Computing the width of a three-dimensional point set: documentation*

Jörg Schwerdt[†] Michiel Smid[†]

February 5, 1999

Contents

*This work was funded in part by a research grant by DAAD.

[†]Department of Computer Science, University of Magdeburg, D-39106 Magdeburg, Germany. E-mail: `{schwerdt,michiel}@isg.cs.uni-magdeburg.de`.

1 Introduction

StereoLithography is a relatively new technology which is used in industry to produce prototypes of CAD models. The input of the StereoLithography process is a triangulated polyhedral CAD model which is sliced by horizontal planes into layers. The StereoLithography apparatus builds the model layer by layer in the positive z-direction.

The time to build one layer is in the range from a few seconds up to dozens of minutes depending on the layer complexity. To keep the building time of the prototype small it is advisable to rotate the CAD model in such a way that its height (w.r.t the z-direction) is as small as possible. Computing this orientation of the CAD model turns out to be equivalent to computing the width of a 3D polyhedron.

In [10, 11], we discuss an implementation of an algorithm of Houle and Toussaint [6] that computes all directions that minimize the width of a polyhedron. The algorithm uses efficient solutions to some well known problems in computational geometry, such as convex hulls in 3D, computing the intersections of line segments, and point location in planar graphs. In fact, the latter two problems have to be solved for segments and graphs that are on the unit sphere rather than in the 2D plane. For more background information and a high-level description of our implementation, the reader is referred to [10, 11].

In this paper, we give a complete documentation of the implementation. The program is written in C++ and uses LEDA [8] (see also `http://www.mpi-sb.mpg.de/LEDA`). In particular, we use efficient data structures from the LEDA library, such as sorted sequences (`sortseq`), which are implemented by skip lists; priority queues (`p_queue`), which are implemented by Fibonacci heaps; and planar maps (`PLANAR_MAP`). Also, we use LEDA's exact rational arithmetic, and floating point filters to solve geometric predicates exactly. Hence, our implementation solves the problem exactly, and is robust.

2 The width problem

We denote the unit sphere in $\mathbb{R}^3$ by $\mathbb{S}^2$, and call the intersection of $\mathbb{S}^2$ and the plane $z = 0$ *equator*. We define the *upper hemisphere* as

$$\mathbb{S}^2_+ := \mathbb{S}^2 \cap \{(x, y, z) \in \mathbb{R}^3 : z \geq 0\}.$$

Similarly, we define the *lower hemisphere* as

$$\mathbb{S}^2_- := \mathbb{S}^2 \cap \{(x, y, z) \in \mathbb{R}^3 : z \leq 0\}.$$

Throughout this paper, we consider a direction $\mathbf{d}$ in $\mathbb{R}^3$ as a point on the unit sphere resp. as a unit vector.

Let $\mathcal{P}$ be an arbitrary polyhedron and let $\mathbf{d}$ be an arbitrary direction in $\mathbb{R}^3$. Let p_1 and p_2 be two parallel planes that are orthogonal to $\mathbf{d}$ and which enclose $\mathcal{P}$. We call the smallest distance between any two such planes p_1 and p_2 the *width of* $\mathcal{P}$ *in direction*

$\mathbf{d}$ and we denote it by $w(\mathbf{d})$. We define the width $W(\mathcal{P})$ of $\mathcal{P}$ as the smallest distance between any two parallel planes enclosing $\mathcal{P}$, i.e.,

$$W(\mathcal{P}) = \min\{w(\mathbf{d}) : \mathbf{d} \in \mathbb{S}^2\}.$$

The width of a three-dimensional point set can be defined in a similar way.

It is clear that the width of $\mathcal{P}$ is equal to the width of $\mathcal{P}$'s vertex set. Houle and Toussaint [6] gave an algorithm which computes the width of a set of n points in $\mathbb{R}^3$ in $O(n^2)$ worst case time. The asymptotically fastest algorithm for computing the width of a three dimensional point set is due to Agarwal and Sharir [1]. Its expected running time is roughly $O(n^{1.5})$.

There are two reasons why we implemented the asymptotically slower algorithm of Houle and Toussaint. First, it computes *all* optimal directions whereas the Agarwal and Sharir algorithm computes only one. Knowing all optimal directions is useful, if we want to optimize another measure among these directions [7]. Second, it is much easier to implement and, as our experiments in Section 11 show, its running time in practice is much lower than the quadratic worst-case running time.

3 The algorithm

It is obvious that the width of the polyhedron $\mathcal{P}$ is equal to the width of the convex hull of $\mathcal{P}$. Therefore, we first compute the convex hull $CH(\mathcal{P})$ using the LEDA implementation `D3_HULL` that computes the convex hull of a point set in $\mathbb{R}^3$. The output of this algorithm is a graph G_{CH} that is the implicit representation of the convex hull.

Let V be a vertex and F a facet of $CH(\mathcal{P})$. We call (V, F) an *antipodal vertex-facet pair* (or VF-pair), if the two parallel planes containing V and F, respectively, enclose $CH(\mathcal{P})$. Note that these two planes are unique. We say that these parallel planes *support* $CH(\mathcal{P})$.

Similarly, two *non-parallel* edges e_0 and e_1 of $CH(\mathcal{P})$ are called an *antipodal edge-edge pair* (or EE-pair), if the two (unique) parallel planes containing e_0 and e_1, respectively, enclose $CH(\mathcal{P})$. Again, we say that these parallel planes *support* $CH(\mathcal{P})$.

Houle and Toussaint point out that the two parallel planes with minimum width either touch $\mathcal{P}$ at a vertex and a facet or at two edges that are not parallel. All other possibilities can be reduced to these two cases, or the planes do not have minimum width. For example, if one plane touches a facet of $\mathcal{P}$ and the other plane touches an edge, it is equivalent to a VF-pair. Or, if the two planes touch two parallel edges, it is possible to rotate the polyhedron and reduce the distance of the planes.

Hence, we can solve the width problem by computing all VF- and EE-pairs of $CH(\mathcal{P})$. Now, we give a short description how the VF- and EE-pairs can be computed. A detailed description can be found in Sections 8 and 9.

Given the graph G_{CH}, we compute its dual representation G. G is a planar graph on the unit sphere $\mathbb{S}^2$ and it is defined as follows. The vertices of G are the facet outer unit normals of G_{CH} and two vertices are connected by an edge in G, if the corresponding facets of G_{CH} share an edge. Note that edges of G are great arcs on $\mathbb{S}^2$. Moreover, edges (resp. facets) of G are in one-to-one correspondence with edges (resp. vertices) of G_{CH}. Since the endpoints p and q of an edge $e \in G$ are outer normals of two neighboring facets of G_{CH}, it is obvious that p and q are not antipodal and that e is always the shorter one of the two great arcs with endpoints p and q.

We cut the graph G along the equator of $\mathbb{S}^2$ in two pieces. We call the part of G that is on the upper (resp. lower) hemisphere G_+ (resp. G'_-). If a vertex (or an entire edge) is on the equator it is contained in both graphs, G_+ and G'_-. An edge crossing the equator is cut, and the upper (resp. lower) part of the edge is contained in G_+ (resp. G'_-).

Now we compute the reflected image G_- of G'_-. G_- is the graph we get by mapping each vertex v of G'_- to $-v$. Thereafter, both graphs G_+ and G_- are on the upper hemisphere $\mathbb{S}^2_+$.

It can be shown that *EE*-pairs are in one-to-one correspondence with intersecting edges of G_+ and G_-. (See [6].) Therefore, we find all *EE*-pairs by computing the intersecting edges of G_+ and G_-. We only compute intersection points that are in the interior of both edges. (We treat degenerate intersection points as *VF*-pairs and find them while computing the *VF*-pairs.) To do this, we adapt the implementation of Bartuschka et al.[2] that is an efficient and robust implementation of the Bentley-Ottmann sweep line algorithm [3]. This implementation makes no assumption about the input, in particular, the segments need not be in general position.

The *VF*-pairs are in one-to-one correspondence with pairs (v, f), where v is a vertex of G_+ (resp. G_-), f is a face of G_- (resp. G_+), and $v \in f$. (See [6].) We compute all *VF*-pairs with an adapted point location algorithm called *slab method* that is due to Dobkin and Lipton [4]. It locates all vertices of G_+ in the graph G_- and vice versa. Since we know all query points in advance, it is not necessary to store the complete point location data structure. We sort the vertices of G_+ and G_- in the order the sweep circle (we sweep on $\mathbb{S}^2_+$) visits them and compute the slabs as in the original algorithm. While sweeping we locate each vertex that is contained in the current slab.

Both these algorithms are based on the *plane sweep paradigm* and they both use the same type of geometric primitives.

The worst-case running time of the algorithm is bounded by

$$O(n^2 + k \log h) = O(n^2 \log n),$$

where n is the number of facets of the polyhedron $\mathcal{P}$, h is the number of facets of $CH(\mathcal{P})$, and k is the number of intersections between edges of G_+ and edges of G_-. (According to the LEDA user manual, the convex hull implementation `D3_HULL` takes $O(n^2)$ time in the worst case, but $O(n \log n)$ time for most inputs.)

3.1 Sweeping on the upper hemisphere

In a plane sweep algorithm for two-dimensional objects, we solve the problem at hand by sweeping a vertical line from left to right over the scene. Sweeping on the upper hemisphere can be thought of as follows. Let the z-axis be vertical. Moreover, let the

x-axis be in the horizontal plane $z = 0$, going from left to right. Finally, let the y-axis be in the plane $z = 0$, going from bottom to top. When sweeping on the upper hemisphere, we move a half-circle from the left part of the equator, along the upper hemisphere to the right part of the equator, while keeping the two endpoints of the half-circle on the y-axis. We can also regard this as rotating a half-plane around the y-axis. We call this half-plane the *sweep half-plane.*

By central projection, it follows that sweeping over $\mathbb{S}^2_+$ is in one-to-one correspondence with sweeping on the plane. It would be possible to project all segments that are on $\mathbb{S}^2_+$ to the plane $z = 1$ and use the standard sweep line algorithms to solve the problems. But this method has a disadvantage: Vertices and edges that are on the equator are projected to infinity. As a result, we would have to be able to compare different points at infinity.

4 Representation of vertices

Our algorithm represents each vertex of the dual graph G_+ resp. G_- as a point on $\mathbb{S}^2$. The vertices are normal vectors of facets of G_{CH}. To compute a vertex as a point on the sphere, we first have to compute the normal vector and then we have to normalize it. This means, we divide each coordinate by the square root $\sqrt{x^2 + y^2 + z^2}$. Since we want to compute exactly and avoid expensive computations, we decided not to normalize the vectors. Thus our vertices are not really points on the unit sphere. The ray from the origin through our vertex intersects the unit sphere in exactly that point where the vertex on the unit sphere would be.

Let a, b and c be three points on the unit sphere and let a' (resp. b', c') be an arbitrary point on the ray from the origin through a (resp. b, c). The basic tests in all our algorithms are orientation tests of three points on $\mathbb{S}^2_+$. We make the observation that there is no difference between the following three tests.

- Test, if c is to the left of (resp. to the right of or on) a great arc from a to b.
- Test, if c is to the left of (resp. to the right of or on) a directed plane through the great arc from a to b.
- Test, if c' is to the left of (resp. to the right of or on) a directed plane through the great arc from a' to b'.

Since the unique primitive in all tests is this orientation test, there is no difference if we use the normal vectors or the normalized normal vectors as endpoints of the segments.

During the paper we will sometimes say “a point on the unit sphere” resp. “a segment on the unit sphere”, because some things are easier to explain. E.g. we can say “a great arc” or things like that. But bear in mind that in the implementation, these points resp. segments are not really on the unit sphere; rather, they *represent* points and segments on the unit sphere.

5 The class sphere_point – a point on the unit sphere

A sphere_point is a non-zero point in $\mathbb{R}^3$ that represents a point on the unit sphere $\mathbb{S}^2$. It is represented by its homogeneous coordinates px, py, pz and pw. Each point has a unique number p_number.

Note, that some of the member functions only work correctly if all sphere_points are on or above the plane $z = 0$. But this is no problem, because these member functions are only used in our segment intersection and point location algorithms and these algorithms only sweep over the upper hemisphere. This allows us to keep the member functions as simple as possible.

5.1 The file sphere_point.h

⟨*sphere_point.h*⟩≡

```
# ifndef _SPHERE_POINT_H
# define _SPHERE_POINT_H

#include <LEDA/integer.h>
#include <LEDA/rational.h>
#include <LEDA/d3_rat_point.h>
#include <ctype.h>
#include <iostream.h>

class sphere_point
{
  ⟨protected part⟩
  ⟨public part⟩
};

extern int orientation( const sphere_point&,
                        const sphere_point&,
                        const sphere_point& );

extern int compare( const sphere_point& a,
                    const sphere_point& b);

# endif /* !_SPHERE_POINT_H */
```

5.1.1 The protected part

A sphere point is a 3D point with homogeneous coordinates and a unique number. The coordinate pw is always positive.

⟨*protected part*⟩≡
```
protected:
integer px;
integer py;
integer pz;
integer pw;
int p_number;
```

5.1.2 The public part

⟨*public part*⟩≡
```
public:
  ⟨constructors⟩
  ⟨destructor, copy constructor and assignment operator⟩
  ⟨functions and operators⟩
```

The constructors

⟨*constructors*⟩≡
```
sphere_point() : px(0), py(0), pz(0), pw(1) {p_number = -1;}

sphere_point(d3_rat_point p, int n = -1)
{
  px = p.X();
  py = p.Y();
  pz = p.Z();
  pw = p.W();
  p_number = n;
}

sphere_point(rat_vector v, int n = -1)
{
  px = v.X();
  py = v.Y();
  pz = v.Z();
  pw = v.W();
  p_number = n;
}

sphere_point(integer x, integer y, integer z, integer w, int n = -1)
{
  if(w>0)
    {
      px = x;
      py = y;
      pz = z;
      pw = w;
    }
  else
    {
      px = -x;
      py = -y;
      pz = -z;
      pw = -w;
    }
  p_number = n;
}
```

Destructor, copy constructor and assignment operator

⟨*destructor, copy constructor and assignment operator*⟩≡

```
~sphere_point() {}

sphere_point& operator=(const sphere_point& p)
{
  if (this == &p) return *this;

  px = p.px;
  py = p.py;
  pz = p.pz;
  pw = p.pw;
  p_number = p.p_number;

  return *this;
}

sphere_point(const sphere_point& p)
{
  px = p.px;
  py = p.py;
  pz = p.pz;
  pw = p.pw;
  p_number = p.p_number;
}
```

Functions and operators The call `p.cmp(a)` compares the `sphere_point p` with the parameter `sphere_point a`. For details see Section 5.2.2. All other functions are self explaining.

⟨*functions and operators*⟩≡

```
d3_rat_point rat_point() const {return(d3_rat_point(X(), Y(), Z(), W()));}
```

```
  integer X() const {return(px);}
  integer Y() const {return(py);}
  integer Z() const {return(pz);}
  integer W() const {return(pw);}

  rational xcoord() const {return(rational(px,pw));}
  rational ycoord() const {return(rational(py,pw));}
  rational zcoord() const {return(rational(pz,pw));}

  int  cmp(const sphere_point& a) const;
  int  number() const {return(p_number);}

  friend bool operator<(const sphere_point& x, const sphere_point& y);
  friend bool operator<=(const sphere_point& x, const sphere_point& y);
  friend bool operator>(const sphere_point& x, const sphere_point& y);
  friend bool operator>=(const sphere_point& x, const sphere_point& y);
  friend bool operator==(const sphere_point& x, const sphere_point& y);

  friend istream& operator» (istream& in, sphere_point& sp);
  friend ostream& operator« (ostream& out, const sphere_point& sp);
```

5.2 The file sphere_point.c

⟨*sphere_point.c*⟩≡
```
  #include <LEDA/d3_rat_plane.h>
  #include <LEDA/integer.h>
  #include "sphere_point.h"

  ⟨orientation⟩
  ⟨compare⟩
  ⟨operators⟩
```

5.2.1 The orientation of three sphere_points

The member function `orientation` computes the orientation of three `sphere_points` (`a`, `b` and `p`) on the upper hemisphere.

Let a', b' and p' be the directions on $\mathbb{S}^2_+$ that are represented by `a`, `b` and `p`, respectively.

Deciding which orientation a', b' and p' have on $\mathbb{S}^2_+$ is equivalent to deciding if point p' is to the left, to the right or on the directed great circle from a' to b'. But this is equivalent to deciding if p' is to the left, to the right or on the directed plane H through a', b' and the origin. Since `a` is on the ray from the origin to a', `a` is on H if and only if a' is on H. Similar observations hold for `b` and `p`. Therefore our orientation test is equivalent to deciding if point `p` is to the left, to the right or on the oriented plane through `a`, `b` and the origin. This latter orientation test is decided by LEDA's `orientation(d3_rat_point u, d3_rat_point v, d3_rat_point x, d3_rat_point y)`.

⟨*orientation*⟩≡
```
int orientation(const sphere_point& a,
                const sphere_point& b,
                const sphere_point& p)
{
  // Precondition: all points are on the upper hemisphere
  return( orientation( a.rat_point(), d3_rat_point(0,0,0,1),
                       b.rat_point(), p.rat_point()) );
}
```

5.2.2 Comparing two `sphere_points`

This member function compares two `sphere_points` `a` (the object itself) and `b`. The compare function decides which point is met first by the sweep half-plane. If it meets both points at the same time, we decide which of the points is greater, as follows. We walk along the sweep-circle from the negative to the positive y-axis. The point we meet first is smaller. If we meet both points simultaneously they are equal. Notice, that this compare function is not limited to the upper hemisphere.

In this function `SP` is a sweep half-plane stopping at the sweep event point `a`. Thus, it is the plane through the point `a` and the y-axis. The outer normal of `SP` is in the sweep direction. The variable `side_of_b` contains the information on which side of `SP` the point `b` is.

⟨*compare*⟩≡
```
int sphere_point::cmp(const sphere_point& b) const
{
  integer aX = X();
  integer aY = Y();
  integer aZ = Z();
  integer aW = W();
  integer bX = b.X();
  integer bY = b.Y();
  integer bZ = b.Z();
  integer bW = b.W();

  ⟨minimum or maximum points⟩

  d3_rat_plane SP( d3_rat_point(0,1,0),
                   d3_rat_point(0,-1,0),
                   d3_rat_point(aX, aY, aZ, aW));
  int side_of_b = SP.side_of(d3_rat_point(bX, bY, bZ, bW));
```

```
    if(aZ > 0)
      {
        ⟨a is above the plane z=0⟩
      }
    else
      {
        if(aZ < 0)
          {
            ⟨a is below the plane z=0⟩
          }
        else  // aZ == 0
          {
            ⟨a is on the equator⟩
          }
      }
  }

  int compare(const sphere_point& a, const sphere_point& b)
  {
    return(a.cmp(b));
  }
```

Handling points on the y-axis Points on the y-axis are on the sweep half-plane from the beginning to the end of the sweep process. We define the point on the positive part as the maximum point, and the other one as the minimum point.

We need some special tests to handle these minimum resp. maximum points in the compare function. First, we test if the point `a` is the minimum or maximum point (x- and z-coordinates are equal to zero). If so, we test if the y-coordinate is less than zero. If this holds `a` is the minimum point. Otherwise it is the maximum point.

Suppose `a` is the minimum point. If `b` is the minimum point as well, both points are equal and we return 0. If not, point `b` is greater than point `a` and we return -1. The case when `a` is the maximum point is analogous.

In the second part we know that `a` is neither the minimum nor the maximum point. We test if `b` is the minimum or the maximum point. If it is the minimum point, it is obvious that `a` is greater than `b` and we return the value 1. If `b` is the maximum point we return -1 because `a` is less than `b`.

⟨*minimum or maximum points*⟩≡
```
  if(aX == 0 && aZ == 0)
  {
    if(aY < 0)      // Is a the minimum point?
      {
        if(bX == 0 && bY < 0 && bZ == 0)
          {
            return(0);
          }
        else
          {
            return(-1);
          }
      }
    else
      {  // aY > 0   a is the maximum point
        if(bX == 0 && bY > 0 && bZ == 0)
          {
            return(0);
          }
        else
          {
            return(1);
          }
      }
  }

  if(bX == 0 && bZ == 0)
  {
    if(bY < 0)     // Is b the minimum point?
      {
        return(1);
      }
    else              // b is the maximum point
      {
        return(-1);
      }
  }
```

a is above the plane z=0 Suppose `side_of_b` is greater than zero. In this case the sweep halfplane `SP` first meets `a` and then `b`. By definition, `a` is less than `b` in this case. We return -1.

Suppose `side_of_b` is equal to zero. If `b` is above the plane $z = 0$, then the sweep halfplane `SP` meets both points at the same time. Now we are sorting the points along `SP` from the negative part of the y-axis to the positive one. How can we decide if a point `a` on `SP` is less than, greater than or equal to `b`? We can do this again by computing the orientation of a plane and the point `b`.

Let l_n be the line through the origin orthogonal to the sweep half-plane `SP`. Suppose we are rotating a halfplane `NSP` around l_n from the negative to the positive part of the y-axis. `NSP` meets the points on the sweep halfplane in increasing order. If `NSP` first meets `a`, then `a` is smaller than `b`. To decide this, we create the plane `NSP` through `a` and the origin orthogonal to `SP` and test on wich side point `b` is. If `b` is on the side that `NSP` has not reached so far it is greater than `a` and we return -1. Otherwise `b` is less than or equal to `a` and we return 1 or 0, respectively.

It is obvious that we can return the value `NSP.side_of(b)`, if we define `NSP` such that the result of the `side_of` test is equal to the value our compare function should obtain. This is so if we define `NSP` in the listed way.

If `side_of_b` is equal to zero and `b` is not above the plane $z = 0$, then `a` is less than `b`, and we return -1.

Suppose `side_of_b` is less than zero. Recall that `a` is on $\mathbb{S}^2_+$. In this case there are two possibilities. The first is when `b` is below the plane $z = 0$. In this case `a` is less than `b` and therefore we return -1. The second is that `b` is on or above the plane $z = 0$. In this case is `a` greater than `b` and therefore we return 1.

⟨*a is above the plane z=0*⟩≡

```
if(side_of_b > 0)
{
  return(-1);
}
else
{
  if(side_of_b == 0)
    {
      if(bZ > 0)
        {
          d3_rat_point sp( aX, aY, aZ, aW );
          d3_rat_plane NSP( sp + SP.normal(),
                            d3_rat_point(0,0,0),
                            sp );

          return(NSP.side_of(b.rat_point()));
        }
      else
        {
          return(-1);
        }
    }
  else  // side_of_b < 0
    {
      if(bZ < 0)
        {
          return(-1);
        }
      else
        {
          return(1);
        }
    }
}
```

a is below the plane z=0 This case is analogous to the previous case.

⟨*a is below the plane z=0*⟩≡
```
if(side_of_b < 0)
{
  return(1);
}
else
{
  if(side_of_b == 0)
    {
      if(bZ < 0)
        {
          d3_rat_point sp(aX, aY, aZ, aW);
          d3_rat_plane NSP( sp + SP.normal(),
                            d3_rat_point(0,0,0),
                            sp );

          return(NSP.side_of(b.rat_point()));
        }
      else
        {
          return(1);
        }
    }
  else  // SP.side_of(a) > 0
    {
      if(bZ < 0)
        {
          return(-1);
        }
      else
        {
          return(1);
        }
    }
}
```

a is on the equator Suppose a is on the equator. There are two possibilities in this case. Either SP meets a at the beginning of the sweep (aX < 0) or after it has sweeped over the entire upper hemisphere (aX > 0). Note that aX $= 0$ is not possible, otherwise a would be the minimum or the maximum point. In this case, we would have returned a value earlier in the compare function.

In the first case (aX < 0) there are again two possibilities. If b is also on SP, then we sort the points along SP from the negative to the positive y-coordinate. To decide the order of a and b, we again use the orthogonal sweep half-plane NSP as mentioned before. If b is not on SP, then a is less than b, and we return -1.

In the second case ($\texttt{aX} > 0$) we obtain three cases. First, if b is below the plane $z = 0$ we know that a is less than b and we return -1. Second, if b is on SP we sweep around the normal of SP. In the third case, we know that b is on or above the plane $z = 0$ but not on SP. In this case we know that a is greater than b and therefore we return 1.

⟨*a is on the equator*⟩≡
```
if(aX < 0)
{
  if(bZ == 0 && bX < 0)
    {
      d3_rat_point sp(aX, aY, aZ, aW);
      d3_rat_plane NSP( sp + SP.normal(),
                        d3_rat_point(0,0,0), sp );
      return(NSP.side_of(b.rat_point()));
    }
  else
    {
      return(-1);
    }
}
else
{
  if(aX > 0)
    {
      if(bZ < 0)
        {
          return(-1);
        }
      else
        {
          if(bZ == 0 && bX > 0)
            {
              d3_rat_point sp(aX, aY, aZ, aW);
              d3_rat_plane NSP( sp + SP.normal(),
                                d3_rat_point(0,0,0), sp );
              return(NSP.side_of(b.rat_point()));
            }
          else
            {
              return(1);
            }
        }
    }
}
```

5.2.3 The operators

⟨*operators*⟩≡
```
bool operator<(const sphere_point& x, const sphere_point& y)
{ return x.cmp(y) < 0; }
bool operator<=(const sphere_point& x, const sphere_point& y)
{ return x.cmp(y) <= 0; }
bool operator>(const sphere_point& x, const sphere_point& y)
{ return x.cmp(y) > 0; }
bool operator>=(const sphere_point& x, const sphere_point& y)
{ return x.cmp(y) >= 0; }
bool operator==(const sphere_point& x, const sphere_point& y)
{ return x.cmp(y) == 0; }

ostream& operator« (ostream& s, const sphere_point& p)
{
  s « p.p_number « " (" « p.X() « "," « p.Y() « ","
    « p.Z() « "," « p.W() « ")";

   return s;
}
istream& operator» (istream& in, sphere_point& p)
{
  // p.X "," p.Y "," p.Z "," p.W
   char c;
   integer x,y,z,w;
   do in.get(c); while (in && isspace(c));
   if (c != '(') in.putback(c);
   in » x;
   do in.get(c); while (isspace(c));
   if (c != ',') in.putback(c);
   in » y;
   do in.get(c); while (isspace(c));
   if (c != ',') in.putback(c);
   in » z;
   do in.get(c); while (isspace(c));
   if (c != ',') in.putback(c);
   in » w;
   do in.get(c); while (c == ' ');
   if (c != ')') in.putback(c);
   p = sphere_point(x,y,z,w);
   return in;
}
```

6 The class sphere_segment – a segment on the unit sphere

An object of the class `sphere_segment` represents a segment, i.e., a great arc, on the unit sphere $\mathbb{S}^2$. It is an edge of the dual representation of the convex polyhedron $CH(\mathcal{P})$. Its endpoints are the normal vectors (`sphere_points`) of two neighboring facets of $CH(\mathcal{P})$. Thus, the two endpoints are not antipodal.

Suppose the endpoints are represented by points a and b on $\mathbb{S}^2$. Let C be the great circle through a and b. Then the `sphere_segment` represents the shortest arc of C containing a and b.

6.1 The file sphere_segment.h

⟨*sphere_segment.h*⟩≡

```
# ifndef _SPHERE_SEGMENT_H
# define _SPHERE_SEGMENT_H

#include <LEDA/d3_rat_point.h>
#include <ctype.h>
#include <iostream.h>
#include "sphere_point.h"

extern int s_seg_number;

class sphere_segment
{
  ⟨protected part⟩
  ⟨public part⟩
};

extern int orientation(const sphere_segment& s, const sphere_point& p);

# endif /* !_SPHERE_SEGMENT_H */
```

6.1.1 The protected part

With each `sphere_segment` we store its two endpoints `normal1` and `normal2`. During our computation we also need the primal edge (of $CH(\mathcal{P})$) of the `sphere_segment`, to be precise we need the endpoints of the primal edge. Therefore we store these endpoints `v1` and `v2` with each `sphere_segment`.

The two remaining variables contain a unique number (`segnumber`) and a boolean value (`intersected_segment`) which is `true` iff the segment is intersected by the equator of $\mathbb{S}^2$ (the plane $z = 0$).

⟨*protected part*⟩≡
```
protected:
  int segnumber;
  sphere_point normal1, normal2;
  d3_rat_point v1, v2;
  bool intersected_segment;
```

6.1.2 The public part

The public part contains the constructors, destructors and some member functions.

⟨*public part*⟩≡
```
public:
  ⟨constructors⟩
  ⟨destructor copy constructor and assignment operator⟩
  ⟨functions and operators⟩
```

The constructors The class `sphere_segment` has three constructors. One constructor taking no argument, one constructor taking two points (`sphere_point`) of the unit sphere as arguments, and one constructor taking two `sphere_points` and the two vertices of the primal representation of this segment as arguments. Using the compare function of Section 5.2.2, we sort the endpoints of the `sphere_segment` such that `normal1` $\leq$ `normal2`. Hence, the `sphere_segments` are directed.

⟨*constructors*⟩≡
```
sphere_segment()
{
  segnumber = s_seg_number++;
  intersected_segment = false;
}

sphere_segment(sphere_point n1, sphere_point n2, bool is = false )
{
  segnumber = s_seg_number++;
  if( n1 < n2 )
    {
      normal1 = n1;
      normal2 = n2;
    }
  else
    {
      normal1 = n2;
      normal2 = n1;
    }
  intersected_segment = is;
}
```

```
sphere_segment(sphere_point n1, sphere_point n2,
               d3_rat_point vert1, d3_rat_point vert2,
               bool is = false )
{
  segnumber = s_seg_number++;
  if( n1 < n2 )
    {
      normal1 = n1;
      normal2 = n2;
    }
  else
    {
      normal1 = n2;
      normal2 = n1;
    }
  intersected_segment = is;
  v1 = vert1;
  v2 = vert2;
}
```

Destructor, copy constructor and assignment operator

⟨*destructor copy constructor and assignment operator*⟩≡

```
~sphere_segment() {}

sphere_segment& operator=(const sphere_segment& p)
{
  if (this == &p) return *this;
  segnumber = p.segnumber;
  normal1 = p.normal1;
  normal2 = p.normal2;
  intersected_segment = p.intersected_segment;
  v1 = p.v1;
  v2 = p.v2;
  return *this;
}

sphere_segment(const sphere_segment& p)
{
  segnumber = p.segnumber;
  normal1 = p.normal1;
  normal2 = p.normal2;
  intersected_segment = p.intersected_segment;
  v1 = p.v1;
  v2 = p.v2;
}
```

Functions and operators First there are functions to get the entries of the `sphere_-segment`. To be precise, there are functions to get the endpoints of the dual segment in different representations (`a`, `b`, `rat_a` and `rat_b`), the endpoints of the primal segment (`vertex1` and `vertex2`), the unique number of the `sphere_segment`, information if the segment is intersected by the equator, information if the segment is trivial, the intersection point of two `sphere_segments` and a test if the `sphere_segment` contains a given point. The function `intersection` has the precondition that an intersection point exists.

⟨*functions and operators*⟩≡
```
sphere_point a() const {return(normal1);}
sphere_point b() const {return(normal2);}

d3_rat_point rat_a() const {return(normal1.rat_point());}
d3_rat_point rat_b() const {return(normal2.rat_point());}

d3_rat_point vertex1() const {return(v1);}
d3_rat_point vertex2() const {return(v2);}

int  number() const {return(segnumber);}
bool intersected() const {return(intersected_segment);}
bool is_trivial() const {return(normal1 == normal2);}
bool intersection(const sphere_segment& ss, sphere_point& p) const;
bool contains(const sphere_point& p) const;

friend bool operator==(const sphere_segment& x, const sphere_segment& y);
friend bool operator!=(const sphere_segment& x, const sphere_segment& y);

friend istream& operator» (istream& in, sphere_segment& sp);
friend ostream& operator« (ostream& out, const sphere_segment& sp);
```

6.2 The file `sphere_segment.c`

⟨*sphere_ segment.c*⟩≡
```
#include <LEDA/rat_point.h>
#include <LEDA/vector.h>
#include <LEDA/integer.h>
#include <LEDA/d3_rat_plane.h>
#include "sphere_segment.h"

int s_seg_number = 0;

⟨orientation⟩
⟨intersection⟩
⟨contains⟩
⟨operators⟩
```

6.2.1 The orientation of a (directed) `sphere_segment` and a `sphere_point`

The function `orientation` computes the orientation of three `sphere_points` on the upper hemisphere. The first and second `sphere_point` are the source and the target point of the `sphere_segment s`. Recall, that `s` is directed. The third point is the given argument `p`.

During the sweep processes we maintain a sorted sequence containing all segments that are in contact with the sweep half-plane. We introduce two sentinels which we call a positive and a negative *infinity segment*. These sentinels are always in the sorted sequence to avoid special cases. Similarly, we introduce *infinity points*. In this orientation test, we first handle these infinity segments and points.

Suppose the segment is not entirely on the equator. Then the orientation of the segment and the point is equal to the orientation of the three points (the source, the target of the segment `s` and the point `p`).

Suppose the segment is on the equator. If the segment `s` contains the point `p` it is obvious that the orientation of `s` and `p` is zero.

Let the *left border* be all points on the equator with negative x-coordinate together with the minimum point, and let the *right border* be all points on the equator with positive x-coordinate together with the maximum point.

Suppose the entire segment is on the left border. If the point `p` is also on the left border the orientation equals zero, because they are both (segment and point) on the sweep half-plane. If the point `p` is not on the left border, the orientation is a rightturn and the result is "-1". The case when the entire segment is on the right border is handled similarly.

Suppose the segment is on the equator and not entirely on the left resp. right border. Then the segment either crosses the positive or the negative part of the y-axis. We test if the minimum resp. the maximum point is on the segment. In Section 6.2.3 we will give some details for this test. If the segment contains the maximum (resp. minimum) point the orientation is a rightturn (resp. leftturn).

⟨*orientation*⟩≡
```
int orientation( const sphere_segment& s, const sphere_point& p)
{
  if(s.a().Z() < 0)   // infinity segment
    {
      if(s.a().X() < 0)   // - infinity segment
        {
          if(p.Z() < 0 && p.X() < 0 ) // - infinity point
            {
              return(0);
            }
          else
            {
              return(1);
            }
        }
    else    // + infinity segment
      {
        if(p.Z() < 0 && p.X() > 0 ) // + infinity point
          {
            return(0);
          }
        else
          {
            return(-1);
          }
      }
  }
else   // regular segment
  {
    if(p.Z() < 0)  // infinity point
      {
        if(p.X() < 0)  // - infinity point
          {
            return(1);
          }
        else   // + infinity point
          {
            return(-1);
          }
      }
    else
      {
        if( s.a().Z() == 0 && s.b().Z() == 0 )
          {                           // segment is on the equator
            if( s.contains(p) )
              {
                return(0);
              }
            if( s.a().X()<=0 && s.b().X()<=0 )
              {                       // segment is on the left border
                if( p.Z()==0 && ( p.X()<0 || ( p.X()==0 && p.Y()<0 ) ) )
                  {
                    return(0);
                  }
                return(-1);
              }
            if( s.a().X()>=0 && s.b().X()>=0 )
              {                       // segment is on the right border
                if( p.Z()==0 && ( p.X()>0 || ( p.X()==0 && p.Y()>0 ) ) )
                  {
                    return(0);
                    }
                  return(1);
                }
              // if s contains the minimum or maximum point
              d3_rat_plane pl( s.a().rat_point(),
                               s.b().rat_point(),
                               s.b().rat_point() + rat_vector(0,0,1,1) );
              int side_of_O = pl.side_of(d3_rat_point(0,0,0,1));
              if(side_of_O == -1) // maximum point
                {
                  return(-1);
                }
              if(side_of_O == 1) // minimum point
                {
                  return(1);
                }
            }
        }
    }
  // regular segment that is not on the equator
  return( orientation(s.a(), s.b(), p) );
}
```

6.2.2 compute the intersection point of two `sphere_segments`

The function call `s1.intersection(s2,p)` computes the intersection point `p` of two `sphere_segments` `s1` and `s2`. It has the precondition that an intersection point exists.

We compute the plane `plane1` (resp. `plane2`) containing `s1` (resp. `s2`) and the origin. Recall that a `sphere_point` is a vector of arbitrary length and so is the intersection point. This intersection point is a vector from the origin with direction parallel to the intersection line of the two planes. We obtain this vector by computing the cross product of the normals of `s1` and `s2`. We have to pay attention that we do not compute the reflected intersection point. Therefore the intersection point is either the cross product or the reflected cross product of the two plane normals.

Since the segments are on the upper hemisphere the intersection point is reflected if its z-coordinate is less than zero. If the intersection point is on the equator we have to test if the `sphere_segments` contains the intersection point. If not, we have to reflect it.

⟨*intersection*⟩≡
```
bool sphere_segment::intersection( const sphere_segment& ss,
                                   sphere_point& p ) const
{
  sphere_point a1 = a();
  sphere_point b1 = b();
  sphere_point a2 = ss.a();
  sphere_point b2 = ss.b();
```

```
int o1 = orientation(*this,a2);
int o2 = orientation(*this,b2);
int o3 = orientation(ss,a1);
int o4 = orientation(ss,b1);

if ( o1 != o2 && o3 != o4 )
  {
    d3_rat_plane plane1 = d3_rat_plane( normal1.rat_point(),
                                        normal2.rat_point(),
                                        d3_rat_point(0,0,0) );
    d3_rat_plane plane2 = d3_rat_plane( ss.a().rat_point(),
                                        ss.b().rat_point(),
                                        d3_rat_point(0,0,0) );

    rat_vector n1 = plane1.normal();
    rat_vector n2 = plane2.normal();

    integer x = (n1.Y()*n2.Z()) - (n2.Y()*n1.Z());
    integer y = (n1.Z()*n2.X()) - (n2.Z()*n1.X());
    integer z = (n1.X()*n2.Y()) - (n2.X()*n1.Y());
    integer w = n1.W()*n2.W();

    if( z < 0 || (z == 0 && !contains(sphere_point(x,y,z,w)) ) )
      {
        x = -x;
        y = -y;
        z = -z;
      }

    p = sphere_point(x,y,z,w);
    return true;
  }
if ( o1 == 0 && contains(a2) && ss.intersected() )
  {
    p = a2;
    return(true);
  }
if ( o2 == 0 && contains(b2) && ss.intersected() )
  {
    p = b2;
    return(true);
  }
if ( o3 == 0 && ss.contains(a1) && intersected() )
  {
    p = a1;
```

```
        return(true);
      }
    if ( o4 == 0 && ss.contains(b1) && intersected() )
      {
        p = b1;
        return(true);
      }
    return(false);
  }
```

6.2.3 Test if the `sphere_segment` contains a `sphere_point`

Since the segments are great arcs, a segment contains a point `h` only if the point lies on the plane `PL` through the segment and the origin. This is necessary but not sufficient. Recall that the segments are on the upper hemisphere.

Assume that `h` is on `PL`. If the segment $(\mathtt{p},\mathtt{q})$ is not completely on the equator, then `h` is on the segment iff $\mathtt{p} \leq \mathtt{h} \leq \mathtt{q}$, where "$\leq$" is the order implied by the compare function of Section 5.2.2. Otherwise, there are two different cases. First, both endpoints are on the same side of the y-axis. In this subcase `h` is on the segment iff $\mathtt{p} \leq \mathtt{h} \leq \mathtt{q}$. Second, the segment crosses the negative or positive part of the y-axis (see Figures 1(a) and (b)).

Suppose the segment crosses the positive part of the y-axis. In this case the point `h` is on the segment if one of the two following statements holds. First, the x-coordinate of `h` is negative and $\mathtt{p} \leq \mathtt{h}$. Second, the x-coordinate of `h` is positive or zero and $\mathtt{q} \leq \mathtt{h}$. Recall that the segment and the point are on the equator. If the segment crosses the negative part of the y-axis, we need some similar tests.

How can we test if the segment crosses the positive or negative part of the y-axis? We only have the endpoints of the segment. The fact that our `sphere_points` are vectors with arbitrary length does not make the test easier.

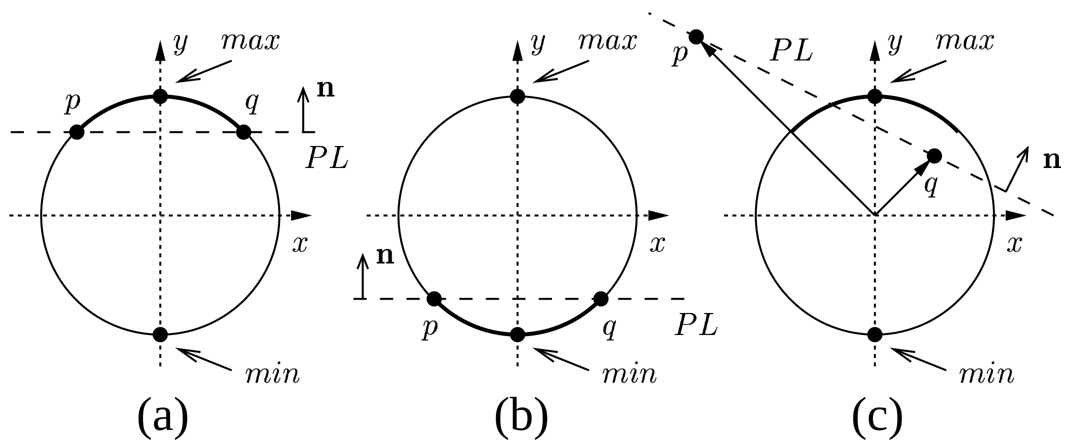

Figure 1: *Does the segment contain the minimum or the maximum point? A view from the position* $(0, 0, \infty)$. *min resp. max is the minimum resp. maximum point.*

Recall that all segments are shorter than half of a great circle and that the segments are directed. We make the observation that the segment `(p,q)` crosses the positive (resp. negative) part of the y-axis iff `p`, `q` and the origin make a rightturn (resp. leftturn). This observation is also true if the endpoints of the segment are not unit vectors (see Figure 1(c)). Let `PL` be a plane through the endpoints `p` and `q` of the segment and a point that is vertically above (w.r.t. the z-direction) `q`. Now, `p`, `q` and the origin make a rightturn iff the origin is on the other side of the normal **n** of `PL`.

⟨*contains*⟩≡
```
bool sphere_segment::contains(const sphere_point& h) const
{
  sphere_point p = a();
  sphere_point q = b();
  d3_rat_plane PL(p.rat_point(), q.rat_point(), d3_rat_point(0,0,0,1));

  if( PL.side_of( h.rat_point() ) != 0 )
    return(false);

  if( p.Z() == 0 && q.Z() == 0 )   // Is the segment on the equator?
    {
      if( p.X() * q.X() > 0 )   // Are both endpoints on the same side
        {                       // of the y-axis?
          if( p <= h && h <= q )
            {
              return(true);
            }
          return(false);
        }
      else
        {
          // Test if the segment crosses the positive or the negative
          // part of the y-axis.
          PL = d3_rat_plane( p.rat_point(),
                             q.rat_point(),
                             q.rat_point() + rat_vector(0,0,1,1) );
          int side_of_O = PL.side_of(d3_rat_point(0,0,0,1));

          if(side_of_O == 1)    // segment crosses the negative part
            {                   // of the y-axis
              if( h.X() <= 0 )
                {
                  if( p >= h )
                    {
                      return(true);
                    }
                  return(false);
                }
```

```
       else
        {
         if( q >= h )
          {
           return(true);
          }
         return(false);
        }
      }

     if(side_of_O == -1)   // segment crosses the positive part
      {                    // of the y-axis
       if( h.X() < 0 )
        {
         if( p <= h )
          {
           return(true);
          }
         return(false);
        }
       else
        {
         if( q <= h )
          {
           return(true);
          }
         return(false);
        }
      }
    }
  }
 // segment is not on the equator
 if( p <= h && h <= q )
  {
   return(true);
  }
 return(false);
}
```

6.2.4 The operators

Since the `sphere_segments` are oriented the operators == and `!=` are easy to define.

⟨*operators*⟩≡
```
  bool operator==(const sphere_segment& x, const sphere_segment& y)
  {
    if(x.normal1 == y.normal1 && x.normal2 == y.normal2)
      return(true);
    return(false);
  }
  bool operator!=(const sphere_segment& x, const sphere_segment& y)
  {
    if(x.normal1 == y.normal1 && x.normal2 == y.normal2)
      return(false);
    return(true);
  }

  ostream& operator« (ostream& s, const sphere_segment& p)
  {
    s « "intersected segment:  " « p.intersected_segment
      « "\nnormal1:  " « p.normal1 « "\nnormal2:  " « p.normal2
      « "\nvertex1:  " « p.v1 « "\nvertex2:  " « p.v2
      « "\nnumber:   " « p.segnumber « endl;
     return s;
  }
  istream& operator» (istream& in, sphere_segment& p)
  {
    // "intersected segment:" p.intersected_segment
    // "normal1:" p.normal1     "normal2:" p.normal2"
    // "vertex1:" p.vertex1     "vertex2:" p.vertex2"

    char c;
    string s;

     do in.get(c); while (isspace(c));
     in.putback(c);
     in » s;
     if(s != "intersected")
       { cerr « "ERROR: sphere_segment input: syntax error" « endl;
         exit(-1);
       }
     in » s;
     if(s != "segment:")
       { cerr « "ERROR: sphere_segment input: syntax error" « endl;
         exit(-1);
       }
     bool is;
     in » is;
```

```
   do in.get(c); while (isspace(c));
   in.putback(c);
   in » s;
   if(s != "normal1:")
     { cerr « "ERROR: sphere_segment input: syntax error" « endl;
       exit(-1);
     }
   sphere_point n1;
   in » n1;
   do in.get(c); while (isspace(c));
   in.putback(c);
   in » s;
   if(s != "normal2:")
     { cerr « "ERROR: sphere_segment input: syntax error" « endl;
       exit(-1);
     }
   sphere_point n2;
   in » n2;

   do in.get(c); while (isspace(c));
   in.putback(c);
   in » s;
   if(s != "vertex1:")
     { cerr « "ERROR: sphere_segment input: syntax error" « endl;
       exit(-1);
     }
   d3_rat_point vert1;
   in » vert1;
   do in.get(c); while (isspace(c));
   in.putback(c);
   in » s;
   if(s != "vertex2:")
     { cerr « "ERROR: sphere_segment input: syntax error" « endl;
       exit(-1);
     }
   d3_rat_point vert2;
   in » vert2;

   do in.get(c); while (isspace(c));
   in.putback(c);
   int h;
   in » h;

   p = sphere_segment(n1, n2, vert1, vert2, is);
   return in;
}
```

7 The file `dual_graph.c`

In this section we compute the dual graph G of the convex hull G_{CH}. To be precise, we compute the dual edges (`sphere_segments`) and the dual points (`sphere_points`). In Sections 8 and 9 we will see that this information is sufficient to solve our problem.

⟨*dual_ graph.c*⟩≡
```
#include <LEDA/list.h>
#include <LEDA/map.h>
#include <LEDA/planar_map.h>
#include <LEDA/rat_vector.h>
#include <LEDA/integer.h>
#include <LEDA/d3_rat_point.h>
#include <LEDA/d3_rat_plane.h>
#include <math.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include <assert.h>
#include "sphere_point.h"
#include "sphere_segment.h"

⟨compare segments⟩
⟨build dual graph⟩
```

7.1 The function `cmp_seg`

This function sorts the segments lexicographically. It is used to eliminate the reversal edges in the list of all edges that we obtain from the LEDA planar map (See Section 7.2.2). Recall that the segments are directed.

⟨*compare segments*⟩≡
```
inline int cmp_seg(const sphere_segment& s1, const sphere_segment& s2)
{
  if(s1.a() < s2.a())
    return(-1);
  else
    if(s1.a() > s2.a())
      return(1);
    else    // s1.a() == s2.a()
      if(s1.b() < s2.b())
        return(-1);
      else
        if(s1.b() > s2.b())
          return(1);
  return(0);
}
```

7.2 The function `build_dual_graph`

This function builds the upper and lower parts G_+ and G_- of the dual graph G.

⟨*build dual graph*⟩≡
```
void build_dual_graph( const PLANAR_MAP<d3_rat_point, int, int>& Poly,
                       map<int, face>& Poly_face,
                       list<sphere_segment>& u_segment,
                       list<sphere_segment>& l_segment,
                       list<sphere_point>& u_point,
                       list<sphere_point>& l_point )
{
  face f, rf;
  int face_nr;
  edge e1, e2, e;
  d3_rat_point a, b, c, ep, rep;
  d3_rat_plane plane;
  sphere_segment seg;
  sphere_point sp;
  list<sphere_segment> S;
  list<sphere_point>   P;
  map<face, sphere_point> dual_node;
  map<face, int> face_number;

  ⟨compute DG vertices⟩
  ⟨compute DG edges⟩
  ⟨compute upper and lower vertices⟩
  ⟨compute upper and lower edges⟩
}
```

7.2.1 Compute the dual graph vertices

We compute a `plane` through each facet `f` of G. The dual of this facet is defined as its outer normal `plane.normal()`. We store some values that help us computing the dual edges without computing the normals again.

⟨*compute DG vertices*⟩≡
```
face_nr = 0;
forall_faces(f, Poly)
{
  e1 = Poly.first_face_edge(f);
  e2 = Poly.face_cycle_succ(e1);

  a = Poly.inf(Poly.source(e1));
  b = Poly.inf(Poly.target(e1));   // target(e1) == source(e2)
  c = Poly.inf(Poly.target(e2));
  assert( ! collinear(a, b, c) );
```

```
    plane = d3_rat_plane(a, b, c);
    sphere_point sp(plane.normal(), face_nr);
    P.append( sp );
    Poly_face[face_nr] = f;
    dual_node[f] = sp;
    face_number[f] = face_nr++;
  }
  cout « "number of dual nodes:   " « P.size() « endl;
```

7.2.2 Compute the dual graph edges

We compute for each edge `e` of G the dual edge `seg`. It has as endpoints the dual of the face of `e` and the dual of the face of the reversal of `e`. Since we compute all edges twice (the edge and its reversal edge) but need them only once, we delete the copies using the `sort` and `unique` functions of the LEDA `list`.

⟨*compute DG edges*⟩≡
```
  forall_faces(f, Poly)
  {
    forall_face_edges(e,f)
      {
        rf = face_of(Poly.reversal(e));    // face of the reversal edge

        a = (dual_node[f]).rat_point();
        b = (dual_node[rf]).rat_point();

        seg = sphere_segment( a, b, Poly.inf(Poly.source(e)),
                              Poly.inf(Poly.target(e)), false );
        S.append( seg );
      }
  }
  // delete the reversal edges
  S.sort(cmp_seg);
  S.unique(cmp_seg);
  cout « "number of dual edges:   " « S.size() « endl « endl;
```

7.2.3 Compute upper and lower vertices

Given a list of all dual vertices, we compute the list `u_point` of the vertices on $\mathbb{S}^2_+$ and the list `l_point` of the remaining vertices. Note, that `l_point` does not contain the vertices on the equator. The vertices in the list `l_point` are reflected to $\mathbb{S}^2_+$.

⟨*compute upper and lower vertices*⟩≡
```
  sphere_point p;

  forall( p, P )
  {
    if( p.Z() >= 0 )
      {
        u_point.append( p );
      }
    if( p.Z() < 0 )    // points on equator are only in u_point
      {
        l_point.append( sphere_point( -p.X(), -p.Y(), -p.Z(), p.W(),
                                      p.number() ) );
      }
  }
```

7.2.4 Compute the sets of upper and lower segments

We test for each segment if it is completely on the upper or lower part of the hemisphere. If so, we append it to the appropriate list `u_segment` resp. `l_segment`. We reflect the segment if it is on the lower part. Thus `u_segment` contains the segments on $\mathbb{S}^2_+$ and `l_segment` contains the reflected segments of $\mathbb{S}^2_-$.

If the segment (`sa`, `sb`) crosses the equator, we cut it. To be precise, we compute the intersection point `ep` of the segment and the equator, thereby obtaining two new segments, (`sa`, `ep`) on $\mathbb{S}^2_+$ and (`ep`, `sb`) on $\mathbb{S}^2_-$. We reflect the last one and append it to the list `l_segment`. The first, we append to `u_segment`.

To compute the intersection point of the segment and the equator, we compute the cross product of two normals `nh` and `norm`. Here `nh` is the normal vector of the plane through the equator. We choose the homogeneous vector $(0, 0, 1, 1)$ as `nh`. `norm` is the normal vector of the plane `splane` through the segment and the origin. We choose the direction for `splane` such that the vector `nh` $\times$ `norm` is on the ray from the origin through the point `ep`. Since the x- and y-coordinates of `nh` are zero the cross product is easy to compute. The z-coordinate of the cross product is equal to zero.

⟨*compute upper and lower edges*⟩≡

```
  sphere_point   sa, sb;

  forall(seg, S)
  {
    sa = seg.a();
    sb = seg.b();

    if( sa.Z() >= 0 && sb.Z() >= 0 )  // sa and sb on the upper sphere
      {
        u_segment.append(seg);
      }
    if( sa.Z() <= 0 && sb.Z() <= 0 )  // sa and sb on the lower sphere
      {
        sphere_point ma( -sa.X(), -sa.Y(), -sa.Z(), sa.W(), sa.number() );
        sphere_point mb( -sb.X(), -sb.Y(), -sb.Z(), sb.W(), sb.number() );
    l_segment.append(sphere_segment( ma, mb, seg.vertex1(),
                                     seg.vertex2(), false ));
  }

// one point on the upper sphere and one on the lower sphere
if( sa.Z() * sb.Z() < 0 )
  {
    // compute the intersection point with the equator
    d3_rat_plane splane;

    if(sa.Z() > 0 && sb.Z() < 0)
      {
        splane = d3_rat_plane( sa.rat_point(), sb.rat_point(),
                               d3_rat_point(0,0,0));
      }
    else
      {
        assert( sa.Z() < 0 && sb.Z() > 0 );
        splane = d3_rat_plane( sb.rat_point(), sa.rat_point(),
                               d3_rat_point(0,0,0) );
      }
    rat_vector norm = splane.normal();

    integer x =  -norm.Y();   // cross product (0,0,1,1) x norm
    integer y = norm.X();
    integer w = norm.W();

    ep  = d3_rat_point( x,  y, 0, w);   // point on the equator
    rep = d3_rat_point(-x, -y, 0, w);   // reflected point on the equator

    assert( splane.contains(ep) && splane.contains(rep) );

    if(sa.Z() > 0)
      {
        sphere_point mb( -sb.X(), -sb.Y(), -sb.Z(), sb.W(),
                         sb.number() );
        u_segment.append(sphere_segment( sa, ep, seg.vertex1(),
                                         seg.vertex2(), true ));
        l_segment.append(sphere_segment( rep, mb, seg.vertex1(),
                                         seg.vertex2(), true ));
      }
    else  // a.Z() < 0
      {
        sphere_point ma( -sa.X(), -sa.Y(), -sa.Z(), sa.W(),
                         sa.number() );
        u_segment.append(sphere_segment( sb, ep, seg.vertex1(),
                                              seg.vertex2(), true ));
              l_segment.append(sphere_segment( rep, ma, seg.vertex1(),
                                              seg.vertex2(), true ));
          }
      }
  }
```

8 The file `sweep_segments.c`

To compute the *EE*-pairs we have to find the intersecting `sphere_segments` of G_+ and G_-. (Refer to Section 3.) We can compute these by adapting the Bentley and Ottmann sweep line algorithm for the segment intersection problem. Since we do not make the assumption that the segments are in general position we adapt the robust and efficient implementation of Bartuschka et al.[2] so that it works for great arcs on the upper hemisphere. Since the code is almost identical to that of [2], we present it without comments.

⟨*sweep_ segments.c*⟩≡

```
  #include <LEDA/sortseq.h>
  #include <LEDA/d3_rat_point.h>
  #include <LEDA/d3_rat_plane.h>
  #include <LEDA/vector.h>
  #include <LEDA/planar_map.h>
  #include <LEDA/map.h>
  #include <LEDA/list.h>
  #include <LEDA/map2.h>
  #include <LEDA/p_queue.h>
  #include <iostream.h>
  #include <fstream.h>
  #include <assert.h>
  #include "sphere_segment.h"

  int               number_of_intersections;
  sphere_point      p_sweep;
  static sphere_segment   lower_sentinel;
  static sphere_segment   upper_sentinel;

  ⟨compare function⟩
  ⟨compute_ intersection function⟩
  ⟨sweep_ segments function⟩
  ⟨EE_ min_ distance function⟩
```

8.1 The function `compare`

The `compare` function has as precondition that the sweep half-plane intersects both segments and at least one source point of a segment is equal to the sweep point `p_sweep`. This function is needed to order the `sphere_segments` according to their intersections with the sweep half-plane. It tests if the `sphere_segment` on which `p_sweep` lies is above (w.r.t. the sweep circle) the other `sphere_segment`.

⟨*compare function*⟩≡
```
int cmp_segments(const sphere_segment& s1, const sphere_segment& s2)
{
  if( s1.a() == s2.a() && s1.b() == s2.b() )
    return 0;

  int s = 0;

  if( p_sweep == s1.a() )
    {
      s = orientation(s2, p_sweep);
    }
  else
    {
      assert( p_sweep == s2.a() );
      s = orientation(s1, p_sweep);
    }
  if(s != 0 || s1.is_trivial() || s2.is_trivial())
    {
      return(s);
    }

  s = orientation(s2, s1.b());
  return s ? s : (s2.number() - s1.number());
}

int compare(const sphere_segment& s1, const sphere_segment& s2)
{ return cmp_segments(s1,s2); }
```

8.2 The function `compute_intersection`

⟨*compute_ intersection function*⟩≡
```
void compute_intersection(sortseq<sphere_point, seq_item>& X_structure,
                          sortseq<sphere_segment, seq_item>& Y_structure,
                          const map2<int,int,seq_item>& inter_dic,
                          seq_item sit0,
                          map<int, sphere_segment>& original,
                          list<sphere_segment>& S1,
                          list<sphere_segment>& S2)
```

```
  {
    seq_item sit1 = Y_structure.succ(sit0);
    sphere_segment s0 = Y_structure.key(sit0);
    sphere_segment s1 = Y_structure.key(sit1);

    if ( orientation(s0,s1.b()) <= 0 && orientation(s1,s0.b()) >=0 )
      {
        seq_item it = inter_dic.operator()(s0.number(),s1.number());
        //seq_item it = inter_dic(s0,s1);
        sphere_point q;
        if (it == nil)
          {
            s0.intersection(s1, q);
            it = X_structure.insert(q,sit0);
          }
        Y_structure.change_inf(sit0, it);

        if ( orientation(s0,s1.b()) < 0 && orientation(s1,s0.b()) > 0 )
          {
            if(original[s0.number()] != lower_sentinel &&
               original[s1.number()] != upper_sentinel)
              {
                number_of_intersections++;
                S1.push(original[s0.number()]);
                S2.push(original[s1.number()]);
              }
          }
      }
  }
```

8.3 The function `sweep_segments`

⟨*sweep_ segments function*⟩≡

```
  void sweep_segments( const list<sphere_segment>& S,
                       list<sphere_segment>& S1,
                       list<sphere_segment>& S2 )
  {
    sortseq<sphere_point, seq_item> X_structure;
    sortseq<sphere_segment, seq_item> Y_structure(cmp_segments);
    map<int, sphere_segment> original;
    map2<int, int, seq_item> inter_dic(nil);
    p_queue<sphere_point, sphere_segment> seg_queue;
    sphere_segment s;

    ⟨initialization⟩
    while ( !X_structure.empty() )
      {
        ⟨extract next event from the X-structure⟩
        ⟨handle passing and ending segments⟩
        ⟨insert starting segments⟩
        ⟨compute new intersections and update X-structure⟩

        X_structure.del_item(event);
      }
    cout « "number of EE-pairs:     " « number_of_intersections « endl;
  }
```

8.3.1 Initialization

⟨initialization⟩≡

```
  number_of_intersections = 0;

  forall(s, S)
  {
    seq_item it1 = X_structure.insert(s.a(), seq_item(nil));
    seq_item it2 = X_structure.insert(s.b(), seq_item(nil));

    if (it1 == it2) continue;  // ignore zero-length segments

    sphere_point p = X_structure.key(it1);
    sphere_point q = X_structure.key(it2);

    sphere_segment s1 = sphere_segment(p, q);
    original[s1.number()] = s;
    seg_queue.insert(s1.a(), s1);
  }

  sphere_point pos_infty_a(10,-1,-1,1);
  sphere_point pos_infty_b(10,1,-1,1);
  sphere_point neg_infty_a(-10,-1,-1,1);
  sphere_point neg_infty_b(-10,1,-1,1);
  lower_sentinel = sphere_segment(neg_infty_a, neg_infty_b);
  upper_sentinel = sphere_segment(pos_infty_a, pos_infty_b);

  p_sweep = lower_sentinel.a();
  Y_structure.insert(upper_sentinel,seq_item(nil));
  Y_structure.insert(lower_sentinel,seq_item(nil));

  seg_queue.insert(upper_sentinel.a(),upper_sentinel);
  sphere_segment next_seg = seg_queue.inf(seg_queue.find_min());
```

8.3.2 Extract next event from the X-structure

⟨*extract next event from the X-structure*⟩≡
```
  seq_item event = X_structure.min();
  p_sweep = X_structure.key(event);
```

8.3.3 Handle passing and ending segments

⟨*handle passing and ending segments*⟩≡
```
  seq_item sit = nil;
  sit = X_structure.inf(event);

  if (sit == nil)
  {
    sit = Y_structure.lookup(sphere_segment(p_sweep,p_sweep));
  }

  seq_item sit_succ  = nil;
  seq_item sit_pred  = nil;
  seq_item sit_first = nil;

  if(sit != nil)
  {
    while ( Y_structure.inf(sit) == event ||
            Y_structure.inf(sit) == Y_structure.succ(sit) )
      {
        sit = Y_structure.succ(sit);
      }

    sit_succ = Y_structure.succ(sit);

    seq_item xit = Y_structure.inf(sit);
    if (xit)
      {
        sphere_segment s1 = Y_structure.key(sit);
        sphere_segment s2 = Y_structure.key(sit_succ);
        inter_dic(s1.number(),s2.number()) = xit;
      }

    bool upperst;
    do
      {
        upperst = false;
        s = Y_structure.key(sit);

        if ( p_sweep == s.b() )   //ending segment
        {
          seq_item it = Y_structure.pred(sit);
          if ( Y_structure.inf(it) == sit )
            {
              upperst = true;
              Y_structure.change_inf(it, Y_structure.inf(sit));
            }
          Y_structure.del_item(sit);
          sit = it;
        }
      else  //passing segment
        {
          if ( Y_structure.inf(sit) != Y_structure.succ(sit) )
            {
              Y_structure.change_inf(sit, seq_item(nil));
            }
          sit = Y_structure.pred(sit);
        }
    } while ( Y_structure.inf(sit) == event || upperst ||
              Y_structure.inf(sit) == Y_structure.succ(sit) );


  sit_pred = sit;
  sit = Y_structure.succ(sit_pred);
  sit_first = sit;

  while ( sit != sit_succ  && Y_structure.succ(sit) != sit_succ)
    {
      seq_item sub_first = sit;
      seq_item sub_last  = sub_first;

      while(  Y_structure.inf(sub_last) == Y_structure.succ(sub_last) )
        sub_last = Y_structure.succ(sub_last);

      if( sub_last != sub_first )
        Y_structure.reverse_items(sub_first, sub_last);

      sit = Y_structure.succ(sub_first);
    }

  if (Y_structure.succ(sit_pred) != sit_succ )
    Y_structure.reverse_items(Y_structure.succ(sit_pred),
                              Y_structure.pred(sit_succ));
}
```

8.3.4 Insert starting segments

⟨*insert starting segments*⟩≡
```
while (p_sweep == next_seg.a() )
{
  seq_item s_sit = Y_structure.locate(next_seg);
  seq_item p_sit = Y_structure.pred(s_sit);

  s   = Y_structure.key(s_sit);

  if( !orientation(s, next_seg.a()) &&
      !orientation(s, next_seg.b() ) )
    sit = Y_structure.insert_at(s_sit, next_seg, s_sit);
  else
    sit = Y_structure.insert_at(s_sit, next_seg, seq_item(nil));

  s = Y_structure.key(p_sit);

  if( !orientation(s, next_seg.a()) &&
      !orientation(s, next_seg.b() ) )
    Y_structure.change_inf(p_sit, sit);

  X_structure.insert(next_seg.b(), sit);

  if ( sit_succ == nil )
    {
      sit_succ = Y_structure.succ(sit);
      sit_pred = Y_structure.pred(sit);
      sit_first = sit_succ;
    }

  seg_queue.del_min();
  next_seg = seg_queue.inf(seg_queue.find_min());
}
```

8.3.5 Compute new intersections and update X-structure

⟨*compute new intersections and update X-structure*⟩≡
```
if (sit_pred != nil)
{
  seq_item xit = Y_structure.inf(sit_pred);
  if ( xit )
    {
      sphere_segment s1 = Y_structure.key(sit_pred);
      sphere_segment s2 = Y_structure.key(sit_first);
      inter_dic(s1.number(),s2.number()) = xit;
      Y_structure.change_inf(sit_pred, seq_item(nil));
      }

    compute_intersection(X_structure, Y_structure, inter_dic,
                         sit_pred, original, S1, S2);
    sit = Y_structure.pred(sit_succ);
    if (sit != sit_pred)
      {
        compute_intersection(X_structure, Y_structure, inter_dic,
                             sit, original, S1, S2);
      }
  }
```

8.4 The function `EE_min_distance`

The algorithm of Section 8.3 computes all possible *EE*-pairs. The function `EE_min_distance` computes the minimum distance among the corresponding pairs of supporting planes. The results of this function are the minimum distance `min_dist`, the number `number_min_distance` of *EE*-pairs with this minimum distance and two lists `S1` and `S2`. `S1` and `S2` contain the segments that are the duals of the edges of the *EE*-pairs. The dual of the *i*th segment in list `S1` forms an *EE*-pair with the dual of the *i*th segment in list `S2`.

⟨*EE_ min_ distance function*⟩≡

```
  void EE_min_distance( list<sphere_segment>& S1,
                        list<sphere_segment>& S2,
                        rational& min_dist,
                        int& number_min_dist,
                        const PLANAR_MAP<d3_rat_point, int, int>& Poly )
  {
    list<sphere_segment> H1;
    list<sphere_segment> H2;
    sphere_segment s1, s2;
    d3_rat_point a1, a2, b1, b2;
    rat_vector v(3);
    rational dist, mdist = -1;

    assert( S1.size() == S2.size() );
    while( ! S1.empty() )
      {
        s1 = S1.pop();
        s2 = S2.pop();

        a1 = s1.vertex1();
        b1 = s1.vertex2();
        a2 = s2.vertex1();
        b2 = s2.vertex2();
        // a1, .., b2 should not be coplanar! -> EE-pair
        assert( orientation(a1, a2, b1, b2) != 0 );

        v = b2 - a2;
        d3_rat_plane p(a1, b1, a1+v);
        dist = p.sqr_dist( a2 );
        assert( dist == p.sqr_dist( b2 ) );

        if( (dist < mdist) || (mdist == -1) )
          {
            mdist = dist;
            H1.clear();
            H2.clear();
            H1.append(s1);
            H2.append(s2);
          }
        else
          {
            if(dist == mdist)
              {
                H1.append(s1);
                H2.append(s2);
              }
          }
      }
    min_dist = mdist;
    number_min_dist = H1.size();
  }
```

9 The file `point_location.c`

We compute the *VF*-pairs using a point location algorithm based on the *slab method* of Dobkin and Lipton. Since we know all points that we want to locate in advance, it is not necessary to store the slabs. (Refer to Section 3.)

⟨*point_location.c*⟩≡

```
  #include <LEDA/plane.h>
  #include <LEDA/map.h>
  #include <LEDA/p_queue.h>
  #include <LEDA/sortseq.h>
  #include <LEDA/d3_rat_plane.h>
  #include <LEDA/planar_map.h>
  #include <assert.h>
  #include <iostream.h>
  #include <fstream.h>
  #include "sphere_segment.h"
```

```
extern sphere_point   p_sweep; // current position of the sweep half-plane
static sphere_segment lower_sentinel;
static sphere_segment upper_sentinel;

extern int compare(const sphere_segment& s1, const sphere_segment& s2);

typedef sortseq<sphere_point,seq_item>     X_structure;
typedef sortseq<sphere_segment,seq_item>   Y_structure;
```

⟨*locate_ one_ point*⟩
⟨*locate_ points*⟩
⟨*VF_ min_ distance*⟩

9.1 The function `locate_one_point`

This function takes a point `p` that we want to locate, and a sorted list `Y` containing all segments that cross a slab. It locates the point `p` in the slab `Y`. The result is an edge of the face that contains the point. We store this edge in a list `E` and the point `p` in a list `V`. Note that the i-th edge in `E` belongs to the i-th vertex in `V`.

⟨*locate_ one_ point*⟩≡

```
void locate_one_point( const sphere_point &p,
                       const Y_structure &Y,
                       list<sphere_point>& V,
                       list<sphere_segment>& E )
{
  seq_item sit;
  sphere_point h = p_sweep;
  p_sweep = p;

  sphere_segment lookseg(p,p);
  sit = Y.lookup(lookseg);

  // is sit nil or a sentinel?
  if( sit == seq_item(nil) || sit == Y.min() || sit == Y.max() )
    {
      sit = Y.locate_succ(lookseg);
      if( sit == nil || sit == Y.min() || sit == Y.max() )
        {
          sit = Y.locate_pred(lookseg);
        }
    }
  assert(sit != nil && sit != Y.min() && sit != Y.max() );

  E.append( Y.key(sit) );
```

```
    V.append( p );
    p_sweep = h;
  }
```

9.2 The function `locate_points`

The precondition of this algorithm is that `L` is a list of non-intersecting **`sphere_segment`**s. These **`sphere_segment`**s are directed, see Section 6.1.2. The algorithm sweeps over $\mathbb{S}^2_+$ (as explained in Section 3) and maintains a sorted list `Y` of all segments intersecting the sweep plane *SP*.

As long as the list `X` of sweep events and the list `P` of points that we want to locate are not empty we do the following. First, we update **`p_sweep`** to the next sweep event. After that, we update the sorted list `Y` of segments that are intersected by *SP*. To be precise, we delete all segments ending in point **`p_sweep`** and then we insert all segments starting in **`p_sweep`**. Finally, we do the point location with all points of `P` that are in this slab. Note that the points in `P` are sorted w.r.t. *SP* (see Section 9.2.1).

⟨*locate_points*⟩≡
```
  void locate_points( list<sphere_point>& P,
                      list<sphere_segment>& L,
                      list<sphere_point>& V,
                      list<sphere_segment>& E )
  {
    ⟨initialization⟩
    sphere_point loc_point = P.pop();
    while( ! X.empty() && ! P.empty() )
      {
        seq_item event = X.min();  // next event
        p_sweep = X.key(event);

        seq_item sit = nil;
        sit = Y.lookup(sphere_segment(p_sweep,p_sweep));
        while( sit != nil )
          {
            Y.del_item(sit);
            sit = Y.lookup(sphere_segment(p_sweep,p_sweep));
          }

        while (p_sweep == next_seg.a() )
          {
            seq_item sit = Y.locate(next_seg);

            sit = Y.insert_at(sit, next_seg, seq_item(nil));
            X.insert(next_seg.b(), sit);

            seg_queue.del_min();
          next_seg = seg_queue.inf(seg_queue.find_min());
        }

      X.del_item(event);

      if( ! X.empty() )
        {
          while( loc_point < X.key(X.min()) && ! P.empty() )
            {
              locate_one_point( loc_point, Y, V, E );
              loc_point = P.pop();
            }
          if( loc_point < X.key(X.min()) && P.empty() )
            {
              locate_one_point( loc_point, Y, V, E );
            }
        }
      else
        {
          while( ! P.empty() )
            {
              locate_one_point( loc_point, Y, V, E );
              loc_point = P.pop();
            }
        }
    }
}
```

9.2.1 Initialization

First we initialize the X structure (sweep events). After that, we introduce some sentinels. These infinity segments are stored in the Y structure to avoid tests like "is the Y structure empty?". The priority queue `seg_queue` contains all segments to the right of the sweep half-plane, sorted by their left endpoint. Thus it is easy to test if a segment has to be inserted at the actual sweep half-plane position.

⟨*initialization*⟩≡
```
P.sort();
X_structure   X;
Y_structure   Y;
p_queue<sphere_point, sphere_segment> seg_queue;
sphere_segment s;

forall(s,L)   // initialize the X structure
{
  seq_item it1 = X.insert(s.a(), seq_item(nil));
  seq_item it2 = X.insert(s.b(), seq_item(nil));
```

```
    if (it1 == it2) continue;  // ignore zero-length segments

    seg_queue.insert(s.a(), s);
  }

  sphere_point pos_infty_a( 10,-1,-1, 1);
  sphere_point pos_infty_b( 10, 1,-1, 1);
  sphere_point neg_infty_a(-10,-1,-1, 1);
  sphere_point neg_infty_b(-10, 1,-1, 1);

  lower_sentinel = sphere_segment(neg_infty_a, neg_infty_b);
  upper_sentinel = sphere_segment(pos_infty_a, pos_infty_b);

  p_sweep = upper_sentinel.a();
  Y.insert(lower_sentinel,seq_item(nil));
  Y.insert(upper_sentinel,seq_item(nil));

  seg_queue.insert(upper_sentinel.a(),upper_sentinel);
  sphere_segment next_seg = seg_queue.inf(seg_queue.find_min());
```

9.3 The function `VF_min_distance`

The point location algorithm of Section 9.2 computes all possible *VF*-pairs. With the function `VF_min_distance` we compute all *VF*-pairs for which the distance between the corresponding supporting planes is minimum. The direction that minimizes the width is determined either by a *VF*-pair or an *EE*-pair (see Section 3). Since we have already computed the *EE*-pairs for which the distance between the corresponding supporting planes is minimum in Section 8.4, we are only interested in *VF*-pairs for which the distance between the corresponding supporting planes is smaller. At the start the minimum distance of the *EE*-pair supporting planes is stored in the parameter `min_dist`. Its value is "-1" if there are no *EE*-pairs.

We loop over all *VF*-pairs that are stored in the lists `P` and `E`. In each iteration, we take a point `p` which is the dual of a facet and a segment `seg` which is the dual of an edge. Thereafter, we compute a plane `p1` through the facet of the dual of `p` which is the "facet" of the *VF*-pair. Note that the primal edge (`seg.vertex1()`, `seg.vertex2()`) is also stored with our dual segment `sphere_segment`. The vertex `v` of the primal edge with maximum distance to `p1` is the "vertex" of the *VF*-pair. We compute the distance `dist` of the parallel planes that are supporting the *VF*-pair. In fact, because of precision problems we always compute the square of the distance. We store all *VF*-pairs with minimum distance of `p1` and `v` in the lists `VERTEX` and `FACE`. At the start of the loop the variable `number_min_dist` contains the number of *EE*-pairs whose supporting planes have minimum distance. At the end we update it to the total number of directions with minimum width.

⟨*VF_ min_ distance*⟩≡
```
 void VF_min_distance( const list<sphere_point>& P,
                       const list<sphere_segment>& E,
                       const map<int, face>& Poly_face,
                       const PLANAR_MAP<d3_rat_point, int, int>& Poly,
                       rational& min_dist,
                       int& number_min_dist )
 {
   list_item       pit, eit;
   sphere_point    p;
   sphere_segment  seg;
   rational        mdist, dist;

   list<face>          FACE;
   list<d3_rat_point>  VERTEX;

   assert( P.size() == E.size() );

   mdist = min_dist;
   eit = E.first();
   pit = P.first();

   while( pit != nil )
     {
       seg = E.inf(eit);
       p   = P.inf(pit);

       assert( Poly_face.defined(p.number()) );
       face f = Poly_face[p.number()];

       d3_rat_point a = Poly.inf( Poly.source( Poly.first_face_edge(f) ) );
       d3_rat_plane p1( a, p.rat_point().to_vector() );
       d3_rat_point v;
       if( p1.sqr_dist( seg.vertex1() ) >= p1.sqr_dist( seg.vertex2() ) )
         {
           v = seg.vertex1();
         }
       else
         {
           v = seg.vertex2();
         }

       dist = p1.sqr_dist(v);
       if( dist <= mdist || mdist == -1 )
         {
           if( dist < mdist || mdist == -1 )
```

```
            {
              mdist = dist;
              FACE.clear();
              VERTEX.clear();
            }
          FACE.append(f);
          VERTEX.append(v);
        }

      eit = E.succ(eit);
      pit = P.succ(pit);
    }

  if( mdist < min_dist || min_dist == -1 )
    {
      min_dist = mdist;
      number_min_dist = VERTEX.size();
    }
  else
    {
      if( mdist == min_dist )
        {
          number_min_dist += VERTEX.size();
        }
    }
}
```

10 The main part of the program

This section describes the main function.

⟨*Cwidth.c*⟩≡

```
#include <LEDA/list.h>
#include <LEDA/map.h>
#include <LEDA/planar_map.h>
#include <LEDA/d3_rat_point.h>
#include <LEDA/d3_hull.h>
#include <stdlib.h>
#include <iostream.h>
#include <fstream.h>
#include "sphere_segment.h"

extern void load_points( const string& name, list<d3_rat_point> &L );
extern void save_points( const string& name, const list<d3_rat_point> &L);
extern void check_arguments(int, char *argv[], int&,
                            list<d3_rat_point>& L,
                            list<rat_vector>& STL );
extern void build_dual_graph( const PLANAR_MAP<d3_rat_point, int, int>& Poly,
                              map<int, face>& Poly_face,
                              list<sphere_segment>& u_segment,
                              list<sphere_segment>& l_segment,
                              list<sphere_point>& u_point,
                              list<sphere_point>& l_point );
extern void sweep_segments( const list<sphere_segment>& S,
                            list<sphere_segment>& S1,
                            list<sphere_segment>& S2 );
extern void EE_min_distance( list<sphere_segment>& S1,
                             list<sphere_segment>& S2,
                             rational& min_dist,
                             int& number_min_dist,
                             const PLANAR_MAP<d3_rat_point, int, int>& Poly );
extern void locate_points( list<sphere_point>& P,
                           list<sphere_segment>& S,
                           list<sphere_point>& V,
                           list<sphere_segment>& E );
extern void VF_min_distance( const list<sphere_point>& P,
                       const list<sphere_segment>& E,
                       const map<int, face>& Poly_face,
                       const PLANAR_MAP<d3_rat_point, int, int>& Poly,
                       rational& min_dist,
                       int& number_min_dist );

int  _rand      = 0;
bool _stl       = false;

int main(int argc, char *argv[])
{
  ⟨initialization⟩

  float time = used_time();

  ⟨convex hull⟩
  ⟨dual graph⟩
  ⟨sweep segment intersection⟩
  ⟨point location⟩

  time = used_time(time);
  cout « "\ntime = " « time « "sec" « endl « endl;
}
```

10.1 The initialization

The function `check_arguments` checks the arguments of the program and either reads the given file or sets the value of the variable `_rand`. The value in `_rand` indicates whether we create random points in the cube, in the ball or on the sphere. If the given file is an *STL* file all points of the polyhedron are stored in the list `L` and the normals are stored in the list `STL`. The duplicates of the points in the *STL* file are removed so `L` contains each point exactly once. If no parameter is given, a default file is loaded.

⟨*initialization*⟩≡
```
int number_rpoints;
list<d3_rat_point> L;
list<rat_vector> STL;
GRAPH<d3_rat_point,int> H;

rational min_dist = -1;   // sqr of the minimum distance
int number_min_dist = 0;  // number of directions with minimum distance

check_arguments(argc, argv, number_rpoints, L, STL);

if(_rand)
{
  cout « "\nCREATE " « number_rpoints « " RANDOM POINTS ";

  switch(_rand)
    {
    case 1:
      cout « "IN CUBE\n\n";
      random_d3_rat_points_in_cube( number_rpoints, 1000, L );
    break;
    case 2:
      cout « "IN BALL\n\n";
      random_d3_rat_points_in_ball( number_rpoints, 1000, L );
    break;
    case 3:
      cout « "ON SPHERE\n\n";
      random_d3_rat_points_on_sphere( number_rpoints, 1000, L );
    break;
    }
  save_points("t",L);
}

if( ! ( _stl ||  _rand ) )
{
  cout « "\nLOAD POINTS";
  load_points( "t", L );
}
```

10.2 Compute the 3D convex hull with LEDA

This function computes the 3D convex hull with the LEDA function D3_HULL and stores the result as a LEDA PLANAR_MAP. A member function of this combinatorial embedding of a planar graph can compute the needed faces.

⟨*convex hull*⟩≡
```
cout « "number of points:   " « L.size() « endl « endl;
cout « "CONVEX HULL\n\n";

D3_HULL( L, H );
L.clear();

list<edge> R;
H.make_map(R);
R.clear();
H.make_planar_map();
H.compute_faces();

PLANAR_MAP<d3_rat_point, int, int> Poly(H);
Poly.compute_faces();
H.clear();

cout « "number of CH nodes:     " « Poly.number_of_nodes() « endl;
cout « "number of CH edges:     " « Poly.number_of_edges()/2 « endl;
cout « "number of CH faces:     " « Poly.number_of_faces() « endl « endl;
```

10.3 Compute the dual graph

To compute the dual graph we simply call the function build_dual_graph which was explained in Section 7.2.

⟨*dual graph*⟩≡
```
cout « "DUAL GRAPH\n\n";
list<sphere_segment> u_segment;
list<sphere_segment> l_segment;
list<sphere_point> u_point;
list<sphere_point> l_point;
map<int, face> Poly_face;   // map from DG point number to Poly face

build_dual_graph( Poly, Poly_face, u_segment, l_segment, u_point, l_point );
```

10.4 Segment intersection

First, we copy all segments of the lists u_segment and l_segment to the list S. Thereafter, we call the segment intersection algorithm. Note that neither the segments in u_segment nor the segments in l_segment intersect each other. The *EE*-pairs are stored in the lists S1 and S2. Note, that the ith sphere_segment of list S1 intersects the ith sphere_segment of list S2. The function EE_min_distance computes among all *EE*-pairs those for which the distance between the corresponding supporting planes is minimum.

⟨*sweep segment intersection*⟩≡
```
  cout « "SWEEP SEGMENTS\n\n";

  list<sphere_segment> S;
  list<sphere_segment> S1, S2;
  sphere_segment seg;
  sphere_point p;

  S = u_segment;
  forall(seg, l_segment)
  {
    S.append(seg);
  }

  sweep_segments(S, S1, S2);
  S.clear();

  EE_min_distance(S1, S2, min_dist, number_min_dist, Poly);
  S1.clear();
  S2.clear();
  cout « "\nEE:\n" « "number of optimal directions:     " « number_min_dist
       « endl « "square of the minimum distance:   " « min_dist.to_double()
       « endl « endl;
```

10.5 Point location

The first call of the function locate_points locates the points of G_- in the graph G_+. The second call locates the points of G_+ in the graph G_-. The results of these function calls are stored in the lists P and E. Here the ith point of list P belongs to the ith segment of list E. Note that the dual of such a pair is a *VF*-pair.

⟨*point location*⟩≡
```
  list<sphere_point> P;
  list<sphere_segment> E;

  cout « "\nLOCATE LOWER POINTS\n";
  locate_points( l_point, u_segment, P, E );
  cout « "LOCATE UPPER POINTS\n\n";
  locate_points( u_point, l_segment, P, E );

  l_point.clear();
  u_point.clear();
```

model	n	h	k	time
`tod21.stl`	1,128	87	20	13
`mj.stl`	2,832	356	29	31
`triad1.stl`	11,352	3,874	3,769	504
`daikin_trt321.stl`	19,402	1,638	3,036	155
`impller.stl`	30,900	414	288	42
`eaton_sp.stl`	41,318	1,065	1,207	114
`4501005.stl`	50,626	2,306	2,063	165
`frame_29.stl`	67,070	388	312	31
`sa600280.stl`	74,350	3,899	2,669	375
`fishb.stl`	213,384	5,459	5,845	566

Table 1: *Performance of our implementation on some polyhedral models. n denotes the number of facets of the model; h and k denote the number of convex hull facets, and the number of EE-pairs, respectively;* time *denotes the time in seconds.*

```
l_segment.clear();
u_segment.clear();

VF_min_distance( P, E, Poly_face, Poly, min_dist, number_min_dist );

cout « "\nTOTAL:\n"
     « "number of optimal directions:     " « number_min_dist « endl
     « "square of the minimum distance:   " « min_dist.to_double()
     « endl « endl;
```

11 Experimental results

In this section, we report on the experiments we did on a SUN Ultra (300 MHz, 512 MByte RAM).

First, we tested our implementation on real-world polyhedral models obtained from Stratasys, Inc. Table 1 gives the test results for ten models, which were chosen to encompass different geometries. For example, `tod21.stl` is a bracket, consisting of a hollow quarter-cylinder, with two flanges at the ends, and a through-hole drilled in one of the flanges. This model has 1,128 facets. The model `mj.stl` is an anvil shaped like a pistol with a square barrel. This model has 2,832 facets. The largest model tested is `fishb.stl`, which has 213,384 facets. Our program computed the width of the latter model within ten minutes.

As can be seen in Table 1, the actual running time of the program heavily depends on the number, h, of facets of the convex hull. This is not surprising, because our compare functions are fairly complex. For each model that we tested, the value of h is much smaller than the number, n, of facets of the model.

N	h	k	min	max	average	variance
1,000	133	67	2.1	3.0	2.5	0.870
5,000	205	97	3.2	5.0	4.1	0.266
10,000	239	110	4.0	5.2	4.8	0.144
20,000	254	120	4.8	6.4	5.4	0.177
30,000	269	119	5.0	6.9	6.0	0.300
40,000	265	121	5.3	7.3	6.1	0.384
50,000	269	121	6.0	7.7	6.5	0.293
60,000	267	113	5.9	7.3	6.6	0.167
70,000	280	123	6.4	8.8	7.2	0.620
80,000	266	115	5.8	8.3	7.1	0.459
90,000	268	117	6.0	8.8	7.4	0.819
100,000	274	118	7.2	8.6	7.8	0.213

Table 2: *Performance of our implementation for points randomly chosen in a cube. For each value of N, we randomly generated ten point sets of size N. h and k denote the average number of convex hull facets, and the average number of EE-pairs, respectively. Although k could be $\Theta(h^2)$, this table shows that in practice, it is slightly less than $h/2$.* min, max, average, *and* variance *denote the minimum, maximum, and average time in seconds, respectively, and the variance.*

Since we only have a limited number of polyhedral models, we also tested our implementation on random point sets. First, we used LEDA's point generator `random_d3-_rat_points_in_cube` to generate random points from a uniform distribution in the cube $[-1000, 1000]^3$. For each value of $N \in \{10^3, \ldots, 10^5\}$, we generated ten point sets of size N. We measured the time of our program after these points were generated. Table 2 shows the minimum, maximum, and average running time in seconds, as well as the variance. This variance was computed using the formula [5, Section 8.2]

$$\frac{t_1^2 + t_2^2 + \ldots + t_m^2}{m-1} - \frac{(t_1 + t_2 + \ldots + t_m)^2}{m(m-1)},$$

where t_i denotes the time of the i-th run, and m denotes the total number of generated point sets (which is ten in our case).

Also in Table 2, the average values of h (the number of facets of the convex hull), and k (the number of *EE*-pairs) are given. Note that for this distribution, the expected value of h is bounded by $O(\log^2 N)$, see Section 4.1 in [9].

Although the worst-case running time of the algorithm is $\Theta(N^2 \log N)$, our experimental results show that on random inputs, the algorithm is much faster. The actual worst-case performance is bounded by $O(N^2 + k \log h)$. As we can see in Table 2, the value of h is much smaller than N. Also, the value of k—which could be as large as $\Theta(h^2)$—is in fact slightly less than $h/2$. Table 2 shows that in practice, the running time is not proportional to N^2: otherwise, doubling N would increase the running time by at least a factor of four. This implies that the constant factor corresponding to the term

N	h	k	min	max	average	variance
1,000	258	141	4.9	5.9	5.3	0.870
5,000	607	325	12.2	14.7	13.4	0.694
10,000	884	477	19.2	20.7	20.0	0.188
20,000	1256	673	28.4	34.9	30.1	3.407
30,000	1535	818	34.6	39.9	37.5	1.815
40,000	1782	962	43.4	46.4	44.1	0.808
50,000	1976	1053	47.1	52.5	50.2	2.493
60,000	2165	1161	53.3	56.9	55.5	1.338
70,000	2346	1254	59.4	64.4	61.4	2.192
80,000	2528	1356	64.7	68.0	66.3	1.352
90,000	2677	1440	68.3	72.7	70.3	1.773
100,000	2810	1501	72.0	79.0	75.0	5.323

Table 3: *Performance of our implementation for points randomly chosen in a ball. For each value of N, we randomly generated ten point sets of size N. h and k denote the average number of convex hull facets, and the average number of EE-pairs, respectively. In this case, the value of k is slightly larger than $h/2$.* min, max, average, *and* variance *denote the minimum, maximum, and average time in seconds, respectively, and the variance.*

$k \log h$ is large, and this term basically determines the running time in practice.

Next, we generated points from a uniform distribution in the ball centered at the origin and having radius 1000, using LEDA's point generator `random_d3_rat_points-_in_ball`. For each value of $N \in \{10^3, \dots, 10^5\}$, we generated ten point sets of size N, and measured the time after these points were generated. The results are given in Table 3. For this distribution, the expected value of h is bounded by $O(\sqrt{N})$, see Section 4.1 in [9]. In this case, the value of k is slightly larger than $h/2$. Again, the running time in practice is not proportional to N^2, but is determined by the term $k \log h$, which has a large constant.

Finally, we generated random point sets that are close to the sphere centered at the origin and having radius 1000, using LEDA's point generator `random_d3_rat_points-_on_sphere`. For each value of $N \in \{10^2, \dots, 10^4\}$, we generated ten point sets of size N, and measured the time after these points were generated. The results are given in Table 4. For this distribution, (almost) all points are on the convex hull. (Recall that h denotes the number of convex hull facets.) In this case, the value of k is about $2h/3$. By doubling the number of points, the running time in practice increases by a factor that is slightly larger than two.

N	h	k	min	max	average	variance
100	196	122	5.8	6.3	6.0	0.021
500	996	635	34.6	36.2	35.3	0.293
1,000	1,996	1,277	73.8	80.7	76.4	4.715
2,000	3,995	2,538	156.6	162.6	158.2	3.089
3,000	5,992	3,809	239.7	245.2	242.5	3.584
4,000	7,982	5,072	325.0	334.9	330.3	9.343
5,000	9,968	6,315	409.5	424.2	416.0	20.160
6,000	11,962	7,549	496.7	533.6	504.1	121.824
7,000	13,933	8,788	583.4	635.0	596.6	320.541
8,000	15,899	9,997	660.0	696.5	677.7	108.740
9,000	17,882	11,226	740.9	822.4	768.6	443.838
10,000	19,814	12,400	842.2	934.6	864.3	850.981

Table 4: *Performance of our implementation for points randomly chosen close to a sphere. For each value of N, we randomly generated ten point sets of size N. h and k denote the average number of convex hull facets, and the average number of EE-pairs, respectively. In this case, the value of k is about $2h/3$.* min, max, average, *and* variance *denote the minimum, maximum, and average time in seconds, respectively, and the variance.*

Acknowledgements

We thank Stratasys, Inc. for allowing us to test our implementation on their polyhedral models.

References

[1] Pankaj K. Agarwal and Micha Sharir. Efficient randomized algorithms for some geometric optimization problems. *Discrete Comput. Geom.*, 16:317–337, 1996.

[2] U. Bartuschka, K. Mehlhorn, and S. Näher. A robust and efficient implementation of a sweep line algorithm for the straight line segment intersection problem. In *Proc. Workshop on Algorithm Engineering*, pages 124–135, Venice, Italy, 1997.

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

[4] D. P. Dobkin and R. J. Lipton. Multidimensional searching problems. *SIAM J. Comput.*, 5:181–186, 1976.

[5] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, Reading, MA, 1989.

[6] M. E. Houle and G. T. Toussaint. Computing the width of a set. *IEEE Trans. Pattern Anal. Mach. Intell.*, PAMI-10:761–765, 1988.

[7] J. Majhi, R. Janardan, M. Smid, and J. Schwerdt. Multi-criteria geometric optimization problems in layered manufacturing. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 19–28, 1998.

[8] K. Mehlhorn and S. Näher. LEDA: a platform for combinatorial and geometric computing. *Commun. ACM*, 38:96–102, 1995.

[9] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1988.

[10] J. Schwerdt, M. Smid, J. Majhi, and R. Janardan. Computing the width of a three-dimensional point set: an experimental study. In *Proc. 2nd Workshop on Algorithm Engineering*, pages 62–73, Saarbrücken, Germany, 1998.

[11] J. Schwerdt, M. Smid, J. Majhi, and R. Janardan. Computing the width of a three-dimensional point set: an experimental study. Report 18, Department of Computer Science, University of Magdeburg, Magdeburg, Germany, 1998.