Parameterized Implementations of Classical Planar Convex Hull Algorithms and Extreme Point Computations*

Stefan Schirra
Max-Planck-Institut für Informatik
Saarbrücken, Germany

December 29, 1997

Abstract

We present C++-implementations of some classical algorithms for computing extreme points of a set of points in two-dimensional space. The template feature of C++ is used to provide generic code, that works with various point types and various implementations of the primitives used in the extreme point computation. The parameterization makes the code flexible and adaptable. The code can be used with primitives provided by the CGAL-kernel, primitives provided by LEDA, and others. The interfaces of the convex hull functions are compliant to the Standard Template Library.

*Work on this project is supported by the ESPRIT IV LTR Project No. 21957 (CGAL).

Contents

1 Introduction

We present C++-implementations of some classical algorithms computing extreme points for a set of points in two-dimensional space. The algorithms are implemented as function templates in order to abstract away representation details. The presented code will be part of release 1.0 of CGAL [17].

In the next section we give a brief review on the history of two-dimensional convex hull computation. In Section 3 we discuss the genericity of our implementations. In particular, we discuss the requirements on the template parameters of our convex hull function templates. Section 5 gives the specification of the implemented funtions. Next we give two small examples. Before we present the actual implementations in Sections 7 to 11, we make some general remarks on the code and discuss some further preliminaries in Section 6. Section 12 presents some implementations of the concept of a convex hull traits class defined in Section 3.1. The files containing the code are presented in Section 13. Finally we present some test code. In an appendix, we present related and less interesting code fragments.

2 Convex Hull Computation in the Plane

The *convex hull* of a finite set of points $P = \{p_0, \ldots, p_{n-1}\}$ in the plane is the smallest convex polygon containing the points. An implicit representation of the convex hull $CH(P)$ is the counterclockwise sequence of vertices of $CH(P)$. A point in P is called *extreme* if it is a vertex of $CH(P)$. It is a well-known result in computational geometry that the convex hull of a set of n points in the plane can be computed in time $O(n \log h)$ [31, 32, 18] where h is the number of vertices of $CH(P)$, and that this bound is tight in the algebraic decision tree model of computation [32].

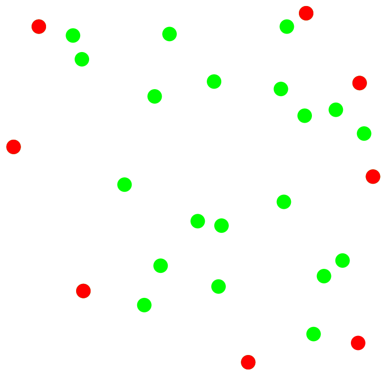

Figure 1: A set of points and its extreme points

There is a rich literature on the planar convex hull problem, especially at the end of the 70s and beginning of the 80s, e.g. [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14, 15, 19, 20, 21, 23, 25, 26, 27, 28, 29, 31, 33, 35, 39, 40, 46, 47]. Two classical algorithms on convex hull computation in the plane were already published in 1972/73: Graham's scan algorithm [24] and Jarvis' gift-wrapping method [30], also known as "Jarvis' march". Already in 1967, Bass and Schubert [13] described another algorithm for computing the convex hull of a planar point set and used it as a subroutine to solve some other geometric problem. If interpreted in the right way, this algorithm already contains the ideas of the "throw-away"-algorithm by Akl and Toussaint [4] published more than ten years later, see [46]. The most notable contribution during the last decade is certainly Chan's algorithm [18].

The choice of algorithms implemented in this report was inspired by the work of Allison and Noga [6]. Since in the literature many ideas have been reinvented or modified, and some of the techniques were slightly incorrect as published, it is difficult to attribute a planar convex hull algorithm to an author or a set of authors. For the lack of a better alternative, we nevertheless use author names to distinguish the implementations of the different methods, analogously to [6].

We give implementations of the following methods for planar convex hull computation:

Graham-Andrew-Algorithm A variant [36] of Andrew's variant [8] of the Graham scan [24]. The Graham scan is related to Sklansky's procedure [43] to compute the convex hull of certain polygons. Andrew's variant is related to a subroutine in the "Akl-Toussaint-algorithm" [4], see [2].

Akl-Toussaint-Algorithm as described originally in [4]. Bhattacharya and Toussaint [14] implemented a slightly modified version, which has better performance for large point sets. As mentioned above, the algorithm of [13] is already very similar to the algorithm presented in [4].

Eddy-Algorithm as described in [21]. Recursive variant of the two-dimensional instance of what is now known as the quickhull-algorithm [11].

Bykat-Algorithm as described in [15]. Non-recursive variant of the two-dimensional instance of what is now known as the quickhull-algorithm [11]. See also [40].

Jarvis-Algorithm Jarvis gift-wrapping method [30].

Asymptotically the two former methods have running time $O(n \log n)$, while the latter three have running time $O(nh)$. For explanation and analysis of the algorithms we refer to the original papers [8, 4, 21, 30], to [5], or to textbooks on computational geometry.

3 Generic Convex Hull Functions

We use the generic programming paradigm knwon from the Standard Template Library, i.e., we use templates to abstract away the representation details of the objects we deal with. Convex hull computation in the plane takes a set of points and reports a sorted sequence of points. Many implementations of convex hull algorithms assume that the input points are stored in a specific container, most often a plain C-array or some implementation of a list data type. Analogously, the return the extreme point sequence in a specific container, most often some implementation of a list data type. The user of such an algorithm has to provide the input and process the output in these specific formats.

In the Standard Template Library, iterators are used to abstract away the details of communication between an algorithm and a data structure on which the algorithm operates. An iterator is an abstract *concept* (or abstraction) of accessing data. The concept iterator is defined by a set of requirements. Every concrete class satisfying these requirements is a model for an iterator. There are different sorts of iterator concepts corresponding to different ways data can be accessed: `InputIterator`, `OutputIterator`, `ForwardIterator`, `BidirectionalIterator`, and `RandomAccessIterator`. For more precise definitions of the requirements on these sorts of iterators we refer to [44, 38, 42] or [16]. The requirements on iterators are chosen such that a pointer (into a C-array) is a model of the concept iterator.

We use the iterator concept to dissassociate the convex hull algorithms from the way input and output points are maintained. The convex hull algorithms get the points via an *iterator range* `[first,last)`. You can think of an iterator as "pointing" to an object, in our case a point. The sequence of points corresponding to an iterator range is obtained by starting with the object pointed to by `first` and then applying the "go-to-successor"-operation (++-operation in C++) and adding the object pointed to after the operation until `last` is reached. For `last`, no object is added. For example, let `Point a[10]` be an array of 10 points, and let `Point* first = a + 2` and `Point* last = a + 6`. Then the sequence of points corresponding to the range `[first,last)` is `a[2], a[3], a[4], a[5]` . For reporting the counterclockwise sequence of extreme points we use an `OutputIterator`. This is in complete analogy to the way algorithms in the Standard Template Library operate on containers.

We implement our convex hull algorithm as function templates parameterized by two iterator types. The names chosen for these template parameter tell you for which sort of iterator concept the actual arguments in an instantiation of the template must be a model. For example we have

⟨*Graham-Andrew function template declaration*⟩≡
```
template <class InputIterator, class OutputIterator>
inline
OutputIterator
CGAL_ch_graham_andrew( InputIterator  first, InputIterator  last,
                       OutputIterator result );
```

Thanks to the iterator concept, our implementations of convex hull algorithms are generic with respect to the way input points and output points are maintained. We are not restricted to a specific kind of "container". We can pass the set of points to our convex hull functions from a plain C-array, any of the standard containers `vector`, `list`, `queue`, from a `leda_list`, or any other "data structure" for which models of the concept iterator exist. We can even pass the points to the functions from an input stream or take them from a `leda_window`. For reporting the output sequence we have the same flexibility.

Here is a concrete example. Points are read from standard input and the counterclockwise sequence of extreme points is reported to standard out. `istream_iterator` and `ostream_iterator` are classes defined in the Standard Template Library.

⟨*ch_example_from_cin_to_cout.C*⟩≡

```
#include <CGAL/basic.h>
#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>
#include <CGAL/ch_graham_andrew.h>

typedef   CGAL_Point_2<CGAL_Cartesian<double> >       Point_2;

int
main()
{
  istream_iterator< Point_2, ptrdiff_t >  in_start( cin );
  istream_iterator< Point_2, ptrdiff_t >  in_ende;
  ostream_iterator< Point_2 >             out( cout, "\n" );
  CGAL_ch_graham_andrew( in_start, in_ende, out );
  return 0;
}
```

So far, we described the genericity of our convex hull algorithms with respect to the way input and output points are maintained. The algorithm called in the example above works on a concrete type of points from the CGAL-kernel: points represented by Cartesian coordinates of type `double`. The points in the CGAL-kernel are parameterized types as well. The requirements on the single template parameter of points in the CGAL-kernel define what is called a *representation class* concept. We can not easily forward this parameterization to a convex hull function, since the point type does not show up in the argument list of the function and hence can not be deduced by the compiler.[1] So we add an additional argument to the function (and use the parameterized CGAL point in the definition of the function template instead of some concrete type):

⟨*Graham-Andrew function template declaration with representation class*⟩≡

```
template <class R, class InputIterator, class OutputIterator>
inline
OutputIterator
CGAL_ch__graham_andrew(InputIterator  first,
                       InputIterator  last,
                       OutputIterator result,
                       CGAL_Point_2<R>* );
```

Now the code is generic with respect to the representation class of the CGAL points, but we still can not use our convex hull code with other point types, e.g. the point types `leda_rat_point` and `leda_point` from LEDA. However, there is nothing that restricts us to CGAL points besides the availabilty of a few predicates on the points. So we could make our implementation more generic by replacing the representation class template parameter by a template parameter for a point type, add requirements on the availability of predicates with certain names to our concept of point, and use these names in the function template. We could avoid this name dependency by providing the predicates via function pointers. However, in terms of efficiency it is better to use function objects [38], i.e. to increase the list of template parameters by some types performing the predicate operations on the points. This opens the

[1]New rules for template functions would allow us to explicitly provide the template arguments using the specialization syntax in a function call. If we would add the representation class type as the first template parameter, we would have to provide only the representation class type in a call, for example `CGAL_ch_graham_andrew< CGAL_Cartesian<int> >(fi,la,re)`, since all others could be deduced.

door for inlining the predicate operations. In the Graham-Andrew-algorithm all we need is a predicate to test whether a point is lexicographically smaller than a second one and a predicate to check whether three points form a leftturn. Here is a function template declaration whose parameter list is extended by parameters for function object types realizing these predicates:

⟨*Graham-Andrew with function object parameters*⟩≡
```
template <class Point_2, class Less_xy, class Leftturn,
          class InputIterator, class OutputIterator>
inline
OutputIterator
CGAL_ch__graham_andrew(InputIterator  first, InputIterator  last,
                       OutputIterator result,
                       Point_2*,
                       const Less_xy &,
                       const Leftturn& );
```

Even if an function template uses only a few types and operations on them, the template parameter list and the argument list of the function become unpleasant. We use the "nested typedefs for name commonality"-idiom [12, 34] to make them shorter. Our function templates expect a single additional (besides the iterator types) template parameter that provides the (names of the) types of geometric objects and predicates. The requirements on this template parameter define the concept *convex hull traits*. A model for the concept "convex hull traits" must specify the types of the primitives involved in the convex hull algorithm. More precisely, it must provide names of these types, either by `typedef`s or as nested types. A model that can be used with the function template declared below must provide names `Traits::Point_2`, `Traits::Less_xy`, and `Traits::Leftturn`.

⟨*Graham-Andrew with function object parameters*⟩+≡
```
template <class InputIterator, class OutputIterator, class Traits>
inline
OutputIterator
CGAL_ch__graham_andrew(InputIterator  first, InputIterator  last,
                       OutputIterator result,
                       const Traits&  ch_traits);
```

The parameterization makes the code flexible and adaptable. It can be used with various implementations of points and primitives on these points, in particular implementations disposed by LEDA or other C++-libraries, or user-defined point and predicate types. Only a traits class providing an appropriate interface to the primitives and points is needed. There are default versions of our convex hull algorithms which have no traits class argument. They operate on the point type `CGAL_Point_2<R>` of the CGAL-kernel [22] using default primitives of this kernel.

3.1 Convex Hull Traits

A traits class for our functions computing extreme points must provide the types of the primitives, i.e. objects and predicates, used in the code. We list each of these primitives together with a short description and requirements on it and call a primitive a *trait* of the algorithm. Not all convex hull functions need all the traits. The traits that are actually required by a function are given in the specification (Section 5) together with the semantics of the function.

Point_2	the points on which the convex hull functions operate.
Less_xy	binary predicate object type comparing *Point_2*s lexicographically. Must provide *bool operator( )(Point_2 p, Point_2 q)*, where *true* is returned if $p <_{\mathrm{xy}} q$. We have $p <_{\mathrm{xy}} q$, iff $p_x < q_x$ or $p_x = q_x$ and $p_y < q_y$, where p_x and p_y denote x- and y-coordinate of point p resp.
Less_yx	same as *Less_xy* with the roles of x and y interchanged.
Greater_xy	same as *Less_xy* with $p <_{\mathrm{xy}} q$ replaced by $p >_{\mathrm{xy}} q$.

Greater_yx — same as *Greater_xy* with the roles of x and y interchanged.

Right_of_line — unary predicate object type. Must provide a constructor taking two *Point_2*s p and q and *bool operator*( )(*Point_2* r), which returns true if r lies right of the directed line through p and q.

Less_dist_to_line — binary predicate object type. Must provide a constructor taking two *Point_2*s p and q and *bool operator*( )(*Point_2* r, *Point_2* s), which returns true if the signed distance of r to the line l_{pq} through p and q is smaller as the the distance of s to l_{pq}. It is used to compute the point right of a line with maximum unsigned distance to the line. The binary predicate must provide a total order compatible to convexity, i.e. for any line segment s one of the endpoints of s is the smallest point among the points on s, with respect to the order given by *Less_dist_to_line*.

Less_rotate_ccw — binary predicate object type. Must provide a constructor taking a *Point_2* e and *bool operator*( )(*Point_2* p, *Point_2* q), where *true* is returned iff a tangent at e to the point set $\{e,p,q\}$ hits p before q when rotated conterclockwise around e. Ties are broken such that the point with larger distance to e is smaller!

Leftturn — predicate object type. Must provide a default constructor and a *bool operator*( )(*Point_2* p, *Point_2* q, *Point_2* r), where *true* is returned iff p,q, and r form a leftturn.

Rightturn — predicate object type. Must provide a default constructor and a *bool operator*( )(*Point_2* p, *Point_2* q, *Point_2* r), where *true* is returned iff p,q, and r form a rightturn.

4 Examples

We give two examples. The first example uses a CGAL random point generator to generate random points in a disc. The convex hull of the points is computed and displayed in a `leda_window` in red. After a mouse click the sequence of extreme points is computed by Bykat's algorithm and displayed in blue. Figure 2 shows an example.

⟨*ch_example_window.C*⟩≡
  ⟨*ch_example_window - includes*⟩
  ⟨*ch_example_window - typedefs*⟩

```
int
main()
{
  vector<Point_2> points;
  points.reserve(500);
  CGAL_Random_points_in_disc_2<Point_2,Creator> g( 200.0);
  CGAL_copy_n( g, 500, back_inserter( points));
  CGAL_Window_stream W(512, 512);
  W.init(-256.0, 255.0, -256.0);
  W << CGAL_RED;
  CGAL_Ostream_iterator<Point_2,CGAL_Window_stream> wo(W);
  copy( points.begin(), points.end(), wo );
  W.read();
  W << CGAL_BLUE;
  // compute convex hull using Bykat-algorithm
  CGAL_ch_bykat( points.begin(), points.end(), wo );
  W.read();
  return 0;
}
```

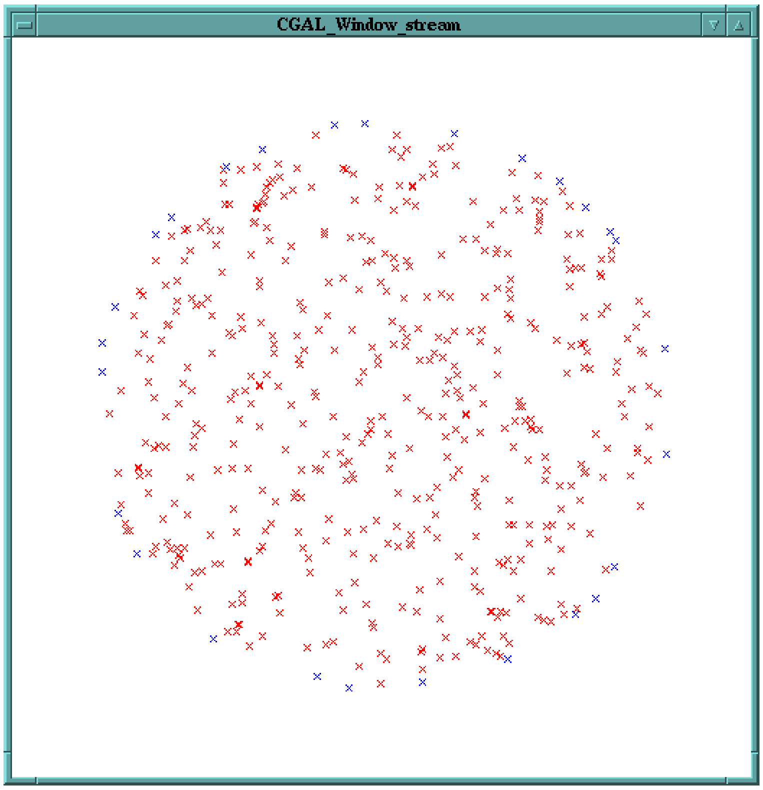

Figure 2: 500 random points in a disc.

⟨*ch_example_window - includes*⟩≡
```
#include <CGAL/basic.h>
#include <vector.h>
#include <CGAL/copy_n.h>
#include <CGAL/point_generators_2.h>
#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>
#include <CGAL/ch_bykat.h>
#include <CGAL/IO/Window_stream.h>
#include <CGAL/IO/Ostream_iterator.h>
```

⟨*ch_example_window - typedefs*⟩≡
```
typedef CGAL_Cartesian<double>                      RepCls;
typedef CGAL_Point_2<RepCls>                        Point_2;
typedef CGAL_Segment_2<RepCls>                      Segment_2;
typedef CGAL_Creator_uniform_2<double,Point_2>  Creator;
```

The second example constructs a CGAL polygon to represent the convex hull of a set of points read from standard input. A CGAL polygon acts like a STL-container. It provides a `insert()` member function to add points to a polygon. In the example, this member function is used by an `insert_iterator`. There is a default convex hull algorithm, see Section 13, which is called by `CGAL_convex_hull_points_2`.

⟨*ch_example_polygon.C*⟩≡
```
#include <CGAL/basic.h>
#include <list.h>
#include <CGAL/leda_integer.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Point_2.h>
```

```
#include <CGAL/convex_hull_2.h>
#include <CGAL/Polygon_2.h>

typedef CGAL_Homogeneous<leda_integer>            RepCls;
typedef CGAL_Point_2<RepCls>                      Point_2;
typedef CGAL_Polygon_2<RepCls, list<Point_2> >    Polygon_2;

int
main()
{
  Polygon_2                                 CH;
  istream_iterator< Point_2, ptrdiff_t >    in_start( cin );
  istream_iterator< Point_2, ptrdiff_t >    in_end;

  CGAL_convex_hull_points_2( in_start, in_end,
                             inserter(CH, CH.vertices_begin()) );
  cout << "The convex hull is " << endl;
  CGAL_set_pretty_mode(cout);
  cout << CH ;
  return 0;
}
```

For example, the output looks like

```
The convex hull is
Polygon_2(
  PointH2(2, 50, 77)
  PointH2(6, 26, 81)
  PointH2(25, 1, 66)
  PointH2(81, 3, 71)
  PointH2(103, 9, 51)
  PointH2(81, 12, 23)
  PointH2(90, 120, 2)
  PointH2(26, 111, 40)
  PointH2(11, 107, 70)
)
```

5 Specifications

All functions allow a user to specify the implementation of the objects and primitives that are used in the function in an traits class which is passed to the function as an argument. For all the functions there is also a default version which operates on CGAL-kernel objects using CGAL-kernel predicates. These default functions have not traits class argument. Their implementation uses a default traits class presented in Section 12.

All functions described below are template functions, where iterator types and traits class type are template parameters. As usual with generic programming the names of the iterator parameters indicate the requirements of the function on the iterators. The functions described below are located in the file given in the include statement above the specification (if no file is shown, the function is located in the same file as its predecessor in the list of specifications). *Traits* is always used to denote the traits class type.

5.1 Algorithms Computing Counterclockwise Sequence of Extreme Points

There are several algorithms for computing the counterclockwise ordered sequence of extreme points of a range of point elements. A default algorithm is called by *CGAL_convex_hull_points_2*( ). Currently, it is *CGAL_ch_akl_toussaint*( ) which uses the Akl-Toussaint algorithm claimed to be the fastest in [5], provided that the iterator is at least a forward iterator. Otherwise, *CGAL_ch_bykat*( ) is used because of its lower iterator requirements. Furthermore there are functions using Andrew's variant of the Graham scan, Eddy's algorithm, its non-recursive equivalent, i.e. Bykat's algorithm, and Jarvis' march resp. They all have the same semantics as *CGAL_convex_hull_points_2*( ). The iterator requirements, however, are different, as indicated by the names chosen for the template parameters.

NOTE: If (expensive) postconditions are checked, see Section 6, for all the functions in this subsection the list of required traits is enlarged by (*Traits*::*Right_of_Line*,) *Traits*::*Leftturn* and *Traits*::*Rightturn*.

`#include <CGAL/convex_hull_2.h>`

OutputIterator CGAL_convex_hull_points_2(*InputIterator first*, *InputIterator last*, *OutputIterator result*, *Traits ch_traits*)

generates the counterclockwise sequence of extreme points of the points in the range [*first*,*last*). The resulting sequence is placed starting at position *result*, and the past-the-end iterator for the resulting sequence is returned. It is not specified, at which point the cyclic sequence of extreme points is cut into a linear sequence.
Preconditions: [*first*,*last*) does not contain *result*.
TRAITS: operates on *Traits*::*Point_2* using *Traits*::*Less_xy*, *Traits*::*Less_yx*, *Traits*::*Greater_xy*, *Traits*::*Greater_yx*, and *Traits*::*Leftturn*.

`#include <CGAL/ch_graham_andrew.h>`

OutputIterator CGAL_ch_graham_andrew(*InputIterator first*, *InputIterator last*, *OutputIterator result*, *Traits ch_traits*)

same as *CGAL_convex_hull_points_2*(*first*, *last*, *result*).
TRAITS: uses *Traits*::*Point_2*, *Traits*::*Leftturn* and *Traits*::*Less_xy*.

`#include <CGAL/ch_akl_toussaint.h>`

OutputIterator CGAL_ch_akl_toussaint(*ForwardIterator first*, *ForwardIterator last*, *OutputIterator result*, *Traits ch_traits*)

same as *CGAL_convex_hull_points_2*(*first*, *last*, *result*).
TRAITS: operates on *Traits*::*Point_2* using *Traits*::*Less_xy*, *Traits*::*Less_yx*, *Traits*::*Greater_xy*, *Traits*::*Greater_yx*, and *Traits*::*Leftturn*.

`#include <CGAL/ch_eddy.h>`

OutputIterator CGAL_ch_eddy(*InputIterator first*, *InputIterator last*, *OutputIterator result*, *Traits ch_traits*)

same as *CGAL_convex_hull_points_2*(*first*, *last*, *result*).
TRAITS: uses *Traits*::*Point_2*, *Traits*::*Less_dist_to_line*, *Traits*::*Right_of_line*, and *Traits*::*Less_xy*, *Traits*::*Greater_xy*.

`#include <CGAL/ch_bykat.h>`

OutputIterator CGAL_ch_bykat(*InputIterator first*, *InputIterator last*, *OutputIterator result*, *Traits ch_traits*)

same as *CGAL_convex_hull_points_2*(*first*, *last*, *result*).
TRAITS: uses *Traits*::*Point_2*, *Traits*::*Less_dist_to_line*, *Traits*::*Right_of_line*, and *Traits*::*Less_xy*, *Traits*::*Greater_xy*.

`#include <CGAL/ch_jarvis.h>`

OutputIterator CGAL_ch_jarvis(*ForwardIterator first*, *ForwardIterator last*, *OutputIterator result*, *Traits ch_traits*)

same as *CGAL_convex_hull_points_2*(*first*, *last*, *result*).
TRAITS: uses *Traits*::*Point_2*, *Traits*::*Less_rotate_ccw*, and *Traits*::*Less_xy*.

5.2 Algorithms Computing Extreme Points in Coordinate Directions

The following functions compute special extreme points. More precisely, as before, not points are computed, but iterators. The function name tells you which extreme points are reported as reference arguments and in which order. n means north, w west, s south, and e east, which stands for maximal y-, minimal x-, minimal y-, and maximal x-coordinate resp.

`#include <CGAL/ch_selected_extreme_points_2.h>`

void CGAL_ch_nswe_point(*ForwardIterator first*, *ForwardIterator last*, *ForwardIterator& n*, *ForwardIterator& s*, *ForwardIterator& w*, *ForwardIterator& e*, *Traits ch_traits*)

traverses the range [*first*,*last*). After execution, the value of n is an iterator in the range such that $*n \geq_{\mathrm{yx}} *it$ for all iterators it in the range. Similarly, for s, w, and e the inequations $*s \leq_{\mathrm{yx}} *it$, $*w \leq_{\mathrm{xy}} *it$, and $*e \geq_{\mathrm{yx}} *it$ hold respectively for all iterators it in the range.
TRAITS: uses *Traits*::*Less_xy*, *Traits*::*Less_yx*, *Traits*::*Greater_xy*, and *Traits*::*Greater_yx*.

void CGAL_ch_ns_point(*ForwardIterator first*, *ForwardIterator last*, *ForwardIterator& n*, *ForwardIterator& s*, *Traits ch_traits*)

traverses the range [*first*,*last*). After execution, the value of n is an iterator in the range such that $*n \geq_{\mathrm{yx}} *it$ for all iterators it in the range. Similarly, for s the inequation $*s \leq_{\mathrm{yx}} *it$ holds for all iterators it in the range.
TRAITS: uses function object types *Traits*::*Less_yx* and *Traits*::*Greater_yx*.

void CGAL_ch_we_point(*ForwardIterator first*, *ForwardIterator last*, *ForwardIterator& w*, *ForwardIterator& e*, *Traits ch_traits*)

traverses the range [*first*,*last*). After execution, the value of w is an iterator in the range such that $*w \leq_{\mathrm{xy}} *it$ for all iterators it in the range. Similarly, for e the inequation $*e \geq_{\mathrm{yx}} *it$ holds for all iterators it in the range.
TRAITS: uses function object types *Traits*::*Less_xy* and *Traits*::*Greater_xy*.

void CGAL_ch_n_point(*ForwardIterator first*, *ForwardIterator last*, *ForwardIterator& n*, *Traits ch_traits*)

traverses the range [*first*,*last*). After execution, the value of n is an iterator in the range such that $*n \geq_{\mathrm{yx}} *it$ for all iterators *it* in the range.
TRAITS: uses *Traits*::*Greater_yx*.

void CGAL_ch_s_point(*ForwardIterator first*, *ForwardIterator last*, *ForwardIterator& s*, *Traits ch_traits*)

traverses the range [*first*,*last*). After execution, the value of s is an iterator in the range such that $*s \leq_{\mathrm{yx}} *it$ for all iterators *it* in the range.
TRAITS: uses *Traits*::*Less_yx*.

void CGAL_ch_e_point(*ForwardIterator first*, *ForwardIterator last*, *ForwardIterator& e*, *Traits ch_traits*)

traverses the range [*first*,*last*). After execution, the value of e is an iterator in the range such that $*e \geq_{\mathrm{yx}} *it$ for for all iterators *it* in the range.
TRAITS: uses *Traits*::*Greater_xy*.

void CGAL_ch_w_point(*ForwardIterator first*, *ForwardIterator last*, *ForwardIterator& w*, *Traits ch_traits*)

traverses the range [*first*,*last*). After execution, the value of w is an iterator in the range such that $*w \leq_{\mathrm{yx}} *it$ for for all iterators *it* in the range.
TRAITS: uses *Traits*::*Less_yx*.

The last four functions are provided for completeness and uniformity. There will be other functions in **CGAL** having an iterator as return value, for instance *ForwardIterator CGAL_left_most_point*(*ForwardIterator first*, *ForwardIterator last*).

5.3 Algorithms Computing a Subsequence of Extreme Points

NOTE: If (expensive) postconditions are checked, see Section 6, for all the functions in this subsection the list of required traits is enlarged by (*Traits*::*Right_of_Line*,) *Traits*::*Leftturn* and *Traits*::*Rightturn*.

```
#include <CGAL/ch_jarvis.h>
```

OutputIterator CGAL_ch_jarvis_march(*ForwardIterator first*, *ForwardIterator last*, *Point start_p*, *Point stop_p*, *OutputIterator result*, *Traits ch_traits*)

generates the counterclockwise ordered subsequence of extreme points between *start_p* and *stop_p* of the points in the range [*first*,*last*), starting at position result with point *start_p*. The last point generated is the point preceding *stop_p* in the counterclockwise order of extreme points.
Precondition: *start_p* and *stop_p* are extreme points with respect to the points in the range [*first*,*last*) and *stop_p* is an element of range [*first*,*last*).
TRAITS: uses *Traits*::*Point_2* $\equiv$ *Point*, and *Traits*::*Less_rotate_ccw*.

```
#include <CGAL/ch_graham_andrew.h>
```

OutputIterator CGAL_ch_graham_andrew_scan(*BidirectionalIterator first*, *BidirectionalIterator last*, *OutputIterator result*, *Traits ch_traits*)

computes the sorted sequence of extreme points which are not left of pq and reports this sequence in a range starting at *result*, where p is the value of *first* and q is the value of *last* -1. The sequence reported starts with p, point q is omitted.
Precondition: The points in [*first*,*last*) are sorted with respect to pq and the range [*first*,*last*) contains at least two different points.
TRAITS: uses *Traits*::*Leftturn* operating on the point type *Traits*::*Point_2*.

5.4 Convexity Checking

`#include <CGAL/convexity_check_2.h>`

bool CGAL_is_ccw_strongly_convex_2(*ForwardIterator first, ForwardIterator last, Traits ch_traits*)

returns true, if the point elements in [*first*,*last*) form a counterclockwise oriented strongly convex polygon.
TRAITS: uses *Traits*::*Leftturn* and *Traits*::*Less_xy*.

bool CGAL_is_cw_strongly_convex_2(*ForwardIterator first, ForwardIterator last, Traits ch_traits*)

returns true, if the point elements in [*first*,*last*) form a clockwise oriented strongly convex polygon.
TRAITS: uses *Traits*::*Rightturn* and *Traits*::*Less_xy*.

bool CGAL_ch_brute_force_check_2(*ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, Traits ch_traits*)

returns true, if all points in [*first1*,*last1*) are not right of the lines defined by consecutive points in the range [*first2*,*last2*), where the range is considered as a cycle.
TRAITS: uses *Traits*::*Right_of_line*.

bool CGAL_ch_brute_force_chain_check_2(*ForwardIterator1 first1, ForwardIterator1 last1, ForwardIterator2 first2, ForwardIterator2 last2, Traits ch_traits*)

returns true, if all points in [*first1*,*last1*) are not right of the lines defined by consecutive points in the range [*first2*,*last2*).
TRAITS: uses *Traits*::*Right_of_line*.

5.5 Time Complexities

CGAL_ch_graham_andrew( ) and *CGAL_ch_akl_toussaint*( ) have worst case running time $O(n \log n)$, where n is the number of points. *CGAL_ch_eddy*( ), *CGAL_ch_bykat*( ), and *CGAL_ch_jarvis*( ) have worst case running time $O(nh)$ each, where n is the number of points and h the number of extreme points. The time complexity of the functions computing extreme points in north, south, east, and west directions is $O(n)$. *CGAL_ch_graham_andrew_scan*( ) has running time $O(n)$, while *CGAL_ch_jarvis_march*( ) has $O(nh')$, where h' is the size of the computed subsequence. The convexity checkers *CGAL_is_ccw_strongly_convex_2*( ) and *CGAL_is_cw_strongly_convex_2*( ) have linear running time.

6 Implementation Preliminaries

We follow CGAL's recommendations for assertions. All non-trivial functions check (expensive) postconditions, some functions check preconditions, too. Throughout the code, assertions are included. Checking preconditions, postconditions, and assertions can be switched off by defining `CGAL_CH_NO_PRECONDITIONS`, `CGAL_CH_NO_POSTCONDITIONS`, and `CGAL_CH_NO_ASSERTIONS` resp. Expensive checking is switched off by default. It can be enabled by defining `CGAL_CH_CHECK_EXPENSIVE`. Assertion handling is located in the file `<CGAL/ch_assertions.h>`, see Appendix A. The implementations make use of the algorithms and containers provided in the Standard Template Library.

For adaptation to compilers and their features and bugs CGAL's configuration tools are used. In some functions the traits class argument is used only if certain checkings are enabled. We use the following macro to use an argument in order to suppress warnings.

⟨*use argument*⟩≡
```
#define CGAL_CH_USE_ARGUMENT(arg)  (void)(arg)
```

6.1 Default Versions

In the default versions of the functions the default convex hull traits class is not visible. We use a trick used in the former reference implementation of the STL provided by Hewlett-Packard: The `value_type()` function for iterators is used to get some information on the type `T` of the objects accessed through an iterator. The `value_type()` function returns an object of type pointer to `T`. With the help of this pointer type, we extract the representation class type of the point type under consideration. Via a template function having a representation class template parameter, a corresponding function with default traits class is called. For example, in

```
template <class InputIterator, class OutputIterator>
OutputIterator
CGAL_ch_foo(InputIterator first, InputIterator last, OutputIterator res)
```

the template function

```
template <class InputIterator, class OutputIterator, class R>
OutputIterator
CGAL_ch__foo(InputIterator first, InputIterator last, OutputIterator res,
             CGAL_Point_2<R>* )
```

is called using `value_type()`. Now that the representation class type is known, the appropriate default traits class `CGAL_convex_hull_traits_2<R>` can be finally passed to

```
template <class InputIterator, class OutputIterator, class Traits>
OutputIterator
CGAL_ch_foo(InputIterator first, InputIterator last, OutputIterator res,
            const Traits& ch_traits)
```

Altogether, for the default version operating on `CGAL_Point_2<R>` points, we have

```
      CGAL_ch_foo(first, last, res )
calls CGAL_ch__foo( first, last, res, value_type(first))
calls CGAL_ch_foo(first, last, res, CGAL_convex_hull_traits_2<R>() )
```

For each function we have these three variants. We declare the first two functions `inline` to avoid performance penalties. The scheme becomes boring very soon, but it is very useful.

6.2 Checking

Postcondition checking is a bit nasty in our functions. Convexity checking is no problem with respect to running time, but it only checks whether a computed sequence forms a counterclockwise oriented polygon. In an *expensive* postcondition check, we search for points outside the counterclockwise oriented convex polygon formed by the reported extreme points. The nasty thing is, that in most cases we can not check the actual output, which was reported through an output iterator, and hence cannot be accessed from the processing function anymore. To overcome this problem we use an iterator type that acts like the tee command in UNIX. The iterator type, called CGAL_Tee_for_output_iterator<OutputIterator,T>, is constructed from an (output) iterator. It forwards the data to this iterator, but it also copies the data to a local container. The local data can be accesses through the CGAL_Tee_for_output_iterator. The local container is maintained like a stream. All copies of a CGAL_Tee_for_output_iterator operate on the same local container. A CGAL_Tee_for_output_iterator is parameterized by the type of the output iterator which it taps and by the value type of the output iterator[2]. For the sake of completeness, the definition of class CGAL_Tee_for_output_iterator is given in Appendix B.

For checking convexity we follow [37].

⟨*checker declaration with traits*⟩≡

```
template <class ForwardIterator, class Traits>
bool
CGAL_is_ccw_strongly_convex_2( ForwardIterator first, ForwardIterator last,
                                const Traits& ch_traits);

template <class ForwardIterator, class Traits>
bool
CGAL_is_cw_strongly_convex_2( ForwardIterator first, ForwardIterator last,
                               const Traits& ch_traits);
```

Here are the functions used to call the functions above with the default traits class CGAL_convex_hull_traits_2<R>.

⟨*checker inline declaration*⟩≡

```
template <class ForwardIterator, class R>
inline
bool
CGAL__is_cw_convex_2( ForwardIterator first, ForwardIterator last,
                      CGAL_Point_2<R>* )
{
  return CGAL_is_cw_strongly_convex_2( first, last,
                                       CGAL_convex_hull_traits_2<R>() );
}

template <class ForwardIterator>
inline
bool
CGAL_is_cw_strongly_convex_2( ForwardIterator first, ForwardIterator last )
{ return CGAL__is_cw_convex_2( first, last, value_type(first) ); }

template <class ForwardIterator, class R>
inline
bool
CGAL__is_ccw_convex_2( ForwardIterator first, ForwardIterator last,
                       CGAL_Point_2<R>* )
{
  return CGAL_is_ccw_strongly_convex_2( first, last,
                                        CGAL_convex_hull_traits_2<R>() );
}

template <class ForwardIterator>
inline
bool
CGAL_is_ccw_strongly_convex_2( ForwardIterator first, ForwardIterator last )
{ return CGAL__is_ccw_convex_2( first, last, value_type(first) ); }
```

In the counterclockwise case we check that triples of consecutive points form a leftturn. Furthermore we check that there is exactly one local minimum with respect to lexicograhical order to ensure the correct

[2] With the iterator traits recently added to the Standard Template Library parameterization by OutputIterator would suffice. However, this is not yet supported by most compilers.

winding number, as suggested in [45].

⟨*checker*⟩≡

```
template <class ForwardIterator, class Traits>
bool
CGAL_is_ccw_strongly_convex_2( ForwardIterator first, ForwardIterator last,
                               const Traits& ch_traits)
{
  typedef  typename Traits::Less_xy      Less_xy;
  typedef  typename Traits::Leftturn     Leftturn;

  Less_xy  smaller_xy;
  Leftturn leftturn;

  ForwardIterator iter1;
  ForwardIterator iter2;
  ForwardIterator iter3;

  if ( first == last) return true;

  iter2 = first;
  iter3 = ++iter2;

  if (iter3 == last ) return true;

  ++iter3;

  if (iter3 == last ) return ( *first != *iter2 );

  iter1 = first;
  short int f = 0;

  while (iter3 != last)
  {
      ⟨check point triple⟩
      ++iter1;
      ++iter2;
      ++iter3;
  }

  iter3 = first;
  ⟨check point triple⟩

  iter1 = iter2;
  iter2 = first;
  ++iter3;
  ⟨check point triple⟩

  return ( f > 1 ) ? false : true;
}
```

⟨*check point triple*⟩≡

```
if ( !leftturn( *iter1, *iter2, *iter3 ) ) return false;
if ( smaller_xy( *iter2, *iter1 ) && smaller_xy( *iter2, *iter3 )) ++f;
```

⟨*checker*⟩+≡

```
template <class ForwardIterator, class Traits>
bool
CGAL_is_cw_strongly_convex_2( ForwardIterator first, ForwardIterator last,
                              const Traits& ch_traits)
{
  typedef  typename Traits::Less_xy      Less_xy;
  typedef  typename Traits::Rightturn    Rightturn;

  Less_xy   smaller_xy;
  Rightturn rightturn;
```

```
  ForwardIterator iter1;
  ForwardIterator iter2;
  ForwardIterator iter3;

  if ( first == last) return true;

  iter2 = first;
  iter3 = ++iter2;

  if (iter3 == last ) return true;

  ++iter3;

  if (iter3 == last ) return ( *first != *iter2 );

  iter1 = first;
  short int f = 0;

  while (iter3 != last)
  {
      ⟨check point triple - cw⟩
      ++iter1;
      ++iter2;
      ++iter3;
  }

  iter3 = first;
  ⟨check point triple - cw⟩

  iter1 = iter2;
  iter2 = first;
  ++iter3;
  ⟨check point triple - cw⟩

  return ( f > 1 ) ? false : true;
}
```

⟨*check point triple - cw*⟩≡

```
if ( !rightturn( *iter1, *iter2, *iter3 ) ) return false;
if ( smaller_xy( *iter2, *iter1 ) && smaller_xy( *iter2, *iter3 )) ++f;
```

For ease of use there is also a brute force containment test. Note that due to the running time ($O(nh)$, which might be quadratic in n), this is not a checker in the sense of [37].

⟨*brute force check declaration with traits*⟩≡

```
template <class ForwardIterator1, class ForwardIterator2, class Traits>
bool
CGAL_ch_brute_force_check_2(ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2,
                            const Traits& ch_traits);
```

⟨*brute force check inline declaration*⟩≡

```
template <class ForwardIterator1, class ForwardIterator2, class R>
inline
bool
CCGAL_ch__brute_force_check_2(ForwardIterator1 first1,ForwardIterator1 last1,
                              ForwardIterator2 first2,ForwardIterator2 last2,
                              CGAL_Point_2<R>* )
{
  return CGAL_ch_brute_force_check_2( first1, last1,
                                      first2, last2,
                                      CGAL_convex_hull_traits_2<R>() );
}
```

```
template <class ForwardIterator1, class ForwardIterator2>
inline
bool
CGAL_ch_brute_force_check_2(ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2)
{
  return CCGAL_ch__brute_force_check_2( first1, last1,
                                        first2, last2,
                                        value_type(first1) );
}
```

In addition, we have the same check without wrap around to check the validity of subparts of the counterclockwise sequence of extreme points.

⟨*brute force chain check declaration with traits*⟩≡

```
template <class ForwardIterator1, class ForwardIterator2, class Traits>
bool
CGAL_ch_brute_force_chain_check_2(ForwardIterator1 first1,
                                  ForwardIterator1 last1,
                                  ForwardIterator2 first2,
                                  ForwardIterator2 last2,
                                  const Traits& ch_traits);
```

⟨*brute force chain check inline declaration*⟩≡

```
template <class ForwardIterator1, class ForwardIterator2, class R>
inline
bool
CCGAL_ch__brute_force_chain_check_2(ForwardIterator1 first1,
                                    ForwardIterator1 last1,
                                    ForwardIterator2 first2,
                                    ForwardIterator2 last2,
                                    CGAL_Point_2<R>* )
{
  return CGAL_ch_brute_force_chain_check_2( first1, last1,
                                            first2, last2,
                                            CGAL_convex_hull_traits_2<R>() );
}
template <class ForwardIterator1, class ForwardIterator2>
inline
bool
CGAL_ch_brute_force_chain_check_2(ForwardIterator1 first1,
                                  ForwardIterator1 last1,
                                  ForwardIterator2 first2,
                                  ForwardIterator2 last2)
{
  return CCGAL_ch__brute_force_chain_check_2( first1, last1,
                                              first2, last2,
                                              value_type(first1) );
}
```

⟨*brute force chain check*⟩≡

```
template <class ForwardIterator1, class ForwardIterator2, class Traits>
bool
CGAL_ch_brute_force_chain_check_2(ForwardIterator1 first1,
                                  ForwardIterator1 last1,
                                  ForwardIterator2 first2,
```

```
                                      ForwardIterator2 last2,
                                      const Traits& )
{
  ⟨chain check⟩
  return true;
}
```

⟨*chain check*⟩≡

```
typedef    typename Traits::Right_of_line    Right_of_line;
ForwardIterator1 iter11;
ForwardIterator2 iter21;
ForwardIterator2 iter22;
if ( first1 == last1) return true;
if ( first2 == last2) return false;
if ( CGAL_successor(first2) == last2 )
{
    while (first1 != last1)
    {
        if ( *first1++ != *first2 ) return false;
    }
    return true;
}
Right_of_line  rol(*first2, *CGAL_successor(first2) );
iter22 = first2;
iter21 = iter22++;
while (iter22 != last2)
{
    rol = Right_of_line( *iter21++, *iter22++ );
    iter11 = find_if( first1, last1, rol );
    if (iter11 != last1 ) return false;
}
```

⟨*brute force check*⟩≡

```
template <class ForwardIterator1, class ForwardIterator2, class Traits>
bool
CGAL_ch_brute_force_check_2(ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2,
                            const Traits& )
{
  ⟨chain check⟩
  rol = Right_of_line( *iter21, *first2 );
  iter11 = find_if( first1, last1, rol );
  if (iter11 != last1 ) return false;
  return true;
}
```

In all function templates the output iterator used to report the counterclockwise sequence of extreme points is called **result**. If postcondition checking is not disabled by a flag we intialize a **CGAL_Tee_for_output_iterator<>** named **res** with the output iterator **result**. We use this iterator to get access to the computed points.

⟨*define res*⟩≡
```
#if defined(CGAL_CH_NO_POSTCONDITIONS) || defined(CGAL_NO_POSTCONDITIONS) \
  || defined(NDEBUG)
OutputIterator  res(result);
#else
CGAL_Tee_for_output_iterator<OutputIterator,Point_2> res(result);
#endif // no postconditions ...
```

Analogously we have

⟨*return res*⟩≡
```
#if defined(CGAL_CH_NO_POSTCONDITIONS) || defined(CGAL_NO_POSTCONDITIONS) \
  || defined(NDEBUG)
return res;
#else
return res.to_output_iterator();
#endif // no postconditions ...
```

where `CGAL_Tee_for_output_iterator<>::to_output_iterator()` returns the output iterator maintained by a `CGAL_Tee_for_output_iterator<>`. The convexity check is called in the following piece of code

⟨*check convexity of generated output*⟩≡
```
CGAL_ch_postcondition( \
    CGAL_is_ccw_strongly_convex_2( res.output_so_far_begin(), \
                                   res.output_so_far_end(), \
                                   ch_traits));
```

6.3 Selected Extreme Points

We start with functions computing special extreme points. The name tells you which extreme points are reported in the reference arguments. `n` means north, `w` west, `s` south, and `e` east, which stands for a point with maximal y-, minimal x-, minimal y-, and maximal x-coordinate resp. For the x-coordinate comparison, `xy`-lexicographical order is used, for y-coordinate comparison `yx`-lexicographical order. We present here the code for the most universal variant. The code for the remaining functions is completely analogous and presented in Appendix C. We start with the declaration of the default versions operating on `CGAL_Point_2<R>` points using primitives from the CGAL-kernel. The appropriate default traits class is derived as sketched above in 6.1 on page 12.

⟨*nswe extremepoints inline declaration*⟩≡
```
template <class ForwardIterator, class R>
inline
void
CGAL_ch__nswe_point( ForwardIterator first, ForwardIterator last,
                     ForwardIterator& n,
                     ForwardIterator& s,
                     ForwardIterator& w,
                     ForwardIterator& e,
                     CGAL_Point_2<R>* )
{ CGAL_ch_nswe_point(first, last, n, s, w, e, CGAL_convex_hull_traits_2<R>()); }

template <class ForwardIterator>
inline
void
CGAL_ch_nswe_point( ForwardIterator first, ForwardIterator last,
                    ForwardIterator& n,
                    ForwardIterator& s,
                    ForwardIterator& w,
                    ForwardIterator& e)
{ CGAL_ch__nswe_point(first, last, n, s, w, e, value_type(first) ); }
```

Next we declare the functions with traits class parameter.

⟨*nswe extremepoints declaration with traits*⟩≡

```
template <class ForwardIterator, class Traits>
void
CGAL_ch_nswe_point( ForwardIterator first, ForwardIterator last,
                    ForwardIterator& n,
                    ForwardIterator& s,
                    ForwardIterator& w,
                    ForwardIterator& e,
                    const Traits& ch_traits);
```

The implementation of the selected extreme point computations are analogous to

⟨*nswe extremepoints*⟩≡

```
template <class ForwardIterator, class Traits>
void
CGAL_ch_nswe_point( ForwardIterator first, ForwardIterator last,
                    ForwardIterator& n,
                    ForwardIterator& s,
                    ForwardIterator& w,
                    ForwardIterator& e,
                    const Traits& )
{
  typename Traits::Less_xy    lexicographically_xy_smaller;
  typename Traits::Less_yx    lexicographically_yx_smaller;
  typename Traits::Greater_xy lexicographically_xy_larger;
  typename Traits::Greater_yx lexicographically_yx_larger;
  n = s = w = e = first;
  while ( first != last )
  {
      if ( lexicographically_xy_smaller( *first, *w ))  w = first;
      if ( lexicographically_xy_larger ( *first, *e ))  e = first;
      if ( lexicographically_yx_larger ( *first, *n ))  n = first;
      if ( lexicographically_yx_smaller( *first, *s ))  s = first;
      ++first;
  }
}
```

7 Andrew's Variant of Graham's Algorithm

The original Graham scan algorithm is described in [24], the variant considered here is given in [8]. At least one of these two variants is discussed in almost all books on computational geometry. Furthermore, it is discussed in [5].

First we present a scan procedure computing a subpart of the counterclockwise sequence of extreme points for a presorted range of points.

⟨*graham scan inline declaration*⟩≡

```
template <class BidirectionalIterator, class OutputIterator, class R>
inline
OutputIterator
CGAL_ch__graham_andrew_scan( BidirectionalIterator first,
                             BidirectionalIterator last,
                             OutputIterator        result,
                             CGAL_Point_2<R>* )
{
```

```
  return CGAL_ch_graham_andrew_scan( first, last, result,
                                     CGAL_convex_hull_traits_2<R>() );
}
```

```
template <class BidirectionalIterator, class OutputIterator>
inline
OutputIterator
CGAL_ch_graham_andrew_scan( BidirectionalIterator first,
                            BidirectionalIterator last,
                            OutputIterator        result )
{ return CGAL_ch__graham_andrew_scan( first, last, result, value_type(first) ); }
```

⟨*graham scan declaration with traits*⟩≡

```
template <class BidirectionalIterator, class OutputIterator, class Traits>
OutputIterator
CGAL_ch_graham_andrew_scan( BidirectionalIterator first,
                            BidirectionalIterator last,
                            OutputIterator        result,
                            const Traits& ch_traits );
```

`CGAL_ch_graham_andrew_scan` assumes that the range defined by `first` and `last` contains at least two different points. Let $p =$`*first` and $q =$`*--last`. `CGAL_ch_graham_andrew_scan` assumes that the points are sorted along the line pq, where p is the smallest and q is the largest point in the order defined with respect to line pq. `CGAL_ch_graham_andrew_scan` computes the sorted sequence of extreme points which are not left of pq and reports this sequence in a range starting at `result`. The sequence reported starts with p, point q is omitted.

Points are scaned in sorted order. A sequence of points is maintained which form a counterclockwise convex chain. If a new point is considered points from the tail of the sequence are removed until we get a strongly convex chain again. Thus a stack would be an appropriate data structure for maintaining the points in the chain. We use a slighly improved version of the algorithm that does an additional sidedness test (with the line through the last point on the chain and `q`) in order to discard non-extreme points as early as possible. For a correctness proof of the implemented version see [36], pages 93–97.

⟨*graham_andrew_scan*⟩≡

```
template <class BidirectionalIterator, class OutputIterator, class Traits>
OutputIterator
CGAL_ch_graham_andrew_scan( BidirectionalIterator first,
                            BidirectionalIterator last,
                            OutputIterator        result,
                            const Traits&         ch_traits)
{
  ⟨graham scan typedefs⟩
  ⟨stack and iterator declaration⟩
  ⟨initialisation⟩
  ⟨compute extreme points⟩
  ⟨return extreme points⟩
}
```

Instead of a stack adaptor (`stack<vector<BidirectionalIterator> >`) we use a vector. This implies less obvious operations (resp. names), but it allows us to copy the stack easily without reversing it.

⟨*stack and iterator declaration*⟩≡

```
vector< BidirectionalIterator >    S;
BidirectionalIterator              alpha;
BidirectionalIterator              beta;
BidirectionalIterator              iter;
CGAL_ch_precondition( first != last );
CGAL_ch_precondition( CGAL_successor(first) != last );
```

We check the precondition that the elements at the ends of the range are different. To have access to both of these elements we move `last` backwards[3]

⟨*initialisation*⟩≡
```
--last;
CGAL_ch_precondition( *first != *last );
S.push_back( last  );
S.push_back( first );
Leftturn    leftturn;
```

We start searching for a point right of pq. If we have found one, we push it on the stack.

⟨*compute extreme points*⟩≡
```
iter = first;
do
{
    ++iter;
}
while (( iter != last ) && !leftturn(*last, *first, *iter) );
if ( iter != last )
{
    S.push_back( iter );
    ⟨actual scan⟩
}
```

`alpha` is the content of the topmost elements on stack `S`, `stack_rev_iter` is put on the element below the topmost, and `beta` is assigned the content of it. `stack_rev_iter` cannot be correctly maintained during the update operations on the stack (by moving it backward and forward with each push- and pop-operation respectively) but has to be moved from `S.rbegin()` to the element below the topmost each time, because insertions and deletions might invalidate (reverse-)iterators on the `vector`.

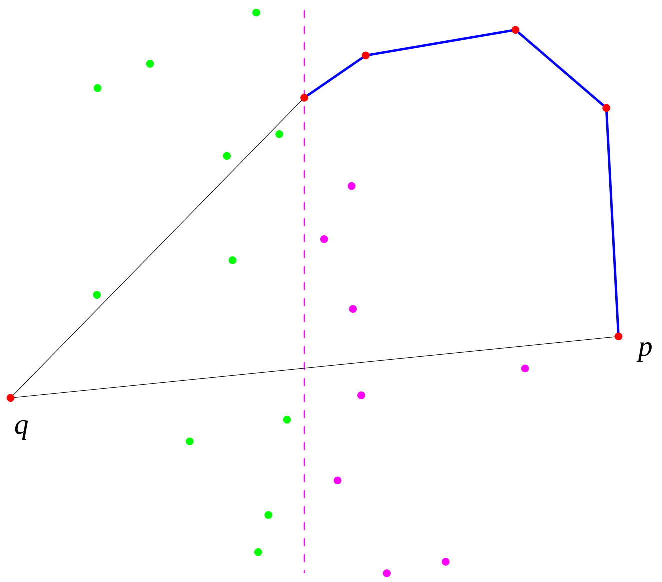

Figure 3: Snapshot of a graham scan on the upper hull.

[3] On page 24-5, the Dec 96 DRAFT working paper for the forthcoming C++ standard states as precondition for operation `--r` that there exists `s` such that `++s == r`. The past-the-end position has such a predecessor.

⟨*actual scan*⟩≡

```
typedef typename vector< BidirectionalIterator >::reverse_iterator  rev_iterator;
rev_iterator  stack_rev_iter = S.rbegin();
alpha = iter;
beta  = *++stack_rev_iter;

for ( ++iter ; iter != last; ++iter )
{
    if ( leftturn(*alpha, *iter, *last) )
    {
        while ( !leftturn(*beta, *alpha, *iter) )
        {
            S.pop_back();
            alpha = beta;
            stack_rev_iter = S.rbegin();
            beta  = *++stack_rev_iter;
            CGAL_ch_assertion(S.size() >= 2);
        }
        S.push_back( iter  );
        beta = alpha;
        alpha = iter;
    }
}
```

q is not copied in the counterclockwise ordered sequence of extreme points.

⟨*return extreme points*⟩≡

```
typedef typename vector< BidirectionalIterator >::iterator  std_iterator;
std_iterator  stack_iter = S.begin();
⟨define res⟩
for ( ++stack_iter;  stack_iter != S.end(); ++stack_iter)
{
    *res++ =  **stack_iter;
}
⟨return res⟩
```

There is a special internally used version, that takes a reference to `OutputIterator`. Whenever this version is used, the calling function ensures that **`result`** is not a temporary object. We need this version to keep the requirements on the iterator low. Remember that output iterators can be copied, but not assigned. Indeed, assignment has different semantics for insert iterators like **`back_inserter`**! For this internal varaint there is no default version provided!

⟨*graham scan with OutputIterator reference declaration with traits*⟩≡

```
template <class BidirectionalIterator, class OutputIterator, class Traits>
OutputIterator
CGAL_ch__ref_graham_andrew_scan( BidirectionalIterator first,
                                 BidirectionalIterator last,
                                 OutputIterator&       result,
                                 const Traits& );
```

⟨*graham scan with OutputIterator reference*⟩≡

```
template <class BidirectionalIterator, class OutputIterator, class Traits>
OutputIterator
CGAL_ch__ref_graham_andrew_scan( BidirectionalIterator first,
                                 BidirectionalIterator last,
                                 OutputIterator&       result,
                                 const Traits&         ch_traits)
{
  ⟨graham scan typedefs⟩
  ⟨stack and iterator declaration⟩
```

```
  ⟨initialisation⟩
  ⟨compute extreme points⟩
  ⟨return extreme points⟩
}
```

The Graham-Andrew algorithms for computing the convex hull points sorts the points and then uses the scan procedure described above to compute upper and lower hull. We first give the declaration and provide the handling of the default version.

⟨*graham hull declaration with traits*⟩≡

```
template <class InputIterator, class OutputIterator, class Traits>
OutputIterator
CGAL_ch_graham_andrew( InputIterator  first,
                       InputIterator  last,
                       OutputIterator result,
                       const Traits& ch_traits );
```

⟨*graham hull inline declaration*⟩≡

```
template <class InputIterator, class OutputIterator, class R>
inline
OutputIterator
CGAL_ch__graham_andrew( InputIterator     first,
                        InputIterator     last,
                        OutputIterator    result,
                        CGAL_Point_2<R>* )
{
  return CGAL_ch_graham_andrew(first, last, result,
                               CGAL_convex_hull_traits_2<R>());
}
template <class InputIterator, class OutputIterator>
inline
OutputIterator
CGAL_ch_graham_andrew( InputIterator  first,
                       InputIterator  last,
                       OutputIterator result )
{ return CGAL_ch__graham_andrew( first, last, result, value_type(first) ); }
```

At first, the points are sorted. After checking for degenerate cases, lower and upper hull are computed by a Graham Andrew scan.

⟨*graham hull*⟩≡

```
template <class InputIterator, class OutputIterator, class Traits>
OutputIterator
CGAL_ch_graham_andrew( InputIterator  first,
                       InputIterator  last,
                       OutputIterator result,
                       const Traits&  ch_traits)
{
  ⟨graham scan typedefs⟩
  ⟨preparing graham scans⟩
  ⟨compute hull parts⟩
}
```

⟨*graham scan typedefs*⟩≡
```
typedef  typename Traits::Less_xy   Less_xy;
typedef  typename Traits::Point_2   Point_2;
typedef  typename Traits::Leftturn  Leftturn;
```

In the sorting step we use the less-than predicate class `Less_xy` from the traits class. We copy the points to a `vector` to sort them. Since we copy the points in a local container, we need an `InputIterator` only in the function interface. After sorting, we check whether all points are equal. `CGAL_ch__ref_graham_andrew_scan` must get at least two different points.

⟨*preparing graham scans*⟩≡
```
if (first == last) return result;
vector< Point_2 >  V;
copy( first, last, back_inserter(V) );
sort( V.begin(), V.end(), Less_xy() );
if ( *(V.begin()) == *(V.rbegin()) )
{
    *result++ = *(V.begin());
    return result;
}
```

Then we call the Graham Andrew scan to compute lower and upper hull, remember counterclockwise. We use an internal version of the Graham Andrew scan, which passes the output iterator by reference.

⟨*compute hull parts*⟩≡
```
⟨define res⟩
CGAL_ch__ref_graham_andrew_scan( V.begin(), V.end(),  res, ch_traits);
CGAL_ch__ref_graham_andrew_scan( V.rbegin(), V.rend(), res, ch_traits);
⟨graham andrew post condition⟩
⟨return res⟩
```

Next we provide the postcondition checking. Since all the convexity checking slows down the algorithm by a constant factor only, it is not labeled "expensive". Since the brute-force containment check is asymptotically slower, it is called "expensive". Member functions `output_so_far_begin()` and `output_so_far_end()` of `CGAL_Tee_for_output_iterator<>` provide access to the computed output.

⟨*graham andrew post condition*⟩≡
```
⟨check convexity of generated output⟩
CGAL_ch_expensive_postcondition( \
    CGAL_ch_brute_force_check_2( \
        V.begin(), V.end(), \
        res.output_so_far_begin(), res.output_so_far_end(), \
        ch_traits));
```

8 Akl Toussaint Algorithm

This algorithm is described in [4]. Moreover, it can be found in [5]. First the extreme points in the coordinate directions (x-y-extremal points) are computed. All points in the 4-gon formed by these points can be thrown away, they cannot be extreme points, cf. Fig. 4. Subproblems are solved by Graham scan. Declarations and definition follow.

⟨*Akl Toussaint alg declaration with traits*⟩≡

```
template <class ForwardIterator, class OutputIterator, class Traits>
OutputIterator
CGAL_ch_akl_toussaint(ForwardIterator first, ForwardIterator last,
                       OutputIterator  result,
                       const Traits& ch_traits);
```

⟨*Akl Toussaint alg inline declaration*⟩≡

```
template <class ForwardIterator, class OutputIterator, class R>
inline
OutputIterator
CGAL_ch__akl_toussaint(ForwardIterator first, ForwardIterator last,
                        OutputIterator  result,
                        CGAL_Point_2<R>* )
{
  return CGAL_ch_akl_toussaint(first, last, result,
                                CGAL_convex_hull_traits_2<R>() );
}

template <class ForwardIterator, class OutputIterator>
inline
OutputIterator
CGAL_ch_akl_toussaint(ForwardIterator first, ForwardIterator last,
                       OutputIterator  result)
{ return CGAL_ch__akl_toussaint( first, last, result, value_type(first) ); }
```

⟨*Akl Toussaint alg*⟩≡

```
template <class ForwardIterator, class OutputIterator, class Traits>
OutputIterator
CGAL_ch_akl_toussaint(ForwardIterator first, ForwardIterator last,
                       OutputIterator  result,
                       const Traits& ch_traits)
{
  ⟨Akl Toussaint typedefs⟩
  ⟨compute x-y-extremal points⟩
  ⟨break into subproblems - throw away⟩
  ⟨solve subproblems by graham scan⟩
}
```

⟨*Akl Toussaint typedefs*⟩≡

```
typedef  typename Traits::Point_2                 Point_2;
typedef  typename Traits::Right_of_line           Right_of_line;
typedef  typename Traits::Less_xy                 Less_xy;
typedef  typename Traits::Greater_xy              Greater_xy;
typedef  typename Traits::Less_yx                 Less_yx;
typedef  typename Traits::Greater_yx              Greater_yx;
```

Actually we compute iterators, not x-y-extremal points. In addition we check for some degenerate cases, especially the one point convex hull. After this check we know that the hull consists of two points at least.

⟨*compute x-y-extremal points*⟩≡

```
if (first == last) return result;
ForwardIterator n, s, e, w;
CGAL_ch_nswe_point( first, last, n, s, w, e, ch_traits);
if ( *n == *s )
```

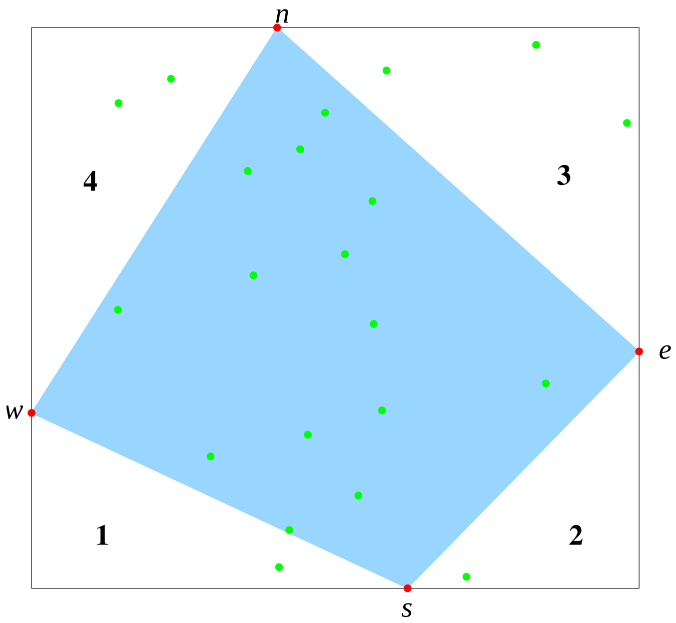

Figure 4: Computation of extremal points and "throw way" in the Akl-Toussaint algorithm.

```
{
    *result++ = *w;
    return result;
}
```

We create containers for the remaining regions. `region3` is the north-east region, `region4` the north-west region, `region1` the south-west region, and `region2` the south-east region. The x-y-extremal points are on the splitting lines, thus they need special handling.

⟨*break into subproblems - throw away*⟩≡

```
vector< Point_2 > region1;
vector< Point_2 > region2;
vector< Point_2 > region3;
vector< Point_2 > region4;
region1.reserve(16);
region2.reserve(16);
region3.reserve(16);
region4.reserve(16);
region1.push_back( *w);
region2.push_back( *s);
region3.push_back( *e);
region4.push_back( *n);
```

Now we partition along the splitting lines.

⟨*break into subproblems - throw away*⟩+≡

```
Right_of_line  rol_we( *w, *e);
Right_of_line  rol_en( *e, *n);
Right_of_line  rol_nw( *n, *w);
Right_of_line  rol_ws( *w, *s);
Right_of_line  rol_se( *s, *e);

CGAL_ch_postcondition_code( ForwardIterator save_first = first; )

for ( ; first != last; ++first )
{
```

```
        if ( rol_we( *first ) )
        {
            if ( rol_ws( *first ) )        region1.push_back( *first );
            else if ( rol_se( *first ) )   region2.push_back( *first );
        }
        else
        {
            if ( rol_en( *first ) )        region3.push_back( *first );
            else if ( rol_nw( *first ) )   region4.push_back( *first );
        }
    }
```

The remaining subproblems are solved by graham scan.

⟨*solve subproblems by graham scan*⟩≡
  ⟨*define res*⟩
  ⟨*sort points in regions*⟩
  ⟨*scan non-empty regions*⟩
  ⟨*return res*⟩

⟨*sort points in regions*⟩≡

```
    sort( CGAL_successor(region1.begin() ), region1.end(), Less_xy() );
    sort( CGAL_successor(region2.begin() ), region2.end(), Less_xy() );
    sort( CGAL_successor(region3.begin() ), region3.end(), Greater_xy() );
    sort( CGAL_successor(region4.begin() ), region4.end(), Greater_xy() );
```

⟨*scan non-empty regions*⟩≡

```
    if ( *w != *s )
    {
        region1.push_back( *s );
        CGAL_ch__ref_graham_andrew_scan( region1.begin(), region1.end(),
                                         res, ch_traits);
    }
    if ( *s != *e )
    {
        region2.push_back( *e );
        CGAL_ch__ref_graham_andrew_scan( region2.begin(), region2.end(),
                                         res, ch_traits);
    }
    if ( *e != *n )
    {
        region3.push_back( *n );
        CGAL_ch__ref_graham_andrew_scan( region3.begin(), region3.end(),
                                         res, ch_traits);
    }
    if ( *n != *w )
    {
        region4.push_back( *w );
        CGAL_ch__ref_graham_andrew_scan( region4.begin(), region4.end(),
                                         res, ch_traits);
    }
    CGAL_ch_postcondition_code( first = save_first; )
    ⟨check convexity of generated output⟩
    CGAL_ch_expensive_postcondition( \
        CGAL_ch_brute_force_check_2( \
            first, last, \
            res.output_so_far_begin(), res.output_so_far_end(), \
            ch_traits)
    );
```

9 Eddy's Algorithm

Eddy's algorithm is presented in [21]. Furthermore, it can be found in [5] or in [41]. It is the two-dimensional quickhull [41, 11].

⟨*Eddy's alg declaration with traits*⟩≡

```
template <class InputIterator, class OutputIterator, class Traits>
OutputIterator
CGAL_ch_eddy(InputIterator first, InputIterator last,
             OutputIterator  result,
             const Traits& ch_traits);
```

⟨*Eddy's alg inline declaration*⟩≡

```
template <class InputIterator, class OutputIterator, class R>
inline
OutputIterator
CGAL_ch__eddy(InputIterator first, InputIterator last,
              OutputIterator  result,
              CGAL_Point_2<R>* )
{
  return CGAL_ch_eddy(first, last, result, CGAL_convex_hull_traits_2<R>() );
}

template <class InputIterator, class OutputIterator>
inline
OutputIterator
CGAL_ch_eddy(InputIterator first, InputIterator last, OutputIterator  result)
{
  return CGAL_ch__eddy( first, last, result, value_type(first) );
}
```

Eddy's algorithms works recursively. In each step we have two extreme points a and b and compute the points c with maximum distance to the line through a and b. Ties are broken by comparing distances to a. Clearly, c is an extreme point and the points in the triangle Δacb can be discarded from further consideration. Recursively the extreme points between a and c (c and b) are computed considering points only which are right of the line through a and c (right of the line through c and b, respectively). As initial extreme points we use the x-extremal points.

⟨*Eddy's alg*⟩≡

```
template <class InputIterator, class OutputIterator, class Traits>
OutputIterator
CGAL_ch_eddy(InputIterator first, InputIterator last,
             OutputIterator  result,
             const Traits& ch_traits)
{
  ⟨Eddy typedefs⟩
  ⟨copy to internal list⟩
  ⟨compute x-extremal points⟩
  ⟨partition points - qh 2D⟩
  ⟨recursive call of the quickhull step⟩
  ⟨return surviving points - qh 2D⟩
}
```

⟨*Eddy typedefs*⟩≡

```
typedef  typename Traits::Point_2                   Point_2;
typedef  typename Traits::Right_of_line             Right_of_line;
typedef  typename Traits::Less_dist_to_line         Less_dist;
```

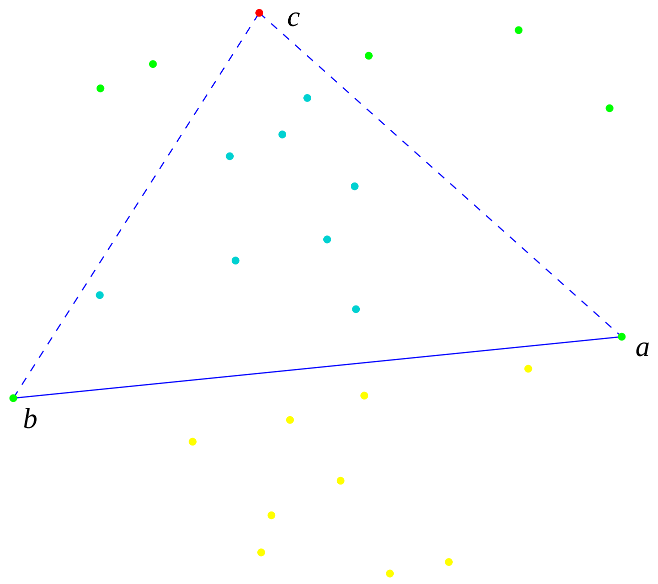

Figure 5: A snapshot of Eddy's algorithm.

Internally we use a list to make deletions efficient.

⟨*copy to internal list*⟩≡
```
if (first == last) return result;
list< Point_2 >   L;
copy( first, last, back_inserter(L) );
```

As in Andrew's variant of Graham's algorithm the first step is the computation of a pair of extremal points. If `*w==*e`, all points are equal, since we compare points lexicographically, i.e. we have a total order.

⟨*compute x-extremal points*⟩≡
```
typedef  list< Point_2 >::iterator  list_iterator;
list_iterator   w, e;
CGAL_ch_we_point(L.begin(), L.end(), w, e, ch_traits);
Point_2 wp = *w;
Point_2 ep = *e;
if ( wp == ep )
{
    *result++ = wp;
    return result;
}
```

Next we partition the points such that all points below the line through the x-extremal points are in the list in front of the points above this line. The x-extremal points are handled separately.

⟨*partition points - qh 2D*⟩≡
```
L.erase(w);
L.erase(e);
e = partition(L.begin(), L.end(), Right_of_line( wp, ep) );
L.push_front(wp);
e = L.insert(e, ep);
```

For the recursion we use function `CGAL_ch__recursive_eddy()`. It does not only compute a new extremal point and rearrange points, but also deletes points and recursively solves the arising subproblems. `CGAL_ch__recursive_eddy()` gets the list `L`, on which Eddy's algorithm is operating internally, and two list iterators `a_it` and `b_it` on `L`. For the sake of efficiency, it has the precondition, that the range between the bounding iterators `a_it` and `b_it` contains at least one point to the right of the line through `*a_it` and `*b_it`.

⟨*recursive eddy*⟩≡

```
template <class List, class ListIterator, class Traits>
void
CGAL_ch__recursive_eddy(List& L,
                        ListIterator  a_it, ListIterator  b_it,
                        const Traits& ch_traits)
{
  ⟨Eddy typedefs⟩
  ⟨assert precondition - recursive eddy⟩
  ⟨compute new extreme point - qh 2D⟩
  ⟨partition and throw away - qh 2D⟩
  ⟨recursively solve subproblems - qh 2D⟩
}
```

For the sake of ease of use and debugging, the precondition, that at least one points is right of the line through `*a_it` and `*b_it`, is checked.

⟨*assert precondition - recursive eddy*⟩≡

```
CGAL_ch_precondition( \
  find_if(a_it, b_it, Right_of_line(*a_it,*b_it)) != b_it);
```

The point with maximal distance to the line through `*a_it` and `*b_it` is a new extreme points. Ties are broken by comparing the distance to point `*a_it`. Since we are interested in the point right of a line with maximum unsigned distance, we are in fact interested in the point with minimum signed distance to the line. Therefore the typedef for `ch_Less_dist` maps it to a `CGAL_Less_negative_dist...` predicate.

⟨*compute new extreme point - qh 2D*⟩≡

```
ListIterator f_it = CGAL_successor(a_it);
ListIterator c_it = max_element( f_it, b_it, Less_dist(*a_it,*b_it) );
Point_2 c = *c_it;
```

Let $a =$`*a_it` and $b =$`*b_it` and c be the new extreme point between a and b. We place all points right of the line through a and c before all points which are right of the line through b and c. Then c is inserted between these two groups. All points inside the triangle Δacb (those points which are in `L` between `a_it` and `b_it` and are neither right of ac nor right of cb) are deleted, besides c.

⟨*partition and throw away - qh 2D*⟩≡

```
c_it = partition( f_it, b_it, Right_of_line(*a_it, c ) );
f_it = partition( c_it, b_it, Right_of_line(c, *b_it ) );
c_it = L.insert(c_it, c);
L.erase( f_it, b_it );
```

Next we recursively solve subproblems.

⟨*recursively solve subproblems - qh 2D*⟩≡

```
if ( CGAL_successor(a_it) != c_it )
{
    CGAL_ch__recursive_eddy( L, a_it, c_it, ch_traits);
}
if ( CGAL_successor(c_it) != b_it )
{
    CGAL_ch__recursive_eddy( L, c_it, b_it, ch_traits);
}
```

Now let's go back to `CGAL_ch__eddy()`. In the partition step we use the `Right_of_line` predicate. The points not satisfying this predicate are above the line through `ep` and `wp` or on it. For the recursion we have to make sure that there is at least one point strictly above the line. Analogously we are allowed to call `CGAL_ch__recursive_eddy()` for the lower hull only if there are points below the line throught the left-most and right-most point.

⟨*recursive call of the quickhull step*⟩≡

```
if ( CGAL_successor(L.begin()) != e )
{
    CGAL_ch__recursive_eddy( L, L.begin(), e, ch_traits);
}
w = find_if( e, L.end(), Right_of_line( ep, wp) );
if ( w == L.end() )
{
    L.erase( ++e, L.end() );
    return copy( L.begin(), L.end(), result );
}
w = L.insert(L.end(), wp);
CGAL_ch__recursive_eddy( L, e, w, ch_traits);
```

Finally we copy the list of extreme points to the range starting at `result`. Note that `wp` is in `L` twice; we exclude the second occurence of `wp` at `w` at the (real) end of the list.

⟨*return surviving points - qh 2D*⟩≡

```
⟨Eddy post condition⟩
return copy( L.begin(), w, result );
```

In Eddy's algorithm we can really check the computed result, since the sequence of extreme points is maintained in a local list.

⟨*Eddy post condition*⟩≡

```
CGAL_ch_postcondition( \
    CGAL_is_ccw_strongly_convex_2( L.begin(), w, ch_traits) );
CGAL_ch_expensive_postcondition( \
    CGAL_ch_brute_force_check_2( first, last, \
                                 L.begin(), w, ch_traits ) );
```

10 Bykat's Algorithm

This is a non-recursive variant of Eddy's algorithm. It was published independently in [15], see also [40].

⟨*Bykat alg declaration with traits*⟩≡

```
template <class InputIterator, class OutputIterator, class Traits>
OutputIterator
CGAL_ch_bykat(InputIterator first, InputIterator last,
              OutputIterator  result,
              const Traits& ch_traits);

template <class InputIterator, class OutputIterator, class Traits>
OutputIterator
CGAL_ch_bykat_with_threshold(InputIterator first, InputIterator last,
                             OutputIterator  result,
                             const Traits& ch_traits);
```

⟨*Bykat alg inline declaration*⟩≡

```
template <class InputIterator, class OutputIterator, class R>
inline
OutputIterator
CGAL_ch__bykat(InputIterator first, InputIterator last,
              OutputIterator  result,
              CGAL_Point_2<R>* )
{
  return CGAL_ch_bykat(first, last, result, CGAL_convex_hull_traits_2<R>() );
}

template <class InputIterator, class OutputIterator>
inline
OutputIterator
CGAL_ch_bykat(InputIterator first, InputIterator last, OutputIterator  result)
{
  return CGAL_ch__bykat( first, last, result, value_type(first) );
}

template <class InputIterator, class OutputIterator, class R>
inline
OutputIterator
CGAL_ch__bykat_with_threshold(InputIterator first, InputIterator last,
                              OutputIterator  result,
                              CGAL_Point_2<R>* )
{
  return CGAL_ch_bykat_with_threshold(first, last, result,
                                      CGAL_convex_hull_traits_2<R>() );
}

template <class InputIterator, class OutputIterator>
inline
OutputIterator
CGAL_ch_bykat_with_threshold(InputIterator first, InputIterator last,
                             OutputIterator  result)
{
  return CGAL_ch__bykat_with_threshold( first, last, result,
                                        value_type(first) );
}
```

⟨*nonrecursive Bykat-Eddy*⟩≡

```
template <class InputIterator, class OutputIterator, class Traits>
OutputIterator
CGAL_ch_bykat(InputIterator first, InputIterator last,
              OutputIterator  result,
              const Traits& ch_traits)
{
  typedef typename Traits::Point_2              Point_2;
  typedef typename Traits::Right_of_line        Right_of_line;
  typedef typename Traits::Less_dist_to_line    Less_dist;

  if (first == last) return result;

  vector< Point_2 >       P;      // Points in subsets
  vector< Point_2 >       H;      // right endpoints of subproblems
  P.reserve(16);
  H.reserve(16);

  typedef vector< Point_2 >::iterator   PointIterator;
  vector< PointIterator > L;      // start of subset range
  vector< PointIterator > R;      // end of subset range
  L.reserve(16);
  R.reserve(16);
  PointIterator           l;
```

```
    PointIterator          r;
    Point_2                a,b,c;

    copy(first,last,back_inserter(P));
    CGAL_ch_we_point(P.begin(), P.end(), l, r, ch_traits);
    a = *l;
    b = *r;
    if ( a == b)
    {
        *result++ = a;
        return result;
    }
    ⟨define res⟩
    H.push_back( a );
    L.push_back( P.begin() );
    R.push_back( l = partition( P.begin(), P.end(), Right_of_line(b,a) ) );
    r = partition( l, P.end(), Right_of_line(a,b) );

    for (;;)
    {
        if ( l != r)
        {
            c = *max_element( l, r, Less_dist(a,b) );
            H.push_back( b );
            L.push_back( l );
            R.push_back( l = partition(l, r, Right_of_line(c,b)) );
            r = partition(l, r, Right_of_line(a,c));
            b = c;
        }
        else
        {
            *res++ = a;
            if ( L.empty() ) break;
            a = b;
            b = H.back(); H.pop_back();
            l = L.back(); L.pop_back();
            r = R.back(); R.pop_back();
        }
    }
    ⟨check convexity of generated output⟩
    CGAL_ch_expensive_postcondition( \
        CGAL_ch_brute_force_check_2( \
            P.begin(), P.end(), \
            res.output_so_far_begin(), res.output_so_far_end(), \
            ch_traits));
    ⟨return res⟩
  }
```

Besides the version, there is an experimental version, that stops at a certain size of the subproblems and calls Graham scan to solve the subproblems.

⟨*bykat with threshold*⟩≡

```
  #define CGAL_ch_THRESHOLD 10
  template <class InputIterator, class OutputIterator, class Traits>
  OutputIterator
  CGAL_ch_bykat_with_threshold(InputIterator   first, InputIterator last,
                               OutputIterator  result,
                               const Traits&   ch_traits)
  {
    typedef typename Traits::Point_2              Point_2;
    typedef typename Traits::Right_of_line        Right_of_line;
    typedef typename Traits::Less_dist_to_line    Less_dist;
```

```
typedef typename Traits::Less_xy               Less_xy;
typedef typename Traits::Greater_xy            Greater_xy;
typedef typename vector< Point_2 >::iterator   PointIterator;

if (first == last) return result;

vector< Point_2 >       P;      // points in subsets
vector< Point_2 >       H;      // right endpoints of subproblems
P.reserve(16);
H.reserve(16);
vector< PointIterator > L;      // start of subset range
vector< PointIterator > R;      // end of subset range
L.reserve(16);
R.reserve(16);
PointIterator           l;
PointIterator           r;
Point_2                 a,b,c;
PointIterator           Pbegin, Pend;

P.push_back(Point_2() );
copy(first,last,back_inserter(P));
P.push_back(Point_2() );
Pbegin = CGAL_successor(P.begin());
Pend   = CGAL_predecessor(P.end());
CGAL_ch_we_point(Pbegin, Pend, l, r, ch_traits);
a = *l;
b = *r;
if ( a == b)
{
    *result++ = a;
    return result;
}
⟨define res⟩
H.push_back( a );
L.push_back( Pbegin );
R.push_back( l = partition( Pbegin, Pend, Right_of_line(b,a) ) );
r = partition( l, Pend, Right_of_line(a,b) );

for (;;)
{
    if ( l != r)
    {
        if ( r-l > CGAL_ch_THRESHOLD )
        {
            c = *max_element( l, r, Less_dist(a,b) );
            H.push_back( b );
            L.push_back( l );
            R.push_back( l = partition(l, r, Right_of_line(c,b)) );
            r = partition(l, r, Right_of_line(a,c));
            b = c;
        }
        else
        {
            swap( a, *--l);
            swap( b, *++r);
            if ( Less_xy()(*l,*r) )
            {
                sort(CGAL_successor(l), r, Less_xy() );
            }
            else
            {
                sort(CGAL_successor(l), r, Greater_xy() );
            }
            CGAL_ch__ref_graham_andrew_scan(l, CGAL_successor(r),
```

```
                                        res, ch_traits);
            swap( a, *l);
            swap( b, *r);
            if ( L.empty() ) break;
            a = b;
            b = H.back(); H.pop_back();
            l = L.back(); L.pop_back();
            r = R.back(); R.pop_back();
        }
    }
    else
    {
        *res++ = a;
        if ( L.empty() ) break;
        a = b;
        b = H.back(); H.pop_back();
        l = L.back(); L.pop_back();
        r = R.back(); R.pop_back();
    }
  }
  ⟨check convexity of generated output⟩
  CGAL_ch_expensive_postcondition( \
      CGAL_ch_brute_force_check_2( \
          Pbegin, Pend, \
          res.output_so_far_begin(), res.output_so_far_end(), \
          ch_traits));
  ⟨return res⟩
}
```

11 Jarvis' Algorithm

Jarvis' algorithm is known as gift-wrapping method or as Jarvis' march (along the conex hull), see also Fig. 6. The original reference is [30]; the method is discussed in [41] and [5] as well. It is more or less provided for the sake of completeness. It is called gift-wrapping method since the intuition is to rotate a line around an extreme points until another point is hit, thereby wrapping the set of points like a gift.

We start with a version that computes a subsequence of the extreme points between two given points. Since a `Traits::Point_2` in a parameter list of a template function stresses current compilers, we have added an additional template parameter `Point`.

⟨*jarvis march declaration with traits*⟩≡

```
template <class ForwardIterator, class OutputIterator,
          class Point, class Traits>
OutputIterator
CGAL_ch_jarvis_march(ForwardIterator first, ForwardIterator last,
                     const Point& start_p, const Point& stop_p,
                     OutputIterator  result,
                     const Traits& ch_traits);
```

⟨*jarvis march inline declaration*⟩≡

```
template <class ForwardIterator, class OutputIterator, class R>
inline
OutputIterator
CGAL_ch_jarvis_march(ForwardIterator first, ForwardIterator last,
                     const CGAL_Point_2<R>& start_p,
                     const CGAL_Point_2<R>& stop_p,
```

```
                        OutputIterator  result )
{
  return CGAL_ch_jarvis_march( first, last,
                               start_p, stop_p,
                               results, CGAL_convex_hull_traits_2<R>() );
}
```

In the implementation we use the binary predicate `Less_rotate_ccw`. The predicate is used in the computation of the first point hit by a tangent line rotated clouterclockwise around an extreme point.

⟨*jarvis march*⟩≡

```
template <class ForwardIterator, class OutputIterator,
          class Point, class Traits>
OutputIterator
CGAL_ch_jarvis_march(ForwardIterator first, ForwardIterator last,
                     const Point& start_p,
                     const Point& stop_p,
                     OutputIterator  result,
                     const Traits& ch_traits)
{
  if (first == last) return result;
  typedef   typename Traits::Less_rotate_ccw       Less_rotate_ccw;
  typedef   typename Traits::Point_2               Point_2;
  CGAL_CH_USE_ARGUMENT(ch_traits);
  ⟨define res⟩
  ⟨Jarvis march assertion 1⟩
  Less_rotate_ccw  rotation_predicate( start_p );
  *res++ = start_p;
  ⟨Jarvis march assertion 2⟩
  ForwardIterator it = min_element( first, last, rotation_predicate );
  while ( *it != stop_p )
```

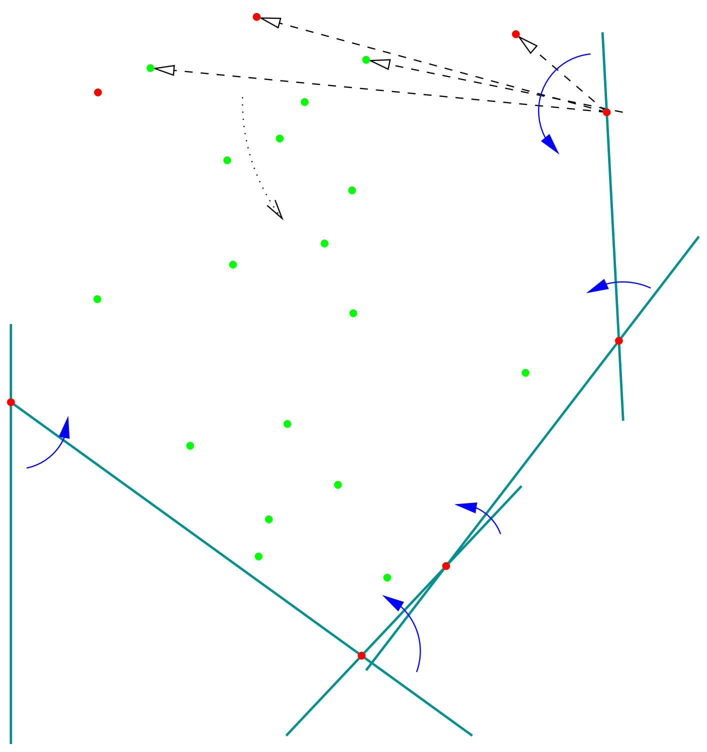

Figure 6: A snapshot of a Jarvis' march along the hull.

```
  {
      ⟨Jarvis march assertion 3⟩
      *res++ = *it;
      ⟨Jarvis march assertion 4⟩
      rotation_predicate = Less_rotate_ccw( *it );
      it = min_element( first, last, rotation_predicate );
  }
  ⟨Jarvis march assertion 5⟩
  ⟨return res⟩
}
```

Jarvis' march has various assertions and code fragments for asssertions.

⟨*Jarvis march assertion 1*⟩≡

```
CGAL_ch_assertion_code( \
    int count_points = 0; )
CGAL_ch_assertion_code( \
    for (ForwardIterator fit = first; fit!= last; ++fit) ++count_points; )
```

⟨*Jarvis march assertion 2*⟩≡

```
CGAL_ch_assertion_code( \
    int constructed_points = 1; )
CGAL_ch_exactness_assertion_code( \
    Point previous_point = start_p; )
```

⟨*Jarvis march assertion 3*⟩≡

```
CGAL_ch_exactness_assertion( \
    *it != previous_point );
CGAL_ch_exactness_assertion_code( \
    previous_point = *it; )
```

⟨*Jarvis march assertion 4*⟩≡

```
CGAL_ch_assertion_code( \
    ++constructed_points;)
CGAL_ch_assertion( \
    constructed_points <= count_points + 1 );
```

⟨*Jarvis march assertion 5*⟩≡

```
⟨check convexity of generated output⟩
CGAL_ch_expensive_postcondition( \
    CGAL_ch_brute_force_check_2(
        first, last, \
        res.output_so_far_begin(), res.output_so_far_end(), \
        ch_traits));
```

In the algorithm computing all extreme points we compute an extreme point *start* and use the march procedure described above to compute all extreme points counterclockwise between *start* and *start* itself.

⟨*jarvis alg declaration with traits*⟩≡

```
template <class ForwardIterator, class OutputIterator, class Traits>
OutputIterator
CGAL_ch_jarvis(ForwardIterator first, ForwardIterator last,
               OutputIterator  result,
               const Traits& ch_traits);
```

⟨*jarvis alg inline declaration*⟩≡

```
template <class ForwardIterator, class OutputIterator, class R>
inline
OutputIterator
CGAL_ch__jarvis(ForwardIterator first, ForwardIterator last,
                OutputIterator  result,
                CGAL_Point_2<R>* )
{
  return CGAL_ch_jarvis( first, last, result, CGAL_convex_hull_traits_2<R>() );
}
template <class ForwardIterator, class OutputIterator>
inline
OutputIterator
CGAL_ch_jarvis(ForwardIterator first, ForwardIterator last,
               OutputIterator  result)
{ return CGAL_ch__jarvis( first, last, result, value_type(first) ); }
```

⟨*jarvis alg*⟩≡

```
template <class ForwardIterator, class OutputIterator, class Traits>
OutputIterator
CGAL_ch_jarvis(ForwardIterator first, ForwardIterator last,
               OutputIterator  result,
               const Traits& ch_traits)
{
  if (first == last) return result;
  ForwardIterator start;
  CGAL_ch_w_point(first, last, start, ch_traits);
  return CGAL_ch_jarvis_march( first, last, *start, *start, result, ch_traits);
}
```

12 Convex Hull Traits Models

The traits of the convex hull functions are described in Section 3. Here we present concrete models of the abstract concept convex hull traits. There are default versions that operate on `CGAL_Point_2<R>` points from the CGAL-kernel. The default version of our convex hull functions, i.e. the functions without a traits argument, are reduced to the implementations of the functions with traits class argument, where a default convex hull traits class is used. The default traits class uses several function objects corresponding to predicate functions defined in the CGAL-kernel. We present these function objects for the sake of completeness in Appendix D.

12.1 Default Traits

We next give the implementation of the default traits class. Since the default point type `CGAL_Point_2<R>` is parameterized by a representation class, this default convex hull traits class is parametrized by a representation class, too. The class template is called `CGAL_convex_hull_traits_2<R>`.

All the primitives used in the default do not use intermediate result, but recompute from the original data. For instance, sidedness with respect to a line through two given points is always reduced to an orientation test. No line is constructed, i.e. no line coefficients are precomputed to simplify further sidedness tests with the same line.

⟨*convex hull traits*⟩≡

```
template <class _R>
class CGAL_convex_hull_traits_2 : public _R
{
public:
```

```
    typedef   _R                                        R;
    typedef   CGAL_Point_2<R>                           Point_2;
    typedef   CGAL_p_Less_xy<Point_2>                   Less_xy;
    typedef   CGAL_p_Less_yx<Point_2>                   Less_yx;
    typedef   CGAL_p_Greater_xy<Point_2>                Greater_xy;
    typedef   CGAL_p_Greater_yx<Point_2>                Greater_yx;
    typedef   CGAL_p_Right_of_line_2p<Point_2>          Right_of_line;
    typedef   CGAL_p_Less_negative_dist_to_line_2p<Point_2> Less_dist_to_line;
    typedef   CGAL_p_Less_rotate_ccw<Point_2>           Less_rotate_ccw;
    typedef   CGAL_p_Leftturn<Point_2>                  Leftturn;
    typedef   CGAL_p_Rightturn<Point_2>                 Rightturn;
    typedef   CGAL_Segment_2<R>                         Segment_2;
  };
```

12.2 Further Convex Hull Traits Models for CGAL

In addition, there is an alternative convex hull traits class using constructive primitives, e.g. sidedness tests where a line equation is computed.

⟨*constructive convex hull traits*⟩≡

```
  template <class _R>
  class CGAL_convex_hull_constructive_traits_2 : public _R
  {
  public:
    typedef   _R                                        R;
    typedef   CGAL_Point_2<R>                           Point_2;
    typedef   CGAL_p_Less_xy<Point_2>                   Less_xy;
    typedef   CGAL_p_Less_yx<Point_2>                   Less_yx;
    typedef   CGAL_p_Greater_xy<Point_2>                Greater_xy;
    typedef   CGAL_p_Greater_yx<Point_2>                Greater_yx;
    typedef   CGAL_r_Right_of_line<R>                   Right_of_line;
    typedef   CGAL_r_Less_negative_dist_to_line<R>      Less_dist_to_line;
    typedef   CGAL_p_Less_rotate_ccw<Point_2>           Less_rotate_ccw;
    typedef   CGAL_p_Leftturn<Point_2>                  Leftturn;
    typedef   CGAL_p_Rightturn<Point_2>                 Rightturn;
    typedef   CGAL_Segment_2<R>                         Segment_2;
  };
```

Finally, we provide a specialisation for `CGAL_Cartesian<double>`, which does some (in the context of exact geometric computation stupid) additional tests in the predicates (with suffix `_safer`).

⟨*convex hull traits for CGAL_Cartesian double* ⟩≡

```
  template <class R> class CGAL_convex_hull_traits_2;

  CGAL_TEMPLATE_NULL
  class CGAL_convex_hull_traits_2< CGAL_Cartesian<double> >
    : public CGAL_Cartesian<double>
  {
  public:
    typedef   CGAL_Cartesian<double>                    R;
    typedef   CGAL_Point_2<R>                           Point_2;
    typedef   CGAL_p_Less_xy<Point_2>                   Less_xy;
    typedef   CGAL_p_Less_yx<Point_2>                   Less_yx;
    typedef   CGAL_p_Greater_xy<Point_2>                Greater_xy;
    typedef   CGAL_p_Greater_yx<Point_2>                Greater_yx;
    typedef   CGAL_p_Right_of_line_2p_safer<Point_2>    Right_of_line;
    typedef   CGAL_p_Less_negative_dist_to_line_2p<Point_2> Less_dist_to_line;
    typedef   CGAL_p_Less_rotate_ccw_safer<Point_2>     Less_rotate_ccw;
```

```
  typedef   CGAL_p_Leftturn<Point_2>                        Leftturn;
  typedef   CGAL_p_Rightturn<Point_2>                       Rightturn;
  typedef   CGAL_Segment_2<R>                               Segment_2;
};
```

12.3 Convex Hull Traits Models for LEDA

The traits classes for LEDA use predicate objects defined in Appendix D. All wrapping of LEDA functions is in done in `inline` functions, which are in `.rat_leda_in_CGAL_2.h`. For the sake of completeness, the code is given in Appendix E.

⟨*leda convex hull traits*⟩≡

```
class CGAL_convex_hull_rat_leda_traits_2
{
public:
  typedef   leda_rat_point                                  Point_2;
  typedef   CGAL_p_Less_xy<Point_2>                         Less_xy;
  typedef   CGAL_p_Less_yx<Point_2>                         Less_yx;
  typedef   CGAL_p_Greater_xy<Point_2>                      Greater_xy;
  typedef   CGAL_p_Greater_yx<Point_2>                      Greater_yx;
  typedef   CGAL_p_Right_of_line_2p<Point_2>                Right_of_line;
  typedef   CGAL_p_Less_negative_dist_to_line_2p<Point_2>   Less_dist_to_line;
  typedef   CGAL_p_Less_rotate_ccw<Point_2>                 Less_rotate_ccw;
  typedef   CGAL_p_Leftturn<Point_2>                        Leftturn;
  typedef   CGAL_p_Rightturn<Point_2>                       Rightturn;
  typedef   leda_rat_segment                                Segment_2;
};
```

Moreover, we define a traits class for plain (non-exact and hence deprecated) geometry in LEDA. Again, the actual wrapping is done in Appendix E.

⟨*plain leda convex hull traits*⟩≡

```
class CGAL_convex_hull_leda_traits_2
{
public:
  typedef   leda_point                                      Point_2;
  typedef   CGAL_p_Less_xy<Point_2>                         Less_xy;
  typedef   CGAL_p_Less_yx<Point_2>                         Less_yx;
  typedef   CGAL_p_Greater_xy<Point_2>                      Greater_xy;
  typedef   CGAL_p_Greater_yx<Point_2>                      Greater_yx;
  typedef   CGAL_p_Right_of_line_2p<Point_2>                Right_of_line;
  typedef   CGAL_p_Less_negative_dist_to_line_2p<Point_2>   Less_dist_to_line;
  typedef   CGAL_p_Less_rotate_ccw<Point_2>                 Less_rotate_ccw;
  typedef   CGAL_p_Leftturn<Point_2>                        Leftturn;
  typedef   CGAL_p_Rightturn<Point_2>                       Rightturn;
  typedef   leda_segment                                    Segment_2;
};
```

13 Files

We provide several algorithms for computing the counterclockwise sequence of extreme points. The function `CGAL_convex_hull_points_2()` provides a default algorithm, not to be confused with the default convex hull traits class used in the default versions of all functions. In particular, there is a default version of the function using default algorithms. Since our convex hull functions have different iterator requirements, `iterator_category()` is used to select an appropriate default algorithm.

⟨*extreme_points declaration with traits*⟩≡

```
template <class InputIterator, class OutputIterator, class Traits>
inline
OutputIterator
CGAL___convex_hull_points_2(InputIterator first, InputIterator last,
                            OutputIterator  result,
                            const Traits& ch_traits,
                            input_iterator_tag )
{ return CGAL_ch_bykat(first, last, result, ch_traits); }

template <class InputIterator, class OutputIterator, class Traits>
inline
OutputIterator
CGAL___convex_hull_points_2(InputIterator first, InputIterator last,
                            OutputIterator  result,
                            const Traits& ch_traits,
                            forward_iterator_tag )
{ return CGAL_ch_akl_toussaint(first, last, result, ch_traits); }

template <class InputIterator, class OutputIterator, class Traits>
inline
OutputIterator
CGAL___convex_hull_points_2(InputIterator first, InputIterator last,
                            OutputIterator  result,
                            const Traits& ch_traits,
                            bidirectional_iterator_tag )
{ return CGAL_ch_akl_toussaint(first, last, result, ch_traits); }

template <class InputIterator, class OutputIterator, class Traits>
inline
OutputIterator
CGAL___convex_hull_points_2(InputIterator first, InputIterator last,
                            OutputIterator  result,
                            const Traits& ch_traits,
                            random_access_iterator_tag )
{ return CGAL_ch_akl_toussaint(first, last, result, ch_traits); }

template <class InputIterator, class OutputIterator, class Traits>
inline
OutputIterator
CGAL_convex_hull_points_2(InputIterator first, InputIterator last,
                          OutputIterator  result,
                          const Traits& ch_traits)
{
  return CGAL___convex_hull_points_2(first, last, result, ch_traits,
                                     iterator_category(first) );
}
```

⟨*extreme_points inline declaration*⟩≡

```
template <class ForwardIterator, class OutputIterator, class R>
inline
OutputIterator
CGAL__convex_hull_points_2(ForwardIterator first, ForwardIterator last,
                           OutputIterator  result,
                           CGAL_Point_2<R>* )
{
  return CGAL_convex_hull_points_2(first, last, result,
                                   CGAL_convex_hull_traits_2<R>() );
}

template <class ForwardIterator, class OutputIterator>
```

```
inline
OutputIterator
CGAL_convex_hull_points_2(ForwardIterator first, ForwardIterator last,
                          OutputIterator  result )
{
  return CGAL__convex_hull_points_2(first, last, result,
                                    value_type(first) );
}
```

Each of the convex hull algorithms is put in a separate file. Furthermore there is a file containing the selected extreme point computation and a file containing the checkers. Finally, there is a file for the default algorithm computing convex hulls in two dimensions. In most cases we have a `.h`-file containing classes, template function declarations and inline functions and a `.C`-file containing the template code. Besides the files containing convexity checker and the file defining the default algorithm for convex hull points, all files have prefix `ch_`:

```
ch_graham_andrew.h
ch_graham_andrew.C
ch_akl_toussaint.h
ch_akl_toussaint.C
ch_eddy.h
ch_eddy.C
ch_bykat.h
ch_bykat.C
ch_jarvis.h
ch_jarvis.C

ch_selected_extreme_points_2.h
ch_selected_extreme_points_2.C
convexity_check_2.h
convexity_check_2.C
convex_hull_2.h
```

Traits classes are provided in

`convex_hull_traits_2.h`	(default)
`convex_hull_cartesian_double_traits_2.h`	(specialisation for `Cartesian<double>`)
`convex_hull_constructive_traits_2.h`	(constructive version)
`convex_hull_rat_leda_traits_2.h`	(for `leda_rat_point`)
`convex_hull_leda_traits_2.h`	(for `leda_point`)

The file `gnu_istream_iterator_value_type_fix.h` contains a bug fix for `g++.2.7.2` and the STL coming with it. The file `ch_utils.h` contains some utilities. In particular, it includes `ch_assertions.h`, which provides assertion handling. Some test routines defined in Section 15 are in the files

```
_test_fct_ch_I_2.h
_test_fct_ch_I_2.C
ch__test.h
ch__test.C
```

All the files mentioned adove are loacted in the subdirectory `include/CGAL`. The example files

```
ch_example_from_cin_to_cout.C
ch_example_window.C
ch_example_window_constructive.C
ch_example_polygon.C
ch_example_leda_rat_point.C
ch_example_leda_point.C
ch_example_number_types1.C
ch_example_number_types2.C
ch_example_timings.C
```

are described in Sections 3, 4, 14, and 15.

All files in this package have a header containing author information and a small copyright notice. Note that `<CGAL/ch_assertions.h>` has been generated automatically by CGAL tools.

⟨*CGAL header*⟩≡
```
// ============================================================================
//
// Copyright (c) 1997 The CGAL Consortium
//
//
//
//
//
// ----------------------------------------------------------------------------
// release       :
// release_date  :
//
```

⟨*author notice*⟩≡
```
// revision      : 1.1
// revision_date : 29 Dec 1997
// author(s)     : Stefan Schirra <Stefan.Schirra@@mpi-sb.mpg.de>
//
// coordinator   : MPI, Saarbruecken
// ============================================================================
```

The `.h`-files contain the mechanism for the default versions. Since all the geometric objects and geometric primitives used in the functions with traits class parameter are provided by this parameter, only some files from STL and the assertion file are included. The mechanism for the default version uses the `CGAL_Point_2<R>` type. `<CGAL/Point_2.h>` is not included, because a user interested only in an own traits class should not be punished with the inclusion of numerous files containing template code from the CGAL-kernel. The mechanism for the the default version is encapsulated in a `#ifdef CGAL_POINT_2_H`, which requires that `<CGAL/Point_2.h>` has been included earlier.

`g++` has problems with `value_type()` of derived classes, especially `istream_iterator<T,Distance>`. Therefore a fix is included with the default traits.

⟨*default includes*⟩≡
```
#include <CGAL/ch_utils.h>
#ifdef CGAL_REP_CLASS_DEFINED
#include <CGAL/convex_hull_traits_2.h>
#if __GNUC__ <= 2 && __GNUC_MINOR__ <= 7
#ifndef CGAL_GNU_ISTREAM_ITERATOR_VALUE_TYPE_FIX_H
#include <CGAL/gnu_istream_iterator_value_type_fix.h>
#endif // CGAL_GNU_ISTREAM_ITERATOR_VALUE_TYPE_FIX_H
#endif // ...GNUC...
#endif // CGAL_REP_CLASS_DEFINED
⟨postcondition includes⟩
```

If postconditions are checked, the checking functions must be known.

⟨*postcondition includes*⟩≡
```
#ifndef CGAL_CH_NO_POSTCONDITIONS
⟨include checker⟩
#endif // CGAL_CH_NO_POSTCONDITIONS
```

⟨*include checker*⟩≡
```
#include <CGAL/convexity_check_2.h>
```

The file for the default traits class

⟨*convex_hull_traits_2.h*⟩≡
```
#ifndef CGAL_CONVEX_HULL_TRAITS_2_H
#define CGAL_CONVEX_HULL_TRAITS_2_H

#include <CGAL/Point_2.h>
#include <CGAL/predicates_on_points_2.h>
#include <CGAL/distance_predicates_2.h>
#include <CGAL/predicate_objects_on_points_2.h>

⟨convex hull traits⟩

#endif // CGAL_CONVEX_HULL_TRAITS_2_H
```

and a specialisation for `double`s.

⟨*convex_hull_cartesian_double_traits_2.h*⟩≡
```
#ifndef CGAL_CONVEX_HULL_TRAITS_CARTESIAN_DOUBLE_2_H
#define CGAL_CONVEX_HULL_TRAITS_CARTESIAN_DOUBLE_2_H

#include <CGAL/Point_2.h>
#include <CGAL/predicates_on_points_2.h>
#include <CGAL/distance_predicates_2.h>
#include <CGAL/predicate_objects_on_points_2.h>

⟨convex hull traits for CGAL_Cartesian double ⟩

#endif // CGAL_CONVEX_HULL_TRAITS_CARTESIAN_DOUBLE_2_H
```

The constructive traits

⟨*convex_hull_constructive_traits_2.h*⟩≡
```
#ifndef CGAL_CONSTRUCTIVE_CONVEX_HULL_TRAITS_2_H
#define CGAL_CONSTRUCTIVE_CONVEX_HULL_TRAITS_2_H

#include <CGAL/Point_2.h>
#include <CGAL/Line_2.h>
#include <CGAL/predicates_on_points_2.h>
#include <CGAL/distance_predicates_2.h>
#include <CGAL/predicate_objects_on_points_2.h>

⟨constructive convex hull traits⟩

#endif // CGAL_CONSTRUCTIVE_CONVEX_HULL_TRAITS_2_H
```

convex_hull_rat_leda_traits_2.h

⟨*convex_hull_rat_leda_traits_2.h*⟩≡
```
⟨CGAL header⟩
// file          : convex_hull_rat_leda_traits_2.h
// source        : convex_hull_2.lw
⟨author notice⟩

#ifndef CONVEX_HULL_RAT_LEDA_TRAITS_H
#define CONVEX_HULL_RAT_LEDA_TRAITS_H

#include <CGAL/ch_utils>
#include <CGAL/rat_leda_in_CGAL_2.h>
#include <CGAL/predicate_objects_on_points_2.h>

⟨leda convex hull traits⟩

#endif // CONVEX_HULL_RAT_LEDA_TRAITS_H
```

convex_hull_leda_traits_2.h

⟨*convex_hull_leda_traits_2.h*⟩≡

```
    ⟨CGAL header⟩
    // file          : convex_hull_leda_traits_2.h
    // source        : convex_hull_2.lw
    ⟨author notice⟩

    #ifndef CONVEX_HULL_LEDA_TRAITS_H
    #define CONVEX_HULL_LEDA_TRAITS_H

    #include <CGAL/ch_utils>
    #include <CGAL/leda_in_CGAL_2.h>
    #include <CGAL/predicate_objects_on_points_2.h>

    ⟨plain leda convex hull traits⟩

    #endif // CONVEX_HULL_LEDA_TRAITS_H
```

The files for selected extreme point computation:

⟨*ch_selected_extreme_points_2.h*⟩≡

```
    ⟨CGAL header⟩
    // file          : ch_selected_extreme_points_2.h
    // source        : convex_hull_2.lw
    ⟨author notice⟩

    #ifndef CGAL_CH_SELECTED_EXTREME_POINTS_2_H
    #define CGAL_CH_SELECTED_EXTREME_POINTS_2_H

    ⟨default includes⟩
    ⟨nswe extremepoints declaration with traits⟩
    #ifdef CGAL_POINT_2_H
    ⟨nswe extremepoints inline declaration⟩
    #endif // CGAL_POINT_2_H
    #ifdef CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION
    #include <CGAL/ch_selected_extreme_points_2.C>
    #endif // CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION

    #endif // CGAL_CH_SELECTED_EXTREME_POINTS_2_H
```

⟨*ch_selected_extreme_points_2.C*⟩≡

```
    ⟨CGAL header⟩
    // file          : ch_selected_extreme_points_2.C
    // source        : convex_hull_2.lw
    ⟨author notice⟩

    #ifndef CGAL_CH_SELECTED_EXTREME_POINTS_2_C
    #define CGAL_CH_SELECTED_EXTREME_POINTS_2_C

    #ifndef CGAL_CH_SELECTED_EXTREME_POINTS_2_H
    #include <CGAL/ch_selected_extreme_points_2.h>
    #endif // CGAL_CH_SELECTED_EXTREME_POINTS_2_H
    ⟨nswe extremepoints⟩

    #endif // CGAL_CH_SELECTED_EXTREME_POINTS_2_C
```

Graham Andrew stuff:

⟨*ch_graham_andrew.h*⟩≡

```
    ⟨CGAL header⟩
    // file          : ch_graham_andrew.h
    // source        : convex_hull_2.lw
    ⟨author notice⟩

    #ifndef CGAL_CH_GRAHAM_ANDREW_H
    #define CGAL_CH_GRAHAM_ANDREW_H

    ⟨default includes⟩
    #include <vector.h>
```

```
#include <algo.h>
⟨graham scan declaration with traits⟩
⟨graham scan with OutputIterator reference declaration with traits⟩
⟨graham hull declaration with traits⟩
#ifdef CGAL_POINT_2_H
⟨graham scan inline declaration⟩
⟨graham hull inline declaration⟩
#endif // CGAL_POINT_2_H

#ifdef CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION
#include <CGAL/ch_graham_andrew.C>
#endif // CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION

#endif // CGAL_CH_GRAHAM_ANDREW_H
```

⟨*ch_graham_andrew.C*⟩≡

```
⟨CGAL header⟩
// file          : ch_graham_andrew.C
// source        : convex_hull_2.lw
⟨author notice⟩

#ifndef CGAL_CH_GRAHAM_ANDREW_C
#define CGAL_CH_GRAHAM_ANDREW_C

#include <CGAL/stl_extensions.h>
#ifndef CGAL_CH_GRAHAM_ANDREW_H
#include <CGAL/ch_graham_andrew.h>
#endif // CGAL_CH_GRAHAM_ANDREW_H
⟨graham_andrew_scan⟩
⟨graham scan with OutputIterator reference⟩
⟨graham hull⟩

#endif // CGAL_CH_GRAHAM_ANDREW_C
```

Akl Toussaint stuff:

⟨*ch_akl_toussaint.h*⟩≡

```
⟨CGAL header⟩
// file          : ch_akl_toussaint.h
// source        : convex_hull_2.lw
⟨author notice⟩

#ifndef CGAL_CH_AKL_TOUSSAINT_H
#define CGAL_CH_AKL_TOUSSAINT_H

⟨default includes⟩
#include <CGAL/ch_selected_extreme_points_2.h>
#include <CGAL/ch_graham_andrew.h>
#include <CGAL/stl_extensions.h>
⟨Akl Toussaint alg declaration with traits⟩
#ifdef CGAL_POINT_2_H
⟨Akl Toussaint alg inline declaration⟩
#endif // CGAL_POINT_2_H

#ifdef CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION
#include <CGAL/ch_akl_toussaint.C>
#endif // CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION

#endif // CGAL_CH_AKL_TOUSSAINT_H
```

⟨*ch_akl_toussaint.C*⟩≡

```
⟨CGAL header⟩
// file          : ch_akl_toussaint.C
// source        : convex_hull_2.lw
⟨author notice⟩
```

```
#ifndef CGAL_CH_AKL_TOUSSAINT_C
#define CGAL_CH_AKL_TOUSSAINT_C

#ifndef CGAL_CH_AKL_TOUSSAINT_H
#include <CGAL/ch_akl_toussaint.h>
#endif // CGAL_CH_AKL_TOUSSAINT_H
⟨Akl Toussaint alg⟩

#endif // CGAL_CH_AKL_TOUSSAINT_C
```

Eddy's algorithm:

⟨*ch_eddy.h*⟩≡

```
⟨CGAL header⟩
// file          : ch_eddy.h
// source        : convex_hull_2.lw
⟨author notice⟩

#ifndef CGAL_CH_EDDY_H
#define CGAL_CH_EDDY_H

⟨default includes⟩
#include <CGAL/ch_selected_extreme_points_2.h>
#include <list.h>
#include <algo.h>
#include <CGAL/stl_extensions.h>
⟨Eddy's alg declaration with traits⟩
#ifdef CGAL_POINT_2_H
⟨Eddy's alg inline declaration⟩
#endif // CGAL_POINT_2_H

#ifdef CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION
#include <CGAL/ch_eddy.C>
#endif // CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION

#endif // CGAL_CH_EDDY_H
```

⟨*ch_eddy.C*⟩≡

```
⟨CGAL header⟩
// file          : ch_eddy.C
// source        : convex_hull_2.lw
⟨author notice⟩

#ifndef CGAL_CH_EDDY_C
#define CGAL_CH_EDDY_C

#ifndef CGAL_CH_EDDY_H
#include <CGAL/ch_eddy.h>
#endif // CGAL_CH_EDDY_H
⟨recursive eddy⟩
⟨Eddy's alg⟩

#endif // CGAL_CH_EDDY_C
```

Bykat algorithm:

⟨*ch_bykat.h*⟩≡

```
⟨CGAL header⟩
// file          : ch_bykat.h
// source        : convex_hull_2.lw
⟨author notice⟩

#ifndef CGAL_CH_BYKAT_H
#define CGAL_CH_BYKAT_H

⟨default includes⟩
```

```
#include <CGAL/ch_selected_extreme_points_2.h>
#include <list.h>
#include <algo.h>
#include <CGAL/stl_extensions.h>
#include <CGAL/ch_graham_andrew.h>
⟨Bykat alg declaration with traits⟩
#ifdef CGAL_POINT_2_H
⟨Bykat alg inline declaration⟩
#endif // CGAL_POINT_2_H

#ifdef CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION
#include <CGAL/ch_bykat.C>
#endif // CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION

#endif // CGAL_CH_BYKAT_H
```

⟨*ch_bykat.C*⟩≡

```
⟨CGAL header⟩
// file          : ch_bykat.C
// source        : convex_hull_2.lw
⟨author notice⟩

#ifndef CGAL_CH_BYKAT_C
#define CGAL_CH_BYKAT_C

#ifndef CGAL_CH_BYKAT_H
#include <CGAL/ch_eddy.h>
#endif // CGAL_CH_BYKAT_H
⟨nonrecursive Bykat-Eddy⟩
⟨bykat with threshold⟩

#endif // CGAL_CH_BYKAT_C
```

Jarvis' march:

⟨*ch_jarvis.h*⟩≡

```
⟨CGAL header⟩
// file          : ch_jarvis.h
// source        : convex_hull_2.lw
⟨author notice⟩

#ifndef CGAL_CH_JARVIS_H
#define CGAL_CH_JARVIS_H

⟨default includes⟩
#include <algo.h>
⟨jarvis march declaration with traits⟩
⟨jarvis alg declaration with traits⟩
#ifdef CGAL_POINT_2_H
⟨jarvis march inline declaration⟩
⟨jarvis alg inline declaration⟩
#endif // CGAL_POINT_2_H

#ifdef CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION
#include <CGAL/ch_jarvis.C>
#endif // CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION

#endif // CGAL_CH_JARVIS_H
```

⟨*ch_jarvis.C*⟩≡

```
⟨CGAL header⟩
// file          : ch_jarvis.C
// source        : convex_hull_2.lw
⟨author notice⟩
```

```
#ifndef CGAL_CH_JARVIS_C
#define CGAL_CH_JARVIS_C

#ifndef CGAL_CH_JARVIS_H
#include <CGAL/ch_jarvis.h>
#endif // CGAL_CH_JARVIS_H
⟨jarvis march⟩
⟨jarvis alg⟩

#endif // CGAL_CH_JARVIS_C
```

The convex hull algorithms header file. Since there are only inline function, no code file is included. `<iterator.h>` is included for iterator tags.

⟨*convex_hull_2.h*⟩≡

```
⟨CGAL header⟩
// file          : convex_hull_2.h
// source        : convex_hull_2.lw
⟨author notice⟩

#ifndef CGAL_CONVEX_HULL_2_H
#define CGAL_CONVEX_HULL_2_H

⟨default includes⟩
#include <CGAL/ch_akl_toussaint.h>
#include <CGAL/ch_bykat.h>
#include <iterator.h>
⟨extreme_points declaration with traits⟩
#ifdef CGAL_POINT_2_H
⟨extreme_points inline declaration⟩
#endif // CGAL_POINT_2_H

#endif // CGAL_CONVEX_HULL_2_H
```

The checker:

⟨*convexity_check_2.h*⟩≡

```
⟨CGAL header⟩
// file          : convexity_check_2.h
// source        : convex_hull_2.lw
⟨author notice⟩

#ifndef CGAL_CONVEXITY_CHECK_2_H
#define CGAL_CONVEXITY_CHECK_2_H

#include <CGAL/stl_extensions.h>
#include <algo.h>
⟨default includes⟩
⟨checker declaration with traits⟩
⟨brute force check declaration with traits⟩
⟨brute force chain check declaration with traits⟩
#ifdef CGAL_POINT_2_H
⟨checker inline declaration⟩
⟨brute force check inline declaration⟩
⟨brute force chain check inline declaration⟩
#endif // CGAL_POINT_2_H

#ifdef CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION
#include <CGAL/convexity_check_2.C>
#endif // CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION

#endif // CGAL_CONVEXITY_CHECK_2_H
```

⟨*convexity_check_2.C*⟩≡
```
⟨CGAL header⟩
// file          : convexity_check_2.C
// source        : convex_hull_2.lw
⟨author notice⟩

#ifndef CGAL_CONVEXITY_CHECK_2_C
#define CGAL_CONVEXITY_CHECK_2_C

#ifndef CGAL_CONVEXITY_CHECK_2_H
#include <CGAL/convexity_check_2.h>
#endif // CGAL_CONVEXITY_CHECK_2_H
⟨checker⟩
⟨brute force check⟩
⟨brute force chain check⟩

#endif // CGAL_CONVEXITY_CHECK_2_C
```

Some utilities

⟨*ch_utils.h*⟩≡
```
⟨CGAL header⟩
// file          : ch_utils.h
// source        : convex_hull_2.lw
⟨author notice⟩

#ifndef CGAL_CONFIG_H
#include <CGAL/config.h>
#endif // CGAL_CONFIG_H
#include <CGAL/ch_assertions.h>

⟨use argument⟩
```

And finally a bug fix for g++.

⟨*value type for istream_iterator*⟩≡
```
template <class T, class Distance>
inline
T*
value_type(const istream_iterator<T, Distance>&)
{ return (T*)(0); }
```

⟨*gnu_istream_iterator_value_type_fix.h*⟩≡
```
⟨CGAL header⟩
// file          : gnu_istream_iterator_value_type_fix.h
// source        : convex_hull_2.lw
⟨author notice⟩

#ifndef GNU_ISTREAM_ITERATOR_VALUE_TYPE_FIX_H
#define GNU_ISTREAM_ITERATOR_VALUE_TYPE_FIX_H

#if __GNUC__ <= 2 && __GNUC_MINOR__ <= 7
#include <iterator.h>
⟨value type for istream_iterator⟩
#endif // ...GNUG...

#endif // GNU_ISTREAM_ITERATOR_VALUE_TYPE_FIX_H
```

14 Advanced Examples

Before we present more example programs, we present some useful collections of `#include`s.

⟨*file stream includes*⟩≡
```
#include <fstream.h>
```

⟨*CGAL basic includes*⟩≡
```
#include <CGAL/basic.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Cartesian.h>
```

⟨*include convex hull algorithms*⟩≡
```
#include <CGAL/ch_akl_toussaint.h>
#include <CGAL/ch_graham_andrew.h>
#include <CGAL/ch_eddy.h>
#include <CGAL/ch_bykat.h>
#include <CGAL/ch_jarvis.h>
```

⟨*include default convex hull traits*⟩≡
```
#include <CGAL/convex_hull_traits_2.h>
```

⟨*include default convex hull traits for Cartesian double*⟩≡
```
#include <CGAL/convex_hull_cartesian_double_traits_2.h>
```

⟨*include constructive convex hull traits*⟩≡
```
#include <CGAL/convex_hull_constructive_traits_2.h>
```

⟨*LEDA number type includes*⟩≡
```
#include <CGAL/leda_integer.h>
#include <CGAL/leda_rational.h>
#include <CGAL/leda_real.h>
```

⟨*stl includes*⟩≡
```
#include <deque.h>
#include <list.h>
#include <vector.h>
```

The next chunk uses LEDA list in place of the lists from STL. Fortunately LEDA is now prefixed (use `-DLEDA_PREFIX`!).

⟨*stl includes with leda*⟩≡
```
#include <LEDA/list.h>
#include <deque.h>
#include <vector.h>
```

14.1 Timing

An interesting issue is run time comparison. Here are some useful chunks for measuring running time, with or without LEDA.

⟨*timing includes*⟩≡
```
#ifndef CGAL_USE_LEDA
#include <stdlib.h>
extern "C" long clock();
#endif // CGAL_USE_LEDA
```

⟨*timing variables: t and delta_t*⟩≡

```
#ifdef CGAL_USE_LEDA
float t, delta_t;
#else
long  t, delta_t;
#endif // CGAL_USE_LEDA
```

⟨*start timing*⟩≡

```
#ifdef CGAL_USE_LEDA
t = used_time();
#else
t = clock();
#endif // CGAL_USE_LEDA
```

⟨*stop timing*⟩≡

```
#ifdef CGAL_USE_LEDA
delta_t = used_time(t);
#else
delta_t = clock() - t;
#endif // CGAL_USE_LEDA
```

We provide a function to measure the running time of the implemented classical convex hull algorithms. It is parameterized by a traits class. Note that Jarvis' march has quadratic running time! Thus run it on examples with many extreme points, e.g. on cocircular points only if you have a lot of time! For example, here are running times for 10 000 random points in a disc and 10 000 random almost cocircular points. points:

```
CGAL_ch_akl_toussaint:         0.11
CGAL_ch_eddy:                  0.14
CGAL_ch_bykat                  0.10
CGAL_ch_bykat_with_threshold:  0.09
CGAL_ch_graham_andrew:         0.14
CGAL_ch_jarvis:                1.93

CGAL_ch_akl_toussaint:         0.16
CGAL_ch_eddy:                  0.45
CGAL_ch_bykat                  0.32
CGAL_ch_bykat_with_threshold:  0.31
CGAL_ch_graham_andrew:         0.15
CGAL_ch_jarvis:               47.89
```

In the `CGAL_ch_timing` function presented next, we assume that the value of the forward iterator used to output the convex hull points is not invalidated during the execution of the convex hull computation (by resizing the container).

⟨*CGAL_ch_timing*⟩≡

```
template <class ForwardIterator1, class ForwardIterator2, class Traits>
void
CGAL_ch_timing( ForwardIterator1 first, ForwardIterator1 last,
                ForwardIterator2 result,
                int iterations,
                const Traits& ch_traits)
{
  int i;
  ⟨timing variables: t and delta_t⟩
  cout << endl;
  ForwardIterator2  restart = result;

  ⟨start timing⟩
  for (i=0; i < iterations; i++)
```

```
  {
      result = restart;
      CGAL_ch_akl_toussaint( first, last , result, ch_traits);
  }
  ⟨stop timing⟩
  cout << "CGAL_ch_akl_toussaint:         " << delta_t << endl;
  ⟨start timing⟩
  for (i=0; i < iterations; i++)
  {
      result = restart;
      CGAL_ch_eddy( first, last , result, ch_traits);
  }
  ⟨stop timing⟩
  cout << "CGAL_ch_eddy:                  " << delta_t << endl;
  ⟨start timing⟩
  for (i=0; i < iterations; i++)
  {
      result = restart;
      CGAL_ch_bykat( first, last , result, ch_traits);
  }
  ⟨stop timing⟩
  cout << "CGAL_ch_bykat                  " << delta_t << endl;
  ⟨start timing⟩
  for (i=0; i < iterations; i++)
  {
      result = restart;
      CGAL_ch_bykat_with_threshold( first, last , result, ch_traits);
  }
  ⟨stop timing⟩
  cout << "CGAL_ch_bykat_with_threshold:  " << delta_t << endl;
  ⟨start timing⟩
  for (i=0; i < iterations; i++)
  {
      result = restart;
      CGAL_ch_graham_andrew( first, last , result, ch_traits);
  }
  ⟨stop timing⟩
  cout << "CGAL_ch_graham_andrew:         " << delta_t << endl;
  ⟨start timing⟩
  for (i=0; i < iterations; i++)
  {
      result = restart;
      CGAL_ch_jarvis( first, last , result, ch_traits);
  }
  ⟨stop timing⟩
  cout << "CGAL_ch_jarvis:                " << delta_t << endl;
}
```

⟨*CGAL_ch_timing declaration*⟩≡

```
template <class ForwardIterator, class OutputIterator, class Traits>
void
CGAL_ch_timing( ForwardIterator first, ForwardIterator last,
                OutputIterator result,
                int iterations,
                const Traits& ch_traits);
```

⟨*ch_timing_2.h*⟩≡

```
⟨CGAL header⟩
// file          : ch_timing_2.h
// source        : convex_hull_2.lw
⟨author notice⟩

#ifndef CGAL_CH_TIMING_2_H
#define CGAL_CH_TIMING_2_H

⟨CGAL_ch_timing declaration⟩

#ifdef CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION
#include <CGAL/ch_timing_2.C>
#endif // CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION

#endif // CGAL_CH_TIMING_2_H
```

⟨*ch_timing_2.C*⟩≡

```
⟨CGAL header⟩
// file          : ch_timing_2.C
// source        : convex_hull_2.lw
⟨author notice⟩

#ifndef CGAL_CH_TIMING_2_C
#define CGAL_CH_TIMING_2_C

#ifndef CGAL_CH_TIMING_2_H
#include <CGAL/ch_timing_2.h>
#endif // CGAL_CH_TIMING_2_H
⟨CGAL_ch_timing⟩

#endif // CGAL_CH_TIMING_2_C
```

In the following program, the input data is read from a file into a `vector`. We copy the input points to another vector which is then definitely large enough to hold the output.

⟨*ch_example_timing.C*⟩≡

```
⟨CGAL basic includes⟩
⟨LEDA number type includes⟩
⟨include default convex hull traits⟩
⟨file stream includes⟩
⟨stl includes⟩
⟨include convex hull algorithms⟩
⟨timing includes⟩
#include <CGAL/ch_timing_2.h>

typedef double                                          nu_type;
typedef CGAL_Cartesian< nu_type >                       RepCls;
typedef CGAL_convex_hull_traits_2<RepCls>               TraitsCls;
typedef TraitsCls::Point_2                              Point_2;

int
main( int argc, char* argv[] )
{
  if (argc != 3)   // assertion
  {
      cerr << "Usage: ch_example_timing datafilename ";
      cerr << "number_of_iterations";
      exit(1);
  }
  vector< Point_2 > V;
  vector< Point_2 > VE;
  ifstream F(argv[1]);
  istream_iterator< Point_2, ptrdiff_t>  in_start( F );
  istream_iterator< Point_2, ptrdiff_t>  in_end;
```

```
    copy( in_start, in_end , back_inserter(V) );
    copy( V.begin(), V.end(), back_inserter(VE) );
    int iterations = atoi( argv[2] );
    CGAL_ch_timing(V.begin(), V.end(), VE.begin(), iterations, TraitsCls() );
    return 0;
  }
```

14.2 Using traits from LEDA

We provide an example with `leda_rat_points`

⟨*ch_example_leda_rat_point.C*⟩≡

```
  #include <LEDA/rat_point.h>
  #include <CGAL/convex_hull_rat_leda_traits_2.h>
  #include <LEDA/plane_alg.h>
  #include <fstream.h>
  #include <CGAL/ch_bykat.h>
  #include <CGAL/ch_timing_2.h>
  typedef leda_rat_point  point;
  ⟨leda test main part 1⟩
    t = used_time();
    CGAL_ch_bykat_with_threshold( LL.begin(), LL.end(), back_inserter(LI),
                                  CGAL_convex_hull_rat_leda_traits_2() );
  ⟨leda test main part 2⟩
```

and an example with `leda_points`.

⟨*ch_example_leda_point.C*⟩≡

```
  #include <LEDA/point.h>
  #include <LEDA/plane_alg.h>
  #include <CGAL/convex_hull_leda_traits_2.h>
  #include <fstream.h>
  #include <CGAL/ch_bykat.h>
  #include <CGAL/ch_timing_2.h>
  typedef leda_point      point;
  ⟨leda test main part 1⟩
    t = used_time();
    CGAL_ch_bykat_with_threshold( LL.begin(), LL.end(), back_inserter(LI),
                                  CGAL_convex_hull_leda_traits_2() );
  ⟨leda test main part 2⟩
```

⟨*leda test main part 1*⟩≡

```
  int
  main( int argc, char* argv[] )
  {
    if (argc != 2)   // assertion
    {
        cerr << "Usage: ch_example_leda...points  datafilename ";
        exit(1);
    }

    float t, delta_t;
    leda_list< point > LL;
    list< point > LI;
    leda_list< point > LR;

    ifstream F(argv[1]);
    istream_iterator< point, ptrdiff_t>  in_start( F );
    istream_iterator< point, ptrdiff_t>  in_end;

    copy( in_start, in_end, back_inserter(LL) );
```

```
  t = used_time();
  LR = CONVEX_HULL( LL );
  delta_t = used_time(t);
  cout << "leda_CONVEX_HULL:          " << delta_t << endl;
```

⟨*leda test main part 2*⟩≡

```
  delta_t = used_time(t);
  cout << "CGAL_bykat_with_threshold: " << delta_t << endl;
  return 0;
}
```

14.3 Another Example with Output to a `leda_window`

Here are a few useful routines for visualization.
The first is a `click_to_continue(CGAL_Window_stream& )`.

⟨*click_to_continue*⟩≡

```
void
click_to_continue(CGAL_Window_stream& W)
{
  double x, y;
  W.read_mouse(x,y);
}
```

Conversion of a cyclic sequence of points into a cyclic sequence of segments is a useful tool for visualization. To use this, the traits class must provide a `Segment_2` type!

⟨*points to segments*⟩≡

```
#include <CGAL/stl_extensions.h>

template <class ForwardIterator, class OutputIterator, class Traits>
OutputIterator
CGAL_ch_from_points_to_segments(ForwardIterator first, ForwardIterator last,
                                OutputIterator result, const Traits& )
{
  typedef  typename Traits::Segment_2   Segment_2;

  if (first == last) return result;
  ForwardIterator it   = first;
  ForwardIterator fifi = CGAL_successor(first);
  while ( fifi != last )
  {
      result = Segment_2(*it,*fifi);
      it     = fifi++;
  }
  result = Segment_2(*it,*first);
  return result;
}
```

⟨*points to segments declaration with traits*⟩≡

```
template <class ForwardIterator, class OutputIterator, class Traits>
OutputIterator
CGAL_ch_from_points_to_segments(ForwardIterator first, ForwardIterator last,
                                OutputIterator result, const Traits& );
```

⟨*points to segments declaration*⟩≡

```
template <class ForwardIterator, class OutputIterator, class R>
inline
OutputIterator
CGAL_ch__from_points_to_segments(ForwardIterator first, ForwardIterator last,
                                 OutputIterator result, CGAL_Point_2<R>* )
{
  return CGAL_ch_from_points_to_segments(first, last, result,
                                         CGAL_convex_hull_traits_2<R>() );
}

template <class ForwardIterator, class OutputIterator>
inline
OutputIterator
CGAL_ch_from_points_to_segments(ForwardIterator first, ForwardIterator last,
                                OutputIterator result)
{
  return CGAL_ch__from_points_to_segments(first,last,result,value_type(first));
}
```

In the next example, `CGAL_convex_hull_constructive_traits<>` are used.

⟨*ch_example_window_constructive.C*⟩≡

```
⟨CGAL basic includes⟩
⟨file stream includes⟩
⟨stl includes⟩
#include <CGAL/Point_2.h>
#include <CGAL/Segment_2.h>
⟨include constructive convex hull traits⟩
#include <CGAL/IO/Window_stream.h>
#include <CGAL/IO/Ostream_iterator.h>
⟨click_to_continue⟩
#include <CGAL/convex_hull_2.h>

⟨points to segments declaration with traits⟩
⟨points to segments declaration⟩
⟨points to segments⟩

typedef   CGAL_Cartesian<double>   RepCls;
typedef   CGAL_convex_hull_constructive_traits_2<RepCls>  TraitsCls;
typedef   TraitsCls::Segment_2     Segment_2;
typedef   TraitsCls::Point_2       Point_2;

CGAL_Window_stream*  W_global_ptr;

int
main( int argc, char* argv[] )
{
  if (argc != 2)   // assertion
  {
      cerr << "Usage: ch_example_window_constructive  datafilename ";
      exit(1);
  }

  CGAL_Window_stream W(532,532);
  W.init(-100,1123,-100);
  W_global_ptr = &W;
  CGAL_Ostream_iterator<Point_2,CGAL_Window_stream>    winptout(W);
  CGAL_Ostream_iterator<Segment_2,CGAL_Window_stream>  winsegout(W);

  vector< Point_2 > V;
  ifstream F(argv[1]);
  istream_iterator< Point_2, ptrdiff_t>  in_start( F );
  istream_iterator< Point_2, ptrdiff_t>  in_end;
  copy( in_start, in_end , back_inserter(V) );
  vector< Point_2 >   VE;
```

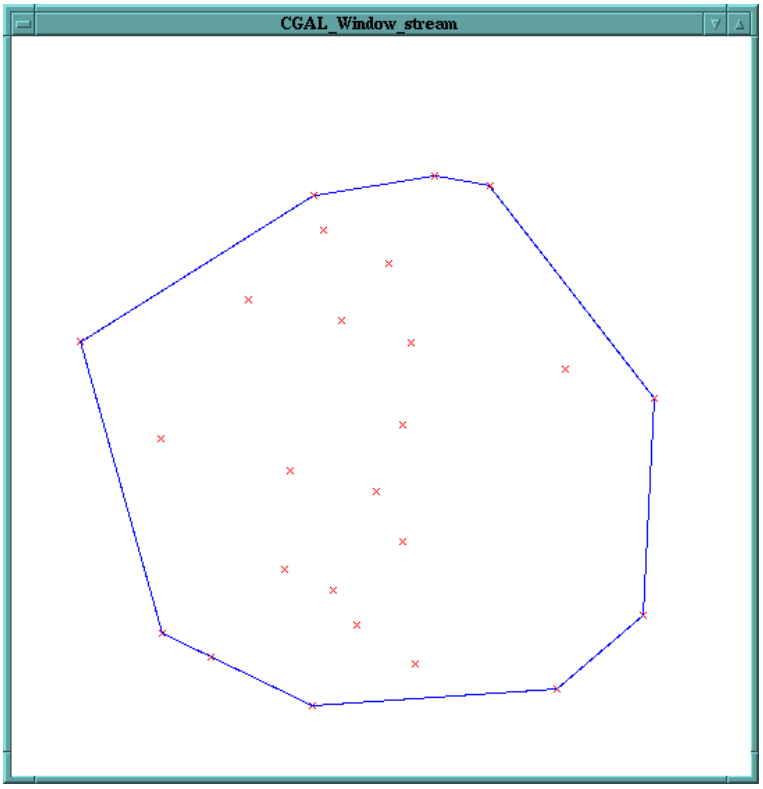

Figure 7: Convex hull polygon computed by `ch_example_window_constructive`.

```
    vector< Segment_2 > VS;

    W.clear();
    W << CGAL_RED;
    copy( V.begin(), V.end(), winptout );
    click_to_continue(W);
    CGAL_convex_hull_points_2( V.begin(), V.end() , back_inserter(VE),
                               TraitsCls() );
    CGAL_ch_from_points_to_segments( VE.begin(), VE.end(), back_inserter(VS),
                                     TraitsCls() );
    W << CGAL_BLUE;
    copy( VS.begin(), VS.end(), winsegout );
    click_to_continue(W);
    return 0;
  }
```

Figure 7 shows an example.

14.4 Cost of Arithmetic

In the last example, different number types are used and the influence of the cost of arithmetic on the running time is illustrated. Putting all the stuff into one file would cause problems for compilers: `virtual memory exhausted`. It seems that too much code has to be instantiated.

⟨*ch_example_number_types1.C*⟩≡

```
  ⟨CGAL basic includes⟩
  #include <CGAL/leda_integer.h>
  #include <CGAL/Gmpz.h>
  #include <CGAL/Point_2.h>
  ⟨include convex hull algorithms⟩
  #include <CGAL/convex_hull_traits_2.h>
```

```
#include <CGAL/convex_hull_rat_leda_traits_2.h>
⟨file stream includes⟩
#include <vector.h>
#include <CGAL/ch_timing_2.h>

typedef CGAL_Cartesian< leda_integer >                  IntegerRepCls;
typedef CGAL_Cartesian< CGAL_Gmpz >                     GmpzRepCls;

typedef CGAL_convex_hull_traits_2< IntegerRepCls >      IntegerTraits;
typedef CGAL_convex_hull_traits_2< GmpzRepCls >         GmpzTraits;
typedef CGAL_convex_hull_rat_leda_traits_2              ratLedaTraits;

typedef IntegerTraits::Point_2          integer_Point_2;
typedef GmpzTraits::Point_2             gmpz_Point_2;
typedef ratLedaTraits::Point_2          rat_Point_2;

int
main( int argc, char* argv[] )
{
  if (argc != 3)   // assertion
  {
      cerr << "Usage: ch_example_number_type1 ";
      cerr << "datafilename number_of_iterations";
      exit(1);
  }

  vector< integer_Point_2 > integer_V;
  vector< integer_Point_2 > integer_VE;
  ifstream integer_F(argv[1]);
  istream_iterator< integer_Point_2, ptrdiff_t> integer_eingabe_start(integer_F );
  istream_iterator< integer_Point_2, ptrdiff_t> integer_eingabe_ende;

  vector< rat_Point_2 > rat_V;
  vector< rat_Point_2 > rat_VE;

  vector< gmpz_Point_2 > gmpz_V;
  vector< gmpz_Point_2 > gmpz_VE;
  ifstream gmpz_F(argv[1]);
  istream_iterator< gmpz_Point_2, ptrdiff_t>  gmpz_eingabe_start( gmpz_F );
  istream_iterator< gmpz_Point_2, ptrdiff_t>  gmpz_eingabe_ende;

  int iterations = atoi( argv[2] );

  cout << endl << "gmpz: " ;
  copy( gmpz_eingabe_start, gmpz_eingabe_ende , back_inserter(gmpz_V) );
  copy( gmpz_V.begin(), gmpz_V.end(), back_inserter(gmpz_VE) );
  CGAL_ch_timing(gmpz_V.begin(), gmpz_V.end(), gmpz_VE.begin(), iterations,
               GmpzTraits() );
  gmpz_V.erase(gmpz_V.begin(), gmpz_V.end());
  gmpz_VE.erase(gmpz_VE.begin(), gmpz_VE.end());

  cout << endl << "integer: " ;
  copy( integer_eingabe_start, integer_eingabe_ende , back_inserter(integer_V) );
  copy( integer_V.begin(), integer_V.end(), back_inserter(integer_VE) );
  CGAL_ch_timing(integer_V.begin(), integer_V.end(), integer_VE.begin(), iterations,
               IntegerTraits() );
  integer_VE.erase(integer_VE.begin(), integer_VE.end());

  vector< integer_Point_2 >::iterator it = integer_V.begin();
  while ( it != integer_V.end() )
  {
      rat_V.push_back( rat_Point_2( (*it).x(), (*it).y() ));
      ++it;
  }
  integer_V.erase(integer_V.begin(), integer_V.end());

  cout << endl << "rat_leda: " ;
```

```
  copy( rat_V.begin(), rat_V.end(), back_inserter(rat_VE) );
  CGAL_ch_timing(rat_V.begin(), rat_V.end(), rat_VE.begin(), iterations,
                 ratLedaTraits() );
  rat_V.erase(rat_V.begin(), rat_V.end());
  rat_VE.erase(rat_VE.begin(), rat_VE.end());
}
```

⟨*ch_example_number_types2.C*⟩≡

```
⟨CGAL basic includes⟩
#include <CGAL/leda_real.h>
#include <CGAL/leda_bigfloat.h>
#include <CGAL/Point_2.h>
⟨include convex hull algorithms⟩
#include <CGAL/convex_hull_traits_2.h>
#include <CGAL/convex_hull_leda_traits_2.h>
⟨file stream includes⟩
#include <vector.h>
#include <CGAL/ch_timing_2.h>

typedef CGAL_Cartesian< leda_real >                   RealRepCls;
typedef CGAL_Cartesian< leda_bigfloat >               BigfloatRepCls;
typedef CGAL_Cartesian< double >                      DoubleRepCls;

typedef CGAL_convex_hull_traits_2< RealRepCls >       RealTraits;
typedef CGAL_convex_hull_traits_2< BigfloatRepCls >   BigfloatTraits;
typedef CGAL_convex_hull_traits_2< DoubleRepCls >     DoubleTraits;
typedef CGAL_convex_hull_leda_traits_2                LedaTraits;

typedef RealTraits::Point_2           real_Point_2;
typedef BigfloatTraits::Point_2       bigfloat_Point_2;
typedef DoubleTraits::Point_2         double_Point_2;
typedef LedaTraits::Point_2           leda_Point_2;

int
main( int argc, char* argv[] )
{
  if (argc != 4)   // assertion
  {
      cerr << "Usage: ch_example_number_type2 ";
      cerr << "datafilename number_of_iterations ";
      cerr << "precision_of_bigfloats";
      exit(1);
  }

  vector< real_Point_2 > real_V;
  vector< real_Point_2 > real_VE;
  ifstream real_F(argv[1]);
  istream_iterator< real_Point_2, ptrdiff_t>  real_eingabe_start( real_F );
  istream_iterator< real_Point_2, ptrdiff_t>  real_eingabe_ende;

  vector< bigfloat_Point_2 > bigfloat_V;
  vector< bigfloat_Point_2 > bigfloat_VE;
  ifstream bigfloat_F(argv[1]);
  istream_iterator< bigfloat_Point_2, ptrdiff_t>  bigfloat_eingabe_start( bigfloat_F );
  istream_iterator< bigfloat_Point_2, ptrdiff_t>  bigfloat_eingabe_ende;

  vector< double_Point_2 > double_V;
  vector< double_Point_2 > double_VE;
  ifstream double_F(argv[1]);
  istream_iterator< double_Point_2, ptrdiff_t> double_eingabe_start(double_F );
  istream_iterator< double_Point_2, ptrdiff_t> double_eingabe_ende;

  vector< leda_Point_2 > leda_V;
  vector< leda_Point_2 > leda_VE;
```

```
  int iterations = atoi( argv[2] );

  cout << endl << "real: " ;
  copy( real_eingabe_start, real_eingabe_ende , back_inserter(real_V) );
  copy( real_V.begin(), real_V.end(), back_inserter(real_VE) );
  CGAL_ch_timing(real_V.begin(), real_V.end(), real_VE.begin(), iterations,
                 RealTraits() );
  real_V.erase(real_V.begin(), real_V.end());
  real_VE.erase(real_VE.begin(), real_VE.end());

  int prec = atoi( argv[3] );
  if ( prec == 0 )
  {
      leda_bigfloat::set_rounding_mode(EXACT);
  }
  else
  {
      leda_bigfloat::set_precision( prec );
  }
  cout << endl << "bigfloat: " ;
  copy( bigfloat_eingabe_start, bigfloat_eingabe_ende , back_inserter(bigfloat_V) );
  copy( bigfloat_V.begin(), bigfloat_V.end(), back_inserter(bigfloat_VE) );
  CGAL_ch_timing(bigfloat_V.begin(), bigfloat_V.end(), bigfloat_VE.begin(), iterations,
                 BigfloatTraits() );
  bigfloat_V.erase(bigfloat_V.begin(), bigfloat_V.end());
  bigfloat_VE.erase(bigfloat_VE.begin(), bigfloat_VE.end());

  cout << endl << "double: " ;
  copy( double_eingabe_start, double_eingabe_ende , back_inserter(double_V) );
  copy( double_V.begin(), double_V.end(), back_inserter(double_VE) );
  CGAL_ch_timing(double_V.begin(), double_V.end(), double_VE.begin(), iterations,
                 DoubleTraits() );
  double_VE.erase(double_VE.begin(), double_VE.end());

  vector< double_Point_2 >::iterator it = double_V.begin();
  while ( it != double_V.end() )
  {
      leda_V.push_back( leda_Point_2( (*it).x(), (*it).y() ));
      ++it;
  }
  double_V.erase(double_V.begin(), double_V.end());

  cout << endl << "leda: " ;
  copy( leda_V.begin(), leda_V.end(), back_inserter(leda_VE) );
  CGAL_ch_timing(leda_V.begin(), leda_V.end(), leda_VE.begin(), iterations,
                 LedaTraits() );
  leda_V.erase(leda_V.begin(), leda_V.end());
  leda_VE.erase(leda_VE.begin(), leda_VE.end());
}
```

15 Test Functions

We start with a test function that gets a range of points, a traits class, and two flags. The flags determine, which algorithm(s) is used and which tests are made. First the flags (the tags should be self-explaining).

⟨*enums for ch testing*⟩≡

```
enum CGAL_ch_Algorithm{ CGAL_ch_JARVIS,
                        CGAL_ch_GRAHAM_ANDREW,
                        CGAL_ch_EDDY,
                        CGAL_ch_BYKAT,
                        CGAL_ch_BYKAT_WITH_THRESHOLD,
```

```
                         CGAL_ch_AKL_TOUSSAINT,
                         CGAL_ch_ALL,
                         CGAL_ch_DEFAULT };
enum CGAL_ch_Check_status{ CGAL_ch_CHECK_ALL,
                           CGAL_ch_CHECK_CONVEXITY,
                           CGAL_ch_CHECK_CONTAINEMENT,
                           CGAL_ch_NO_CHECK };
```

and then the test function. Since current compilers had problems with a previous piece of code, we explicitly define three different versions.

⟨*CGAL_ch_test*⟩≡

```
template <class InputIterator, class Traits>
bool
CGAL_ch__test(InputIterator first, InputIterator last, const Traits& ch_traits)
{
  CGAL_ch_Algorithm    alg         = CGAL_ch_ALL;
  CGAL_ch_Check_status check_level = CGAL_ch_CHECK_ALL;
  ⟨CGAL_ch_test body⟩
}
template <class InputIterator, class Traits>
bool
CGAL_ch__test(InputIterator first, InputIterator last, const Traits& ch_traits,
              CGAL_ch_Algorithm alg )
{
  CGAL_ch_Check_status check_level = CGAL_ch_CHECK_ALL;
  ⟨CGAL_ch_test body⟩
}
template <class InputIterator, class Traits>
bool
CGAL_ch__test(InputIterator first, InputIterator last, const Traits& ch_traits,
              CGAL_ch_Algorithm alg, CGAL_ch_Check_status check_level)
{
  ⟨CGAL_ch_test body⟩
}
```

⟨*CGAL_ch_test body*⟩≡

```
typedef typename Traits::Point_2   Point_2;
vector< Point_2 >     VI;
vector< Point_2 >     VO;
copy( first, last, back_inserter( VI ) );
typedef typename vector< Point_2 >::iterator  V_iter;
V_iter VIfirst = VI.begin();
V_iter VIlast  = VI.end();
switch (alg)
{
    case CGAL_ch_JARVIS:
            CGAL_ch_jarvis(VIfirst, VIlast, back_inserter(VO), ch_traits);
            break;
    case CGAL_ch_GRAHAM_ANDREW:
            CGAL_ch_graham_andrew(VIfirst, VIlast, back_inserter(VO),
                                  ch_traits);
            break;
    case CGAL_ch_EDDY:
            CGAL_ch_eddy(VIfirst, VIlast, back_inserter(VO),
                         ch_traits);
            break;
    case CGAL_ch_AKL_TOUSSAINT:
            CGAL_ch_akl_toussaint( VIfirst, VIlast, back_inserter(VO),
```

```
                                   ch_traits);
                break;
        case CGAL_ch_BYKAT:
                CGAL_ch_bykat(VIfirst, VIlast, back_inserter(VO),
                              ch_traits);
                break;
        case CGAL_ch_BYKAT_WITH_THRESHOLD:
                CGAL_ch_bykat_with_threshold(VIfirst, VIlast, back_inserter(VO),
                                             ch_traits);
                break;
        case CGAL_ch_ALL:
                return
                     CGAL_ch__test(VIfirst, VIlast, ch_traits,
                                   CGAL_ch_JARVIS, check_level)
                  && CGAL_ch__test(VIfirst, VIlast, ch_traits,
                                   CGAL_ch_GRAHAM_ANDREW, check_level)
                  && CGAL_ch__test(VIfirst, VIlast, ch_traits,
                                   CGAL_ch_EDDY, check_level)
                  && CGAL_ch__test(VIfirst, VIlast, ch_traits,
                                   CGAL_ch_BYKAT, check_level)
                  && CGAL_ch__test(VIfirst, VIlast, ch_traits,
                                   CGAL_ch_BYKAT_WITH_THRESHOLD, check_level)
                  && CGAL_ch__test(VIfirst, VIlast, ch_traits,
                                   CGAL_ch_AKL_TOUSSAINT, check_level);
        case CGAL_ch_DEFAULT:
        default:
                CGAL_convex_hull_points_2( VIfirst, VIlast, back_inserter(VO),
                                           ch_traits);
                break;
    }
    switch (check_level)
    {
        case CGAL_ch_CHECK_CONVEXITY:
                return CGAL_is_ccw_strongly_convex_2( VO.begin(), VO.end(),
                                                      ch_traits);
        case CGAL_ch_CHECK_CONTAINEMENT:
                return CGAL_ch_brute_force_check_2( VIfirst, VIlast,
                                                    VO.begin(), VO.end(),
                                                    ch_traits);
        case CGAL_ch_NO_CHECK:
                return true;
        case CGAL_ch_CHECK_ALL:
        default:
                return CGAL_is_ccw_strongly_convex_2( VO.begin(), VO.end(),
                                                      ch_traits)
                    && CGAL_ch_brute_force_check_2( VIfirst, VIlast,
                                                    VO.begin(), VO.end(),
                                                    ch_traits);
    }
```

And its declaration.

⟨*CGAL_ch__test declaration*⟩≡

```
template <class InputIterator, class Traits>
bool
CGAL_ch__test(InputIterator first, InputIterator last,
              const Traits& ch_traits,
              CGAL_ch_Algorithm alg,
              CGAL_ch_Check_status check_level);

template <class InputIterator, class Traits>
bool
```

```
CGAL_ch__test(InputIterator first, InputIterator last,
              const Traits& ch_traits,
              CGAL_ch_Algorithm alg);

template <class InputIterator, class Traits>
bool
CGAL_ch__test(InputIterator first, InputIterator last,
              const Traits& ch_traits);
```

The test function is put in a separate file.

⟨*ch__test.h*⟩≡

```
⟨CGAL header⟩
// file          : ch__test.h
// source        : convex_hull_2.lw
⟨author notice⟩

#ifndef CGAL_CH__TEST_H
#define CGAL_CH__TEST_H

#include <CGAL/convex_hull_2.h>
⟨include checker⟩
⟨include convex hull algorithms⟩

⟨enums for ch testing⟩
⟨CGAL_ch__test declaration⟩

#ifdef CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION
#include <CGAL/ch__test.C>
#endif // CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION

#endif // CGAL_CH__TEST_H
```

⟨*ch__test.C*⟩≡

```
⟨CGAL header⟩
// file          : ch__test.C
// source        : convex_hull_2.lw
⟨author notice⟩

#ifndef CGAL_CH__TEST_C
#define CGAL_CH__TEST_C

#ifndef CGAL_CH__TEST_H
#include <CGAL/ch__test.h>
#endif // CGAL_CH__TEST_H

⟨CGAL_ch__test⟩

#endif // CGAL_CH__TEST_C
```

The following batch test routine assumes that the point type from the traits class is constructable from three `ints`. In the kernel tests functions, there is an explicit conversion to `R::RT`. For the traits class, however, no number type trait was required, since it is used nowhere else.

⟨*test fct*⟩≡

```
template <class Traits>
bool
CGAL__test( const Traits& chI )
{
  typedef typename Traits::Point_2   Point_2;

  cout << "Testing CGAL_ch";
  vector< Point_2 >  Cocircular_points;
  Cocircular_points.push_back( Point_2( 39, 80, 89 ));
  Cocircular_points.push_back( Point_2( 180, 299, 349 ));
  Cocircular_points.push_back( Point_2( -3, -4, 5 ));
```

```
Cocircular_points.push_back( Point_2( -651, 260, 701 ));
Cocircular_points.push_back( Point_2( 180, -19, 181 ));
Cocircular_points.push_back( Point_2( -153, 104, 185 ));
Cocircular_points.push_back( Point_2( -247, -96, 265 ));
Cocircular_points.push_back( Point_2( -32, 255, 257 ));
Cocircular_points.push_back( Point_2( 45, -28, 53 ));
Cocircular_points.push_back( Point_2( -12, -35, 37 ));
assert( ! CGAL_ch_brute_force_check_2( \
           Cocircular_points.begin(), Cocircular_points.end(), \
           Cocircular_points.begin(), Cocircular_points.end(), chI ));
assert( CGAL_ch_brute_force_check_2( \
           Cocircular_points.begin(), Cocircular_points.begin(), \
           Cocircular_points.begin(), Cocircular_points.end(), chI ));
assert( CGAL_ch__test( Cocircular_points.begin(), \
                      Cocircular_points.end(), \
                      chI, CGAL_ch_ALL, CGAL_ch_CHECK_CONVEXITY ));
vector< Point_2 > extreme_points;
CGAL_convex_hull_points_2(Cocircular_points.begin(), Cocircular_points.end(),
                         back_inserter(extreme_points), chI );
assert( CGAL_is_ccw_strongly_convex_2( extreme_points.begin(), \
                                      extreme_points.begin(), \
                                      chI ));
assert( CGAL_is_cw_strongly_convex_2( extreme_points.rend(),
                                     extreme_points.rend(), \
                                     chI ));
assert( CGAL_is_ccw_strongly_convex_2( extreme_points.begin(), \
                             extreme_points.begin() + 1,  chI ));
assert( CGAL_is_cw_strongly_convex_2( extreme_points.begin(), \
                             extreme_points.begin() + 1,  chI ));
assert( CGAL_is_ccw_strongly_convex_2( extreme_points.begin(), \
                                      extreme_points.end(), \
                                      chI ));
assert( CGAL_is_cw_strongly_convex_2( extreme_points.rbegin(),
                                     extreme_points.rend(), \
                                     chI ));

cout << '.';
vector< Point_2 > Collinear_points;
Collinear_points.push_back( Point_2( 16, 20, 1 ));
Collinear_points.push_back( Point_2( 46, 40, 1 ));
Collinear_points.push_back( Point_2( 76, 60, 1 ));
Collinear_points.push_back( Point_2( 106, 80, 1 ));
Collinear_points.push_back( Point_2( -14, 0 , 1));
Collinear_points.push_back( Point_2( 136, 100, 1 ));
assert( CGAL_ch__test( Collinear_points.begin(), \
                      Collinear_points.end(), chI ));

cout << '.';
vector< Point_2 > Multiple_points;
Multiple_points.push_back( Point_2( 17, 80, 1 ));
Multiple_points.push_back( Point_2( 17, 80, 1 ));
Multiple_points.push_back( Point_2( 17, 80, 1 ));
Multiple_points.push_back( Point_2( 17, 80, 1 ));
assert( CGAL_ch_brute_force_check_2( \
            Multiple_points.begin(), Multiple_points.end(),\
            Multiple_points.begin(), Multiple_points.begin() + 1, chI ));
assert( CGAL_is_ccw_strongly_convex_2(Multiple_points.begin(), \
                                     Multiple_points.begin(), chI ));
assert( CGAL_ch__test( Multiple_points.begin(), \
                      Multiple_points.end(), chI ));
assert( CGAL_ch__test( Multiple_points.begin() + 2, \
                      Multiple_points.begin() + 3, chI ));
```

```
  cout << '.';
  vector< Point_2 > Iso_rectangle_points;
  Iso_rectangle_points.push_back( Point_2( 15, 0, 1 ));
  Iso_rectangle_points.push_back( Point_2( 45, 0, 1 ));
  Iso_rectangle_points.push_back( Point_2( 70, 0, 10 ));
  Iso_rectangle_points.push_back( Point_2( 12, 0, 1 ));
  Iso_rectangle_points.push_back( Point_2( 56, 118, 1 ));
  Iso_rectangle_points.push_back( Point_2( 27, 118, 1 ));
  Iso_rectangle_points.push_back( Point_2( 56, 118, 1 ));
  Iso_rectangle_points.push_back( Point_2( 112, 118, 1));
  Iso_rectangle_points.push_back( Point_2( 0, 9, 1));
  Iso_rectangle_points.push_back( Point_2( 0, 78, 1));
  Iso_rectangle_points.push_back( Point_2( 0, 16, 1));
  Iso_rectangle_points.push_back( Point_2( 0, 77, 1));
  Iso_rectangle_points.push_back( Point_2( 150, 56, 1));
  Iso_rectangle_points.push_back( Point_2( 150, 57, 1));
  Iso_rectangle_points.push_back( Point_2( 150, 58, 1));
  Iso_rectangle_points.push_back( Point_2( 150, 58, 1));
  assert( CGAL_ch__test( Iso_rectangle_points.begin(), \
                         Iso_rectangle_points.end(), chI ));
  assert( CGAL_ch__test( Iso_rectangle_points.begin(), \
                         Iso_rectangle_points.begin()+3, chI ));
  assert( CGAL_ch__test( Iso_rectangle_points.begin()+4, \
                         Iso_rectangle_points.begin()+7, chI ));
  assert( CGAL_ch__test( Iso_rectangle_points.begin()+5, \
                         Iso_rectangle_points.begin()+5, chI ));

  cout << "done" << endl;
  return true;
}
```

and its declaration

⟨*test fct declaration*⟩≡

```
template <class Traits>
bool
CGAL__test( const Traits& chI );
```

⟨*_test_fct_ch_I_2.h*⟩≡

⟨*CGAL header*⟩

```
// file          : _test_fct_ch_I_2.h
// source        : convex_hull_2.lw
```

⟨*author notice*⟩

```
#ifndef CGAL__TEST_FCT_CH_I_2_H
#define CGAL__TEST_FCT_CH_I_2_H

#include <assert.h>
#include <CGAL/ch__test.h>
```

⟨*test fct declaration*⟩

```
#ifdef CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION
#include <CGAL/_test_fct_ch_I_2.C>
#endif // CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION

#endif // CGAL__TEST_FCT_CH_I_2_H
```

⟨*_test_fct_ch_I_2.C*⟩≡

⟨*CGAL header*⟩

```
// file          : _test_fct_ch_I_2.C
// source        : convex_hull_2.lw
```

⟨*author notice*⟩

```
#ifndef CGAL__TEST_FCT_CH_I_2_C
#define CGAL__TEST_FCT_CH_I_2_C

#ifndef CGAL__TEST_FCT_CH_I_2_H
#include <CGAL/_test_fct_ch_I_2.h>
#endif // CGAL__TEST_FCT_CH_I_2_H
```
⟨*test fct*⟩
```
#endif // CGAL__TEST_FCT_CH_I_2_C
```

Here is a simple testfile, calling the above functions to test the computation of extreme points and checking the result. The test fails with `ints`. `CGAL_Cartesian<int>` is not suitable, because homogeneous coordinates are used to initialize points in `CGAL__test()` and the Carteesian coordinates are not integral. For `CGAL_Homogeneous<int>` it does not work either, too large `ints` arise during computations.

⟨*ch_test.C*⟩≡
  ⟨*CGAL basic includes*⟩
  ⟨*LEDA number type includes*⟩
  ⟨*include default convex hull traits for Cartesian double*⟩
  ⟨*include default convex hull traits*⟩
  ⟨*include constructive convex hull traits*⟩
  ⟨*file stream includes*⟩
  ⟨*stl includes*⟩
```
#include <CGAL/_test_fct_ch_I_2.h>

int
main()
{
  CGAL_convex_hull_traits_2< CGAL_Homogeneous<leda_integer> >    ch_H_integer;
  CGAL_convex_hull_traits_2<CGAL_Homogeneous<double> >           ch_H_double;
  CGAL_convex_hull_traits_2< CGAL_Cartesian<leda_rational> >     ch_C_rational;
  CGAL_convex_hull_traits_2< CGAL_Cartesian<double> >            ch_C_double;
  CGAL_convex_hull_constructive_traits_2<CGAL_Homogeneous<leda_integer> >
                                                                 cch_H_integer;
  CGAL_convex_hull_constructive_traits_2<CGAL_Homogeneous<double> >
                                                                 cch_H_double;

  cout << "Homogeneous<integer>:   ";
  CGAL__test( ch_H_integer );
  cout << "Cartesian<rational>:    ";
  CGAL__test( ch_C_rational );
  cout << "Homogeneous<double>:    ";
  CGAL__test( ch_H_double );
  cout << "Cartesian<double>:      ";
  CGAL__test( ch_C_double );
  cout << "Homogeneous<integer>: C ";
  CGAL__test( cch_H_integer );
  cout << "Homogeneous<double>:  C ";
  CGAL__test( cch_H_double );

  return 0;
}
```

15.1 CGAL Test

We break the tests into smaller units to decrease the work for compilers, especially the non-lazy ones (with the "lazy instantiation"-bug).

⟨*test files common part 1*⟩≡
⟨*CGAL basic includes*⟩
⟨*include default convex hull traits for Cartesian double*⟩
⟨*include default convex hull traits*⟩
⟨*include constructive convex hull traits*⟩
⟨*file stream includes*⟩
⟨*stl includes*⟩
```
#ifdef CGAL_USE_LEDA
#include <CGAL/leda_integer.h>
#include <CGAL/leda_rational.h>
#else
#include <CGAL/Gmpz.h>
#endif // CGAL_USE_LEDA
#include <CGAL/_test_fct_ch_I_2.h>

int
main()
```

⟨*test files common part 2*⟩≡
```
  return 0;
```

⟨*ch_test_SC.C*⟩≡
⟨*CGAL header*⟩
```
// file          : ch_test_SC.C
// source        : convex_hull_2.lw
```
⟨*author notice*⟩
⟨*test files common part 1*⟩
```
{
#ifdef CGAL_USE_LEDA
  CGAL_convex_hull_traits_2< CGAL_Cartesian<leda_rational> >     ch_C_rational;
  cout << "Cartesian<rational>:     ";
  CGAL__test( ch_C_rational );
#else
  CGAL_convex_hull_traits_2< CGAL_Cartesian<CGAL_Quotient<CGAL_Gmpz> > >
                                                                 ch_C_Qgmp;
  cout << "Cartesian<Quotient<Gmpz> > >:     ";
  CGAL__test( ch_C_Qgmp );
#endif // CGAL_USE_LEDA
  CGAL_convex_hull_traits_2< CGAL_Cartesian<double> >            ch_C_double;
  cout << "Cartesian<double>:     ";
  CGAL__test( ch_C_double );
```
⟨*test files common part 2*⟩
```
}
```

⟨*ch_test_SH.C*⟩≡
⟨*CGAL header*⟩
```
// file          : ch_test_SH.C
// source        : convex_hull_2.lw
```
⟨*author notice*⟩
⟨*test files common part 1*⟩
```
{
#ifdef CGAL_USE_LEDA
  CGAL_convex_hull_traits_2< CGAL_Homogeneous<leda_integer> >    ch_H_integer;
  cout << "Homogeneous<integer>:     ";
  CGAL__test( ch_H_integer );
#else
  CGAL_convex_hull_traits_2< CGAL_Homogeneous<CGAL_Gmpz> >       ch_H_gmp;
  cout << "Homogeneous<gmp>:     ";
  CGAL__test( ch_H_gmp );
#endif // CGAL_USE_LEDA
  CGAL_convex_hull_traits_2<CGAL_Homogeneous<double> >           ch_H_double;
```

```
    cout << "Homogeneous<double>:   ";
    CGAL__test( ch_H_double );
```
⟨test files common part 2⟩
```
}
```

⟨ch_test_CH.C⟩≡

⟨CGAL header⟩
```
// file          : ch_test_CH.C
// source        : convex_hull_2.lw
```
⟨author notice⟩

⟨test files common part 1⟩
```
{
#ifdef CGAL_USE_LEDA
  CGAL_convex_hull_constructive_traits_2< CGAL_Homogeneous<leda_integer> >
                                                               cch_H_integer;
  cout << "Homogeneous<integer>: C   ";
  CGAL__test( cch_H_integer );
#else
  CGAL_convex_hull_constructive_traits_2< CGAL_Homogeneous<CGAL_Gmpz> >
                                                               cch_H_gmp;
  cout << "Homogeneous<gmp>: C   ";
  CGAL__test( cch_H_gmp );
#endif // CGAL_USE_LEDA
  CGAL_convex_hull_constructive_traits_2<CGAL_Homogeneous<double> >
                                                               cch_H_double;
  cout << "Homogeneous<double>: C ";
  CGAL__test( cch_H_double );
```
⟨test files common part 2⟩
```
}
```

References

[1] S. G. Akl. Two remarks on a convex hull algorithm. *Inform. Process. Lett.*, 8(2):108–109, 1979.

[2] S. G. Akl. Corrigendum on convex hull algorithms. *Inform. Process. Lett.*, 10(3):168, 1980.

[3] S. G. Akl and G. T. Toussaint. Efficient convex hull algorithms for pattern recognition applications. In *Proc. 4th IEEE Internat. Conf. Pattern Recogn.*, pages 483–487, Kyoto, Japan, 1978.

[4] S. G. Akl and G. T. Toussaint. A fast convex hull algorithm. *Inform. Process. Lett.*, 7(5):219–222, 1978.

[5] D. C. S. Allison and M. T. Noga. Some performance tests of convex hull algorithms. *BIT*, 24:2–13, 1984.

[6] D. C. S. Allison and M. T. Noga. Computing the convex hull of a set of points. *Computer Physics Commun.*, 43(3):381–386, 1987.

[7] K. R. Anderson. A reevaluation of an efficient algorithm for determining the convex hull of a finite planar set. *Inform. Process. Lett.*, 7(1):53–55, 1978.

[8] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Inform. Process. Lett.*, 9(5):216–219, 1979.

[9] D. Avis. Comments on a lower bound for convex hull determination. *Inform. Process. Lett.*, 11:126, 1980.

[10] D. Avis. On the complexity of finding the convex hull of a set of points. *Discrete Appl. Math.*, 4:81–86, 1982.

[11] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The Quickhull algorithm for convex hulls. *ACM Trans. Math. Software*, 22(4):469–483, Dec. 1996.

[12] J. J. Barton and L. R. Nackman. *Scientific and Engineering C++*. Addison-Wesley, Reading, MA, 1997.

[13] L. J. Bass and S. R. Schubert. On finding the disc of minimum radius containing a given set of points. *Math. Comput.*, 12:712–714, 1967.

[14] B. K. Bhattacharya and G. T. Toussaint. Time- and storage-efficient implementation of an optimal convex hull algorithm. *Image Vision Comput.*, 1:140–144, 1983.

[15] A. Bykat. Convex hull of a finite set of points in two dimensions. *Inform. Process. Lett.*, 7:296–298, 1978.

[16] Working paper for draft proposed international standard for information systems – programming language c++. ANSI X3, Information Processing Systems, December 1996. `http://www.maths.warwick.ac.uk/c++/pub/`.

[17] CGAL project. *see* `http://www.cs.ruu.nl/CGAL/`.

[18] T. M. Y. Chan. Output-sensitive results on convex hulls, extreme points, and related problems. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 10–19, 1995.

[19] F. Dévai and T. Szendrényi. Comments on convex hull of a finite set of points in two dimensions. *Inform. Process. Lett.*, 9:141–142, 1979.

[20] L. P. Devroye. How to reduce the average complexity of convex hull finding algorithms. *Comput. Math. Appl.*, 7:299–308, 1981.

[21] W. F. Eddy. A new convex hull algorithm for planar sets. *ACM Trans. Math. Softw.*, 3:398–403 and 411–412, 1977.

[22] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. The cgal kernel: A basis for geometric computation. In M. C. Lin and D. Manocha, editors, *ACM Workshop on Applied Computational Geometry*, pages 191–202, Philadelphia, Pennsylvenia, May, 27–28 1996. Lecture Notes in Computer Science 1148.

[23] A. Fournier. Comments on convex hull of a finite set of points in two dimensions. *Inform. Process. Lett.*, 8:173, 1979.

[24] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inform. Process. Lett.*, 1:132–133, 1972.

[25] P. J. Green and B. W. Silverman. Constructing the convex hull of a set of points in the plane. *Comput. J.*, 22:262–266, 1979.

[26] D. Gries and I. Stojmenović. A note on Graham's convex hull algorithm. *Inform. Process. Lett.*, 25:323–327, 1987.

[27] C. C. Handley. Efficient planar convex hull algorithm. *Image Vision Comput.*, 3:29–35, 1985.

[28] B. Harris. A fast algorithm for finding the convex hull of a set of points in the plane. Internal Report, School Public Urban Policy, Univ. Pennsylvania, Philadelphia, PA, 1978.

[29] S. Hertel. An almost trivial convex hull algorithm for a presorted point set in the plane. Report A83/08, Fachber. Inform., Univ. Saarlandes, Saarbrücken, West Germany, 1983.

[30] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Inform. Process. Lett.*, 2:18–21, 1973.

[31] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? In *Proc. 20th Allerton Conf. Commun. Control Comput.*, pages 35–42, Monticello, Illinois, 1982.

[32] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM J. Comput.*, 15:287–299, 1986.

[33] J. Koplowitz and D. Jouppi. A more efficient convex hull algorithm. *Inform. Process. Lett.*, 7:56–57, 1978.

[34] K. Kreft and A. Langer. Iterators in the standard C++ library. *C++ Report*, 8(10):27–32, Nov.-Dec. 1996.

[35] M. M. McQueen and G. T. Toussaint. On the ultimate convex hull algorithm in practice. *Pattern Recogn. Lett.*, 3:29–34, 1985.

[36] K. Mehlhorn. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry.* Springer Verlag, 1984.

[37] K. Mehlhorn, S. Näher, T. Schilz, S. Schirra, R. Seidel, M. Seel, and Uhrig. C. Checking geometric programs or verification of geometric structures. In *Proc. 12th Annual ACM Symp. on Computational Geometry*, pages 159–165, 1996.

[38] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison-Wesley, 1996.

[39] M. Orlowski. On the conditions for success of Sklansky's convex hull algorithm. *Pattern Recogn.*, 16:579–586, 1983.

[40] M. H. Overmars and J. van Leeuwen. Further comments on Bykat's convex hull algorithm. *Inform. Process. Lett.*, 10:209–212, 1980.

[41] F. Preparata and M.I. Shamos. *Computational Geometry.* Springer Verlag, 1985.

[42] Silicon Graphics Computer Systems, Inc. Standard template library programmer's guide. `http://www.sgi.com/Technology/STL/`, 1997.

[43] J. Sklansky. Measuring concavity on rectangular mosaic. *IEEE Trans. Comput.*, C-21:1355–1364, 1972.

[44] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 3rd edition, 1997.

[45] G. F. Sullivan, D. S. Wilson, and G. M. Masson. Certification of computational results. *IEEE Trans. Comput.*, 44(7):833–847, 1995.

[46] G. T. Toussaint. A historical note on convex hull finding algorithms. *Pattern Recogn. Lett.*, 3:21–28, 1985.

[47] P. van Emde Boas. On the $\Omega(n \log n)$ lower-bound for convex hull and maximal vector determination. *Inform. Process. Lett.*, 10:132–136, 1980.

A Assertions

The following code was originally generated by CGAL tools written by Geert-Jan Giezeman and Sven Schönherr.

⟨*ch_assertions.h*⟩≡

```
⟨CGAL header⟩
// file         : ch_assertions.h
// source       : convex_hull_2.lw
⟨no author notice⟩

#ifndef CGAL_CH_ASSERTIONS_H
#define CGAL_CH_ASSERTIONS_H

#ifndef CGAL_ASSERTIONS_H
#  include <CGAL/assertions.h>
#endif

// macro definitions
// =================
// assertions
// ----------

#if defined(CGAL_CH_NO_ASSERTIONS) || defined(CGAL_NO_ASSERTIONS) \
  || defined(NDEBUG)
#  define CGAL_ch_assertion(EX) ((void)0)
#  define CGAL_ch_assertion_msg(EX,MSG) ((void)0)
#  define CGAL_ch_assertion_code(CODE)
#else
#  define CGAL_ch_assertion(EX) \
   ((EX)?((void)0):CGAL_assertion_fail( # EX , __FILE__, __LINE__, 0))
#  define CGAL_ch_assertion_msg(EX,MSG) \
   ((EX)?((void)0):CGAL_assertion_fail( # EX , __FILE__, __LINE__, MSG))
#  define CGAL_ch_assertion_code(CODE) CODE
#endif // CGAL_CH_NO_ASSERTIONS

#if defined(CGAL_CH_NO_ASSERTIONS) || defined(CGAL_NO_ASSERTIONS) \
  || (!defined(CGAL_CH_CHECK_EXACTNESS) && !defined(CGAL_CHECK_EXACTNESS))\
  || defined(NDEBUG)
#  define CGAL_ch_exactness_assertion(EX) ((void)0)
#  define CGAL_ch_exactness_assertion_msg(EX,MSG) ((void)0)
#  define CGAL_ch_exactness_assertion_code(CODE)
#else
#  define CGAL_ch_exactness_assertion(EX) \
   ((EX)?((void)0):CGAL_assertion_fail( # EX , __FILE__, __LINE__, 0))
#  define CGAL_ch_exactness_assertion_msg(EX,MSG) \
   ((EX)?((void)0):CGAL_assertion_fail( # EX , __FILE__, __LINE__, MSG))
#  define CGAL_ch_exactness_assertion_code(CODE) CODE
#endif // CGAL_CH_NO_ASSERTIONS

#if defined(CGAL_CH_NO_ASSERTIONS) \
  || defined(CGAL_NO_ASSERTIONS) \
  || (!defined(CGAL_CH_CHECK_EXPENSIVE) && !defined(CGAL_CHECK_EXPENSIVE)) \
  || defined(NDEBUG)
#  define CGAL_ch_expensive_assertion(EX) ((void)0)
#  define CGAL_ch_expensive_assertion_msg(EX,MSG) ((void)0)
#  define CGAL_ch_expensive_assertion_code(CODE)
#else
#  define CGAL_ch_expensive_assertion(EX) \
   ((EX)?((void)0):CGAL_assertion_fail( # EX , __FILE__, __LINE__, 0))
#  define CGAL_ch_expensive_assertion_msg(EX,MSG) \
   ((EX)?((void)0):CGAL_assertion_fail( # EX , __FILE__, __LINE__, MSG))
#  define CGAL_ch_expensive_assertion_code(CODE) CODE
#endif // CGAL_CH_NO_ASSERTIONS
```

```
#if defined(CGAL_CH_NO_ASSERTIONS) || defined(CGAL_NO_ASSERTIONS) \
  || (!defined(CGAL_CH_CHECK_EXACTNESS) && !defined(CGAL_CHECK_EXACTNESS))\
  || (!defined(CGAL_CH_CHECK_EXPENSIVE) && !defined(CGAL_CHECK_EXPENSIVE)) \
  || defined(NDEBUG)
#  define CGAL_ch_expensive_exactness_assertion(EX) ((void)0)
#  define CGAL_ch_expensive_exactness_assertion_msg(EX,MSG) ((void)0)
#  define CGAL_ch_expensive_exactness_assertion_code(CODE)
#else
#  define CGAL_ch_expensive_exactness_assertion(EX) \
   ((EX)?((void)0):CGAL_assertion_fail( # EX , __FILE__, __LINE__, 0))
#  define CGAL_ch_expensive_exactness_assertion_msg(EX,MSG) \
   ((EX)?((void)0):CGAL_assertion_fail( # EX , __FILE__, __LINE__, MSG))
#  define CGAL_ch_expensive_exactness_assertion_code(CODE) CODE
#endif // CGAL_CH_NO_ASSERTIONS

// preconditions
// -------------

#if defined(CGAL_CH_NO_PRECONDITIONS) || defined(CGAL_NO_PRECONDITIONS) \
  || defined(NDEBUG)
#  define CGAL_ch_precondition(EX) ((void)0)
#  define CGAL_ch_precondition_msg(EX,MSG) ((void)0)
#  define CGAL_ch_precondition_code(CODE)
#else
#  define CGAL_ch_precondition(EX) \
   ((EX)?((void)0):CGAL_precondition_fail( # EX , __FILE__, __LINE__, 0))
#  define CGAL_ch_precondition_msg(EX,MSG) \
   ((EX)?((void)0):CGAL_precondition_fail( # EX , __FILE__, __LINE__, MSG))
#  define CGAL_ch_precondition_code(CODE) CODE
#endif // CGAL_CH_NO_PRECONDITIONS

#if defined(CGAL_CH_NO_PRECONDITIONS) || defined(CGAL_NO_PRECONDITIONS) \
  || (!defined(CGAL_CH_CHECK_EXACTNESS) && !defined(CGAL_CHECK_EXACTNESS))\
  || defined(NDEBUG)
#  define CGAL_ch_exactness_precondition(EX) ((void)0)
#  define CGAL_ch_exactness_precondition_msg(EX,MSG) ((void)0)
#  define CGAL_ch_exactness_precondition_code(CODE)
#else
#  define CGAL_ch_exactness_precondition(EX) \
   ((EX)?((void)0):CGAL_precondition_fail( # EX , __FILE__, __LINE__, 0))
#  define CGAL_ch_exactness_precondition_msg(EX,MSG) \
   ((EX)?((void)0):CGAL_precondition_fail( # EX , __FILE__, __LINE__, MSG))
#  define CGAL_ch_exactness_precondition_code(CODE) CODE
#endif // CGAL_CH_NO_PRECONDITIONS

#if defined(CGAL_CH_NO_PRECONDITIONS) || defined(CGAL_NO_PRECONDITIONS) \
  || (!defined(CGAL_CH_CHECK_EXPENSIVE) && !defined(CGAL_CHECK_EXPENSIVE)) \
  || defined(NDEBUG)
#  define CGAL_ch_expensive_precondition(EX) ((void)0)
#  define CGAL_ch_expensive_precondition_msg(EX,MSG) ((void)0)
#  define CGAL_ch_expensive_precondition_code(CODE)
#else
#  define CGAL_ch_expensive_precondition(EX) \
   ((EX)?((void)0):CGAL_precondition_fail( # EX , __FILE__, __LINE__, 0))
#  define CGAL_ch_expensive_precondition_msg(EX,MSG) \
   ((EX)?((void)0):CGAL_precondition_fail( # EX , __FILE__, __LINE__, MSG))
#  define CGAL_ch_expensive_precondition_code(CODE) CODE
#endif // CGAL_CH_NO_PRECONDITIONS

#if defined(CGAL_CH_NO_PRECONDITIONS) || defined(CGAL_NO_PRECONDITIONS) \
  || (!defined(CGAL_CH_CHECK_EXACTNESS) && !defined(CGAL_CHECK_EXACTNESS))\
  || (!defined(CGAL_CH_CHECK_EXPENSIVE) && !defined(CGAL_CHECK_EXPENSIVE)) \
  || defined(NDEBUG)
#  define CGAL_ch_expensive_exactness_precondition(EX) ((void)0)
```

```
#  define CGAL_ch_expensive_exactness_precondition_msg(EX,MSG) ((void)0)
#  define CGAL_ch_expensive_exactness_precondition_code(CODE)
#else
#  define CGAL_ch_expensive_exactness_precondition(EX) \
   ((EX)?((void)0):CGAL_precondition_fail( # EX , __FILE__, __LINE__, 0))
#  define CGAL_ch_expensive_exactness_precondition_msg(EX,MSG) \
   ((EX)?((void)0):CGAL_precondition_fail( # EX , __FILE__, __LINE__, MSG))
#  define CGAL_ch_expensive_exactness_precondition_code(CODE) CODE
#endif // CGAL_CH_NO_PRECONDITIONS

// postconditions
// --------------

#if defined(CGAL_CH_NO_POSTCONDITIONS) || defined(CGAL_NO_POSTCONDITIONS) \
  || defined(NDEBUG)
#  define CGAL_ch_postcondition(EX) ((void)0)
#  define CGAL_ch_postcondition_msg(EX,MSG) ((void)0)
#  define CGAL_ch_postcondition_code(CODE)
#else
#  define CGAL_ch_postcondition(EX) \
   ((EX)?((void)0):CGAL_postcondition_fail( # EX , __FILE__, __LINE__, 0))
#  define CGAL_ch_postcondition_msg(EX,MSG) \
   ((EX)?((void)0):CGAL_postcondition_fail( # EX , __FILE__, __LINE__, MSG))
#  define CGAL_ch_postcondition_code(CODE) CODE
#endif // CGAL_CH_NO_POSTCONDITIONS

#if defined(CGAL_CH_NO_POSTCONDITIONS) || defined(CGAL_NO_POSTCONDITIONS) \
  || (!defined(CGAL_CH_CHECK_EXACTNESS) && !defined(CGAL_CHECK_EXACTNESS))\
  || defined(NDEBUG)
#  define CGAL_ch_exactness_postcondition(EX) ((void)0)
#  define CGAL_ch_exactness_postcondition_msg(EX,MSG) ((void)0)
#  define CGAL_ch_exactness_postcondition_code(CODE)
#else
#  define CGAL_ch_exactness_postcondition(EX) \
   ((EX)?((void)0):CGAL_postcondition_fail( # EX , __FILE__, __LINE__, 0))
#  define CGAL_ch_exactness_postcondition_msg(EX,MSG) \
   ((EX)?((void)0):CGAL_postcondition_fail( # EX , __FILE__, __LINE__, MSG))
#  define CGAL_ch_exactness_postcondition_code(CODE) CODE
#endif // CGAL_CH_NO_POSTCONDITIONS

#if defined(CGAL_CH_NO_POSTCONDITIONS) || defined(CGAL_NO_POSTCONDITIONS) \
  || (!defined(CGAL_CH_CHECK_EXPENSIVE) && !defined(CGAL_CHECK_EXPENSIVE)) \
  || defined(NDEBUG)
#  define CGAL_ch_expensive_postcondition(EX) ((void)0)
#  define CGAL_ch_expensive_postcondition_msg(EX,MSG) ((void)0)
#  define CGAL_ch_expensive_postcondition_code(CODE)
#else
#  define CGAL_ch_expensive_postcondition(EX) \
   ((EX)?((void)0):CGAL_postcondition_fail( # EX , __FILE__, __LINE__, 0))
#  define CGAL_ch_expensive_postcondition_msg(EX,MSG) \
   ((EX)?((void)0):CGAL_postcondition_fail( # EX , __FILE__, __LINE__, MSG))
#  define CGAL_ch_expensive_postcondition_code(CODE) CODE
#endif // CGAL_CH_NO_POSTCONDITIONS

#if defined(CGAL_CH_NO_POSTCONDITIONS) || defined(CGAL_NO_POSTCONDITIONS) \
  || (!defined(CGAL_CH_CHECK_EXACTNESS) && !defined(CGAL_CHECK_EXACTNESS))\
  || (!defined(CGAL_CH_CHECK_EXPENSIVE) && !defined(CGAL_CHECK_EXPENSIVE)) \
  || defined(NDEBUG)
#  define CGAL_ch_expensive_exactness_postcondition(EX) ((void)0)
#  define CGAL_ch_expensive_exactness_postcondition_msg(EX,MSG) ((void)0)
#  define CGAL_ch_expensive_exactness_postcondition_code(CODE)
#else
#  define CGAL_ch_expensive_exactness_postcondition(EX) \
   ((EX)?((void)0):CGAL_postcondition_fail( # EX , __FILE__, __LINE__, 0))
```

```
#  define CGAL_ch_expensive_exactness_postcondition_msg(EX,MSG) \
   ((EX)?((void)0):CGAL_postcondition_fail( # EX , __FILE__, __LINE__, MSG))
#  define CGAL_ch_expensive_exactness_postcondition_code(CODE) CODE
#endif // CGAL_CH_NO_POSTCONDITIONS

// warnings
// --------

#if defined(CGAL_CH_NO_WARNINGS) || defined(CGAL_NO_WARNINGS) \
  || defined(NDEBUG)
#  define CGAL_ch_warning(EX) ((void)0)
#  define CGAL_ch_warning_msg(EX,MSG) ((void)0)
#  define CGAL_ch_warning_code(CODE)
#else
#  define CGAL_ch_warning(EX) \
   ((EX)?((void)0):CGAL_warning_fail( # EX , __FILE__, __LINE__, 0))
#  define CGAL_ch_warning_msg(EX,MSG) \
   ((EX)?((void)0):CGAL_warning_fail( # EX , __FILE__, __LINE__, MSG))
#  define CGAL_ch_warning_code(CODE) CODE
#endif // CGAL_CH_NO_WARNINGS

#if defined(CGAL_CH_NO_WARNINGS) || defined(CGAL_NO_WARNINGS) \
  || (!defined(CGAL_CH_CHECK_EXACTNESS) && !defined(CGAL_CHECK_EXACTNESS))\
  || defined(NDEBUG)
#  define CGAL_ch_exactness_warning(EX) ((void)0)
#  define CGAL_ch_exactness_warning_msg(EX,MSG) ((void)0)
#  define CGAL_ch_exactness_warning_code(CODE)
#else
#  define CGAL_ch_exactness_warning(EX) \
   ((EX)?((void)0):CGAL_warning_fail( # EX , __FILE__, __LINE__, 0))
#  define CGAL_ch_exactness_warning_msg(EX,MSG) \
   ((EX)?((void)0):CGAL_warning_fail( # EX , __FILE__, __LINE__, MSG))
#  define CGAL_ch_exactness_warning_code(CODE) CODE
#endif // CGAL_CH_NO_WARNINGS

#if defined(CGAL_CH_NO_WARNINGS) || defined(CGAL_NO_WARNINGS) \
  || (!defined(CGAL_CH_CHECK_EXPENSIVE) && !defined(CGAL_CHECK_EXPENSIVE)) \
  || defined(NDEBUG)
#  define CGAL_ch_expensive_warning(EX) ((void)0)
#  define CGAL_ch_expensive_warning_msg(EX,MSG) ((void)0)
#  define CGAL_ch_expensive_warning_code(CODE)
#else
#  define CGAL_ch_expensive_warning(EX) \
   ((EX)?((void)0):CGAL_warning_fail( # EX , __FILE__, __LINE__, 0))
#  define CGAL_ch_expensive_warning_msg(EX,MSG) \
   ((EX)?((void)0):CGAL_warning_fail( # EX , __FILE__, __LINE__, MSG))
#  define CGAL_ch_expensive_warning_code(CODE) CODE
#endif // CGAL_CH_NO_WARNINGS

#if defined(CGAL_CH_NO_WARNINGS) || defined(CGAL_NO_WARNINGS) \
  || (!defined(CGAL_CH_CHECK_EXACTNESS) && !defined(CGAL_CHECK_EXACTNESS))\
  || (!defined(CGAL_CH_CHECK_EXPENSIVE) && !defined(CGAL_CHECK_EXPENSIVE)) \
  || defined(NDEBUG)
#  define CGAL_ch_expensive_exactness_warning(EX) ((void)0)
#  define CGAL_ch_expensive_exactness_warning_msg(EX,MSG) ((void)0)
#  define CGAL_ch_expensive_exactness_warning_code(CODE)
#else
#  define CGAL_ch_expensive_exactness_warning(EX) \
   ((EX)?((void)0):CGAL_warning_fail( # EX , __FILE__, __LINE__, 0))
#  define CGAL_ch_expensive_exactness_warning_msg(EX,MSG) \
   ((EX)?((void)0):CGAL_warning_fail( # EX , __FILE__, __LINE__, MSG))
#  define CGAL_ch_expensive_exactness_warning_code(CODE) CODE
#endif // CGAL_CH_NO_WARNINGS

#endif
```

and an empty 'author notice' for this file.

⟨*no author notice*⟩≡
```
// revision      : 1.1
// revision_date : 29 Dec 1997
// author(s)     :
//
// coordinator   : MPI, Saarbruecken (<Stefan.Schirra@@mpi-sb.mpg.de>)
// ============================================================================
```

B Tee Iterator

A nasty detail with the iterator concept in generic programming is that you cannot really check the computed output, since you don't have access to the data reported through an output iterator. The `CGAL_Tee_for_output_iterator` described next forwards data trhrough the iterator used to construct it while copying data to a container accessible from the `CGAL_Tee_for_output_iterator`. All copies of a `CGAL_Tee_for_output_iterator` (note: only copies, i.e. not all `CGAL_Tee_for_output_iterators`) share the same container in which they copy the data. This container behaves like an output stream: new data are added at the end (back) of the container and all copies of the same iterator, i.e. all identical copies can add data there. Thus `CGAL_Tee_for_output_iterators` behave like ostream iterators with respect to copying data. Because of this behavior, `CGAL_Tee_for_output_iterator` are tagged as output iterators. Nevertheless they provide a value type.

⟨*tee for output iterator*⟩≡
```
template <class T> class CGAL__Tee_for_output_iterator_rep;

template <class OutputIterator, class T>
class CGAL_Tee_for_output_iterator
  : public CGAL_Handle, public output_iterator
{
  typedef typename vector<T>::iterator  iterator;
  typedef T                             value_type;

public:
  CGAL_Tee_for_output_iterator(const OutputIterator& o) : o_it(o)
  {  PTR = (CGAL_Rep*) new CGAL__Tee_for_output_iterator_rep<T>(); }

  CGAL_Tee_for_output_iterator<OutputIterator,T>&
  operator=(const T& value)
  {
    ptr()->output_so_far.push_back(value);
    *o_it = value;
    return *this;
  }

  CGAL_Tee_for_output_iterator<OutputIterator,T>&
  operator*()
  { return *this; }

  CGAL_Tee_for_output_iterator<OutputIterator,T>&
  operator++()
  {
    ++o_it;
    return *this;
  }

  CGAL_Tee_for_output_iterator<OutputIterator,T>
  operator++(int)
  {
    CGAL_Tee_for_output_iterator<OutputIterator,T> tmp = *this;
    o_it++;
    return tmp;
```

```
    }

    iterator
    output_so_far_begin()
    { return ptr()->output_so_far.begin(); }

    iterator
    output_so_far_end()
    { return ptr()->output_so_far.end(); }

    OutputIterator&
    to_output_iterator()
    { return o_it; }

    CGAL__Tee_for_output_iterator_rep<T>*
    ptr()
    { return (CGAL__Tee_for_output_iterator_rep<T>*)PTR; }

  protected:
    OutputIterator o_it;
  };

  template <class T>
  class CGAL__Tee_for_output_iterator_rep : public CGAL_Rep
  {
  public:
    vector<T> output_so_far;
  };

  template <class OutputIterator, class T>
  inline
  output_iterator_tag
  iterator_category(const CGAL_Tee_for_output_iterator<OutputIterator,T>&)
  { return output_iterator_tag(); }

  template <class OutputIterator, class T>
  inline
  T*
  value_type(const CGAL_Tee_for_output_iterator<OutputIterator,T>&)
  { return (T*)0; }
```

The following file is included in `CGAL/stl_extensions.h`.

⟨*Tee_for_output_iterator.h*⟩≡

```
  ⟨CGAL header⟩
  // file          : Tee_for_output_iterator.h
  // source        : convex_hull_2.lw
  ⟨author notice⟩

  #ifndef CGAL_TEE_FOR_OUTPUT_ITERATOR_H
  #define CGAL_TEE_FOR_OUTPUT_ITERATOR_H

  #include <iterator.h>
  #include <vector.h>
  #include <CGAL/Handle.h>

  ⟨tee for output iterator⟩

  #endif // CGAL_TEE_FOR_OUTPUT_ITERATOR_H
```

C Further Selected Extreme Point Computation Code

⟨*nswe extremepoints inline declaration*⟩+≡

```
template <class ForwardIterator, class R>
inline
void
CGAL_ch_n_point( ForwardIterator first, ForwardIterator last,
                 ForwardIterator& n,
                 CGAL_Point_2<R>* )
{
  CGAL_ch_n_point(first, last, n, CGAL_convex_hull_traits_2<R>() );
}

template <class ForwardIterator>
inline
void
CGAL_ch_n_point( ForwardIterator first, ForwardIterator last,
                 ForwardIterator& n)
{
  CGAL_ch__n_point(first, last, n, value_type(first) );
}

template <class ForwardIterator, class R>
inline
void
CGAL_ch__s_point( ForwardIterator first, ForwardIterator last,
                  ForwardIterator& s,
                  CGAL_Point_2<R>* )
{
  CGAL_ch_s_point(first, last, s, CGAL_convex_hull_traits_2<R>() );
}

template <class ForwardIterator>
inline
void
CGAL_ch_s_point( ForwardIterator first, ForwardIterator last,
                 ForwardIterator& s)
{
  CGAL_ch__s_point(first, last, s, value_type(first) );
}

template <class ForwardIterator, class R>
inline
void
CGAL_ch__e_point( ForwardIterator first, ForwardIterator last,
                  ForwardIterator& e,
                  CGAL_Point_2<R>* )
{
  CGAL_ch__e_point(first, last, e, CGAL_convex_hull_traits_2<R>() );
}

template <class ForwardIterator>
inline
void
CGAL_ch__e_point( ForwardIterator first, ForwardIterator last,
                  ForwardIterator& e)
{
  CGAL_ch__e_point(first, last, e, value_type(first) );
}

template <class ForwardIterator, class R>
inline
void
CGAL_ch__w_point( ForwardIterator first, ForwardIterator last,
```

```
                  ForwardIterator& w,
                  CGAL_Point_2<R>* )
{
  CGAL_ch_w_point(first, last, w, CGAL_convex_hull_traits_2<R>() );
}

template <class ForwardIterator>
inline
void
CGAL_ch_w_point( ForwardIterator first, ForwardIterator last,
                      ForwardIterator& w)
{
 CGAL_ch__w_point(first, last, w, value_type(first) );
}

template <class ForwardIterator, class R>
inline
void
CGAL_ch__ns_point( ForwardIterator first, ForwardIterator last,
                   ForwardIterator& n,
                   ForwardIterator& s,
                   CGAL_Point_2<R>* )
{
  CGAL_ch_ns_point(first, last, n, s, CGAL_convex_hull_traits_2<R>() );
}

template <class ForwardIterator>
inline
void
CGAL_ch_ns_point( ForwardIterator first, ForwardIterator last,
                  ForwardIterator& n,
                  ForwardIterator& s)
{
  CGAL_ch__ns_point(first, last, n, s, value_type(first) );
}

template <class ForwardIterator, class R>
inline
void
CGAL_ch__we_point( ForwardIterator first, ForwardIterator last,
                   ForwardIterator& w,
                   ForwardIterator& e,
                   CGAL_Point_2<R>* )
{
  CGAL_ch_we_point(first, last, w, e, CGAL_convex_hull_traits_2<R>() );
}

template <class ForwardIterator>
inline
void
CGAL_ch_we_point( ForwardIterator first, ForwardIterator last,
                  ForwardIterator& w,
                  ForwardIterator& e)
{
  CGAL_ch__we_point(first, last, w, e, value_type(first) );
}
```

⟨*nswe extremepoints declaration with traits*⟩+≡

```
template <class ForwardIterator, class Traits>
void
CGAL_ch_ns_point( ForwardIterator first, ForwardIterator last,
                  ForwardIterator& n,
                  ForwardIterator& s,
```

```
                    const Traits& ch_traits );

template <class ForwardIterator, class Traits>
void
CGAL_ch_we_point( ForwardIterator first, ForwardIterator last,
                  ForwardIterator& w,
                  ForwardIterator& e,
                  const Traits& ch_traits );

template <class ForwardIterator, class Traits>
void
CGAL_ch_n_point( ForwardIterator first, ForwardIterator last,
                 ForwardIterator& n,
                 const Traits& ch_traits );

template <class ForwardIterator, class Traits>
void
CGAL_ch_s_point( ForwardIterator first, ForwardIterator last,
                 ForwardIterator& s,
                 const Traits& ch_traits );

template <class ForwardIterator, class Traits>
void
CGAL_ch__e_point( ForwardIterator first, ForwardIterator last,
                  ForwardIterator& e,
                  const Traits& ch_traits );

template <class ForwardIterator, class Traits>
void
CGAL_ch_w_point( ForwardIterator first, ForwardIterator last,
                 ForwardIterator& w,
                 const Traits& ch_traits );
```

⟨*nswe extremepoints*⟩+≡

```
template <class ForwardIterator, class Traits>
void
CGAL_ch_we_point( ForwardIterator first, ForwardIterator last,
                  ForwardIterator& w,
                  ForwardIterator& e,
                  const Traits& ch_traits)
{
 typename Traits::Less_xy    lexicographically_xy_smaller;
 typename Traits::Greater_xy lexicographically_xy_larger;
 w = e = first;
 while ( first != last )
 {
    if ( lexicographically_xy_smaller( *first, *w ))  w = first;
    if ( lexicographically_xy_larger ( *first, *e ))  e = first;
    ++first;
 }
}

template <class ForwardIterator, class Traits>
void
CGAL_ch_ns_point( ForwardIterator first, ForwardIterator last,
                  ForwardIterator& n,
                  ForwardIterator& s,
                  const Traits& )
{
 typename Traits::Less_yx    lexicographically_yx_smaller;
 typename Traits::Greater_yx lexicographically_yx_larger;
 n = s = first;
 while ( first != last )
 {
```

```
    if ( lexicographically_yx_smaller( *first, *s ))  s = first;
    if ( lexicographically_yx_larger ( *first, *n ))  n = first;
    ++first;
 }
}

template <class ForwardIterator, class Traits>
void
CGAL_ch_n_point( ForwardIterator first, ForwardIterator last,
                 ForwardIterator& n,
                 const Traits& )
{
 typename Traits::Greater_yx lexicographically_yx_larger;
 n = first;
 while ( first != last )
 {
    if ( lexicographically_yx_larger ( *first, *n ))  n = first;
    ++first;
 }
}

template <class ForwardIterator, class Traits>
void
CGAL_ch_s_point( ForwardIterator first, ForwardIterator last,
                 ForwardIterator& s,
                 const Traits& )
{
 typename Traits::Less_yx    lexicographically_yx_smaller;
 s = first;
 while ( first != last )
 {
    if ( lexicographically_yx_smaller( *first, *s ))  s = first;
    ++first;
 }
}

template <class ForwardIterator, class Traits>
void
CGAL_ch_e_point( ForwardIterator first, ForwardIterator last,
                 ForwardIterator& e,
                 const Traits& )
{
 typename Traits::Greater_xy  lexicographically_xy_larger;
 e = first;
 while ( first != last )
 {
    if ( lexicographically_xy_larger( *first, *e ))  e = first;
    ++first;
 }
}

template <class ForwardIterator, class Traits>
void
CGAL_ch_w_point( ForwardIterator first, ForwardIterator last,
                 ForwardIterator& w,
                 const Traits& )
{
 typename Traits::Less_xy    lexicographically_xy_smaller;
 w = first;
 while ( first != last )
 {
```

```
    if ( lexicographically_xy_smaller( *first, *w ))  w = first;
    ++first;
 }
}
```

D Predicate Objects on Points

We provide some functions objects for predicates on points. These function objects are useful in combination with the `<algorithm>`s provided by the Standard Template Library. We start with unary sidedness predicates in Subsection D.1. In Subsections D.2 to D.5 we provide binary predicate objects related to orders and hence to sorting. In Subsection D.6 we provide orientation predicates. The function objects use corresponding functions from the CGAL-kernel. Those function objects that are parameterized by a point type, can be used with other point types as well, if predicate functions with the names used in the CGAL-kernel exist.

D.1 Sidedness Predicates

Often a set of points is partitioned by a line. To enable the use of `partition`-algorithm we provide function objects for sidedness predicates with respect to a (directed) line defined by two points. We have two versions, one explicitly constructing the line, and one remembering the points without line construction. The latter allows us to use the original point data in the sidedness test instead of the computed line data, which might be affected by rounding errors. They have suffix `_2p`.

⟨*sidedness predicates constructing a line*⟩≡

```
template <class R>
class CGAL_r_Right_of_line
{
public:
        CGAL_r_Right_of_line(const CGAL_Point_2<R>& a,
                             const CGAL_Point_2<R>& b)
         : l_ab( a, b )
        {}

  bool  operator()(const CGAL_Point_2<R>& c) const
        {
          if ( l_ab.is_degenerate() ) return false;
          return (l_ab.oriented_side(c) == CGAL_ON_NEGATIVE_SIDE);
        }

private:
  CGAL_Line_2<R>    l_ab;
};
```

⟨*sidedness predicates constructing a line*⟩+≡

```
template <class R>
class CGAL_r_Left_of_line
{
public:
        CGAL_r_Left_of_line(const CGAL_Point_2<R>& a,
                            const CGAL_Point_2<R>& b)
         : l_ab( a, b )
        {}

  bool  operator()(const CGAL_Point_2<R>& c) const
        { return (l_ab.oriented_side(c) == CGAL_ON_POSITIVE_SIDE); }

private:
  CGAL_Line_2<R>    l_ab;
};
```

⟨*sidedness predicates on points*⟩≡
```
template <class Point>
class CGAL_p_Left_of_line_2p
{
public:
        CGAL_p_Left_of_line_2p(const Point& a, const Point& b)
         : p_a(a), p_b(b)
        {}

  bool  operator()(const Point& c) const
        { return CGAL_leftturn( p_a, p_b, c ); }

private:
    Point p_a;
    Point p_b;
};

template <class Point>
class CGAL_p_Right_of_line_2p
{
public:
        CGAL_p_Right_of_line_2p(const Point& a, const Point& b)
         : p_a(a), p_b(b)
        {}

  bool  operator()(const Point& c) const
        { return CGAL_rightturn( p_a, p_b, c ); }

private:
    Point p_a;
    Point p_b;
};

template <class Point>
class CGAL_p_Right_of_line_2p_safer
{
public:
        CGAL_p_Right_of_line_2p_safer(const Point& a, const Point& b)
         : p_a(a), p_b(b)
        {}

  bool  operator()(const Point& c) const
        {
          if ( (c == p_a) || ( c == p_b ) ) return false;
          return CGAL_rightturn( p_a, p_b, c );
        }

private:
    Point p_a;
    Point p_b;
};
```

D.2 Lexicographical Order

We provide function objects used to sort points with respect to different lexicographical orders. Following STL-notation the names of the function objects contain `Less` and `Greater` instead of `smaller` and `larger` which are used in CGAL.

⟨*lex order predicates*⟩≡
```
template <class Point>
struct CGAL_p_Less_xy
{
  bool operator()( const Point& p1, const Point& p2)
       { return CGAL_lexicographically_xy_smaller( p1, p2); }
```

```
};

template <class Point>
struct CGAL_p_Greater_xy
{
  bool operator()( const Point& p1, const Point& p2)
       { return CGAL_lexicographically_xy_larger( p1, p2); }
};

template <class Point>
struct CGAL_p_Less_yx
{
  bool operator()( const Point& p1, const Point& p2)
       { return CGAL_lexicographically_yx_smaller( p1, p2); }
};

template <class Point>
struct CGAL_p_Greater_yx
{
  bool operator()( const Point& p1, const Point& p2)
       { return CGAL_lexicographically_yx_larger( p1, p2); }
};
```

D.3 Counterclockwise Rotation Order

The following binary predicate decides whether point `p` is hit before `q` by a line rotated counterclockwise about `rot_point` before `q`. If the three points are collinear the furthest point is considered to be smaller. The predicate is used in Jarvis algorithm [30] for computing the next extreme point in counterclockwise order, i.e. the next point wrapped by the gift-wrapping. We start with a version using the

- `bool CGAL_collinear_are_ordered_along_line(Point, Point, Point)`

predicate. Another version, marked with an additional `_E` for experimental, uses a predicate for comparing distances to a point.

- `bool CGAL_has_larger_dist_to_point(Point, Point, Point)`

⟨*less - rotate ccw*⟩≡
```
template <class Point>
class CGAL_p_Less_rotate_ccw
{
public:
        CGAL_p_Less_rotate_ccw(const Point& p)
        : rot_point(p)
        {}

  bool  operator()(const Point& p, const Point& q)
        {
          CGAL_Orientation ori = CGAL_orientation(rot_point, p, q);
          if ( ori == CGAL_LEFTTURN )
          {
              return true;
          }
          else if ( ori == CGAL_RIGHTTURN )
          {
              return false;
          }
          else
          {
              if (p == rot_point) return false;
              if (q == rot_point) return true;
              if (p == q)         return false;
              return  CGAL_collinear_are_ordered_along_line( rot_point, q, p);
```

```
          }
        }
  ⟨set rotation center⟩
private:
    Point  rot_point;
};
template <class Point>
class CGAL_p_Less_rotate_ccw_safer
{
public:
        CGAL_p_Less_rotate_ccw_safer(const Point& p)
        : rot_point(p)
        {}
  bool  operator()(const Point& p, const Point& q)
        {
          if (p == rot_point) return false;
          if (q == rot_point) return true;
          if (p == q)         return false;
          CGAL_Orientation ori = CGAL_orientation(rot_point, p, q);
          if ( ori == CGAL_LEFTTURN )
          {
              return true;
          }
          else if ( ori == CGAL_RIGHTTURN )
          {
              return false;
          }
          else
          {
              return  CGAL_collinear_are_ordered_along_line( rot_point, q, p);
          }
        }
  ⟨set rotation center⟩
private:
  Point  rot_point;
};
template <class Point>
class CGAL_p_Less_rotate_ccw_E
{
public:
        CGAL_p_Less_rotate_ccw_E(const Point& p)
        : rot_point(p)
        {}
  bool  operator()(const Point& p, const Point& q)
        {
          CGAL_Orientation ori = CGAL_orientation(rot_point, p, q);
          if ( ori == CGAL_LEFTTURN )
          {
              return true;
          }
          else if ( ori == CGAL_RIGHTTURN )
          {
              return false;
          }
          else
          {
              return  CGAL_has_larger_dist_to_point( rot_point, p, q) ;
          }
        }
```

```
  ⟨set rotation center⟩

private:
  Point  rot_point;
};
```

To avoid copying, there is a function to set the rotation center.

⟨*set rotation center*⟩≡

```
void  set_rotation_center( const Point& p)
      { rot_point = p; }
```

D.4 Orders Based on Distance to Line

The less-than operations described in this section are based on the signed distance to a line given by two points. Ties are broken by comparing the distance to the first of the two points defining the line. Again we provide versions that explicitly construct the line on which the predicate is based and versions maintaining the points defining the line. The latter have suffix _2p.

⟨*distance to line predicate on points*⟩≡

```
template <class Point>
class CGAL_p_Less_dist_to_line_2p
{
public:
        CGAL_p_Less_dist_to_line_2p(const Point& a, const Point& b)
         : p_a(a), p_b(b)
        {}

  bool  operator()(const Point& c, const Point& d) const
        {
          CGAL_Comparison_result
            res = CGAL_cmp_signed_dist_to_line( p_a, p_b, c, d);
          if ( res == CGAL_LARGER )
          {
              return false;
          }
          else if ( res == CGAL_SMALLER )
          {
              return true;
          }
          else
          {
              return CGAL_lexicographically_xy_smaller( c, d );
          }
        }

private:
  Point          p_a;
  Point          p_b;
};
```

⟨*distance to line predicate constructing a line*⟩≡

```
template <class R>
class CGAL_r_Less_dist_to_line
{
public:
        CGAL_r_Less_dist_to_line(const CGAL_Point_2<R>& a,
                                 const CGAL_Point_2<R>& b)
         : l_ab( a, b )
        {}
```

```
  bool  operator()(const CGAL_Point_2<R>& c, const CGAL_Point_2<R>& d) const
        {
          CGAL_Comparison_result res = CGAL_cmp_signed_dist_to_line(l_ab, c, d);
          if ( res == CGAL_LARGER )
          {
              return false;
          }
          else if ( res == CGAL_EQUAL )
          {
              return CGAL_lexicographically_xy_smaller( c, d );
          }
          else
          {
              return true;
          }
        }
private:
  CGAL_Line_2<R>    l_ab;
};
```

Sometimes, we are interested in maximizing the distance of the points right of the directed line. Since the signed distance of these points is negative minimizing the signed distance gives us the maximum distance point that we are looking for.

⟨*distance to line predicate on points*⟩+≡

```
template <class Point>
class CGAL_p_Less_negative_dist_to_line_2p
{
public:
        CGAL_p_Less_negative_dist_to_line_2p(const Point& a, const Point& b)
         : p_a(a), p_b(b)
        {}
  bool  operator()(const Point& c, const Point& d) const
        {
          CGAL_Comparison_result
            res = CGAL_cmp_signed_dist_to_line( p_a, p_b, c, d);
          if ( res == CGAL_LARGER )
          {
              return true;
          }
          else if ( res == CGAL_SMALLER )
          {
              return false;
          }
          else
          {
              return CGAL_lexicographically_xy_smaller( c, d );
          }
        }
private:
  Point          p_a;
  Point          p_b;
};
```

⟨*distance to line predicate constructing a line*⟩+≡

```
template <class R>
class CGAL_r_Less_negative_dist_to_line
{
public:
```

```
        CGAL_r_Less_negative_dist_to_line(const CGAL_Point_2<R>& a,
                                          const CGAL_Point_2<R>& b)
         : l_ab( a, b )
        {}
   bool operator()(const CGAL_Point_2<R>& c, const CGAL_Point_2<R>& d) const
        {
          CGAL_Comparison_result res = CGAL_cmp_signed_dist_to_line(l_ab, c, d);
          if ( res == CGAL_LARGER )
          {
              return true;
          }
          else if ( res == CGAL_EQUAL )
          {
              return CGAL_lexicographically_xy_smaller( c, d );
          }
          else
          {
              return false;
          }
        }
private:
  CGAL_Line_2<R>    l_ab;
};
```

D.5 Direction Based Orders

Sometimes one would like to sort points according to directions others than the directions of the coordinate axes. Here we provide a predicate object that takes a direction and compares points according to this direction. Ties are broken by the total lexicographical `xy`-order. The constructor creates the line through the origin with the direction rotated counterclockwise by 90 degrees. The predicate is useful for the computation of a tangent point of a point set in a given direction. We use

⟨*order in a given direction*⟩≡

```
template <class R>
class CGAL_r_Less_in_direction
{
public:
        typedef typename  R::RT   RT;
        CGAL_r_Less_in_direction( const CGAL_Direction_2<R>& dir )
        : l( CGAL_Point_2<R>( RT(0) , RT(0) ),
             CGAL_Direction_2<R>(-(dir.dy()), dir.dx() ))
        {}
   bool operator()(const CGAL_Point_2<R>& c, const CGAL_Point_2<R>& d)
        {
          CGAL_Comparison_result res = CGAL_cmp_signed_dist_to_line(l, c, d);
          if ( res == CGAL_LARGER )
          {
              return true;
          }
          else if ( res == CGAL_EQUAL )
          {
              return  CGAL_lexicographically_xy_smaller( c, d) ;
          }
          else
          {
              return false;
          }
```

```
        }
private:
  CGAL_Line_2<R>  l;
};
```

D.6 Orientation Predicates

In this section, we provide function objects corresponding to the orientation predicates in the CGAL-kernel. They are parameterized with a point type and can be used with any point if predicate functions with the names used in the CGAL-kernel exist.

⟨*orientation predicate objects*⟩≡

```
template <class Point>
struct CGAL_p_Leftturn
{
  bool  operator()(const Point& p, const Point& q, const Point& r) const
        { return CGAL_leftturn(p,q,r); }
};

template <class Point>
struct CGAL_p_Rightturn
{
  bool  operator()(const Point& p, const Point& q, const Point& r) const
        { return CGAL_rightturn(p,q,r); }
};

template <class Point>
struct CGAL_p_Orientation
{
  CGAL_Orientation
        operator()(const Point& p, const Point& q, const Point& r) const
        { return CGAL_orientation(p,q,r); }
};
```

File predicate_objects_on_points_2.h

Since all the code is defined (inline) in the classes there is only a .h-file and no .C-file to be included on demand.

⟨*predicate_objects_on_points_2.h*⟩≡

```
⟨CGAL header⟩
// file          : predicate_objects_on_points_2.h
// source        : convex_hull_2.lw
⟨author notice⟩

#ifndef CGAL_PREDICATE_OBJECTS_ON_POINTS_2_H
#define CGAL_PREDICATE_OBJECTS_ON_POINTS_2_H

#include <CGAL/user_classes.h>
⟨sidedness predicates on points⟩
⟨lex order predicates⟩
⟨distance to line predicate on points⟩
⟨less - rotate ccw⟩
⟨sidedness predicates constructing a line⟩
⟨distance to line predicate constructing a line⟩
⟨order in a given direction⟩
⟨orientation predicate objects⟩

#endif // CGAL_PREDICATE_OBJECTS_ON_POINTS_2_H
```

E Some Wrapping for LEDA Geometry

In order to use the predicate objects defined in Appendix D we have to provide the global functions that are used there, i.e. to map the names used there to the ones used in LEDA (or to provide functions having no equivalent direct realization in LEDA).

The global predicates used by the predicate objects given in Appendix D are

- `bool CGAL_leftturn(Point, Point, Point)`
- `bool CGAL_rightturn(Point, Point, Point)`
- `CGAL_Orientation CGAL_orientation(Point, Point, Point)`
- `bool CGAL_lexicographically_xy_smaller(Point, Point)`
- `bool CGAL_lexicographically_yx_smaller(Point, Point)`
- `bool CGAL_lexicographically_xy_larger(Point, Point)`
- `bool CGAL_lexicographically_yx_larger(Point, Point)`
- `bool CGAL_collinear_are_ordered_along_line(Point, Point, Point)`
- `CGAL_Comparison_result CGAL_cmp_signed_dist_to_line(Point, Point, Point, Point)`

We do not provide the functions

- `bool CGAL_has_larger_dist_to_point(Point, Point, Point)`
- `bool CGAL_has_smaller_dist_to_point(Point, Point, Point)`

which are used in experimental versions (`_E`) of the predicates only.

`rat_leda_in_CGAL_2.h`

For the orientation predicates we basically have to map names.

⟨*inline orientation predicates*⟩≡

```
inline
bool
CGAL_leftturn( const leda_rat_point & p,
               const leda_rat_point & q,
               const leda_rat_point & r)
{ return  left_turn(p,q,r); }

inline
bool
CGAL_rightturn( const leda_rat_point & p,
                const leda_rat_point & q,
                const leda_rat_point & r)
{ return  right_turn(p,q,r); }
```

For the orientation test we have to adjust the return type in addition. A cast presumes knowledge on the value of the constants in enum `CGAL_Orientation`, thus a switch statement would be safer.

⟨*inline orientation predicates*⟩+≡

```
inline
CGAL_Orientation
CGAL_orientation( const leda_rat_point & p,
                  const leda_rat_point & q,
                  const leda_rat_point & r)
{ return (CGAL_Orientation)orientation(p,q,r); }
```

In the lexicographical comparison operations we use `static` member functions `cmp_xy(const leda_rat_point&, const leda_rat_point&)` and `cmp_yx(const leda_rat_point&, const leda_rat_point&)` of LEDA class `leda_rat_point`. That's a bit dangerous, because they are undocumented and hence subject to change without notice.

⟨*lex comparison*⟩≡
```
inline
bool
CGAL_lexicographically_xy_smaller( const leda_rat_point & p,
                                   const leda_rat_point & q)
{ return ( leda_rat_point::cmp_xy(p,q)  <  0 ); }

inline
bool
CGAL_lexicographically_yx_smaller( const leda_rat_point & p,
                                   const leda_rat_point & q)
{ return ( leda_rat_point::cmp_yx(p,q)  <  0 ); }

inline
bool
CGAL_lexicographically_xy_larger( const leda_rat_point & p,
                                  const leda_rat_point & q)
{ return ( leda_rat_point::cmp_xy(p,q)  >  0 ); }

inline
bool
CGAL_lexicographically_yx_larger( const leda_rat_point & p,
                                  const leda_rat_point & q)
{ return ( leda_rat_point::cmp_yx(p,q)  >  0 ); }
```

Finally, a predicate used to resolve degeneracies in Jarvis' march. We make it `inline` such that we neither need a lib for interactions with LEDA nor have to make the CGAL lib dependent on LEDA stuff. I'm not sure that the function body is correct in the degenerate case, where `q` is equal to one of the other points. The LEDA manual is not very precise here. I guess, for `rat_segment`s "contains" means "including the endpoints".

⟨*collinear between*⟩≡
```
inline
bool
CGAL_collinear_are_ordered_along_line( const leda_rat_point & p,
                                       const leda_rat_point & q,
                                       const leda_rat_point & r)
{ return (leda_rat_segment(p,r).contains(q) && ( q != p ) && ( q != r )); }
```

The distance comparison with respect to a line was not well supported before LEDA 3.6; there was no predicate using floating-point filters.

⟨*dist to line*⟩≡
```
inline
CGAL_Comparison_result
CGAL_cmp_signed_dist_to_line( const leda_rat_point & p, const leda_rat_point & q,
                              const leda_rat_point & r, const leda_rat_point & s )
{
#if ( __LEDA__ >= 360 )
  return (CGAL_Comparison_result)cmp_signed_dist(p,q,r,s);
#else
  leda_rat_line  l(p,q);
  int  r_or = orientation( l, r );
  int  s_or = orientation( l, s );
  if ( r_or != s_or )
  {
      return (CGAL_Comparison_result)( r_or < s_or );
  }
```

```
  else
  {
      return (CGAL_Comparison_result)(r_or *( sign(l.sqr_dist(r) - l.sqr_dist(s) )));
  }
#endif  // __LEDA__ >= 360
}
```

⟨*rat_leda_in_CGAL_2.h*⟩≡

```
⟨CGAL header⟩
// file          : rat_leda_in_CGAL_2.h
// source        : convex_hull_2.lw
⟨author notice⟩

#ifndef RAT_LEDA_IN_CGAL_H
#define RAT_LEDA_IN_CGAL_H

#include <CGAL/enum.h>
#include <LEDA/rat_point.h>
#include <LEDA/rat_segment.h>
#include <LEDA/rat_line.h>

⟨inline orientation predicates⟩
⟨lex comparison⟩
⟨collinear between⟩
⟨dist to line⟩

#endif // RAT_LEDA_IN_CGAL_H
```

leda_in_CGAL_2.h

Analogously, we warp plain LEDA geometry.

⟨*leda inline predicates*⟩≡

```
inline
bool
CGAL_leftturn( const leda_point & p,
               const leda_point & q,
               const leda_point & r)
{ return  left_turn(p,q,r); }

inline
bool
CGAL_rightturn( const leda_point & p,
                const leda_point & q,
                const leda_point & r)
{ return  right_turn(p,q,r); }

inline
CGAL_Orientation
CGAL_orientation( const leda_point & p,
                  const leda_point & q,
                  const leda_point & r)
{ return (CGAL_Orientation)orientation(p,q,r); }

inline
bool
CGAL_lexicographically_xy_smaller( const leda_point & p,
                                   const leda_point & q)
{ return ( leda_point::cmp_xy(p,q)  <  0 ); }

inline
bool
CGAL_lexicographically_yx_smaller( const leda_point & p,
                                   const leda_point & q)
{ return ( leda_point::cmp_yx(p,q)  <  0 ); }
```

```
inline
bool
CGAL_lexicographically_xy_larger( const leda_point & p,
                                  const leda_point & q)
{ return ( leda_point::cmp_xy(p,q)  >  0 ); }

inline
bool
CGAL_lexicographically_yx_larger( const leda_point & p,
                                  const leda_point & q)
{ return ( leda_point::cmp_yx(p,q)  >  0 ); }

inline
bool
CGAL_collinear_are_ordered_along_line( const leda_point & p,
                                       const leda_point & q,
                                       const leda_point & r)
{
  return
  (   (( leda_point::cmp_xy(p,q)<=0 )&&( leda_point::cmp_xy(q,r)<=0 ))
    ||(( leda_point::cmp_xy(r,q)<=0 )&&( leda_point::cmp_xy(q,p)<=0 )) );
}

inline
CGAL_Comparison_result
CGAL_cmp_signed_dist_to_line( const leda_point & p, const leda_point & q,
                              const leda_point & r, const leda_point & s )
{
#if ( __LEDA__ >= 360 )
  return (CGAL_Comparison_result)cmp_signed_dist(p,q,r,s);
#else
  leda_line  l(p,q);
  int  r_or = orientation( l, r );
  int  s_or = orientation( l, s );
  if ( r_or != s_or )
  {
      return (CGAL_Comparison_result)( r_or < s_or );
  }
  else
  {
      return (CGAL_Comparison_result)(r_or *( sign(l.sqr_dist(r) - l.sqr_dist(s) )));
  }
#endif  // __LEDA__ >= 360
}
```

⟨*leda_in_CGAL_2.h*⟩≡

```
⟨CGAL header⟩
// file          : leda_in_CGAL_2.h
// source        : convex_hull_2.lw
⟨author notice⟩

#ifndef LEDA_IN_CGAL_H
#define LEDA_IN_CGAL_H

#include <CGAL/enum.h>
#include <LEDA/point.h>
#include <LEDA/segment.h>
#include <LEDA/line.h>
⟨leda inline predicates⟩

#endif // LEDA_IN_CGAL_H
```