MAX-PLANCK-INSTITUT FÜR INFORMATIK

Implementation of a Sweep Line Algorithm for the Straight Line Segment Intersection Problem

Kurt Mehlhorn Stefan Näher

MPI-I-94-160 October 1994

INFORMATIK

Im Stadtwald
66123 Saarbrücken
Germany

Implementation of a Sweep Line Algorithm for the Straight Line Segment Intersection Problem

Kurt Mehlhorn Stefan Näher

MPI–I–94–160 October 1994

Implementation of a Sweep Line Algorithm for the Straight Line Segment Intersection Problem *

Kurt Mehlhorn and Stefan Näher
Max-Planck-Institut für Informatik,
66123 Saarbrücken, Germany

1. Abstract.

We describe a robust and efficient implementation of the Bentley-Ottmann sweep line algorithm [1] based on the LEDA library of efficient data types and algorithms [7]. The program computes the planar graph G induced by a set S of straight line segments in the plane. The nodes of G are all endpoints and all proper intersection points of segments in S. The edges of G are the maximal relatively open subsegments of segments in S that contain no node of G. All edges are directed from left to right or upwards. The algorithm runs in time $O((n+s)\log n)$ where n is the number of segments and s is the number of vertices of the graph G. The implementation uses exact arithmetic for the reliable realization of the geometric primitives and it uses floating point filters to reduce the overhead of exact arithmetic.

Contents

*This work was supported in part by the German Ministry for Research and Technology (Bundesministerium für Forschung und Technologie) under grant ITS 9103 and by the ESPRIT Basic Research Actions Program under contract No. 7141 (project ALCOM II).

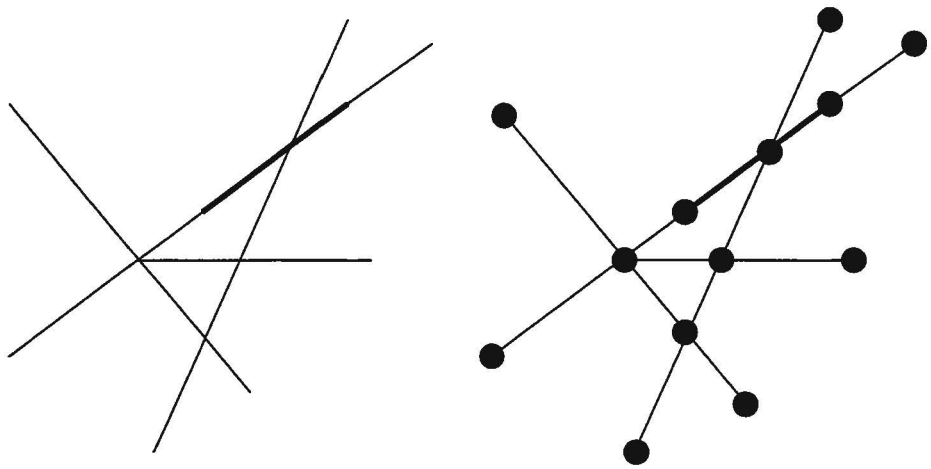


Figure 1: A set S of segments and the induced planar graph.

2. Introduction.

Let S be a set of n straight-line segments in the plane and let $G(S)$ be the graph induced by S. The vertices of $G(S)$ are all endpoints of segments in S and all intersection points between segments in S. The edges of G are the maximal relatively open and connected subsets of segments in S that contain no vertex of $G(S)$. Figure 1 shows an example. Note that the graph $G(S)$ contains no parallel edges even if S contains segments that overlap.

Bentley and Ottmann [1] have shown how to compute $G(S)$ in time $O((n+m)\log n)$ where m is the number of pairs of intersecting segments in S. The algorithm is based on the plane-sweep paradigm. We refer the reader to [5, section VIII.4] [10, section 7.2.3], and [3, section 35.2] for a discussion of the plane sweep paradigm.

In this paper we describe an implementation of the Bentley-Ottmann algorithm. More precisely, we define a procedure

sweep_segments(**list**⟨**rat_segment**⟩ &*seglist*, **GRAPH**⟨**rat_point**, **rat_segment**⟩ &G, **bool** *use_filter* = *true*)

that takes a list *seglist* of **rat_segments** (segments whose enpoints have rational coordinates) and computes the graph G induced by it. For each vertex v of G it also computes its position in the plane, a **rat_point** (a point with rational coordinates), and for each edge e of G it computes a segment containing it. The implementation is based on the LEDA platform for combinatorial and geometric computing [7, 9]. In LEDA a **rat_segment** is specified by its two endpoints (of type **rat_point**) and

a **rat_point** is specified by its homogeneous coordinates (X, Y, W) of type **Int**. The type **Int** is the type of arbitrary precision integers.

The implementation makes no assumptions about the input, in particular, segments may have length zero, may overlap, several segments may intersect in the same point, endpoints of segments may lie in the interior of other segments, and the homogeneous coordinates of the endpoints may be arbitrary integers.

We have achieved this generality by following two principles.

- We treat degeneracies as first class citizens and not as an afterthought [2]. In particular, we reformulated the plane-sweep algorithm so that it can handle all geometric situations. The details will be given in section 3. The reformulation makes the description of the algorithm shorter since we do not distinguish between three kinds of events but have only one kind of event and it also makes the algorithm faster. The algorithm now runs in time $O((n+s)\log n)$ where s is the number of vertices of G. Note that $s \leq n+m$ and that m can be as large as s^2. The only previous algorithm that could handle all degeneracies is due to Myers [6]. Its expected running time for random segments is $O(n \log n + m)$ and its worst case running time is $O((n+m)\log n)$.

- We evaluate all geometric tests exactly. We use arbitrary precision integer arithmetic for all geometric computations. So all tests are computed exactly and we do not have to worry about numerical precision. Of course, we have to pay for the overhead of arbitrary precision integer arithmetic. In order to keep the overhead low we followed the suggestion of Fortune and van Wyk [4] and implemented a floating point filter, i.e., all tests are first performed using floating point arithmetic (if *use_filter* is set to *true*) and only if the result of the floating point computation is inconclusive we perform the costly exact computation. The floating point filter improves the running time by a factor of up to 4 depending on the problem instance.

We feel that the implementation of the combinatorial part of the algorithm is quite elegant but that the implementation of the geometric part is still cumbersome. This is mainly due to the fact that the floating point is visible on the level of the aplication program (here the sweep program). We are currently exploring strategies to hide the floating point filter from the user.

This paper is structured as follows. In section 3 we describe the (generalized) plan-sweep algorithm. Section 4 and 18 give the details of the implementation: the former section describes the combinatorial part and the latter section describes the geometric primitives. The floating point filter is also discussed there. Section 29 contains some experimental results.

3. The Algorithm.

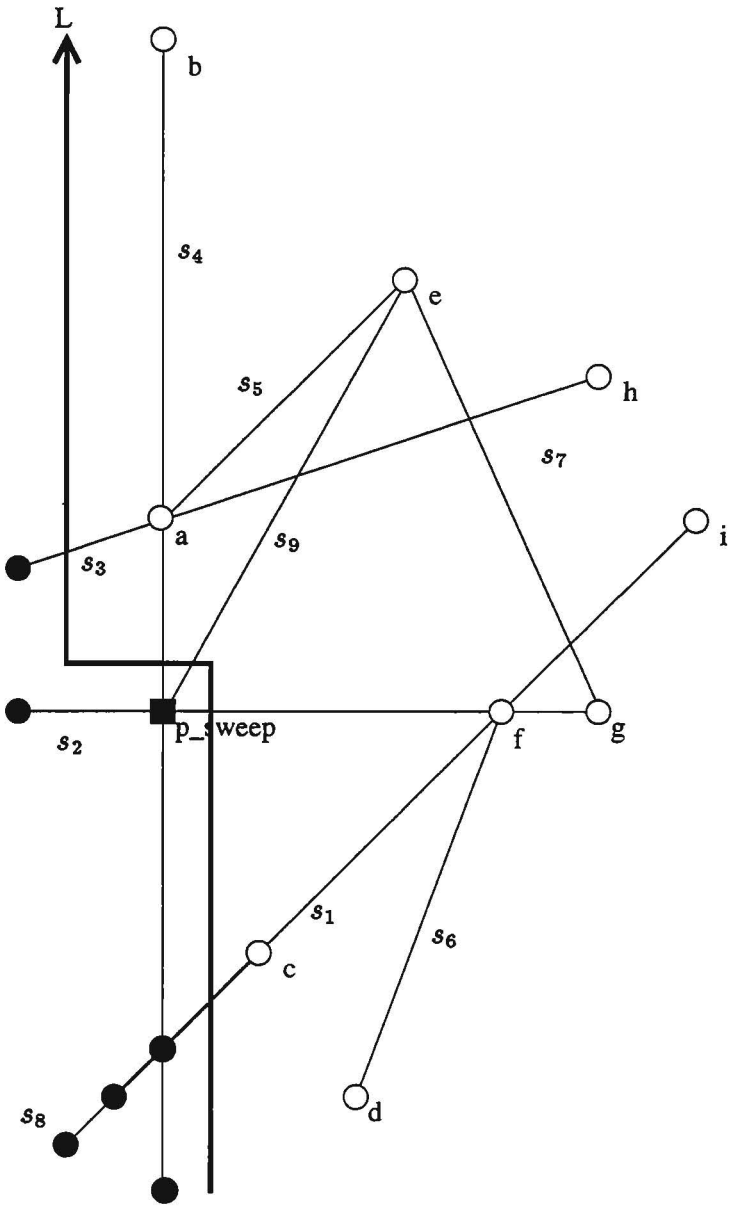
In the sweep-line paradigm a line is moved from left to right across the plane and the output (here the graph $G(S)$) is constructed incrementally as it evolves behind the sweep line. One maintains two data structures to keep the construction going: The so-called *Y-structure* contains the intersection of the sweep line with the scene (here the set S of line segments) and the so-called *X-structure* contains the events where the sweep has to be stopped in order to add to the output or to update the X- or Y-structure. In the line segment intersection problem an event occurs when the sweep line hits an endpoint of some segment or an intersection point. When an event occurs, some nodes and edges are added to the graph $G(S)$, the Y-structure is updated, and maybe some more events are generated. When the input is in general position (no three lines intersecting in a common point, no endpoint lying on a segment, no two endpoints or intersections having the same x-coordinate, no vertical lines, no overlapping segments) then at most one event can occur for each position of the sweep line and there are three clearly distiguishable types of events (left endpoint, right endpoint, intersection) with easily describable associated actions, cf. [5](section VIII.4). We want to place no restrictions on the input and therefore need to proceed slightly differently. We now describe the required changes.

We define the sweep line by a point $p\_sweep = (x\_sweep, y\_sweep)$. Let ϵ be a positive infinitesimal (readers not familiar with infinitesimals may think of ϵ as an arbitrarily small positive real). Consider the directed line L consisting of a vertical upward ray ending in point $(x\_sweep + \epsilon, y\_sweep + \epsilon)$ followed by a horizontal segment ending in $(x\_sweep - \epsilon, y\_sweep + \epsilon)$ followed by a vertical upward ray. We call L the *sweep line*. Note that no endpoint of any segment lies on L, that no two segments of S intersect L in the same point except if the segments overlap, and that no non-vertical segment of S intersects the horizontal part of L. All three properties follow from the fact that ϵ is arbitrarily small but positive. Figure 2 illustrates the definition of L and the data structures used in the algorithm: The Y-structure, the X-structure, and the graph G.

The Y-structure contains all segments intersecting the sweep line L ordered as their intersections with L appear on the directed line L. For segments intersecting L in the same point (these segments necessarily have the same underlying line) only the segment extending further to the right is stored in the Y-structure.

In the example of Figure 2 the sweep line intersects the segments s_8, s_1, s_2, s_9, s_4, and s_3. The segments s_8 and s_1 intersect L in the same point and s_1 extends further to the right. Thus s_1 is stored in the Y-structure and s_8 is not. The Y-structure therefore consists of five items, one each for segments s_1,s_2, s_9, s_4, and s_3.

The X-structure is a sorted sequence. It contains an item for each point in $St \cup E \cup I$, where St is the set of all start points of segments that lie to the right of L, E is the set of all endpoints of segments that intersect L, and I is defined as follows. Every point in I is the intersection $s \cap s'$ of two segments s and s' adjacent in the Y-structure that lies to the right of the sweep line. The ordering of the points in the



Y-structure

$< s_3, nil >$

$< s_4, xit_a >$

$< s_9, nil >$

$< s_2, nil >$

$< s_1, xit_f >$

X-structure: $< a, sit_4 >, < b, sit_4 >, < c, sit_1 >, < d, nil >, < e, sit_9 >$
$< f, sit_1 >, < g, sit_2 >, < h, sit_3 >, < i, sit_1 >$

Figure 2: A scene of 9 segments. The segments s_1 and s_8 overlap. The Y-structure contains segments s_1, s_2, s_9, s_4, and s_3 and the X-structure contains points a, b, c, d, e, f, g, h, and i. An item in the X-structure containing point p is denoted xit_p and an item in the Y-structure containing segment s_i is denoted sit_i. The vertices of the graph G are shown as full circles.

X-structure is the lexicographic ordering, i.e., (x, y) is before (x', y') if either $x < x'$ or $x = x'$ and $y < y'$.

In the example of Figure 2 we have $St = \{a, d, e\}, E = \{b, c, f, e, g, i, h\}$ and $I = \{f, a\}$. The X-structure therefore contains 9 items, one each for points a, b, c, d, e, f, g, h, and i.

We next define the informations associated with the items of both structures. These informations serve to link the items in the X-structure with the items in the Y-structure and vice versa. In particular, any item in the X-structure is a pair $\langle p,sit \rangle$ where p is a point and sit is either nil or an item in the Y-structure and any item in the Y-structure is a pair $\langle s,\text{xit} \rangle$ where s is a segment and xit is either nil or an item in the X-structure. Let $\langle p,sit \rangle$ be any item in the X-structure. If $p \in I \cup E$ then sit is an item in the Y-structure such that the segment associated with sit contains p. If $p \in St \setminus (I \cup E)$ then $sit = nil$. Next consider an item $\langle s,xit \rangle$ in the Y-structure and let s' be the segment associated with the successor item in the Y-structure. If $s \cap s'$ exists and lies to the right of the sweep line then xit is an item in the X-structure and $s \cap s'$ is the point associated with that item. If $s \cap s'$ either does not exist or does not lie to the right of L then $xit = nil$.

In our example, the Y-structure contains the items $\langle s1,xit\_f \rangle$, $\langle s2,nil \rangle$, $\langle s9,nil \rangle$, $\langle s4,xit\_a \rangle$ and $\langle s3,nil \rangle$ where xit_a and xit_f are the items of the X-structure with associated points a and f respectively. Let's turn to the items of the X-structure next. All items except $\langle d,nil \rangle$ point back to the Y-structure. If sit_i denotes the item $\langle s\_i,... \rangle$, $i \in \{1, 2, 9, 4, 3\}$, of the Y-structure then the items of the X-structure are $\langle a,sit\_4 \rangle$, $\langle b,sit\_4 \rangle$, $\langle c,sit\_1 \rangle$, $\langle d,nil \rangle$, $\langle e,sit\_9 \rangle$, $\langle f,sit\_1 \rangle$, $\langle g,sit\_2 \rangle$, $\langle h,sit\_3 \rangle$ and $\langle i,sit\_1 \rangle$.

The graph G is the part of $G(S)$ to the left of L. With each vertex of G we store its position and with each edge of G we store a segment containing it.

There is one additional piece of information that we need to keep. For each segment s contained in the Y-structure we store the rightmost vertex of G lying on s.

We can now give the details of the algorithm. Initially, G and the Y-structure are empty, and the X-structure contains the left endpoints of all segments in S.

In order to process an event we proceed as follows. Let $\langle p,sit \rangle$ be the first item in the X-structure. We may assume inductively that all invariants hold for $p\_sweep = (p.x, p.y - 2\epsilon)$. Note that this is true initially, i.e., before the first event is removed from the X-structure. We now show how to establish the invariants for $p = p\_sweep$. We proceed in seven steps.

1. We add a node v at position p to our graph G.

2. We determine all segments in the Y-structure containing the point p. These segments form a possibly empty subsequence of the Y-structure.

3. For each segment in the subsequence we add an edge to the graph G. Its right endpoint is v and its other endpoint is the vertex stored with the segment s.

4. We delete all segments ending in p from the Y-structure.

5. We reverse the order of the subsequence in the Y-structure. This amounts to moving the sweep line across the point p.

6. We add all segments starting in p to the Y-structure and then associate the node v with all segments in the Y-structure containing v. If a newly inserted segment is collinear to an already existing segment we make sure to only keep the segment extending further to the right.

7. We update the events associated with the items of the Y-structure. We remove the events associated with the predecessor of the subsequence and the last item of the subsequence and we generate new events for the predecessor of the first item and the last item after the reversal of the subsequence.

This completes the description of how to process the event $\langle p, \ldots \rangle$. The invariants now hold for $p\_sweep = p$ and hence also for $p\_sweep = (p'.x, p'.y - 2\epsilon)$ where $\langle p', \ldots \rangle$ is the new first element of the X-structure.

4. The Implementation.

The implementation follows the algorithm closely. There are two main differences.

- We add two infinitely long horizontal segments with y-coordinate $+\infty$ and $-\infty$ respectively. They serve as sentinels and simplify many tests.
- We maintain all points twice: once by their exact homogeneous coordinates and once by floating point approximations to these coordinates. All tests are first performed using the floating point approximations and only if the floating point filter gives no conclusive result the test is performed using exact arithmetic.

We use sorted sequences, graphs, rational points and segments, big integers, and a floating point filter from LEDA, and some functions of the C++ maths library. We have to include the corresponding header files.

⟨ include statements 4 ⟩ ≡

```
#include <LEDA/sortseq.h>
#include <LEDA/graph.h>
#include <LEDA/rat_point.h>
#include <LEDA/rat_segment.h>
#include <LEDA/Int.h>
#include <LEDA/Float.h>
#include <math.h>
```

This code is used in section 6.

5. Let us briefly explain these types; for a detailed discussion we refer the reader to the LEDA manual [9]. **Int** is the type of arbitrary precision integers and **Float** is the type of floating point approximation to integers. The type **Float** is defined in section 24. The types **rat_point** and **rat_segment** realize points and segments in the plane with rational coordinates. An **rat_point** is specified by its homogeneous coordinates of type **Int**. If p is a **rat_point** then $p.X(\,)$, $p.Y(\,)$, and $p.W(\,)$ return its homogeneous coordinates and if X, Y, and W are of type **Int** and $W \neq 0$ then **rat_point**(X,Y) and **rat_point**(X,Y,W) create the **rat_point** with homogeneous coordinates $(X,Y,1)$ and (X,Y,W) respectively. A **rat_segment** is specified by its two endpoints; so if p and q are **rat_point**s then **rat_segment**(p,q) constructs the directed **rat_segment** with startpoint p and endpoint q. If s is a **rat_segment** then $s.start(\,)$ and $s.end(\,)$ return the start- and endpoint of s respectively and $s.dx(\,)$ and $s.dy(\,)$ return the normalized x- and y-difference of the segment, i.e., if (px,py,pw) and (qx,qy,qw) are the homogeneous coordinates of the start- and endpoint of s then $s.dx(\,)$ returns $px \cdot qw - qx \cdot pw$ and $s.dy(\,)$ returns $py \cdot qw - qy \cdot pw$. The slope of a segment s is given by $s.dy()/s.dx()$, but be careful. The slope might be infinite (if the segment is vertical) or undefined (if the segment has length zero).

6. The Basic Program Structure.

Our program has the following structure.

```
⟨ include statements 4 ⟩
⟨ global types and declarations 7 ⟩
⟨ geometric primitives 18 ⟩
void sweep(list⟨rat_segment⟩ &S, GRAPH⟨rat_point, rat_segment⟩
        &G, bool use_filter = true)
{
  ⟨ local declarations 9 ⟩
  ⟨ initialization 10 ⟩
  ⟨ sweep 12 ⟩
  ⟨ clean-up 11 ⟩
}
```

7. During the sweep we use two local types **myPoint** and **mySegment**. They are extensions of the LEDA types **rat_point** and **rat_segment** which we use to make the program more efficient. A **myPoint** consists of a **rat_point** plus floating point approximations to the homogeneous coordinates of the point. Tests on **myPoints**, e.g. the compare function, are first evaluated using the floating point approximations and the exact test is only performed if the floating point filter gives insufficient information. The details will be described in section 18.

A **mySegment** consists of two **myPoints** p and q, the underlying LEDA segment *seg*, floating point approximations for the expressions $dx = px * qw - qx * pw$ and $dy = py * qw - qy * pw$ which are often used in the program, and the last node *last_node* of the output graph G lying on the segment (intially *nil*).

We make both types pointer types to avoid the overhead of copying. Note that the objects of both types have multiple occurrences, e.g., in **myPoints** occur in **mySegments** and also in the X-structure.

We also need to say something about memory management. Our program allocates storage for **myPoints** and **mySegments**. LEDA's memory mangement feature is used to allocate this storage in big chunks and thus to avoid the overhead of frequent calls to *malloc*. To free the memory again we use different strategies for points and segments. A segment (but not its constituent points) is deleted when it is removed from the Y-structure. All points are collected in a hand-crafted linear list and are deleted in section ⟨ clean-up 11 ⟩ at the end of *sweep*.

```
⟨ global types and declarations 7 ⟩ ≡
  class MyPointRep;
  static MyPointRep *first_myPoint = 0;
  struct MyPointRep {
    rat_point pt;
    Float x;
    Float y;
```

```
    Float w;
    int count;
    MyPointRep *next;
    MyPointRep(const rat_point &p)
    {
      pt = p;
      x = Float(p.X( ));
      y = Float(p.Y( ));
      w = Float(p.W( ));
      count = 0;
      next = first_myPoint;
      first_myPoint = this;
    }
    LEDA_MEMORY(MyPointRep)
  };
  typedef MyPointRep *myPoint;
  struct MySegmentRep {
    myPoint start;
    myPoint end;
    rat_segment seg;
    Float dx;
    Float dy;
    node last_node;
    MySegmentRep(const rat_segment &s)
    {
      start = new MyPointRep (s.start( ));
      end = new MyPointRep (s.end( ));
      seg = s;
      dx = Float(s.dx( ));
      dy = Float(s.dy( ));
      last_node = nil;
    }
    MySegmentRep(const myPoint &p)
       // creates the zero-length segment (p,p)
    {
      start = p;
      end = p;
      seg = rat_segment(p->pt,p->pt);
      last_node = nil;
    }
    LEDA_MEMORY(MySegmentRep)
  };
  typedef MySegmentRep *mySegment;
```

See also section 8.

This code is used in section 6.

8. The programs uses three global variables: *Infinity* is a big integer constant that is used as a safe approximation for ∞; it will be initialized to a power of two greater than the maximal absolute value of any input coordinate. *p_sweep* is a **myPoint** that defines the current sweep position. *use_filter* is a flag that indicates whether the floating point filter should be applied, i.e., whether floating point computations should be used before doing exact arithmetic.

```
⟨ global types and declarations 7 ⟩ +≡
  Int Infinity;
  myPoint p_sweep;
  bool use_filter;
#if defined (STATISTICS)
  int cmp_points_count;
  int cmp_points_failed;
  int exact_cmp_points_count;
  int cmp_segments_count;
  int exact_cmp_segments_count;
#endif
```

9. In the local declarations section of function *sweep* we introduce the data types for the event queue (*X_structure*) and for the sweep line (*Y_structure*). For the X-structure we use a sorted sequence of points with the lexicographic ordering of their coordinates, and for the Y-structure a sorted sequence of segments with the linear order defined by the sequence intersections of the segments with the sweep line at its current position (*p_sweep*) from bottom to top.

The X-structure contains all so far known event points right and above of the current sweep line position (*p_sweep*), i.e. all start and end points of segments and intersections of segments being adjacent in the sweep line. The X-structure associates with each event point p an item of the Y-structure (**seq_item**) as information; if there are only starting segments at p the information is *nil*, otherwise the information is an item in the Y-structure containing one of the segments passing through or ending in p.

Vice versa we associate with each segment s in the Y-structure that intersects its successor in some point p the corresponding item $< p, \ldots >$ in the X-structure as information. We call this item the *current intersection item* of s. If s does not intersect its successor its current intersection item is *nil*. Furthermore, we maintain for every point p a counter *count* that gives the number of all segments having $< p, \ldots >$ as current intersection item. We will use this counter to decide whether a point can be removed from the X-structure. It can be removed if *count* is zero. Finally, we store with each segment s the last created node of the output graph *last_node* lying on s.

⟨ local declarations 9 ⟩ ≡
  sortseq⟨myPoint, seq_item⟩ *X_structure*;
  sortseq⟨mySegment, seq_item⟩ *Y_structure*;

This code is used in section 6.

10. We now come to the initialization of the data structures. We first compute a big integer *Infinity* that can be used as a safe approximation for ∞. We start the sweep at $(-\infty, -\infty)$. At its initial position the sweep line (i.e. the Y-structure) contains two infinitely long horizontal segments (*lowersentinel* and *uppersentinel*) with y-coordinates $-\infty$ and $+\infty$ respectively, and the output graph G is empty.

We create for each **rat_segment** in S the corresponding **mySegment** (reorienting if necessary) and insert its left endpoint together with the information *nil* into the X-structure. We use the fact that a sorted sequence contains at most one item for every key and that a second insert operation with the same key only changes the information of the item to make the left endpoints of all segments unique. This makes equality tests between endpoints of segments much cheaper, since now the **myPoint** pointers can be compared directly (using the $\equiv$ operator) instead of having to call an expensive compare function.

Finally, we sort all segments according to their left endpoints into a list *S_Sorted* by calling the LEDA list sorting operation *S_Sorted.sort*(*cmp_seg*). Here *cmp_seg* is a compare function defined in section 18.

⟨ initialization 10 ⟩ ≡
#if defined (*STATISTICS*)
  cmp_points_count = 0;
  cmp_points_failed = 0;
  exact_cmp_points_count = 0;
  cmp_segments_count = 0;
  exact_cmp_segments_count = 0;
#endif
  ::*use_filter* = *use_filter*;
  /* compute an upper bound *Infinity* for the input coordinates */
  Infinity = 1;
  rat_segment *s*;
  forall (*s*, *S*)
    while (*abs*(*s.X1*( )) ≥ *Infinity* ∨ *abs*(*s.Y1*( )) ≥ *Infinity* ∨ *abs*(*s.X2*( )) ≥ *Infinity* ∨ *abs*(*s.Y2*( )) ≥ *Infinity*) *Infinity* *= 2;
  p_sweep = **new MyPointRep** (**rat_point**(−*Infinity*, −*Infinity*));
  mySegment *uppersentinel* = **new MySegmentRep** (**rat_segment**(−*Infinity*, *Infinity*, *Infinity*, *Infinity*));
  mySegment *lowersentinel* = **new MySegmentRep** (**rat_segment**(−*Infinity*, −*Infinity*, *Infinity*, −*Infinity*));

```
  Y_structure.insert(uppersentinel, nil);
  Y_structure.insert(lowersentinel, nil);
  G.clear( );
  list⟨mySegment⟩ S_Sorted;
  forall (s, S) {
    /* mySegments are always oriented from left to right or (if vertical) from
    bottom to top */
    if (s.X1( ) > s.X2( ) ∨ (s.X1( ) ≡ s.X2( ) ∧ s.Y1( ) > s.Y2( )))
      s = rat_segment(s.end( ), s.start( ));
    mySegment s1 = new MySegmentRep (s);
    S_Sorted.append(s1);
    seq_item it = X_structure.insert(s1→start, nil);
    s1→start = X_structure.key(it);
    s1→start→count++;
  }
  S_Sorted.sort(cmp_mySeg);
```

This code is used in section 6.

11. To clean everything up we need to remove the two sentinels and all **myPoints**.

```
⟨ clean-up 11 ⟩ ≡
  {
    delete (uppersentinel);
    delete (lowersentinel);
    myPoint p = first_myPoint;
    while (first_myPoint ≠ nil) {
      p = first_myPoint→next;
      delete (first_myPoint);
      first_myPoint = p;
    }
#if defined (STATISTICS)
    if (use_filter) {
      cout ≪ string("compare␣points:␣␣%6d␣/␣%4d␣(%5.2f\%)␣␣(\
          %5.2f\%␣failed)␣␣", cmp_points_count, exact_cmp_points_count,
          (100.0 * exact_cmp_points_count)/cmp_points_count,
          (100.0 * cmp_points_failed)/cmp_points_count);
      newline;
      cout ≪ string("compare␣segments:␣%6d␣/␣%4d␣␣(%5.2f\%)",
          cmp_segments_count, exact_cmp_segments_count,
          (100.0 * exact_cmp_segments_count)/cmp_segments_count);
      newline;
    }
#endif
```

```
}
```

This code is cited in section 7.

This code is used in section 6.

12. We now come to the heart of procedure sweep: the processing of events. Let $event =< p, sit >$ be the first event in the X-structure and assume inductively that our data structure is correct for $p\_sweep = (p.x, p.y - 2\epsilon)$. Our goal is to to change p_sweep to p, i.e., move the sweep line across the point p. We perform the following actions.

We first add a vertex v with position p to the output graph G. Then, we handle all segments passing through or ending at p_sweep. Finally, we insert all segments starting at p_sweep into the Y-structure, check for possible intersections between pairs of segments now adjacent in the Y-structure, and update the X-structure. After having processed the *event* we delete it from the X-structure.

```
⟨ sweep 12 ⟩ ≡
  while (¬X_structure.empty( )) {
    seq_item event = X_structure.min( );
    seq_item sit = X_structure.inf(event);
    myPoint p = X_structure.key(event);
    node v = G.new_node(p.pt);
    p_sweep = p;
    ⟨ handle passing and ending segments 13 ⟩
    ⟨ insert starting segments and compute new intersections 17 ⟩
    X_structure.del_item(event);
  }
```

This code is used in section 6.

13. We first handle all segments passing through or ending in point p. How can we find them?

Recall that the current event is $< p, sit >$ and that $sit \neq nil$ iff $p \in I \cup E$. If $sit \neq nil$ then p is contained in the segment associated with sit. If $sit = nil$ then $p \in St \backslash (I \cup E)$. In this case there is at most segment in the Y-structure containing p. We may determine this segment by looking up the zero-length segment $(p.pt, p.pt)$ in the Y-structure. We explain in section 22 why this works.

After the lookup we have either $sit = nil$ and then no segment in the Y-structure contains p or $sit \neq nil$ and then the segment associated with sit contains p. In the latter case we determine all such segments and update the graph G and the Y-structure.

We also declare two items *sit_pred* and *sit_succ* and initialize them to *nil*. If the Y-structure contains a segment containing p then *sit_pred* and *sit_succ* will be the set to the predecessor and successor item of the subsequence of segments containing p, otherwise they stay *nil*.

```
⟨ handle passing and ending segments 13 ⟩ ≡
  seq_item sit_succ = nil;
  seq_item sit_pred = nil;
  if (sit ≡ nil) {
    MySegmentRep s(p);      // create a zero length segment s = (p,p)
    sit = Y_structure.lookup(&s);
  }
  if (sit ≠ nil) {
    ⟨ find subsequence of ending or passing segments 14 ⟩
    ⟨ construct edges and delete ending segments 15 ⟩
    ⟨ reverse subsequence of segments passing through p 16 ⟩
  }
```

This code is used in section 12.

14. Taking *sit* as an entry point into the Y-structure we determine all segments incident to p from the left or from below. These segments form a subsequence of the Y-structure. Let *sit_first* and *sit_last* denote the first and the last item of the subsequence and let *sit_pred* be the predecessor of *sit_first* and *sit_succ* the successor of *sit_last*. Note that the information of all items in this subsequence is equal to the current event item *event*, except for *sit_last* whose information is either *nil* or a different item in the X-structure resulting from an intersection with *sit_succ*.

Note also that the identification of the subsequence of segments incident to p takes constant time per element of the sequence. Moreover, the constant is small since the test whether p is incident to a segment involves no geometric computation but only equality tests between items.

Note finally that the code is particularly simple due to our sentinel segments: *sit_first* can never be the first item of the Y-structure and *sit_last* can never be the last.

```
⟨ find subsequence of ending or passing segments 14 ⟩ ≡
  seq_item sit_last = sit;
  while (Y_structure.inf(sit_last) ≡ event) sit_last = Y_structure.succ(sit_last);
  sit_succ = Y_structure.succ(sit_last);
  sit_pred = Y_structure.pred(sit);
  while (Y_structure.inf(sit_pred) ≡ event) sit_pred = Y_structure.pred(sit_pred);
  seq_item sit_first = Y_structure.succ(sit_pred);
```

This code is used in section 13.

15. We can now add edges to the graph G. For each segment in the subsequence between *sit_first* and *sit_last* inclusive we construct an edge. Let *sit* be any such item and let s be the segment associated with *sit*. We construct an edge connecting *s→last_node* and v and label it with the segment s. We also either delete the item

from the Y-structure (if the segment ends at p) or change its information to *nil* (if the segment does not end at p) to reflect the fact that no intersection event is now associated with the segment. In the former case we free the storage reserved for the segment.

At the end we have to update variables *sit_first* and *sit_last* since the corresponding items may have been deleted.

```
⟨ construct edges and delete ending segments 15 ⟩ ≡
  seq_item i1 = sit_pred;
  seq_item i2 = sit_first;
  while (i2 ≠ sit_succ) {
    mySegment s = Y_structure.key(i2);
    G.new_edge(s->last_node, v, s->seg);
    s->last_node = v;
    if (p ≡ s->end)      // ending segment
    {
      Y_structure.del_item(i2);
      delete s;
    }
    else {      // continuing segment
      if (i2 ≠ sit_last)  Y_structure.change_inf(i2, nil);
      i1 = i2;
    }
    i2 = Y_structure.succ(i1);
  }
  sit_first = Y_structure.succ(sit_pred);
  sit_last = Y_structure.pred(sit_succ);
```

This code is used in section 13.

16. All segments remaining in the subsequence pass through node v and moving the sweep line through *p_sweep* inverses the order of the segments in the subsequence. The subsequence is non-empty iff *sit_last* $\neq$ *sit_pred*. Reversing the subsequence destroys the adjacency of pairs (*sit_pred*,*sit_first*) and (*sit_last*,*sit_succ*) in the Y-structure and hence we have to set the current intersection event of *sit_pred* and *sit_last* (i.e. the associated information in the Y-structure) to *nil*, decrement the corresponding counters, and delete the items from the X-structure if the counters are zero now. If the subsequence is empty we only need to change the intersection event associated of *sit_pred* to *nil*. Finally, we reverse the subsequence by calling *Y_structure.reverse_items*(*sit_first*, *sit_last*).

```
⟨ reverse subsequence of segments passing through p 16 ⟩ ≡
  seq_item xit = Y_structure.inf(sit_pred);
  if (xit ≠ nil) {
    if (--X_structure.key(xit)->count ≡ 0)  X_structure.del_item(xit);
```

```
    Y_structure.change_inf(sit_pred, nil);
  }
  if (sit_last ≠ sit_pred) {
    xit = Y_structure.inf(sit_last);
    if (xit ≠ nil) {
      if (--X_structure.key(xit)->count ≡ 0) X_structure.del_item(xit);
      Y_structure.change_inf(sit_last, nil);
    }
    Y_structure.reverse_items(sit_first, sit_last);
  }
```

This code is used in section 13.

17. The last step in handling the event point p is to insert all segments starting at p into the Y-structure and to test the new pairs of adjacent items (*sit_pred*,...) and (...,*sit_succ*) for possible intersections. If there were no segments passing through or ending in p then the items *sit_succ* and *sit_pred* are still *nil* and we have to compute them now.

We use the sorted list *S_Sorted* to find all segments to be inserted. As long as the first segment *seg* in *S_Sorted* starts at point p we remove it from the list, insert it into the Y-structure, add its right endpoint to the X-structure, and set *seg->last_node* to v. If the segment has length zero we simply discard it and if *sit_succ* and *sit_pred* are still undefined (*nil*) then we use the first inserted segment to define them. In this case, we also have to change the possible current intersection event of *sit_pred* to *nil* since it is no longer adjacent to *sit_succ*. If all segments starting in p have length zero then *sit_succ* and *sit_pred* remain undefined.

We need to say more clearly how to insert a segment s into the Y-structure. If the Y-structure contains no segment with the same underlying line then we simply add the segment. Otherwise, let s' be the segment in the Y-structure with the same underlying line. We replace s by s' if s' extends further to the right than s and do nothing otherwise.

After having inserted all segments starting at p, we test whether *sit_succ* (*sit_pred*) intersects its predecessor (successor) in the Y-structure.

```
⟨ insert starting segments and compute new intersections 17 ⟩ ≡
  while (¬S_Sorted.empty( ) ∧ p ≡ S_Sorted.head( )->start) {
    mySegment Seg = S_Sorted.pop( );
    /* first insert the right endpoint of Seg into the X-structure */
    seq_item end_it = X_structure.insert(Seg->end, nil);
    Seg->end = X_structure.key(end_it);
    Seg->end->count++;
    /* note that the following test uses the fact that two endpoints are equal if an
    only if the corresponding pointer values (myPoints) are equal */
    if (Seg->start ≡ Seg->end)      // Seg has length zero, nothing to do
```

```
    {
      delete Seg;
      continue;
    }
    sit = Y_structure.locate(Seg);
    if (compare(Seg, Y_structure.key(sit)) ≠ 0) {
      /* Seg is not collinear with the segment associated with sit. We simply insert
      Seg into the Y-structure */
      sit = Y_structure.insert_at(sit, Seg, nil);
      Seg->last_node = v;
    }
    else {
      /* Seg is collinear with the segment associated with sit. If Seg is longer then
      we use Seg and otherwise we do nothing. */
      mySegment Seg_old = Y_structure.key(sit);
      if (compare(Seg->end, Seg_old->end) > 0) {
        /* Seg extends further to the right or above replace Seg_old by Seg. */
        Seg_old->seg = Seg->seg;
        Seg_old->end = Seg->end;
      }
      delete Seg;      // not needed anymore
    }
    X_structure.change_inf(end_it, sit);
    if (sit_succ ≡ nil) {
      sit_succ = Y_structure.succ(sit);
      sit_pred = Y_structure.pred(sit);
      /* sit_pred is no longer adjacent to sit_succ we have to change its current
      intersection event to nil and delete the corresponding item in the X-structure
      if necessary */
      seq_item xit = Y_structure.inf(sit_pred);
      if (xit ≠ nil) {
        if (--X_structure.key(xit)->count ≡ 0) X_structure.del_item(xit);
        Y_structure.change_inf(sit_pred, nil);
      }
    }
  }
  /* compute possible intersections */
  if (sit_succ ≠ nil)      // v is an isolated vertex otherwise
  {
    compute_intersection(X_structure, Y_structure, sit_pred);
    sit = Y_structure.pred(sit_succ);
    if (sit ≠ sit_pred) compute_intersection(X_structure, Y_structure, sit);
  }
```

This code is used in section 12.

18. Geometric Primitives.

It remains to define the geometric primitives used in the implementation. We need four:

- a compare function for **myPoints** which orders points according to the lexicographic ordering of their coordinates. It defines the linear order used in the X-structure.
- a compare-function for **mySegments** which given two segments intersecting the sweep line L determines the order of the intersections on L. It defines the linear order used in the Y-structure.
- a second compare function *cmp_mySeg* for **mySegments** which orders segments according to the order of their left endpoint. It is used to sort the list *S_Sorted* in the beginning.
- a function *compute_intersection* that decides whether two segments intersect and if so whether the intersection is to the right of the sweep line. If both tests are positive it also makes the required changes to the X- and Y-structure.

We define the compare functions for **myPoints** and **mySegments** in two steps. We first write two function templates *cmp_points* and *cmp_segments* for comparing points and segments given by their homogenous coordinates, independently from the actual (numerical) type of the coordinates. These templates can be used to compare points and segments with coordinates of any integer type *int_type* that supports addition, subtraction, multiplication, and a special sign function $Sign(x)$ that returns $+1$ if $x > 0$, 0 if $x = 0$, -1 if $x < 0$, and a special integer value *NO_IDEA* if the test cannot be performed in a conclusive way.

We start with writing a compare function *cmp_points*(*x1*, *y1*, *w1*, *x2*, *y2*, *w2*) that compares two points with homogeneous coordinates (x_1, y_1, w_1) and (x_2, y_2, w_2). A point (x_1, y_1, w_1) precedes a point (x_2, y_2, w_2) if the pair $(x_1/w_1, y_1/w_1)$ lexicographically precedes the pair $(x_2/w_2, y_2/w_2)$, i.e., if $(x_1 w_2 > x_2 w_1)$ or $(x_1 w_2 = x_2 w_1$ and $y_1 w_2 > y_2 w_1)$.

```
⟨ geometric primitives 18 ⟩ ≡
  template⟨class int_type⟩ inline int cmp_points(const int_type&x1, const
          int_type&y1, const int_type&w1, const int_type&x2, const
          int_type&y2, const int_type&w2)
  {
    int s = Sign(x1 * w2 - x2 * w1);
    return (s ≠ 0) ? s : Sign(y1 * w2 - y2 * w1);
  }
```

See also sections 19, 20, 21, and 23.

This code is used in section 6.

19. Next we write the compare function *cmp_segments* for segments.

A segment is stored as the pair of its endpoints and a point is stored by its homogeneous coordinates, i.e., as a triple (X, Y, W) of integers (type **Int**). The x- and y-coordinates of the point are X/W and Y/W respectively. *cmp_segments* takes the coordinates of two segments s_1 and s_2 intersecting the sweep line L. We assume that both segments have non-zero length and treat the case that one of them has zero length in the next section. If both segments have the same underlying line then we return 0. So let us assume that the underlying lines are different. Let l denote the vertical line through *p_sweep*. Only one of the segments can be vertical. If s_1 is vertical and hence intersects L in $(x\_sweep, y\_sweep + \epsilon)$ then s_1 is before s_2 iff *p_sweep* is below $l \cap s2$. If s_2 is vertical then s_1 is before s_2 iff $l \cap s1$ is not above *p_sweep*. Assume now that neither s_1 nor s_2 is vertical. If $l \cap s1$ and $l \cap s2$ are distinct then s_1 is before s_2 iff $l \cap s1$ is below $l \cap s2$. If $l \cap s1$ and $l \cap s2$ are identical and not above *p_sweep* then s_1 is before s_2 iff s_1 has the smaller slope. Finally, if the intersections are identical and above *p_sweep* then s_1 is before s_2 iff s_1 has the larger slope.

```
⟨ geometric primitives 18 ⟩ +≡
  template⟨class int_type⟩ int cmp_segments(const int_type&px, const
          int_type&py, const int_type&pw, const int_type&spx, const
          int_type&spy, const int_type&spw, const int_type&sqx, const
          int_type&sqy, const int_type&sqw, const int_type&rx, const
          int_type&ry, const int_type&rw, const int_type&dx, const
          int_type&dy, const int_type&sdx, const int_type&sdy)
  { /* We first test whether the underlying lines are identical. The lines are identical if the three slopes dy/dx, sdy/sdx, and mdy/mdx are equal */
    int_type T1 = dy * sdx - sdy * dx;
    int sign1 = Sign(T1);
    if (sign1 ≡ 0 ∨ sign1 ≡ NO_IDEA) {
      int_type mdx = sqx * pw - px * sqw;
      int_type mdy = sqy * pw - py * sqw;
      int sign2 = Sign(dy * mdx - mdy * dx);
      if (sign2 ≡ 0 ∨ sign2 ≡ NO_IDEA) {
        int sign3 = Sign(sdy * mdx - mdy * sdx);
        if (sign3 ≡ 0 ∨ sign3 ≡ NO_IDEA)
          return (sign3 ≡ 0 ∧ sign3 ≡ 0 ∧ sign3 ≡ 0) ? 0 : NO_IDEA;
      }
    }
      /* The underlying lines are different; in particular, at most one of the lines is vertical. We first deal with the cases that one of the lines is vertical. A segment (p, q) is vertical iff px * qw - qx * pw is equal to zero. Since dx is an optimal floating point approximation of this integer value, a segment is vertical iff its dx-value is zero. */
    if (dx ≡ 0) {
```

/* $dx = 0$, i.e., $s1$ is vertical and $s2$ is not vertical; $l \cap s2$ is above p_sweep iff $(spy/spw + sdy/sdx(rx/rw - spx/spw) - ry/rw) > 0$; in this case we return -1 */
int $i = Sign((spy * sdx - spx * sdy) * rw + (sdy * rx - ry * sdx) * spw)$;
if $(i \equiv NO\_IDEA)$ **return** NO_IDEA;
return $(i \leq 0)$? 1 : -1;
}
if $(sdx \equiv 0)$ {
/* $sdx = 0$, i.e., $s2$ is vertical but $s1$ is not vertical; we return -1 if $l \cap s1$ is below or equal to p_sweep iff $(py/pw + dy/dx(rx/rw - px/pw) - ry/rw) \leq 0$. */
int $i = Sign((py * dx - px * dy) * rw + (dy * rx - ry * dx) * pw)$;
if $(i \equiv NO\_IDEA)$ **return** NO_IDEA;
return $(i \leq 0)$? -1 : 1;
}
/* Neither $s1$ nor $s2$ is vertical. We compare $l \cap s1$ and $l \cap s2$. We have

$$y(l \cap s1) - y(l \cap s2) = \frac{py}{pw} + \frac{dy}{dx}(\frac{rx}{rw} - \frac{px}{pw}) - \frac{spy}{spw} - \frac{sdy}{sdx}(\frac{rx}{rw} - \frac{spx}{spw}).$$

If the difference is non-zero then we return its sign. If the difference is zero then we return -1 iff the common intersection is not above p_sweep and $s1$ has the smaller slope or the intersection is above p_sweep and $s1$ has the larger slope. */
int_type $T2 = sdx * spw * (py * dx * rw + dy * (rx * pw - px * rw)) - dx * pw * (spy * sdx * rw + sdy * (rx * spw - spx * rw))$;
int $sign2 = Sign(T2)$;
if $(sign2 \equiv NO\_IDEA)$ **return** NO_IDEA;
if $(sign2 \neq 0)$ **return** $sign2$;
/* Now we know that the difference is zero, i.e., $s1$ and $s2$ intersect in a point I. We compare slopes:
$s1$ has larger slope than $s2$ iff $T1 * dxs * dx > 0$; note that orienting the lines from left to right makes all dx values non-negative , i.e., $s1$ has larger slope than $s2$ iff $sign(T1) = 1$ */
int_type $T3 = (py * dx - px * dy) * rw + (dy * rx - ry * dx) * pw$;
/* The common intersection I is above p_sweep iff $T3 * rw * dx * pw > 0$. In this case we return $-sign(T1)$ and $sign(T1)$ otherwise. Note that all dx and w values are non-negative i.e., $sign(T3 * rw * dx * pw) = sign(T3)$ */
int $sign3 = Sign(T3)$;
if $(sign3 \equiv NO\_IDEA)$ **return** NO_IDEA;
return $(sign3 \leq 0)$? $sign1$: $-sign1$;
}

20. Finally, we define the compare functions for **myPoints** and **mySegments** by first calling *cmp_points* and *cmp_segments* on the floating point filter coordinates

(of type **Float**) of the corresponding points and segments. In the case that these calls do not return a reliable result (i.e. return *NO_IDEA*) we call them again with the exact coordinates (of type **Int**).

```
⟨ geometric primitives 18 ⟩ +≡
  int compare(const myPoint &a, const myPoint &b)
  {
  /* floating point filter version for myPoints */
#if defined (STATISTICS)
    cmp_points_count++;
#endif
    int c = NO_IDEA;
    /* if not explicitely turned off we first use floating point arithmetic */
    if (use_filter) c = cmp_points(a->x, a->y, a->w, b->x, b->y, b->w);
    /* if the floating point computation is not reliable, i.e., the result is NO_IDEA
    we use exact arithmetic (Int) */
    if (c ≡ NO_IDEA) {
      c = cmp_points(a->pt.X(), a->pt.Y(), a->pt.W(), b->pt.X(), b->pt.Y(), b->pt.W());
#if defined (STATISTICS)
      exact_cmp_points_count++;
      if (cmp_points(double(a->x), double(a->y), double(a->w), double(b->x),
            double(b->y), double(b->w)) ≠ c) cmp_points_failed++;
#endif
    }
    return c;
  }
  int compare(const mySegment &s1, const mySegment &s2)
  {
    int c = NO_IDEA;
    /* if not explicitely turned off we first try the floating point computation */
#if defined (STATISTICS)
    cmp_segments_count++;
#endif
    if (use_filter) c = cmp_segments(s1->start->x, s1->start->y, s1->start->w,
        s2->start->x, s2->start->y, s2->start->w, s2->end->x, s2->end->y, s2->end->w,
        p_sweep->x, p_sweep->y, p_sweep->w, s1->dx, s1->dy, s2->dx, s2->dy);
    /* If the result is not reliable we call the exact compare for the underlying
    rat_segments. */
    if (c ≡ NO_IDEA) {
      c = cmp_segments(s1->seg.X1(), s1->seg.Y1(), s1->seg.W1(), s2->seg.X1(),
          s2->seg.Y1(), s2->seg.W1(), s2->seg.X2(), s2->seg.Y2(), s2->seg.W2(),
          p_sweep->pt.X(), p_sweep->pt.Y(), p_sweep->pt.W(), s1->seg.dx(),
          s1->seg.dy(), s2->seg.dx(), s2->seg.dy());
#if defined (STATISTICS)
      exact_cmp_segments_count++;
```

```
#endif
    }
    return c;
  }
```

21. For the initialization of the list *S_Sorted* we need a second compare function *cmp_mySeg* for **mySegments**. It simply compares the left endpoints.

```
⟨ geometric primitives 18 ⟩ +≡
  int cmp_mySeg(const mySegment &s1, const mySegment &s2)
  {
    int c = NO_IDEA;
    if (use_filter)      // floating point compare
      c = cmp_points(s1→start→x, s1→start→y, s1→start→w, s2→start→x, s2→start→y,
          s2→start→w);
    if (c ≡ NO_IDEA)     // exact compare
      c = cmp_points(s1→seg.X1( ), s1→seg.Y1( ), s1→seg.W1( ), s2→seg.X1( ),
          s2→seg.Y1( ), s2→seg.W1( ));
    return c;
  }
```

22. What does compare do when one of the segments, say *s1*, has both endpoints equal to *p_sweep* and the Y-structure contains at most one segment containing *p_sweep*. That's exactly the situation in section 13. When *s2* contains *p_sweep* the underlying lines are found identical and compare returns 0. When *s2* does not contain *p_sweep* then the underlying lines are declared different. Also *s1* is declared vertical and *s2* is not vertical since it would otherwise contain *p_sweep*. We now compare $l \cap s2$ and *p_sweep* and return the result. We conclude that the call *Y_structure.locate*(s) where s is the zero-length segment (*p→pt*, *p→pt*) in section 13 has the desired effect.

23. Finally, we define a function *compute_intersection* that takes an item *sit0* of the Y-structure and determines whether the segment associated with *sit0* intersects the segment associated with the successor of *sit0* right or above of the sweep line. If so it updates the X- and the Y-structure.

The function first tests whether the underlying straight lines intersect right or above of the sweep line by comparing their slopes. This test is performed with floating point arithmetic and repeated with exact arithmetic if the floating point test gives no reliable result. If the segments do intersect right or above of the sweep line it is checked whether the point of intersection lies on both segments, i.e., is not larger than the endpoints of the segments.

```
⟨ geometric primitives 18 ⟩ +≡
  void compute_intersection(sortseq⟨myPoint, seq_item⟩ &X_structure,
          sortseq⟨mySegment, seq_item⟩ &Y_structure, seq_item sit0)
  {
    seq_item sit1 = Y_structure.succ(sit0);
    mySegment seg0 = Y_structure.key(sit0);
    mySegment seg1 = Y_structure.key(sit1);
    /* seg1 is the successor of seg0 in the Y-structure, hence, the underlying lines
    intersect right or above of the sweep line iff the slope of seg0 is larger than the
    slope of seg1. */
    if (use_filter) {
      int i = Sign(seg0->dy * seg1->dx - seg1->dy * seg0->dx);
      if (i ≡ −1 ∨ i ≡ 0) return;
      /* slope(s0) ≤ slope(s1) */
    }
    rat_segment s0 = seg0->seg;
    rat_segment s1 = seg1->seg;
    Int w = s0.dy( ) * s1.dx( ) − s1.dy( ) * s0.dx( );
    if (sign(w) > 0)      // slope(s0) > slope(s1)
    {
      Int c1 = s0.X2( ) * s0.Y1( ) − s0.X1( ) * s0.Y2( );
      Int c2 = s1.X2( ) * s1.Y1( ) − s1.X1( ) * s1.Y2( );
      /* The underlying lines intersect in a point right or above of the sweep line.
      We still have to test whether it lies on both segments. */
      Int x = c2 * s0.dx( ) − c1 * s1.dx( );
      Int d0 = x * s0.W2( ) − s0.X2( ) * w;
      if (sign(d0) > 0) return;
      if (x * s1.W2( ) > s1.X2( ) * w) return;
      Int y = c2 * s0.dy( ) − c1 * s1.dy( );
      if (sign(d0) ≡ 0 ∧ y * s0.W2( ) > s0.Y2( ) * w) return;
      myPoint Q = new MyPointRep (rat_point(x, y, w));
      seq_item xit = X_structure.insert(Q, sit0);
      X_structure.key(xit)->count++;
      Y_structure.change_inf(sit0, xit);
    }
  }
```

24. A Floating Point Filter.

The type **Float** provides a clean and efficient way to approximately compute with large integers. Consider an expression E with integer operands and operators $+, -$, and $*$, and suppose that we want to determine the sign of E. In general, the integer arithmetic provided by our machines does not suffice to evaluate E since intermediate results might overflow. Resorting to arbitrary precision integer arithmetic is a costly process. An alternative is to evaluate the expression using floating point arithmetic, i.e., to convert the operands to doubles and to use floating-point addition, subtraction, and multiplication. Of course, only an approximation $\tilde{E}$ of the true value E^* is computed. However, $\tilde{E}$ might still be able to tell us something about the sign of E. If $\tilde{E}$ is far away from zero (the forward error analysis carried out in the next section gives a precise meaning to "far away") then the signs of $\tilde{E}$ and E agree and if $\tilde{E}$ is zero then we may be able to conclude under certain circimstances that E is zero. Again, forward error analysis can be used to say what 'certain circumstances' are. The type **Float** encapsulates this kind of approximate integer arithmetic. Any integer (= object of type **Int**) can be converted to a **Float**, **Float**s can be added, subtracted, multiplied, and their sign can be computed: for any **Float** x the function $Sign(x)$ returns either the sign of x (-1 if $x < 0$, 0 if $x = 0$, and $+1$ if $x > 0$) or the special value *NO_IDEA*. If x approximates X, i.e., X is the integer value abtained by an exact computation, then $Sign(x) \neq NO\_IDEA$ implies that $Sign(x)$ is actually the sign of X if $Sign(x) = NO\_IDEA$ then no claim is made about the sign of X.

Declaration

Float x declares x as a variable of type **Float**

Operations

Float Float(Int i**)**	converts i to a float
Float $+$**(Float** a**, Float** b**)**	Addition
Float $-$**(Float** a**, Float** b**)**	Subtraction
Float $*$**(Float** a**, Float** b**)**	Multiplication
$\{-1, 0, +1, NO\_IDEA\}$*Sign*(**Float** x)	as described above

A **Float** is represented by a double (its value) and an error bound. An operation on **Float**s performs the corresponding operation on the values and also computes the error bound for the result. For this reason the cost of a **Float** operation is about four times the cost of the corresponding operation on doubles. Rules 5 to 10 below are used to compute the error bounds.

25. Floating Point Numbers.

A *floating point number* is a number of the form

$$a = m \cdot 2^e,$$

*We misuse notation and use E to denote the expression and its value.

where $e \in [e_{min}..e_{max}]$ is the *exponent*, and m is the *mantissa*. The mantissa is a rational number of the form

$$m = v \cdot \sum_{1 \le i \le l} m_i \cdot 2^{-i},$$

where $v \in \{-1, 1\}$ is the sign, l is the *mantissa length*, and the m_i's are the *digits* of the mantissa. We have $m_i \in \{0, 1\}$ and either $m = 0$ or $m_1 = 1$. One also says that the mantissa is *normalized*. We also assume that $l \le e_{\max}$. The latter assumption implies that all integers whose absolute value is less than 2^{l+1} can be presented as floating point numbers.

Let $\mathcal{F}$ denote the set of all floating point numbers; the set $\mathcal{F}$ clearly depends on the exponent range $[e_{min}..e_{max}]$ and the mantissa length l. In our examples, we use $l = 53$, $e_{min} = -1022$, and $e_{max} = 1023$. This choice corresponds to the IEEE double precision floating point standard.

The absolute values of all non-zero floating point numbers lie between $min\_abs = 2^{e_{\min}-1}$ and $max\_abs = (1 - 2^{-l})2^{e_{\max}}$. Call a real number *representable* if it is either 0 or its absolute value lies in the interval $[min\_abs, max\_abs]$ and let the size $size(a)$ of a real number a be the smallest power of two larger than its absolute value,i.e.,

$$size(a) = \begin{cases} 0 & \text{if } a = 0 \\ 2^{\lceil \log |a| \rceil} & \text{if } a \neq 0 \end{cases}$$

Then $size(a)/2 \le |a| \le size(a)$. If a number a is representable then there is a floating point number $fl(a) \in \mathcal{F}$ such that

$$|a - fl(a)| \le 1/2 \cdot 2^{-l} \cdot size(a) \le 2^{-l}|a| \tag{1}$$

The number $fl(a)$ can be obtained from a by rounding in the l-th most significant position. We call $fl(a)$ a *floating point approximation* of a and use eps to denote 2^{-l}. Note that $fl(a)$ is a floating point number that is closest to a. The set $\mathcal{F}$ of floating point numbers is not closed under the arithmetic operations addition, subtraction, and multiplication, i.e., if f_1 and f_2 are floating point numbers and op is any one of the arithmetic operations then $f_1 op f_2$ does not necessarily belong to $\mathcal{F}$. The machine implementation of the operation (we use $\oplus, \ominus$, and $\odot$ to denote the implementation of $+, -$, and $\cdot$ respectively) therefore returns a floating point approximation of the exact result, i.e., a number $\tilde{f}$ such that

$$|\tilde{f} - f_1 op f_2| \le 1/2 \cdot eps \cdot size(f_1 op f_2) \le eps \cdot |f_1 op f_2| \tag{2}$$

or equivalently

$$\tilde{f} = (1 + \epsilon) \cdot (f_1 op f_2)$$

for some ϵ with $|\epsilon| \le eps$.

Note that if $f_1 op f_2$ is a floating point number then $\tilde{f} = f_1 op f_2$ since the computed result is the floating point number closest to the exact result. In particular, if f_1

and f_2 are both powers of 2 then $f_1 \odot f_2 = f_1 \cdot f_2$ and if $f_1 = f_2$ and f_1 is a power of 2 then $f_1 \oplus f_2 = f_1 + f_2 = 2 \cdot f_1$. The floating point operations $\oplus$ and $\odot$ also satisfy a monotonicity property, namely, if $f_1 \leq g_1$ and $f_2 \leq g_2$ then $f_1 \oplus f_2 \leq g_1 \oplus g_2$ and $f_1 \odot f_2 \leq g_1 \odot g_2$. For the second inequality we also need $f_1 \geq 0$ and $f_2 \geq 0$.

We next turn to expression evaluation. Let E be an arithmetic expression, e.g., $E = a \cdot b - c \cdot d$. When we evaluate E using floating point arithmetic we obtain $\tilde{E} = a \odot b \ominus c \odot d$. What is the relationship between the exact value E and the computed value $\tilde{E}$? We have

$$\begin{aligned} a \odot b &= (1+\epsilon_1) \cdot a \cdot b \\ c \odot d &= (1+\epsilon_2) \cdot c \cdot d \\ a \odot b \ominus c \odot d &= (1+\epsilon_3)(a \odot b - c \odot d) \end{aligned}$$

for some constants ϵ_1, ϵ_2, ϵ_3 with $|\epsilon_i| \leq eps$ and hence

$$\begin{aligned} \tilde{E} &= (a \cdot b \cdot (1+\epsilon_1) - c \cdot d \cdot (1+\epsilon_2))(1+\epsilon_3) \\ &= E + E \cdot \epsilon_3 + (a \cdot b \cdot \epsilon_1 - c \cdot d \cdot \epsilon_2)(1+\epsilon_3). \end{aligned}$$

The error

$$\tilde{E} - E = E \cdot \epsilon_3 + a \cdot b \cdot \epsilon_1 - c \cdot d \cdot \epsilon_2 + (a \cdot b \cdot \epsilon_1 - c \cdot d \cdot \epsilon_2) \cdot \epsilon_3$$

is therefore essentially the sum of the errors introduced by considering the three operations individually. In addition, there is a 'high-order' term.

One can drop the higher-order term by expressing the error in terms of upper bounds on the inputs of an expression rather than the actual input values themselves. This leads to weaker but more managable error bounds.

26. A Floating Point Filter.

Throughout this section E denotes an expression involving real operands and the arithmetic operations addition, subtraction, and multiplication. We use E to also denote the value of the expression. We assume that the operands and more generally the values of all subexpressions to be representable and use $\tilde{(E)}$ to denote the value of the expression when evaluated with floating point arithmetic, i.e., first all operands are replaced by their floating point approximations and then all operations by their floating point counterparts. Our goal is to derive an easily computed bound for the difference between the computed value $\tilde{E}$ and the exact value E.

To this end we define for every expression E its *measure* $mes(E)$ and its *index* $ind(E)$. The measure $mes(E)$ is a power of two which bounds the size of E and $\tilde{E}$ from above, i.e.,

$$size(E) \leq mes(E) \text{ and } size(\tilde{E}) \leq mes(E) \tag{3}$$

The index $ind(E)$ bounds the difference of the computed result $\tilde{E}$ and the exact value E as a multiple of $eps \cdot mes(E)$, i.e.,

$$|\tilde{E} - E| \leq ind(E) \cdot eps \cdot mes(E). \tag{4}$$

The crucial observation is now that both quantities are easily computed inductively. Here are the rules:

If $E = a$, where a is a representable real number, then

$$mes(E) = size(a) \tag{5}$$

and

$$ind(E) = \begin{cases} 0 & \text{if } a \in \mathcal{F} \\ 1/2 & \text{otherwise} \end{cases} \tag{6}$$

if $E = E_1 \pm E_2$ then

$$mes(E) \quad = \quad 2 \cdot \max(mes(E_1), mes(E_2)) \tag{7}$$

and

$$ind(E) \quad = \quad (1 + ind(E_1) + ind(E_2))/2, \tag{8}$$

and if $E = E_1 \cdot E_2$ then

$$mes(E) \quad = \quad mes(E_1) \cdot mes(E_2) \tag{9}$$

and

$$ind(E) \quad = \quad 1/2 + ind(E_1) + ind(E_2). \tag{10}$$

Lemma 1 *Let E be an expression with real operands and assume that the values of all subexpressions of E are representable. Then rules (5) to (10) compute quantities $mes(E)$ and $ind(E)$ satisfying (3) and (4).*

Proof: : If $E = a$, where a is a real, then $\tilde{E} = fl(a)$. Clearly, $|a| \leq size(a)$ and $|fl(a)| \leq size(a)$ (note however that $|a| \leq size(fl(a))$ does not always hold). This establishes (3). Inequality (1) implies (4).

If $E = E_1 op E_2$ then let $\tilde{E}$, $\tilde{E}_1$, and $\tilde{E}_2$ be the computed values for E, E_1, and E_2. Then $|E_i| \leq mes(E_i)$, $|\tilde{E}_i| \leq size(\tilde{E}_i) \leq mes(E_i)$, and $|\tilde{E}_i - E_i| \leq ind(E_i) \cdot eps \cdot mes(E_i)$. Use $err(E_i)$ to abreviate $\tilde{E}_i - E_i$.

Assume first that $E = E_1 + E_2$; the argument for subtraction is analogous. Clearly, $size(E) = size(E_1 + E_2) \leq 2 \cdot \max(size(E_1), size(E_2)) \leq 2 \cdot (mes(E_1), mes(E_2)) = mes(E)$ and $size(\tilde{E}) = size(\tilde{E}_1 \oplus \tilde{E}_2) \leq size(\max(size(\tilde{E}_1), size(\tilde{E}_2)) \oplus \max(size(\tilde{E}_1), size(\tilde{E}_2))) \leq size(2 \cdot \max(mes(E_1), mes(E_2))) \leq size(mes(E)) = mes(E)$ where the first inequality follows from the monotonicity of $\oplus$ and the second inequality follows from the

fact that equal powers of two are added exactly. Similar reasoning shows that $size(\tilde{E}_1 + \tilde{E}_2) \leq mes(E)$. We turn to the error bound next. We have

$$\begin{aligned} |\tilde{E} - E| &= |\tilde{E}_1 \oplus \tilde{E}_2 - (\tilde{E}_1 + \tilde{E}_2) + \tilde{E}_1 + \tilde{E}_2 - E_1 - E_2| \\ &\leq 1/2 \cdot eps \cdot size(\tilde{E}_1 + \tilde{E}_2) + err(E_1) + err(E_2) \\ &\leq (1/2 \cdot mes(E) + ind(E_1) \cdot mes(E_1) + ind(E_2) \cdot mes(E_2)) \cdot eps \\ &= (1 + ind(E_1) + ind(E_2))/2 \cdot mes(E) \cdot eps, \end{aligned}$$

where the first inequality follows from (2).

Assume next that $E = E_1 \cdot E_2$. We leave it to the reader to show that $size(E) \leq mes(E)$, $size(\tilde{E}) \leq mes(E)$, and $size(\tilde{E}_1 \cdot \tilde{E}_2) \leq mes(E)$. For the error bound we obtain

$$\begin{aligned} |\tilde{E} - E| &= |\tilde{E}_1 \odot \tilde{E}_2 - \tilde{E}_1 \cdot \tilde{E}_2 + (E_1 + err(E_1)) \cdot (E_2 + err(E_2)) - E_1 \cdot E_2| \\ &\leq 1/2 \cdot eps \cdot size(\tilde{E}_1 \cdot \tilde{E}_2) + err(E_1) \cdot |E_2| + |\tilde{E}_1| \cdot err(E_2) \\ &\leq (1/2 \cdot mes(E) + ind(E_1) \cdot mes(E_1) \cdot size(E_2) + size(E_1) \cdot ind(E_2) \cdot mes(E_2) \cdot eps \\ &= (1/2 + ind(E_1) + ind(E_2)) \cdot eps \cdot mes(E) \end{aligned}$$

This completes the proof of the lemma. ∎

How are we going to use the error estimates? Assume that E is an expression involving integral operands and operators $+, -$, and $*$. Assume further that we want to know the sign of E. We evaluate E using floating point arithmetic, i.e., we convert the operands into floating point numbers and then use the floating point operations $\oplus$, $\ominus$, and $\odot$, and we also compute the quantities $mes(E)$ and $ind(E)$ using rules (5) to (10). Let $\tilde{E}$ be the computed floating point approximation for E. Note that $\tilde{E}$ is an integer. We have:
If $|\tilde{E}| > ind(E) \cdot mes(E) \cdot eps$ then $sign(\tilde{E}) = sign(E)$, i.e., the sign of $\tilde{E}$ is reliable. This follows immediately from $|\tilde{E} - E| \leq ind(E) \cdot mes(E) \cdot eps$.
If $|\tilde{E}| \leq ind(E) \cdot mes(E) \cdot eps < 1$ then $\tilde{E} = E = 0$. This follows from $|\tilde{E} - E| \leq ind(E) \cdot mes(E) \cdot eps$ and the fact that E and $\tilde{E}$ are integers.
If $|\tilde{E}| \leq ind(E) \cdot mes(E) \cdot eps$ and $ind(E) \cdot mes(E) \cdot eps \geq 1$ then the signs of E and $\tilde{E}$ may be different.

27. Implementation.

The filter is realized by a data type (class) **Float** in C++. **Float**, can be used in the same way as the built-in numerical type **double**. The only exception is that the result of some tests may be unreliable. In particular there is a function *Sign*(**Float** x) that tries to compute the sign of x. Possible results are -1, O, $+1$ and *NO_IDEA*. The last value indicates that the sign of x could not be computed.

Each instance of type **Float** has three data members: *num*, *mes*, and *ind*. *num* is the floating-point approximation of the integer x to be represented, $mes = mes(x)$ and $ind = ind(x)$ are the bounds defined in the previous section. We define initialization and operations $+$, $-$, $*$ following the rules given in Lemma 1.

```
⟨ Float.h 27 ⟩ ≡
#include <LEDA/basic.h>
#include <LEDA/Int.h>
#include <math.h>
  const double eps0 = ldexp(1, −53);      // machine ε = 2^−53
  const int NO_IDEA = 2;
  class Float {
    double num; double mes; float ind;
    Float(double d, double m, float i) { num = d; mes = m; ind = i; }
  public: Float( ) { num = 0; mes = 0; ind = 0; }
    Float(Int i)
    { /* rules (5) and (6). Note that ldexp(1, x) is 2^x and that Ilog(i) is ⌊log |i|⌋
       for i ≠ 0 and −1 for i = 0. Thus Ilog(i − 1) + 1= ⌈log i⌉ for i > 0. */
      if (i ≡ 0) { num = 0; mes = 0; ind = 0; }
      else {
        num = i.todouble( );
        mes = (i > 0) ? ldexp(1, log(i) + 1) : ldexp(1, log(−i) + 1);
        ind = (mes ≤ 53) ? 0 : 0.5;
      }
    }
    operator double ( ) const { return num; }
    friend Float operator+(const Float &a, const Float &b)
    {     // rules (7) and (8)
      return Float(a.num + b.num, 2 ∗ ((a.mes > b.mes) ? a.mes : b.mes),
          (a.ind + b.ind + 1)/2);
    }
    friend Float operator−(const Float &a, const Float &b)
    {     // rules (7) and (8)
      return Float(a.num − b.num, 2 ∗ ((a.mes > b.mes) ? a.mes : b.mes),
          (a.ind + b.ind + 1)/2);
    }
    friend Float operator ∗ (const Float &a, const Float &b)
    {     // tules (9) and (10)
      return Float(a.num ∗ b.num, a.mes ∗ b.mes, (a.ind + b.ind + 0.5));
    }
    friend int Sign(const Float &f)
    {
      double eps = f.ind ∗ f.mes ∗ eps0;
      if (f.num > eps) return +1;
      if (f.num < −eps) return −1;
      if (eps < 1) return 0;
      return NO_IDEA;
    }
  };
```

28. Note that the quantities *mes* and *ind* are computed using double- and single-precision floating point arithmetic respectively. We might therefore incur rounding error in the computation of *ind* and the computation of *mes* might overflow. Note however, that *ind* is always a sum of powers of two and that the exponent of the smallest power is related to the depth of nesting of the expression defining the number. Hence we have rounding error in the computation of *ind* only for expression of depth more than 20. Such expressions do not occur in the sweep program. The quantitity *mes* is always a power of two and hence its computation is exact. It might overflow however, if *mes* exceeds 2^{1024}. This will make the floating point ineffective but not incorrect since any overflown value is set to ∞ according to the IEEE-floating point standard. Namely, if $mes = \infty$ then *eps* in *Sign* evaluates to ∞ and hence *Sign* returns *NO_IDEA*.

29. Experiments and Efficiency.

We performed tests on three kinds of test data: difficult inputs, random inputs and hand-crafted examples.

- Difficult inputs: Let $size$ be a random k-bit integer and let $y = 2size/n$. We intersect n segments where the i-th segment has endpoints $(size + rx1,\ 2 \cdot size - i \cdot y + ry1)$ and $(3size + rx2,\ 2size + i \cdot y + ry2)$ where $rx1$, $rx2$, $ry1$, $ry2$ are random integers in $[-s, s]$ for some small integer s.
- Random inputs: We generated n (between 1 and several hundred) segments with random coordinates between $-size$ and $size$ for some parameter $size$. For $size$ we used large values to test the correctness and efficiency of the floating point filter and the long integer arithmetic and small values to test our claim that we can handle all degeneracies (a set of 100 segments whose endpoints have integer coordinates between -3 and +3 is highly degenerate).
- Hand-crafted examples: We handcrafted some examples that exhibited a lot of degeneracies. We also asked students to break the code and offered DM 100.- for the first counter-example. We have not paid yet.

Table 1 gives the result for the difficult inputs with $s = 3$, $k = 10, 15, 20, \cdots, 100$, and $n = 100$. It lists the number of intersection and the running times with and without the floating point filter. Furthermore it gives the percentage of the comparisons of points in the X-structure that were left undecided by the floating point filter (failure rate) and the percentage of tests where the floating point computation would have decided incorrectly (error rate).

Comparing the x-coordinates of two intersection points is tantamount to testing the sign of a fifth degree homogeneous polynomial in the coordinates of the segment endpoints. These coordinates are of the form $size + rx$, $3size + rx$, $2size + i \cdot y + ry$, $2size - i \cdot y + ry$. Thus, if we write the polynomial in terms of the perturbations rx and ry we essentially obtain a term of order $size^5$ independent of the perturbations (this term is actually zero but has maximal size in intermediate results) plus a term or order $size^4$ times a linear function in the perturbations plus a term of order $size^3$ times a quadratic function in the perturbations $\cdots$. Since the input coordinates are not equal to $size$ but vary between $size$ and $3size$ the various terms are actually spread out over some orders of magnitude.

In the floating point filter analysis the error bound δ is about $3size^5 \cdot 2^{-53}$. We conclude that the floating point filter becomes worthless for k about 50 and that we are starting to loose the linear terms for k about 40. For smaller k the filter fails only in those rare cases where the second order term must decide. This is well reflected in Table 1.

Table 2 gives the results for 100 segments whose endpoints have random k bit coordinates. The experiments inidicate that the floating point filter reduces the running time of the program by a factor of up to 4. Why does the failure rate jump so

k	V	without filter	with filter	failures	errors
10	1323	0.78	0.37	0.05%	0.00%
15	1323	0.68	0.40	0.04%	0.00%
20	1314	0.70	0.38	0.12%	0.05%
25	1325	0.77	0.42	0.08%	0.05%
30	1318	0.75	0.45	1.48%	0.73%
35	1321	1.03	0.52	0.94%	0.39%
40	1312	1.27	0.55	2.17%	1.02%
45	1325	1.27	0.77	20.98%	2.19%
50	1323	1.45	1.23	73.33%	19.98%
55	1325	1.90	1.68	86.35%	44.74%
60	1325	1.33	1.52	83.52%	58.07%
65	1321	2.20	1.90	89.66%	65.21%
70	1323	2.07	1.87	86.88%	63.41%
75	1316	2.57	2.30	89.93%	67.89%
80	1323	2.62	2.08	89.90%	67.33%
85	1321	2.63	2.98	83.11%	62.74%
90	1313	2.78	2.48	90.96%	67.88%
95	1319	3.03	3.07	81.74%	62.33%
100	1323	3.12	3.13	89.19%	67.05%

Table 1: The difficult example with 100 segments.

dramatically when k reaches 210 and jumps once more when $k = 350$. Recall that the comparison of the x-coordinate of two intersection points amounts to computing the sign of a $5k$-bit number and that the comparison of an intersection with an endpoint amounts to computing the sign of a $3k$-bit number. The qunatitiy *mes* in the floating point filter will be essentially 2^{5k} in the first case and 2^{3k} in the second case. Thus the computation of *mes* overflows when $k \geq 210$ in the first case and when $k \geq 350$ in the second case and the comparison is resolved by exact arithmetic.

We also want to comment on the column labeled error, i.e., on the percentage of tests where the floating point filter would have decided incorrectly. In our second example the error rate seems to converge to 50%, i.e., the floating point computation decides randomly. In the difficult example the error rate seems to converge to 2/3. We are not able to explain this phenomenon.

More experiments with different floating point filters are described in [8].

k	V	without filter	with filter	failures	errors
10	1298	0.57	0.35	0.00%	0.00%
30	1373	0.87	0.53	0.00%	0.00%
50	1526	1.78	0.78	0.00%	0.00%
70	1298	2.17	0.70	0.00%	0.00%
90	1417	2.05	0.83	0.00%	0.00%
110	1144	2.08	0.77	0.00%	0.00%
130	1222	2.82	1.25	0.00%	0.00%
150	1434	4.72	1.37	0.00%	0.00%
170	1189	3.65	1.25	0.00%	0.00%
190	1463	6.20	1.63	0.00%	0.00%
210	1401	7.63	8.12	54.66%	34.06%
230	1270	10.58	8.48	51.53%	29.26%
250	1608	14.05	9.72	56.75%	33.74%
270	1226	9.03	6.68	46.28%	29.02%
290	1576	14.55	12.27	49.03%	28.17%
310	1260	15.12	11.10	46.25%	28.21%
330	1280	12.93	11.35	48.15%	29.52%
350	1250	13.25	16.80	87.45%	42.02%
370	1343	24.97	21.37	86.42%	46.91%
390	1420	20.55	20.50	87.96%	46.30%
400	1391	22.38	23.30	89.26%	47.10%

Table 2: 100 random segments, coordinates are random k-bit integers.

30. Conclusion.

We have given an implementation of the Bentley-Ottmann plane sweep algorithm for line segment intersection. The implementation is complete and reliable in the sense that it will work for all input instances. It is asympotically more efficient than previous algorithms for the same task; its running time depends on the number of vertices in the intersection graph and not on the number of pairs of intersecting segments. It also achieves a low constant factor in its running time by means of a floating point filter.

The use of LEDA makes the implementation of the combinatorial part of the algorithm quite elegant. In the geometric part we have not achieved the same level of elegance yet. We are currently exploring whether the floating point filter can be completely hidden from the user as suggested in [11].

References

[1] J.L. Bentley and T.A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28:643–647, 1979.

[2] C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. of the 5th ACM-SIAM Symp. on Discrete Algorithms*, pp. 16–23, 1994.

[3] T.H. Cormen and C.E. Leiserson and R.L. Rivest. Introduction to Algorithms. MIT Press/McGraw-Hill Book Company, 1990.

[4] S. Fortune and C. van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. of the 9th ACM Symp. on Computational Geometry*, pp. 163–172, 1993.

[5] K. Mehlhorn. Data Structures and Efficient Algorithms. Springer Publishing Company, 1984.

[6] E. Myers. An $O(E \log E + I)$ expected time algorithm for the planar segment intersection problem. *SIAM J. Comput.*, pp. 625 –636, 1985.

[7] K. Mehlhorn and S. Näher. LEDA: A library of efficient data types and algorithms. In *LNCS*, 379:88–106, 1989.

[8] K. Mehlhorn and S. Näher. The Implementation of Geometric Algorithms. 13th World Computer Congress IFIP 94, Elsevier Science B.V., Vol. 1, pp. 223–231, 1994.

[9] S. Näher. LEDA Manual. Technical Report No. MPI–I–93–109. Max-Planck-Institut für Informatik, 1993.

[10] F. Preparata and M.I. Shamos. Computational Geometry: An Introduction. Springer Publishing Company, 1985

[11] Ch. Yap and Th. Dubé. The exact computation paradigm. In *Computing in Euclidian Geometry*, World Scientific Press, 1994. (To appear, 2nd edition).

Index

List of Refinements

⟨ `Float.h` 27 ⟩
⟨ clean-up 11 ⟩ Cited in section 7. Used in section 6.
⟨ construct edges and delete ending segments 15 ⟩ Used in section 13.
⟨ find subsequence of ending or passing segments 14 ⟩ Used in section 13.
⟨ geometric primitives 18, 19, 20, 21, 23 ⟩ Used in section 6.
⟨ global types and declarations 7, 8 ⟩ Used in section 6.
⟨ handle passing and ending segments 13 ⟩ Used in section 12.
⟨ include statements 4 ⟩ Used in section 6.
⟨ initialization 10 ⟩ Used in section 6.
⟨ insert starting segments and compute new intersections 17 ⟩ Used in section 12.
⟨ local declarations 9 ⟩ Used in section 6.
⟨ reverse subsequence of segments passing through p 16 ⟩ Used in section 13.
⟨ sweep 12 ⟩ Used in section 6.