Lecture 1

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 [6, **?**]). 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 0.396 is rounded to the approximate result 0.40.

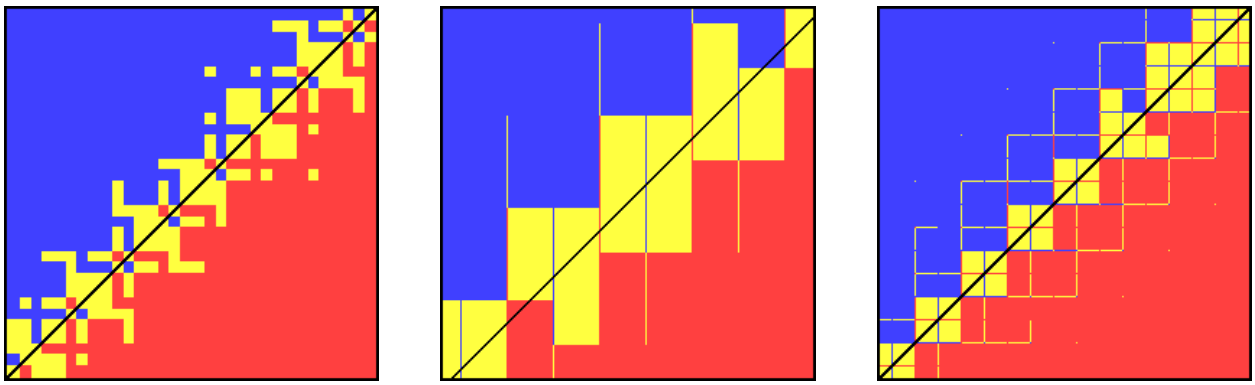
We will see in this lecture that floating point arithmetic is a pure substitute for real arithmetic and that the floating point implementation of our algorithm can produce very strange results. We hope that, after seeing these examples, our students look forward to the solution techniques that we present in later lectures. The core of a C++ implementation of our algorithm is given in Section 1.2. The full code can be found in the companion web page [2] of article [9] on which this lecture is based.

1.1 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)**

$\begin{pmatrix} 0.50000000000002531 \\ 0.5000000000000171 \end{pmatrix}$ $\begin{pmatrix} 17.300000000000001 \\ 17.300000000000001 \end{pmatrix}$ $\begin{pmatrix} 24.00000000000005 \\ 24.0000000000000517765 \end{pmatrix}$ **(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 1.1: 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 1.1(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

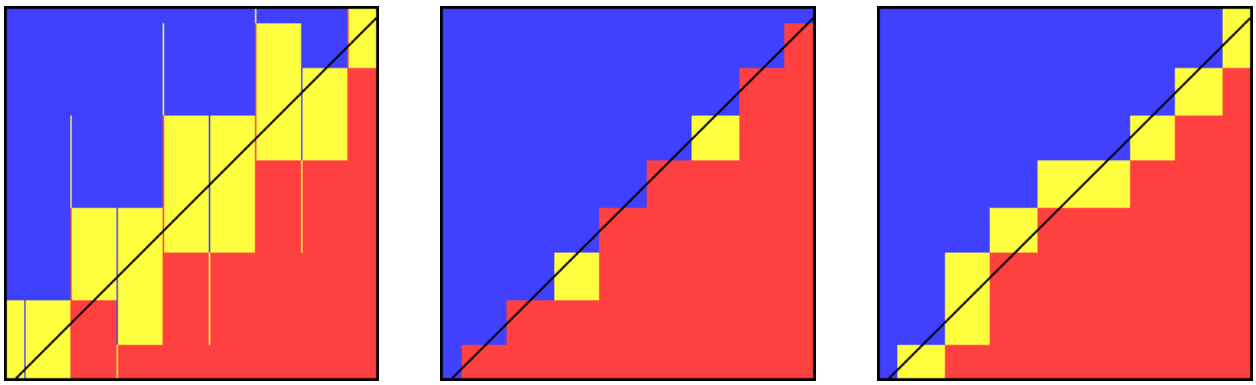


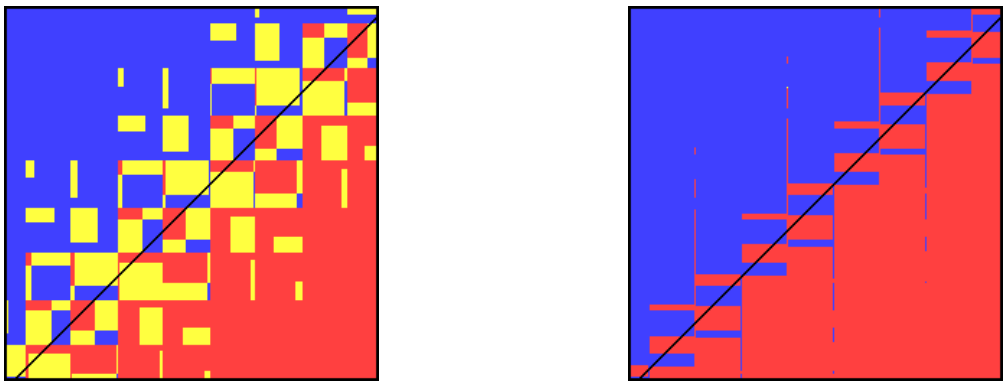
Figure 1.2: We repeat the example from Figure 1.1(b) and show the result for all three distinct choices for the pivot; namely p on the left, q in the middle, and r on the right. All figures exhibit sign reversal.

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 1.1(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 1.1 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 0.1: Download the code from the web page of the course and perform your own experiments. ♢



(a) **(b)**

Figure 1.3: Examples of the impact of extended double arithmetic. We repeat the example from Figure 1.1(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 1.1. 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 1.2 displays the effect of the different choices of the pivot point on the example of Figure 1.1(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 1.3 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.

1.2 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 [3], 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));
}
```

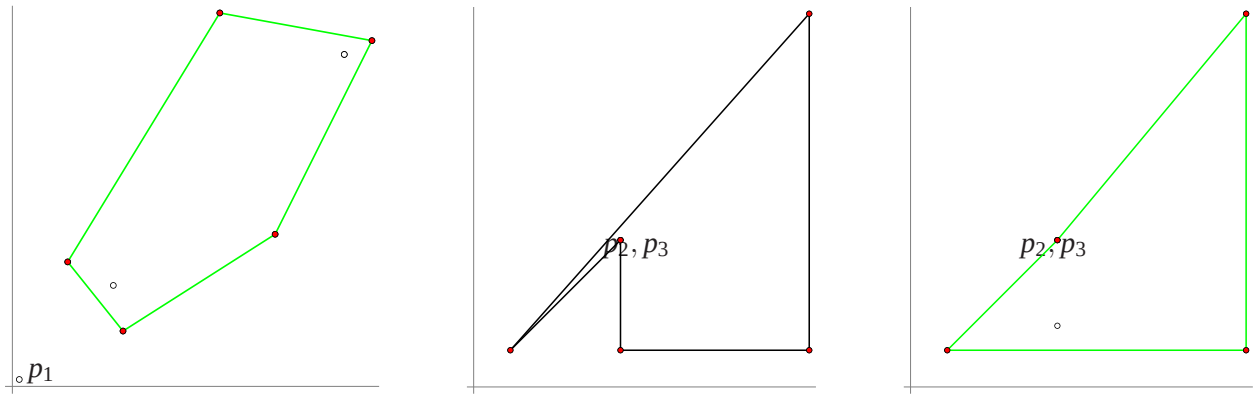
For the initial three non-collinear points we scan the input sequence and maintain its convex hull of up to two extreme points until we run out of input points or we find a third extreme point for the convex hull. From there on we scan the remaining points in our main `convex_hull` function as shown 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 1.4: 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);
}
```

1.3 The Impact on the Convex Hull Algorithm

Let us next see the impact of approximate arithmetic on our convex hull algorithm. Figure 1.4 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 1.4(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 1.5(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).

1.4 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 1.4(a) and 1.6(a) show the computed convex hull; a point that is clearly extreme is left out of the hull.

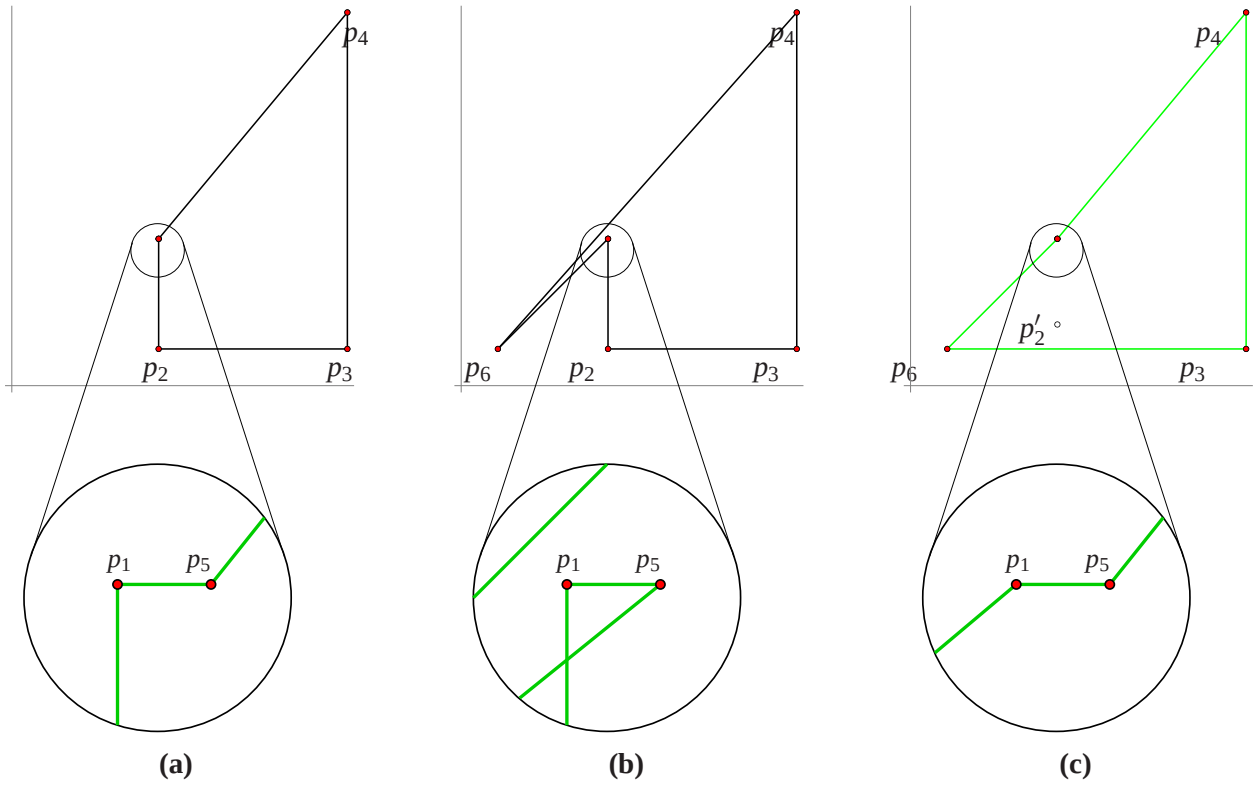


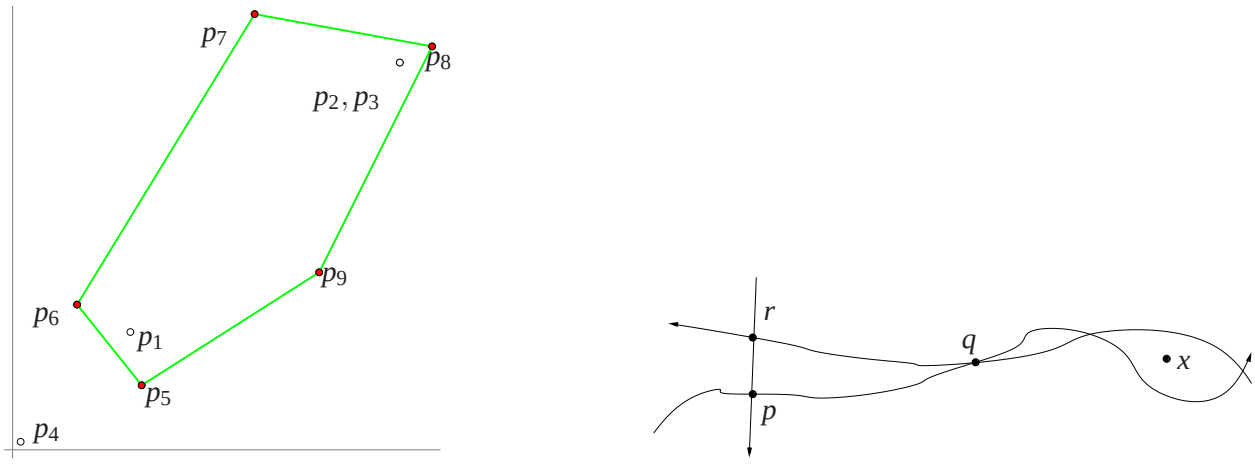
Figure 1.5: **(a)** The hull constructed after processing points p_1 to p_5. Points p_1 and p_5 lie close to each other and are indistinguishable in the upper figure. The magnified schematic view below shows that we have a concave corner at p_5. The point p_6 sees the edges (p_1, p_2) and (p_4, p_5), 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 1.6(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 1.6: **(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).

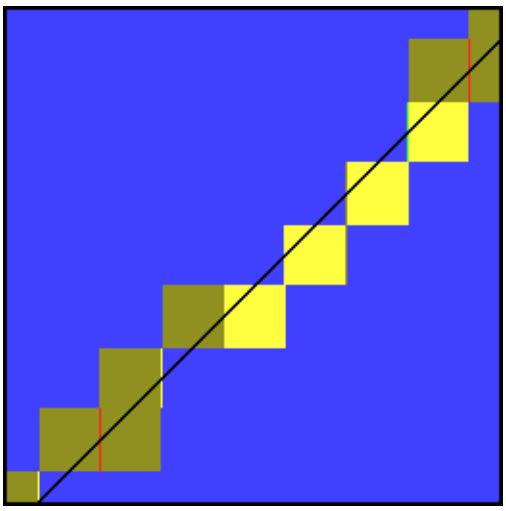
the two nearly parallel edges. In view of Figure 1.1 such a point might be mis-classified with respect to one of the edges and hence would be unable to see any edge of the triangle. So we started with the points used in Figure 1.1(b), i.e., $p_1 \approx (17,17)$, $p_2 \approx (24,24) \approx p_3$, where 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 1.7(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 1.7(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 1.8 illustrates this choice. A concrete example follows:

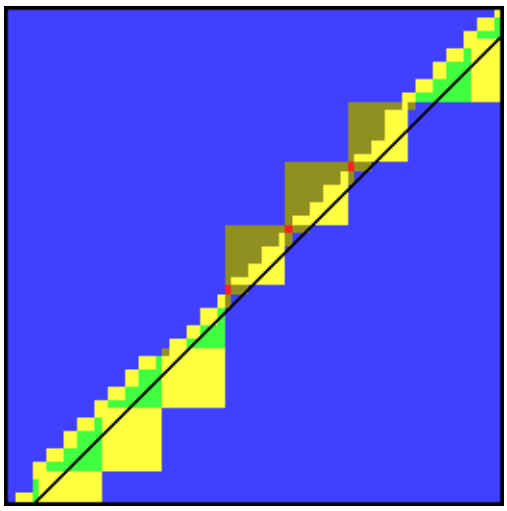
$$
\begin{array}{lcll}
p_1 & = & (\ 200.0, & 49.200000000000003) \\
p_2 & = & (\ 100.0, & 49.600000000000001) \\
p_3 & = & (-233.33333333333334, & 50.93333333333333\ ) \\
p_4 & = & (\ 166.66666666666669, & 49.333333333333336)
\end{array}
\qquad
\begin{array}{l}
\textit{float\_orient}(p_1,p_2,p_3) > 0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

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 1.7: 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 1.1. **(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.

1.5 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:

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

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

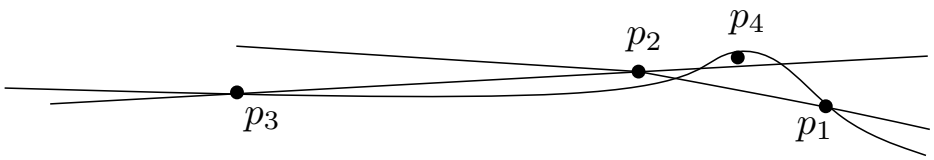


Figure 1.8: Schematic view of a point seeing all hull edges: The point p_4 sees all edges of the triangle (p_1, p_2, p_3).

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 1.6(a)). Second, a misclassified point may create a slightly non-convex polygon. This small error is amplified by later insertions (see Figure 1.4(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 **??** 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 **??** 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.

1.6 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 [2, 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 1. *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.,

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

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?

1.7 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 $P \subseteq U_\varepsilon(\text{conv}S)$ and $\text{conv}S \subseteq U_\varepsilon(P)$, 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 [**?**, 3, 10, 8]. 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 [7, 5]. 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.

1.8 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

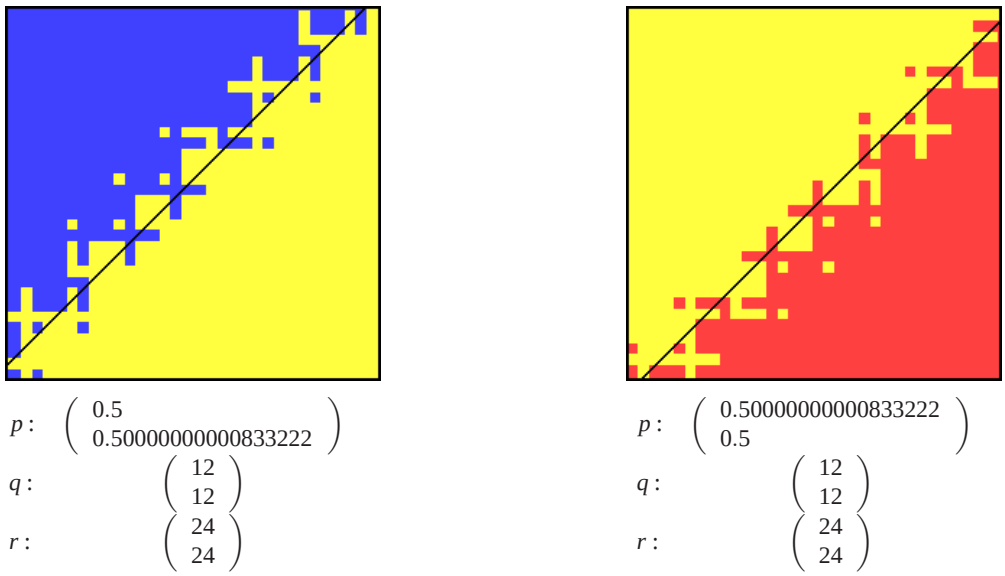


Figure 1.9: The effect of epsilon-tweaking: The figures show the result of repeating the experiment of Figure 1.1(a), but using an absolute epsilon tolerance value of $\varepsilon = 10^{-10}$, i.e., three points are declared collinear if *float_orient* returns a value less than or equal to 10^{-10} in 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 1.5 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 1.1, 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 1.9.

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.

1.9 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 [2, 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 [4], Fortune's example for a variant of Graham's scan [**?**], Shewchuk's example for divide-and-conquer Delaunay triangulation [11], Ramshaw's braided lines [10, Section 9.6.2], Schirra's example for convex hulls [10, Section 9.6.1], and Mehlhorn and Näher's examples for the sweep line algorithm for line segment intersection and boolean operations on polygons [10, Sections 10.7.4 and 10.8.4]. This lecture is based on an article by Kettner et al. [9].

Software and hardware reliability goes much beyond geometric computing. A version of the Pentium chip contained an error in the division hardware [1]. 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

1.10 Implementation Notes

1.11 Exercises

Exercise 0.2: Formulate more guarantees. ♢

Bibliography

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

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

[3] 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.

[4] 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.

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

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

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

[8] 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.

[9] 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.

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

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