast, the algorithm of computational geometry are non-self-correcting as we have seen in our convex hull algorithms. Consider, for example, the Jacobi algorithm for solving a symmetric linear system $Ax = b$. We write A as $L + D + R$, where D is a diagonal matrix consisting of the diagonal entries of A, L is a lower triangular matrix consisting of the below-diagonal elements of A, and R is an upper triangular matrix consisting of the above-diagonal elements of A. Then $R = L^T$, since A is assumed to be symmetric.

LEMMA 8. *The Jacobi-iteration*

$$x_{k+1} = -D^{-1}(L+R)x_k + D^{-1}b$$

converges for every initial value x_0 against the solution of $Ax = b$, if A is strictly diagonally dominant, i.e.,

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}| \quad \text{for all } i.$$

Proof. We argue in two steps. We first assume that the iteration converges and show that the fixpoint of the iteration is the solution of $Ax = b$. In the second step, we show that the iteration converges.

Let x^* be a fixpoint of the iteration, i.e., $x^* = -D^{-1}(L+R)x^* + D^{-1}b$. Then

$$\begin{aligned} x^* = -D^{-1}(L+R)x^* + D^{-1}b &\iff Dx^* = -(L+R)x^* = b \\ &\iff (D+L+R)x^* = b \\ &\iff x^* = A^{-1}b. \end{aligned}$$

Let $G = -D^{-1}(L+R)$ and $c = D^{-1}b$. Then $x^* = Gx^* + c$. We next estimate the distance from x_k to the fixpoint x^*. We have

$$\begin{aligned} x_k - x^* &= Gx_{k-1} + c - (Gx^* + c) \\ &= G(x_{k-1} - x^*) \\ &= G^k(x_0 - x^*) \end{aligned}$$

and hence $\|x_k - x^*\| \leq \|G\|^k \|x_0 - x^*\|$ for any matrix norm. The infinity norm of G is less than one. Observe that the sum of the absolute values of the entries of the i-th row of G is $\sum_{j \neq i} \frac{|a_{ij}|}{|a_{ii}|}$ which is less than one since A is assumed to be diagonally dominant. □

Assume next, that we make an error in every iteration, i.e, we compute $x_{k+1} = Gx_k + c + e_k$ for some vector e_k with $\|e_k\| \leq \varepsilon$. Then

$$\begin{aligned} x_k &= Gx_{k-1} + c + e_{k-1} \\ &= G(G(x_{k-2} + c + e_{k-2}) + c + e_{k-1} \\ &= G^2 x_{k-2} + (G+I)c + Ge_{k-2} + Ie_{k-1} \\ &= \ldots \\ &= G^k x_0 + \sum_{1 \leq i \leq k} G^{i-1}c + \sum_{1 \leq i \leq k} G^{i-1}e_{k-i}. \end{aligned}$$

The first two terms converge against $x^* = A^{-1}b$; observe that we know already that the exact iteration converges against x^*. The norm of the last term is bounded by

$$\| \sum_{1 \leq i \leq k} G^{i-1}e_{k-i} \| \leq \sum_{1 \leq i \leq k} \|G\|^{i-1} \|e_{k-i}\| \leq \frac{\varepsilon}{1 - \|G\|}.$$

We conclude that the total error stays bounded. Moreover, any error made in a particular step is dampened by $\|G\|$ in any later step.

Many problems of numerical analysis are continuous functions from input to output. For example, the eigenvalues of a matrix are continuous functions of the entries of the matrix. In contrast, most problems in geometric computing are non-continuous functions.

However, numerical analysis also treats non-continuous problems. Linear system solving is a non-continuous function. The system $Ax = b$ has a solution if and only if b is in the span of the columns of A. Thus solving a linear system implicitly answers a yes-no question, namely whether b is in the span of the columns of A. This is, however, not the view of numerical analysis.

- Numerical analysis calls such problems ill-posed or at least ill-conditioned.
- We use arithmetic to make yes/no decisions, e.g., does p lie on ℓ or not?

3.7 Reliable (Geometric) Computing

What can we do? Before discussing solution, we clearly state the goal. We want reliable implementations. We call a program *reliable* if it does what it claims to do, if it comes with a guarantee. Guarantees come in different flavors.

(1) The strongest guarantee is to solve the problem for all inputs. For the example of the convex hull, this amounts to computing the extreme vertices of the hull for all sets S of input points. (2) A weaker, but still very strong, guarantee is to solve the problem approximately for all inputs. For example, we might compute a convex polygon P such that $P \subseteq U_\varepsilon(\operatorname{conv} S)$ and $\operatorname{conv} S \subseteq U_\varepsilon(P)$, where ε is a small positive constant, say $\varepsilon = 0.01$ and U_ε denotes ε-neighborhood. (3) Or we might give one of the guarantees above, but only if the coordinates of all input points are integers bounded by M, say $M = 2^{20}$. (4) Or we might guarantee that the program never crashes and always produces a polygon. Usually, this polygon is close (with an unspecified meaning of close) to the convex hull. (5) Or we guarantee nothing.

We find guarantees 4 and 5 too weak. We will teach you techniques for achieving guarantees 1 to 3. The techniques come in three kinds. The first approach is to ensure that the implementations of geometric predicates always return the correct result. It is known as the exact geometric computation (EGC) paradigm and has been adopted for the software libraries LEDA, CGAL and CORE LIBRARY [**?**, 24, 45, 40]. It implements a Real-RAM to the extent needed by a particular algorithm and is the approach mainly advocated in this book. The second approach is to perturb the input so that the floating-point implementation is guaranteed to produce the correct result on the perturbed input [35, 30]. We discuss this approach in Lecture **??**. The third approach is to change the algorithm so that it can cope with the floating-point implementation of its geometric predicates and still computes something meaningful. The definition of "meaningful" is crucial and difficult. This approach is problem-specific. We discuss it in Lecture **??**.

Reliability is our main concern, but efficiency is also of utmost importance. Efficiency comes in two flavors. On the theoretical side, we aim for algorithms with low asymptotic running time. On the practical side, we aim for programs that can compete with non-reliable alternatives.

3.8 Non-Solutions

Maybe, the reader finds that the problem should have an easy fix. We discuss two approaches that are frequently suggested, but definitely do not solve the problem.

The first approach is specific to the planar convex hull problem. A frequently heard reaction to the examples presented in this lecture is that all examples exploit the fact that the first few points are nearly collinear. If one starts with a "roundish" hull, or at least starts with a hull formed from the points of minimal and maximal x- and y- coordinates, the problem will go away. We have two answers to this suggestion: Firstly, neither way can cope with the situation that all input points are nearly collinear, and secondly, the

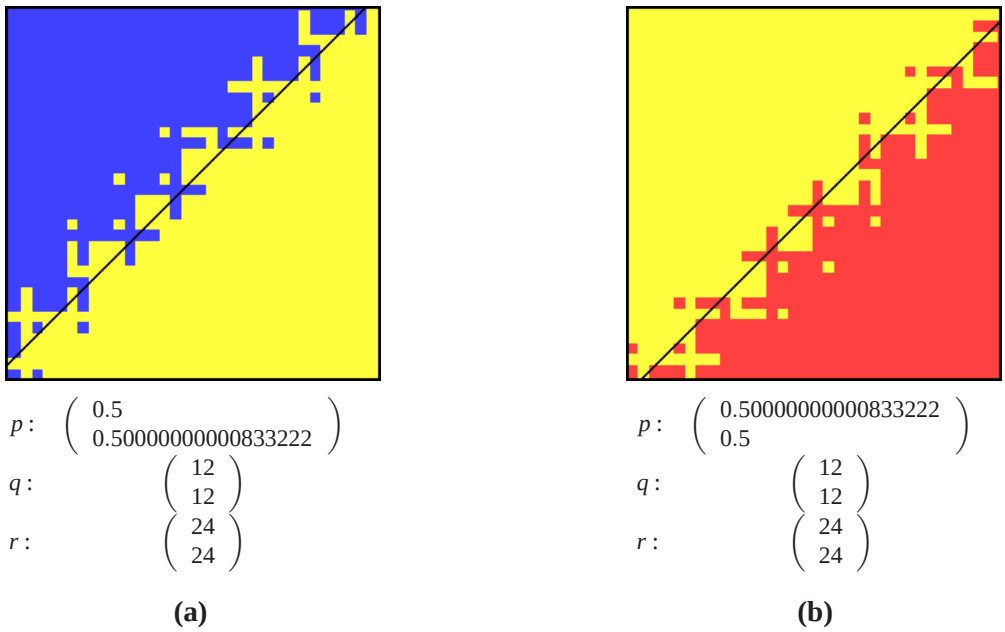


Figure 3.9: The effect of epsilon-tweaking: The figures show the result of repeating the experiment of Figure 3.1(a), but using an absolute epsilon tolerance value of $\varepsilon = 10^{-10}$, i.e., three points are declared collinear if *float_orient* returns a value less than or equal to 10^{-10} in absolute value. The yellow region of collinearity widens, but its boundary is as fractured as before. Figure (a) shows the boundary in the direction of the positive *y*-axis, and Figure (b) shows the boundary in the direction of the positive *x*-axis. The figures are color coded: Yellow (red, blue, resp.) pixels represent collinear (negative, positive, resp.) orientation. The black lines correspond to the lines $\mathsf{Orientation}(p,q,r) = \pm\varepsilon$.

example in Figure 3.5 falsifies this suggestion. Observe that we have a "roundish" hull after the insertion of the points p_1 to p_4 and then the next two insertions lead the algorithm astray. The example can be modified to start with points of minimal and maximal *x*- coordinates first, which we suggest as a possible course exercise.

Epsilon-tweaking is another frequently suggested and used remedy, i.e., instead of comparing exactly with zero, one compares with a small (absolute or relative) tolerance value epsilon. Epsilon-tweaking simply activates rounding to zero. In the planar hull example, this will make it more likely for points outside the current hull not to see any edges because of enforced collinearity and hence the failure that a point outside the hull will see no edge of the hull will still occur. In the examples of Section 3.1, the yellow band in the visualizations of collinear pixels becomes wider, but its boundary remains as fractured as it is in the comparison with zero, see Figure 3.9.

Another objection argues that the examples are unrealistic since they contain near collinear point triples or points very close together (actually the usual motivation for Epsilon-tweaking). Of course, the examples have to look like this, otherwise there would not be room for rounding errors. But they are realistic; firstly, practical experience shows it. Secondly, degeneracies, such as collinear point triples, are on purpose in many data sets, since they reflect the design intent of a CAD construction or in architecture. Representing such collinear point triples in double precision arithmetic and further transformations lead to rounding errors that turn these triples into close to collinear point triples. And thirdly, increasingly larger data sets increase the

chance to have a bad triple of points just by bad luck, and a single failure suffices to ruin the computation.

3.9 Where Do We Stand?

Where do we stand?

1. We have $O(n \log n)$ algorithms for computing convex hulls in the plane. These algorithms use only simple operations on points, namely lexicographic order and orientation.
2. If floating point arithemetic is used for implementing the orientation test, disaster can happen.

What can we do? We are in good shape as long as we can guarantee that lexicographic order and orientation is determined correctly. So it seems natural to restrict the coordinates to subsets of $\mathbb{R}$, where we can guarantee this. We will see in the next lecture that we can do so for $\mathbb{Q}$ and also for the set of floating point numbers. Later in the course, we will see how to do so for algebraic expressions and then algebraic numbers.

3.10 Historical Notes

Numerical analysts are well aware of the pitfalls of floating point computation [**?**]. Forsythe's paper and many numerical analysis textbooks, see for example [17, page 9], contain instructive examples of how popular algorithms, e.g., Gaussian elimination, can fail when used with floating point arithmetic. These examples have played a guiding role in the development of robust numerical methods.

The first implementations of geometric algorithms were either restricted the input so that integer arithmetic was sufficient or used floating point arithmetic as the implementation of real arithmetic. Many implementers reported that they found it very cumbersome to get their implementations to work. KM had the following experiences. He asked a student to implement an algorithm for Voronoi diagrams of line segments; see Figure **??**. The implementation worked only for a small number of examples. More seriously, the first implementations of geometric algorithms in LEDA would not work on all inputs; all of them would break for some inputs.

The literature contains a small number of documented failures due to numerical imprecision, e.g., Forrest's seminal paper on implementing the point-in-polygon test [25], Fortune's example for a variant of Graham's scan [**?**], Shewchuk's example for divide-and-conquer Delaunay triangulation [52], Ramshaw's braided lines [45, Section 9.6.2], Schirra's example for convex hulls [45, Section 9.6.1], and Mehlhorn and Näher's examples for the sweep line algorithm for line segment intersection and boolean operations on polygons [45, Sections 10.7.4 and 10.8.4]. This lecture is based on an article by Kettner et al. [42].

Software and hardware reliability goes much beyond geometric computing. A version of the Pentium chip contained an error in the division hardware [4]. The error costed Intel millions of dollars. The Ariane V rocket was lost because of a bug in the control software. FURTHER EXAMPLES IN Chee's write-up

3.11 Implementation Notes

3.12 Exercises

Exercise 0.4: Formulate more guarantees. ♢

Lecture 4

Number Types I

We will study arbitrary precision integers, rationals, fixed precision floating point numbers, and arbitrary precision floating point numbers. In later lectures, we will learn about algebraic expressions and general algebraic numbers. We start out with a short discussion of arbitrary precision integers and rationals. The bulk of the lecture will be about floating point numbers.

Floating point numbers are of the form

$$s \cdot m \cdot 2^e$$

where s is a *sign bit* (-1 or $+1$), m is a non-negative number called *mantissa* and e is an integer called *exponent*. The number of digits available for the mantissa is either fixed (all hardware floating point systems) or arbitrary (most software floating point systems). The exponent either comes from a fixed range (hardware floating point numbers and some software floating point systems) or is arbitrary (some software floating point systems). Already the first programmable computer offered floating point numbers. In 1938, Konrad Zuse completed the "Z1", the first programmable computer. It worked with 22-bit floating-point numbers having a 7-bit exponent, 15-bit significant (including one implicit bit), and a sign bit. The Z3, completed in 1941, implemented floating point arithmetic exceptions with representations for plus and minus infinity and undefined. The first commercial computers offering floating point arithmetic in hardware are Zuse's Z4 in 1950, followed by the IBM 704 in 1954. The IEEE standard 754-1985 [36] defines single and double precision floating point arithmetic which is implemented in hardware on all modern processors. Floating point arithmetic (hardware and software) is the workhorse for all scientific and geometric computations and therefore we need to study it carefully. The preceding statement concerning the importance of floating point computations seems to contradict the findings of Lecture 3. It does not. In the preceding lecture, we showed that a naive substitution of floating point arithmetic for real arithmetic does not work. In the course we will learn that the wise use of floating point arithmetic is one of cornerstones of reliable and efficient geometric computing. *We will teach you how to draw reliable conclusions from approximate arithmetic.*

4.1 Built-In Integers and Arbitrary Precision Integers

Hardware and programming languages provide fixed precision integer arithmetic, usually in signed and unsigned form. Let w be the word size of the machine and let $m = 2^w$. Most current workstations have $w = 32$ or $w = 64$. The unsigned integers consist of the integers between 0 and $m-1$ (both inclusive) and arithmetic is modulo m. The signed integers form an interval [`MININT`,`MAXINT`]. On most machines signed integers are represented in two's complement. Then $\texttt{MININT} = -2^{w-1}$ and $\texttt{MAXINT} = 2^{w-1} - 1$. An arithmetic

operation on signed integers may produce a result outside the range of representable numbers; one says that the operation underflows or overflows. The treatment of overflow and underflow is not standardized, in particular, it is not guaranteed that they lead to a runtime error, in fact they usually do not. For example, the addition `MAXINT` + `MAXINT` has result -2 on the KM's machine, since adding $011\ldots1$ to itself yields $11\ldots10$, which is the representation of -2 in two's complement.

Arbitrary integers are readily implemented in software, for example, in packages [31] and [37, Class BigInteger]. The running time of addition and subtraction is linear in the number of digits. All packages implement some form of fast integer multiplication. Depending on the method used, the running time of multiplication is $O(L^{\log 3})$ or $O(L\log L\log\log L)$, where L is the number of digits in the operands.

Exercise 0.5: The greatest common divisor of two integers x and y with $x \geq y \geq 0$ can be computed by the recursion $\texttt{GCD}(x,y) = x$ if $y = 0$ and $\texttt{GCD}(x,y) = \texttt{GCD}(y, x \mod y)$ if $y > 0$. Prove that the number of recursive calls is at most proportional to the length of y. Hint: Assume $x > y$ and let $x_0 = x$ and $x_1 = y$. For $i > 1$ and $x_{i-1} \neq 0$ let $x_i = x_{i-2} \mod x_{i-1}$. Let $x_k = 0$ be the last element in the sequence just defined. Relate this sequence to the gcd-algorithm. Show that $x_{k-1} > 0$ and $x_{i-2} \geq x_{i-1} + x_i$ for $i < k$. Conclude that x_{k-j} is at least as large as the j-th Fibonacci number. ♢

Exercise 0.6: The standard algorithm for multiplying two L-bit integers has running time $O(L^2)$. Karatsuba's method ([41]) runs in time $O(L^{\log 3})$. In order to multiply two numbers x and y it writes $x = x_1 \cdot 2^{L/2} + x_2$ and $y = y_1 \cdot 2^{L/2} + y_2$, where x_1, x_2, y_1, and y_2 have $L/2$ bits. Then it computes $z = (x_1 + x_2) \cdot (y_1 + y_2)$ and observes that $x \cdot y = x_1 \cdot y_1 \cdot 2^L + (z - x_1y_1 - x_2y_2) \cdot 2^{L/2} + x_2y_2$. In this way only three multiplications of $L/2$-bit integers are needed to multiply two L-bit integers. The standard algorithm requires four. ♢

4.2 Rational Numbers

A rational number is the quotient of two integers. Addition and multiplication of rational numbers are exact. A rational is normalized, if numerator and denominator are relatively prime. Normalization requires to find the greatest common divisor of numerator and denominator and two divisions to remove it. Normalization is fairly costly. However, one should be aware that some algorithms lead to non-normalized numbers and require normalization for efficiency. A prime example is Gaussian elimination. Consider Gaussian elimination of a 3×3 matrix.

$$\begin{pmatrix} a & b & e \\ c & d & f \\ g & h & i \end{pmatrix} \to \begin{pmatrix} a & b & e \\ 0 & d-b(c/a) & f-e(c/a) \\ 0 & h-b(g/a) & i-e(g/a) \end{pmatrix}$$

$$\to \begin{pmatrix} a & b & e \\ 0 & (ad-bc)/a & (af-ec)/a) \\ 0 & (ah-bg)/a & (ai-eg)/a \end{pmatrix}$$

$$\to \begin{pmatrix} a & b & e \\ 0 & (ad-bc)/a & (af-ec)/a) \\ 0 & 0 & (ai-eg)/a - \frac{(ah-bg)/a}{(ad-bc)/a}(af-ec)/a \end{pmatrix}$$

We now have a close look at the element in position $(3,3)$. We have:

$$(ai-eg)/a - \frac{(ah-bg)/a}{(ad-bc)/a}(af-ec)/a = \frac{(ai-eg)(ad-bc)-(ah-bg)(af-ec)}{a(ad-bc)}$$
$$= \frac{\text{all terms containing } a + (egbc-bgec)}{a(ad-bc)},$$

i.e., numerator and denominator contain the common factor a. If common factors are not cleared out in Gaussian elimination, the length of the numbers grows exponentially in the dimension of the matrix. If entries are kept in normalized form, Gaussian elimination is polynomial [**?**].

The use of rational arithmetic is inefficient and should be avoided.

4.3 Floating Point Numbers

We start out with a definition of binary floating point systems. We explain the representation of numbers and the key properties of floating point arithmetic. We move on to derive error bounds for the evaluation of expressions. We will use them extensively in the course: for optimized evaluations of geometric predicates in this lecture, as the basis for an efficient linear kernel (Lecture **??**), for the analysis of perturbation techniques (Lecture **??**), as the computational basis for the exact evaluation of algebraic expressions (Lecture **??**) and, more generally, arithmetic with algebraic numbers (Lecture **??**).

Hardware floating point arithmetic is standardized in the IEEE floating point standard [32, 33, 36]. A floating point number is specified by a sign s, a mantissa m, and an exponent e. The sign is $+1$ or -1. The mantissa consists of t bits $m_1, \ldots, m_t$, and e is an integer in the range $[e_{min}, e_{max}]$. The range of possible exponents contains zero and $e_{min} = -\infty$ and/or $e_{max} = +\infty$ is allowed.

TODO: does $e_{min} = -\infty$ really make sense? Then F is dense in $\mathbb{R}$ at 0. Check that all arguments stay valid. TODO

The number represented by the triple (s,m,e) is as follows:

- If $e_{min} < e \leq e_{max}$, the number is $s \cdot (1 + \sum_{1 \leq i \leq t} m_i 2^{-i}) \cdot 2^e$. This is called a *normalized* number.
- If $e = e_{min}$ then the number is $s \cdot \sum_{1 \leq i \leq t} m_i 2^{-i} 2^{e_{min}+1}$. This is called a *subnormal* number. Observe that the exponent is $e_{min}+1$. This is to guarantee that the distance of the largest subnormal number $(1-2^{-t})2^{e_{min}+1}$ and the smallest normalized number $12^{e_{min}+1}$ is small.
- In addition, there are the special numbers $-\infty$ and $+\infty$ and a symbol NaN which stands for not-a-number. It is used as an error indicator, e.g., for the result of a division by zero.

Double precision floating point numbers are represented in 64 bits. One bit is used for the sign, 52 bits for the mantissa ($t = 52$) and 11 bits for the exponent. These 11 bits are interpreted as an integer $f \in [0..2^{11}-1] = [0..2047]$. The exponent $e = f - 1023$; $f = 2047$ is used for the special values and hence $e_{min} = -1023$ and $e_{max} = 1023$. The rules for $f = 2047$ are:

- If all m_i are zero and $f = 2047$ then the number is $+\infty$ or $-\infty$ depending on s.
- In $f = 2047$ and some m_i is non-zero, the triple represents NaN (= not a number).

Let $F = F(t, e_{min}, e_{max})$ be the set of real numbers (including $+\infty$ and $-\infty$) that can be represented as above. A number in F is called *representable*, a number in $\mathbb{R} \setminus F$ is called *non-representable*. Observe that for normalized numbers, the leading 1 is not stored. It is sometimes called the hidden bit. The largest positive representable number (except for ∞) is $\texttt{MAX}_F = (2 - 2^{-t}) \cdot 2^{e_{max}}$, the smallest positive representable number is ???? $\texttt{MIN}_F = 2^{-t} \cdot 2^{e_{min}+1} = 2^{-t+e_{min}+1}$, and the smallest positive normalized representable number is $\texttt{MINNORM}_F = 1 \cdot 2^{e_{min}+1} = 2^{e_{min}+1}$. We define the *normal range* of F as

$$[-\texttt{MAX}_F, -\texttt{MINNORM}_F] \cup [\texttt{MINNORM}_F, \texttt{MAX}_F]$$

??? and the *subnormal range* as the open interval $(-\texttt{MINNORM}_F, +\texttt{MINNORM}_F)$. Observe that 0 lies in the subnormal range. The *range* of F is the closed interval $[-\texttt{MAX}_F, +\texttt{MAX}_F]$. We require $\texttt{MINNORM}_F \leq 2-t$. This guarantees $\texttt{MIN}_F^{1/2} \geq \texttt{MINNORM}_F$.

Exercise 0.7: Specialize the definitions above to double precision floating point numbers. ♢

4.3.1 Rounding

F is a discrete subset of $\mathbb{R}$. For any real x, let[1] $\mathit{flu}(x)$ be the smallest floating point number greater then or equal to x and let $\mathit{fld}(x)$ be the largest floating point number smaller than or equal to x, i.e.,

$$\mathit{flu}(x) = \min\{z \in F \mid x \leq z\} \quad \text{and} \quad \mathit{fld}(x) = \max\{z \in F \mid z \leq x\}.$$

If x is representable, $\mathit{flu}(x) = \mathit{fld}(x) = x$. If $x > \texttt{MAX}_F$, $\mathit{flu}(x) = +\infty$ and $\mathit{fld}(d) = \texttt{MAX}_F$, and if $0 \leq x \leq \texttt{MIN}_F$, $\mathit{flu}(x) = \texttt{MIN}_F$ and $\mathit{fld}(x) = 0$.

Rounding a real number x yields $\mathit{flu}(x)$ or $\mathit{fld}(x)$. There are several rounding modes: *Rounding away from zero* yields $\mathit{flu}(x)$ for a nonnegative x and $\mathit{fld}(x)$ for a negative x. *Rounding towards zero* yields $\mathit{fld}(x)$ for a nonnegative x and $\mathit{flu}(x)$ for a negative x. *Rounding to nearest* yields $\mathit{flu}(x)$ or $\mathit{fld}(x)$ depending on which number is closer to x. If both numbers are equally close, i.e., $x = (\mathit{flu}(x) + \mathit{fld}(x))/2$, the result of the rounding has an even last bit in the mantissa. The latter rule makes the rounding deterministic; also there is empirical evidence [**?**] that "rounding to even" in the case of ties has superior computational properties. Rounding to nearest is the default rounding mode in the IEEE standard and we follow this convention. We use $\mathrm{fl}(x)$ to denote the result of rounding x to the nearest floating point number. If $x > \texttt{MAX}_F$, we define $\mathrm{fl}(x) = \infty$, and if $x < -\texttt{MAX}_F$, we define $\mathrm{fl}(x) = -\infty$. The following theorem states that rounding of numbers in the normal range incurs a small relative error.

THEOREM 9. *If $x \in \mathbb{R}$ lies in the normal range,*

$$\max(|x - \mathit{flu}(x)|, |x - \mathit{fld}(d)|) \leq 2^{-t} \min(|x|, |\mathit{fld}(x)|, |\mathit{flu}(x)|) \tag{1}$$

and

$$|x - \mathit{fl}(x)| \leq 2^{-t-1} \min(|x|, |\mathit{fl}(x)|). \tag{2}$$

If $|x| > \texttt{MAX}_F$, $|x - \mathit{fl}(x)| \leq 2^{-t-1} |\mathit{fl}(x)|$.

[1] *flu* stands for "float-up" and *fld* stands for "float-down".

Proof. We may assume that x is positive. Then $\texttt{MINNORM}_F \le x \le \texttt{MAX}_F$ and hence $x = m2^e$ for some m and e with $1 \le m < 2$ and $e_{min} \le e \le e_{max}$. If $e = e_{max}$, we have in addition $m \le 2 - 2^{-t}$. The distance between adjacent floating point numbers with exponent e is 2^{-t+e}. Also, $\min(|x|, |fld(x)|, |flu(x)|) \ge 2^e$. Thus

$$\max(|x - flu(x)|, |x - fld(d)|) \le 2^{-t+e} \le 2^{-t} \min(|x|, |fld(x)|, |flu(x)|).$$

The second claim follows from $|x - \mathrm{fl}(x)| \le 2^{-t-1+e}$. Finally, if $|x| > \texttt{MAX}_F$, $|\mathrm{fl}(x)| = \infty$ and this implies the third claim. □

For subnormal numbers, the relative error of rounding may be arbitrarily large. For example for, $x = \texttt{MIN}_F/2$ we have $\mathrm{fl}(x) = 0$ and hence $|\mathrm{fl}(x) - x| = x$. Relative to x, the error is 1, and relative to $\mathrm{fl}(x)$, the error in $+\infty$. However, the absolute error is bounded.

LEMMA 10. *Let $x \in \mathbb{R}$ be in the subnormal range. Then*

$$|x - fl(x)| \le 2^{-t-1+e_{min}+1} = 2^{-t-1}\texttt{MINNORM}_F.$$

Proof. The distance between subnormal floating point numbers is $2^{-t+e_{min}+1}$. □

The quantities 2^{-t} and 2^{-t-1} are so important that they deserve a name. We call $\varepsilon = 2^{-t}$ the *precision* of the floating point system and $\mathbf{u} = 2^{-t-1}$ the *unit of roundoff*.

THEOREM 11 (Quality of Rounding Function). *For any real x,*

$$|x - fl(x)| \le \mathbf{u} \max(|fl(x)|, \texttt{MINNORM}_F) \tag{3}$$

4.3.2 Arithmetic on Floating Point Numbers

Arithmetic on floating point numbers is only approximate; it incurs roundoff error. Although floating point arithmetic is inherently inexact, the IEEE standard guarantees that the result of any arithmetic operation is close to the exact result, frequently as close as possible. It is important to distinguish between mathematical operations and their floating point implementations. We use $\oplus$, $\ominus$, $\odot$, and $\oslash$ for the floating point implementations of addition, subtraction, multiplication and division, respectively. Only in this lecture, we use $^{1/2}$ for the square-root operation and $\sqrt{\ }$ for its floating point implementation. Generally, we use $\tilde{\circ}$ for the floating point implementation of $\circ$. *The floating point implementations of the operations $+$, $-$, $\cdot$, $/$, and $^{1/2}$ yield the best possible result.* This is an axiom of floating point arithmetic.

DEFINITION 1. *If $x, y \in F$ and $\circ \in \{+, -, \cdot, /\}$ then*

$$x \,\tilde{\circ}\, y = fl(x \circ y)$$

and

$$\sqrt{x} = fl(x^{1/2}).$$

As an immediate consequence of this definition and Theorem 11 we obtain:

THEOREM 12 (Error Bound for Single Operations). *If $x,y \in F$ and $\circ \in \{+,-,\cdot,/\}$ then*

$$|x \tilde{\circ} y - x \circ y| \leq \mathbf{u} \max(|x \tilde{\circ} y|, \mathtt{MINNORM}_F) \tag{4}$$

$$|x \circ y| \leq (1+\mathbf{u}) \max(|x \tilde{\circ} y|, \mathtt{MINNORM}_F) \tag{5}$$

$$\left|\sqrt{x} - x^{1/2}\right| \leq \mathbf{u} \min(x^{1/2}, \sqrt{x}). \tag{6}$$

$$x^{1/2} \leq (1+\mathbf{u})\sqrt{x}. \tag{7}$$

$$\sqrt{x} \leq (1+\mathbf{u})x^{1/2}. \tag{8}$$

Proof. Inequality (4) follows immediately from Theorem 11 and inequality (5) is a short calculation.

$$|x \circ y| \leq |x \circ y - x \tilde{\circ} y| + |x \tilde{\circ} y| \leq \mathbf{u} \max(|x \tilde{\circ} y|, \mathtt{MINNORM}_F) + |x \tilde{\circ} y| \leq (1+\mathbf{u}) \max(|x \tilde{\circ} y|, \mathtt{MINNORM}_F).$$

Inequality (6) certainly holds if $x = 0$ and hence $x^{1/2} = \sqrt{x} = 0$ or if $x = +\infty$ and hence $x^{1/2} = \sqrt{x} = \infty$. If $x > 0$, and hence $x \geq MINF$, we have $x^{1/2} \geq \mathtt{MINNORM}_F$ and hence $\sqrt{x} \geq \mathtt{MINNORM}_F$. Inequality (6) then follows from (2). Inequalities (7) and (8) are immediate consequences of (6). □

Observe that the floating point operations $\oplus$, $\ominus$, $\odot$, $\oslash$ and $\sqrt{\ }$ must return the exact result if this is representable. This is too much to ask for more complex operation, for example logarithms or exponentials. There one requires that the implementation either returns the exact result (if representable) or one of the two adjacent floating point numbers.

We will also need the following properties.

(a) 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 $a_1 \odot a_2 \leq b_1 \odot b_2$.

(b) Multiplication by a power of two incurs no roundoff error, i.e., if $a \in F$ is a power of two, $b \in F$ and $2a$ and ab are in the range of F, then $a \oplus a = 2 \cdot a$ and $a \odot b = a \cdot b$.

(c) If $a+b$ is representable, then $a \oplus b = a+b$ and if ab is representable $a \odot b = ab$.

(d) If $x \in \mathbb{N}$, $x < 2^{t+1}$ and $t \leq e \leq e_{max}$, then $x2^e$ is representable.

The IEEE standard also defines the results for "strange" combinations of arguments. Of course, division by zero yields NaN. Also, if one of the arguments of an addition is NaN or the addition has no defined result, e.g., $-\infty + \infty$, then the result is NaN.

Exercise 0.8: Let $a,b \in F$ with $\frac{1}{2} \leq \frac{a}{b} \leq 2$. Show that $a \ominus b = a - b$. This was first observed by Sterbenz [53]. ♢

Exercise 0.9: Assume for this exercise that point coordinates are doubles in $[1/2,1]$. Show

- Orientation$(p,q,r) = 0$ implies *float_orient*$(p,q,r) = 0$.
- *float_orient*$(p,q,r) \neq 0$ implies Orientation$(p,q,r) =$ *float_orient*(p,q,r).
- What does this mean for a figure such as Figure **??**?
- Can you find examples as in Section 3.3 when point coordinates are restricted to doubles in $[1/2,1]$?

♢

4.3.3 Floating Point Integers

We briefly discuss the use of double precision hardware floating point arithmetic for 53-bit integer arithmetic. Let us call an integer a *floating point integer* if it belongs to the interval $I := [-(2^{53}-1)..2^{53}-1]$. The numbers in I can be represented as double precision floating point numbers. Consider a non-negative integer $x = \sum_{0\le i\le 53} x_i 2^i \in I$. If $x = 0$, x is a double. If $x > 0$, let j be maximal such that $x_j \neq 0$. Then $x = (1+\sum_{1\le i\le j} x_{j-i}2^{-i})2^j$ and hence x is a double. Double precision floating point arithmetic on numbers in I is exact.

LEMMA 13. *Assume $x \in I$, $y \in I$ and $x \circ y \in I$ where $\circ \in \{+,-,\cdot\}$. Then $x \tilde{\circ} y = x \circ y$.*

Lemma 13 is useful if points have integer Cartesian or homogeneous coordinates of bounded size.

LEMMA 14. *Assume that points have integral Cartesian coordinates. Then*

$$(b_x - a_x)\cdot(c_y - a_y) - (b_y - a_y)\cdot(c_x - a_x)$$

is computed without roundoff error if the absolute value of all coordinates is bounded by $2^L - 1$, where $2(L+1)+1 \le 53$.

Proof. The absolute value of the expression is strictly bounded by

$$(2^L + 2^L)\cdot(2^L + 2^L) + (2^L + 2^L)\cdot(2^L + 2^L) = 2^{2L+3}.$$

Thus if $2L+3 \le 53$, the value is in I and hence computed correctly. □

Exercise 0.10: Prove an analogous lemma for the orientation predicate and points with integer homogeneous coordinates and for the side-of-circle predicate and points with integer Cartesian or homogeneous coordinates. ♢

Built-in 32-bit integer arithmetic can only handle integers whose absolute value is bounded by $2^{31}-1$. So it supports the orientation predicate for integer coordinates with at most 14 bits. In contrast, doubles support the orientation predicate for integer coordinates with up to 25 bits. One may paraphrase this observation as *doubles are the better ints*.

4.4 An Optimized Evaluation Order for the Orientation Predicate

TODO, Chee's note are a good source.

4.5 An Error Analysis for Arithmetic Expressions

We study the evaluation of simple arithmetic operations in floating point arithmetic. Any real is an arithmetic expression and if A and B are arithmetic expression, then are $A+B$, $A-B$, $A\cdot B$, and $A^{1/2}$. The latter assumes that the value of A is non-negative. For an arithmetic expression E, let $\tilde{E}$ the result of evaluating E with floating point arithmetic. We want to bound

$$\left|\tilde{E} - E\right|.$$

E	condition	$\tilde{E}$	m_E	d_E
a	a is non-representable	$\mathrm{fl}(a)$	$\max(\mathtt{MINNORM}_F, \lvert\mathrm{fl}(a)\rvert)$	1
a	a is representable	a	$\max(\mathtt{MINNORM}_F, \lvert a\rvert)$	0
$A+B$		$\tilde{A}\oplus\tilde{B}$	$m_A \oplus m_B$	$1+\max(d_A,d_B)$
$A-B$		$\tilde{A}\ominus\tilde{B}$	$m_A \oplus m_B$	$1+\max(d_A,d_B)$
$A\cdot B$		$\tilde{A}\odot\tilde{B}$	$\max(\mathtt{MINNORM}_F, m_A \odot m_B)$	$1+d_A+d_B$
$A^{1/2}$	$\tilde{A} < \mathbf{u}m_A$	0	$2^{(t+1)/2}\sqrt{m_A}$	$2+d_A$
$A^{1/2}$	$\tilde{A} \geq \mathbf{u}m_A$	$\sqrt{\tilde{A}}$	$\max(\sqrt{\tilde{A}}, m_A \oslash \sqrt{\tilde{A}})$	$2+d_A$

Table 4.1: The recursive definition of m_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 m_E and ind_E; $\oplus$, $\ominus$, $\odot$, and $\oslash$ denote the floating point implementations of addition, subtraction, and multiplication, and $\sqrt{\ }$ denotes the floating point implementation of the square-root operation. Observe that $m_E = \infty$ if either $m_A = \infty$ or $m_B = \infty$.

Such a bound can be used to draw a reliable conclusion about the sign of an expression, because

$$\left|\tilde{E} - E\right| \leq B \quad \text{and} \quad \left|\tilde{E}\right| > B \quad \text{implies} \quad \operatorname{sign}(E) = \operatorname{sign}(\tilde{E}).$$

This observation is very important. It shows that we may be able to determine the sign of an expression with floating point arithmetic although it might be impossible to determine its value with floating point arithmetic.
We will derive a bound of the form

$$\left|E - \tilde{E}\right| \leq B \quad \text{where} \quad B = ((1+\mathbf{u})^{d_E} - 1) \cdot m_E \leq (d_E + 2) \odot \mathbf{u} \odot m_E,$$

and d_E and m_E are defined in Table 4.1. The intuitive interpretation is as follows: m_E upper bounds $\tilde{E}$ and d_E measures the levels of rounding. The operators $+$, $-$, and $\cdot$ introduce one additional level of rounding, the square-root-operator accounts for two levels. In an addition, the arguments contribute the maximum of their levels, and in a multiplication, the arguments contribute their sum. Each level of rounding increases the range of uncertainty by a multiplicative factor of $1+\varepsilon$. The subtraction of a -1 reflects the fact that we are interested in the error.

Before we establish the error bound, we will show that $((1+\mathbf{u})^d - 1)$ is approximately equal to $d\mathbf{u}$ and we will also give an example.

LEMMA 15. *If $d \leq \sqrt{1/\mathbf{u}} - 1$ then $((1+\mathbf{u})^d - 1) \leq (d+1)\mathbf{u}$. For all d, $((1+\mathbf{u})^d - 1) \geq d\mathbf{u}$.*

Proof. We have

$$(1+\mathbf{u})^d - 1 = \sum_{1 \leq i \leq d} \binom{d}{i} \mathbf{u}^i \leq \sum_{i \geq 1} (d \cdot \mathbf{u})^i = d\mathbf{u}/(1-d\mathbf{u}).$$

Next observe that $d\mathbf{u}/(1-d\mathbf{u}) \leq (1+d)\mathbf{u}$ iff $d/(1-d\mathbf{u}) \leq (1+d)$ iff $d \leq d+1-d^2\mathbf{u}-d\mathbf{u}$ iff $d(d+1) \leq 1/\mathbf{u}$. This is certainly the case when $(d+1)^2 \leq \mathbf{u}$ or $d \leq \sqrt{1/\mathbf{u}} - 1$. The lower bound follows immediately from the expansion of $(1+\mathbf{u})^d$. □

The condition $d \leq \sqrt{1/\mathbf{u}} - 1$ is hardly constraining. For $\mathbf{u} = 2^{-53}$, it amounts to $d < 2^{26.5}$. As an example, we use the orientation predicate for points a, b, and c given by their Cartesian coordinates. Then

$$\mathsf{Orientation}(a,b,c) = (b_x - a_x)\cdot(c_y - a_y) - (b_y - a_y)\cdot(c_x - a_x).$$

We compute the d-value of this expression. The degree of any argument is one, the degree of $(b_x - a_x)$ is 2, the degree of $(b_x - a_x)\cdot(c_y - a_y)$ is 5 and the degree of the entire expression is 6. We conclude that the error of evaluating $\mathsf{Orientation}(a,b,c)$ with floating point arithmetic is at most

$$7\cdot\mathbf{u}\cdot m_{\mathsf{Orientation}(a,b,c)}.$$

This bound is worth to be formulated as a Lemma.

LEMMA 16. *If points are given by their Cartesian coordinates and the orientation predicate is computed by the formula above, the roundoff error in a floating point evaluation is bounded by* $7\cdot\mathbf{u}\cdot m_{\mathsf{Orientation}(a,b,c)}$ *(*$8\odot\mathbf{u}\odot m_{\mathsf{Orientation}(a,b,c)}$*).*

Lemma 16 leads to the following code for evaluation of the orientation predicate. We assume that the Cartesian coordinates belong to some number type *NT* for which we have exact arithmetic available. We first convert all coordinates to a floating point number and then evaluate the orientation precision with floating point arithmetic. If the absolute value of the floating point result is sufficiently big, we return its result. If it is too small we resort to exact computation.

```
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) > 8 * uu * mes ) return (sign Etilde);
// exact evaluation
NT E = (qx - px)*(ry - py) - (qy - py)*(rx - px);
return sign E;
}
```

Exercise 0.11: Assume that a point p is given by its homogeneous coordinates (px, py, pw). Assuming $\mathrm{sign}(aw\cdot bw\cdot cw) = 1$, we have

$$\mathsf{Orientation}(a,b,c) = aw\cdot(bx\cdot cy - by\cdot cx) - bw\cdot(ax\cdot cy - ay\cdot cx) + cw\cdot(ax\cdot by - ay\cdot bx).$$

Compute the d-value of this expression. ♢

Exercise 0.12: Assume that for i, $1 \leq i \leq 8$, x_i is an integer with $|x_i| \leq 2^{20}$. Evaluate the expression $((x_1 + x_2)\cdot(x_3 + x_4))\cdot x_5 + (x_6 + x_7)\cdot x_8$ with double precision floating point arithmetic. Derive a bound for the maximal difference between the exact result and the computed result. ♢

THEOREM 17 (Error Bound for Arithmetic Expressions). *If* m_E *and* d_E *are computed according to Table 4.1 then*

$$m_E \geq \mathtt{MINNORM}_F \quad \textit{and} \quad m_E \geq \left|\tilde{E}\right| \quad \textit{and} \quad \left|\tilde{E} - E\right| \leq ((1+\mathbf{u})^{d_E} - 1)\cdot m_E$$

Proof. We use induction on the structure of the expression E. The claims $m_E \geq \texttt{MINNORM}_F$ and $m_E \geq \left|\tilde{E}\right|$ follow immediately from the table and the monotonicity of floating point arithmetic. For the third claim we have to work harder. We use induction on the structure of E and start by observing that the claim is obvious if $m_E = \infty$. The base case is obvious. If $E = a$ and a is representable, $\tilde{E} = E$. If a is non-representable we invoke Theorem 11.

For the induction step we make a case distinction according to the operation combining A and B. Assume first that $E = A + B$. Then

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

Inequality (4) bounds the first term by $\mathbf{u} \max(\left|\tilde{A} \oplus \tilde{B}\right|, \texttt{MINNORM}_F)$. Next observe that

$$\max(\left|\tilde{A} \oplus \tilde{B}\right|, \texttt{MINNORM}_F) \leq \max(m_A \oplus m_B, \texttt{MINNORM}_F) = \max(m_E, \texttt{MINNORM}_F) = m_E$$

by monotonicity of floating point arithmetic and since $m_E \geq \texttt{MINNORM}_F$. For the other two terms we use the induction hypothesis to conclude

$$\begin{aligned}
\left|\tilde{A} - A\right| + \left|\tilde{B} - B\right| &\leq ((1+\mathbf{u})^{d_A} - 1) \cdot m_A + ((1+\mathbf{u})^{d_B} - 1) \cdot m_B \\
&\leq ((1+\mathbf{u})^{\max(d_A, d_B)} - 1) \cdot (m_A + m_B) \\
&\leq ((1+\mathbf{u})^{\max(d_A, d_B)} - 1) \cdot (1+\mathbf{u}) \cdot m_E \qquad \text{by inequality (5).}
\end{aligned}$$

Putting the two bounds together we obtain:

$$\begin{aligned}
\left|\tilde{E} - E\right| &\leq [\mathbf{u} + ((1+\mathbf{u})^{\max(d_A, d_B)} - 1) \cdot (1+\mathbf{u})] \cdot m_E \\
&= [(1+\mathbf{u})^{1+\max(d_A, d_B)} - 1] \cdot m_E.
\end{aligned}$$

Subtractions are treated completely analogously.

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

$$\left|\tilde{E} - E\right| = \left|\tilde{A} \odot \tilde{B} - A \cdot B\right| \leq \left|\tilde{A} \odot \tilde{B} - \tilde{A} \cdot \tilde{B}\right| + \left|\tilde{A} \cdot \tilde{B} - A \cdot \tilde{B}\right| + \left|A \cdot \tilde{B} - A \cdot B\right|.$$

Inequality (4) and monotonicity of floating point arithmetic bound the first term by

$$\mathbf{u} \max(\left|\tilde{A} \odot \tilde{B}\right|, \texttt{MINNORM}_F) \leq \mathbf{u} \max(m_A \odot m_B, \texttt{MINNORM}_F) = \mathbf{u} m_E.$$

For the second term we use the induction hypothesis to conclude

$$\begin{aligned}
\left|\tilde{A} \cdot \tilde{B} - A \cdot \tilde{B}\right| &= \left|\tilde{A} - A\right| \cdot \left|\tilde{B}\right| \\
&\leq ((1+\mathbf{u})^{d_A} - 1) \cdot m_A \cdot m_B \\
&\leq ((1+\mathbf{u})^{d_A} - 1) \cdot (1+\mathbf{u}) \cdot \max(m_A \odot m_B, \texttt{MINNORM}_F) \qquad \text{by inequality (5)} \\
&= ((1+\mathbf{u})^{d_A} - 1) \cdot (1+\mathbf{u}) \cdot m_E,
\end{aligned}$$

and for the third term we conclude similarly

$$\begin{aligned}
\left|A \cdot \tilde{B} - A \cdot B\right| &= |A| \cdot \left|\tilde{B} - B\right| \\
&\leq (\left|\tilde{A}\right| + \left|\tilde{A} - A\right|) \cdot \left|\tilde{B} - B\right| \\
&\leq (1+\mathbf{u})^{d_A} \cdot m_A \cdot ((1+\mathbf{u})^{d_B} - 1) \cdot m_B \\
&\leq (1+\mathbf{u})^{1+d_A} \cdot ((1+\mathbf{u})^{d_B} - 1) \cdot \max(m_A \odot m_B, \texttt{MINNORM}_F) \qquad \text{by inequality (5)} \\
&= (1+\mathbf{u})^{1+d_A} \cdot ((1+\mathbf{u})^{d_B} - 1) \cdot m_E
\end{aligned}$$

Putting the three bounds together, we obtain

$$\begin{aligned}\left|\tilde{E}-E\right| &\le (\mathbf{u}+(1+\mathbf{u})\cdot((1+\mathbf{u})^{d_A}-1)+(1+\mathbf{u})^{1+d_A}\cdot((1+\mathbf{u})^{d_B}-1))m_E\\ &= (\mathbf{u}+(1+\mathbf{u})^{d_A+1}-1-\mathbf{u}+(1+\mathbf{u})^{1+d_A+d_B}-(1+\mathbf{u})^{1+d_A})m_E\\ &= ((1+\mathbf{u})^{1+d_A+d_B}-1)m_E\end{aligned}$$

and the induction step is completed for the case of multiplications.

We finally come to square roots, $E = A^{1/2}$. We distinguish cases according to the relative size of $\tilde{A}$ and m_A. Assume first that $\tilde{A}$ is tiny compared to m_A, formally, $\tilde{A} < \mathbf{u}\cdot m_A$. We set $\tilde{E} = 0$. Then

$$\begin{aligned}\left|\tilde{E}-A^{1/2}\right| &= \left|A^{1/2}\right|\\ &\le (\left|\tilde{A}\right|+\left|\tilde{A}-A\right|)^{1/2}\\ &\le (\mathbf{u}\cdot m_A+((1+\mathbf{u})^{d_A}-1)\cdot m_A)^{1/2}\\ &\le (\mathbf{u}+((1+\mathbf{u})^{d_A}-1))^{1/2}(1+\mathbf{u})\cdot\sqrt{m_A} && \text{by inequality (7)}\\ &\le ((1+\mathbf{u})^{d_A+2}-1)\cdot\sqrt{m_A}\mathbf{u}^{-1/2},\end{aligned}$$

where the last inequality uses

$$\begin{aligned}(\mathbf{u}+((1+\mathbf{u})^{d_A}-1))^{1/2}(1+\mathbf{u}) &= [(\mathbf{u}+((1+\mathbf{u})^{d_A}-1))(1+\mathbf{u})^2]^{1/2}\\ &\le ((1+\mathbf{u})^{d_A+2}-1)^{1/2}\\ &\le (\mathbf{u}(d_A+3))^{1/2}\\ &\le (\mathbf{u}(d_A+2))\mathbf{u}^{-1/2}\\ &\le ((1+\mathbf{u})^{d_A+2}-1)\mathbf{u}^{-1/2}.\end{aligned}$$

Assume next that $\tilde{A} \ge \mathbf{u}\cdot m_A$. Then

$$\begin{aligned}\left|\sqrt{\tilde{A}}-A^{1/2}\right| &\le \left|\sqrt{\tilde{A}}-\tilde{A}^{1/2}\right|+\left|\tilde{A}^{1/2}-A^{1/2}\right|\\ &\le \mathbf{u}\cdot\sqrt{\tilde{A}}+\frac{\left|\tilde{A}-A\right|}{\tilde{A}^{1/2}+A^{1/2}} && \text{by inequality (6)}\\ &\le \mathbf{u}\cdot\sqrt{\tilde{A}}+\frac{((1+\mathbf{u})^{d_A}-1)\cdot m_A}{\tilde{A}^{1/2}}\\ &\le \mathbf{u}\cdot\sqrt{\tilde{A}}+((1+\mathbf{u})^{d_A}-1)(1+\mathbf{u})\cdot\frac{m_A}{\sqrt{\tilde{A}}} && \text{by inequality (8)}\\ &\le \mathbf{u}\cdot\sqrt{\tilde{A}}+((1+\mathbf{u})^{d_A}-1)(1+\mathbf{u})^2\cdot\max(m_A\oslash\sqrt{\tilde{A}},\mathtt{MINNORM}_F) && \text{by inequality (5)}\\ &\le (\mathbf{u}+((1+\mathbf{u})^{d_A}-1)(1+\mathbf{u})^2)\cdot\max(m_A\oslash\sqrt{\tilde{A}},\sqrt{\tilde{A}},\mathtt{MINNORM}_F)\\ &= ((1+\mathbf{u})^{d_A+2}-1)\cdot\max(m_A\oslash\sqrt{\tilde{A}},\sqrt{\tilde{A}}),\end{aligned}$$

where the last inequality follows from $\tilde{A} > 0$ and hence $\sqrt{\tilde{A}} \ge \mathtt{MINNORM}_F$. This completes the induction step for the case of square roots. □

THEOREM 18. *If* $d_E \le \sqrt{1/\mathbf{u}}-1$ *then*

$$\left|E-\tilde{E}\right| \le (d_E+1)\cdot\mathbf{u}\cdot m_E \le (d_E+2)\odot m_E\odot\mathbf{u}.$$

X	$\tilde{X}$	c_X	k_X	d_X
a	$\mathrm{fl}(a)$	1	1	1
$A+B$	$\tilde{A}\oplus\tilde{B}$	c_A+c_B	$\max(k_A,k_B)$	$1+\max(d_A,d_B)$
$A-B$	$\tilde{A}\ominus\tilde{B}$	c_A+c_B	$\max(k_A,k_B)$	$1+\max(d_A,d_B)$
$A\cdot B$	$\tilde{A}\odot\tilde{B}$	c_Ac_B	k_A+k_B	$1+d_A+d_B$

Table 4.2: The recursive definition of c_X, k_X and d_X. The first column contains the case distinction according to the syntactic structure of X, the second column contains the rule for computing $\tilde{X}$ and the third to fifth columns contain the rules for computing c_X, k_X, and d_X.

Proof. Follows immediately from Theorem 17 and Lemma 15. □

Exercise 0.13: Consider the computation of m_E according to Table 4.1. Show that the rule for square roots cannot lead to overflow (if $e_{max} > t+1$). Give examples, where the rules for addition, subtraction, and multiplication overflow.

Answer: We have $m_A < (2-1/2^t)2^{e_{max}}$. There are two rules for computing $E = m_{A^{1/2}}$. If $\tilde{A} < \mathbf{u}m_A$, we define $m_E = 2^{(t+1)/2} \odot \sqrt{m_A}$. The square-root operation cannot overflow; if the multiplication overflows we certainly have $\sqrt{m_A} > 2^{e_{max}-(t+1)/2}$ or $m_A > 2^{2e_{max}-(t+1)} > 2^{e_{max}}$, a contradiction. If $\tilde{A} \geq \mathbf{u}m_A$, we define $m_E = \max(\sqrt{\tilde{A}}, m_A \oslash \sqrt{\tilde{A}})$. Since $\tilde{A} \leq m_A$, the computation of $\sqrt{\tilde{A}}$ cannot overflow. Also, since $\tilde{A} \geq \mathbf{u}m_A$, $\sqrt{\tilde{A}} \geq \mathbf{u}^{1/2}\sqrt{m_A}$ and hence

$$m_A \oslash \sqrt{\tilde{A}}) \leq m_A \oslash \mathbf{u}^{1/2}\sqrt{m_A} \leq 2^{(t+1)/2}(1+\mathbf{u})^3\sqrt{m_A}$$

and we already shown that the latter quantity does not overflow. ♢

4.6 A Simplified Error Analysis for Polynomial Expressions

The error bounds of the preceding section are for machine consumption and not for human consumption. They should be used to filter the evaluation of geometric predicates. For the analysis of perturbation methods in Lecture **??** a weaker and simpler bound suffices. We next derive such a bound for polynomial expressions, i.e., expressions using only additions, subtractions, and multiplications. We show that

$$\left|\tilde{E} - E\right| \leq ((1+u)^{d_E} - 1)c_E M^{k_E},$$

where d_E, c_E and k_E are defined as in Table 4.2 and M is the smallest power of two such that

$$M \geq \max(1, \max\{\mathrm{lf}(|a|) \mid a \text{ is an operand in } E\}).$$

Exercise 0.14: Prove $M \geq \mathit{flu}(|a|)$ for all operands a in E. ♢

THEOREM 19. *Let M be defined as above. Then for every subexpression X of E,*

$$c_X \geq 1 \quad \textit{and} \quad k_X \geq 0 \quad \textit{and} \quad \left|\tilde{X} - X\right| \leq ((1+\mathbf{u})^{d_X} - 1)c_X M^{k_X},$$

where c_X, k_X and d_X are defined as in Table 4.2. This assumes that $c_X M^{k_X}$ is representable[2] *for all X. The latter assumption also guarantees that the computation of no m_X overflows.*

Proof. We use structural induction. Observe that the rules for d_X are the same as in Theorem 17. It therefore suffices to prove

$$m_X \leq c_X M^{k_X}$$

for all X. This is clear for operands. If $X = a \in \mathbb{R}$, $m_X = \max(\texttt{MINNORM}_F, \mathrm{fl}(a)) \leq M$. Consider an addition or subtraction next. Then

$$m_X = m_A \oplus m_B \leq c_A M^{k_A} \oplus c_B M^{k_B} \leq c_a M^{k_X} \oplus c_B M^{k_X} = (c_A + c_B) M^{k_X} = c_X M^{k_X},$$

where the next to last equality follows from the assumption that $c_X M^{k_X}$ is representable. Finally, we come to a multiplication. If $m_X = \texttt{MINNORM}_F$, the claim is obvious since $M \geq 1$, $k_X \geq 0$ and $c_X \geq 1$. So assume $m_X = m_A \odot m_B$. Then

$$m_X = m_A \odot m_B \leq c_A M^{k_A} \odot c_B M^{k_B} = (c_A c_B) M^{k_X} = c_X m^{k_X},$$

where again the next to last equality follows from our assumption that $c_X M^{k_X}$ is representable.

Finally, since $0 \leq m_X \leq c_X M^{k_X}$ and the latter quantity is assumed to be representable, the computation of m_X does not overflow. □

We continue our discussion of the orientation predicate for points a, b, and c given by their Cartesian coordinates. Then

$$\mathsf{Orientation}(a,b,c) = \mathrm{sign}((b_x - a_x)\cdot(c_y - a_y) - (b_y - a_y)\cdot(c_x - a_x)).$$

We already determined the degree of this expression as 6. The c- and k-values are as follows. For any argument, both values are one, for $X = b_x - a_x$, we have $c_X = 2$ and $k_X = 1$, for $X = (b_x - a_x)\cdot(c_y - a_y)$, we have $c_X = 4$ and $k_X = 2$, and finally for the entire expression we have $c_X = 8$ and $k_X = 2$. We conclude that the roundoff error in evaluating $\mathsf{Orientation}(a,b,c)$ with floating point arithmetic is at most

$$7 \cdot \mathbf{u} \cdot 8 \cdot M^2 = 56 \cdot \mathbf{u} \cdot M^2.$$

where M is the smallest non-negative power of two bounding all Cartesian coordinates. In particular, if $M = 2^{10}$ and double precision arithmetic is used, the error is at most $54 \cdot 2^{-53} \cdot 2^{20} \leq 2^{-27}$. Next recall that the expression underlying Orientation is twice the signed area of the triangle $\Delta(a,b,c)$. Thus, if coordinates are at most 2^{10} and the (unsigned) area of $\Delta(a,b,c)$ is at least 2^{-26}, then $\mathit{float\_orient}(a,b,c) = \mathsf{Orientation}(a,b,c)$. So *float_orient* errs only for very skinny triangles. Figure 3.1 suggested this, but now we know for sure. We will exploit the correctness of *float_orient* for non-skinny triangles in Lecture **??**.

Exercise 0.15: Redo the analysis above for points given by their homogeneous coordinates. We continue our discussion of the orientation predicate for points given by their homogeneous coordinates. Assuming $\mathrm{sign}(aw, bw, cw) = 1$, we have

$$\mathsf{Orientation}(a,b,c) = aw\cdot(bx\cdot cy - by\cdot cx) - bw\cdot(ax\cdot cy - ay\cdot cx) + cw\cdot(ax\cdot by - ay\cdot bx).$$

[2] This is certainly the case if $c_X \leq 2^{t+1}$ and $M^{k_X} \leq 2^{e_{max}}$.

We already determined the degree of this expression as 8. The c- and k-values are as follows. For any argument, both values are one, for $X = bx \cdot cy$, we have $c_X = 1$ and $k_X = 2$, for $X = (bx \cdot cy - by \cdot cx)$, we have $c_X = 2$ and $k_X = 2$, for $X = aw \cdot (bx \cdot cy - by \cdot cx)$ we have $c_X = 2$ and $k_X = 3$, for $X = aw \cdot (bx \cdot cy - by \cdot cx) - bw \cdot (ax \cdot cy - ay \cdot cx)$ we have $c_X = 4$ and $k_X = 3$, and finally for the entire expression we have $c_X = 6$ and $k_X = 3$. We conclude that the roundoff error in evaluating $\mathsf{Orientation}(p,q,r)$ with floating point arithmetic is at most

$$9 \cdot \mathbf{u} \cdot 6 \cdot M^3 = 54 \cdot \mathbf{u} \cdot M^3.$$

where M is the smallest non-negative power of two bounding the absolute value of all arguments. In particular, if $M = 2^{10}$ and double precision arithmetic is used, the error is at most $54 \cdot 2^{-53} \cdot 2^{30} \leq 2^{-17}$. If, we increase mantissa length to 99, the error bound becomes 2^{-64}. ♢

Exercise 0.16: Assume that for i, $1 \leq i \leq 8$, x_i is an integer with $|x_i| \leq 2^{20}$. Evaluate the expression $((x_1 + x_2) \cdot (x_3 + x_4)) \cdot x_5 + (x_6 + x_7) \cdot x_8$ with double precision floating point arithmetic. Derive a bound for the maximal difference between the exact result and the computed result. ♢

Exercise 0.17: Extend Theorem **??** to include square-roots. This requires to extend Table **??** and the proof of the theorem. We do not have a satisfactory answer for this exercise. ♢

4.7 A More Precise Error Analysis*

[[I will probably move this section to the chapter on deciding the sign of algebraic expressions.]]

Consider the expression

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

when $a \gg b$. The error analysis of Section 4.5 assumes that the error in the subtraction may be as large as

$$\mathbf{u} m_E \approx \mathbf{u}(2a+b).$$

However, the actual error is approximately

$$\mathbf{u} \cdot \tilde{E} \approx \mathbf{u} \cdot b,$$

which is much smaller. Can we improve our error analysis? Recall our formulae for estimating the error in additions (subtractions) and multiplications. We use err_E to denote $\tilde{E} - E$. For $E = A + B$, we have

$$\begin{aligned} err_E = \left|\tilde{A} \oplus \tilde{B} - (A+B)\right| &\leq \left|\tilde{A} \oplus \tilde{B} - (\tilde{A} + \tilde{B})\right| + \left|\tilde{A} - A\right| + \left|\tilde{B} - B\right| \\ &\leq \mathbf{u}\left|\tilde{A} \oplus \tilde{B}\right| + \left|\tilde{A} - A\right| + \left|\tilde{B} - B\right| \qquad \leq \mathbf{u} \odot \left|\tilde{E}\right| + err_A + err_B). \end{aligned}$$

and for $E = A \cdot B$, we have

$$\begin{aligned} |err_E| = \left|\tilde{A} \odot \tilde{B} - A \cdot B\right| &= \left|\tilde{A} \odot \tilde{B} - \tilde{A} \cdot \tilde{B} + \tilde{A} \cdot \tilde{B} - A \cdot \tilde{B} + A \cdot \tilde{B} - A \cdot B\right| \\ &\leq \mathbf{u}\left|\tilde{A} \odot \tilde{B}\right| + \left|\tilde{A} - A\right| \cdot \left|\tilde{B}\right| + |A| \left|\tilde{B} - B\right| \\ &\leq \mathbf{u}\left|\tilde{E}\right| + |err_A| \cdot \left|\tilde{B}\right| + |A| \cdot |err_B| \end{aligned}$$

These error bounds are more costly to evaluate than the bounds in Section 4.5. We will use them in Chapter **??**.

4.8 Arbitrary Precision Floating Point Numbers

In Section 4.3, we introduced the floating point system $F(t, e_{min}, e_{max})$. Software floating point systems are usually more flexible. They allow the user to change t during the computation, either by setting it to a fixed value at the beginning of the computation or by changing it freely during a computation. For some value, one wants a mantissa length of 1000 bits, and for another value, one wants 2000 bits, and for another value, one wants no rounding [3] Exponents are arbitrary integer, i.e., $e_{min} = -\infty$ and $e_{max} = +\infty$. The systems also support the different rounding modes of the IEEE standard. The mode can either be chosen for the entire computation or for a single operation.

As an example, consider the following LEDA program snippet computing an approximation of Euler's number $e \approx 2.71$. Let m be an integer. Our goal is to compute a bigfloat z such that $|z-e| \leq 2^{-m}$. Euler's number is defined as the value of the infinite series $\sum_{n\geq 0} 1/n!$. The simplest strategy to approximate e is to sum a sufficiently large initial fragment of this sum with a sufficiently long mantissa, so as to keep the total effect of the rounding errors under control. Assume that we compute the sum of the first n_0 terms with a mantissa length of t bits for still to be determined values of n_0 and t, i.e., we execute the following program.

```
bigfloat::set_rounding_mode(TOZERO);
bigfloat::set_precision(t);
bigfloat z = 2; integer fac = 2; int n = 2;
while (n < n0)
  { // fac = n! and z approximates 1/0! + ... + 1/(n-1)!
    z = z + 1/bigfloat(fac);
    n++; fac = fac * n;
  }
```

Let z_0 be the final value of z. Then z_0 is the value of $\sum_{n<n_0} 1/n!$ computed with bigfloat arithmetic with a mantissa length of t binary places. We have incurred two kinds of errors in this computation: a truncation error since we summed only an initial segment of an infinite series and a rounding error since we used floating point arithmetic to sum the initial segment. Thus,

$$|e-z_0| \leq \left|e - \sum_{n<n_0} 1/n!\right| + \left|\sum_{n<n_0} 1/n! - z_0\right|$$

$$= \sum_{n\geq n_0} 1/n! + \left|\sum_{n<n_0} 1/n! - z_0\right|$$

The first term is certainly bounded by $2/n_0!$ since, for all $n \geq n_0$, $n! = n_0! \cdot (n_0+1) \cdot \ldots \cdot n \geq n_0! \cdot 2^{n-n_0}$ and hence $\sum_{n\geq n_0} 1/n! \leq 1/n_0! \cdot (1+1/2+1/4+\ldots) \leq 2/n_0!$. What can we say about the total rounding error? We observe that we use one floating point division and one floating point addition per iteration and that there are $n_0 - 2$ iterations. Also, since we set the rounding mode to rounding-to-zero, the value of z always stays below e and hence stays bounded by 3. Thus, the results of all bigfloat operations are bounded by 3 and hence each bigfloat operation incurs a rounding error of at most $3 \cdot 2^{-t}$. Thus

$$|e-z_0| \leq 2/n_0! + 2n_0 \cdot 3 \cdot 2^{-t}.$$

[3] Additions, subtractions, and multiplications are exact if no rounding is performed and mantissas are allowed to have arbitrary length.

We want the right-hand side to be less than 2^{-m-1}; it will become clear in a short while why we want the error to be bounded by 2^{-m-1} and not just 2^{-m}. This can be achieved by making both terms less than 2^{-m-2}. For the first term this amounts to $2/n_0! \leq 2^{-m-2}$. We choose n_0 minimal with this property and observe that if we use the expression $fac.length() < m+3$ as the condition of our while loop then this n_0 will be the final value of n; $fac.length()$ returns the number of bits in the binary representation of fac. From $n_0! \geq 2^{n_0}$ and the fact that n_0 is minimal with $2/n_0! \leq 2^{-m-2}$ we conclude $n_0 \leq m+3$ and hence $6n_0 2^{-t} \leq 6(m+3)\cdot 2^{-t} \leq 2^{-m-2}$ if $t \geq 2m$; actually, $t \geq m+\log(m+3)+5$ suffices. The following program implements this strategy and computes z_0 with $|e-z_0| \leq 2^{-m-1}$.

We could output z_0, but z_0 is a number with $2m$ binary places and hence suggests a quality of approximation which we are not guaranteeing. Therefore, we round z_0 to the nearest number with a mantissa length of $m+3$ bits. Since $z_0 \leq 3$ this will introduce an additional error of at most $3\cdot 2^{-m-3} \leq 2^{-m-1}$. We conclude that the program below computes the desired approximation of Euler's number.

```
bigfloat::set_precision(2*m);
bigfloat::set_rounding_mode(TOZERO);
bigfloat z = 2; integer fac = 2; int n = 2;
while ( fac.length() < m + 3 )
  {  // fac = n! and z approximates 1/0! + 1/1! + ... + 1/(n-1)!
    z = z + 1/bigfloat(fac);
    n++; fac = fac * n;
  }
// |z - e| <= 2^{m-1} at this point

z = round(z,m+3,TONEAREST);
}
```

Exercise 0.18: Show how to compute π with an error less than 2^{-200}. ♢

4.9 Notes

In notes we do historical notes, implementation notes, and pointers to additional material.

Error analysis for floating point computations was pioneered by Wilkinson [**?**]. Most books on numerical analysis contain a section on error analysis. Detailed discussions can be found in [**?**]. The analysis presented here is based on [27, 44, 45, 29].

The optimal choice of pivot in the orientation test is discussed in [28].

Error bounds similar to the ones derived in this lecture are used as floating point filters in the linear kernels of LEDA and CGAL. We discuss linear kernels in the next lecture.

Arbitrary precision integer and floating point arithmetic is provided by several software packages. Popular packages are the GNU Multiple Precision Arithmetic Library [31] and and the Java [37] classes BigInteger and BigDecimal. The former package is the most comprehensive.

The orientation test and the side-of-circle test amount to computing the sign of a determinant. In low dimensions, it is easy and efficient to expand the determinant into an arithmetic formula. In higher dimensions, this becomes infeasible. An obvious method for computing the sign of a determinant is to compute the value of the determinant and then take its sign. Better algorithms are discussed in [14, 2, 6].

The following sentence is from the LEDA book. We need a similar sentence in the introduction. 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 [38, 27, 26, 12, 55, 14, 44, 11, 9, 8, 7, 46, 18, 3, 56, 49, 6].

4.10 Material for the Lecture

It is not clear yet, where the following remarks should go.

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 m_E requires only one addition. We concluded from our experiments in [44] that the additional cost is not warranted for the rational kernel.

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

However, the necessary conditional branching could impair performance significantly. If one is willing to invest that time, one could also think of using an exact implementation scheme based on floating-point filter techniques, e.g. [27, 52], see [54] for results of an experimental comparison. Further details are beyond the scope of this paper.

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 parlor (TODO: is this correct, or is it only C++ parlor), 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
t the inputs 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 4: 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 0.19: Formulate a model of the basic kernel, in which points are represented by their homogeneous coordinates. ♢

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 are 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::
  std::cout << "Orientation(o,p1,pl) = " << kernel.orientation(o,p1,pl) << std::
  std::cout << "Orientation(o,p1,pr) = " << kernel.orientation(o,p1,pr) << std::

  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::
  std::cout << "Orientation(o,p1,pl) = " << kernel.orientation(o,p1,pl) << std::
  std::cout << "Orientation(o,p1,pr) = " << kernel.orientation(o,p1,pr) << std::

  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"
  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 0.20: 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 4 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 16, this implementation is correct.

Exercise 0.21: 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 [45, 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

		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 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 $\left|\tilde{E}\right| \leq B$ 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 0.22: 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 insight[1]. 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

[1]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.

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[2] 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[3]. We use points a', b', c', and d' with integer Cartesian coordinates $\lfloor a_x D \rfloor$, $\lfloor a_y D \rfloor$, ... 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 $p(a_x, a_y, \ldots)$. 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 $-1 < \delta_{a_x} \leq 0$, 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 $a_x D$, $a_y D$, ... and degree $4-i$ in the δ_{a_x}, δ_{a_y}, ... Since the four points a, b, c, and d are co-circular, we have

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

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 17, 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 $p(a'_x, a'_y, \ldots)$ 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 5.3.

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

[3] 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.

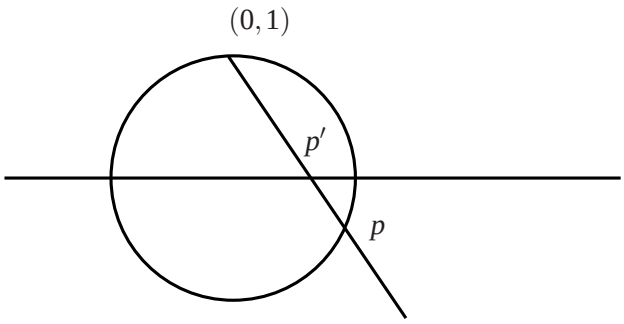


Figure 5.1: Point $p = (p_x, p_y)$ lies on the unit circle, point $p' = (a, 0)$ lies on the x-axes, and points $(0,1)$, p, and p' lies on a common line. .

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

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

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

LEMMA 20. *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-axes, see Figure 5.1. If $p = (p_x, p_y)$ lies on the unit circle, $p' = (a, 0)$ lies on the x-axis, and $(1,0)$, p and p' lie on a common line, then

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

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 0.23: 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 4.8; $\tilde{a}$ is the desired rational approximation. The fraction obtained in this way has a numerator and denominator of s bits.

One can obtain 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 21 (Dirichlet, 1842). *For any real x and any positive ε there is a rational number p/q such that*

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

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, $0 \leq\leq M$, 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 $\left| f_j - f_i \right| \leq 1/M$. Assume $j > i$. Then

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

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 $x_0, x_1, x_2, \ldots$ of reals and a sequence $a_0, a_1, a_2, \ldots$ of integers as follows.

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

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 $[a_0; a_1, a_2, \ldots]$ 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

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

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

LEMMA 22. *Let $x \in \mathbb{R}_{\geq 0}$ and let $[a_0;a_1,a_2,\ldots]$ be the continued fraction expansion of x. Define $p_{-2}=0$, $q_{-2}=1$, $p_{-1}=1$, $q_{-1}=0$, and*

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

Then

1. $\dfrac{p_n}{q_n} = [a_0;a_1,\ldots,a_n]$ *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. $\frac{p_{-2}}{q_{-2}} < \frac{p_0}{q_0} < \frac{p_2}{q_2} < \ldots \leq x \leq \ldots \frac{p_3}{q_3} < \frac{p_1}{q_1} < \frac{p_{-1}}{q_{-1}}$.
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

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

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 $[a_0;a_1,\ldots,a_n] = p_n/q_n$. 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

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

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

LEMMA 23. *Let $x \in \mathbb{R}_{\geq 0}$ and let $[a_0; a_1, a_2, \ldots]$ 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,

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

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 0.24: Give more details on how to compute a rational approximation of a with error at most ε. ♢

Exercise 0.25: 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. ♢

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 [14, 2, 6]. 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 [27, 52].

Section 5.6 is based on [12].

[[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 $\left|\tilde{E}\right| > B$. Since E is twice the signed area (see Lemma 3) 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 [19] 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.

Lecture 6

Delaunay Triangulations and Voronoi Diagrams

discuss an algorithm for Delaunay Triangulations, e.g., randomized incremental. Discuss Voronoi diagrams as the dual.

also do conceptual perturbation: walk through a triangulation. to get the code right. This is discussed in the LEDAbook and also in my 2000 course notes.

6.1 Notes

Lecture 7

Perturbation

Computational geometers tend to formulate their algorithms for inputs in general position. What is an input in general position? General position is always defined with respect to a set of predicates. A set of points is in general position with respect to the orientation predicate if no three points are collinear. It is in general position with respect to the side-of-circle predicate if no four points are co-circular. It is general position with respect to the orientation predicate and the side-of-circle predicate if no three points are collinear and no four points are co-circular. Generally, if $f_1, \ldots, f_k$ are functions of geometric objects, then a set of objects is in general position with respect to these functions, if all function evaluations for objects in the set yield nonzero.

Geometric algorithms branch on the outcome geometric predicates. In general, the branches are three-way branches: positive sign, negative sign, and zero. If the input is in general position, the zero branch is never taken. This simplifies the algorithm. We have already seen several examples to this effect. In the convex hull algorithm, we had to distinguish between visibility and weak visibility and we had to cope with inputs that are contained in a lower dimensional subspace. In the Delaunay triangulation algorithm, we had to cope with co-circular points and with inputs that are contained in a lower dimensional subspace.

So the general position assumption simplifies the life of an algorithm designer. However, at the cost of the programmer. A program has to cope with all inputs and so has to deal with degenerate inputs. What can a programmer do? There are essentially two approaches:

- Redesign the algorithm so that it handles degenerate inputs.
- Use perturbation to bring the input into general position.

Whenever we discuss an algorithm in this book, we follow the first approach. We make sure that the algorithms works for all inputs. In this lecture and the next, we study perturbation techniques. *The idea is to solve the problem not on the given input, but on a nearby input. The nearby input is obtained by perturbing the given input. The perturbed input will then be in general position and, since it is near the original input, the result for the perturbed input will hopefully still be useful.* This hope has to be substantiated in any application of the perturbation technique. We cannot make general claims with respect to this hope. We give a positive and a negative example. If the input objects are derived from some physical measurement, then a perturbation within the precision of the measuring device should be acceptable. On the other hand, for an algorithm whose task is to decide whether the input is in general position, perturbation makes no sense.

Exercise 0.26: Go through the examples in the first lecture. For which of them is perturbation a reasonable technique? Discuss two additional examples of your own choice. ♢

Perturbation comes in two flavors: symbolic and numerical. In symbolic perturbation, one perturbs inputs by infinitesimal amounts, and in numerical perturbation, one actually changes the coordinates. (REWRITE).

7.1 Symbolic Perturbation

It is convenient to summarize the input into a single vector $x \in \mathbb{R}^N$. For example, if the input is n points in the plane, we would set $N = 2n$ and pack all $2n$ coordinates into a single vector. A test function is then simply a function $f : \mathbb{R}^N \mapsto \mathbb{R}$. Let F be a collection of test functions. For example, if an algorithm uses the geometric predicates lex-compare, orientation, and side-of-circle for n points in the plane, F contains $\binom{n}{2}$ test functions corresponding to lex-compare (one for each pair of distinct points), $\binom{n}{3}$ test functions corresponding to orientation, and $\binom{n}{4}$ test functions corresponding to side-of-circle.

DEFINITION 2. *Let $f : \mathbb{R}^N \mapsto \mathbb{R}$ be a test function and $\sigma = f^{-1}(0)$ be its zero set. We call f* well-behaved *if every straight line ℓ is either contained in σ or every bounded segment of ℓ intersects σ in finitely many points.*

Many functions are well-behaved, e.g., all polynomials and all rational functions. In particular, for any geometric test used in this book, the underlying function is well-behaved.

THEOREM 24. *Let F be a collection of well-behaved continuous functions and let $a \in \mathbb{R}^N$ be a vector that is in general position with respect to F, i.e., $f(a) \neq 0$ for all $f \in F$. Then for any $f \in F$ and any $q \in \mathbb{R}^N$*

$$\overline{f}(q) := \lim_{\varepsilon \to 0+} \operatorname{sign} f(q + \varepsilon(a - q))$$

exists and is non-zero. Moreover, if $f(q) \neq 0$, $\overline{f}(q) = \operatorname{sign} f(q)$.

Proof. The function $\varepsilon \mapsto q + \varepsilon(a - q)$ defines a line ℓ passing through q and a. Since $f(a) \neq 0$, ℓ is not contained in σ and hence the segment $\overline{qa}$ intersects σ only finitely often. Thus there is an $\varepsilon_0 > 0$ such that $f(q + \varepsilon(a - q)) \neq 0$ for $0 < \varepsilon < \varepsilon_0$. Since f is continuous, $\operatorname{sign} f(q + \varepsilon(a - q))$ is constant for $0 < \varepsilon < \varepsilon_0$. Thus $\overline{f}(q)$ exists and is non-zero.

Assume next that $f(q) \neq 0$. Since f is continuous, there is an $\varepsilon_0 > 0$ such that $f(q + \varepsilon(a - q)) \neq 0$ for $0 \leq \varepsilon < \varepsilon_0$. Again by continuity, $\operatorname{sign} f(q) = \overline{f}(q)$. □

COROLLARY 25. *Consider any algorithm that branches only on the sign of a function f from a class F of well-behaved continuous functions applied to the input $q \in \mathbb{R}^N$. Also assume that $a \in \mathbb{R}^N$ that is non-degenerate for all $f \in F$. Branching on $\overline{f}(q)$ instead of on $\operatorname{sign} f(q)$ has the following effect:*

- *The zero branch is never taken, and*
- *If q is in general position, the computation does not change.*

Proof. This follows immediately from Theorem 24. Since $\overline{f}(q) \neq 0$ for all q, the zero branch is never taken, and since $\overline{f}(q) = \operatorname{sign} f(q)$ whenever $f(q) \neq 0$, the computation does not change for an input in general position. □

The corollary may be paraphrased as *if you know just one input in general position, any input can be perturbed into general position*. We still need to address two questions. How do we find inputs in general position and how can we compute $\overline{f}(q)$? We address both questions first for the orientation predicate of n points in the plane.

LEMMA 26. *The points $a_i = (i, i^2)$, $1 \leq i \leq n$, are in general position with respect to the orientation predicate.*

Proof. Lines intersect the parabola $y = x^2$ in at most two points. Thus no three a_i are collinear. □

We next discuss how to evaluate the orientation predicate. Assume our inputs are the points q_i, $1 \leq i \leq n$. We replace q_i by $q_i + \varepsilon(a_i - q_i)$. For three distinct points q_i, q_j, and q_k, we then have:

$$\overline{\mathsf{Orientation}}(q_i,q_j,q_k) = \lim_{\varepsilon\to 0+} \operatorname{sign} \begin{vmatrix} 1 & (1-\varepsilon)x(q_i)+\varepsilon i & (1-\varepsilon)y(q_i)+\varepsilon i^2 \\ 1 & (1-\varepsilon)x(q_j)+\varepsilon j & (1-\varepsilon)y(q_j)+\varepsilon j^2 \\ 1 & (1-\varepsilon)x(q_k)+\varepsilon k & (1-\varepsilon)y(q_k)+\varepsilon k^2 \end{vmatrix}$$

Expansion and collecting terms according to powers of ε yields

$$= \mathsf{Orientation}(q_i,q_j,q_k) + \lim_{\varepsilon\to 0+} \operatorname{sign} \left(\varepsilon P(q_i,q_j,q_k) + \varepsilon^2 \begin{vmatrix} 1 & i & i^2 \\ 1 & j & j^2 \\ 1 & k & k^2 \end{vmatrix} \right),$$

where $P(q_i,q_j,q_k,i,j,k)$ is a polynomial. Thus

$$\overline{\mathsf{Orientation}}(q_i,q_j,q_k) = \begin{cases} \mathsf{Orientation}(q_i,q_j,q_k) & \text{if } \mathsf{Orientation}(q_i,q_j,q_k) \neq 0 \\ \operatorname{sign}(P(q_i,q_j,q_k)) & \text{if } \mathsf{Orientation}(q_i,q_j,q_k) = 0 \text{ and } P(q_i,q_j,q_k) \neq 0 \\ \mathsf{Orientation}(a_i,a_j,a_k) & \text{if } \mathsf{Orientation}(q_i,q_j,q_k) = 0 = P(q_i,q_j,q_k) \end{cases}$$

We next address the equations more generally. We exhibit inputs in general position for the set of test functions introduced in the introductory paragraph. We do so for arbitrary dimension d and not only for the plane. We consider n points chosen from the positive branch (i.e., $t > 0$) of the moment curve $t \mapsto (t, t^2, \ldots, t^d)$. No two points on this curve agree in any coordinate. No $d+1$ points lie in a common hyperplane. Consider the equation $a_0 + \sum_{1\leq i\leq d} a_i x_i$ of any hyperplane. Plugging $x = (t, t^2, \ldots, t^d)$ into this equation gives a polynomial of degree d in t. We conclude that the hyperplane intersects the moment curve in at most d points. Finally, the positive branch of the moment curve intersects no sphere in $d+2$ or more points. Let $\sum_{1\leq i\leq d}(x_i - c_i)^2 - r^2 = 0$ be the equation of a sphere. Plugging $x = (t, t^2, \ldots, t^d)$ into this equation gives the following polynomial in t:

$$\sum_{1\leq i\leq d} (t^i - c_i)^2 - r^2.$$

Descartes rule of signs (Theorem **??**) states that the number of positive roots of a polynomial is bounded by the number of sign changes in its coefficient sequence. The polynomial above can have at most $d+1$ sign changes since the coefficients of the powers t^j with $j > d$ are nonnegative (any such coefficient is either zero or one).

We first show how to compute $\overline{f}(q)$ for polynomials f. We use q_1 to q_N to denote the coordinates of $\mathbb{R}^N$ and assume that $f(q_1, \ldots, q_n)$ is a polynomial of total degree d. Then.

$$f(q + \varepsilon(a-q)) = f(q_1 + \varepsilon(a_1 - q_1), \ldots, q_N + \varepsilon(a_N - q_N)) = \sum_{0\leq i\leq d} p_i(q_1, \ldots, q_n)\varepsilon^i,$$

where the p_i are polynomials of total degree at most d. We claim

$$\overline{f}(q) = \text{sign} p_i(q) \quad \text{where } i = \min\{j \mid p_j(q) \neq 0\}.$$

We know from Theorem **??** that the sign of $f(q+\varepsilon(a-q))$ is constant and nonzero for sufficiently small ε. Therefore at least one $p_j(q)$ must be non-zero. Let i be minimal with $p_i(q) \neq 0$. Then

$$f(q+\varepsilon(a-q)) = p_i(q)\varepsilon^i \left(1 + \sum_{j>i} \frac{p_j(q)}{p_i(q)} \varepsilon^{j-i}\right).$$

Let $M = \max \left|p_j(q)/p_i(q)\right|$. Then $\left|\sum_{j>i} p_j(q)/p_i(q)\varepsilon^{j-i}\right| \leq M/(1-\varepsilon) < 1/2$ for sufficiently small ε.

7.2 Numerical Perturbation

[[the following is copied from Funke/Klein/Mehlhorn/Schmitt. It needs to rewritten so that it fits better.]]

7.3 Some Words of Caution

Perturbation is not a cure-all. It removes burden from the algorithm designer and implementer. However, it has two drawbacks.

The running time of an algorithm may increase as a result of perturbation. We give two examples. Assume we are given n line segments passing through the origin. We will see in Section **??** that we can compute their arrangement in time $O(n \log n)$. However, perturbing the line segments into general position (no three intersect in a point) will generate an arrangement with $\Theta(n^2)$ intersection points. The second example is even more extreme. Assume we are given n identical points in $\mathbb{R}^d$. Any sensible convex hull algorithm should be able to handle this input in linear time. However, the perturbation scheme of Section 7.1 moves the n points onto the d-dimensional moment curve. The resulting hull will have $\Omega n^{\lfloor d/2 \rfloor}$ facets and hence any algorithm will need time $\Omega n^{\lfloor d/2 \rfloor}$ for computing the hull of the perturbed points.

Exercise 0.27: Prove bound for points on the moment curve. ♢

The second drawback is that we solve the problem on a perturbed input and not on the original input. The output for the perturbed input may tell us little about the output for the original input.

The symbolic scheme has another drawback. It requires exact computation.

Neither approach to perturbation will apply if some test function is identically zero. For example, if one tests whether a point p lies on a line involving p as one of the defining points, the outcome will be "on line" no matter who one perturbs the input. The reader may think that test functions that are identically zero can only arise as a consequence of stupid programming. However, they can also arise because the algorithm designer misses a theorem, see Figure 7.1.

7.4 Notes

[20] introduced symbolic perturbation and applied it to the orientation predicate. [22, 51, 21, 57] extended and simplified the technique. Our presentation follows [51]. An implementation of the scheme is available in CGAL [15]. CITATION IS INCOMPLETE.

Section **??** is based on [30].

Section 7.3 is based on [10].

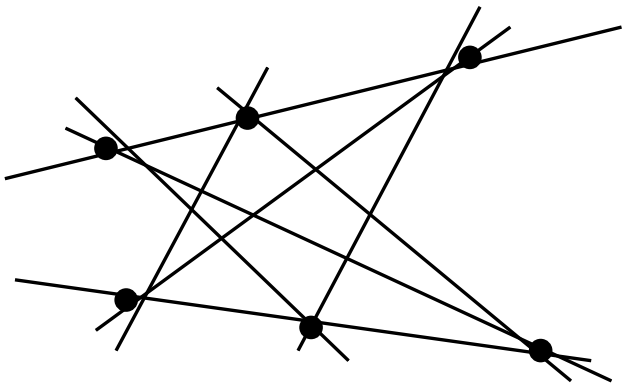


Figure 7.1: p_1, p_2, p_3 are three arbitrary points on a line ℓ_1 and q_1, q_2, q_3 are three arbitrary points on a line ℓ_2. For $1 \leq i \leq 3$ let $\{j,k\} = \{1,2,3\} \setminus i$ and let r_i be the intersection of $\ell(p_j,q_k)$ and $\ell(p_k,q_j)$. Pappus (ca. 300 AD) proved that r_1, r_2 and r_3 are collinear. So perturbing the input will not help.

7.5 Proposed Contents

discuss SoS by Edelsbrunner and Muecke, Seidel

discuss controlled perturbation. This can be based on the SODA article by Funke/Klein/Mehlhorn/Schmitt. Reference to Devillers/Preparata.

also do conceptual perturbation: walk through a triangulation. to get the code right. This is discussed in the LEDAbook and also in my 2000 course notes.

Bibliography

[1] A. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9:216–219, 1979.

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

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

[4] M. Blum and H. Wasserman. Reflections on the pentium division bug. *IEEE Transaction on Computing*, 45(4):385–393, 1996.

[5] J.-D. Boissonnat and M. Yvinec. *Algorithmic Geometry*. Cambridge University Press, Cambridge, 1998.

[6] 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.

[7] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. A strong and easily computable separation bound for arithmetic expressions involving square roots. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*, pages 702–709, 1997. `www.mpi-sb.mpg.de/~mehlhorn/ftp/sepbound.ps`.

[8] C. Burnikel, S. Funke, and M. Seel. Exact arithmetic using cascaded computation. In *Proceedings of the 14th Annual Symposium on Computational Geometry (SCG'98)*, pages 175–183, 1998.

[9] C. Burnikel, K. Mehlhorn, and S. Schirra. How to compute the Voronoi diagram of line segments: Theoretical and experimenta l results. In *Proceedings of the 2nd Annual European Symposium on Algorithms - ESA'94*, volume 855 of Lecture Notes in Computer Science, pages 227–239. Springer, 1994.

[10] C. Burnikel, K. Mehlhorn, and S. Schirra. On Degeneracy in Geometric Computations. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'94)*, pages 16–23, 1994.

[11] C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'94)*, pages 16–23, 1994.

[12] 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.

[13] K. Clarkson and P. Shor. Applications of random sampling in computational geometry, II. *Journal of Discrete and Computational Geometry*, 4:387–421, 1989.

[14] K. L. Clarkson. Safe and effective determinant evaluation. *IEEE Foundations of Computer Sci.*, 33:387–395, 1992.

[15] J. Comes and M. Ziegelmann. An easy to use implementation of linear perturbations within CGAL. In *Algorithm Engineering*, pages 169–182, 1999.

[16] M. de Berg, M. Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 1997.

[17] P. Deuflhard and A. Hohmann. *Numerische Mathematik: Eine algorithmisch orientierte Einführung*. Walter de Gruyter, 1991.

[18] O. Devillers, G. Liotta, F. 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 Universi ty, 1997.

[19] O. Devillers and F. Preparata. A probabilistic analysis of the power of arithmetic filters. *Discrete & Computational Geometry*, 20:523–547, 1998.

[20] H. Edelsbrunner and E. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, Jan. 1990.

[21] I. Emiris, J. Canny, and R. Seidel. Efficient perturbations for handling geometric degeneracies. *Algorithmica*, 19:219–242, 1997.

[22] I. Z. Emiris and J. F. Canny. A general approach to removing degeneracies. *SIAM Journal on Computing*, 24(3):650–664, June 1995.

[23] A. Fabri, E. Fogel, B. Gärtner, M. Hoffmann, L. Kettner, S. Pion, M. Teillaud, R. Veltkamp, and M. Yvinec. The CGAL manual. 2003. Release 3.0.

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

[25] A. R. Forrest. Computational geometry in practice. In R. A. Earnshaw, editor, *Fundamental Algorithms for Computer Graphics*, volume F17 of *NATO ASI*, pages 707–724. Springer-Verlag, 1985.

[26] S. Fortune. Robustness issues in geometric algorithms. In *Proceedings of the 1st Workshop on Applied Computational Geometry: Towards Geometric Engineering (WACG'96)*, volume 1148 of Lecture Notes in Computer Science, pages 9–13, 1996.

[27] 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.

[28] S. J. Fortune. Numerical stability of algorithms for 2d Delaunay triangulations. *Int'l. J. Comput. Geometry and Appl.*, 5(1):193–213, 1995.

[29] S. Funke. Exact arithmetic using cascaded computation. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1997.

[30] S. Funke, C. Klein, K. Mehlhorn, and S. Schmitt. Controlled Perturbation for Delaunay Triangulations. SODA, pages 1047–1056, 2005.

[31] GMP (GNU Multiple Precision Arithmetic Library). `http://gmplib.org/`.

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

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

[34] R. L. Graham. An efficient algorithm for determining the convex hulls of a finite point set. *Information Processing Letters*, 1:132–133, 1972.

[35] Halperin and Shelton. A perturbation scheme for spherical arrangements with application to molecular modeling. *CGTA: Computational Geometry: Theory and Applications*, 10, 1998.

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

[37] Java. `http://www.java.com/en/`.

[38] M. Jünger, G. Reinelt, and D. Zepf. Computing correct Delaunay triangulations. *Computing*, 47:43–49, 1991.

[39] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *Proceedings of the 15th Annual ACM Symposium on Computational Geometry*, pages 351–359, Miami, Florida, 1999.

[40] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A Core library for robust numerical and geometric computation. In *15th ACM Symp. Computational Geometry*, pages 351–359, 1999.

[41] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7(7):595–596, 1963.

[42] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom Examples of Robustness Problems in Geometric Computations. In *ESA*, volume 3221 of *LNCS*, pages 702–713, 2004. full paper to appear in CGTA.

[43] LEDA (Library of Efficient Data Types and Algorithms). `www.algorithmic-solutions.com`.

[44] K. Mehlhorn and S. Näher. The implementation of geometric algorithms. In *Proceedings of the 13th IFIP World Computer Congress*, volume 1, pages 223–231. Elsevier Science B.V. North-Holland, Amsterdam, 1994. `www.mpi-sb.mpg.de/~mehlhorn/ftp/ifip94.ps`.

[45] K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[46] K. Mehlhorn, S. Näher, M. Seel, R. Seidel, T. Schilz, S. Schirra, and C. Uhrig. Checking geometric programs or verification of geometric structures. *Computational Geometry*, 12(1-2):85–103, 1999.

[47] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[48] K. Mulmuley. *Computational Geometry*. Prentice Hall, 1994.

[49] S. Schirra. Robustness and precision issues in geometric computation. to appear, preliminary version available as MPI report.

[50] M. Seel. Eine Implementierung abstrakter Voronoidiagramme. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1994.

[51] R. Seidel. The nature and meaning of perturbations in geometric computing. *Discrete & Computational Geometry*, 19(1):1–17, 1998.

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

[53] P. Sterbenz. *Floating Point Computation*. Prentice Hall, 1974.

[54] J. Tusch and S. Schirra. Experimental comparison of the cost of approximate and exact convex hull computation in the plane. In *CCCG*, 2006.

[55] C. Yap. Towards exact geometric computation. In *Proceedings of the 5th Canadian Conference on Computational Geometry (CCCG'93)*, pages 405–419, 1993.

[56] C. Yap and T. Dube. The exact computation paradigm. In *Computing in Euclidean Geometry II*. World Scientific Press, 1995.

[57] C.-K. Yap. Geometric consistency theorem for a symbolic perturbation scheme. *J. Comput. Syst. Sci.*, 40(1):2–18, 1990.