An Implementation of a Convex Hull Algorithm

Version 2.0

Jörg Dorchain

Erklärung

Ich erkläre die vorliegende Arbeit selbständig im Sinne der Diplomprüfungsordnung erstellt und ausschließlich die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Saarbrücken, den

Jörg Dorchain

Acknowledgements

I wish to thank the persons who made this work possible: Kurt Mehlhorn and Stefan Näher for giving me the theme, Michael Müller and Joachim Ziegler for giving me their full source code of Version 1.0, Kurt Mehlhorn for surveillance and giving useful hints when necessary, Christian Uhrig who I asked a lot of questions about LEDA and C++, Tanja Maurer and Thomas Ziegler for proof-reading and Alexander Eckert (9 years old) for magnificiently producing random inputs.

Abstract

This is an implementation of an incremental construction algorithm for convex hulls in $\mathbb{R}^d$ based on [3] using *Literate Programming* and *Leda* in *C++*. The structure of the program and data types are widely based on [6]. The authors had the kind of giving me their complete source as a start of my work. While they did the basic data structure, I added a deletion procedure and numerical correctness, improved the stability of the algorithm and introduced some other features to make it usable as an abstract data type by an average LEDA-user. In [6], the authors included a solver for Gauß'-Elimination and a routine for computing hyperplanes which are soon to be included into LEDA as data types of their own. I built in prelimary releases of these types that were kindly made available to me by Kurt Mehlhorn. Especially the implementation of the *segment-walk*-algorithm became considerably easier. When fiddeling them in and doing other necessary change to the original code the modular concept of the literate programming was very useful.

Keywords

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

Contents

1. Introduction.

This is an implementation of an incremental construction algorithm for convex hulls in $\mathbb{R}^d$ using *Literate Programming* (cf. [4]) and *LEDA* (cf. [7]) in C++. The algorithm has been developed by Clarkson, Mehlhorn and Seidel (cf. [3]). In [2], a minor modification of this algorithm is described which maintains convex hulls in arbitrary dimensions without any non-degeneracy assumption.

2. There is a demo program to exercise the implemented data type. While the implementation is given in section 92, we explain here how it is used. It is called `chull` with several options.

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. (default)
- **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 information 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. (default)

The search methods implement different algorithms for finding facets a point sees. Which you choose doesn't incluence the behaviour of the program except for running time. They are discussed later in detail.

3. The 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 use *literate programming*. From *LEDA* (a Library of Efficient Data types and Algorithms) we take 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 [3] and [2]. The citations appear in a smaller font and are terminated by a mention of the source (e.g., cited text (cf. reference)).

4. 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 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 conv R we maintain a triangulation $\Delta(\pi_i)$ of conv R_i: a simplical complex whose union is 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 conv R_i: it consists of all facets of $\Delta(\pi_i)$ which are incident to only one simplex of $\Delta(\pi_i)$. If $x \in$ 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 aff R_i that is supported by F and contains 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., $x_i \notin$ 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 vert(S) denote the set of vertices that define this simplex. It is clear that $\Delta(\pi)$ contains a 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 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. 3)

5. 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

---

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

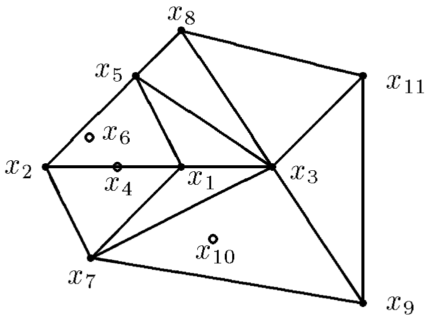

Figure 1: A triangulation. The dimension jumps are the points x_1, x_2, and x_5.

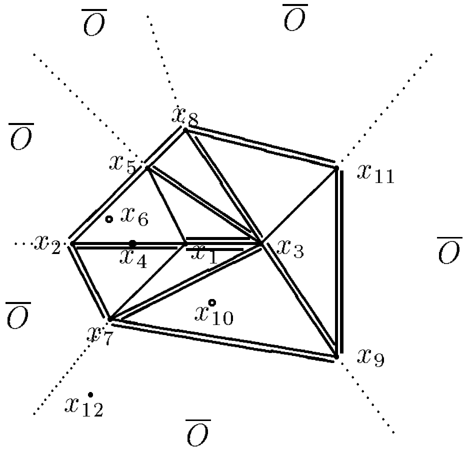

Figure 2: An extended triangulation

such that the peak lies in the positive halfspace, and for each simplex S and vertex $x \in \operatorname{vert} S$ we store the other simplex[2] sharing the facet with vertex set $\operatorname{vert}(S) \setminus \{x\}$. 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. 4)

$\overline{O}$ 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 $\operatorname{conv}(x_2, x_7)$ and $\operatorname{conv}(x_7, x_9)$. The simplex opposite to the vertex x_1 with respect to the simplex $\operatorname{conv}(x_1, x_3, x_5)$ is the simplex $\operatorname{conv}(x_3, x_5, x_8)$. The vertex opposite to x_1 in this simplex is the vertex x_8.

---

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

6. 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 [3]. 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], p. 4)
We call this search method the *visibility search method.*

Another search method is as follows.
Locate x in T by walking along the segment $\overline{Ox}$ beginning at O. If this walk enters a simplex whose peak is the anti-origin, then an x-visible current facet has been found. Otherwise, a simplex of T containing x has been found, showing that $x \in \operatorname{conv} R$. In the former case, find all x-visible hull facets by a search of the simplices incident to the anti-origin. These simplices form a connected set in the neighbourhood graph. (cf. [3],p. 7)
This method is called the *segment walking method.*

7. We also give an overview of the deletion algorithm. The global plan is quite simple. When a point x is deleted from R, we change the triangulation T so that in effect x was never added. This is in the spirit of §2. The effect of the deletion of x on the triangulation is easy to describe. All simplices having x as a vertex disappear (If x is not a vertex of T then T does not change). The new simplices of T resulting from the deletion of x all have base facets visible to x, with peak vertices inserted after x. These are the simplices that would have been included if x had not been inserted into R. Let $R(x)$ be the set of points of R that are contained in simplices with vertex x, and also inserted after x. We will, in effect, reinsert the points of $R(x)$ in the order in which they were inserted into R, constructing only those simplices that have bases visible to x. On a superficial level, this describes the deletion process. The details follow. (cf. [3], p. 10)

8. The Basic Structure of the Program.

We want to separate our program in three parts: a header file containing all definitions for inclusion in other programs, a code file containing all implementations, and a short demo program mainly used for debugging and presentation. The fundamental data structures for the simplicial complex will be called **class Simplex** and **class Triangulation**. With these terms, we can give a short overview of the program now.

9. These parts go into the header file.

```
⟨ chull.h  9 ⟩ ≡
#ifndef CHULL_H
#define CHULL_H
  ⟨ Header files to be included  12 ⟩
  ⟨ class Triangulation  15 ⟩
#endif
```

10. This is everything that produces code.

```
  ⟨ class Simplex  19 ⟩
  ⟨ Member functions of class Triangulation  16 ⟩
  ⟨ Friend functions of class Triangulation  48 ⟩
```

11. Our demo routine.

```
⟨ main.c  11 ⟩ ≡
  ⟨ Main program  92 ⟩
```

12. From *LEDA* we use the data types **array**, **list**, *h_array*, **integer**, **vector**, **dictionary** and we use streams for I/O.

```
⟨ Header files to be included  12 ⟩ ≡
#include <LEDA/array.h>
#include <LEDA/list.h>
#include <LEDA/h_array.h>
#include <LEDA/integer.h>
#include <LEDA/vector.h>
#include <LEDA/dictionary.h>
#include <LEDA/stream.h>
```

See also sections 13 and 14.

This code is used in section 9.

13. 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 12 ⟩ +≡

```
#include <LEDA/window.h>
#include <LEDA/plane.h>
```

14. We use the datatypes **d_rat_point** and **hyperplane**[3] for numerically correct computations in conjunction with their basic type (LEDA-)**integer** where we need to. They provide easy to use member- and friend-functions hiding several Gaussian eliminations from the code. They will be integrated into LEDA soon.

⟨ Header files to be included 12 ⟩ +≡

```
#include "d_rat_point.h"
#include "hyperplane.h"
```

---

[3] Many thanks to Kurt Mehlhorn for giving me working prelimary versions of them.

15. The Fundamental Data Structures.

Now we can begin to define our fundamental data structures (cf. Section 5). 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. 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 83) 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

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

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 O, we have to remind that $O = quasi\_center/(dcur + 1)$. Furthermore, we store a **list** of all constructed simplices (*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 more efficient 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 [3] 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.

In order to support the efficient computation of the set $R(x)$[4] we need to augment our data structure slightly. We assume that each point stores a pointer to some simplex containing it and that every simplex stores a list of the points contained in it. (cf. [3], p. 12) The pointers to the simplex for each point are stored in a LEDA *h_array simplex*. We use *h_array*s for the insertion order of the points in the list *coordinates*, and to store the position of x in the *points*-list iff x is an interior point (*nil* if it is a vertex). These *h_array*s are mainly of use when deleting a point.

---

[4] This set is explained in the discussion of the deletion process (section 51)

```
⟨class Triangulation 15⟩ ≡
  class Simplex;    // forward reference
  enum search_method {
    VISIBILITY, SEGMENT_WALK, MODIFIED_VISIBILITY
  };
  class Triangulation {
private:
  list⟨d_rat_point⟩ coordinates;    // the coordinate vectors of the x_i
  dictionary⟨d_rat_point, list⟨list_item⟩ ∗⟩ co_index;
      // the index in coordinates for each point
  int co_nr;    // current max. order nr for the point in coordinates
  h_array < list_item , int > order_nr;    //order number for each point
  h_array < list_item , Simplex ∗ > simplex;    // a simplex x belongs to
  h_array < list_item , list_item > position;
      // position of x in simplex[item_x]→points, nil if x is a vertex
  int dcur;    // dimension of the current convex hull
  int dmax;    // dimension of the coordinate vectors
  Simplex ∗origin_simplex;    // pointer to the origin simplex
  d_rat_point quasi_center;
      // sum of the coordinate vectors of the vertices of the origin simplex
  list⟨Simplex ∗⟩ all_simplices;    // list of all simplices
  ⟨Further member declarations of class Triangulation 26⟩
  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 d_rat_point &x);    // insertion routine
  bool member(const d_rat_point &x);    // to test if
  void del(const d_rat_point &x);    // deletion routine
  d_rat_point find_closest_point(const d_rat_point &x);
      // for mouse-convenience
  void show(window &W);    // draws the triangulation onto the screen
  void print_all();    // calls print() for all simplices
  void print_extremes(ostream & o);    // print outer points
  list⟨d_rat_point⟩ points();    // return coordinates
  Triangulation(int d = 0, search_method m = SEGMENT_WALK);
      // constructor function with default arguments
  ∼Triangulation();    // destructor function
  Triangulation(Triangulation &T);    // copy-constructor
  Triangulation &operator=(const Triangulation &T);
      //currently disabled
  friend ostream& operator≪ ( ostream & , Triangulation & ) ;
      // I/O for
  friend istream& operator≫ ( istream & , Triangulation & ) ;
```

```
    // Triangulations
  } ;
```

This code is used in section 9.

16. 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**( )).

```
⟨ Member functions of class Triangulation 16 ⟩ ≡
  int Triangulation :: created_simplices ( )
  {
    Simplex Dummy(2);
    static dummys_created = 0;
    dummys_created++;
    return Dummy.sim_nr − dummys_created;
  }
```

See also sections 17, 18, 21, 27, 28, 29, 30, 31, 32, 36, 38, 41, 45, 46, 47, 50, 51, 57, 81, 83, 85, 86, 88, 90, and 91.

This code is used in section 10.

17. The constructors for **class Triangulation** are easy to implement. The default search method is segment walking.

```
⟨ Member functions of class Triangulation 16 ⟩ +≡
Triangulation :: Triangulation(int d, search_method m):
      order_nr(−1), simplex(nil), position(nil)
  {
    co_nr = 0;
    dcur = −1;
    dmax = d;
    searched_simplices = 0;
    origin_simplex = nil;
    method = m;
    inner_simplex = nil;
  }
Triangulation :: Triangulation(Triangulation &T) :
      order_nr(−1), simplex(nil), position(nil)
  {
    d_rat_point v;
    co_nr = 0;
    dcur = −1;
    dmax = T.dmax;
    searched_simplices = 0;
    origin_simplex = nil;
    method = T.method;
    inner_simplex = nil;
    forall (v, T.coordinates) insert(v);      // this is the actual copying
  }
```

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

```
⟨Member functions of class Triangulation 16⟩ +≡
  Triangulation ::~Triangulation ( )
  {
    Simplex *S;
    forall (S, all_simplices) delete (S);
  }
```

19. 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., $V\text{→}vertices[k] \equiv v$, and w is the l-th vertex of W, then $V\text{→}opposite\_vertices[k] \equiv l$ and vice versa $W\text{→}opposite\_vertices[l] \equiv k$.

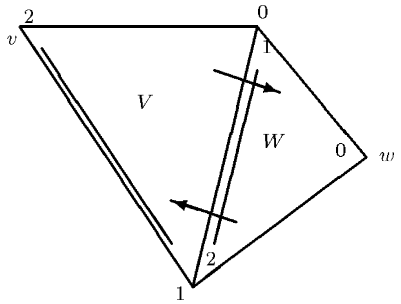

Figure 3: The connection of two simplices V and W

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→neighbors*[2] ≡ *W* and *W→neighbors*[0] ≡ *V*, indicated 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 hyperplane which contains the facet. The normal vector must be oriented in such a way that the vertex opposite to the face lies in the positive halfspace. When we need the equation of a hyperplane for a facet i of a simplex S (i.e, the facet opposite to the i-th vertex of S), we call the function *compute_plane*( ) (or more often *sees_facet*( ), which is more comfortable and often suffices.) (cf. Section 83), which are members of **class Triangulation**. Once we have computed these values for a facet, we store them in the array *facets* 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 *facets* become invalid. Therefore, we store in an array *valid_equations* the time, i.e. the current dimension *dcur* of the convex hull, when the value of the facet's equation was computed. The functions *sees_facet*( ) and *compute_plane*( ) 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] $\neq$ *dcur*. Then the values are computed new and *valid_equations*[i] is set to *dcur*. Initially, they are set to -1 to denote that none are valid.

The **list** *points* holds the position of the inner points of the simplex (points inserted after the peak vertex that lie in the simplex).

In the implementation of the deletion process we must not forget that we may have to set *valid_equations*[i] to -1 again for all i and all simplices if we delete a vertex which was a dimension jump. This sounds expensive, but we have to look anyway at all vertices of all simplices of the remaining hull when we reduce the dimension, so invalidating the plane equations adds only a constant amount of time.

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).

In the deletion process we find it useful to have the **list_item** of a simplex handy for updating the **list** *all_simplices*.

```
#define anti_origin  nil

⟨ class Simplex 19 ⟩ ≡
#include "chull.h"
  class Simplex {
    friend class Triangulation;
       // Triangulation has unrestricted access
  private:
    int sim_nr;      // useful for debugging; unique number for each simplex
    list_item this_item;      // points into all_simplices, for easy deletion
    array⟨list_item⟩ vertices;
```

```
      // pointers to the coordinate vectors of the vertices
    array⟨Simplex ∗⟩ neighbors;
    array⟨int⟩ opposite_vertices;      // indices of opposite vertices
    array⟨hyperplane⟩ facets;
    array⟨int⟩ valid_equations;
      // dimension in which corresponding hyperplane was computed
    bool visited;
      // used to mark simplices when traversing the triangulation
    list⟨list_item⟩ points;
      // pointer into coordinates of other points located in this vertex
    Simplex(int dmax);      // constructor function
    ∼Simplex() { } ;      // destructor function
    LEDA_MEMORY(Simplex);
  };
```

See also section 20.

This code is used in section 10.

20. The constructor for **class Simplex** sets the size of the arrays and marks the simplex as not visited.

```
⟨ class Simplex 19 ⟩ +≡
Simplex :: Simplex(int dmax):
        vertices(0, dmax), neighbors(0, dmax), opposite_vertices(0, dmax),
            facets(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++) {
      neighbors[i] = nil;      // to avoid illegal pointers when using print()
      valid_equations[i] = −1;
    }
    visited = false;
  }
```

21. The Insert Procedure.

We treat now the insertion procedure as described in Section 6. 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 16 ⟩ +≡
  void Triangulation :: insert(const d_rat_point &x) { dic_item dic_x;
          // add x to the points already inserted and store its position in item_x
    if (x.dim( ) ≠ dmax)
      error_handler(99, "chull:␣incorrect␣dimension");
    list_item item_x = coordinates.append(x);
    order_nr[item_x] = co_nr++;      //save the insertion order of x
        // store position of x in coordinates in a dictionary
    if ((dic_x = co_index.lookup(x)) ≡ nil)
            { dic_x = co_index.insert (x, new list⟨list_item⟩ ) ;
        } co_index.inf(dic_x)→append(item_x);
        if (dcur ≡ −1) {      // x is the first point to be inserted
          ⟨ Initialize the triangulation 22 ⟩
        }
        else if ((dcur < dmax) ∧ is_dimension_jump(x)) {
              // see Section 85
          ⟨ Dimension jump 40 ⟩
        }
        else {
          ⟨ Non-dimension jump 23 ⟩
        }
        }
```

22. 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. Also the *simplex*[*item_x*] value is clear.

```
⟨ Initialize the triangulation 22 ⟩ ≡
  Simplex *outer_simplex;       // a pointer to the outer simplex
  dcur = 0;      // we jump from dimension -1 to dimension 0
  origin_simplex = new Simplex (dmax);
  origin_simplex→this_item = all_simplices.append(origin_simplex);
  outer_simplex = new Simplex (dmax);
  outer_simplex→this_item = all_simplices.append(outer_simplex);
  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;
  simplex[item_x] = origin_simplex;
```

This code is used in section 21.

23. We discuss now how to handle insertions that are not dimension jumps. If the current dimension is zero, we append x to the points interior to the *origin_simplex*, since it is the same point as the only one we already inserted. Otherwise we first compute the set of all x-visible hull facets. This is described in detail in Section 26. As a result of this step, we get in *visible_simplices* the list of unbounded simplices whose base facets see x, or in *inner_simplex* the simplex in which x in located if x lies in the interior of the current hull. If *visible_simplices* is empty, then x lies within the current hull and we have to add it to the simplex it lies in (needed for later deletion). Otherwise, we have to modify some simplices and to add some new ones as described in Section 6. Also the neighborhood information has to be updated.

```
⟨Non-dimension jump 23⟩ ≡
      // when we come here in dim 0, we inserted the same point twice
  if (dcur > 0) {
    find_visible_facets(x);      // see Section 27
    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→vertices[0].
        */
        S→vertices[0] = item_x;
        simplex[item_x] = S;
        ⟨For each horizon ridge add the new simplex 24⟩
      }
      visible_simplices.clear();
      ⟨Update the neighborhood relationship 25⟩
    }
    else      // we have to add x to the simplex it lies in
    {
      position[item_x] = inner_simplex->points.append(item_x);
      simplex[item_x] = inner_simplex;
      inner_simplex = nil;
    }
  }
  else {
    position[item_x] = origin_simplex->points.append(item_x);
    simplex[item_x] = origin_simplex;
  }
```

This code is used in section 21.

24. 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. [3], p. 8)

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->vertices = S->vertices;
T->vertices[k] = item_x;
T->vertices[0] = anti_origin;
```

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

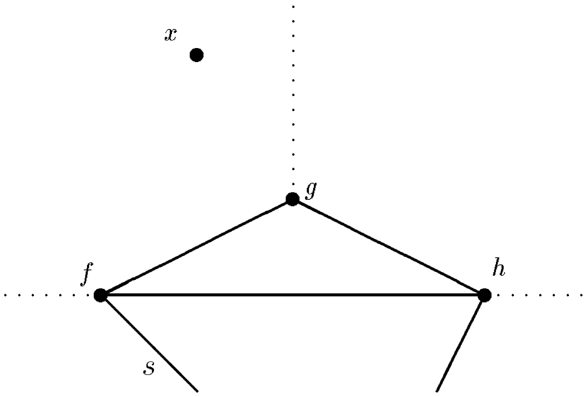

Figure 4: Before x is inserted

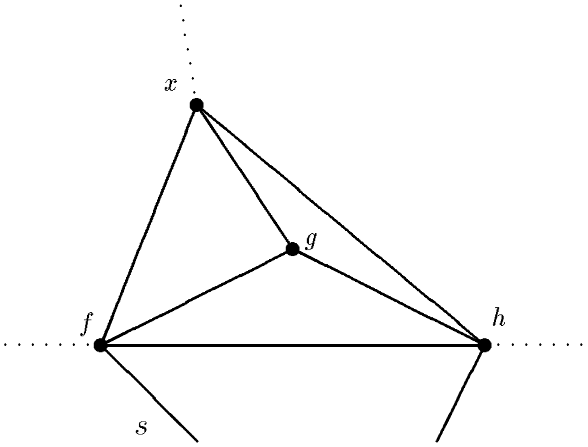

Figure 5: After x has been inserted

the second line by

```
T→vertices[k] = S→vertices[dcur];
T→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 $1 \leq j < dcur$, are computed in the next section.

```
⟨For each horizon ridge add the new simplex 24⟩ ≡
  for (int k = 1; k ≤ dcur; k++) {
    if (sees_facet(S→neighbors[k], 0, x) ≤ 0) {
        // x doesn't see the base facet of the neighbor
      Simplex *T = new Simplex (dmax);
      T→this_item = 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 Ō */
      int ii;
      for (ii = 0; ii ≤ dcur; ii++) T→vertices[ii] = S→vertices[ii];
      T→vertices[k] = S→vertices[dcur];
      T→vertices[dcur] = item_x;
      T→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→neighbors[dcur] = S→neighbors[k];
      T→opposite_vertices[dcur] = S→opposite_vertices[k];
      T→neighbors[0] = S;
      T→opposite_vertices[0] = k;
      /* Also set the reverse pointers from those two neighbors to the new
      simplex */
      S→neighbors[k]→neighbors[S→opposite_vertices[k]] = T;
      S→neighbors[k]→opposite_vertices[S→opposite_vertices[k]] = dcur;
      S→neighbors[k] = T;
      S→opposite_vertices[k] = 0;
    }
  }
```

This code is used in section 23.

25. We now complete the update of the neighborhood relation. How the neighborhood relationship has to be updated is described in [3] 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 $\text{conv}(f \cup \{\overline{O}\})$; $\overline{V} \in \mathcal{V}$ [5] 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

---

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

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. [3], p. 8)

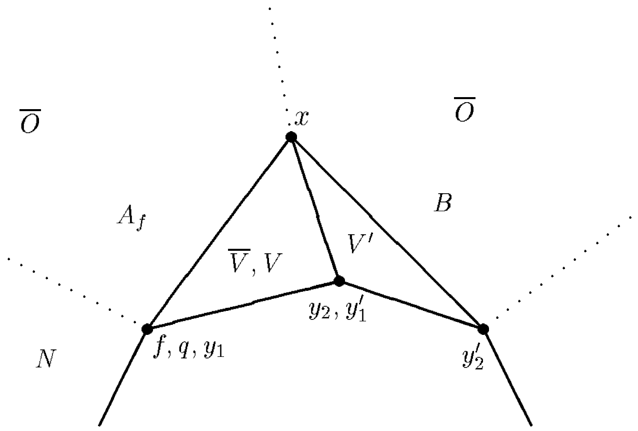

Figure 6: Updating the neighborhood relation

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. 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 $T = V =\vee$ *Af*→*neighbors*[0] as described in [3]. 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 25 ⟩ ≡
  Simplex *Af;
  forall (Af, NewSimplices) {
    for (int k = 1; k < dcur; k++) {
          // for all vertices q of Af except x and O̅ find the opposite neighbor
      Simplex *T = Af→neighbors[0];
      int y1;
      for (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 ∈ S
        /* find new y1 */
        int new_y1;
        for (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 23.

26. Finding x-visible Hull Facets.

For finding the x-visible hull facets, we implement 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 *modified-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*( ).

The roof function for these is *find_visible_facets*( ). It switches to the appropriate function und does the cleanup.

```
⟨ Further member declarations of class Triangulation 26 ⟩ ≡
  void visibility_search(Simplex *S, const d_rat_point &x);
  Simplex *search_to_outside(Simplex *S, const d_rat_point &x);
  Simplex *segment_walk(const d_rat_point &x);
  void collect_visible_simplices(Simplex *S, const d_rat_point &x);
  void clear_visited_marks(Simplex *S);
  void find_visible_facets(const d_rat_point &x);
  list⟨Simplex *⟩ visible_simplices;     // result of find_visible_facets( )
  Simplex *inner_simplex;
     // where x is located if visible_simplices is empty
```

See also sections 34, 39, 56, 80, 82, 84, 87, and 89.

This code is used in section 15.

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

```
⟨ Member functions of class Triangulation 16 ⟩ +≡
  void Triangulation ::find_visible_facets(const d_rat_point &x)
  {
    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);
      if ((visible_simplices.empty()) ∧ (inner_simplex ≡ nil))
        inner_simplex = origin_simplex;
      break;
    case MODIFIED_VISIBILITY:
      last_simplex = search_to_outside(origin_simplex, x);
      if (last_simplex ≠ nil)      // if x is not an interior point
        collect_visible_simplices(last_simplex, x);
            // generates list of unbounded simplices with x-visible base facet
      else if (inner_simplex ≡ nil)      // x is interior but no simplex found
        inner_simplex = origin_simplex;      // so its the origin simplex
      clear_visited_marks(origin_simplex);
      break;
    case SEGMENT_WALK:
    default: last_simplex = segment_walk(x);
      if (last_simplex→vertices[0] ≡ 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);
      }
      else      // x lies in the interior
        inner_simplex = last_simplex;
      break;
    }
  }
```

28. How we can implement *visibility_search*( ) is described in Section 6: 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 16 ⟩ +≡
  void Triangulation :: visibility_search(Simplex *S,
          const d_rat_point &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 (sees_facet(T, 0, x) > 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
          visible_simplices.push(T);
              // we have found a visible simplex and store it
        else      // test if x lies within T
        {
          bool in = true;
          int j;
          for (j = 1; j ≤ dcur; j++)
            if (sees_facet(T, j, x) < 0) in = false;
          if (in) inner_simplex = T;
        }
        visibility_search(T, x);      // do the recursive search
      }
    }
  }
}
```

29. 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 16 ⟩ +≡
  Simplex *Triangulation :: search_to_outside(Simplex *S,
          const d_rat_point &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 (sees_facet(T, 0, x) > 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
          else {
            bool in = true;      // test if x sees the all facets
            int j;
            for (j = 1; j ≤ dcur; j++)
              if (sees_facet(T, j, x) < 0) in = false;
            if (in) inner_simplex = T;
          }
          Simplex *result = search_to_outside(T, x);
          if (result ≠ nil) return result;
        }
      }
      return nil;
    }
```

30. 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 16 ⟩ +≡
  void Triangulation :: collect_visible_simplices(Simplex *S,
          const d_rat_point &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 ≤ dcur; i++) {
      Simplex *T = S→neighbors[i];      // for all neighbors T of S
      if (¬T→visited ∧ T→vertices[0] ≡ anti_origin)
            // if the i-th neighbor has not been visited yet
            // and is an outer simplex
        if (sees_facet(T, 0, x) > 0)
              // if x sees the base facet of the i-th neighbor
          collect_visible_simplices(T, x);      // do the recursive collecting
    }
  }
```

31. 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 16 ⟩ +≡
  void Triangulation :: clear_visited_marks(Simplex *S)
  {
    S→visited = false;      // clear the visited-bit
    for (int i = 0; i ≤ 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
  }
```

32. 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 $\underline{quasi\_center}/(dcur+1)$ 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. In order to guarantee that the ray $\overrightarrow{Ox}$ passes only through the interior of facets we perturb the point O. The perturbation scheme is similiar to the well known perturbation method for the simplex algorithm. Let ϵ be a positiv infinitesimal, let $\mathcal{E}$ denote the point $(\epsilon, \epsilon^2, \ldots, \epsilon^d)$ and let $O^\epsilon = O + \mathcal{E}$.

The function *lambda_cmp*( ) (s. section 36) computes the order in which the ray $\overrightarrow{Ox}$ intersects various hyperplanes. Only here the perturbation plays a role. We now develop the mathematics underlying this subroutine.

Let h be a hyperplane. A hyperplane is the zero-set of an affine function $h(x) = \sum_{0 \le i < d} h_i x_i + h_d$, where $x = (x_0, \ldots, x_{d-1})$ is a point given by its cartesian coordinates. Let $\overline{h}(x) = \sum_{0 \le i < d} h_i x_i$. Then $h(x) = \overline{h}(x) + h_d$, $\overline{h}$ is linear, and $\overline{h}(x+y) = \overline{h}(x) + \overline{h}(y)$, $\overline{h}(\lambda x) = \lambda \overline{h}(x)$, $h(x-y) = h(x) - \overline{h}(y)$, and $h(x) - h(y) = \overline{h}(x) - \overline{h}(y)$.

The points on the ray $\overrightarrow{Ox}$ satisfy the equation

$$r(\lambda) = O^\epsilon + \lambda(x - O^\epsilon).$$

The parameter value λ for which the ray r intersects the hyperplane h is determined by the equation $0 = h(r(\lambda)) = h(O^\epsilon) + \overline{h}(\lambda(x - O^\epsilon)) = h(O^\epsilon) + \lambda(\overline{h}(x) - \overline{h}(O^\epsilon)) = h(O^\epsilon) + \lambda(h(x) - h(O^\epsilon))$. Thus

$$\lambda_h = -\frac{h(O^\epsilon)}{h(x) - h(O^\epsilon)}. \tag{1}$$

The perturbation method guaranties that the denominator is non-zero. Since $O^\epsilon = O + \mathcal{E}$ we have $h(x) - h(O^\epsilon) = h(x) - h(O) - \overline{h}(\mathcal{E}) = h(x) - h(O) - \epsilon \cdot \sum_{0 \le i < d} h_i \epsilon^i$. We conclude that

$$\operatorname{sign}(h(x) - h(O^\epsilon)) = \begin{cases} \operatorname{sign}(h(x) - h(O)) & \text{, if } h(x) - h(O) \neq 0 \\ -\operatorname{sign}(h_i) & \text{, if } \begin{array}{l} h(x) - h(O) = 0, \\ \text{and } i \text{ is minimal with } h_i \neq 0 \end{array} \end{cases} \tag{2}$$

and analogous

$$\operatorname{sign}(h(O^\epsilon)) = \begin{cases} \operatorname{sign}(h(O)) & \text{, if } h(O) \neq 0 \\ \operatorname{sign}(h_i) & \text{, if } \begin{array}{l} h(O) = 0, \\ \text{and } i \text{ is minimal with } h_i \neq 0 \end{array} \end{cases} \tag{3}$$

Note that at least one of the coefficients $h_0, \ldots, h_{d-1}$ is non-zero. Hence $h(x) - h(O^\epsilon)$ and $h(O^\epsilon)$ are always non-zero.

Let g be another hyperplane and let $\lambda_g = -g(O^\epsilon)/(g(x) - g(O^\epsilon))$ be the parameter value for the intersection of $\overrightarrow{Ox}$ with g. This ray intersects g before h iff $\lambda_g < \lambda_h$. We have

$$\begin{aligned} \lambda_g < \lambda_h &\iff \frac{-g(O^\epsilon)}{g(x) - g(O^\epsilon)} < \frac{-h(O^\epsilon)}{h(x) - h(O^\epsilon)} \\ &\iff g(O^\epsilon) \cdot (h(x) - h(O^\epsilon)) >^\sigma h(O^\epsilon) \cdot (g(x) - g(O^\epsilon)) \end{aligned} \tag{4}$$

where $>^\sigma$ indicates $>$ iff the signs of the two denominators are equal and denotes $<$ otherwise. Using $h(O^\epsilon) = h(O) + \overline{h}(\mathcal{E})$ and simplifying we obtain

$$\begin{aligned} \lambda_g < \lambda_h &\iff g(O)h(x) - h(O)g(x) + \overline{g}(\mathcal{E})h(x) - \overline{h}(\mathcal{E})g(x) >^\sigma 0 \\ &\iff \underbrace{g(O)h(x) - h(O)g(x) + \epsilon \cdot \sum_{0 \le i < d} \epsilon^i (g_i h(x) - h_i g(x))}_{E} >^\sigma 0 \end{aligned} \tag{5}$$

Under what circumstances is $\lambda_g = \lambda_h$ possible? If $\lambda_g = \lambda_h$ then the expression E must be zero. This is only possible if either $h(x) = g(x) = 0$, i.e., x lies on h and on g, or if $h = g$.[6]

The variable *in* tells us the number of the facet through which we have entered current simplex. It changes, 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*( ). We stop our walk, if we have found the simplex containing x (this might be an unbounded simplex, of course). We use two arrays *fx* and *fO* to store the values of the plane equations at x and O which we will need several times. We must not forget that these values have to be divided by $x[0]$ or $O[0]$ respectivly (the common denominator of the homogeneous coordinates) to get the correct value.

According to [5] the denominator is always positive, so it does not change the result of LEDA's *sign*-function for **integer**s, which we use for a quick comparison against 0.

```
⟨Member functions of class Triangulation 16⟩ +≡
  Simplex *Triangulation::segment_walk(const d_rat_point &x)
  {
    Simplex *S = origin_simplex;      // we start at the origin simplex
    bool x_in_S = false;
       // 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
    d_rat_point O = quasi_center/(dcur + 1);      // our center point
```

---

[6] Since $g(x) \neq 0$ implies $h_i = g_i h(x)/g(x)$ for $0 \le i < d$ and hence $\overline{h}(y) = h(x)/g(x)\overline{g}(y)$ for all y. From $0 = g(O)h(x) - h(O)g(x)$ we conclude further that $0 = \overline{g}(O)h(x) + g_d h(x) - \overline{h}(O)g(x) - h_d g(x) = \overline{g}(O)h(x) + g_d h(x) - (h(x)/g(x))\overline{g}(O)g(x) - h_d g(x) = g_d h(x) - h_d g(x)$ and therefore $h_d = g_d h(x)/g(x)$.

```
    array⟨integer⟩ fx(0, dcur);       // hᵢ(x)∗x[0]
    array⟨integer⟩ fO(0, dcur);       // hᵢ(O)∗O[0]
    while (¬x_in_S) {
      searched_simplices++;       // only for statistical reasons
      ⟨ Compute the arrays fx and fO and test whether x ∈ S 33 ⟩
      if (¬x_in_S) {
        ⟨ Go to the next Simplex on the ray Ox→ 35 ⟩
      }
    }
    return S;
  }
```

33. The computation of the arrays is easily done by calling *S→facet*[*i*].*value_at*( ). At the same time, we can test whether x lies in the current simplex.

```
⟨ Compute the arrays fx and fO and test whether x ∈ S 33 ⟩ ≡
  {
    x_in_S = true;
        // remains true until we find a facet which doesn't see x
    if (S→vertices[0] ≠ anti_origin)
        // otherwise we have reached the outside
    {
      for (i = 0; i ≤ dcur; i++) {
        compute_plane(S, i);       // just in case we need it the first time
            // remember to divide these values by the denominator of the point
        fx[i] = S→facets[i].value_at(x);
        fO[i] = S→facets[i].value_at(O);
        if (sign(fx[i]) < 0)       // see manual
          x_in_S = false;
      }
    }
  }
```

This code is used in section 32.

34. For the comparison of two λ's, we use the function *lambda_cmp*( ) defined in section 36. This function contains the actual perturbation method. A call *lambda_cmp*(*S*, *Od*, *xd*, *gx*, *gO*, *g*, *hx*, *hO*, *h*) returns true iff $\lambda_g < \lambda_h$. We also need a function *lambda_negative*( ) (s. section 38) which tells us wether λ_h is negative or not.

```
⟨ Further member declarations of class Triangulation 26 ⟩ +≡
  bool lambda_cmp(Simplex ∗S, const integer &Od, const integer &xd,
      const integer &gx, const integer &gO, int g,
      const integer &hx, const integer &hO, int h);
  bool lambda_negative(Simplex ∗S, const integer &Od, const integer
      &xd,
      const integer &nx, const integer &nO, int i);
```

35. Now we describe how to find the facet with number *out* through which we will leave the current simplex. Basically we have to find a facet *out* such that for all facets $i \neq out$ $\lambda_{out} < \lambda_i$. However there is an additional condition to meet to go only forward on the ray. If we are in the *origin_simplex*, $in \equiv -1$ and we entered the simplex from nowhere. Therefore we additionally have to test if λ is positive. If we are not in the *origin_simplex*, our λ_{out} has to be greater then λ_{in}, because otherwise the ray intersected facet *out* prior and we are looking for the next intersection and not for the last. To put it into formulas, facet i becomes the new candidate for the exit facet *out*, iff

$$\lambda_{in} < \lambda_i < \lambda_{out}$$

where λ_{in} is assumed to be 0 when $in \equiv -1$. 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 $\overrightarrow{Ox}$ 35 ⟩ ≡

```
int start = 0;    // prior to this facet we don't need to compare
int out;    // our hypothesis for desired facet
/* this loop terminates because x doesn't lie in S when we come here and
dcur is at least 1 */
while ((start ≡ in) ∨    // don't compare with ourselves
((in ≡ −1) ∧ lambda_negative(S, O[0], x[0], fx[start], fO[start], start)) ∨
    // when in the origin simplex we must not start with a negative lambda
((in ≠ −1) ∧ lambda_cmp(S, O[0], x[0], fx[start], fO[start], start, fx[in],
        fO[in], in)))    // otherwise we must not start with λ_start < λ_in
  start++;    // move one ahead
out = start++;
for (i = start; i ≤ dcur; i++)    // compare it to all others
{
  if ((i ≠ in) ∧    // we don't go back ....
  (i ≠ out))    // don't compare with ourselves
    if ((lambda_cmp(S, O[0], x[0], fx[i], fO[i], i, fx[out], fO[out], out)) ∧
            // the basic comparison (λ_i < λ_out)
    ((in ≠ −1) ∨ (¬lambda_negative(S, O[0], x[0], fx[i], fO[i], i))) ∧
            // additionally for the origin_simplex
    ((in ≡ −1) ∨ (lambda_cmp(S, O[0], x[0], fx[in], fO[in], in, fx[i], fO[i], i))))
          // additionally for all other simplices (λ_in < λ_i)
        out = i;
}
in = S→opposite_vertices[out];
S = S→neighbors[out];
```

This code is used in section 32.

36. It remains to describe how we decide whether $\lambda_g < \lambda_h$. As we have already mentioned we perturb[7] O to get $O^\epsilon := O + \mathcal{E}$, where $\mathcal{E} = (\epsilon, \epsilon^2, \ldots, \epsilon^{dcur})$ for some sufficiently small $\epsilon > 0$. Thus, if we consider ϵ small enough, the

---

[7] This perturbation method was proposed by Kurt Mehlhorn

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. In equation 5 we developed a condition to test.

The expression E contains two parts: one independ of $\mathcal{E}$ and one depending on $\mathcal{E}$. When the first part is $\neq 0$, it determines the sign of E because ϵ is small enough. If it is zero, the sign is determined by the first factor next to ϵ^i that is $\neq 0$.

Now let's have a look at the implementation. In LEDA a **d_rat_point** $x = (x_0, \ldots, x_{d-1})$ is represented in homogeneous coordinates $(G_0, \ldots, G_d)$ with $G_d > 0$ and $x_i = G_i / G_d$. Let

$$H(x) = \sum_{0 \leq i \leq d} h_i G_i = G_d \cdot \left(\sum_{0 \leq i < d} \frac{h_i G_i}{G_d} + h_d \right) = G_d \cdot h(x)$$

We draw some simple conclusions:

$$\begin{aligned} & g(O)h(x) - h(O)g(x) = \tfrac{G(O)H(x)}{o_d G_d} - \tfrac{H(O)G(x)}{o_d G_d} \\ \Longrightarrow \quad & \operatorname{sign}(g(O)h(x) - h(O)g(x)) = \operatorname{sign}(G(O)H(X) - H(O)G(x)) \qquad (6) \\ & g_i h(x) - h_i g(x) = \tfrac{g_i H(x)}{G_d} - \tfrac{h_i G(x)}{G_d} \\ \Longrightarrow \quad & \operatorname{sign}(g_i h(x) - h_i g(x)) = \operatorname{sign}(g_i H(x) - h_i G(x)) \qquad (7) \end{aligned}$$

Finally observe that $H(p) = h.value\_at(p)$ for a **d_rat_point** p and a **hyperplane** h. In conjunction with equation 5 it is now easy to decide whether $\lambda_g < \lambda_h$.

lambda_cmp( ) gets as parameter the simplex S the facets belong to, o_d which is Od in the current implementation, $G_d = xd$, $G(x) = gx$, $G(O) = gO$, the number g of the facet supporting the hyperplane g, and the analogous parts for the hyperplane h.

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

```
⟨Member functions of class Triangulation 16⟩ +≡
  bool Triangulation::lambda_cmp(Simplex *S,
            const integer &Od, const integer &xd,
            const integer &gx, const integer &gO, int g,
            const integer &hx, const integer &hO, int h)
  {
    bool sigma;
    int diffsign;
    bool diffgr0;
    int i;
    ⟨Decide whether >^σ is < or > 37⟩
    i = 1;      // First we test the parts which are not depending on ℰ.
    diffsign = sign(gO * hx − hO * gx);      // left part of E in 5 using 6
    while (diffsign ≡ 0) {
      /∗ the comparison depends on the factor of ℰ when we enter this while-
      loop ∗/
      diffsign = sign(S→facets[g][i] ∗ hx − S→facets[h][i] ∗ gx);
          // right part of E (∑ . . .) in 5 using 7
      i++;
    }
    diffgr0 = (diffsign > 0);
    return ¬(diffgr0 ⊕ sigma);
  }
```

37. Here we examine the direction of $>^\sigma$. We use another simple conclusion

$$
\begin{aligned}
h(x) - h(O) &= \frac{H(x)}{G_d} - \frac{H(O)}{o_d} = \frac{o_d H(x) - G_d H(O)}{G_d o_d} \\
\Longrightarrow \quad \operatorname{sign}(h(x) - h(O)) &= \operatorname{sign}(o_d H(x) - G_d H(O)) \qquad (8)
\end{aligned}
$$

and equation 2 to inspect the sign of the denominators in equation 4.

⟨ Decide whether $>^\sigma$ is $<$ or $>$ 37 ⟩ ≡

```
  {
    int lsign, rsign;
    int i;      // first compute the sign of the first denominator
    i = 1;
    lsign = sign(Od ∗ gx − xd ∗ gO);
    while (lsign ≡ 0)      // lsign depends on ℰ
      lsign = −sign(S→facets[g][i++]);      // now the second
    i = 1;
    rsign = sign(Od ∗ hx − xd ∗ hO);
    while (rsign ≡ 0)      // rsign depends on ℰ
      rsign = −sign(S→facets[h][i++]);
    sigma = (lsign ≡ rsign);
  }
```

This code is used in section 36.

38. To decide upon the sign of a λ_h, we use the function *lambda_negative*( ) which is similar to *lambda_cmp*( ) and returns true iff $\lambda_h < 0$. We have to decide upon the sign of the numerator and denominator of equation 1. Using our developed formulas 2 and 3 and our conclusion 8 this thing does not become a problem either. We do not forget the '−'-sign in front of the right expression in equation 1.

⟨ Member functions of class Triangulation 16 ⟩ +≡

```
  bool Triangulation :: lambda_negative(Simplex ∗S,
          const integer &Od, const integer &xd,
          const integer &hx, const integer &hO, int h)
  {
    int zsign, nsign;      // signs of numerator and denominator
    int i;      // first the numerator
    $i = 1$;
    $zsign = sign(hO)$;
    while $(zsign \equiv 0)$  $zsign = sign(S\text{-}facets[h][i{+}{+}])$;
            // now for the denominator
    $i = 1$;
    $nsign = sign(Od * hx - xd * hO)$;
    while $(nsign \equiv 0)$  $nsign = -sign(S\text{-}facets[h][i{+}{+}])$;
    return $(zsign \equiv nsign)$;
}
```

39. 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** 26 ⟩ +≡
  void *dimension_jump*(**Simplex** $*S$, **list_item** x);

40. Before we do a dimension jump, we compute the new center of the extended origin simplex and set the mapping of the new point approbriate.

⟨ Dimension jump 40 ⟩ ≡
  dcur ++;
  quasi_center += x;
  simplex[*item_x*] = *origin_simplex*;
  dimension_jump(*origin_simplex*, *item_x*);
  clear_visited_marks(*origin_simplex*);

This code is used in section 21.

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

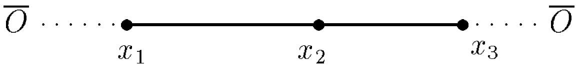

Figure 7: We are in dimension 1

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 four 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 six simplices. For the simplices conv(x_1, x_2) and conv(x_2, x_3) we have added two unbounded simplices below 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 $\overline{O}$.

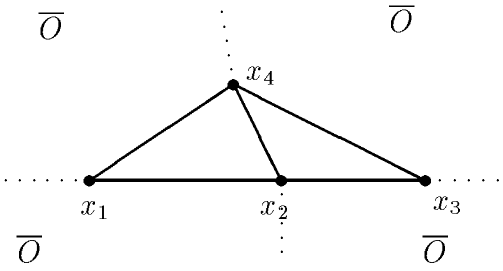

Figure 8: A dimension jump

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 $v_0, \ldots,$ $v_{\text{dcur-1}}, x$. 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 16 ⟩ +≡

```
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 42 ⟩
  }
  /* The neighbor opposite to x might not yet exist. We call 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 43 ⟩
  }
  else {
    ⟨ Complete neighborhood information if F is bounded 44 ⟩
  }
}
```

42. 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 42 ⟩ ≡

```
S_new = S→neighbors[dcur] = new Simplex (dmax);
S_new→this_item = all_simplices.append(S_new);
S→opposite_vertices[dcur] = 0;
S_new→vertices[0] = anti_origin;
for (int k = 1; k ≤ dcur; k++) S_new→vertices[k] = S→vertices[k − 1];
```

This code is used in section 41.

43. 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 $\mathrm{vert}(F) \subset \mathrm{vert}(T)$ and $x \notin \mathrm{vert}(T)$. 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 $\mathrm{vert}(S\_new') = \mathrm{vert}(F') \cup \{\overline{O}\}$. Since $\mathrm{vert}(F) \setminus \{\overline{O}\} \subset \mathrm{vert}(F')$ we have $\mathrm{vert}(F) \subset \mathrm{vert}(S\_new')$. Furthermore $x \notin \mathrm{vert}(S\_new')$. 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 43 ⟩ ≡

```
S→neighbors[dcur] = S→neighbors[0]→neighbors[dcur];
S→opposite_vertices[dcur] = S→opposite_vertices[0] + 1;
```

This code is used in section 41.

44. 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 42. 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 44 ⟩ ≡

```
for (int k = 0; k < dcur; k++) {
  if (S→neighbors[k]→vertices[0] ≡ 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 Ō 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 41.

45. Output Routines.

In order to demonstrate our program, we now add to **Triangulation** a function *show*( ), which draws (in the special case *dmax* ≡ 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*→*vertices*[0] ≠ *anti_origin*) and with $v = 1$ if S is unbounded (i.e., *S*→*vertices*[0] ≡ *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 16 ⟩ +≡
  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
        d_rat_point x = coordinates.contents(S→vertices[v]);
        point a = convert(x);
        for (int e = v + 1; e ≤ dcur; e++) {
              // draw undrawn edges incident to vertex
          d_rat_point y = coordinates.contents(S→vertices[e]);
          point b = convert(y);
              // 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
    d_rat_point x;
    forall (x, coordinates) {
      point a = convert(x);
      W.draw_point(a);
    }
  }
```

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

```
⟨ Member functions of class Triangulation 16 ⟩ +≡
  void Triangulation :: print_all( )
```

```
{
  Simplex *S;
  cout ≪ "\n␣dcur␣" ≪ dcur ≪ "␣origin_simplex:␣";
  if (origin_simplex ≠ nil) cout ≪ origin_simplex→sim_nr;
  else cout ≪ "none";
  cout ≪ "␣quasi_center:␣" ≪ quasi_center ≪ endl;
  forall (S, all_simplices) {
    print(S);
  }
}
```

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

```
⟨ Member functions of class Triangulation 16 ⟩ +≡
  void Triangulation::print(Simplex *S)
  {
    list_item p;
    cout ≪ "\n[" ≪ S→sim_nr ≪
        "]------------------------------------------------------\n";
    for (int i = 0; i ≤ dcur; i++) {
      if (S→vertices[i] ≡ anti_origin) cout ≪ "[xx]␣anti-origin";
      else cout ≪ '[' ≪ order_nr[S→vertices[i]] ≪ ']' ≪
            coordinates.contents(S→vertices[i]);
      cout ≪ "␣[";
      if (S→neighbors[i])
        cout ≪ S→neighbors[i]→sim_nr ≪ "]␣<->␣[" ≪ S→opposite_vertices[i];
      else cout ≪ "*";
      cout ≪ "]␣";
      if ((S→vertices[0] ≠ anti_origin ∨ i ≡ 0) ∧ dcur > 0) {
        cout ≪ ";␣␣facet:␣" ≪ S→facets[i];
        cout ≪ ";␣valid_equations:␣" ≪ S→valid_equations[i];
      }
      cout ≪ endl;
    }
    cout ≪ "Points:␣";
    forall (p, S→points)
      cout ≪ '[' ≪ order_nr[p] ≪ ']' ≪ coordinates.contents(p) ≪ '␣';
    cout ≪ endl;
    cout.flush();
  }
```

48. This is a function that puts the triangulation to a stream by simply writing its defining points to it. That is done in a way that it can be directly used as input for another Triangulation.

```
⟨ Friend functions of class Triangulation 48 ⟩ ≡
  ostream & operator≪(ostream & o, Triangulation &T)
  {
    d_rat_point v;
    int i;
    o ≪ T.dmax ≪ endl ≪ T.coordinates.size( ) ≪ endl;
    forall (v, T.coordinates) {
      for (i = 0; i ≤ T.dmax; i++) o ≪ v.ccoord(i);
      o ≪ endl;
    }
    return o;
  }
```

See also section 49.

This code is used in section 10.

49. This function merges the current triangulation with the Triangulation in the input stream. The exspected format is the same as in the main function.

```
⟨ Friend functions of class Triangulation 48 ⟩ +≡
  istream & operator≫(istream & i, Triangulation &T)
  {
    int dim;
    i ≫ dim;
    if (dim ≠ T.dmax)
      error_handler(20, "chull:␣different␣dimensions␣in␣stream");
          // cannot merge
    i ≫ dim;      // ignore number of points
    d_rat_point v(dim);
    while (¬i.eof( )) {
      i ≫ v;
      T.insert(v);
    }
    return i;
  }
```

50. We should have a function that outputs all outer points of the triangulation, i.e. those located in unbounded simplices. This is useful for constructing a hull describing the same set as the original hull but with a minimal amount of points We print each point only once, so we test if it is already printed.

```
⟨ Member functions of class Triangulation 16 ⟩ +≡
  void Triangulation ::print_extremes(ostream & o) { Simplex *sim;
      list_item p;
      int i; h_array < list_item , bool > printed(false);
      list⟨list_item⟩ points;
forall (sim, all_simplices)
  if (sim→vertices[0] ≡ anti_origin)
    for (i = 1; i ≤ dcur; i++)
          // the zeroth point is the anti_origin
      points.append(sim→vertices[i]);
o ≪ dmax ≪ endl ≪ points.size( ) ≪ endl;
forall (p, points) {
  if (¬printed[p]) {
    for (i = 0; i ≤ dmax; i++) o ≪ coordinates[p].ccoord(i);
    o ≪ endl;
    printed[p] = true;
  }
}
}
```

51. The Deletion Procedure.

We now come to the deletion procedure. First we repeat the overview from section 7.

The global plan is quite simple. When a point x is deleted from R, we change the triangulation T so that in effect x was never added. This is in the spirit of §2. The effect of the deletion of x on the triangulation is easy to describe. All simplices having x as a vertex disappear (If x is not a vertex of T then T does not change). The new simplices of T resulting from the deletion of x all have base facets visible to x, with peak vertices inserted after x. These are the simplices that would have been included if x had not been inserted into R. Let $R(x)$ be the set of points of R that are contained in simplices with vertex x, and also inserted after x. We will, in effect, reinsert the points of $R(x)$ in the order in which they were inserted into R, constructing only those simplices that have bases visible to x. On a superficial level, this describes the deletion process. The details follow. (cf. [3], p. 10)

We may run into six cases when we delete a point x.

- x is not a point of the current hull, so we are done.
- x is the only point of a triangulation. We set the triangulation to an initial state just as if no point was ever added.
- x lies in the interior of some simplex. We delete x from the list of inner points of this simplex.
- Deleting x reduces the dimension of the current hull. We have to reverse the process for inserting a dimension jump (section 39).
- Deleting x leaves a "hole" in the hull to be filled with new simplices induced by points inserted after x.
- x is a dimension jump, but there is another point prevending us from reducing the dimension of the hull. We search this point, declare it as new dimension jump and can go on almost exactly as in the previous case.

The first three cases are rather trivial. We use a **dictionary** *co_index* to look up the position (**list_item**) of a point in the **list** *coordinates*. If we find none, x is not in the hull. Otherwise we get the **list_item** *item_x* of x from the **dictionary**. We will use *item_x* throughout this function as it has more information assigned to it than x and it is unique. x by itself is not always unique, as there may be many points with the same coordinates. By taking the last one of all possible **list_item**s assigned to points with the same coordinates as x we will remove an inner point if x was inserted more than once, improving our running time a bit this way. When we found x in the current hull, we test if it is the only point and reset if triangulation if it is so. Next we test wether x is an inner point by simply looking at it's *position* value. If it is a vertex we handle the remaing cases in section 54.

```
⟨ Member functions of class Triangulation 16 ⟩ +≡
  void Triangulation :: del(const d_rat_point &x)
  {
    dic_item dic_x;
    /* find x in the defining point set. By taking the last inserted point,
    we find inner points when the same point is inserted more often, making
    deletion easier this way. */
    if ((dic_x = co_index.lookup(x)) ≡ nil) return;
          // x not in the hull ⇒ nothing to do
    list_item item_x = co_index.inf(dic_x)→Pop( );
    if (co_index.inf(dic_x)→empty( )) {
      delete co_index.inf(dic_x);
      co_index.del_item(dic_x);
    }
    if (coordinates.length( ) ≡ 1)       // x is the last point in the hull
    {
      ⟨ clean up triangulation 52 ⟩
      return;
    }
    if ((dcur ≡ 0) ∨ (position[item_x] ≠ nil))       // x is not a vertex
    {
      ⟨ delete inner point 53 ⟩
      return;
    }
    ⟨ handle non-triv cases 54 ⟩
  }
```

52. Here we set our triangulation to an initial state as it was before the first point was inserted. We free all memory we allocated dynamically, clear the lists of points and simplices, and set *dcur*, *origin_simplex*, *simplex*, and *position* to their initial state.

```
⟨ clean up triangulation 52 ⟩ ≡
  { Simplex *sim;
  forall (sim, all_simplices) delete sim;
  all_simplices.clear( );
  origin_simplex = nil;
  coordinates.clear( );
  dcur = −1;
  simplex = h_array < list_item , Simplex * > (nil);
  position = h_array < list_item , list_item > (nil);
  order_nr = h_array < list_item , int > (−1);
  }
```

This code is used in section 51.

53. Here we delete an inner point of a simplex. This simplex is directly determined by the pointer *simplex*[*item_x*]. In the same way the position of x in the **list** *points* of the simplex is determined directly by the *position* h_array. So it can be deleted in (exspected) constant time.

⟨ delete inner point 53 ⟩ ≡

```
{
  Simplex *sim;
  sim = simplex[item_x];
  sim→points.del_item(position[item_x]);
  position[item_x] = nil;
  simplex[item_x] = nil;
  order_nr[item_x] = −1;
  coordinates.del_item(item_x);
}
```

This code is used in section 51.

54. We need some additional information about the triangulation when we delete a vertex.

We store some additional information in the insertion process[8]. Each point x_k is stored in exactly one simplex in $\Delta(\pi_k)$ that contains x_k in its closure[9]. The insertion algorithm gives us such a simplex. Furthermore we store the set of dimension jumps in a list[10]. For a vertex x_l of $\overline{\Delta}(\pi)$ let $S(x_l)$ denote the set of simplices with vertex x_l and let $S_k(x_l)$ be the set of simplices in $S(x_l)$ whose peak index is at most k. Furthermore let $R(x_l)$ denote the set of points stored in the simplices in $S(x_l)$ and let $P(x_l)$ be the set of vertices that are opposite to x_l (x and y are *opposite* if there is a facet F of $\overline{\Delta}$[11] such that $S(F \cup x)$ and $S(F \cup y)$ are Simplices of $\overline{\Delta}$). (cf. [2], p. 5)

So we introduce the variables Sx to hold the information of $S(x)$, Rx holding $R(x)$, and Px holding $P(x)$. The only difference is that x is not contained in Rx as it should according to the citation above. We can omit it because we know this fact implicitely. We also want to know if x is a dimension jump (i.e. a vertex of the *origin_simplex*), so we use *is_dj* for that. It may have three values: 0 when x is not a dimension jump, 1 when x is a dimension jump and the new dimension jump lies on the same halfspace as x, and 2 when x is a dimension jump and the new dimension jump lies on the other halfspace as x. Finally, *facet* denotes the index of the facet of a simplex in Sx opposite to x. You could also say that it is the index of *item_x* in the *vertices*-array of a simplex in Sx.

Our case distinction of the last three cases is based on the following:

If x_i was a dimension jump we have either $dim(R \setminus \{x_i\}) = \dim R - 1$ or we get a new dimension jump, say x_j. Deletion of x_i reduces dimension if all other points lie in $\mathrm{aff}(DJ \setminus \{x_i\})$. If $P(x_i) = \emptyset$ and $\mathrm{aff}(R(x_i) \setminus \{x_i\}) = \mathrm{aff}(DJ \setminus \{x_i\})$ then x_i reduces dimension. (cf. [2], p. 6) Otherwise we have to find a new dimension jump and can go on as if we delete a normal vertex. The test is performed in a loop.

---

[8] s. section 27 for actual implementation.

[9] This only holds for non-vertices.

[10] The *vertices* of the *origin_simplex* are such a list.

[11] IMHO should be Δ, see source code for the difference

Putting each point to be tested in a plane equation takes less time then setting up an array and then calling *is_containted_in_affine_hull*( ) (for **d_rat_point**s), which solves a system of linear equations.

```
⟨ handle non-triv cases 54 ⟩ ≡
  { int is_dj = 0;
  list⟨Simplex *⟩ Sx;     // list of pointers to simplices containing x
  h_array < Simplex *,  int > facet(−1);     // facet[Sx]=facet towards x
      list⟨list_item⟩ Rx;
         // pointers into list of points that are in simplices in Sx
      list⟨list_item⟩ Px;     // pointer into list of points opposite to x
      ⟨ set up variables 55 ⟩
      if ((is_dj ≠ 0) ∧ (dcur > 0)) {
        if (Px.empty()) {
          bool hulls_equal = true;
          list_item p;
          forall (p, Rx)
            if (sees_facet(origin_simplex, facet[origin_simplex],
                  coordinates[p]) ≠ 0)  hulls_equal = false;
          if (hulls_equal) {
            ⟨ reduce dimension 58 ⟩
            return;
          }
        }
        ⟨ get new dimension jump 59 ⟩
      }
      ⟨ delete non-dimjmp 63 ⟩
      }
```

This code is used in section 51.

55. First we compute *Sx* by a recursiv function. Then we set *Px* to the list of all points that are opposite to x. Remember that we consider x and y as opposite if there is a facet F of Δ such that $S(F \cup x)$ and $S(F \cup y)$ are simplices of Δ. We say Δ in contrast to $\overline{\Delta}$ as we want the *anti_origin* not to be in *Px*.

The computation of *Rx* is a bit more difficult. To determine $R(x)$, check first wether x is a vertex of the simplices pointed to by x. If not, x is removed and we are done. If so, construct the set $R(x)$ by inspection of all simplices incident to x[12]. This takes time proportional to d times $\mid R(x) \mid$ plus the number of removed simplices. Sorting the points in $R(x)$ by the time of insertion takes time $O(\min\{n, \mid R(x) \mid \log\log n\})$, where the former bound is obtained by bucketsort and the latter bound comes from the use of bounded ordered dictionaries. (cf. [3], p. 12)

Since x may be a vertex of more than one simplex we have to exclude doubles. Additionally we want to insert the inner points of simplices in *Sx*. So we use a LEDA *h_arraynot_in_Rx* of **bool** to filter the doubles out. Now we sort *Rx* according to the insertion time (stored in *order_nr*) using *bucket sort.* Our

---

[12] These are the simplices in *Sx*.

sort-function is an interface to the *bucket_sort*( )-function for LEDA **list**s. We need this for our goal to make the triangulation look as if x was never added, so we have to preserve the insertion order.

```
⟨set up variables 55⟩ ≡
  { int i;
  list_item p;
  Simplex *sim, *neighbor; h_array < list_item , bool > not_in_Rx(true);
      h_array < list_item , bool > not_in_Px(true);
  is_dj = collect_Sx(simplex[item_x], item_x, Sx, facet);
  clear_visited_marks(simplex[item_x]);
  forall (sim, Sx) {
    neighbor = sim→neighbors[facet[sim]];
    p = neighbor→vertices[sim→opposite_vertices[facet[sim]]];
    /* This if-clause is the difference between Δ̄ and Δ. */
    if ((neighbor→vertices[0] ≠ anti_origin) ∧ (not_in_Px[p] ≡ true)) {
      Px.append(neighbor→vertices[sim→opposite_vertices[facet[sim]]]);
      not_in_Px[p] = false;
    }
    for (i = 0; i ≤ dcur; i++) {
      p = sim→vertices[i];
      if ((p ≠ anti_origin) ∧ (p ≠ item_x) ∧ (not_in_Rx[p] ≡ true)) {
        Rx.append(p);
        not_in_Rx[p] = false;
      }
    }
    forall (p, sim→points) {
      Rx.append(p);
      vertex[p] = false;
    }
  }
  sort(Rx); }
```

This code is used in section 54.

56. This function collects the simplices that belong to Sx and sets up the *h_arrayfacet* for the facets towards x of these simplices. It returns 0 if x if not a dimension jump and 1 otherwise. It is intended to be called with *simplex*[*item_x*] as initial S and assumes that *position*[*item_x*] ≡ *nil*, i.e. that x is a vertex and not an inner point.

```
⟨Further member declarations of class Triangulation 26⟩ +≡
  int collect_Sx (Simplex *S, list_item item_x,
          list⟨Simplex *⟩ &Sx, h_array < Simplex *,  int > &facet ) ;
```

57. We first mark the simplex S as visited. Then we test if our current simplex is the *origin_simplex*. If so it is inserted at the top of the list for later convinence, and a flag is set to indicate that x is a dimension jump. All other simplices are simply appended. Then we determine the *facet*[S] as the facet with the same index as x in the *vertices*-array of S. We now recursivly visit all neighbors of S that are not visited yet. We omit the one opposite to x because it cannot contain x as a vertex. The return values of the recursive calls are or'ed.

```
⟨ Member functions of class Triangulation 16 ⟩ +≡
  int Triangulation :: collect_Sx (Simplex *S, list_item item_x,
          list⟨Simplex *⟩ &Sx, h_array < Simplex *, int > &facet )
    {
      int i;
      int is_dj = 0;
      S→visited = true;
      if (S ≡ origin_simplex) {
        is_dj = 1;
        Sx.push(S);      // insert the origin_simplex first
      }
      else Sx.append(S);      // all other simplices as they appear
      for (i = 0; i ≤ dcur; i++) {
        if (S→vertices[i] ≡ item_x) facet[S] = i;
        else if (S→neighbors[i]→visited ≡ false)
          is_dj |= collect_Sx(S→neighbors[i], item_x, Sx, facet);
      }
      return is_dj;
    }
```

58. Here we reduce the dimension of a triangulation. This is easier as one might think.

We simply remove all unbounded simplices that do not have x_i in their vertex set and remove x_i from the vertex set of the remaining simplices. (cf. [2], p. 6)

Furthermore, other things to do are only for purposes of the internal housekeeping. (Mostly remove all references to simplices in Sx from other simplices.) Special care is needed to maintain the neighborhood relation intact when reordering the arrays.

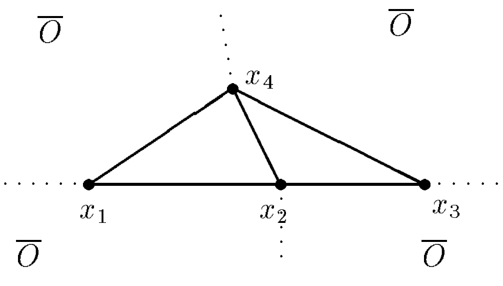

Figure 9: We have this

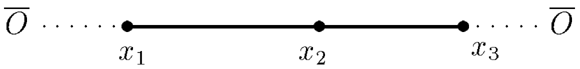

Figure 10: We want this

This is the reversal of the process described in Section 41. You might recognize the figures.

```
⟨ reduce dimension 58 ⟩ ≡
  {
    Simplex *sim, *neighbor;
    int j;

    while (¬Sx.empty()) {
      sim = Sx.pop();
      if (sim→vertices[0] ≠ anti_origin) {
        neighbor = sim→neighbors[facet[sim]];
            // these are to delete, as they were added with x
        all_simplices.del_item(neighbor→this_item);
        delete neighbor;
      }
      for (j = facet[sim]; j < dcur; j++)
          // delete x from the vertex set
      {
        sim→vertices[j] = sim→vertices[j + 1];
        sim→neighbors[j] = sim→neighbors[j + 1];
        sim→opposite_vertices[j] = sim→opposite_vertices[j + 1];
      }
      for (j = 0; j < dcur; j++) {      // adjust the neighborhood relation
        if (facet[sim→neighbors[j]] < sim→opposite_vertices[j])
          sim→opposite_vertices[j]--;
              // here we invalid the plane equations
        sim→valid_equations[j] = −1;
            // and we keep our simplex-pointer valid
        simplex[sim→vertices[j]] = sim;
      }     // restore default state of some arrays
      sim→vertices[dcur] = nil;
      sim→opposite_vertices[dcur] = −1;
      sim→neighbors[dcur] = nil;
      sim→valid_equations[dcur] = −1;
    }
    dcur--;      // some care for the internal management
    quasi_center −= x;
    position[item_x] = nil;
    simplex[item_x] = nil;
    order_nr[item_x] = −1;
    coordinates.del_item(item_x);
  }
```

This code is used in section 54.

59. Here we handle the deletion of a dimension jump that does not reduce the dimension of the current hull because there is another point that can serve as dimension jump. We have to find it, to reconstruct the new *origin_simplex* and so on. A bit of theory as base for our algorithm:
By Lemma 2.1 dimension jumps can be moved to the front of the insertion order. Hence we have

$$\Delta(\pi \setminus \{x_i\}) = \Delta(\sigma\pi \setminus (\{x_i\} \cup DJ))$$

if $x_i \notin DJ$ and

$$\Delta(\pi \setminus \{x_i\}) = \Delta(\sigma'\pi \setminus DJ)$$

if $x_i \in DJ$, where σ and σ' are arbitrary permutations of DJ and $DJ \setminus \{x_i\}$ resp. Hence we can assume $DJ \subset R_i$ wlog. If $x_i \in DJ$ and x_i is a new dimension jump we have by Lemma 2.2

$$\Delta(\pi \setminus \{x_i\}) = \Delta(\sigma'\pi_{i-1}x_j\pi \setminus (R_i \cup DJ \cup \{x_j\})).$$

So we can reinsert x_j first, thereby making sure that all dimension jumps are already inserted when we reinsert the remaining points. The last paragraph descibes reinsertion of a dimension jump.

[...]

Let us finally consider reinsertion of a new dimension jump x_j. Since a new dimension jump is reinserted first we can assume $j = i + 1$ wlog. (cf. [2, p. 6])

So we do (or better prepare everything so that is is done in section 63 here). By making x the "peak" of the *origin_simplex* we can avoid a special treatment.

```
⟨get new dimension jump 59⟩ ≡
  {
    list_item new_dj;     // pointer in coordinates
    list_item p;
    int f, h;
    Simplex *sim;
    hyperplane v;

    ⟨find new dimjump 60⟩
    f = facet[origin_simplex];
    p = origin_simplex->vertices[0];
    origin_simplex->vertices[0] = item_x;
    origin_simplex->vertices[f] = p;
    sim = origin_simplex->neighbors[0];
    origin_simplex->neighbors[0] = origin_simplex->neighbors[f];
    origin_simplex->neighbors[f] = sim;
    h = origin_simplex->opposite_vertices[0];
    origin_simplex->opposite_vertices[0] = origin_simplex->opposite_vertices[f];
    origin_simplex->opposite_vertices[f] = h;
    v = origin_simplex->facets[0];
    origin_simplex->facets[0] = origin_simplex->facets[f];
    origin_simplex→facets[f] = v;
    h = origin_simplex→valid_equations[0];
    origin_simplex→valid_equations[0] = origin_simplex→valid_equations[f];
    origin_simplex→valid_equations[f] = h;
    facet[origin_simplex] = 0;
    if (sees_facet(origin_simplex, facet[origin_simplex],
          coordinates[new_dj]) < 0)
        // if x and new_dj lie in different halfspace
      ⟨ take care for new origin_simplex 62 ⟩      // adjust our center point
    quasi_center −= x;
    quasi_center += coordinates[new_dj];
      // the new_dj replaces the old also with its order_nr
    order_nr[new_dj] = order_nr[item_x];
      // continue as non-dimjump (section 63)
  }
```

This code is used in section 54.

60. Finding another dimension jump is described as follows.
Otherwise we get a new dimension jump x_j where

$$j = min\{k; x_k \notin \mathrm{aff}(DJ \setminus \{x_i\})\}.$$

(cf. [2, p. 6]) We have to build a list of all *possible_points*. It consists of *Rx* and *Px*. They cannot be simply concatenated because of the implementation of LEDA's *list.conc*()-function, which changes both lists involved. So we have to copy them before we concatenate them. Now we sort our *possible_points* according to their insertition order, so we can take the first point in the list that matches the condition.

```
⟨ find new dimjump 60 ⟩ ≡
  {
    list⟨list_item⟩ possible_points;
    list⟨list_item⟩ h;
    possible_points = Rx;
    h = Px;
    possible_points.conc(h);
    sort(possible_points);      // note that x is not in possible_points
    /* find the first point not in the affine hull of the remaining dimjmps */
    do {
      new_dj = possible_points.pop( );
    } while (sees_facet(origin_simplex, facet[origin_simplex],
          coordinates[new_dj]) ≡ 0);
  }
```

This code is used in section 59.

61. If x_i and x_{i+1} are on the same side of $\mathrm{aff}(DJ \setminus \{x_i\})$, then we take the simplices in $S_i(x_i)$, replace x_i by x_{i+1} and add them to $\overline{\Delta}$. B_{i+1} is the collection of facets on the boundary of the union of these simplices that are not opposite to x_{i+1}. (cf. [2, p. 6])

We do so in section 63. The only thing to make sure is that the new *origin_simplex* is the first simplex built. Therefore we inserted it to the top of list *Sx* (s. section 55). This guaranties the first simplex built to fill the hole to be our new *origin_simplex*, with *new_dj* as peak. So it remains nothing to do for now. Setting the new *origin_simplex* can only be done when it is built. (s. section 76)

62. The process for this case is described as follows: If x_i and x_{i+1} are on different sides of aff$(DJ \setminus \{x_i\})$, no simplices are added. (cf. [2, p. 6]) First we mark this case by setting *new_dj* to 2. Now we determine the new *orgigin_simplex* as the simplex that shares the facet opposite to x with the old *origin_simplex*. We also set the *facet* value for the new simplex to its correct value. Then we assign the new *origin_simplex*. After the *origin_simplex* is handled this way, the new facets are collected in section 64, so we do not have to do anything more here.

```
⟨ take care for new origin_simplex 62 ⟩ ≡
  {
    Simplex *sim;
    is_dj = 2;
    sim = origin_simplex→neighbors[facet[origin_simplex]];
    facet[sim] = origin_simplex→opposite_vertices[facet[origin_simplex]];
    origin_simplex = sim;
  }
```

This code is used in section 59.

63. Deleting a Vertex.

This is the general case for deleting a vertex of a simplex. Deleting a vertex affects only simplices in Sx. All others remain unchanged.
Hence it is not necessary to reinsert points that are not in $R(x_i)$. We assume inductively that we have $\Delta(\pi_{k-1} \setminus \{x_i\})$ and $\Delta(\pi \setminus \{x_i\} \cap \Delta(\pi)$ and the set B_{k-1} of facets of $CH(\pi_{k-1} \setminus x_i)$, that are x_i-visible or contain x_i in their affine hull and are new. If $x_i \notin DJ$, B_i is the collection of facets opposite to x_i in the simplices in $S_i(x_i)$. Reinsertion of x_k means to add a simplex $S(F, x_k)$ for every x_k-visible facet F of B_{k-1}. The procedure for finding x_k-visible facets of B_{k-1} is analogous to [3]. (cf. [2] p. 6)
We have a "hole" in the triangulation consisting of the simplices containing x. How it is filled shows figure 11. In this way we handle all points. B corresponds to *newfacets* and *vbfacets*. We actually do not store the real base facet, but the facet of the neighbor that shares the facet we want. This way we avoid using simplices from Sx, of which we not always know if they are already deleted or not. Besides we need the neighbors anyway for updating their neighborhood.

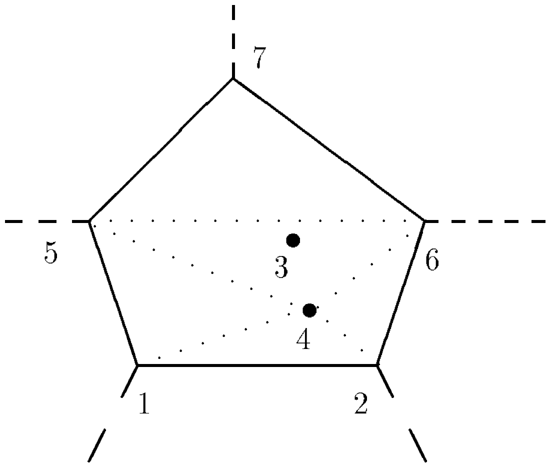

Figure 11: Point 3 is deleted from the triangulation.
Resinserted points are 4 (previously inner point of some simplex) 5, 6, 7.

Unfortunately we could have deleted an outer point, which was limiting the current triangulation. We handle this special case by inserting the anti-origin $\overline{O}$ as the last point because it sees all the remaing uncovered facets, but not storing it in any simplex if it has no visible new base facets. ($\overline{O}$ does per definition not lie in any simplex.) The strategy for this is shown in figure 12.

Another special case are inner points inserted *before* x. These must lie on the facet towards x (or even have the same coordinates as some vertex) because if this would not be the case, they would have let to the creation of a simplex of their own. So they will remain inner points, although they now belong to a different simplex. They have to be reinserted for this reason, as their simplex has x as a vertex and is to be deleted, even if their *order_nr* is lower than *order_nr*$[x]$.

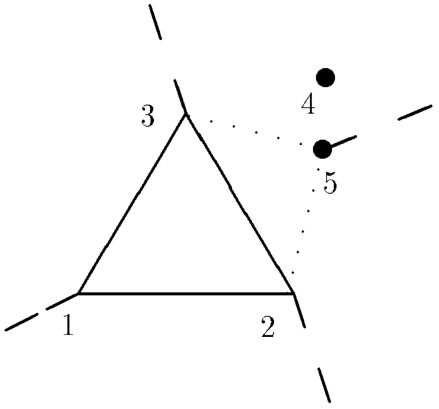

Figure 12: Point 4 is deleted.
Reinsertion of point 5 induces one new bounded and two new unbounded simplices.

Two major informations are to manage. First, the set of facets without a neighbor. Second the set of facets without a neighbor visible by the current point. For both we use similiar constructions: a **list** of pointers to simplices and a **list** of facets (described by their order number) associated with each simplex. So we can traverse the list of uncovered facet by traversing the list of simplices containing uncovered facets and then for each such simplex traversing the list of its uncovered facets. The set of uncovered facets is described by *newfacets* and *visi_facet*, the set of x-visible facets by *vbfacets* and *xvisi_facets*.

We reinsert the points in the same order as we we inserted them originally. Therefore we sorted *Rx*. For each point we find the facets it sees for building simplices out of them. A slightly different strategy applies for the *anti_origin* for that. When we found any visible facets we build the simplices with the current point as peak. Otherwise we try to insert the point in an already built simplex as inner point. When it was a vertex but not the *anti_origin* we look for horizon ridges resulting from this point. Then we go on to the next point.

When all points are inserted we handle the case of a newly built *origin_simplex*. Then we adjust the *simplex*[] pointers for all vertices of the new simplices. Now we have done everything and finally delete x from the list of points.

```
⟨delete non-dimjmp 63⟩ ≡
  { h_array < Simplex *,
      list⟨int⟩ * > visi_facet(nil);
        // stores facet of simplex that shares the facet (neighbor)
      list⟨Simplex *⟩ newfacets;     // to be ”provided”
      h_array < Simplex *,
        list⟨int⟩ * > xvisi_facet(nil);
        list⟨Simplex *⟩ vbfacets;     // visible by current point
        list⟨Simplex *⟩ newsimplices;     //filling the ”hole” of Sx
        list_item p;     // the point we're reinserting
      Simplex *sim;
    ⟨initialize newfacets 64⟩
    Rx.append((list_item) anti_origin);      // $\overline{O}$ for unbounded simplices
    while (¬Rx.empty()) {
      p = Rx.pop();
      if ((order_nr[p] ≥ order_nr[item_x]) ∨
          // only vertices after x are reinserted
      (p ≡ anti_origin) ∨       // or the anti-origin
      (position[p] ≠ nil))      // or inner points
      {
        if ((position[p] ≡ nil) ∨       // vertices
        (order_nr[p] ≥ order_nr[item_x]))
              // inner points inserted after x
          if (p ≠ anti_origin) ⟨collect visible base facets 65⟩
          else ⟨copy newfacets to visible facets 66⟩;
        if (¬vbfacets.empty()) ⟨build simplices for p 67⟩
        else if (p ≠ anti_origin) ⟨insert point in a new simplex 74⟩
        if ((vertex[p] ≡ true) ∧ (p ≠ anti_origin)) ⟨add to newfacets 75⟩
              // see if there are horizon ridges
      }
    }
    ⟨special treatment of newly built origin simplex 76⟩
    ⟨handle lost simplices for points 77⟩
    forall (sim, Sx)       // only unbounded left
    {
      all_simplices.del_item(sim→this_item);
      delete sim;
    }
    simplex[item_x] = nil;
    position[item_x] = nil;
    order_nr[item_x] = −1;
    coordinates.del_item(item_x); }
```

This code is used in section 54.

64. We initialize *newfacets* with the base facets of all simplices with x as peak. Additionally we take some other facets when x is a dimension jump and *new_dj* lies in a different halfspace than x ($is\_dj \equiv 2$). Here B_{i+1} is the collection of facets opposite to x_i in the simplices in $S_{i+1}(x_i)$. (cf. [2], p. 6) This simply means that we have to compare the insertion order number of the peak vertex of each simplex in *Sx* with the order number of the *new_dj*. If it is lower, the facet towards x is taken. Because we made x "peak" of the *origin_simplex* we can handle this as the general case. Remember that we do not store these facet but the facet of the simplex that shares it.

We use some auxiliary **list_item**s to walk through the list, because LEDA objects to modify the current item in an iteration. So we use the item following the current one as iterator to satisfy LEDA. We have to keep this thing in mind when we finish an iteration. Then we have to make the next item the current. We can iterate **list**s only by items, so we have to dereference them to get the information in a list one by one. The same thing also hold for almost all iterations of **list**s in this chapter.

We look up the neighbor of the simplex in *Sx* we wish to store in *newfacets*. *newfacets* constist of pointers to **list**s so we have to create one if we need to. In *visi_facet* we store the index of the point describing our facet. This index is relative to neighbor, because we want to store the facet opposite to the one that we really mean. Next we can delete this simplex because we don't need it anymore. To ease testing we set the neighborhood pointer back to nil for the deleted simplex.

```
⟨initialize newfacets 64⟩ ≡
  {
    list_item s, n;
    Simplex *sim, *neighbor;

    s = Sx.first_item( );
    while (s ≠ nil)      // initially all base facets of x are considered
    {
      n = Sx.next_item(s);
      sim = Sx[s];
      if ((sim→vertices[0] ≡ item_x) ∨
              // all cases (x≡ peak(O))
      ((is_dj ≡ 2) ∧ (sim→vertices[0] ≠ anti_origin) ∧
              // not the anti_origin
      (order_nr[sim→vertices[0]]  ≤
          order_nr[origin_simplex→vertices[facet[origin_simplex]]])))
         // see [2] p. 6
      {
        neighbor = sim→neighbors[facet[sim]];
        if (visi_facet[neighbor] ≡ nil) {
          visi_facet[neighbor] = new  list⟨int⟩;
          newfacets.append(neighbor);
        }
        visi_facet[neighbor]→append(sim→opposite_vertices[facet[sim]]);
        neighbor→neighbors[sim→opposite_vertices[facet[sim]]] = nil;
           // the simplex is not needed anymore
        all_simplices.del_item(sim→this_item);
        delete sim;
        Sx.del_item(s);
      }
      s = n;
    }
  }
```

This code is used in section 63.

65. In this section we scan through the *newfacets* to see if they are visible by the point to be reinserted. If so they are inserted in *vbfacets* and deleted from *newfacets*. Besides we take care of the creation of lists when they are needed and destroy them when they are empty.

The outer **while**-loop scans through all simplices in *newfacets*. The inner **while**-loop scans through all "uncovered" facets by a simplex in *newfacets*. If it is visible by the point indicated by p then we move it to *xvisi_facet*[*sim*], eventually creating a new list when needed. Back in the outer loop we check if we still need the list of visible facets for the current simplex.

```
⟨collect visible base facets 65⟩ ≡
  {     // Glob. Vars: p
    list_item f, n, l, m;
    Simplex *sim;
    int i;
    f = newfacets.first_item();
    while (f ≠ nil)      // forall simplices
    {
      n = newfacets.next_item(f);
      sim = newfacets[f];
      l = visi_facet[sim]→first_item();
      while (l ≠ nil)      // forall facets
      {
        m = visi_facet[sim]→next_item(l);
        i = (*visi_facet[sim])[l];
        if (sees_facet(sim, i, coordinates[p]) < 0) {
          if (xvisi_facet[sim] ≡ nil) {
            xvisi_facet[sim] = new list⟨int⟩;
            vbfacets.append(sim);
          }
          xvisi_facet[sim]→append(i);
          visi_facet[sim]→del_item(l);
        }
        l = m;
      }
      if (visi_facet[sim]→empty()) {
        delete visi_facet[sim];
        visi_facet[sim] = nil;
        newfacets.del_item(f);
      }
      f = n;
    }
  }
```

This code is used in section 63.

66. This is the simpler case of the previous section when the *anti_origin* is reinserted to create new unbounded simplices if we delete an outer point. All newfacets are visible. Separatly copying them is not necessary by using LEDA's *list.conc*()-function. It concatenates the second list after the first by changing some LEDA-internal pointers with the result of *vbfacets* consisting of the old *vbfacets* (empty) followed by the contents of *newfacets*. *newfacets* is empty after this step.

Additionally we collect all bounded simplices left over in *Sx*. The loop is almost the same as in section 65, except that we take every bounded facet in the **if**-clause. Still remember that we have the opposite facet of the one from a simplex in *Sx* in *vbfacets* and *xvisi_facet*.

⟨ copy newfacets to visible facets 66 ⟩ ≡

```
{
  list_item s, n;
  Simplex *sim, *neighbor;
  vbfacets.conc(newfacets);        // assume vbfacets should be empty
  xvisi_facet = visi_facet;
  s = Sx.first_item( );
  while (s ≠ nil) {
    n = Sx.next_item(s);
    sim = Sx[s];
    if (sim→vertices[0] ≠ anti_origin) {      // bounded simplices only
      neighbor = sim→neighbors[facet[sim]];
      if (xvisi_facet[neighbor] ≡ nil) {
        xvisi_facet[neighbor] = new list⟨int⟩;
        vbfacets.append(neighbor);
      }
      xvisi_facet[neighbor]→append(sim→opposite_vertices[facet[sim]]);
      neighbor→neighbors[sim→opposite_vertices[facet[sim]]] = nil;
      all_simplices.del_item(sim→this_item);
      delete sim;
      Sx.del_item(s);
    }
    s = n;
  }
}
```

This code is used in section 63.

67. For all visible base facets we build a new simplex with current point p as peak. Here is the place where we need *vbfacets* and *xvisi_facet*. The outer **while**-loop goes through all simplices containing at least one facet visible to x, the inner **while**-loop walks through all facets belonging to a simplex. The *pop*( )-function of LEDA return the first element of a list and removes it from the list (similiarly to a stack), so *vbfacets* will be empty again after this step.

⟨ build simplices for p 67 ⟩ ≡

```
{     // Glob. Vars: p
  Simplex *sim;
  int f;
```

```
    while (¬vbfacets.empty()) {
      sim = vbfacets.pop();
      while (¬xvisi_facet[sim]→empty()) {
        f = xvisi_facet[sim]→pop();
        ⟨ build simplex 68 ⟩      //and look for neighbors and new facets
      }
      delete xvisi_facet[sim];
      xvisi_facet[sim] = nil;
    }
  }
```

This code is used in section 63.

68. We found a visible facet and build a new simplex with this facet as base and the current point as peak. The only neighbor we know at this time is the neighbor sharing the base facet of the new simplex. Still remember that *sim* and *f* denote the opposite of the base facet of this simplex, so this neighborhood information is filled in easily.

What are the points of the new simplex? The zeroth point is already given by *p*. The remaining points are those of *sim* (the simplex that shares the base face of the new simplex) except the one opposite to the base facet of *new_sim*. This point has the same index as the facet opposite to it, namely *f*. So we know all points of the new simplex and fill them in. The other neighborhood information is filled in later. Now the *new_sim* is added to *all_simplices* and *newsimplices*.

```
⟨ build simplex 68 ⟩ ≡
  {     // Glob. Vars: p, sim, f
    int i, j;
    Simplex *new_sim;
    new_sim = new Simplex (dmax);
    new_sim→vertices[0] = p;
    new_sim→neighbors[0] = sim;
    new_sim→opposite_vertices[0] = f;
    sim→neighbors[new_sim→opposite_vertices[0]] = new_sim;
    sim→opposite_vertices[new_sim→opposite_vertices[0]] = 0;
    for (i = 0, j = 1; i ≤ dcur; i++)
      if (i ≠ new_sim→opposite_vertices[0])
        new_sim→vertices[j++] = sim→vertices[i];
    ⟨ put in neighborhood and newfacets 69 ⟩
    new_sim→this_item = all_simplices.append(new_sim);
    newsimplices.append(new_sim);
  }
```

This code is used in section 67.

69. We have to complete the neighborhood of a newly built simplex. That means we have to look for a neighbor that shares a facet with the new simplex for every facet except the base facet. If we find none then this facet will get its neighbor later when we come here again for another simplex just been built. We mark this state in an **array** *newf* to add it to *newfacets* later.

First we try to find a neighbor among the simplices in *Sx*. There are two cases to distinguish. When we try to complete a bounded simplex we use a different strategy than for an unbounded simplex. It is possible for the new simplex to have another facet from this set that was not visible by its peak point, e.g the last simplex built to fill a hole in the triangulation. Next we test if a simplex recently built to reconstruct the triangulation has a facet in common with the new simplex if no neighbor was found in the previous try.

When a facet is left uncovered, it is "new" and added to *newfacets*. Note that we start with facet 1 as the base facet is always covered by the step in the previous section.

Now that we know all our neighbors we can update *newfacets* for the next simplex that is to built.

```
⟨ put in neighborhood and newfacets 69 ⟩ ≡
  {    // Glob. Vars: new_sim
    int i;
    array⟨bool⟩ newf (0, dcur);
       // indicates whether new_sim already has a neighbor opposite to vertex i
    newf [0] = false;
    for (i = 1; i ≤ dcur; i++) {
      newf [i] = true;
      if (new_sim→vertices[0] ≠ anti_origin) ⟨ test edge facets 70 ⟩
      else ⟨ look for neighborhood of unbounded simplex 71 ⟩
      if (newf [i] ≡ true)     // no edge facet coverd
        ⟨ test newfacets 72 ⟩
    }
    ⟨ collect new facets 73 ⟩
  }
```

This code is used in section 68.

70. Here we look for the neighbor's data in every "old" simplex in *Sx* to copy them. If the new and the old simplex have the same facet looking towards x, we can copy the neighborhood data and the old simplex is not needed anymore, so we delete it. For testing we use the *facets_equal*( )-function defined in section 90. i indicates the facet for which we are searching a neighbor. When we found one, we copy its relevant data in the *new_sim*'s structure and set *newf* [i] to *false*.

```
⟨ test edge facets 70 ⟩ ≡
  {    // Glob. Vars: i, new_sim, newf
    list_item s, n;
    Simplex *sim;

    s = Sx.first_item( );
    while (s ≠ nil)     // first the egdes of the "hole"
```

```
 {
   n = Sx.next_item(s);
   sim = Sx[s];
   if (facets_equal(new_sim, i, sim, facet[sim]))      // see Section 90
   {
     new_sim->neighbors[i] = sim->neighbors[facet[sim]];
     new_sim->opposite_vertices[i] = sim->opposite_vertices[facet[sim]];
     newf[i] = false;
     all_simplices.del_item(sim->this_item);
     delete sim;
     Sx.del_item(s);
   }
   s = n;
 }
}
```

This code is used in section 69.

71. We have to set up the neighborhood of a newly inserted unbounded simplex. We look at every facet of every simplex remaining in *Sx* if it has a facet (exept its base) in common with the *new_sim*, and if this is true we can copy the neighborhood information. Unfortunately we cannot delete *sim* from *Sx* after that because *sim* may have more than one facet from which we have to derive the neighborhood relations. (see figure 13.) The last thing to do is to set *newf*[*i*] to *false*.

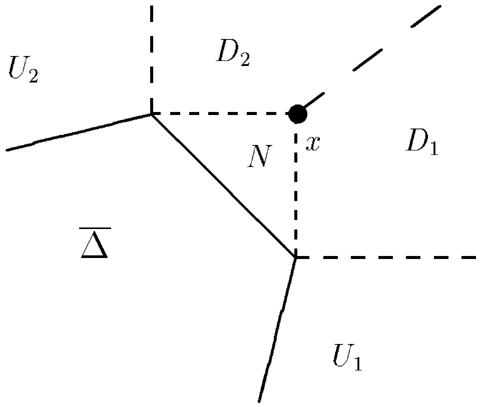

Figure 13: x is deleted, causing the unbounded simplices D_1 and D_2 to disappear. N is the new unbounded Simplex replacing D_1 and D_2, for which the neighborhood is to be obtained.

⟨look for neighborhood of unbounded simplex 71⟩ ≡

```
{     // Glob. Vars i, new_sim, newf
  int j;
  Simplex *sim;
```

```
    forall (sim, Sx) {
      for (j = 1; j ≤ dcur; j++)
        if (facets_equal(new_sim, i, sim, j))      // see Section 90
        {
          new_sim→neighbors[i] = sim→neighbors[j];
          new_sim→opposite_vertices[i] = sim→opposite_vertices[j];
          newf[i] = false;
        }
    }
  }
```

This code is used in section 69.

72. Now we look at every facet in *newfacets* and *visi_facet* for a facet shared with the *i*-th facet of *new_sim*. We use again the double **while**-loop with the auxiliary **list_item**s as in prior sections to compare every facet in these lists with the *i*-th facet of *new_sim*. When we found a neighbor, we put its data in the *new_sim*'s structure and set *newf*[*i*] to *false*. Additionally we can delete this facet from *visi_facet* because it now has a neighbor and is therefore no longer a possible base facet for later built simplices. We may delete *visi_facet*[*sim*] when it is empty after the inner loop.

⟨ test newfacets 72 ⟩ ≡

```
  {     // Glob. Vars: new_sim, i
    list_item s, n, l, m;
    Simplex *sim;
    int j;
    s = newfacets.first_item( );
    while (s ≠ nil) {
      n = newfacets.next_item(s);
      sim = newfacets[s];
      l = visi_facet[sim]→first_item( );
      while (l ≠ nil) {
        m = visi_facet[sim]→next_item(l);
        j = visi_facet[sim]→contents(l);
        if (facets_equal(new_sim, i, sim, j))      // see Section 90
        {
          new_sim→neighbors[i] = sim;
          new_sim→opposite_vertices[i] = j;
          newf[i] = false;
          visi_facet[sim]→del_item(l);
        }
        l = m;
      }
      if (visi_facet[sim]→empty( )) {
        delete visi_facet[sim];
        visi_facet[sim] = nil;
        newfacets.del_item(s);
      }
      s = n;
    }
  }
```

This code is used in section 69.

73. We have to run through *newf* to test whether a facet is new. If this is the case we append it to the list of *newfacets* and *visi_facet* (if needed creating one for the *new_sim*). Otherwise we can now complete the neighborhood pointers from the neighbor back to the simplex itself. Setting the pointers back to *new_sim* each time may be overkill, but setting the right pointer never is definitivly worse then setting it twice.

⟨ collect new facets 73 ⟩ ≡

```
  {     // Glob. Vars: newf, new_sim
    int i;
    for (i = 1; i ≤ dcur; i++)
      if (newf[i] ≡ true) {
        new_sim→neighbors[i] = nil;
        if (visi_facet[new_sim] ≡ nil) {
          visi_facet[new_sim] = new list⟨int⟩;
          newfacets.append(new_sim);
        }
        visi_facet[new_sim]→append(i);
      }
      else {
        new_sim→neighbors[i]→neighbors[new_sim→opposite_vertices[i]] =
            new_sim;
        new_sim→neighbors[i]→opposite_vertices[new_sim→opposite_vertices[i]] =
            i;
      }
  }
```

This code is used in section 69.

74. Now we are in the situation that we did not find any visible facets for a reinserted point. So we have to find a simplex already built (list *newsimplices*) the point lies in and append it to its list of interior points. We try this only for non-vertices, as we would sometimes insert a point twice otherwise.

A special case are unbounded simplices. We enter this test only if we know the current point is an interior point of some simplex. So when it lies in an unbounded simplex, it must lie on the base facet. (We cannot compute a plane equation for the other facets of an unbounded simplex.) The next thing is that we do not want an unbounded simplex to have interior points, so we attach it to its neighbor that shares the base facet.

When we found no simplex containing this point it will be built soon, so we try the whole procedure again at later time and append the point again to *Rx*.

⟨ insert point in a new simplex 74 ⟩ ≡

```
{     // Glob. Vars: p
  bool in = false;
  Simplex *sim;
  int i;
  if (position[p] ≠ nil) {      // a vertex will always be a vertex
    newsimplices.init_iterator();
    newsimplices.move_iterator(forward);
    while ((¬in) ∧ (newsimplices.get_iterator() ≠ nil)) {
      in = true;
      sim = newsimplices[newsimplices.get_iterator()];
      if (sees_facet(sim, 0, coordinates[p]) < 0) in = false;
      if (sim→vertices[0] ≠ anti_origin)
            // unbounded simplices only have one computable facet
        for (i = 1; (i ≤ dcur) ∧ (in ≡ true); i++)
          if (sees_facet(sim, i, coordinates[p]) < 0)
              //see also Section 28
            in = false;
      newsimplices.move_iterator(forward);
    }
    if (in) {
      if (sim→vertices[0] ≡ anti_origin) sim = sim→neighbors[0];
      position[p] = sim→points.append(p);
      simplex[p] = sim;
    }
    else     // Try again later to fit in this interior point
      Rx.append(p);
  }
}
```

This code is used in section 63.

75. We found no base facet visible for this vertex and no simplex containing it. So we look in *Sx* for all simplices with *p* as peak and add them to *newfacets*. This procedure looks similiar to the one in section 64.

⟨ add to newfacets 75 ⟩ ≡

```
{     // Glob. Vars: p
  list_item s, n;
  Simplex *sim, *neighbor;
  s = Sx.first_item();
  while (s ≠ nil) {
    n = Sx.next_item(s);
    sim = Sx[s];
    if (sim→vertices[0] ≡ p)      // if p adds a horizon ridge
    {
      neighbor = sim→neighbors[facet[sim]];
```

```
        if (visi_facet[neighbor] ≡ nil) {
          visi_facet[neighbor] = new list⟨int⟩;
          newfacets.append(neighbor);
        }
        visi_facet[neighbor]→append(sim→opposite_vertices[facet[sim]]);
        neighbor→neighbors[sim→opposite_vertices[facet[sim]]] = nil;
        all_simplices.del_item(sim→this_item);
        delete sim;
        Sx.del_item(s);
      }
      s = n;
    }
  }
```

This code is used in section 63.

76. This is the special case where we have a new *origin_simplex* that has just been built. We cannot assign this value before as it is newly computed.

```
⟨ special treatment of newly built origin simplex 76 ⟩ ≡
  {
    if (is_dj ≡ 1) {
      origin_simplex = newsimplices.head();
    }
  }
```

This code is used in section 63.

77. We have to correct the *simplex*[ ]-array for the vertices of the newly built simplices. The interior points were handled during their insertion (section 74). We cannot do this earlier because sometimes the simplex belonging to a vertex did not exist yet at its insertion time. We do not want to attach any points or vertices to unbounded simplices. We avoid this inconvenience by assigning their vertices to their bounded neighbor (nr. 0). Since they are shared, this is correct. We have to reset *position* in case we made a former interior point to a vertex, to keep the data structure consistent.

```
⟨ handle lost simplices for points 77 ⟩ ≡
  {
    Simplex *sim, *sim2;
    int i;
    forall (sim, newsimplices) {
      if (sim→vertices[0] ≠ anti_origin)      // bounded simplex
        sim2 = sim;
      else      // unbounded simplex
        sim2 = sim→neighbors[0];      // th bounded neighbor
      for (i = 0; i ≤ dcur; i++) {
        simplex[sim2→vertices[i]] = sim2;
        position[sim2→vertices[i]] = nil;
      }
    }
  }
```

This code is used in section 63.

78. You may have noticed that this chapters handles the case of deleting a vertex in a different way than described in [3]. I decided to implement a straight forward algorithm that does everything necessary only when needed because the algorithm in [3] is even more complicated than this one. They use the following data structure:

(A) a triangulation T which consists of $T(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{k-1})$ and the simplices in $T(\pi) \cap T(\pi \setminus i)$,

(B) the set $B = B_{k-1}$, its neighborhood graph, and for each facet $F \in B$ the simplex in T incident to F and the equation of the hyperplane supporting F,

(C) a dictionary for set set of ridges in B.

(cf. [3], p. 13)

Additionally they use a variation of segment walk to find the visible facets for a point in constant time. I use a linear search here, so the algorithm implemented here runs by a factor of the size of the list of new facets slower than it is possible. For the sake of simplicity I renounced implementing this segment walk. Therefore it would be necessary not to delete simplices of Sx after getting the information about facets out of them but to preserve them as they are needed for the segment walk. This would probably increase the running time by a considerably amount.

79. Support Functions.

These are functions for convenience or easier handling of some aspects of the triangulation. They are very short and easy to implement, so they are not worth a chapter on each own and gathered here together.

80. The following functions allow more comfort for input e.g. from mouse. It chooses the point in the current hull which is closest to a given point x.

```
⟨Further member declarations of class Triangulation 26⟩ +≡
  vector make_vector(const d_rat_point &p);
```

81. We use **vector**s to make life easier, but at the cost of some precision. Although when you work with a mouse it is to hard to hit the same pixel twice and for such cases this function is intended.

```
⟨Member functions of class Triangulation 16⟩ +≡
  vector Triangulation :: make_vector(const d_rat_point &p)
  {
    int i;
    vector v(p.dim());
    for (i = 1; i ≤ p.dim(); i++) v[i − 1] = p[i].todouble()/p[0].todouble();
    return v;
  }
  d_rat_point Triangulation :: find_closest_point(const d_rat_point &x)
  {
    double dist;
    d_rat_point p(dmax), cp(dmax);
    vector d(dmax);
    dist = MAXDOUBLE;    // approximately +∞
    forall (p, coordinates) {
      d = make_vector(p − x);
      if (d.length() < dist) {
        dist = d.length();
        cp = p;
      }
    }
    return cp;
  }
```

82. Hyperplanes are only constructed when needed. These functions take care of it.

```
⟨Further member declarations of class Triangulation 26⟩ +≡
  int sees_facet(Simplex *S, int f, const d_rat_point &x);
  void compute_plane(Simplex *S, int j);
```

83. A hyperplane take all points of a simplex except one as its base. This special point is the point describing it (i.e., has the same index). It lies in the positive halfspace of the hyperplane. When this point is the *anti_origin*, we cannot compute with it. Therefore we take the origin ($O = quasi\_center/(dcur+1)$) and place it on the negative side of the hyperplane. When we are not in maximum dimension, the normal vector has to lie in the affine hull of the current hull. This means, it is a linear combination of the spanning vectors derived from the vertices of the current simplex. When this is an unbounded simplex, we take the origin instead of *anti_origin*. It is affinely independent of all other vertices because it lies in the interior of the hull and the other points on the edge.

When it is possible we take the shortcut of using the precomputed plane of the neighbor simplex. It describes the opposite space but has the same equation, so we simply copy it with the sign of the coefficients inverted.

sees_facet( ) additionally returns the side of the hyperplane where x lies. Often this is the only interest for computing a hyperplane.

⟨ Member functions of class Triangulation 16 ⟩ +≡

```
int Triangulation :: sees_facet(Simplex *S, int f, const d_rat_point &x)
{
  if (S→valid_equations[f] ≠ dcur) compute_plane(S, f);
  return which_side(S→facets[f], x);
}
void Triangulation :: compute_plane(Simplex *S, int f)
{
  if (S→valid_equations[f] ≠ dcur) {
    Simplex *NS;
    int o;
    NS = S→neighbors[f];
    o = S→opposite_vertices[f];
    if ((NS ≠ nil) ∧ (NS→valid_equations[o] ≡ dcur))      // copy the reverse
    {
      S→facets[f] = NS→facets[o].reverse( );
    }
    else     // compute ourselves
    {
      int i, j;
      array⟨d_rat_point⟩ P(1, dcur);
      j = 1;
      for (i = 0; i ≤ dcur; i++)
        if (i ≠ f) P[j++] = coordinates[S→vertices[i]];
      if (dcur ≡ dmax) {
        if (S→vertices[f] ≡ anti_origin)
          S→facets[f] = hyperplane(P, quasi_center/(dcur + 1), −1);
        else S→facets[f] = hyperplane(P, coordinates[S→vertices[f]], 1);
      }
      else {
          array⟨integer_vector⟩ N(1, dcur);
          d_rat_point p0;
          if (S→vertices[f] ≡ anti_origin)
            p0 = quasi_center/(dcur + 1);     // works, we are convex!
          else p0 = coordinates[S→vertices[f]];
          j = 1;
          for (i = 0; i ≤ dcur; i++)
            if (i ≠ f)
              N[j++] = make_direction_from_point(coordinates[S→vertices[i]] −
                  p0);
          if (S→vertices[f] ≡ anti_origin)
            S→facets[f] = hyperplane(P, N, p0, −1);
          else S→facets[f] = hyperplane(P, N, p0, 1);
        }
      }
      S→valid_equations[f] = dcur;
    }
  }
```

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

```
⟨ Further member declarations of class Triangulation 26 ⟩ +≡
  bool is_dimension_jump(const d_rat_point &x);
```

85. 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 *is_contained_in_affine_hull*( ), which gets as argument an array of all the vectors to test.

```
⟨ Member functions of class Triangulation 16 ⟩ +≡
  bool Triangulation :: is_dimension_jump(const d_rat_point &x)
  {
    array⟨d_rat_point⟩ A(0, dcur);
    int i;
    for (i = 0; i ≤ dcur; i++)
      A[i] = coordinates.contents(origin_simplex→vertices[i]);
    return ¬is_contained_in_affine_hull(A, x);
  }
```

86. Sometimes it is useful to know whether a point is contained in the convex hull. The tests to perform are similiar to those when inserting a point, but no action is taken. The first case is an empty hull. It surely has no points in it. The next case is that the point to test would be a dimension jump. If so it can not be a member of the hull. When we have come this far we know that x lies in the affine hull of the triangulation. We call *find_visible_facets*( ) to see whether it belongs to the interior of the hull. If not we have to *clear*( ) the list and return false.

⟨ Member functions of class Triangulation 16 ⟩ +≡

```
bool Triangulation :: member(const d_rat_point &x)
{
  if (dcur ≡ −1) return false;      // no points – no members
  if (dcur < dmax)
    if (is_dimension_jump(x)) return false;
  find_visible_facets(x);
  if (visible_simplices.empty( )) return true;
  visible_simplices.clear( );
  return false;
}
```

87. We sometimes want to sort lists of **list_item**s describing points according to their *order_nr*. We use LEDA's *bucket_sort*( ) for **list**s for this. We have to declare the order function needed by bucketsort and the *sort*( ) function using it. *curr_tria* is the replacement for the **this**-pointer that we cannot use, because the *order_nof* function is called from within *bucket_sort*( ) and has to be static for this.

⟨ Further member declarations of **class Triangulation** 26 ⟩ +≡

```
static int order_nof(const list_item &co);
static Triangulation ∗curr_tria;
void sort(list⟨list_item⟩ &L);
```

88. This is the implementation. Static members are necessary because they are called from within *bucket_sort*( ) and cannot determine the object instance (which triangulation) they belong to.

⟨ Member functions of class Triangulation 16 ⟩ +≡

```
Triangulation ∗Triangulation :: curr_tria;
    // we need to reserve space for it
int Triangulation :: order_nof(const list_item &co)
{
  return curr_tria→order_nr[co];
}
void Triangulation :: sort(list⟨list_item⟩ &L)
{
  curr_tria = this;
  L.bucket_sort(0, co_nr, order_nof);
}
```

89. This function tests whether two facets of two simplices are equal.

```
⟨Further member declarations of class Triangulation 26⟩ +≡
  bool facets_equal(Simplex ∗sim1, int facet1, Simplex ∗sim2, int facet2);
```

90. Two facets are equal if they contain the same points. This test is an easy run through the *vertices*[ ] array in each simplex omitting the vertex corresponding to the facet being compared. This takes only $O(n^2)$ because we are only comparing pointers (**list_item**) to vectors, not whole vectors themselves. Note also that we do not use the **operator**≡ for **hyperplane**s because two facets may have the same affine hull (**hyperplane**) but defined by different sets of points.

```
⟨Member functions of class Triangulation 16⟩ +≡
  bool Triangulation::facets_equal(Simplex ∗sim1, int facet1, Simplex
          ∗sim2, int facet2)
  {
    int i, j;
    for (i = 0; i ≤ dcur; i++)
      if (i ≠ facet1) {
        for (j = 0; j ≤ dcur; j++) {
          if (j ≡ facet2) continue;
          if (sim1→vertices[i] ≡ sim2→vertices[j]) break;
        }
        if (j ≡ dcur + 1)      // not found
          return false;
      }
    return true;     // all found
  }
```

91. For test purposes we want to know all points inserted by now. So here they are.

```
⟨Member functions of class Triangulation 16⟩ +≡
  list⟨d_rat_point⟩ Triangulation::points()
  {
    return coordinates;
  }
```

92. The Demo Program.

We described how to use the demo programm earlier, but we repeat it here for convenience.

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. (default)
- **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 information 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. (default)

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 92 ⟩ ≡
#include "chull.h"
#include <time.h>      // we use time( ) and its relatives
  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 93.

This code is used in section 11.

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

```
⟨ Main program 92 ⟩ +≡
  enum input_method {
    MOUSE, KEYS, INPUTFILE
  };
  main(int argc, char **argv)
  {
    ⟨ Read the command line 94 ⟩
    ⟨ Process the data 95 ⟩;
  }
```

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

```
⟨ Read the command line 94 ⟩ ≡
  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 walk
  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 93.

95. Here is how we process the data.

⟨ Process the data 95 ⟩ ≡

```
  /* 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 96 ⟩
    }
    exit(0);
    break;
```

```
case KEYS: cout ≪ "Dimension␣of␣coordinate␣vectors:␣";
  cin ≫ dimension;
  break;
case INPUTFILE: file_in ≫ dimension;
  file_in ≫ number_of_points;
  break;
}
⟨ Input from keyboard or file 97 ⟩
```

This code is used in section 93.

96. 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. With a click of the middle mouse button the point next to the mouse pointer is deleted. The triangulation will be updated immediately. A click of the right mouse button ends the program. The input points are automatically logged to the file `chull.pts`.

```
⟨ Input from mouse 96 ⟩ ≡
  window W;
  W.clear( );
  Triangulation T(2, m);
     // we are working in the plane with search method m
  array⟨integer⟩ L(0, 2);      // for creation of x
  double a, b;
  file_ostream protocol("chull.pts");
  int mouse = 0;      // variable to indicate which mouse button was pressed
  L[0] = 100;
  time_t now = time(nil);      // What's the time, please?
  protocol ≪ "Chull␣protocol␣from␣" ≪ ctime(&now) ≪ endl;
     // write heading
  do {
    mouse = W.read_mouse(a, b);
       // read the window coordinates into a and b
    L[1] = a ∗ 100;
    L[2] = b ∗ 100;
    d_rat_point x(L, homogeneous);
    if (mouse ≡ 1) {      // left button pressed
      protocol ≪ "insert␣" ≪ x ≪ endl;
      T.insert(x);
    }
    if (mouse ≡ 2) {      // middle button pressed
      x = T.find_closest_point(x);
      protocol ≪ "delete␣" ≪ x ≪ endl;
      T.del(x);
    }
    W.clear( );
    T.show(W);
    if (print_simplices) T.print_all( );
  } while (mouse ≠ 3);       // while mouse click is not the right button
  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 95.

97. 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`). After the the input is read in, the triangulation is cut down piece by piece by deleting the inserted points in random order in the inputs is not drawn onto the screen or or with every mouseclick by deleting the point nearest to the center of the window.

```
⟨ Input from keyboard or file 97 ⟩ ≡
  Triangulation T(dimension, m);
  d_rat_point x(dimension);
  int mouse;
  array⟨integer⟩ L(0, 2);
  L[0] = 100;
  if (dimension ≡ 2 ∧ ¬suppress) {
    window W;
    W.clear( );
    do {
      if (read_from ≡ KEYS) cin ≫ x;
      else file_in ≫ x;
      if (¬(file_in.eof( ) ∨ cin.eof( ))) T.insert(x);
      if (¬suppress ∧ draw_all) {
        W.clear( );
        T.show(W);
      }
      if (print_simplices) T.print_all( );
    } while (¬(file_in.eof( ) ∨ cin.eof( )));
    W.clear( );
    T.show(W);
    cout  ≪  "Press␣the␣right␣but\
        ton␣in␣the␣drawing␣window␣to␣terminate,\n" ≪
        "or␣middle␣button␣to␣delete␣a␣random␣point.\n";
    cout.flush( );
    while ((mouse = W.read_mouse( )) ≠ 3) {
      if (mouse ≡ 2) {
        L[1] = 50;
        L[2] = 50;
        d_rat_point x(L, homogeneous);
        x = T.find_closest_point(x);
        T.del(x);
        W.clear( );
        T.show(W);
      }
    }
  }
  else {
    do {
      if (read_from ≡ KEYS) cin ≫ x;
      else file_in ≫ x;
      if (¬(file_in.eof( ) ∨ cin.eof( ))) T.insert(x);
      if (print_simplices) T.print_all( );
    } while (¬(file_in.eof( ) ∨ cin.eof( )));
    list⟨d_rat_point⟩ points = T.points( );
    points.permute( );
    cout ≪ "\n␣insert␣finished,␣now␣deleting␣points\
        ␣in␣random␣order\n\n";
    cout.flush( );
    forall (x, points) T.del(x);
  }
  cout ≪ endl ≪ "Searched␣Simplices:␣" ≪ T.searched_simplices ≪ endl;
  cout ≪ "Simplices␣created:␣" ≪ T.created_simplices( ) ≪ endl;
  cout.flush( );      // only for statistical reasons
```

This code is used in section 95.

98. Running Time Measurements.

We show some running times of our program. The equipment used for this measures was a PC based on an Intel Pentium processor with 75MHz and 8MB ram running Linux.

The tests were performed on a random input created by the following program.

```
⟨rand.c 98⟩ ≡
  /* The Random Vector Generator */
#include <stdlib.h>
#include <time.h>
  typedef unsigned long int ulong;
  int main(argc, argv)
      int argc;
      char **argv;
  {
    ulong dim;
    ulong num;
    ulong i, j;
    if (argc ≠ 3) {
      printf("%s:␣<dim>␣<num>\n", argv[0]);
      return 10;
    }
    dim = strtoul(argv[1], Λ, 10);
    num = strtoul(argv[2], Λ, 10);
    srandom(time(Λ));
    printf("%ld\n%ld\n", dim, num);
    for (i = 0; i < num; i++) {
      for (j = 0; j < dim; j++) printf("%10ld␣␣", random());
      printf("\n");
    }
  }
```

99. The time used was measured using the UNIX '`time`'-command in the following way: '`time -p chull f testinput n s <Method>`', where `<Method>` was set to `V`, `M`, and `S` respectively for the same inputfile. The times shown in the following figure are the '`user`' time in seconds. The '`real`' and the '`sys`' time depend very much on swapping, which the system did heavily during the tests.

dim 2	time (in `s`) searched simplices		
points	V	M	S
256	0.38	0.34	0.64
	2108	2050	1689
512	0.60	0.57	1.01
	3421	3344	2710
1024	1.19	1.14	2.03
	7004	6883	5515
2048	2.33	2.19	4.02
	14617	14472	10940
4096	4.87	4.61	8.44
	30617	30350	23204
8192	8.59	7.99	15.04
	51323	51084	41497
16384	23.67	21.87	44.42
	160550	160033	121246

dim 3	time (in `s`) searched simplices		
points	V	M	S
256	3.03	2.50	2.9
	7290	6360	3169
512	5.94	5.09	5.72
	16498	14673	6609
1024	9.65	8.45	9.64
	27502	25125	11736
2048	16.90	15.07	16.75
	50710	47181	21096
4096	38.93	35.75	37.00
	126708	121531	48827
8192	74.15	68.25	72.91
	246240	238783	98783
16384	161.68	151.45	150.50
	556527	545401	200879

dim 4	time (in `s`) searched simplices		
points	V	M	S
128	8.88	7.08	6.09
	8921	6382	2703
256	19.27	15.20	12.31
	23992	17484	5777
512	40.80	32.16	24.34
	57648	45781	12323
1024	80.54	65.25	43.97
	123678	102260	23678

dim 5	time (in `s`) searched simplices		
points	V	M	S
128	69.17	48.28	34.36
	33461	18606	7144

Running Times. Each test was run three times on a different input set. The tables show the average values.

There are some things to observe. First, the *modified-visibility*-method is always slightly faster then the unchanged *visibility*-method. Next, the *segment-walk*-method searches significantly fewer simplices than the *modified-visisibility*-method, which searches slightly fewer simplices than the *visibility*-method. However this results not in a better running time for the *segment-walk*-method before some size of the triangulation is reached.

100. Useful Literature.

References

[1] C. Bradford Barber, David P. Dobkin, Hannu Huhdanpaa "The quickhull Algorithm for Convex Hull" Geometry Center Technical 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", available from Chr. Burnikel at the Max-Planck-Institut für Informatik, Saarbrücken

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

[4] 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`

[5] Kurt Mehlhorn, "Sources and Documentation of the Data Types `d_rat_point` and `hyperplane`", available from the author at the Max-Planck-Institut für Informatik, Saarbrücken

[6] Michael Müller, Joachim Ziegler: "An Implementation of a Convex Hull Algorithm, Version 1.0", Technischer Bericht, MPI-I-94-105, Max-Planck-Institut für Informatik, Saarbrücken, 1994.

[7] Stefan Näher, Christian Uhrig: "The LEDA User Manual (Version R 3.2)", Technischer Bericht, MPI-I-95-1-002, Max-Planck-Institut für Informatik, Saarbrücken, 1995.

List of Figures

Index

List of Refinements

⟨ Add a new unbounded simplex 42 ⟩ Used in section 41.
⟨ Complete neighborhood information if F is bounded 44 ⟩ Used in section 41.
⟨ Complete neighborhood information if F is unbounded 43 ⟩ Used in section 41.
⟨ Compute the arrays fx and fO and test whether $x \in S$ 33 ⟩ Used in section 32.
⟨ Decide whether $>^{\sigma}$ is $<$ or $>$ 37 ⟩ Used in section 36.
⟨ Dimension jump 40 ⟩ Used in section 21.
⟨ For each horizon ridge add the new simplex 24 ⟩ Used in section 23.
⟨ Friend functions of class Triangulation 48, 49 ⟩ Used in section 10.
⟨ Further member declarations of **class Triangulation** 26, 34, 39, 56, 80, 82, 84, 87, 89 ⟩ Used in section 15.
⟨ Go to the next Simplex on the ray $\overrightarrow{Ox}$ 35 ⟩ Used in section 32.
⟨ Header files to be included 12, 13, 14 ⟩ Used in section 9.
⟨ Initialize the triangulation 22 ⟩ Used in section 21.
⟨ Input from keyboard or file 97 ⟩ Used in section 95.
⟨ Input from mouse 96 ⟩ Used in section 95.
⟨ Main program 92, 93 ⟩ Used in section 11.
⟨ Member functions of class Triangulation 16, 17, 18, 21, 27, 28, 29, 30, 31, 32, 36, 38, 41, 45, 46, 47, 50, 51, 57, 81, 83, 85, 86, 88, 90, 91 ⟩ Used in section 10.
⟨ Non-dimension jump 23 ⟩ Used in section 21.
⟨ Process the data 95 ⟩ Used in section 93.
⟨ Read the command line 94 ⟩ Used in section 93.
⟨ Update the neighborhood relationship 25 ⟩ Used in section 23.
⟨ add to newfacets 75 ⟩ Used in section 63.
⟨ build simplex 68 ⟩ Used in section 67.
⟨ build simplices for p 67 ⟩ Used in section 63.
⟨ `chull.h` 9 ⟩
⟨ class Simplex 19, 20 ⟩ Used in section 10.
⟨ class Triangulation 15 ⟩ Used in section 9.
⟨ clean up triangulation 52 ⟩ Used in section 51.
⟨ collect new facets 73 ⟩ Used in section 69.
⟨ collect visible base facets 65 ⟩ Used in section 63.
⟨ copy newfacets to visible facets 66 ⟩ Used in section 63.
⟨ delete inner point 53 ⟩ Used in section 51.
⟨ delete non-dimjmp 63 ⟩ Used in section 54.
⟨ find new dimjump 60 ⟩ Used in section 59.
⟨ get new dimension jump 59 ⟩ Used in section 54.
⟨ handle lost simplices for points 77 ⟩ Used in section 63.
⟨ handle non-triv cases 54 ⟩ Used in section 51.
⟨ initialize newfacets 64 ⟩ Used in section 63.
⟨ insert point in a new simplex 74 ⟩ Used in section 63.
⟨ look for neighborhood of unbounded simplex 71 ⟩ Used in section 69.
⟨ `main.c` 11 ⟩
⟨ put in neighborhood and newfacets 69 ⟩ Used in section 68.
⟨ `rand.c` 98 ⟩
⟨ reduce dimension 58 ⟩ Used in section 54.

⟨set up variables 55⟩ Used in section 54.
⟨special treatment of newly built origin simplex 76⟩ Used in section 63.
⟨take care for new origin_simplex 62⟩ Used in section 59.
⟨test edge facets 70⟩ Used in section 69.
⟨test newfacets 72⟩ Used in section 69.