An Implementation of a Convex Hull Algorithm Version 1.0

Michael Müller Joachim Ziegler

MPI–I–94–105 February 1994

Acknowledgements

This work was supported in part by the ESPRIT Basic Research Actions Program of the EC under contract No. 7141(ALCOM II) and the BMFT (Förderungskennzeichen ITS 9103).

Abstract

We give an implementation of an incremental construction algorithm for convex hulls in $\mathbb{R}^d$ using *Literate Programming* and *LEDA* in C++. We treat convex hulls in arbitrary dimensions without any non-degeneracy assumption. The main goal of this paper is to demonstrate the benefits of the literate programming approach. We find that the time we spent for the documentation parts is well invested. It leads to a much better understanding of the program and to much better code. Besides being easier to understand and thus being much easier to modify, it is first at all much more likely to be correct. In particular, a literate program takes much less time to debug. The difference between traditional straight forward programming and literate programming is somewhat like the difference between having the idea to a proof of some theorem in mind versus actually writing it down accurately (and thereby often recognizing that the proof is not as easy as one thought).

Keywords

Literate Programming, Convex Hull, WEB, LEDA, Object Oriented Programming, Computational Geometry, Incremental Construction

Contents

1. Introduction.

We give an implementation of an incremental construction algorithm for convex hulls in $\mathbb{R}^d$ using *Literate Programming* (cf. [5]) and *LEDA* (cf. [6, 7]) in C++. The algorithm has been developed by Clarkson, Mehlhorn and Seidel (cf. [4]). In [2], a minor modification of this algorithm is described which maintains convex hulls in arbitrary dimensions without any non-degeneracy assumption.

2. Our main goal was to show how a complex algorithm can be implemented in a way such that everybody can easily understand the program. Therefore, we used *literate programming*. From *LEDA* (a Library of Efficient Data types and Algorithms) we took some useful and well known data structures. The reader not familiar with *LEDA* should not worry about lines of code like
list$\langle$**vector**$\rangle$ L;
because they all have their natural meaning: L is a list of vectors. All *LEDA*-commands are selfexplanatory.

We will first introduce the notation and describe the strategy of the algorithm. To do so, we will essentially cite parts of [4] and [2]. The citations appear in a smaller font and are terminated by a mention of the source (e.g., cited text (cf. reference)).

3. The convex hull is constructed incrementally.

Let $R = \{x_1, \ldots, x_n\}$ be the multi–set of points whose convex hull has to be maintained and let $\pi = x_1 \ldots x_n$ be the insertion order. Let $\pi_i = x_1 \ldots x_i$, $R_i = \{x_1, \ldots, x_i\}$ and let $\text{conv}\, R_i$ be the convex hull of the points in R_i. Let $d = \dim R$ be the dimension of the convex hull of R and let $DJ = \{x_{j_1}, x_{j_2}, \ldots, x_{j_{d+1}}\}$ with $1 \le j_1 \le \ldots \le j_{d+1} \le n$ be the set of dimension jumps where x_k is called a *dimension jump* if $\dim R_{k-1} < \dim R_k$. Clearly, $j_1 = 1$. In the incremental construction of $\text{conv}\, R$ we maintain a triangulation $\Delta(\pi_i)$ of $\text{conv}\, R_i$: a simplicial complex whose union is $\text{conv}\, R_i$ (a simplical complex is a collection of simplices such that the intersection of any two is a face of each[1]). The vertices of the simplices in $\Delta(\pi_i)$ are points in R_i. The triangulation $\Delta(\pi_i)$ induces a triangulation $CH(\pi_i)$ of the boundary of $\text{conv}\, R_i$: it consists of all facets of $\Delta(\pi_i)$ which are incident to only one simplex of $\Delta(\pi_i)$. If $x \in \text{aff}\, R_i$ then a facet F of $CH(\pi_i)$ is called *visible from* x or x*–visible* (we also say: x can see the facet) if x does not lie in the closed halfspace of $\text{aff}\, R_i$ that is supported by F and contains $\text{conv}\, R_i$.

The triangulation $\Delta(\pi_1)$ consists of the single simplex $\{x_1\}$. For $i \ge 2$, the triangulation $\Delta(\pi_i)$ is obtained from $\Delta(\pi_{i-1})$ as follows. If x_i is a dimension jump, i.e., $x_i \notin \text{aff}\, R_{i-1}$, then x_i is added to the vertex set of every simplex of $\Delta(\pi_{i-1})$. If x_i is not a dimension jump then a simplex $S(F \cup \{x_i\}) = \text{conv}(F \cup \{x_i\})$ is added for every x_i–visible facet of $CH(\pi_{i-1})$. Figure 1 gives an example. For a simplex S let $\text{vert}(S)$ denote the set of vertices that define this simplex. It is clear that $\Delta(\pi)$ contains a

[1]Note that the empty set is a facet of every simplex.

simplex whose vertex set is precisely the set of dimension jumps. We call this simplex the *origin simplex* of $\Delta(\pi)$. For every simplex S (besides the origin simplex) we call the vertex in $\text{vert}(S) - DJ$, that has been inserted last, the *peak* of S and the facet of S opposite to the peak the *base facet* of S. (cf. [2], p. 4)

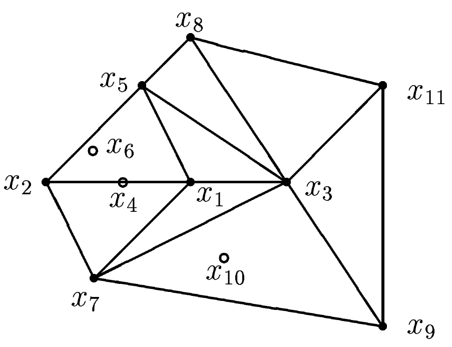

Figure 1: A triangulation. The dimension jumps are the points x_1, x_2, and x_5.

4. It is convenient to extend $\Delta(\pi)$ to a triangulation $\overline{\Delta}(\pi)$ by also making the facets of $CH(\pi)$ the base facet of some simplex: $\overline{\Delta}(\pi)$ is obtained from $\Delta(\pi)$ by adding a simplex $S(F \cup \{\overline{O}\})$ with base facet F and peak $\overline{O}$ for every facet F of $CH(\pi)$. Here $\overline{O}$ is a fictitious point without geometric meaning. We propose to store the triangulation $\overline{\Delta}(\pi)$ as the set of its simplices together with some additional information: For each simplex $S \in \overline{\Delta}(\pi)$ we store its set of vertices, the equation of its base facet normalized such that the peak lies in the positive halfspace, and for each simplex S and vertex $x \in \text{vert}\, S$ we store the other simplex [2]sharing the facet with vertex set $\text{vert}(S) \setminus \{x\}$. We also store a pointer to the origin simplex and a suitable representation of aff R, e.g., a maximal set of affinely independent points. The simplical complex $\overline{\Delta}(\pi_1)$ consists of two simplices: the bounded simplex $S(\{x_1\})$ and the unbounded simplex $S(\{\overline{O}\})$. (A simplex is called *bounded* if $\overline{O}$ does not belong to its vertex set and it is called unbounded otherwise.) (cf. [2], p. 5)

$\overline{O}$ is also called the *anti-origin*. Figure 2 shows the extended triangulation of the example of Figure 1. The points are numbered according to their insertion time. A base facet is indicated by an extra line. You can see that only the origin simplex has no base facet. All outer simplices have $\overline{O}$ as peak. The point x_{12} sees the base facets $\text{conv}(x_2, x_7)$ and $\text{conv}(x_7, x_9)$. The simplex opposite to the vertex x_1 with respect to the simplex $\text{conv}(x_1, x_3, x_5)$ is the simplex $\text{conv}(x_3, x_5, x_8)$. The vertex opposite to x_1 in this simplex is the vertex x_8.

[2]They mean: a pointer to the other simplex

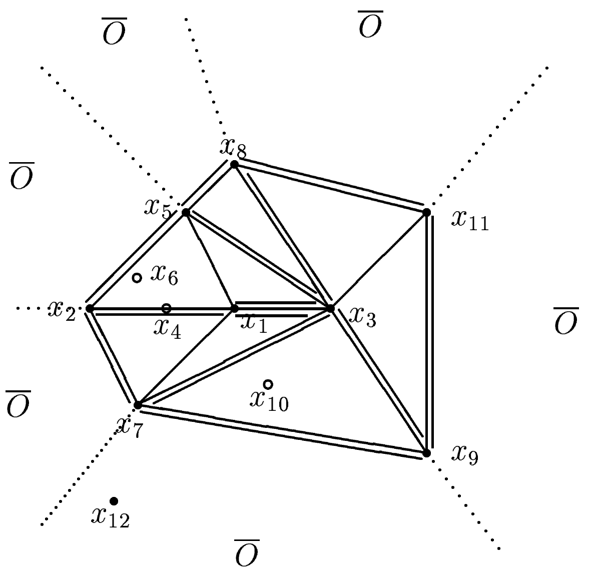

Figure 2: An extended triangulation

5. We give additional details of the insertion process. Consider the addition of the i–th point $x = x_i$, $i \geq 2$. First decide whether x is a dimension jump (an $O(d^3)$ test). If x is a dimension jump then add x_i to vert S for every simplex of $\overline{\Delta}(\pi_{i-1})$ and add the simplex $S(F \cup \{\overline{O}\})$ to $\overline{\Delta}(\pi_i)$ for every bounded simplex F of $\overline{\Delta}(\pi_{i-1})$.

If x_i is not a dimension jump then we proceed as described in [4]. We first compute all x_i–visible facets F of $CH(\pi_{i-1})$ and then update the extended triangulation $\overline{\Delta}$ as follows: For each x_i–visible facet F of $CH(\pi_{i-1})$ ($\equiv$ x_i–visible base facet of an unbounded simplex in $\overline{\Delta}(\pi_{i-1})$) we alter the simplex $S(F \cup \{\overline{O}\})$ of $\overline{\Delta}(\pi_{i-1})$ into $S(F \cup \{x_i\})$. Moreover, for each new hull facet $F \in CH(\pi_i) \setminus CH(\pi_{i-1})$ we add the unbounded simplex $S(F \cup \{\overline{O}\})$. In other words, for each horizon ridge f of $CH(\pi_{i-1})$, i.e., ridge where exactly one of the incident facets is x_i–visible, we add the simplex $S(f \cup \{x_i, \overline{O}\})$. The set of x_i–visible facets F of $CH(\pi_{i-1})$ can be found by visiting simplices according to the rule: Starting at the origin simplex visit any neighbor of a visited simplex that has an x_i–visible base facet. (cf. [2], pp. 5–6)
We call this search method the *visibility search method.*

Another search method is as follows. Walk through the simplices along a segment $\overrightarrow{Ox_i}$ from a point O in the origin simplex to x_i. This leads us to the simplex containing x_i. If this simplex is unbounded, we have found an x_i-visible hull facet from which we can reach all other x_i-visible hull facets. This method is called the *segment walking method* (cf. [4], p. 11).

6. The basic structure of the program.

The fundamental data structures for the simplicial complex will be called **class Simplex** and **class Triangulation**. With these terms, we can now give a short overview of the program.

⟨ Header files to be included 7 ⟩
⟨ class Simplex 15 ⟩
⟨ class Triangulation 11 ⟩
⟨ Member functions of class Triangulation 18 ⟩
⟨ Main program 54 ⟩

7. From *LEDA* we use the data types **array**, **list**, and we use streams for I/O.

⟨ Header files to be included 7 ⟩ ≡
```
#include <LEDA/array.h>
#include <LEDA/list.h>
#include <LEDA/stream.h>
```
See also sections 8 and 10.
This code is used in section 6.

8. In order to show triangulations on the screen, we implement a function that draws the triangulation onto the screen in the two dimensional case using *LEDA*'s **window** type. Therefore, we have to include the appropriate *LEDA* header files. We are working with the X11R5 (xview) window environment.

⟨ Header files to be included 7 ⟩ +≡
```
#include <LEDA/window.h>
#include <LEDA/plane.h>
```

9. We have added to the *LEDA*-type **matrix** a function *linear_solver*( ), which solves a system of linear equations in the following sense: given a **matrix** A and a **vector** b, the function applies the Gaussian elimination algorithm for solving a linear system and returns a list of vectors which characterize the affine-linear space of the solution. When the list returned is empty, there is no solution at all. Otherwise, the list has the form $(a, d_1, \ldots, d_r)$, which means that $d_1, \ldots, d_r$ are the spanning vectors of the affine-space and a is a base point which lies in that space. Upon this function a function *affine_dependency*( ) is based, which determines for a given list of vectors whether they are affine-linearly dependent. Almost all of the calculations needed are carried out in these two functions. They are not yet part of *LEDA*, but soon will be. We give the code of these functions in Section 60.
In the current version of *LEDA*, the entries of a vector and of a matrix are of type **double**, but in later versions the user will be able to choose between several

numerical data types, for example **double** and **rational**. In the algorithm we use a macro **number** defined as **double** or **rational**, but it can be easily changed to any other type providing the operations =,+,−,*,/ and an absolute value function like *fabs*( ). To choose rational arithmetics just define a macro `RATIONAL`. The macro **number** will be defined in `linalg.h` accordingly.

10. We include the appropriate header files depending on the kind of arithmetics we use.

```
⟨ Header files to be included 7 ⟩ +≡
#ifdef RATIONAL
#include "rat_matrix.h"
#else
#include <LEDA/matrix.h>
#endif
#include "linalg.h"      // see Sec. 64
```

11. The fundamental data structures.

Now we can begin to define our fundamental data structures (cf. Section 4, page 2). The whole simplicial complex will be managed by the **class Triangulation**. In this class, we store the coordinate vectors of the points given so far (**list** *coordinates*), the dimension of the convex hull of these points (**int** *dcur*), the dimension of the coordinate vectors of the input points (**int** *dmax*) and a pointer to the origin simplex, from which we can reach all other simplices. also store the coordinates of a point which lies in the interior the origin simplex. When we compute the equation for the base facet of an unbounded simplex, it is useful to know a point which lies in the interior of the origin simplex (cf. Section 17) and we also need such a point as a starting point for the segment walking method. An appropriate point is the center point of the origin simplex

$$p = \sum_{i=0}^{dcur} \frac{v_i}{dcur + 1},$$

where $v_0, \cdots, v_{dcur}$ are the coordinate vectors of the vertices of the origin simplex. To avoid the numerically problematic division, we store in the variable *quasi_center* only the sum of the v_i's and when we need p, we have to remind that $p = quasi\_center/(dcur+1)$. Furthermore, we store a list of all constructed simplices (**list** *all_simplices*) which makes it easier to traverse all simplices (for instance in the destructor of the class or when displaying the simplicial complex). With this list, the interested reader may implement a copy constructor for the class.

During the insertion of some x_i, we have to find the x_i-visible facets of $CH(\pi_{i-1})$. For this purpose, we have implemented three search methods: the visibility search method and the segment walking method described in [4] and a modification of the visibility search method. For the selection of the search method, we introduce an enumeration type with the elements `VISIBILITY`, `MODIFIED_VISIBILITY` and `SEGMENT_WALK`, respectively.

The public member *method* of **Triangulation** determines the search method to be used; it can be changed by the user at any time and its default value is `SEGMENT_WALK`. Each of these search methods stores its result (i.e., pointers to the unbounded simplices having the x_i-visible facets of $CH(\pi_{i-1})$ as base facets) in the list *visible_simplices*.

As the representation of aff R, we use the (affine linearly independent) vertices of the origin simplex.

```
⟨ class Triangulation 11 ⟩ ≡
  enum search_method {
    VISIBILITY, SEGMENT_WALK, MODIFIED_VISIBILITY
  };
  class Triangulation {
  private:
```

```
  list⟨vector⟩ coordinates;    // the coordinate vectors of the x_i
  int dcur;    // dimension of the current convex hull
  int dmax;    // dimension of the coordinate vectors
  Simplex *origin_simplex;    // pointer to the origin simplex
  vector quasi_center;
     // sum of the coordinate vectors of the vertices of the origin simplex
  list⟨Simplex *⟩ all_simplices;    // list of all simplices
  list⟨Simplex *⟩ visible_simplices;    // result of search method
  ⟨ Further member declarations of class Triangulation 17 ⟩
  void print(Simplex *);    // writes some statistics about S to stdout
public:
  search_method method;
  int searched_simplices;    // used for statistical reasons
  int created_simplices( );
     // returns the number of simplices that have been created
  void insert(const vector &x);    // insertion routine
  void show(window &W);
     // draws the triangulation onto the screen
  void print_all( );    // calls print() for all simplices
  Triangulation(int d, search_method m = SEGMENT_WALK);
     // constructor function with default argument
  ~Triangulation( );    // destructor function
};
```

See also sections 12, 13, and 14.

This code is used in section 6.

12. At the end of the program we want to be able to print the number of simplices that have been created. If a simplex is the k-th simplex created, its component *sim_nr* gets k (cf. **Simplex**::**Simplex**( )).

```
⟨ class Triangulation 11 ⟩ +≡
  int Triangulation :: created_simplices( )
  {
    Simplex Dummy(2);
    static dummys_created = 0;
    dummys_created++;
    return Dummy.sim_nr − dummys_created;
  }
```

13. The constructor for **class Triangulation** is easy to implement. The default search method is segment walking.

```
⟨ class Triangulation 11 ⟩ +≡
  Triangulation :: Triangulation(int d, search_method m)
  {
    dcur = −1;
    dmax = d;
    searched_simplices = 0;
    origin_simplex = nil;
    method = m;
  }
```

14. In the destructor for **Triangulation**, we have to release the storage which was allocated for the simplices.

```
⟨ class Triangulation 11 ⟩ +≡
  Triangulation :: ~Triangulation( )
  {
    Simplex *S;
    forall (S, all_simplices) delete (S);
  }
```

15. Now we define the **class Simplex**. We make **class Triangulation** a friend of **class Simplex**, so that it can access every private member of **class Simplex**. For each simplex, we store its vertices as an array *vertices* of pointers to the corresponding occurrences in the list *coordinates* of **class Triangulation**. For the anti-origin we store *nil*. The array has length $dmax + 1$ since a simplex has at most $dmax + 1$ vertices. When the current hull has dimension *dcur*, only the array elements 0 to *dcur* are used. Furthermore, we use the following convention:

the peak vertex of the simplex is always *vertices*[0].

In order to represent the neighborhood relation, we use a second array *neighbors*, such that *neighbors*[k] points to the simplex opposite to the vertex *vertices*[k].

Given a vertex v of a simplex V, let W be the neighbor of V opposite to v. It is also useful to find the vertex w opposite to v, i.e., the vertex w of W which is not a vertex of V. For this purpose, we use an array *opposite_vertices*: if v is the k-th vertex of V, i.e., $V\to vertices[k] \equiv v$, and w is the l-th vertex of W, then $V\to opposite\_vertices[k] \equiv l$ and vice versa $W\to opposite\_vertices[l] \equiv k$.

Figure 3 illustrates the connection between two adjacent simplices V and W. The numbers that stand outside the simplices are the numbers of the vertices of V, the others being the numbers of the vertices of W. In both simplices, the vertex with number 0 is the peak vertex. The connectivity of V and W is expressed as follows: we have $V\to neighbors[2] \equiv W$ and $W\to neighbors[0] \equiv V$, indicated

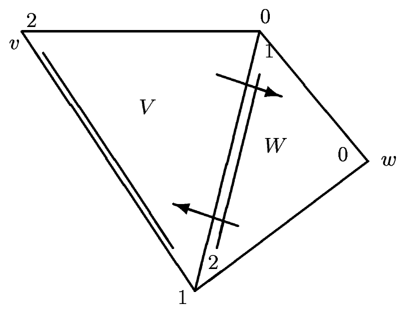

Figure 3: The connection of two simplices V and W

by the corresponding arrows. Furthermore, we have V→*opposite_vertices*[2] ≡ 0 and vice versa W→*opposite_vertices*[0] ≡ 2.

For the test whether a point sees a facet of a given simplex, we need the normal vector and the right side of the equation $normal \cdot x = alpha$ of the hyperplane which contains the facet. The normal vector must be oriented such that the vertex opposite to the face lies in the positive halfspace. When we need *normal* or *alpha* for a facet i of a simplex S (i.e, the facet opposite to the i-th vertex of S), we call the function *normal*(S, i) or *alpha*(S, i), respectively (cf. Section 17), which are members of **class Triangulation**. Once we have computed these values for a facet, we store them in the arrays *normal_values* and *alpha_values* of the corresponding simplex so that they not have to be computed again when they are used the next time. Unfortunately, after a dimension jump all entries of *normal_values* and *alpha_values* become invalid. Therefore, we store in an array *valid_equations* the time, i.e. the current dimension *dcur* of the convex hull, when the values of the facet's equation was computed. The functions *normal*(S, i) and *alpha*(S, i) check whether the respective values of the i-th facet of simplex S are still valid and if not they compute them. Then they return the valid values. The values are invalid iff *valid_equations*$[i]$ < *dcur*. Then the values are computed anew and *valid_equations*$[i]$ is set to *dcur*. Initially, they are set to 0. (In dimension 0 we never need the plane equations).

In the implementation of the deletion process we must not forget that we may have to set *valid_equations*$[i]$ to 0 again for all i and all simplices if we delete a vertex which was a dimension jump.

We also need a mark to indicate visited simplices when we traverse the triangulation (e.g., for the visibility search or for traversing all simplices when we process a dimension jump).

#define *anti_origin* *nil*

⟨ class Simplex 15 ⟩ ≡
  class Simplex {

```
    friend class Triangulation;
        // Triangulation has unrestricted access
  private:
    int sim_nr;     // useful for debugging; unique number for each simplex
#ifdef USE_LEDA_ARRAYS
    array⟨list_item⟩ vertices;
        // pointers to the coordinate vectors of the vertices
    array⟨Simplex *⟩ neighbors;
    array⟨int⟩ opposite_vertices;     // indices of opposite vertices
    array⟨vector⟩ normal_values;     // normal vectors of the facets
    array⟨number⟩ alpha_values;
        // right side of the equation normal · x = alpha
    array⟨int⟩ valid_equations;
        // dimension in which corresponding equation was computed
#else
    list_item *vertices;
        // pointers to the coordinate vectors of the vertices
    Simplex **neighbors;
    int *opposite_vertices;     // indices of opposite vertices
    vector *normal_values;     // normal vectors of the facets
    number *alpha_values;
        // right side of the equation normal · x = alpha
    int *valid_equations;
        // dimension in which corresponding equation was computed
#endif
    bool visited;
        // used to mark simplices when traversing the triangulation
    Simplex(int dmax);     // constructor function
#ifdef USE_LEDA_ARRAYS
    ~Simplex() { } ;     // destructor function
#else
    ~Simplex();     // destructor function
#endif
    LEDA_MEMORY(Simplex);
  };
```

See also section 16.

This code is used in section 6.

16. The constructor for **class Simplex** sets the size of the arrays, allocates storage for the normal vectors and marks the simplex as not visited.

⟨ class Simplex 15 ⟩ +≡

```
#ifdef USE_LEDA_ARRAYS
```

```
Simplex::Simplex(int dmax): vertices(0, dmax), neighbors(0, dmax),
          opposite_vertices(0, dmax), normal_values(0, dmax), alpha_values(0,
          dmax), valid_equations(0, dmax)
  {
    static int lfdnr = 0;
       // each simplex gets a unique number (for debugging)
    sim_nr = lfdnr++;
    for (int i = 0; i <= dmax; i++) {
      normal_values[i] = vector(dmax);      // initialize the normal vectors
      neighbors[i] = nil;      // to avoid illegal pointers when using print()
    }
    visited = false;
  }
#else
  Simplex::Simplex(int dmax)
  {
    static int lfdnr = 0;
       // each simplex gets a unique number (for debugging)
    vertices = new list_item [dmax + 1];
    neighbors = new Simplex * [dmax + 1];
    opposite_vertices = new int [dmax + 1];
    normal_values = new vector [dmax + 1];
    alpha_values = new number [dmax + 1];
    valid_equations = new int [dmax + 1];
    sim_nr = lfdnr++;
    for (int i = 0; i <= dmax; i++) {
      normal_values[i] = vector(dmax);      // initialize the normal vectors
      neighbors[i] = nil;      // to avoid illegal pointers when using print()
      valid_equations[i] = 0;
    }
    visited = false;
  }
  Simplex::~Simplex()      // destructor function
  {
    delete vertices;
    delete neighbors;
    delete opposite_vertices;
    delete normal_values;
    delete alpha_values;
    delete valid_equations;
  }
#endif
```

17. Computing plane equations.

We treat now the functions for computing the values *normal* and *alpha* for the plane equation

$$normal \cdot x = alpha$$

for the facet of a simplex. In Section 15, we mentioned that we call functions *normal*(S, i) and *alpha*(S, i), respectively, when we need one of these values for the i-th facet of the simplex S. These functions check whether the values have already been computed (and if they are still valid) and if this is not the case, they call the function *plane_equation*( ) to compute them.

```
⟨ Further member declarations of class Triangulation 17 ⟩ ≡
  void plane_equation(Simplex *S, int dir);
      // defined as a member of Triangulation
  vector &normal(Simplex *S, int i);
  number alpha(Simplex *S, int i);
```

See also sections 25, 31, 39, and 45.

This code is used in section 11.

18.
```
⟨ Member functions of class Triangulation 18 ⟩ ≡
  vector &Triangulation::normal(Simplex *S, int i)
  {
    if (S→valid_equations[i] ≠ dcur) plane_equation(S, i);
    return S→normal_values[i];
  }
  number Triangulation::alpha(Simplex *S, int i)
  {
    if (S→valid_equations[i] ≠ dcur) plane_equation(S, i);
    return S→alpha_values[i];
  }
```

See also sections 19, 24, 26, 33, 34, 35, 36, 37, 42, 44, 47, 51, 52, and 53.

This code is used in section 6.

19. The parameter *dir* of the function *plane_equation*( ) determines the facet of the simplex S whose equation has to be computed; it is the facet opposite to the *dir*-th vertex of S. The result is stored in S→*normal_values*[*dir*] and S→*alpha_values*[*dir*], respectively. The equation is normalized, such that the vertex *dir* lies in the positive halfspace. We first check whether we have already computed the plane equation for the neighbor of S opposite to the *dir*-th vertex. If so, we can take the values from there and only have to reverse the signs.

```
⟨ Member functions of class Triangulation 18 ⟩ +≡
  void Triangulation::plane_equation(Simplex *S, int dir)
  {
    if (S→neighbors[dir]→valid_equations[S→opposite_vertices[dir]] ≡ dcur) {
          // if the corresponding equation of the neighbor is valid
      S→normal_values[dir] =
          −S→neighbors[dir]→normal_values[S→opposite_vertices[dir]];
      S→alpha_values[dir] =
          −S→neighbors[dir]→alpha_values[S→opposite_vertices[dir]];
    }
    else {
      ⟨ Compute the plane equation 20 ⟩
    }
    S→valid_equations[dir] = dcur;
  }
```

20. The system of equations that leads us to *normal* and *alpha* is a little bit complicated. By a hyperplane, we always mean an affine-linear subspace of the *dmax*-dimensional space which has dimension $dmax - 1$. Let $p_0, \ldots, p_{dcur}$ be the vertices of S. Choose one vertex $p_{base} \neq p_{dir}$ of S (we actually choose $base = 0$ if $dir \neq 0$ and $base = 1$ otherwise). It is clear, that *normal* must be orthogonal to all connecting vectors $p_i - p_{base}$, $0 \le i \le dcur$, $i \neq base$, $i \neq dir$. This gives us $dcur - 1$ equations with *dmax* variables. If $dcur = dmax$ this system of equations determines the vector *normal* (up to a multiplication with a constant factor).

If $dcur < dmax$, we also require that the hyperplane is orthogonal to the affine subspace spanned by our current set of points, i.e., we postulate that *normal* is a linear combination of the direction vectors that span the plane in which the current original simplex lies, that is, of the current space we are working in. If we didn't postulate this, it would be possible that the computed hyperplane contains the vertex with number *dir*. Let $q_0, \ldots, q_{dcur}$ denote the vertices of the origin simplex. Then the above condition is equivalent to:

There are $\lambda_1, \ldots, \lambda_{dcur}$ such that

$$normal + \sum_{i=1}^{dcur} \lambda_i (q_i - q_0) = 0. \qquad (1)$$

Thus, we have another *dmax* equations and another *dcur* variables. The corresponding system of equations looks as follows: (for saving space we abbreviate *dcur* as c, *dmax* as m and *dir* as d).

$$M \cdot \begin{pmatrix} normal_0 \\ \vdots \\ normal_m - 1 \\ \lambda_1 \\ \vdots \\ \lambda_c \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

with

$$M = \begin{pmatrix} p_{1,0} - p_{0,0} & \cdots & p_{1,m-1} - p_{0,m-1} & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & & \vdots \\ p_{d-1,0} - p_{0,0} & \cdots & p_{d-1,m-1} - p_{0,m-1} & 0 & \cdots & 0 \\ p_{d+1,0} - p_{0,0} & \cdots & p_{d+1,m-1} - p_{0,m-1} & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & & \vdots \\ p_{c,0} - p_{0,0} & \cdots & p_{c,m-1} - p_{0,m-1} & 0 & \cdots & 0 \\ 1 & \cdots & 0 & q_{1,0} - q_{0,0} & \cdots & q_{c,0} - q_{0,0} \\ \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 1 & q_{1,m-1} - q_{0,m-1} & \cdots & q_{c,m-1} - q_{0,m-1} \end{pmatrix}.$$

Altogether, we have only $dmax + dcur - 1$ equations but $dmax + dcur$ variables. This means that we will get an affine space of dimension 1 for possible *normal* vectors and *raw_linear_solver*( ) will compute a base point and one direction vector. Since the right side of all equations is 0, the zero vector is a solution of the system (the system is *homogeneous*). Therefore, the direction vector itself lies in the space of solution. Because it can't be the zero vector (then the dimension of the solution space were 0, a contradiction), we can simply take it as a non-trivial solution from which we obtain *normal*.

To compute *alpha*, the right side of the equation, we see that a point x lies in the hyperplane iff $normal \cdot (x - p_{base}) = 0$. This means, that we can take $normal \cdot p_{base}$ as *alpha*. Finally, we must test in which halfspace the vertex *dir* lies. If it does not lie in the positive halfspace defined by *normal* and *alpha*, we simply change the sign of *normal* and *alpha*, then it does. Note that we have to treat here another special case: if the vertex selected by *dir* is the anti-origin, the hyperplane must separate the anti-origin from any point inside the simplicial complex, so we simply test the point $quasi\_center/(dcur + 1)$, which is an interior point of the origin simplex. After this test, normal will point to the outside of the simplicial complex.

All we have to do now is to fill a C-array appropriately with the matrix and the right side, to feed it to *raw_linear_solver*( ), extract *normal* from the solution, compute *alpha* and adjust the sign of *normal*.

⟨ Compute the plane equation 20 ⟩ ≡
  ⟨ Set up the matrix 21 ⟩
  /* now we can call *raw_linear_solver*( ), which returns us a list L characterizing the space of the solution */

```
  list⟨vector⟩ $L$;
  $L = raw\_linear\_solver(M, rows, cols)$;
  ⟨ Extract normal from L 22 ⟩
  ⟨ Compute the right hand side alpha and adjust the sign of normal 23 ⟩
```
This code is used in section 19.

21. To avoid many **new**s and **delete**s we reallocate the memory for the matrix only if the size of the matrix has changed with respect to the last call. If $dcur = dmax$, condition (1) is obviously satisfied and we can omit the corresponding part of the system.

```
⟨ Set up the matrix 21 ⟩ ≡
  static int rows, cols;      // size of the matrix
  static number **M;
     // pointer to memory allocated for matrix and right side of equation
  int i, row, col;      // for stepping through the matrix
  /* we only reallocate memory if the sieze of the matrix has changed */
  if ($rows \neq (dcur \equiv dmax$ ? $dmax - 1 : dmax + dcur - 1) \vee$
        $cols \neq (dcur \equiv dmax$ ? $dmax : dmax + dcur)$) {
    if (M) {      // don't free memory prior to first "re"-allocation
      for ($row = 0$; $row < rows$; $row$++) delete $M[row]$;
      delete M;
    }
    $rows = (dcur \equiv dmax$ ? $dmax - 1 : dmax + dcur - 1)$;
    $cols = (dcur \equiv dmax$ ? $dmax : dmax + dcur)$;
    M = new number * [rows];
    for ($row = 0$; $row < rows$; $row$++) {
      $M[row]$ = new number $[cols + 1]$;      // store right side also in M
      $M[row][cols] = 0$;
    }
  }
  /* Now we set up the equations for: normal is orthogonal to all connecting
  vectors $p_i - p_{base}$, $0 \le i \le dcur$, $i \neq base$, $i \neq dir$. With row, we step through
  the rows $0, \ldots, dcur - 2$ of M. With col, we fill the columns with the vectors
  $p_i - p_{base}$ */
  int $base = (dir \equiv 0$ ? $1 : 0)$;
  vector difference(dcur);      // gets the value of $p_i - p_{base}$
  for ($i = 0, row = 0$; $row < dcur - 1$; $row$++, $i$++) {
    while (($i \equiv dir$) ∨ ($i \equiv base$)) $i$++;
          // do not insert $p_{dir} - p_{base}$ and $p_{base} - p_{base}$
    difference = coordinates.contents(S→vertices[i]) −
        coordinates.contents(S→vertices[base]);      // $p_i - p_{base}$
    for ($col = 0$; $col < dmax$; $col$++) $M[row][col] = difference[col]$;
}
if $(dcur \neq dmax)$ {
      // we need the equations corresponding to condition (1)
   for $(row = 0;\ row < dcur - 1;\ row{+}{+})$ {
      for $(col = dmax;\ col < cols;\ col{+}{+})\ M[row][col] = 0;$
            // zero matrix in the upper right corner
   }
   /* the equations for: $normal$ lies in the affine hull */
   for $(row = dcur - 1;\ row < rows;\ row{+}{+})$
      for $(col = 0;\ col < dmax;\ col{+}{+})\ M[row][col] = (col \equiv row\ ?\ 1 : 0);$
            // unit matrix in the lower left corner
   for $(col = dmax;\ col < cols;\ col{+}{+})$ {
         // fill $M$ column by column with $q_{col-dmax+1} - q_0$
      $difference =$
            $coordinates.contents(origin\_simplex{\rightarrow}vertices[col - dmax + 1]) -$
            $coordinates.contents(origin\_simplex{\rightarrow}vertices[0]);$
      for $(row = dcur - 1;\ row < rows;\ row{+}{+})$
         $M[row][col] = difference[row];$
   }
}
```
This code is used in section 20.

22. We mentioned already that the space of solution of our system $M \cdot x = b$ has dimension 1 and that the base point (i.e., the first element of L) is the zero vector. So we take the second element of L as a non-trivial solution of the system. *normal* consists of the first *dmax* entries of the solving vector of the system.

⟨Extract *normal* from L 22⟩ ≡
```
vector $help$;
   // we will throw away the last $dcur$ components of this vector
/* remove the base point from the list as described above */
$L.pop();$
/* the direction vector is a non-trivial solution */
$help = L.pop();$
/* compute $normal$ from $help$ copying the first $dmax$ entries */
for (int $j = 0;\ j < dmax;\ j{+}{+})\ S{\rightarrow}normal\_values[dir][j] = help[j];$
```
This code is used in section 20.

23. We compute *alpha* and adjust the sign of *normal*.

⟨ Compute the right hand side *alpha* and adjust the sign of *normal* 23 ⟩ ≡

```
S→alpha_values[dir] = S→normal_values[dir] *
      coordinates.contents(S→vertices[base]);
if (S→vertices[dir] ≡ anti_origin) {      // if vertex dir is the anti-origin
   /* in the following expression we have to do an explicit cast since otherwise
   several overloaded operator * ( )-functions would match this use. */
   if (S→normal_values[dir] * quasi_center >
         S→alpha_values[dir] * number(dcur + 1)) {
         // does normal point to the interior of the triangulation?
      S→normal_values[dir] = −S→normal_values[dir];
      S→alpha_values[dir] = −S→alpha_values[dir];
   }
}
else {
   if (S→normal_values[dir] * coordinates.contents(S→vertices[dir]) <
         S→alpha_values[dir]) {
        // does p_dir lie in the negative halfspace ?
      S→normal_values[dir] = −S→normal_values[dir];
      S→alpha_values[dir] = −S→alpha_values[dir];
   }
}
```

This code is used in section 20.

24. The insert procedure.

We treat now the insertion procedure as described in Section 5. For the insertion of a point x, we distinguish three cases:

- x is the first point to be inserted.
- x is a dimension jump (and not the first point to be inserted).
- x is not a dimension jump.

```
⟨Member functions of class Triangulation 18⟩ +≡
  void Triangulation::insert(const vector &x)
  { /* add x to the points already inserted and store its position in item_x */
    list_item item_x = coordinates.append(x);
    if (dcur ≡ −1) {      // x is the first point to be inserted
      ⟨Initialize the triangulation 27⟩
    }
    else if ((dcur < dmax) ∧ is_dimension_jump(x)) {      // see Section 26
      ⟨Dimension jump 46⟩
    }
    else {
      ⟨Non-dimension jump 28⟩
    }
  }
```

25. We need a function *is_dimension_jump*( ), which tells us whether x is a dimension jump or not.

```
⟨Further member declarations of class Triangulation 17⟩ +≡
  bool is_dimension_jump(const vector &x);
```

26. How can we test whether x is a dimension jump? x is a dimension jump iff x does not lie in the affine hull of the vertices of the origin simplex. Since all these vertices are affine-linearly independent by our construction, we only have to test whether x and all these vertices are affine-linearly dependent. We test this by using the function *affine_dependency*( ) (cf. Section 75), which gets as argument a list of all the vectors to test.

```
⟨Member functions of class Triangulation 18⟩ +≡
  bool Triangulation::is_dimension_jump(const vector &x)
  {
    list⟨vector⟩ L;      // the list for affine_dependency( )
    /* we insert x and all vertices of the origin simplex into L */
    $L.push(x)$;
    for (int $i = 0$; $i \le dcur$; $i$++)
      $L.push(coordinates.contents(origin\_simplex \to vertices[i]))$;
    return $(\neg affine\_dependency(L))$;
  }
```

27. When the first point x is inserted, we must initialize our triangulation, that means, we must build the first simplices by hand. This is easy to do. When we only have one point, the simplicial complex consists of two simplices: the origin simplex, containing x as peak, and an outer simplex *outer_simplex* having the anti-origin as its peak. They both point to one another in a natural way. The origin simplex has no base facet by definition, and because *dcur* is 0 *outer_simplex* has a (-1)-dimensional base facet, that means, it has no base facet either. The center point of the origin simplex is clearly x.

⟨Initialize the triangulation 27⟩ ≡

```
  Simplex *outer_simplex;      // a pointer to the outer simplex
  $dcur = 0$;      // we jump from dimension -1 to dimension 0
  all_simplices.append(origin_simplex = new Simplex (dmax));
  all_simplices.append(outer_simplex = new Simplex (dmax));
  origin_simplex→vertices[0] = item_x;      // x is the only point and the peak
  origin_simplex→neighbors[0] = outer_simplex;
  origin_simplex→opposite_vertices[0] = 0;
  outer_simplex→vertices[0] = anti_origin;
  outer_simplex→neighbors[0] = origin_simplex;
  outer_simplex→opposite_vertices[0] = 0;
  quasi_center = x;
```

This code is used in section 24.

28. We discuss now how to handle insertions that are not dimension jumps. We first compute the set of all x-visible hull facets. This is described in detail in Section 31. As a result of this step, we get in *visible_simplices* the list of unbounded simplices whose base facets see x. If there are none, then x lies within the current hull and we are done. Otherwise, we have to modify some simplices and to add some new ones as described in Section 5. Also the neighborhood information has to be updated.

⟨Non-dimension jump 28⟩ ≡

```
  ⟨Find x-visible hull facets 32⟩
  if (¬visible_simplices.empty()) {
    list⟨Simplex *⟩ NewSimplices;
        // Simplices created to store horizon ridges
    Simplex *S;
    forall $(S, visible\_simplices)$ {
      /* For each $x$-visible facet $F$ of $CH(\pi_{i-1})$ alter the simplex
      $S(F \cup \{\overline{O}\})$ of $\overline{\Delta}(\pi_{i-1})$ into $S(F \cup \{x_i\})$. Note that $\overline{O}$ is the peak,
      i.e., $S\text{→}vertices[0]$. */
      $S\text{→}vertices[0] = item\_x$;
      ⟨ For each horizon ridge add the new simplex 29 ⟩
    }
    $visible\_simplices.clear()$;
    ⟨ Update the neighborhood relationship 30 ⟩
  }
```

This code is used in section 24.

29. We now describe, how to update the neighborhood relationship and to compute the equations of the base facets of the new simplices.

At this point, we have found the current hull facets seeing x, in the form of the simplices whose base facets see x and with the anti-origin as their peak vertex. Let $\mathcal{V}$ be the set of such simplices. Now we update T by altering these simplices, and creating some others. The alteration is simply to replace the anti-origin with x in every simplex in $\mathcal{V}$.

The new simplices correspond to new hull facets. Such facets are the hull of x and a horizon ridge f; a *horizon ridge* is a $(d-2)$–dimensional face of conv R with the property that exactly one of the two incident hull facets sees x. Each horizon ridge f gives rise to a new simplex A_f with base facet conv$(f \cup \{x\})$ and peak $\overline{O}$. For each horizon ridge of conv R there is a non-base facet G of a simplex in $\mathcal{V}$ such that x does not see the base facet of the other simplex incident to the facet G. Thus the set of horizon ridges is easily determined. (cf. [4], p. 11)

The figures 4 and 5 illustrate the situation. In figure 4, x sees the facets conv(f, g) and conv (g, h). There are two horizon ridges: the points f and h. The non-base facet G of the above text is the segment s which x does not see. In figure 5, x has been inserted. Two new unbounded simplices corresponding to the two horizon ridges have been added.

We find all horizon ridges incident to an updated simplex S with x-visible base facet by testing all its neighbors (except for the one opposite to its peak) whether their base facet is x-visible. If the base facet of a neighbor is not x-visible, we have found a horizon ridge f and have to create a new simplex T with base facet conv$(f \cup \{x\})$ and peak $\overline{O}$. We collect all new simplices in the list *NewSimplices*.

We use the index k to run through the neighbors of S. When we have identified a horizon ridge, the vertices of the new simplex T are the vertices of S with the k-th vertex replaced by x. The peak of T is the anti-origin $\overline{O}$. We could therefore initialize the vertex set of the new simplex T by

```
  $T\text{→}vertices = S\text{→}vertices$;
  $T\text{→}vertices[k] = item\_x$;
  $T\text{→}vertices[0] = anti\_origin$;
```

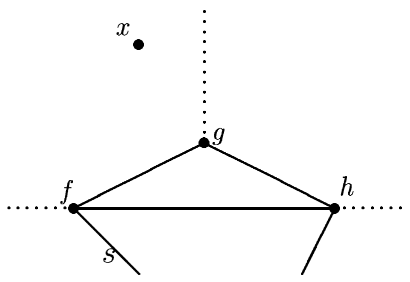

Figure 4: Before x is inserted

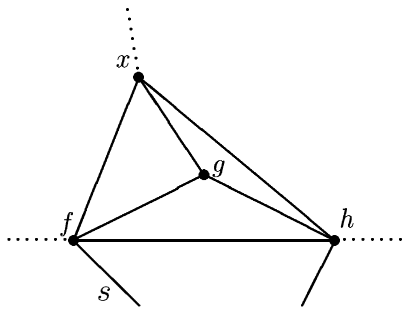

Figure 5: After x has been inserted

In order to facilitate the update of the neighborhood relation, we proceed slightly differently: we make x the highest numbered vertex of T, i.e., we replace the second line by

```
    $T\to vertices[k] = S\to vertices[dcur]$;
    $T\to vertices[dcur] = item\_x$;
```

What are the neighbors of the new simplex T? The neighbor opposite to $\overline{O}$ is S and the neighbor opposite to x is the neighbor of the old S (i.e., S before the replacement of its peak $\overline{O}$ by x) incident to $f \cup \overline{O}$. The neighbors opposite to the j-th vertex of T, with $1 \le j < dcur$, are computed in the next section.

```
⟨For each horizon ridge add the new simplex 29⟩ ≡
  for (int $k = 1$; $k \le dcur$; $k$++) {
    if ($normal(S\to neighbors[k], 0) * x \le alpha(S\to neighbors[k], 0)$) {
          // $x$ doesn't see the base facet of the neighbor
      Simplex $*T$ = new Simplex ($dmax$);
      $all\_simplices.append(T)$;
      $NewSimplices.append(T)$;
      /* Take the vertices of $S$ as the vertices of the new simplex, replacing
      the current vertex by the $dcur$-th, the first by $x$ and the peak by $\overline{O}$ */
      int $ii$;
      for ($ii = 0$; $ii \le dcur$; $ii$++) $T\to vertices[ii] = S\to vertices[ii]$;
      $T\to vertices[k] = S\to vertices[dcur]$;
      $T\to vertices[dcur] = item\_x$;
      $T\to vertices[0] = anti\_origin$;
      /* set the pointers to the two neighbors we already know and update
      the corresponding entries in the $opposite\_vertices$-arrays */
      $T\to neighbors[dcur] = S\to neighbors[k]$;
      $T\to opposite\_vertices[dcur] = S\to opposite\_vertices[k]$;
      $T\to neighbors[0] = S$;
      $T\to opposite\_vertices[0] = k$;
      /* Also set the reverse pointers from those two neighbors to the new
      simplex */
      $S\to neighbors[k]\to neighbors[S\to opposite\_vertices[k]] = T$;
      $S\to neighbors[k]\to opposite\_vertices[S\to opposite\_vertices[k]] = dcur$;
      $S\to neighbors[k] = T$;
      $S\to opposite\_vertices[k] = 0$;
    }
  }
```

This code is used in section 28.

30. We now complete the update of the neighborhood relation. How the neighborhood relationship has to be updated is described in [4] as follows.

It remains to update the neighbor relationship. Let $A_f = S(\text{conv}(f \cup \{x\}), \overline{O})$ be a new simplex corresponding to horizon ridge f. In the old triangulation (before adding x) there were two simplices $\overline{V}$ and N incident to the facet $\text{conv}(f \cup \{\overline{O}\})$; $\overline{V} \in \mathcal{V}$ [3] and $N \notin \mathcal{V}$. In the updated triangulation $\overline{V}$ is replaced by a new simplex V that has the same base but peak x. The neighbor of A_f opposite to x is N and the neighbor opposite to $\overline{O}$ is V. Now consider any vertex $q \in f$ and let $\mathcal{S} = \mathcal{S}_{f,q}$ be the set of simplices with peak x and including $\text{vertex}(f) \setminus \{q\} \cup \{x\}$ in their vertex set; for a face f we use $\text{vertex}(f)$ to denote the set of vertices contained in f. We will show that the neighbor of A_f opposite to q can be determined by a simple walk through $\mathcal{S}$. This walk amounts to a rotation about the $(d-2)$–face $\text{conv}(\text{vertex}(f) \setminus \{q\} \cup \{x\})$. Note first that $V \in \mathcal{S}$. Consider next any simplex $S = S(F, x) \in \mathcal{S}$. Then $F = \text{conv}(f \setminus \{q\} \cup \{y_1, y_2\})$ for some vertices y_1 and y_2. Thus S has at most two neighbors in $\mathcal{S}$, namely the neighbors opposite to y_1 and y_2 respectively. Also, V has at most one neighbor in $\mathcal{S}$, namely the neighbor opposite to q (Note that the neighbor opposite to y, where $\text{conv}(f \cup \{y\})$ is the base facet of V, is the simplex $A_f \notin \mathcal{S}$.). The neighbor relation thus induces a path on the set $\mathcal{S}$ with V being one end of the path. Let V' with base facet $\text{conv}(f \setminus \{q\} \cup \{y_1, y_2\})$ be the other end of the path. Assume that the neighbor of V' opposite to y_1, call it B, does not belong to $\mathcal{S}$ and that $y_1 = q$ if $V = V'$, i.e., the path has length zero. The simplex B includes $\text{vertex}(f) \setminus \{q\} \cup \{y_2, x\}$ in its vertex set and does not have peak x. Thus B has peak $\overline{O}$ and hence B is the neighbor of A_f opposite to q. This completes the description of the update step. (cf. [4], p. 11)

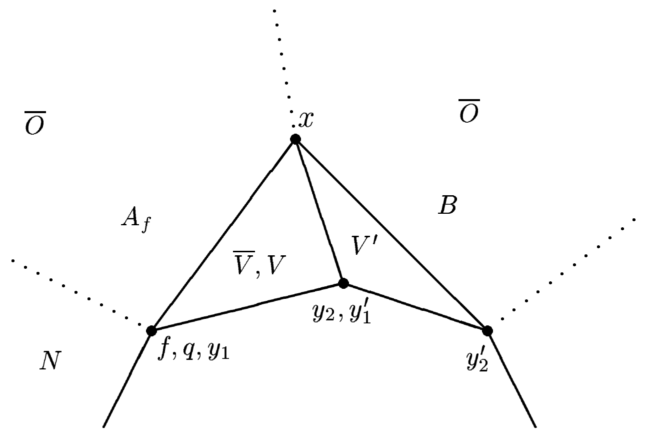

Figure 6: Updating the neighborhood relation

Figure 6 illustrates the situation described above in the two dimensional case. y_1' and y_2' are the new values of y_1 and y_2 after one rotation around x.

[3] $\mathcal{V}$ is the set of outer simplices which see x

This is the only rotation to be made. Then the neighbor of q with respect to A_f is found. It is B.

We implement the update of the neighborhood information as follows. For all new simplices corresponding to horizon ridges, the pointers to the neighbors opposite to x and $\overline{O}$ are already set (cf. the previous section). It remains to do the following for every new simplex A_f corresponding to horizon ridge f:

> For all vertices q of A_f except x and $\overline{O}$ find the neighbor of A_f opposite to q and set the corresponding neighbor pointer.

Note that we do not need to set the pointer from the neighbor we have found to A_f, since the neighbor is also a new simplex and hence this pointer will be (or has been) set anyhow.

Determining the neighbor of A_f opposite to q is done as follows. We walk through the simplices T along the path through $\mathcal{S}$ starting at $T = V = Af\text{→}neighbors[0]$ as described in [4]. As long as $T \in \mathcal{S}$ (i.e., the peak of T is x) we go to the neighbor T' of T opposite to y_1 (for $T = V$ we have $y_1 = q$). The new y_1 is the node of T' equal to the vertex y_2 of T. We store the indices of the vertices corresponding to y_1 and y_2 in two variables *y1* and *y2* respectively. In V, y_2 is the vertex opposite to $\overline{O}$ with respect to A_f. If $T' \notin \mathcal{S}$ (i.e., the peak of T' is not x) we have found the neighbor B of A_f opposite to q.

```
⟨Update the neighborhood relationship 30⟩ ≡
  Simplex *Af;
  forall (Af, NewSimplices) {
    for (int k = 1; k < dcur; k++) {
          // for all vertices q of Af except x and $\overline{O}$ find the opposite neighbor
      Simplex *T = Af→neighbors[0];
      for (int y1 = 0; T→vertices[y1] ≠ Af→vertices[k]; y1 ++) ;
            // exercise: show that we can also start with y1 = 1
      int y2 = Af→opposite_vertices[0];
      while (T→vertices[0] ≡ item_x) {      // while $T \in \mathcal{S}$
        /* find new $y_1$ */
        for (int new_y1 = 0; T→neighbors[y1]→vertices[new_y1] ≠
              T→vertices[y2]; new_y1 ++) ;
            // exercise: show that we can also start with new_y1 = 1
        y2 = T→opposite_vertices[y1];
        T = T→neighbors[y1];
        y1 = new_y1;
      }
      Af→neighbors[k] = T;      // update the neighborhood relationship
      Af→opposite_vertices[k] = y1;      // update the opposite neighbor
    }
  }
```

This code is used in section 28.

31. Finding x-visible hull facets.

For finding the x-visible hull facets, we have implemented three search methods. The first method, the visibility search method, visits all simplices with x-visible base facet using depth first search starting in the origin simplex. It is implemented in the function *visibility_search*( ).

The second method is a modification of the visibility search method. The difference is that if it has once reached an outer simplex, it restricts its search space to unbounded simplices. It uses the function *search_to_outside*( ), which is similar to *visibility_search*( ) except that it stops when it has reached an unbounded simplex. It returns a pointer to the unbounded simplex that it has reached or *nil* if x lies in the interior of the hull. If it has reached an outer simplex, all unbounded x-visible simplices are collected using the function *collect_outer_simplices*( ).

The third method is the segment walking method. This method walks through the simplices which are intersected by a ray $\overrightarrow{Ox}$ from a point O in the origin simplex to x. It returns a pointer to the simplex it has reached (even if this is a bounded simplex). The unbounded x-visible simplices are also collected using the function *collect_outer_simplices*( ).

The visibility search method and the function *collect_outer_simplices*( ) mark visited simplices as visited using the *visited* variable. We unmark them using the function *clear_visited_marks*( ).

```
⟨ Further member declarations of class Triangulation 17 ⟩ +≡
  void visibility_search(Simplex *S, const vector &x);
  Simplex *search_to_outside(Simplex *S, const vector &x);
  Simplex *segment_walk(const vector &x);
  void collect_visible_simplices(Simplex *S, const vector &x);
  void clear_visited_marks(Simplex *S);
```

32. The variable *method* which can be changed interactively by the user switches between the several search methods.

```
⟨ Find x-visible hull facets 32 ⟩ ≡
  Simplex *last_simplex;
     // the simplex in which modified visibility search
     // and segment walking have stopped
  switch (method) {
  case VISIBILITY: visibility_search(origin_simplex, x);
        // generates list of unbounded simplices with x-visible base facet
    clear_visited_marks(origin_simplex);
    break;
  case MODIFIED_VISIBILITY:
    last_simplex = search_to_outside(origin_simplex, x);
  if ($last\_simplex \neq nil$)      // if $x$ is not an interior point
    $collect\_visible\_simplices(last\_simplex, x)$;
        // generates list of unbounded simplices with $x$-visible base facet
  $clear\_visited\_marks(origin\_simplex)$;
  break;
case SEGMENT_WALK:
default: $last\_simplex = segment\_walk(x)$;
  if ($last\_simplex \to vertices[0] \equiv anti\_origin$) {
        // if $x$ is not an interior point
    $collect\_visible\_simplices(last\_simplex, x)$;
        // generates list of unbounded simplices with $x$-visible base facet
    $clear\_visited\_marks(last\_simplex)$;
  }
  break;
}
```

This code is used in section 28.

33. How we can implement *visibility_search*( ) is described in Section 5: starting at the origin simplex, we visit any unvisited neighbor of a visited simplex that has an x-visible base facet. Note that by this rule, we do not have to test the origin simplex (which by definition has indeed no base facet). The **class Triangulation** has a member list *visible_simplices*, in which we store the outer simplices seeing x. The function *visibility_search*( ) is recursive and gets as arguments a reference to the vector x and a pointer $*S$ to the simplex to be visited.

```
⟨Member functions of class Triangulation 18⟩ +≡
void Triangulation::$visibility\_search$(Simplex $*S$, const vector $\&x$)
{
  $searched\_simplices$++;      // only for statistical reasons
  $S \to visited = true$;      // we have visited $S$ and never come back...
  for (int $i = 0$; $i \le dcur$; $i$++) {
    Simplex $*T = S \to neighbors[i]$;      // for all neighbors $T$ of $S$
    if ($\neg T \to visited$) {      // if the $i$-th neighbor has not been visited yet
      if ($normal(T, 0) * x > alpha(T, 0)$) {
            // if $x$ sees the base facet of the $i$-th neighbor
        if ($T \to vertices[0] \equiv anti\_origin$)
              // if the $i$-th neighbor is an outer simplex
          $visible\_simplices.push(T)$;
              // we have found a visible simplex and store it
        $visibility\_search(T, x)$;      // do the recursive search
      }
    }
    }
  }
```

34. Here is the first part of the possibly faster modified visibility search method: search from the origin simplex to the outside, then search on the outer facets recursively with depth first search. If x is an outer point, that means it is contained in one of the outer simplices, the function returns a pointer to the first outer simplex that is found. If x is an inner point, the function returns *nil*. When we say "possibly faster", we have in mind that the searching to the outside (which is nothing but depth first search) will take exactly the same way as the normal visibility search if x is an interior point, so the time we have spent is unfortunately the same. It might only be faster if x lies not in the current hull.

⟨ Member functions of class Triangulation 18 ⟩ +≡

```
  Simplex *Triangulation::search_to_outside(Simplex *S, const vector
          &x)
  {
    searched_simplices++;      // only for statistical reasons
    S→visited = true;      // we have visited S and never come back...
    for (int i = 0; i ≤ dcur; i++) {
      Simplex *T = S→neighbors[i];      // for all neighbors T of S
      if (¬T→visited)      // if the i-th neighbor has not been visited yet
        if (normal(T, 0) * x > alpha(T, 0)) {
              // if x sees the base facet of the i-th neighbor
          if (T→vertices[0] ≡ anti_origin)
                // if the i-th neighbor is an outer simplex
            return T;      // we have found to the outside
          Simplex *result = search_to_outside(T, x);
          if (result ≠ nil) return result;
        }
    }
    return nil;
  }
```

35. Now we collect all outer simplices which are visible from x. The collection process starts from an outer simplex S.

⟨ Member functions of class Triangulation 18 ⟩ +≡

```
  void Triangulation::collect_visible_simplices(Simplex *S, const vector
          &x)
  {
    searched_simplices++;      // only for statistical reasons
  $S$→*visited* = *true*;    // we have visited $S$ and never come back...
  *visible_simplices*.*push*($S$);    // store $S$ as a visible simplex
  **for** (**int** $i = 0$; $i \le dcur$; $i$++) {
    **Simplex** $*T$ = $S$→*neighbors*[$i$];    // for all neighbors $T$ of $S$
    **if** ($\neg T$→*visited* $\wedge$ $T$→*vertices*[0] $\equiv$ *anti_origin*)
          // if the $i$-th neighbor has not been visited yet
          // and is an outer simplex
      **if** (*normal*($T$, 0) $* x >$ *alpha*($T$, 0))
            // if $x$ sees the base facet of the $i$-th neighbor
        *collect_visible_simplices*($T$, $x$);    // do the recursive collecting
  }
}
```

36. After a visibility search, we always must clear the *visited*-bits of the visited simplices. This is done by the recursive function *clear_visited_marks*( ). It is very similar to the function *visibility_search*( ). When we start this function, we also call it with the origin simplex as its argument.

⟨ Member functions of class Triangulation 18 ⟩ +≡

```
void Triangulation :: clear_visited_marks(Simplex *S)
{
  $S$→*visited* = *false*;    // clear the *visited*-bit
  **for** (**int** $i = 0$; $i \le dcur$; $i$++)    // for all neighbors of $S$
    **if** ($S$→*neighbors*[$i$]→*visited*)    // if the $i$-th neighbor has been visited
      *clear_visited_marks*($S$→*neighbors*[$i$]);    // clear its bit recursively
}
```

37. The following function implements the segment walk method to find the simplex containing the point x. Let O denote the "origin", i.e., any point in the origin simplex; we can take *quasi_center*/($dcur + 1$) for O. The strategy is very simple: we start at the origin simplex and walk along the ray $\overrightarrow{Ox}$ through the simplices intersected by this ray until we reach the simplex containing x. There might be degenerate inputs, that means that the ray $\overrightarrow{Ox}$ passes through a vertex of the triangulation or that a segment of $\overrightarrow{Ox}$ lies whithin a facet of a traversed simplex. In order to deal with such degenerate inputs we will perturb O. The details of the perturbation will be described in Section 42. The perturbation scheme is similar to the well-known perturbation method for the simplex algorithm.

Assume now that on our walk we entered a simplex S through the facet *in* (i.e., the facet opposite to the *in*-th vertex of S) and we want to find out through which facet $\overrightarrow{Ox}$ leaves S. The points on the ray $\overrightarrow{Ox}$ satisfy the equation

$$r(\lambda) = O + \lambda(x - O) \quad \text{with} \quad 0 \le \lambda \le 1.$$

The hyperplane which contains the i-th facet of S is given by the equation

$$n^{(i)}p = \alpha^{(i)},$$

where $n^{(i)} = \mathit{normal}(S,i)$ is the normal vector and $\alpha^{(i)} = \mathit{alpha}(S,i)$ is the right side of the plane equation of the i-th facet of S. Thus substituting $r(\lambda)$ for p, we get

$$\lambda^{(i)} = \frac{\alpha^{(i)} - n^{(i)}O}{n^{(i)}(x-O)},$$

and this $\lambda^{(i)}$ yields the intersection of the ray with the hyperplane. We do not need to worry about the case that the denominator in the above expression could be zero, because we never really compute any $\lambda^{(i)}$ (this is due to the perturbation method described in Section 42). Since the vector $x - O$ points from O to x, the values of λ increase along $\overrightarrow{Ox}$. Hence the facet *out* through which we leave S is the facet for which $\lambda^{(out)}$ is the smallest $\lambda^{(i)}$ which is larger than $\lambda^{(in)}$. We really say *larger* and not *larger or equal* (which might be the case in the above equation), because by the perturbation, two different hyperplanes intersect $\overrightarrow{Ox}$ in two different points except if they both contain x, i.e., $\lambda = 1$. Since we only search for *out* if $x \notin S$, we have $\lambda^{(in)} < \lambda^{(out)} < 1$ and thus $\lambda^{(out)} \neq \lambda^{(i)}$ for all $i \neq out$. How we have already mentioned, we do not actually compute the values of the $\lambda^{(i)}$. We only need to know for some i and j whether $\lambda^{(i)} < \lambda^{(j)}$ or not. This decision will be made by a function *lambda_cmp*( ) (cf. Section 42). Only in this function, the perturbation of O plays a role.

The two variables *in* and *out* tell us the number of the facet through which we have entered and through which we will leave the current simplex, respectively. They change, when we walk from simplex to simplex. When we start our walk, there is no facet through which we have entered the current simplex (which is the origin simplex). This is indicated by setting the variable *in* to -1 at the beginning of *segment_walk*( ). In this case, we search for the facet of the origin simplex which is intersected by the "negative" part of $\overrightarrow{Ox}$, i.e., we search the facet whose hyperplane intersects $\overrightarrow{Ox}$ with the largest possible negative value for λ and consider this as the entry facet. This gives us a starting value for *in*. We stop our walk, if we have found the simplex containing x (this might be an unbounded simplex, of course). We use two arrays *nx* and *nO* to store the values of the scalar products $n^{(i)}x$ and $n^{(i)}O$ which we will need several times.

When we have found the simplex S containing x, we stop and return it.

⟨ Member functions of class Triangulation 18 ⟩ +≡

```
Simplex *Triangulation::segment_walk(const vector &x)
{
  Simplex *S = origin_simplex;     // we start at the origin simplex
  bool x_in_S;
      // indicates whether we have found the simplex containing x
  int in = −1;     // entry facet of the origin simplex
```

```
    int $i$;      // for treating every facet of $S$
#ifdef USE_LEDA_ARRAYS
    array⟨number⟩ $nx(0, dcur)$;      // scalar products $n^{(i)}x$
    array⟨number⟩ $nO(0, dcur)$;      // scalar products $n^{(i)}O$
#else
    number $*nx$ = new number $[dcur + 1]$;      // scalar products $n^{(i)}x$
    number $*nO$ = new number $[dcur + 1]$;      // scalar products $n^{(i)}O$
#endif
    while ($true$) {
      $searched\_simplices$++;      // only for statistical reasons
      ⟨ Compute the arrays $nx$ and $nO$ and test whether $x \in S$ 38 ⟩
      if ($x\_in\_S$) {
#ifndef USE_LEDA_ARRAYS
        delete $nx$;
        delete $nO$;
#endif
        return $S$;
      }
      /* We cannot set the next statement in front of this while-loop because
      we need the values of the arrays $nx$ and $nO$ to compute the initial value
      of $in$ */
      if ($in \equiv -1$) {      // if we are still in the origin simplex
        ⟨ Find the facet with largest $\lambda < 0$ 40 ⟩
      }
      ⟨ Go to the next Simplex on the ray $\overrightarrow{Ox}$ 41 ⟩
    }
  }
```

38. The computation of the arrays is easily done by scalar multiplications. At the same time, we can test whether x lies in the current simplex.

```
⟨ Compute the arrays $nx$ and $nO$ and test whether $x \in S$ 38 ⟩ ≡
  $x\_in\_S = true$;      // remains true until we find a facet which doesn't see $x$
  if ($S \rightarrow vertices[0] \neq anti\_origin$)
        // otherwise we have reached the outside
    for ($i = 0$; $i \leq dcur$; $i$++) {
      $nx[i] = normal(S, i) * x$;
      $nO[i] = normal(S, i) * quasi\_center$;      // this is $n^{(i)}O(dcur + 1)$
      if ($nx[i] < alpha(S, i)$) $x\_in\_S = false$;
    }
```

This code is used in section 37.

39. When we start the segment walking method in the origin simplex we have to find the facet whose corresponding hyperplane intersects $\overrightarrow{Ox}$ with the largest $\lambda < 0$. For the comparison of two λ's, we use the function *lambda_cmp*( ) defined in Section 42. This function contains the actual perturbation method. A call *lambda_cmp*$(S, nx[i], nO[i], i, nx[j], nO[j], j)$ returns true iff $\lambda^{(i)} < \lambda^{(j)}$. We also need a function *lambda_negative*( ) defined in Section 44, which decides whether $\lambda^{(i)}$ is negative or not.

```
⟨ Further member declarations of class Triangulation 17 ⟩ +≡
  bool lambda_cmp(Simplex *S, number nix, number niO, int i, number
       njx, number njO, int j);
  bool lambda_negative(Simplex *S, number nx, number nO, int i);
```

40. Now we can search for the "starting facet" in the origin simplex.

```
⟨ Find the facet with largest $\lambda < 0$ 40 ⟩ ≡
  for ($i = 0$; $i \le dcur$; $i$++) {
    if (lambda_negative($S, nx[i], nO[i], i$))      // $\lambda^{(i)} < 0$ ?
      if ($in \equiv -1 \lor$ lambda_cmp($S, nx[in], nO[in], in, nx[i], nO[i], i$))  $in = i$;
  }
```

This code is used in section 37.

41. Now we describe how to find the facet with number *out* through which we will leave the current simplex. We have to find *out* such that for all $i \neq out$ either $\lambda^{(i)} < \lambda^{(in)}$ or $\lambda^{(out)} < \lambda^{(i)}$. Note that we have $\lambda^{(i)} = \lambda^{(j)}$ for $i \neq j$ only if $\lambda^{(i)} = \lambda^{(j)} = 1$ since we perturb O (cf. Section 42). But $\lambda^{(out)} < 1$ always holds since $x \notin S$. When we enter the new simplex we have to set *in* to the index of the facet over which we enter it. The index of a facet is the index of the vertex opposite to it. Hence we set *in* to *S*→*opposite_vertices*[*out*].

```
⟨ Go to the next Simplex on the ray $\overrightarrow{Ox}$ 41 ⟩ ≡
  int out = −1;      // we have not yet found the desired facet
  for ($i = 0$; $i \le dcur$; $i$++) {
    if ($i \equiv in$) continue;      // we surely do not go back on the ray...
    if (lambda_cmp($S, nx[in], nO[in], in, nx[i], nO[i], i$))      // if $\lambda^{(in)} < \lambda^{(i)}$
      if ($out \equiv -1 \lor$ lambda_cmp($S, nx[i], nO[i], i, nx[out], nO[out], out$))
            // if we have not yet a candidate for out
            // or if the current $\lambda^{(i)}$ is better then the old one
          out = i;
  }
  in = S→opposite_vertices[out];
  S = S→neighbors[out];
```

This code is used in section 37.

42. It remains to describe how we decide whether $\lambda^{(i)} < \lambda^{(j)}$. As we have already mentioned we perturb[4] O to get $O^{\mathcal{E}} := O + \mathcal{E}$, where $\mathcal{E} = (\epsilon, \epsilon^2, \ldots, \epsilon^{dcur})$ for some sufficiently small $\epsilon > 0$. Thus, if we consider ϵ small enough, the perturbated ray will not go through any vertex or any other intersection of the hyperplanes of the triangulation. Therefore, the facets which are intersected by this ray are totally linearly ordered. We have $\lambda^{(i)} < \lambda^{(j)}$ iff

$$\frac{\alpha^{(i)} - n^{(i)} O^{\mathcal{E}}}{n^{(i)}(x - O^{\mathcal{E}})} < \frac{\alpha^{(j)} - n^{(j)} O^{\mathcal{E}}}{n^{(j)}(x - O^{\mathcal{E}})} \tag{2}$$

$\Longleftrightarrow$

$$n^{(j)}(x - O^{\mathcal{E}})(\alpha^{(i)} - n^{(i)} O^{\mathcal{E}}) <^{\sigma} n^{(i)}(x - O^{\mathcal{E}})(\alpha^{(j)} - n^{(j)} O^{\mathcal{E}}), \tag{3}$$

where $<^{\sigma}$ is defined to be $<$ if the signs of the two denominators are equal and $>$ otherwise. If $n(x - O) \neq 0$, the sign of $n(x - O^{\mathcal{E}})$ is equal to the sign of $n(x - O)$, because we can choose ϵ sufficiently small. Otherwise, we have

$$n(x - O^{\mathcal{E}}) = n(x - O) - n\mathcal{E} = -n\mathcal{E},$$

and thus the sign of the expression is equal to the inverse sign of n_k, where k is the smallest index for which $n_k \neq 0$ holds (because the term $n_k \epsilon^k$ is dominating).

After a horrendous computation, we find out that (3) is equivalent to

$$\begin{array}{rl} & \alpha^{(i)} n^{(j)} x - \alpha^{(i)} n^{(j)} O - (n^{(j)} x)(n^{(i)} O) - (\alpha^{(i)} n^{(j)} + (n^{(j)} x) n^{(i)})\mathcal{E} \\ <^{\sigma} & \alpha^{(j)} n^{(i)} x - \alpha^{(j)} n^{(i)} O - (n^{(i)} x)(n^{(j)} O) - (\alpha^{(j)} n^{(i)} + (n^{(i)} x) n^{(j)})\mathcal{E} \end{array}$$

If the parts which do not depend on $\mathcal{E}$ are different, they can be used to decide whether $\lambda^{(i)} < \lambda^{(j)}$ (because ϵ can be chosen arbitrarily small, so that the absolute value of the parts depending on $\mathcal{E}$ become smaller than any positive number). Otherwise, we take the minimal k such that

$$\alpha^{(i)} n_k^{(j)} + (n^{(j)} x) n_k^{(i)} \neq \alpha^{(j)} n_k^{(i)} + (n^{(i)} x) n_k^{(j)}$$

and can also make our decision.

The computations for the comparison are carried out in the function *lambda_cmp*( ), which is a member function of **Triangulation** and gets the current simplex S as an input parameter. It also gets the values of the scalar products $n^{(i)} x$, $n^{(i)} O \cdot (dcur + 1)$, $n^{(j)} x$ and $n^{(j)} O \cdot (dcur + 1)$ in the form of the input parameters *nix*, *niO*, *njx* and *njO*, respectively. Furthermore, it gets the indices i and j of the hyperplanes to be compared in order to access *alpha*(S, i), *alpha*(S, j), *normal*(S, i) and *normal*(S, j).

We first have to decide upon $<^{\sigma}$. For this reason, we use a variable *sigma* which is true iff $<^{\sigma}$ is $>$.

[4]This perturbation method was proposed by Kurt Mehlhorn

In the following, we have to do several explicit casts to **number**, since otherwise several overloaded **operator** $*$ ( )-functions would match the corresponding expressions which leads to compiler errors.

We need sevaral times the expression $njx * \mathbf{number}(dcur + 1) - njO$ and $nix * \mathbf{number}(dcur + 1) - niO$, respectively. Tests have shown that in large examples we spent a significiant amount of time evaluating these expressions, so we do it only once in each call of *lambda_cmp*( ) and store its value in the variable *njx_njO* and *nix_niO*, respectively.

```
⟨ Member functions of class Triangulation 18 ⟩ +≡
  bool Triangulation::lambda_cmp(Simplex *S, number nix, number
          niO, int i,
          number njx, number njO, int j)
  {
    bool sigma = false;     // we assume $<^{\sigma}$ to be $<$
    number njx_njO = njx * number(dcur + 1) − njO;
    number nix_niO = nix * number(dcur + 1) − niO;
    ⟨ Decide whether $<^{\sigma}$ is $<$ or $>$ 43 ⟩
    /* Now we test the parts which are not depending on $\mathcal{E}$. Remember that
    $niO = n^{(i)}O(dcur + 1)$, so we have to “adjust” nix and njx */
    number diff = njx_njO * alpha(S, i) − njx * niO
        − (nix_niO * alpha(S, j) − nix * njO);
    if (diff < 0)     // $\lambda^{(i)} <^{\sigma} \lambda^{(j)}$
      return ¬sigma;
    else if (diff > 0) return sigma;
    else {     // the comparison depends on the factor of $\mathcal{E}$
      for (int k = 0; k < dcur; k++) {
        diff = −(alpha(S, i) * normal(S, j)[k] + njx * normal(S, i)[k])
            + (alpha(S, j) * normal(S, i)[k] + nix * normal(S, j)[k]);
        if (diff ≠ 0)     // $\lambda^{(i)} <^{\sigma} \lambda^{(j)}$ if diff < 0
          return (diff < 0 ? true : false);
      }
      /* If the program reaches the next line of code, we have $k \equiv dcur$. This
      means that even the $\mathcal{E}$- factors are equal. In this case, we have $\lambda^{(i)} = \lambda^{(j)}$
      and can return false */
      return false;
    }
  }
```

43. Here we examine the direction of $<^{\sigma}$.

```
⟨ Decide whether $<^{\sigma}$ is $<$ or $>$ 43 ⟩ ≡
  /* We examine the denominators of inequality (2); we first look at the left
  denominator */
```

```
if ($nix\_niO < 0$)      // again, remember that $niO = n^{(i)}O(dcur + 1)$
  $sigma = \neg sigma$;
else if ($nix\_niO \equiv 0$) {
  for (int $k = 0$; ; $k$++)
       // search the smallest $k$ with $normal(S, i)[k] \neq 0$
    if ($normal(S, i)[k] \neq 0$) {
      if ($normal(S, i)[k] > 0$) $sigma = true$;
      else $sigma = false$;
      break;
    }
}
/* and now at the right one */
if ($njx\_njO < 0$)      // again, remember that $njO = n^{(j)}O(dcur + 1)$
  $sigma = \neg sigma$;
else if ($njx\_njO \equiv 0$) {
  for (int $k = 0$; ; $k$++)
       // search the smallest $k$ with $normal(S, j)[k] \neq 0$
    if ($normal(S, j)[k] \neq 0$) {
      if ($normal(S, j)[k] > 0$) $sigma = \neg sigma$;
      break;
    }
}
```
This code is used in section 42.

44. To decide upon the sign of a $\lambda^{(i)}$, we use the function *lambda_negative*( ) which is similar to *lambda_cmp*( ) and returns true iff $\lambda^{(i)} < 0$. Since

$$\lambda^{(i)} = \frac{\alpha^{(i)} - n^{(i)}O}{n^{(i)}(x - O)},$$

we have to check whether the signs of the numerator and the denominator are different. First note that we have always $x \neq O$, because in this case we would have found out that x lies in the origin simplex and left *segment_walk*( ) earlier. Note also that always $\lambda \neq 0$ since $\lambda = 0$ would imply that O lies on a facet of the origin simplex, a contradiction.

If by chance the denominator is zero, the ray $\overrightarrow{Ox}$ intersects facet i in an infinite point. We can return *false*, because such a facet is never a candidate for the intersection with the largest $\lambda < 0$.

⟨ Member functions of class Triangulation 18 ⟩ +≡
```
bool Triangulation::lambda_negative(Simplex *$S$, number $nx$, number
        $nO$, int $i$)
{
  bool $negative = false$;      // we assume that $\lambda$ is positive
```

```
    if $(nx * \textbf{number}(dcur + 1) - nO < 0)$  $negative = true$;
            // if the denominator is less then zero, inverse the sign
    else if $(nx * \textbf{number}(dcur + 1) - nO \equiv 0)$  return $false$;
            // like discussed above
    if $(alpha(S, i) * \textbf{number}(dcur + 1) - nO > 0)$  return $(negative)$;
            // the sign does not change
    else return $(\neg negative)$;        // the sign changes
}
```

45. The Dimension Jump.

If the point being inserted is a dimension jump, we have to add it to the set of vertices of every simplex of the extended triangulation $\overline{\Delta}(\pi_{i-1})$ and for every simplex F of $\Delta(\pi_{i-1})$, we have to add a new simplex $S(F \cup \{\overline{O}\})$ whose base facet is the corresponding simplex of the old triangulation and whose peak is the *anti_origin*. To do so, we visit all simplices of the old triangulation starting at the origin simplex and visiting all neighbors of a visited simplex recursively. This is done by the function *dimension_jump*( ).

⟨ Further member declarations of **class Triangulation** 17 ⟩ +≡
  void *dimension_jump*(**Simplex** $*S$, **list_item** x);

46. Before we do a dimension jump, we compute the new center of the extended origin simplex.

⟨ Dimension jump 46 ⟩ ≡
  dcur++;
  quasi_center += x;
  dimension_jump(*origin_simplex*, *item_x*);
  clear_visited_marks(*origin_simplex*);

This code is used in section 24.

47. In this section we describe the function *dimension_jump*( ). Before we do this, we give an example of a dimension jump in the the two dimensional case. The following figure shows a typical constellation of vertices before a dimension jump.

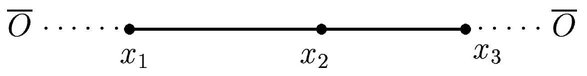

Figure 7: We are in dimension 1

The origin simplex is conv(x_1, x_2). The point x_3 is *not* a dimension jump, because it lies on the line through x_1 and x_2. At this point we have 4 simplices: two unbounded ones to the left and to the right with the anti-origin as their peak, the origin simplex and the simplex conv(x_2, x_3) with peak x_3 and the base facet x_2.

Now we do a dimension jump by inserting a point x_4 not colinear with the other ones.

We have jumped to dimension 2. Now we have 6 simplices. For the simplices conv(x_1, x_2) and conv(x_2, x_3) we have added two unbounded simplices below

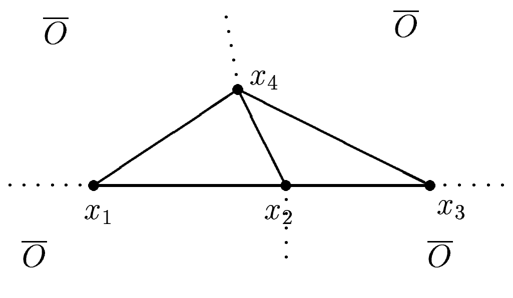

Figure 8: A dimension jump

having $\overline{O}$ as peak. The origin simplex is now conv(x_1, x_2, x_4). The simplex conv(x_2, x_3, x_4) has base facet conv(x_2, x_3) and peak x_4. It is the neighbor of the simplex with the vertices x_2, x_3 and $\overline{O}$ opposite to the vertex $\overline{O}$.

We can divide the simplices of $\overline{\Delta}(\pi_i)$ into three classes:

- *Bounded extended simplices*: they result from bounded simplices of $\overline{\Delta}(\pi_{i-1})$ by adding x to the set of vertices.
- *Unbounded extended simplices*: they result from unbounded simplices of $\overline{\Delta}(\pi_{i-1})$ by adding x to the set of vertices.
- *New simplices*: they result from bounded simplices of $\overline{\Delta}(\pi_{i-1})$ by adding $\overline{O}$ to the set of vertices.

In this and the subsequent sections we will use the following notation. For a simplex F of $\overline{\Delta}(\pi_{i-1})$ let $v_0, \ldots, v_{dcur\,-\,1}$ be its vertices (v_0 is the peak). Let $S = S(F \cup \{x\})$ denote the simplex resulting from extending F. If F is bounded, let *S_new* $= S(F \cup \{\overline{O}\})$ be the new simplex constructed for F. The simplices of $\overline{\Delta}(\pi_i)$ look as follows:

- A (bounded or unbounded) extended simplex S has the vertices $v_0, \ldots,$ $v_{dcur\,-\,1}, x$. Since the peak of a simplex is defined to be the vertex inserted last that was not a dimension jump, the peak of S is the same as the peak of F. Thus we append x to the list of vertices, i.e., we write the appropriate entries at position *dcur* into the arrays *vertices*, *neighbors* and *opposite_vertices*.
- A new simplex *S_new* has the vertices $\overline{O}, v_0, \ldots, v_{dcur\,-\,1}$, where $\overline{O}$ is the peak.

In the following description we will continue to make the distinction between a simplex F of $\overline{\Delta}(\pi_{i-1})$ and the extended simplex S resulting from it. In the implementation, both correspond to the parameter S of *dimension_jump*( ). So every occurrence of F in the following corresponds to S in the program.

dimension_jump( ) works as follows. Starting at the origin simplex it visits all simplices of $\overline{\Delta}(\pi_{i-1})$ using depth-first-search. When a simplex F is visited it is declared visited, x is added to its set of vertices (this turns F into $S = S(F \cup \{x\})$), and if the simplex is bounded then a new unbounded simplex $S\_new = S(F \cup \{\overline{O}\})$ is created. Then all neighbors of F in $\overline{\Delta}(\pi_{i-1})$ are visited recursively. (Note that only the neighbors *F→neighbors*[0], ..., *F→neighbors*[*dcur* − 1] are inspected). Once all neighbors are visited we update the neighbor relation. There we distinguish cases according to whether the simplex is bounded or not.

```
⟨ Member functions of class Triangulation 18 ⟩ +≡
  void Triangulation :: dimension_jump(Simplex *S, list_item x)
  {
    Simplex *S_new;
    S→visited = true;
    S→vertices[dcur] = x;
    if (S→vertices[0] ≠ anti_origin) {      // S is bounded iff peak ≠ Ō
      ⟨ Add a new unbounded simplex 48 ⟩
    }
    /* The neighbor opposite to x might not yet exist. We make a call of
    dimension_jump( ) for all unvisited neighbors of S. */
    for (int k = 0; k ≤ dcur − 1; k++) {      // for all neighbors of F
      if (¬S→neighbors[k]→visited) dimension_jump(S→neighbors[k], x);
    }
    if (S→vertices[0] ≡ anti_origin) {
      ⟨ Complete neighborhood information if F is unbounded 49 ⟩
    }
    else {
      ⟨ Complete neighborhood information if F is bounded 50 ⟩
    }
  }
```

48. For every bounded simplex F of $\overline{\Delta}(\pi_{i-1})$ we add a new simplex $S\_new = S(F \cup \{\overline{O}\})$ with peak $\overline{O}$. It is the neighbor of the bounded extended simplex $S = S(F \cup \{x\})$ opposite to x, and $\overline{O}$ is the vertex opposite to x. For all vertices v of S different from x the neighbor of S opposite to v is the simplex $S(F' \cup \{x\})$ where F' is the neighbor of F opposite to v. Thus no action is required in the algorithm.

```
⟨ Add a new unbounded simplex 48 ⟩ ≡
  S_new = S→neighbors[dcur] = new Simplex (dmax);
  all_simplices.append(S_new);
  S→opposite_vertices[dcur] = 0;
```

```
    $S\_new \to vertices[0] = anti\_origin;$
    for (int $k = 1$; $k \le dcur$; $k$++) $S\_new \to vertices[k] = S \to vertices[k-1];$
```
This code is used in section 47.

49. We discuss how to compute the neighbors of unbounded extended simplices. The neighbor of an unbounded extended simplex $S = S(F \cup \{x\})$ opposite to x is the simplex T with $\text{vert}(F) \subset \text{vert}(T)$ and $x \notin \text{vert}(T)$. Consider the neighbor $F' \in \overline{\Delta}(\pi_{i-1})$ of F opposite to $\overline{O}$. F' is bounded. Hence we constructed a simplex S_new' with $\text{vert}(S\_new') = \text{vert}(F') \cup \{\overline{O}\}$. Since $\text{vert}(F) \setminus \{\overline{O}\} \subset \text{vert}(F')$ we have $\text{vert}(F) \subset \text{vert}(S\_new')$. Furthermore $x \notin \text{vert}(S\_new')$. Thus $T = S\_new'$ is the neighbor of S opposite to x. We reach T from F (or S, respectively) by first going to the 0-th neighbor (that is F' or S', respectively) and then going to the *dcur*-th neighbor of S' which is $S\_new' = S(F' \cup \{x\}) = T$. The vertex opposite to x with respect to S is the vertex w opposite to $\overline{O}$ with respect to F. Note that if w is the i-th vertex of F' then it is the $(i+1)$-st vertex of S_new' since we have inserted the anti-origin in *vertices*[0].

As in the previous section (bounded extended simplex), the neighborhood information for vertices $v \neq x$ of S is the same as for F and hence there is nothing to do for them.

⟨ Complete neighborhood information if F is unbounded 49 ⟩ ≡
```
    $S \to neighbors[dcur] = S \to neighbors[0] \to neighbors[dcur];$
    $S \to opposite\_vertices[dcur] = S \to opposite\_vertices[0] + 1;$
```
This code is used in section 47.

50. Let F be a bounded simplex of $\overline{\Delta}(\pi_{i-1})$. It gives rise to the extended simplex $S = S(F \cup \{x\})$ and the new simplex $S\_new = S(F \cup \{\overline{O}\})$. The neighbors of S were already computed in Section 48. We still need to determine the neighbors of S_new. In order to create the neighborhood information for a new simplex S_new, we step through the neighbors of F.

To find the neighbor of S_new opposite to $v \neq \overline{O}$ consider the neighbor $F' \in \overline{\Delta}(\pi_{i-1})$ of F opposite to v. If F' is unbounded, the neighbor of S_new opposite to v is S' and the vertex opposite to v is x. If F' is bounded, the neighbor of S_new opposite to v is the simplex S_new' constructed for F' and the vertex opposite to v remains the same as in F. Note that a pointer to S_new' has been added to the *neighbors* array of F' at position *dcur* during a recursive or a previous call of *dimension_jump*( ).

The neighbor of a new simplex S_new opposite to $\overline{O}$ is S. The vertex opposite to $\overline{O}$ is x. Recall that the k-th vertex of S is the $k+1$-st vertex of S'.

```
⟨ Complete neighborhood information if $F$ is bounded 50 ⟩ ≡
  for (int $k = 0$; $k < dcur$; $k$++) {
    if ($S$→$neighbors[k]$→$vertices[0] \equiv anti\_origin$) {    // if $F'$ is unbounded
      $S\_new$→$neighbors[k+1] = S$→$neighbors[k]$;
          // the neighbor of $S\_new$ opposite to $v$ is $S'$
      $S\_new$→$opposite\_vertices[k+1] = dcur$;    // $x$ stands in position $dcur$
    }
    else {    // $F'$ is bounded
      $S\_new$→$neighbors[k+1] = S$→$neighbors[k]$→$neighbors[dcur]$;
          // neighbor of $S\_new$ opposite to $v$ is $S\_new'$
      $S\_new$→$opposite\_vertices[k+1] = S$→$opposite\_vertices[k] + 1$;
          // ... vertex opposite to $v$ remains the same ...
          // again remember the 'shifting' of the vertices one step to the right
    }
  }
  /* the simplex opposite to $\overline{O}$ with respect to $S\_new$ is $S$, and the vertex is $x$
  */
  $S\_new$→$neighbors[0] = S$;
  $S\_new$→$opposite\_vertices[0] = dcur$;
```

This code is used in section 47.

51. Output Routines.

In order to demonstrate our program, we now add to **Triangulation** a function *show*( ), which draws (in the special case $dmax \equiv 2$) the simplicial complex into a *LEDA*-**window**. Running through the list *all_simplices* we draw each simplex. For each simplex, we draw its vertices and for each vertex of a simplex we draw the edges connecting it to the other vertices of the simplex. Clearly we do not draw the anti-origin and the edges incident to it. Thus the **for**-loop which steps through all vertices starts with $v = 0$ if S is bounded (i.e., $S\rightarrow vertices[0] \neq anti\_origin$) and with $v = 1$ if S is unbounded (i.e., $S\rightarrow vertices[0] \equiv anti\_origin$). Furthermore, we draw every point that we have inserted so far onto the screen (there may be many points that are not vertices of any simplex). We do this by running through the list *coordinates*.

```
⟨Member functions of class Triangulation 18⟩ +≡
  void Triangulation::show(window &W)
  { /* We first draw every simplex */
    Simplex *S;
    forall (S, all_simplices) {
      for (int v = (S→vertices[0] ≡ anti_origin ? 1 : 0); v ≤ dcur; v++) {
            // for each vertex except the anti-origin
        vector x = coordinates.contents(S→vertices[v]);
        point a(x[0], x[1]);
        for (int e = v + 1; e ≤ dcur; e++) {
              // draw undrawn edges incident to vertex
          vector y = coordinates.contents(S→vertices[e]);
          point b(y[0], y[1]);
          /* draw the edges of unbounded simplices as thick lines */
          if (S→vertices[0] ≡ anti_origin) W.set_line_width(3);
          else W.set_line_width(1);
          W.draw_segment(a, b);
        }
      }
    }
    /* Now we draw every point */
    vector x;
    forall (x, coordinates) {
      point a(x[0], x[1]);
      W.draw_point(a);
    }
  }
```

52. *print_all*( ) prints information about all simplices to *stdout* (similar to show). This was useful for debugging. The information of a single simplex is printed by the function *print*( ).

```
⟨ Member functions of class Triangulation 18 ⟩ +≡
  void Triangulation::print_all( )
  {
    Simplex *S;
    forall (S, all_simplices) print(S);
  }
```

53. Here is a short function that prints the data of a simplex.

```
⟨ Member functions of class Triangulation 18 ⟩ +≡
  void Triangulation::print(Simplex *S)
  {
    cout ≪ "\n[" ≪ S→sim_nr ≪
        "]--------------------------------------------------\n";
    for (int i = 0; i ≤ dcur; i++) {
      if (S→vertices[i] ≡ anti_origin) cout ≪ "anti";
      else cout ≪ coordinates.contents(S→vertices[i]);
      cout ≪ "␣[";
      if (S→neighbors[i]) cout ≪ S→neighbors[i]→sim_nr;
      else cout ≪ "*";
      cout ≪ "]␣";
      if ((S→vertices[0] ≠ anti_origin ∨ i ≡ 0) ∧ dcur > 0) {
            // cout ≪ ";␣normal:␣" ≪ normal(S, i);
        cout ≪ ";␣normal:␣" ≪ S→normal_values[i];
            // cout ≪ ";␣␣alpha:␣" ≪ alpha(S, i);
        cout ≪ ";␣␣alpha:␣" ≪ S→alpha_values[i];
        cout ≪ ";␣valid_equations:␣" ≪ S→valid_equations[i];
      }
      cout ≪ endl;
    }
    cout.flush( );
  }
```

54. The main function.

There are three ways to feed the data into the program: we can take the input from the keyboard, from a file or via mouse input from a graphics window (only if we work in dimension 2). If the input is taken from the keyboard or from a file, the first number must be an integer specifying the dimension of the following coordinate vectors. If the input is taken from a file, the second number in the file is read but ignored by our program (in order to be able to use input files that are created by the program `rbox` which generates random input files; it is a tool of the `QHULL`–system (cf. [1])). The remaining numbers in the file are taken as the coordinates of the points. We can call the program from a shell with the following command line arguments in an arbitrary order:

- **m**: read input from mouse.
- **k**: read input from keys, first entering the dimension we will work in, then the coordinates of the points. The input process stops with an end-of-file (`ctrl-D`).
- **f**: read input from a file whose name must be given as the next argument in the command line.
- **p**: print informations about all simplices after each insertion.
- **n**: no display: when working in dimension 2 only draw the final result.
- **s**: suppress any display when working in dimension 2
- **V**: use the visibility search method.
- **M**: use the modified visibility search method.
- **S**: use the segment walking method.

We first give a function that tells the user the correct usage of the command line arguments of the program when he makes a mistake when invoking the program.

In the command line, the user can give any number of the above arguments, but only the last ones are valid.

⟨ Main program 54 ⟩ ≡
```
  void tell_usage(string prg_name)
     // prg_name is the name of the executable program
  {
    cout ≪ "Usage:␣" ≪ prg_name ≪
         "␣[␣m␣|␣k␣|␣f␣filename␣|␣p␣|␣n␣|␣s␣|␣V␣|␣M␣|␣S]*" ≪ endl;
       // a regular expression
    exit(1);
  }
```
See also section 55.
This code is used in section 6.

55. The main program first reads in the command line setting the options, then it processes the data.

```
⟨Main program 54⟩ +≡
  enum input_method {
    MOUSE, KEYS, INPUTFILE
  };
  main(int argc, char **argv)
  {
    ⟨Read the command line 56⟩
    ⟨Process the data 57⟩;
  }
```

56. In the command line, every option consists of a single character.

```
⟨Read the command line 56⟩ ≡
  string_istream args(argc, argv);
      // create an input stream from the command line
  string prg_name;    // the name of the compiled, executable program
  args ≫ prg_name;    // get the name from the command line
  string option;    // the options we will take from the command line
  string data_file = "/dev/null";
      // the name of the file that contains the data;
  /* data_file is initialized to "/dev/null" to avoid complicated special treat-
  ment when no input file is specified */
  int dimension;    // the dimension we will work in
  int number_of_points;
      // appears in input files generated by rbox, not used by our program
  input_method read_from = MOUSE;    // default: read from mouse
  search_method m = SEGMENT_WALK;    // default: segment walking
  bool draw_all = true;    // draw every insert step (if dimension ≡ 2)
  bool suppress = false;    // suppress any display (if dimension ≡ 2)
  bool print_simplices = false;
      // print information about all simplices after an insert
  while (true) {
    args ≫ option;
    if (args.eof()) break;
        // as long as we have command line arguments
    if (option.length() ≠ 1)
         // if the current argument has more than one character
      tell_usage(prg_name);    // tell the correct usage of the program
    switch (option[0]) {    // which option is to be processed?
    case 'm': read_from = MOUSE;
      break;
    case 'k': read_from = KEYS;
      break;
    case 'f':
      /* this argument must be followed by another argument which is taken
      as the name of a file from which we read the data */
      args ≫ data_file;    // get the filename
      if (args.eof())    // print error message if no file is specified
        tell_usage(prg_name);
      read_from = INPUTFILE;    // we read from a file
      break;
    case 'p': print_simplices = true;
      break;
    case 'n': draw_all = false;
      break;
    case 's': suppress = true;
      break;
    case 'V': m = VISIBILITY;
      break;
    case 'M': m = MODIFIED_VISIBILITY;
      break;
    case 'S': m = SEGMENT_WALK;
      break;
    default: tell_usage(prg_name);
      break;
    }
  }
```

This code is used in section 55.

57. Here is how we process the data.

⟨ Process the data 57 ⟩ ≡

```
  /* if the input is not taken from the mouse, we need a file from which we
  read the data */
  file_istream file_in(data_file);    // file_istream is a LEDA type
  if (¬file_in) {
    cout ≪ "unable␣to␣open␣file␣" ≪ data_file ≪ endl;
    exit(2);
  }
  switch (read_from) {
  case MOUSE:
    {
      ⟨ Input from mouse 58 ⟩
    }
    exit(0);
    break;
  case KEYS: cout ≪ "Dimension␣of␣coordinate␣vectors:␣";
    cin ≫ dimension;
    break;
  case INPUTFILE: file_in ≫ dimension;
    file_in ≫ number_of_points;      // we do not use this value
    break;
  }
  ⟨ Input from keyboard or file 59 ⟩
```

This code is used in section 55.

58. We use LEDA's **window** type to implement a graphical input tool. We are working with the X11R5 (xview) window system.

By a click of the left mouse button, we can input a new two dimensional point into the whole complex. Then the triangulation will be drawn onto the screen. The convex hull is represented by thick lines, whereas the other lines of the triangulation are drawn as thin lines. A click of the right mouse button ends the program. The input points are automatically logged to the file `chull.pts`.

```
⟨ Input from mouse 58 ⟩ ≡
  window W;
  Triangulation T(2, m);
     // we are working in the plane with search method m
  double a, b;      // coordinates of a point in the window
  file_ostream protocol("chull.pts");
  int mouse = 0;      // variable to indicate which mouse button was pressed
  protocol ≪ 2 ≪ endl;      // write the dimension to chull.pts
  while (mouse ≠ 3) {      // while mouse click is not the right button
    mouse = W.read_mouse(a, b);
       // read the window coordinates into a and b
    if (mouse ≡ 1) {      // left button pressed
      vector x(2);      // get a two dimensional vector
      x[0] = a;
      x[1] = b;
      protocol ≪ x ≪ endl;
      T.insert(x);
    W.clear();
    T.show(W);
    if (print_simplices) T.print_all();
  }
}
cout ≪ endl ≪ "Searched␣Simplices:␣" ≪ T.searched_simplices ≪ endl;
   // only for statistical reasons
```

This code is used in section 57.

59. If we take the input from the keyboard or from a file, we read for each point its *dimension* coordinates and insert it. If *dimension* ≡ 2 we also open a graphics window in order to display the triangulation. The window remains on the screen until the right mouse button is pressed in it. Keyboard input is terminated by an end-of-file (`ctrl-D`).

⟨Input from keyboard or file 59⟩ ≡

```
Triangulation T(dimension, m);
vector x(dimension);
if (dimension ≡ 2 ∧ ¬suppress) {
  window W;
  while (¬(file_in.eof() ∨ cin.eof())) {
    if (read_from ≡ KEYS) cin ≫ x;
    else file_in ≫ x;
    T.insert(x);
    if (¬suppress ∧ draw_all) {
      W.clear();
      T.show(W);
    }
    if (print_simplices) T.print_all();
  }
  W.clear();
  T.show(W);
  cout ≪ "Press␣the␣right␣button␣in␣the␣drawing␣w\
      indow␣to␣terminate.\n";
  cout.flush();
  while (W.read_mouse() ≠ 3) ;
}
else {
  while (¬(file_in.eof() ∨ cin.eof())) {
    if (read_from ≡ KEYS) cin ≫ x;
    else file_in ≫ x;
    T.insert(x);
    if (print_simplices) T.print_all();
    }
  }
  cout ≪ endl ≪ "Searched␣Simplices:␣" ≪ T.searched_simplices ≪ endl;
  cout ≪ "Simplices␣created:␣" ≪ T.created_simplices( ) ≪ endl;
      // only for statistical reasons
```

This code is used in section 57.

60. Solving a system of linear equations.

In this part, we add to the LEDA-type **matrix** a function which solves a system of linear equations in the following sense: given a **matrix** A and a **vector** b, the function applies the Gaussian elimination algorithm for solving a linear system and returns a list of vectors which characterize the affine-linear space of the solution. When the list returned is empty, there is no solution at all. Otherwise, the list has the form $(a, d_1, \ldots, d_r)$, which means that $d_1, \ldots, d_r$ are the spanning vectors of the affine-space and a is a base point which lies in that space.

In the subsequent part we give two applications of the algorithm which we need: determine whether a set of vectors is linearly dependent or affine-linearly dependent.

61. The program has the following structure:

```
⟨ linalg.c 61 ⟩ ≡
  ⟨ Header files for linalg 63 ⟩
  ⟨ The function 65 ⟩
  ⟨ Dependency tests 74 ⟩
```

62. In the actual version of LEDA, the entries of a matrix are of type **double**, but in later versions the user will be able to choose between several kinds of arithmetical operation, for example rational arithmetics. Therefore, we define here a macro **number** and use it in the algorithm defined as **double** or **rational**, but it can be easily changed to any other type providing the operations =,+,−,*,/ and an absolute value function like *fabs*( ) for C++'s **double** type. (If you want to compute in a prime field for example, you can forget about the *fabs*( ) function. Alternatively to *fabs*( ) you can also use the <-operator.) The only lines of code that must be changed are the ones in which we update the pivot-element (see below) and test it against zero. Because of the low stability of C++'s type **double**, we never test a **double**-variable to be zero. We concern it as zero, if its absolute value is less than a constant `EPSILON`.

We have also implemented some routines which compute with rational arithmetics and thus guarantee to be exact in every case. To choose this kind of arithmetics one simply has to define the `RATIONAL` macro. The macro definitions depend on whether we compute with rational arithmetics or not.

```
⟨ Macro definitions 62 ⟩ ≡
#ifdef RATIONAL
#define number rational
#define vector rat_vector
#define matrix rat_matrix
#define EPSILON 0
```

#define number_abs *abs*
    // for **rational**, the function is called "*abs*( )"
#else
#define number double
#define EPSILON $1 \cdot 10^{-10}$
#ifdef USE_FABS
#define number_abs *fabs*
#else
#define number_abs(a) $((a) < 0\ ?\ -(a) : (a))$
#endif
#endif
This code is used in section 64.

63. We include the following LEDA header files, depending on the kind of arithmetics we use:

⟨ Header files for linalg 63 ⟩ ≡
#include `<LEDA/list.h>`
#include `<LEDA/array.h>`
#include `<stream.h>`
#include `<math.h>`
#ifdef RATIONAL
#include `"rat_matrix.h"`
#else
#include `<LEDA/matrix.h>`
#endif
#include `"linalg.h"`
This code is used in section 61.

64. We create the following header file for the functions we will write. The main work of *linear_solver*( ) is done in the function *raw_linear_solver*( ) which gets a pointer C to a 2-dimensional C-array in which the calculation is done Furthermore, it gets as parameters *rows* and *cols* the dimensions of the matrix A of the underlaying system of equations. (I.e., since C also contains the right side b, C is an array of dimension $rows \times (cols + 1)$.). We can save some LEDA overhead when we call *raw_linear_solver*( ) directly, avoiding the initialization of the LEDA-**matrix** A. We make use of this in *plane_equation*( ).

⟨ `linalg.h` 64 ⟩ ≡
  ⟨ Macro definitions 62 ⟩
  list⟨**vector**⟩ *linear_solver*(**matrix** &A, **const vector** &b);
  list⟨**vector**⟩ *raw_linear_solver*(**number** **C, **int** *rows*, **int** *cols*);
  bool *linear_dependency*(**list**⟨**vector**⟩ L);
  bool *affine_dependency*(**list**⟨**vector**⟩ L);

65. In the function *linear_solver*( ) we only have to initialize the matrix in which *raw_linear_solver*( ) calculates and call *raw_linear_solver*( ). We only want to allocate new memory for C if the dimensions have changed with respect to the last call. Thus we make C, *rows* and *cols* static.

```
⟨ The function 65 ⟩ ≡
  list⟨vector⟩ linear_solver(matrix &A, const vector &b)
  {
    static number **C;
        // the matrix in which we will calculate (C = (A|b))
    static int rows;     // the number of rows of A
    static int cols;     // the number of columns of A
    int i, j;
    /* reallocate memory for C if necessary */
    if (rows ≠ A.dim1( ) ∨ cols ≠ A.dim2( )) {
      if (C) {
        for (i = 0; i < rows; i++) delete C[i];     // delete each row
        delete C;
      }
      rows = A.dim1( );
      cols = A.dim2( );
      C = new number * [rows];
      for (i = 0; i < rows; i++) C[i] = new number [cols + 1];
    }
    /* copy A and b into C */
    for (i = 0; i < rows; i++) {
      for (j = 0; j < cols; j++) C[i][j] = A(i, j);
      C[i][cols] = b[i];
    }
    return raw_linear_solver(C, rows, cols);
  }
```

See also section 66.

This code is used in section 61.

66. We can now give an overview of *raw_linear_solver*( ).

```
⟨ The function 65 ⟩ +≡
  list⟨vector⟩ raw_linear_solver(number **C, int rows, int cols)
  {
    list⟨vector⟩ L;     // the list that is returned
    int i, j, k;     // indices to step through the matrix
    ⟨ Make upper diagonal form 67 ⟩
    ⟨ Make unity matrix on the left side 71 ⟩
    ⟨ Compute the list L which contains the solution 72 ⟩
  ⟨ Deallocate memory and return the list L 73 ⟩
}
```

67. This part of the function changes C into upper diagonal form. Here is the description of the first step. We choose the pivot-element p (the element $C_{i,j}$ with which we make appropriate columns to zero) the element of highest absolute value in C. Note that the last column of C is b and that we don't search for p in this column. Note also that this rule guarantees maximal arithmetical stability. Having found p in row i and column j, we interchange row i with row 1 and column j with column 1, so that p stands now in $C_{0,0}$. In the **array** *var*, we store the indices of the variables for every column. Then we use the pivot-element p to set $C_{0,1}, C_{0,2}, \ldots = 0$. The second step is like the first one, but it only works on the submatrix starting at $C_{1,1}$. If in this process p is found zero, we have upper diagonal form.

```
⟨ Make upper diagonal form 67 ⟩ ≡
#ifdef USE_LEDA_ARRAYS
  array⟨int⟩ var(0, cols − 1);
#else
  int *var = new int [cols];
#endif
  number p;
  for (i = 0; i < cols; i++) var[i] = i;
      // at the beginning, variable x_i stands in column i
  /* here comes the main loop */
  for (i = 0; i < cols; i++)
  {
    p = −1;      // initialize p to a negative value
    ⟨ Search the pivot-element, interchange rows and columns 68 ⟩
    ⟨ Do the pivoting 69 ⟩
  }
  ⟨ Test whether the system has a solution 70 ⟩
```

This code is used in section 66.

68. We find the pivot-element by searching through the actual submatrix. Since the interior of this loop is executed very often, we use some dirty C constructions.

```
⟨ Search the pivot-element, interchange rows and columns 68 ⟩ ≡
  int p_row, p_col;      // position of p
  number *N;      // to step through the elements of a row of C
```

```
for ($j = i$; $j < rows$; $j$++) {      // step through rows $i$ to $rows - 1$
  $N = \&(C[j][cols])$;      // in each row start with $N$ at the right end
  for ($k = cols - i + 1$; $k$−−; ) {
        // step through columns $cols - i + 1 + i - 1$ to $i - 1 + 1$
    /* $N$ still points to the element right of the current column */
    if ($*$−−$N > p \lor *N < -p$) {      // new pivot element found
      $p$ = number_abs($*N$);
      $p\_row = j$;      // store the position of the new $p$
      $p\_col = k + i - 1$;
    }
  }
}
/* if $p$ is zero, we can stop. The test whether $p$ is zero actually is a test
whether $p$ is less then EPSILON */
if ($p \leq$ number(EPSILON)) break;      // exit the main loop
/* We interchange rows $i$ and $p\_row$ by interchanging the pointers */
number $*help = C[i]$;
$C[i] = C[p\_row]$;
$C[p\_row] = help$;
/* We interchange columns $i$ and $p\_col$ by copying */
if ($p\_col > i$)
  for ($j = 0$; $j < rows$; $j$++) {
    number $swap = C[j][i]$;
    $C[j][i] = C[j][p\_col]$;
    $C[j][p\_col] = swap$;
  }
  /* We store the interchanging of the variables in $var$ */
int $dummy = var[i]$;
$var[i] = var[p\_col]$;
$var[p\_col] = dummy$;
```

This code is used in section 67.

69. Now we are ready to do the pivot-step with the element $C_{i,i}$. Again we do a bit dirty C stuff to save a little time.

⟨ Do the pivoting 69 ⟩ ≡

```
$p = C[i][i]$;      // note that p was always positive !
$C[i][i] = 1$;      // $C_{i,i}$ becomes 1
for ($k = i + 1$; $k \leq cols$; $k$++)      // treat row $i$
  $C[i][k] = C[i][k]/p$;
number $*Cji$;      // to step through $C[j][i]$
number $*Cii = \&(C[i][i])$;      // will remain uncanged
```

```
for ($j = i + 1$; $j < rows$; $j$++) {        // for each row below row $i$
  $Cji = \&(C[j][i])$;
  number $factor = *Cji$;
  $k = cols - i$;
  while ($k$−−)      // for each $C[j][i+k]$ with $i + k \in [i \ldots cols - 1]$
    $Cji[k]$ −= $factor * Cii[k]$;      // note that $Cji[k] \equiv C[j][i+k]$
}
```

This code is used in section 67.

70. We introduce a new variable *max_row*, which holds the highest index of a non-zero row created by the elimination algorithm. If we have stopped with $p = 0$, we must test whether the system has a solution at all: if there is now in the rightmost column for b in a row whose index is greater then *max_row* a value not equal to zero, then the system has no solution at all.

⟨ Test whether the system has a solution 70 ⟩ ≡

```
int $max\_row = i - 1$;      // we have gone one row too far in the loop
if ($p \leq$ EPSILON)      // we have stopped with $p = 0$
  for ($j = max\_row + 1$; $j < rows$; $j$++)
    if (number_abs($C[j][cols]$) > EPSILON) return $L$;
          // we return the empty list $L$
```

This code is used in section 67.

71. We now have the diagonal elements $C_{0,0}, \ldots, C_{max\_row,max\_row} = 1$. All elements below them are zero. We make all elements above them zero, too. Then we have a unity matrix E_{max_row+1} in the upper left corner.

⟨ Make unity matrix on the left side 71 ⟩ ≡

```
for ($i = max\_row$; $i \geq 0$; $i$−−) {      // go up in the matrix
  number $*Cji$;
  number $*Cii = \&(C[i][i])$;      // will remain uncanged
  for ($j = i - 1$; $j \geq 0$; $j$−−) {      // for each row above row $i$
    $Cji = \&(C[j][i])$;
    number $factor = *Cji$;
    $k = cols - i$;
    while ($k$−−) {      // for each $C[j][i+k]$ with $i + k \in [i \ldots cols - 1]$
      $Cji[k]$ −= $factor * Cii[k]$;      // note that $Cji[k] \equiv C[j][i+k]$
    }
  }
}
```

This code is used in section 66.

72. We now have E_{max_row+1} in the upper left corner, which means that we have *max_row* + 1 variables, which depend on the *cols* − (*max_row* + 1) free variables. The vector *var* tells us, which variables are free and which are dependent. The affine space of the solution is thus generated by *cols* − (*max_row* + 1) spanning vectors and has a base point *base*. For every free variable, we create a vector. Then we give for every dependent vector its depending parts to the appropriate spanning vectors and to *base*. The dimension of the spanned space is stored in the variable *dim_of_span*.

```
⟨ Compute the list L which contains the solution 72 ⟩ ≡
  int dim_of_span = cols − (max_row + 1);
  /* dim_of_span > 0 means that the solution contains more than one point */
#ifdef USE_LEDA_ARRAYS
  array⟨vector⟩ span(0, (dim_of_span > 0) ? dim_of_span − 1 : 0);
#else
  vector *span = new vector [(dim_of_span > 0) ? dim_of_span : 1];
#endif
  vector base(cols);      // base is initialized to be the 0-vector
  for (i = 0; i < dim_of_span; i++) span[i] = base;
      // initialize span[i] to be the 0-vector
  /* Next, we set the components of those spanning vectors to 1, which belong
  to the free variables */
  for (i = 0; i < dim_of_span; i++) span[i][var[max_row + 1 + i]] = 1;
  /* Now we treat every dependent variable */
  for (i = 0; i ≤ max_row; i++) {
    base[var[i]] = C[i][cols];
    for (k = max_row + 1; k < cols; k++)
      span[k − max_row − 1][var[i]] = −C[i][k];
  }
```

This code is used in section 66.

73. After we have deallocated the space for C, we can return the list L.

```
⟨ Deallocate memory and return the list L 73 ⟩ ≡
  for (i = 0; i ≤ cols − max_row − 2; i++) L.push(span[i]);
  /* O.K., let's insert base at the beginning of L and we are done ! */
  L.push(base);
#ifndef USE_LEDA_ARRAYS
  delete var;
  delete span;
#endif
  return L;
```

This code is used in section 66.

74. Dependency tests.

We first give an algorithm which determines whether a given list of vectors $v_1, \ldots, v_r$ are linearly dependent. To test them, we ask if there are $\alpha_1, \ldots, \alpha_r$, not all equal to zero, such that $\sum_{i=1}^{r} \alpha_i v_i = 0$.

```
⟨ Dependency tests 74 ⟩ ≡
  bool linear_dependency(list⟨vector⟩ L)
  /* returns true iff the vectors in L are linearly dependent. */
  { /* we simply create the appropriate matrix and give it to linear_solver( ),
       the left side b being zero */
    int cols = L.size( );     // the number of vectors
    int rows = L.head( ).dim( );     // the dimension we are working in
    if (cols > rows) return true;
         // more then rows vectors are always dependent
    matrix A(rows, cols);
    vector b(rows);     // note that b is by default the zero vector
    vector v;
    int row;
    int col = 0;
    forall (v, L) {     // fill the columns of A with the vectors in L
      for (row = 0; row < rows; row++) A(row, col) = v[row];
      col++;
    }
    list⟨vector⟩ R;     // result list for the call of linear_solver( )
    R = linear_solver(A, b);
    if (R.size( ) > 1) return true;
       // they are surely dependent because we have infinitely many
       // (α_1, ..., α_cols) that solve the system above
    /* Otherwise, we check whether any α_i is not equal to 0 */
    vector test = R.pop( );
    for (row = 0; row < cols; row++) {
      if (number_abs(test[row]) > EPSILON) return true;
    }
    /* The only case left now is that only the trivial linear combination of the
    v_i is zero, that is, they are linearly independent */
    return false;
  }
```

See also section 75.

This code is used in section 61.

75. The next function tests whether the vectors $v_1, \ldots, v_r$ are affine-linearly dependent. Note that this is the case iff $v_2 - v_1, \ldots, v_r - v_1$ are linearly dependent. Therefore, we can simply use the function *linear_dependency*( ) for the test.

```
⟨ Dependency tests 74 ⟩ +≡
  bool affine_dependency(list⟨vector⟩ L)
  /* returns true iff the vectors in L are affine-linearly dependent. */
  {
    list⟨vector⟩ Help;
    vector v1;
    v1 = L.pop( );
    while (¬L.empty( )) Help.push(L.pop( ) − v1);
    return linear_dependency(Help);
  }
```

76. Useful literature.

References

[1] C. Bradford Barber, David P. Dobkin, Hannu Huhdanpaa "The quickhull Algorithm for Convex Hull" Geometry Center Techical Report GCG53, University of Minnesota, available via anonymous ftp from `geom.umn.edu:pub/qhull.tar.Z`

[2] Christoph Burnikel, Kurt Mehlhorn, Stefan Schirra "On Degeneracy in Geometric Computations"

[3] K.E. Clarkson, K. Mehlhorn, Raimund Seidel "Four Results on Randomized Incremental Constructions" Technical Report MPI-I-92-112, March 1992

[4] K.E. Clarkson, K. Mehlhorn, Raimund Seidel "Four Results on Randomized Incremental Constructions" revised version of February 15, 1993

[5] Donald Knuth, Silvio Levy "The CWEB System of Structured Documentation" available via anonymous ftp from `labrea.stanford.edu:/pub/cweb/cweb.tar.gz` or `ftp.th-darmstadt.de:/pub/programming/literate-programming/c.c++/cweb.tar.gz`

[6] Kurt Mehlhorn, Stefan Näher: "LEDA, a Library of Efficient Data Types and Algorithms", Proceedings of the 14th Symposium on Mathematical Foundations of Computer Science, LNCS Vol. 379, 88-106, 1989, (to appear in Communications of the ACM)

[7] Stefan Näher: "LEDA Manual Version 3.0", Technical Report, MPI-I-93-106, Max-Planck-Institut für Informatik, Saarbrücken, 1993.

Index

List of Refinements

⟨ Add a new unbounded simplex 48 ⟩ Used in section 47.
⟨ Complete neighborhood information if F is bounded 50 ⟩ Used in section 47.
⟨ Complete neighborhood information if F is unbounded 49 ⟩ Used in section 47.
⟨ Compute the arrays nx and nO and test whether $x \in S$ 38 ⟩ Used in section 37.
⟨ Compute the list L which contains the solution 72 ⟩ Used in section 66.
⟨ Compute the plane equation 20 ⟩ Used in section 19.
⟨ Compute the right hand side *alpha* and adjust the sign of *normal* 23 ⟩ Used in section 20.
⟨ Deallocate memory and return the list L 73 ⟩ Used in section 66.
⟨ Decide whether $<^\sigma$ is $<$ or $>$ 43 ⟩ Used in section 42.
⟨ Dependency tests 74, 75 ⟩ Used in section 61.
⟨ Dimension jump 46 ⟩ Used in section 24.
⟨ Do the pivoting 69 ⟩ Used in section 67.
⟨ Extract *normal* from L 22 ⟩ Used in section 20.
⟨ Find x-visible hull facets 32 ⟩ Used in section 28.
⟨ Find the facet with largest $\lambda < 0$ 40 ⟩ Used in section 37.
⟨ For each horizon ridge add the new simplex 29 ⟩ Used in section 28.
⟨ Further member declarations of **class Triangulation** 17, 25, 31, 39, 45 ⟩ Used in section 11.
⟨ Go to the next Simplex on the ray $\overrightarrow{Ox}$ 41 ⟩ Used in section 37.
⟨ Header files for linalg 63 ⟩ Used in section 61.
⟨ Header files to be included 7, 8, 10 ⟩ Used in section 6.
⟨ Initialize the triangulation 27 ⟩ Used in section 24.
⟨ Input from keyboard or file 59 ⟩ Used in section 57.
⟨ Input from mouse 58 ⟩ Used in section 57.
⟨ Macro definitions 62 ⟩ Used in section 64.
⟨ Main program 54, 55 ⟩ Used in section 6.
⟨ Make unity matrix on the left side 71 ⟩ Used in section 66.
⟨ Make upper diagonal form 67 ⟩ Used in section 66.
⟨ Member functions of class Triangulation 18, 19, 24, 26, 33, 34, 35, 36, 37, 42, 44, 47, 51, 52, 53 ⟩ Used in section 6.
⟨ Non-dimension jump 28 ⟩ Used in section 24.
⟨ Process the data 57 ⟩ Used in section 55.
⟨ Read the command line 56 ⟩ Used in section 55.
⟨ Search the pivot-element, interchange rows and columns 68 ⟩ Used in section 67.
⟨ Set up the matrix 21 ⟩ Used in section 20.
⟨ Test whether the system has a solution 70 ⟩ Used in section 67.
⟨ The function 65, 66 ⟩ Used in section 61.
⟨ Update the neighborhood relationship 30 ⟩ Used in section 28.
⟨ class Simplex 15, 16 ⟩ Used in section 6.
⟨ class Triangulation 11, 12, 13, 14 ⟩ Used in section 6.
⟨ `linalg.c` 61 ⟩

⟨`linalg.h` 64⟩