

Chapter 2

Basic Arrangements

We start with a formal definition of two-dimensional arrangements, and proceed with an introduction to the data structure used to represent the incidence relations among features of two-dimensional arrangements, namely, the *doubly-connected edge list*, or DCEL for short. Then we describe the main class of the *2D Arrangements* package and the functions it supports. This chapter contains the basic material you need to know in order to use CGAL arrangements.

2.1 Representation of Arrangements: The DCEL

Given a set \mathcal{C} of planar curves, the *arrangement* $\mathcal{A}(\mathcal{C})$ is the subdivision of the plane into zero-dimensional, one-dimensional, and two-dimensional *cells*,¹ called *vertices*, *edges*, and *faces*, respectively, induced by the curves in \mathcal{C} .

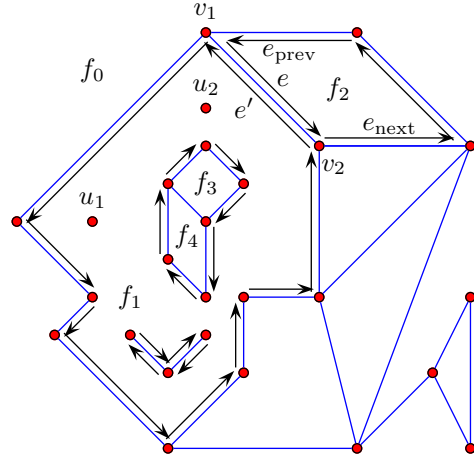
The curves in \mathcal{C} can intersect each other (a single curve may also be self-intersecting or may comprise several disconnected branches) and are not necessarily *x-monotone*.² We construct in two steps a collection \mathcal{C}'' of *x-monotone* subcurves that are pairwise disjoint in their interiors as follows. First, we decompose each curve in \mathcal{C} into maximal *x-monotone* subcurves and possibly isolated points, obtaining the collection \mathcal{C}' . Note that an *x-monotone* curve cannot be self-intersecting. Then, we decompose each curve in \mathcal{C}' into maximal connected subcurves not intersecting any other curve (or point) in \mathcal{C}' in its interior. The collection \mathcal{C}'' contains isolated points if the collection \mathcal{C}' contains such points. The arrangement induced by the collection \mathcal{C}'' can be conveniently embedded as a planar graph, the vertices of which are associated with curve endpoints or with isolated points, and the edges of which are associated with subcurves. It is easy to see that the faces of $\mathcal{A}(\mathcal{C})$ are the same as the faces of $\mathcal{A}(\mathcal{C}'')$. There are possibly more vertices in $\mathcal{A}(\mathcal{C}'')$ than in $\mathcal{A}(\mathcal{C})$ —the vertices where curves were cut into *x-monotone* (non-intersecting) pieces; accordingly there may also be more edges in $\mathcal{A}(\mathcal{C}'')$. This graph can be represented using a *doubly-connected edge list* (DCEL) data structure, which consists of containers of vertices, edges, and faces and maintains the incidence relations among these cells. It is one of a family of combinatorial data structures called *halfedge data structures* (HDS), which are edge-centered data structures capable of maintaining incidence relations among cells of, for example, planar subdivisions, polyhedra, or other orientable, two-dimensional surfaces embedded in space of an arbitrary dimension. Geometric interpretation is added by classes built on top of the halfedge data structure.

The DCEL data structure represents each edge using a pair of directed *halfedges*, one going from the *xy*-lexicographically smaller (left) endpoint of the curve towards the *xy*-lexicographically

¹We use the term *cell* to describe the various dimensional entities in the induced subdivision. Sometimes, the term *face* is used for this purpose in the literature. However, in this book, we use the term *face* to describe a *two-dimensional* cell.

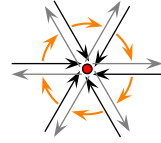
²A continuous planar curve C is *x-monotone* if every vertical line intersects it at most once. For example, a non-vertical line segment is always *x-monotone*, and so is the graph of any continuous function $y = f(x)$. A circle of radius r centered at (x_0, y_0) is not *x-monotone*, as the vertical line $x = x_0$ intersects it at $(x_0, y_0 - r)$ and at $(x_0, y_0 + r)$. For convenience, we always deal with *weakly x-monotone* curves, which include vertical linear curves.

Fig. 2.1: An arrangement of interior-disjoint line-segments with some of the DCEL records that represent it. The unbounded face f_0 has a single connected component that forms a hole inside it, and this hole consists of several faces. The halfedge e is directed from its source vertex v_1 to its target vertex v_2 . This halfedge, together with its twin e' , corresponds to a line segment that connects the points associated with v_1 and v_2 and separates the face f_1 from f_2 . The predecessor e_{prev} and successor e_{next} of e are part of the chain that forms the boundary of the face f_2 . The face f_1 has a more complicated structure, as it contains two holes in its interior: One hole contains two faces f_3 and f_4 , while the other hole consists of just two edges. f_1 also contains two isolated vertices u_1 and u_2 in its interior.

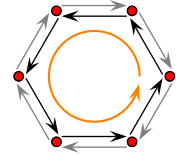


larger (right) endpoint, and the other, known as its *twin* halfedge, going in the opposite direction.

As each halfedge is directed, it has a *source* vertex and a *target* vertex. Halfedges are used to separate faces and to connect vertices, with the exception of *isolated vertices* (representing isolated points), which are disconnected. If a vertex v is the target of a halfedge e , we say that v and e are *incident* to each other. The halfedges incident to a vertex v form a circular list sorted in clockwise order around this vertex; see the figure to the right. (An isolated vertex has no incident halfedges.)



An *edge* of an arrangement is a maximal portion of a curve between two vertices of the arrangement. Each edge is represented in the DCEL by a pair of twin halfedges. Each halfedge e stores a pointer to its *incident face*, which is the face lying to its left. Moreover, every halfedge is followed by another halfedge sharing the same incident face, such that the target vertex of the halfedge is the same as the source vertex of the next halfedge. The halfedges around faces form circular chains, such that all halfedges of a chain are incident to the same face and wind along its boundary; see the figure above. We call such a chain a *connected component of the boundary*, or CCB for short.



The unique CCB of halfedges winding in a counterclockwise orientation along a face boundary is referred to as the *outer CCB* of the face. For the time being, let us consider only arrangements of bounded curves, such that exactly one unbounded face exists in every arrangement. The unbounded face does not have an outer boundary. Any other connected component of the boundary of the face is called a *hole*, or *inner CCB*, and can be represented as a circular chain of halfedges winding in a clockwise orientation around it. Note that a hole does not necessarily correspond to a face at all, as it may have no area, or alternatively it may contain several faces inside it. Every face can have several holes in its interior, or may contain no holes at all. In addition, every face may contain isolated vertices in its interior. See Figure 2.1 for an illustration of the various DCEL features.

So much on the abstract description of the DCEL. This is the underlying data structure of the CGAL arrangement class. Occasionally, this is the only data structure we need, especially when we are only concerned with traversing the arrangement.

2.2 The Main Arrangement Class

The main component in the *2D Arrangements* package is the `Arrangement_2<Traits,Dcel>` class template. An instance of this template is used to represent planar arrangements. The class template provides the interface needed to construct such arrangements, traverse them, and

maintain them.

The design of the *2D Arrangements* package is guided by two aspects of modularity as follows: (i) the separation of the representation of the arrangements and the various geometric algorithms that operate on them, and (ii) the separation of the topological and geometric aspects of the planar subdivision. The latter separation is exhibited by the two template parameters of the `Arrangement_2` class template; their description follows.

- The `Traits` template parameter should be substituted with a model of one of the geometry traits concepts, for example, the `ArrangementBasicTraits_2` concept. A model of this traits concept defines the types of x -monotone curves and two-dimensional points, `X_monotone_curve_2` and `Point_2`, respectively, and supports basic geometric predicates on them.

In this chapter we always use `Arr_non_caching_segment_traits_2` as our traits-class model in order to construct arrangements of line segments. In Chapter 3 we also use `Arr_segment_traits_2` as our traits-class model. In Chapter 4 we use `Arr_linear_traits_2` to construct arrangements of linear curves (i.e., lines, rays, and line segments). The *2D Arrangements* package contains several other traits classes that can handle other types of curves, such as polylines (continuous piecewise-linear curves), conic arcs, and arcs of rational functions. We exemplify the usage of these traits classes in Chapter 5. A few additional models have been developed by other groups of researchers.

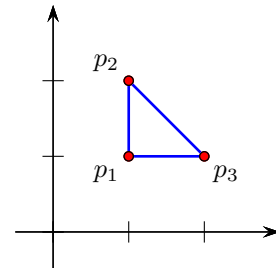
- The `Dcel` template parameter should be substituted with a class that models the `ArrangementDcel` concept, which is used to represent the topological layout of the arrangement. This parameter is substituted with `Arr_default_dcel<Traits>` by default, and we use this default value in this and in the following three chapters. However, in many applications it is necessary to extend the DCEL features. This is done by substituting the `Dcel` parameter with a different type; see Section 6.2 for further explanation and examples.

The function template `print_arrangement_size()` listed below prints out quantitative measures of a given arrangement. While in what follows it is used only by examples, it demonstrates well the use of the member functions `number_of_vertices()`, `number_of_edges()`, and `number_of_faces()`, which return the number of vertices, edges, and faces of an arrangement, respectively. The function template is defined in the header file `Arr_print.h`.

```
template <typename Arrangement>
void print_arrangement_size(const Arrangement& arr)
{
    std::cout << "The_arrangement_size:" << std::endl
               << "    |V|=" << arr.number_of_vertices()
               << "    |E|=" << arr.number_of_edges()
               << "    |F|=" << arr.number_of_faces() << std::endl;
}
```

You can also obtain the number of halfedges of an arrangement using the member function `number_of_halfedges()`. Recall that the number of halfedges is always twice the number of edges.

Example: The simple program listed below constructs an arrangement of three connected line-segments forming a triangle. It uses the CGAL *Cartesian kernel* (see Section 1.4.4) with an integral-number type to instantiate the `Arr_segment_traits_2` class template. The resulting arrangement consists of two faces, a bounded triangular face and the unbounded face. Constructing and maintaining arrangements using limited-precision numbers, such as `int`, works properly only under severe restrictions, which in many cases render the program not very useful. In this example, however, the points are far apart, and constructions of new geometric objects do not occur. Thus, it is safe to use `int` after all. The program constructs an arrangement induced by three line segments that are pairwise disjoint in their interior, prints out the number of faces, and ends. It uses the `insert()` free-function, which inserts the segments into



the arrangement; see Section 3.4. It uses the member function `number_of_faces()` to obtain the number of faces (two in this case). We give more elaborate examples in the rest of this chapter. The programs in those examples rely on computing with numbers of arbitrary precision, which guarantees robust execution and correct results.

```
// File: ex_triangle.cpp

#include <CGAL/ Cartesian.h>
#include <CGAL/ Arr_non_caching_segment_traits_2.h>
#include <CGAL/ Arrangement_2.h>

typedef int                                     Number_type;
typedef CGAL:: Cartesian<Number_type>          Kernel;
typedef CGAL:: Arr_non_caching_segment_traits_2<Kernel> Traits;
typedef Traits::Point_2                        Point;
typedef Traits::X_monotone_curve_2             Segment;
typedef CGAL:: Arrangement_2<Traits>           Arrangement;

int main()
{
    Point      p1(1, 1), p2(1, 2), p3(2, 1);
    Segment    cv[] = {Segment(p1, p2), Segment(p2, p3), Segment(p3, p1)};
    Arrangement arr;
    insert(arr, &cv[0], &cv[sizeof(cv)/sizeof(Segment)]);
    std::cout << "Number_of_faces:_" << arr.number_of_faces() << std::endl;
    return 0;
}
```



Try: Modify the program above so that it inserts the number of vertices and halfedges as well as the number of faces into the standard output-stream.

2.2.1 Traversing the Arrangement

The simplest and most fundamental arrangement operations are the various traversal methods, which allow you to systematically go over the relevant features of the arrangement at hand.

Since the arrangement is represented as a DCEL, which stores containers of vertices, halfedges, and faces, the `Arrangement_2` class template supplies iterators for these containers. For example, if `arr` is an `Arrangement_2` object, the calls `arr.vertices_begin()` and `arr.vertices_end()` return iterators of the nested `Arrangement_2::Vertex_iterator` type that define the valid range of arrangement vertices. The value type of this iterator is `Arrangement_2::Vertex`. Moreover, the vertex-iterator type is convertible to `Arrangement_2::Vertex_handle`, which serves as a pointer to a vertex. As we show next, all functions related to arrangement features accept handle types as input parameters and return handle types as their output. A handle models the STL concept `TrivialIterator`.³ Throughout this book, we use the identifiers `v`, `he`, and `f` to refer to a vertex handle, a halfedge handle, and a face handle, respectively.

In addition to the iterators for arrangement vertices, halfedges, and faces, the `Arrangement_2` class template also provides an iterator for edges, namely `Arrangement_2::Edge_iterator`. The value type of this iterator is `Arrangement_2::Halfedge`, which is used to represent one of the twin halfedges associated with the edge. The calls `arr.edges_begin()` and `arr.edges_end()` return iterators that define the valid range of arrangement edges.

All iterator, circulator,⁴ and handle types also have non-mutable (`const`) counterparts. These non-mutable iterators are useful for traversing an arrangement without changing it. For example,

³A handle is a lightweight object that behaves like a pointer; hence, it is more efficient to pass handles around.

⁴A *circulator* is used when traversing a circular list, such as the list of halfedges incident to a vertex.

the arrangement has a mutable member-function called `arr.vertices_begin()` that returns an `Arrangement_2::Vertex_iterator` object and another non-mutable member-function that returns an `Arrangement_2::Vertex_const_iterator` object. In fact, all methods listed in this section that return an iterator, a circulator, or a handle have non-mutable counterparts. It should be noted that, for example, an `Arrangement_2::Vertex_handle` is convertible into an `Arrangement_2::Vertex_const_handle`, but not the other way around.

Conversions of non-mutable handles to the corresponding mutable handles are nevertheless possible. They can be performed using the overloaded member-function `non_const_handle()`. There are three variants that accept a non-mutable handle to a vertex, a halfedge, or a face, respectively. Only mutable objects of type `Arrangement_2` can call the `non_const_handle()` method; see, e.g., Section 3.1.1.

Traversal Methods for an Arrangement Vertex

A vertex v of an arrangement induced by bounded curves is always associated with a geometric entity, namely, with a `Point_2` object, which can be obtained by `v->point()`. Recall that `v` identifies a vertex handle; hence, we treat it as a pointer.

The call `v->is_isolated()` determines whether the vertex v is isolated or not. Recall that the halfedges incident to a non-isolated vertex, namely the halfedges that share a common target vertex, form a circular list around this vertex. The call `v->incident_halfedges()` returns a circulator of the nested type `Arrangement_2::Halfedge_around_vertex_circulator` that enables the traversal of this circular list around a given vertex v in a clockwise order. The value type of this circulator is `Arrangement_2::Halfedge`. By convention, the target of the halfedge is v . The call `v->degree()` evaluates to the number of the halfedges incident to v .



Example: The function below prints all the halfedges incident to a given arrangement vertex (assuming that the `Point_2` type can be inserted into the standard output-stream using the `<<` operator). The arrangement type is the same as in the simple example (coded in `ex_triangle.cpp`) above.

```
template <typename Arrangement>
void print_incident_halfedges (typename Arrangement::Vertex_const_handle v)
{
    if (v->is_isolated()) {
        std::cout << "The_vertex_(" << v->point() << ")_is_isolated" << std::endl;
        return;
    }
    std::cout << "The_neighbors_of_the_vertex_(" << v->point() << ")_are:";
    typename Arrangement::Halfedge_around_vertex_const_circulator first, curr;
    first = curr = v->incident_halfedges();
    do std::cout << "(" << curr->source()->point() << ")";
    while (++curr != first);
    std::cout << std::endl;
}
```

If v is an isolated vertex, the call `v->face()` obtains the face that contains v .

Traversal Methods for an Arrangement Halfedge

A halfedge e of an arrangement induced by bounded curves is associated with an `X_monotone_curve_2` object, which can be obtained by `he->curve()`, where `he` identifies a handle to e .

The calls `he->source()` and `he->target()` return handles to the halfedge source-vertex and target-vertex, respectively. You can obtain a handle to the twin halfedge using `he->twin()`. Note that from the definition of halfedges in the DCEL structure, the following invariants always hold:

- `he->curve()` is equivalent to `he->twin()->curve()`,
- `he->source()` is equivalent to `he->twin()->target()`, and

- `he->target()` is equivalent to `he->twin()->source()`.

Every halfedge has an incident face that lies to its left, which can be obtained by `he->face()`. Recall that a halfedge is always one link in a connected chain (CCB) of halfedges that share the same incident face. The `he->prev()` and `he->next()` calls return handles to the previous and next halfedges in the CCB, respectively.

As the CCB is a circular list of halfedges, it is only natural to traverse it using a circulator. Indeed, `he->ccb()` returns an `Arrangement_2::Ccb_halfedge_circulator` object for traversing all halfedges along the connected component of `he`. The value type of this circulator is `Arrangement_2::Halfedge`.



Example: The function template `print_ccb()` listed below prints all x -monotone curves along a given CCB (assuming that the `Point_2` and the `X_monotone_curve_2` types can be inserted into the standard output-stream using the `<<` operator).

```
template <typename Arrangement>
void print_ccb(typename Arrangement::Ccb_halfedge_const_circulator circ)
{
    std::cout << "(" << circ->source()->point() << ")";
    typename Arrangement::Ccb_halfedge_const_circulator curr = circ;
    do {
        typename Arrangement::Halfedge_const_handle he = curr;
        std::cout << "~~~|" << he->curve() << "|~~~"
            << "(" << he->target()->point() << ")";
    } while (++curr != circ);
    std::cout << std::endl;
}
```

Traversal Methods for an Arrangement Face

An `Arrangement_2` object `arr` that identifies an arrangement of bounded curves always has a single unbounded face. The call `arr.unbounded_face()` returns a handle to this face. Note that an empty arrangement contains nothing *but* the unbounded face.

Given a handle to a face f , you can issue the call `f->is_unbounded()` to determine whether the face f is unbounded. Bounded faces have an outer CCB, and the `outer_ccb()` method returns a circulator of type `Arrangement_2::Ccb_halfedge_circulator` for traversing the halfedges along this CCB. Note that the halfedges along this CCB wind in a *counterclockwise* order around the outer boundary of the face.

A face can also contain disconnected components in its interior, namely, holes and isolated vertices. You can access these components as follows:

- You can obtain a pair of `Arrangement_2::Hole_iterator` iterators that define the range of holes inside a face f by calling `f->holes_begin()` and `f->holes_end()`.

The value type of this iterator is `Arrangement_2::Ccb_halfedge_circulator`, defining the CCB that winds in a *clockwise* order around a hole.

- The calls `f->isolated_vertices_begin()` and `f->isolated_vertices_end()` return `Arrangement_2::Isolated_vertex_iterator` iterators that define the range of isolated vertices inside the face f . The value type of this iterator is `Arrangement_2::Vertex`.



Example: The function template `print_face()` listed below prints the outer and inner boundaries of a given face. It uses the function template `print_ccb()` listed above.

```
template <typename Arrangement>
void print_face(typename Arrangement::Face_const_handle f)
{
    // Print the outer boundary.
```

```

    if (f->is_unbounded()) std::cout << "Unbounded_face._" << std::endl;
    else {
        std::cout << "Outer_boundary:_" ;
        print_ccb<Arrangement>(f->outer_ccb());
    }

    // Print the boundary of each of the holes.
    int index = 1;
    typename Arrangement::Hole_const_iterator hole;
    for (hole = f->holes_begin(); hole != f->holes_end(); ++hole, ++index) {
        std::cout << "Hole_##" << index << ":_";
        print_ccb<Arrangement>(*hole);
    }

    // Print the isolated vertices.
    typename Arrangement::Isolated_vertex_const_iterator iv;
    for (iv = f->isolated_vertices_begin(), index = 1;
         iv != f->isolated_vertices_end(); ++iv, ++index)
        std::cout << "Isolated_vertex_##" << index << ":_";
        << "(" << iv->point() << ")" << std::endl;
}

```



Example: The function template `print_arrangement()` listed below prints the features of a given arrangement. The file `arr_print.h`, includes the definitions of this function, as well as the definitions of all other functions listed in this section. This concludes the preview of the various traversal methods.

```

template <typename Arrangement>
void print_arrangement(const Arrangement& arr)
{
    CGAL_precondition(arr.is_valid());

    // Print the arrangement vertices.
    typename Arrangement::Vertex_const_iterator vit;
    std::cout << arr.number_of_vertices() << "_vertices:" << std::endl;
    for (vit = arr.vertices_begin(); vit != arr.vertices_end(); ++vit) {
        std::cout << "(" << vit->point() << ")";
        if (vit->is_isolated()) std::cout << "_Isolated." << std::endl;
        else std::cout << "_degree_" << vit->degree() << std::endl;
    }

    // Print the arrangement edges.
    typename Arrangement::Edge_const_iterator eit;
    std::cout << arr.number_of_edges() << "_edges:" << std::endl;
    for (eit = arr.edges_begin(); eit != arr.edges_end(); ++eit)
        std::cout << "[" << eit->curve() << "]" << std::endl;

    // Print the arrangement faces.
    typename Arrangement::Face_const_iterator fit;
    std::cout << arr.number_of_faces() << "_faces:" << std::endl;
    for (fit = arr.faces_begin(); fit != arr.faces_end(); ++fit)
        print_face<Arrangement>(fit);
}

```

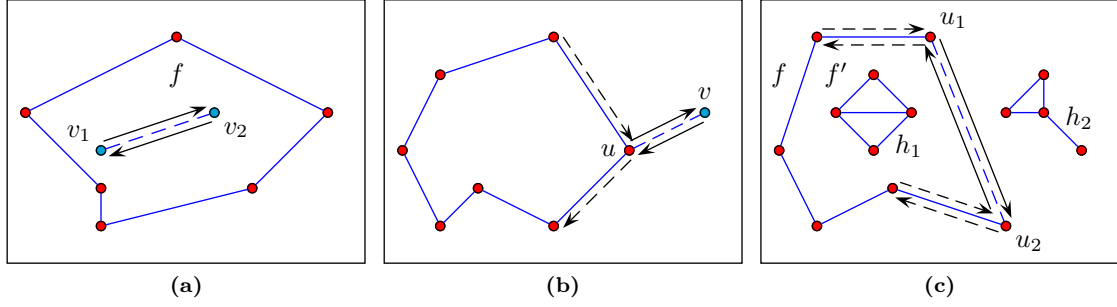


Fig. 2.2: Illustrations of the various specialized insertion procedures. The inserted x -monotone curve is drawn as a dashed line, surrounded by two solid arrows that represent the pair of twin halfedges added to the DCEL. Existing vertices are drawn as dark discs, while new vertices are drawn as light discs. Existing halfedges that are affected by the insertion operations are drawn as dashed arrows. (a) Inserting a curve that induces a new hole inside the face f . (b) Inserting a curve from an existing vertex u that corresponds to one of its endpoints. (c) Inserting an x -monotone curve, the endpoints of which correspond to existing vertices u_1 and u_2 . In this case the new pair of halfedges close a new face f' . The hole h_1 , which belonged to f before the insertion, becomes a hole in this new face.

2.2.2 Modifying the Arrangement

In this section we review the various member functions of the `Arrangement_2` class template that allow you to modify the topological structure of the arrangement through the introduction of new edges or vertices or the modification or removal of existing edges or vertices.

The arrangement member-functions that insert new x -monotone curves into the arrangement, thus enabling the construction of a planar subdivision, are rather specialized, as they assume that the interior of the inserted curve is disjoint from all existing arrangement vertices and edges, and in addition require a priori knowledge of the location of the inserted curve. Indeed, for most purposes it is more convenient to construct an arrangement using the free (global) insertion functions, which relax these restrictions. However, as these free functions are implemented in terms of the specialized insertion functions, we start by describing the fundamental functionality of the arrangement class, and describe the operation of the free functions in Chapter 3.

Inserting Non-Intersecting x -Monotone Curves

The most trivial functions that allow you to modify the arrangement are the specialized functions for the insertion of an x -monotone curve the interior of which is disjoint from the interior of all other curves in the existing arrangement and does not contain any point of the arrangement. In addition, these functions require that the location of the curve in the arrangement be known.

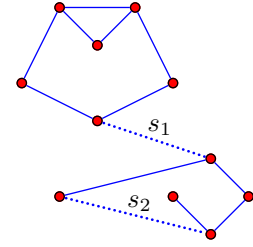
The rather harsh restrictions on the inserted curves enable an efficient implementation. While inserting an x -monotone curve, the interior of which is disjoint from all curves in the existing arrangement, is quite straightforward, as we show next, (efficiently) inserting a curve that intersects with the curves already in the arrangement is much more complicated and requires the application of nontrivial geometric algorithms. The decoupling of the topological arrangement representation from the various algorithms that operate on it dictates that the general insertion operations be implemented as free functions that operate on the arrangement and the inserted curve(s); see Section 3.4 for more details and examples.

When an x -monotone curve is inserted into an existing arrangement, such that the interior of this curve is disjoint from the interior of all curves in the arrangement, only the following three scenarios are possible, depending on the status of the endpoints of the inserted curve:

1. If both curve endpoints do not correspond to any existing arrangement vertex we have to create two new vertices, corresponding to the curve endpoints, and connect them using a pair of twin halfedges. This halfedge pair forms a new hole inside the face that contains the

curve in its interior; see Figure 2.2a for an illustration.

2. If exactly one endpoint corresponds to an existing arrangement vertex (we distinguish between a vertex that corresponds to the left endpoint of the inserted curve and one that corresponds to its right endpoint), we have to create a new vertex that corresponds to the other endpoint of the curve and to connect the two vertices by a pair of twin halfedges that form an “antenna” emanating from the boundary of an existing connected component; see Figure 2.2b. (Note that if the existing vertex used to be isolated, this operation is actually equivalent to forming a new hole inside the face that contains this vertex.)
3. If both endpoints correspond to existing arrangement vertices, we connect these vertices using a pair of twin halfedges. (If one or both vertices are isolated, this case reduces to case (2) or case (1), respectively.) The two following subcases may occur:
 - Two disconnected components are merged into a single connected component (as is the case with the segment s_1 in the figure to the right).
 - A new face is created, which is split from an existing arrangement face. In this case we also have to examine the holes and isolated vertices in the existing face and move the relevant ones to belong to the new face (as is the case with the segment s_2 in the figure to the right); see also Figure 2.2c.



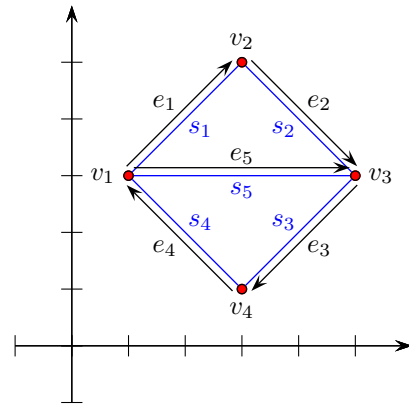
The `Arrangement_2` class template offers insertion functions that perform the special insertion procedures listed above, namely `insert_in_face_interior()`, `insert_from_left_vertex()`, `insert_from_right_vertex()`, and `insert_at_vertices()`. The first function accepts an x -monotone curve c and a handle to an arrangement face f that contains this curve in its interior. The other functions accept an x -monotone curve c and handles to the existing vertices that correspond to the curve endpoint(s). Each of the four functions returns a handle to one of the twin halfedges that have been created; more precisely:

- `insert_in_face_interior(c, f)` returns a handle to the halfedge directed from the vertex corresponding to the left endpoint of c towards the vertex corresponding to its right endpoint.
- `insert_from_left_vertex(c, v)` and `insert_from_right_vertex(c, v)` each returns a handle to the halfedge, the source of which is the vertex v , and the target of which is the new vertex that has just been created.
- `insert_at_vertices(c, v1, v2)` returns a handle to the halfedge directed from v_1 to v_2 .



Example: The program below demonstrates the usage of the four specialized insertion functions. It creates an arrangement of five line segments s_1, \dots, s_5 , as depicted in the figure to the right.⁵ The first line segment s_1 is inserted in the interior of the unbounded face, while the four succeeding line segments s_2, \dots, s_5 are inserted using the vertices created by the insertion of preceding segments. The arrows in the figure mark the direction of the halfedges e_1, \dots, e_5 returned from the insertion functions, to make it easier to follow the flow of the program. The resulting arrangement consists of three faces, where the two bounded faces form together a hole in the unbounded face.

Two header files are included in the code, in order to make this and the following examples more compact. The



⁵Notice that in all figures in this book the coordinate axes are drawn only for illustrative purposes and are *not* part of the arrangement.

file `arr_inexact_construction_segments.h` is listed immediately after the program. The file `arr_print.h` is introduced in Section 2.2.1.

```
// File: ex_edge_insertion.cpp
```

```
#include "arr_inexact_construction_segments.h"
#include "arr_print.h"

int main()
{
    Point          p1(1, 3), p2(3, 5), p3(5, 3), p4(3, 1);
    Segment        s1(p1, p2), s2(p2, p3), s3(p3, p4), s4(p4, p1), s5(p1, p3);

    Arrangement    arr;
    Halfedge_handle e1 = arr.insert_in_face_interior(s1, arr.unbounded_face());
    Vertex_handle   v1 = e1->source();
    Vertex_handle   v2 = e1->target();
    Halfedge_handle e2 = arr.insert_from_left_vertex(s2, v2);
    Vertex_handle   v3 = e2->target();
    Halfedge_handle e3 = arr.insert_from_right_vertex(s3, v3);
    Vertex_handle   v4 = e3->target();
    Halfedge_handle e4 = arr.insert_at_vertices(s4, v4, v1);
    Halfedge_handle e5 = arr.insert_at_vertices(s5, v1, v3);

    print_arrangement(arr);
    return 0;
}
```

As mentioned above, in all examples listed in this chapter and some of the examples listed in the following chapter the `Traits` parameter of the `Arrangement_2<Traits, Dcel>` class template is substituted with an instance of the `Arr_segment_traits_2<Kernel>` class template. In these examples the `Arr_segment_traits_2` class template is instantiated with the predefined CGAL kernel that evaluates predicates in an exact manner, but constructs geometric objects in an inexact manner, as none of these examples construct new geometric objects. In the remaining examples listed in the next chapter, as well as in most other examples listed in the book, the traits class-template is instantiated with a kernel that evaluates predicates and constructs geometric objects, both in an exact manner; see Section 1.4.4 for more details about the various kernels. The statements below define the types for arrangements of line segments common to all examples that do not construct new geometric objects. They are kept in the header file `arr_inexact_construction_segments.h`.

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Arr_non_caching_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>

typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;
typedef Kernel::FT Number_type;

typedef CGAL::Arr_non_caching_segment_traits_2<Kernel> Traits;
typedef Traits::Point_2 Point;
typedef Traits::X_monotone_curve_2 Segment;

typedef CGAL::Arrangement_2<Traits> Arrangement;
typedef Arrangement::Vertex_handle Vertex_handle;
typedef Arrangement::Halfedge_handle Halfedge_handle;
typedef Arrangement::Face_handle Face_handle;
```

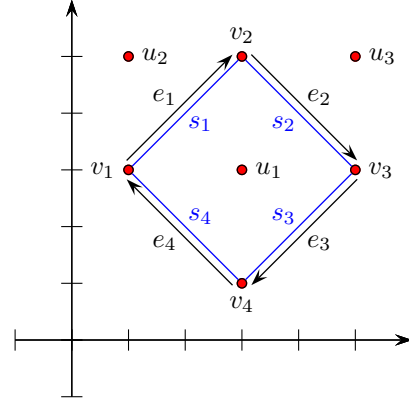
Manipulating Isolated Vertices

Isolated points are in general simpler geometric entities than curves, and indeed the member functions that manipulate them are easier to understand.

The call `arr.insert_in_face_interior(p, f)` inserts an isolated point p , located in the interior of a given face f , into the arrangement and returns a handle to the arrangement vertex associated with p it has created. Naturally, this function has the precondition that p is an isolated point; namely, it does not coincide with any existing arrangement vertex and does not lie on any edge. As mentioned in Section 2.2.1, it is possible to obtain the face containing an isolated vertex v by calling `v->face()`. The member function `remove_isolated_vertex(v)` accepts a handle to an isolated vertex v as input and removes it from the arrangement.



Example: The program below demonstrates the usage of the arrangement member-functions for manipulating isolated vertices. It first inserts three isolated vertices, u_1 , u_2 , and u_3 , located inside the unbounded face of the arrangement. Then it inserts four line segments, s_1, \dots, s_4 , that form a square hole inside the unbounded face; see the figure above for an illustration. Finally, it traverses the vertices and removes those isolated vertices that are still contained in the unbounded face (u_2 and u_3 in this case).



// File: ex_isolated_vertices.cpp

```
#include "arr_inexact_construction_segments.h"
#include "arr_print.h"
```

```
int main()
{
    // Insert isolated points.
    Arrangement      arr;
    Face_handle      uf = arr.unbounded_face();
    Vertex_handle    u1 = arr.insert_in_face_interior(Point(3, 3), uf);
    Vertex_handle    u2 = arr.insert_in_face_interior(Point(1, 5), uf);
    Vertex_handle    u3 = arr.insert_in_face_interior(Point(5, 5), uf);

    // Insert four segments that form a square-shaped face.
    Point            p1(1, 3), p2(3, 5), p3(5, 3), p4(3, 1);
    Segment          s1(p1, p2), s2(p2, p3), s3(p3, p4), s4(p4, p1);

    Halfedge_handle  e1 = arr.insert_in_face_interior(s1, uf);
    Vertex_handle    v1 = e1->source();
    Vertex_handle    v2 = e1->target();
    Halfedge_handle  e2 = arr.insert_from_left_vertex(s2, v2);
    Vertex_handle    v3 = e2->target();
    Halfedge_handle  e3 = arr.insert_from_right_vertex(s3, v3);
    Vertex_handle    v4 = e3->target();
    Halfedge_handle  e4 = arr.insert_at_vertices(s4, v4, v1);

    // Remove the isolated vertices located in the unbounded face.
    Arrangement::Vertex_iterator curr, next = arr.vertices_begin();
    for (curr = next++; curr != arr.vertices_end(); curr = next++) {
        // Keep an iterator to the next vertex, as curr might be deleted.

```

```

    if (curr->is_isolated() && curr->face() == uf)
        arr.remove_isolated_vertex(curr);
}

print_arrangement(arr);
return 0;
}

```

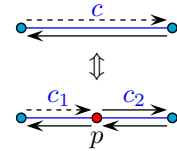


Try: A more efficient way to remove the isolated vertices that lie inside the unbounded face is to obtain the unbounded face, and then traverse its isolated vertices, removing them as they are visited. Replace the code in the program above that traverses all vertices and removes the isolated ones with code that obtains the unbounded face and then traverses its isolated vertices, removing them as they are visited.

Manipulating Halfedges

While reading the previous subsection you learned how to insert new points that induce isolated vertices into the arrangement. You may wonder now how you can insert a new point that lies on an x -monotone curve that is associated with an existing arrangement edge.

The introduction of a vertex, the geometric mapping of which is a point p that lies on an x -monotone curve, requires the splitting of the curve in its interior at p . The two resulting subcurves induce two new edges, respectively. In general, the `Arrangement_2` class template relies on the geometry traits to perform such a split. As a matter of fact, it relies on the geometry traits to perform all geometric operations. To insert a point p that lies on an x -monotone curve associated with an existing edge e into the arrangement \mathcal{A} , you must first construct the two curves c_1 and c_2 , which are the two subcurves that result from splitting the x -monotone curve associated with the edge e at p . Then, you have to issue the call `arr.split_edge(he, c1, c2)`, where `arr` identifies the arrangement \mathcal{A} and `he` is a handle to one of the two halfedges that represent the edge e . The function splits the two halfedges that represent e into two pairs of halfedges, respectively. Two new halfedges are incident to the new vertex v associated with p . The function returns a handle to the new halfedge, the source of which is the source vertex of the halfedge handled by `he`, and the target of which is the new vertex v . For example, if the halfedge drawn as a dashed line at the top in the figure above is passed as input, the halfedge drawn as a dashed line at the bottom is returned as output.



The reverse operation is also possible. Consider a vertex v of degree 2 that has two incident edges e_1 and e_2 associated with two curves c_1 and c_2 , respectively, such that the union of c_1 and c_2 results in a single continuous x -monotone curve c of the type supported by the traits class in use. To merge the edges e_1 and e_2 into a single edge associated with the curve c , essentially removing the vertex v from the arrangement identified by `arr`, you need to issue the call `arr.merge_edge(he1, he2, c)`, where `he1` and `he2` are handles to halfedges representing e_1 and e_2 , respectively.

Finally, the call `arr.remove_edge(he)` removes the edge e from the arrangement, where `he` is a handle to one of the two halfedges that represents e . Note that this operation is the reverse of an insertion operation, so it may cause a connected component to split into two, or two faces to merge into one, or a hole to disappear. By default, if the removal of e causes one of its end vertices to become isolated, this vertex is removed as well. However, you can control this behavior and choose to keep the isolated vertices by supplying additional Boolean flags to `remove_edge()` indicating whether the source or the target vertices are to be removed should they become isolated.



Example: The example program below shows how the edge-manipulation functions can be used. The program works in three steps, as demonstrated in Figure 2.3. Note that the program uses the fact that `split_edge()` returns one of the new halfedges (after the split) that has the same direction as the original halfedge (the first parameter of the function) and is directed towards the split point. Thus, it is easy to identify the vertices u_1 and u_2 associated with the split points.

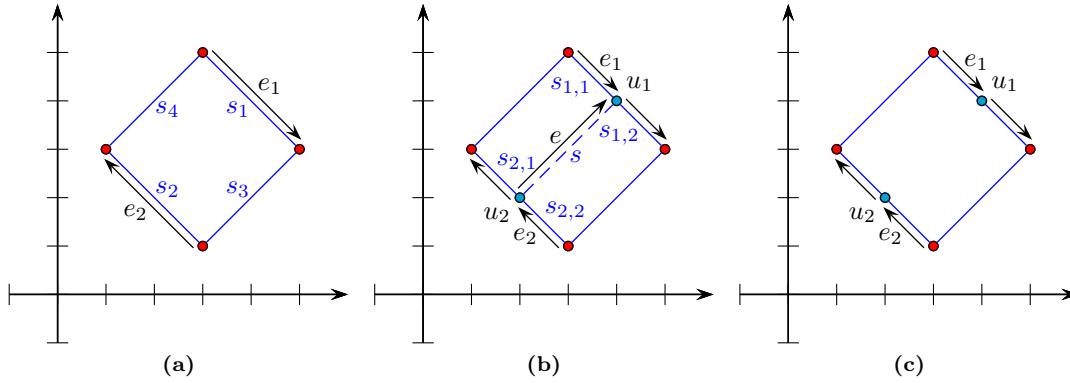


Fig. 2.3: The three steps of the example program `ex_edge_manipulation.cpp`. In Step (a) it constructs an arrangement of four line segments. In Step (b) the edges e_1 and e_2 are split, and the split points are connected with a new segment s that is inserted into the arrangement. This operation is undone in Step (c), where e is removed from the arrangement, rendering its end vertices u_1 and u_2 redundant. We therefore remove these vertices by merging their incident edges and go back to the arrangement depicted in (a).

```
// File: ex_edge_manipulation.cpp

#include "arr_inexact_construction_segments.h"
#include "arr_print.h"

int main()
{
    // Step (a) - construct a rectangular face.
    Point      q1(1, 3), q2(3, 5), q3(5, 3), q4(3, 1);
    Segment    s4(q1, q2), s1(q2, q3), s3(q3, q4), s2(q4, q1);

    Arrangement    arr;
    Halfedge_handle e1 = arr.insert_in_face_interior(s1, arr.unbounded_face());
    Halfedge_handle e2 = arr.insert_in_face_interior(s2, arr.unbounded_face());

    e2 = e2->twin();    // as we wish e2 to be directed from right to left
    arr.insert_at_vertices(s3, e1->target(), e2->source());
    arr.insert_at_vertices(s4, e2->target(), e1->source());
    std::cout << "After_step_(a):" << std::endl;
    print_arrangement(arr);

    // Step (b) - split e1 and e2 and connect the split points with a segment.
    Point      p1(4,4), p2(2,2);
    Segment    s1_1(q2, p1), s1_2(p1, q3), s2_1(q4, p2), s2_2(p2, q1), s(p1, p2);

    e1 = arr.split_edge(e1, s1_1, s1_2);
    e2 = arr.split_edge(e2, s2_1, s2_2);
    Halfedge_handle e = arr.insert_at_vertices(s, e1->target(), e2->target());
    std::cout << std::endl << "After_step_(b):" << std::endl;
    print_arrangement(arr);

    // Step (c) - remove the edge e and merge e1 and e2 with their successors.
    arr.remove_edge(e);
}
```

```

arr.merge_edge(e1, e1->next(), s1);
arr.merge_edge(e2, e2->next(), s2);
std::cout << std::endl << "After_step_(c):" << std::endl;
print_arrangement(arr);
return 0;
}

```

— advanced —

The member functions `modify_vertex()` and `modify_edge()` modify the geometric mappings of existing features of the arrangement. The call `arr.modify_vertex(v, p)` accepts a handle to a vertex v and a reference to a point p , and sets p to be the point associated with the vertex v . The call `arr.modify_edge(he, c)` accepts a handle to one of the two halfedges that represent an edge e and a reference to a curve c , and sets c to be the x -monotone curve associated with e . (Note that both halfedges are modified; that is, both expressions `he->curve()` and `he->twin()->curve()` evaluate to c after the modification.) These functions have preconditions that p is geometrically equivalent to `v->point()` and c is equivalent to `e->curve()`, respectively.⁶ If these preconditions are not met, the corresponding operation may invalidate the structure of the arrangement. At first glance it may seem as if these two functions are of little use. However, you should keep in mind that there may be extraneous data (probably non-geometric) associated with the point objects or with the curve objects, as defined by the traits class. With these two functions you can modify this data; see more details in Section 5.5.

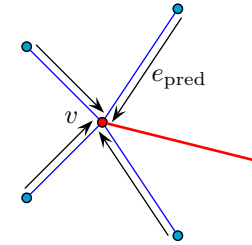
In addition, you can use these functions to replace a geometric object (a point or a curve) with an equivalent object that has a more compact representation. For example, if we use some simple rational-number type to represent the point coordinates, we can replace the point $(\frac{20}{40}, \frac{99}{33})$ associated with some vertex v with an equivalent point with normalized coordinates, namely $(\frac{1}{2}, 3)$.

— advanced —

— advanced —

Advanced Insertion Functions

Assume that the specialized insertion function `insert_from_left_vertex(c, v)` is given a curve c , the left endpoint of which is already associated with a non-isolated vertex v . Namely, v has already several incident halfedges. It is necessary in this case to locate the exact place for the new halfedge mapped to the newly inserted curve c in the circular list of halfedges incident to v . More precisely, in order to complete the insertion, it is necessary to locate the halfedge e_{pred} directed towards v such that c is located between the curves associated with e_{pred} and the next halfedge in the clockwise order in the circular list of halfedges around v ; see the figure to the right. This may take $O(d)$ time, where d is the degree of the vertex v .⁷ However, if the halfedge e_{pred} is known in advance, the insertion can be carried out in constant time, and without performing any geometric comparisons.



The `Arrangement_2` class template provides advanced versions of the specialized insertion functions for a curve c , namely `insert_from_left_vertex(c, he_pred)` and `insert_from_right_vertex(c, he_pred)`. These functions accept a handle to the halfedge e_{pred} as specified above, instead of a handle to the vertex v . They are more efficient, as they take constant time and do not perform any geometric operations. Thus, you should use them when the halfedge e_{pred} is

⁶Roughly speaking, two curves are equivalent iff they have the same graph. In Section 5.1.1 we give a formal definition of curves and curve equivalence.

⁷We can store the handles to the halfedges incident to v in an efficient search structure to obtain $O(\log d)$ access time. However, as d is usually very small, this may lead to a waste of storage space without a meaningful improvement in running time in practice.

known. In cases where the vertex v is isolated or the predecessor halfedge for the newly inserted curve is not known, the simpler versions of these insertion functions should be used. Similarly, the member function `insert_at_vertices()` is overloaded with two additional versions as follows. One accepts two handles to the two predecessor halfedges around the two vertices v_1 and v_2 that correspond to the curve endpoints. The other one accepts a handle to one vertex and a handle to the predecessor halfedge around the other vertex.



Example: The program below shows how to construct an arrangement of eight pairwise interior-disjoint line-segments s_1, \dots, s_8 , as depicted in the figure to the right, using the specialized insertion functions that accept predecessor halfedges. The corresponding halfedges e_1, \dots, e_8 are drawn as arrows. Note that the point p_0 is initially inserted as an isolated point and later on is connected to the other four vertices to form the four bounded faces of the final arrangement.

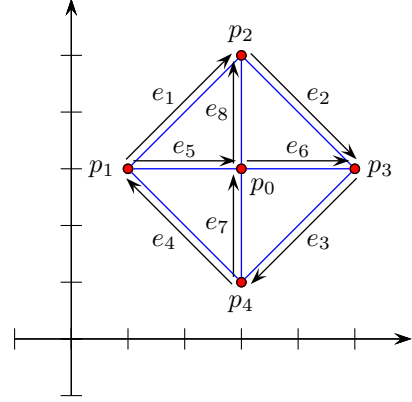
// File: *ex_special_edge_insertion.cpp*

```
#include "arr_inexact_construction_segments.h"
#include "arr_print.h"
```

```
int main()
{
    Point          p0(3, 3), p1(1, 3), p2(3, 5), p3(5, 3), p4(3, 1);
    Segment        s1(p1, p2), s2(p2, p3), s3(p3, p4), s4(p4, p1);
    Segment        s5(p1, p0), s6(p0, p3), s7(p4, p0), s8(p0, p2);

    Arrangement    arr;
    Vertex_handle  v0 = arr.insert_in_face_interior(p0, arr.unbounded_face());
    Halfedge_handle e1 = arr.insert_in_face_interior(s1, arr.unbounded_face());
    Halfedge_handle e2 = arr.insert_from_left_vertex(s2, e1);
    Halfedge_handle e3 = arr.insert_from_right_vertex(s3, e2);
    Halfedge_handle e4 = arr.insert_at_vertices(s4, e3, e1->twin());
    Halfedge_handle e5 = arr.insert_at_vertices(s5, e1->twin(), v0);
    Halfedge_handle e6 = arr.insert_at_vertices(s6, e5, e3->twin());
    Halfedge_handle e7 = arr.insert_at_vertices(s7, e4->twin(), e6->twin());
    Halfedge_handle e8 = arr.insert_at_vertices(s8, e5, e2->twin());

    print_arrangement(arr);
    return 0;
}
```



It is possible to perform even more refined operations on an `Arrangement_2` object given specific topological information. As most of these operations are very fragile and do not test preconditions on their input in order to gain efficiency, they are not included in the public interface of the `Arrangement_2` class template. Instead, the `Arr_accessor<Arrangement>` class template enables access to these internal arrangement operations; see more details in the Reference Manual.

_____ *advanced* _____

2.2.3 Input/Output Functions

In some cases, you may like to save an arrangement object constructed by some application, so that later on it can be restored. In other cases you may like to create nice drawings that represent arrangements constructed by some application. These drawings can be printed or displayed on a

computer screen and dynamically change as the arrangement itself changes.

Input/Output Stream

Consider an arrangement that represents a very complicated geographical map, and assume that there are applications that need to answer various queries on this map. Naturally, you can store the set of curves that induce the arrangement, but this implies that you would need to construct the arrangement from scratch each time you wish to reuse it. A more efficient solution is to write the arrangement to a file in a format that other applications can read.

The *2D Arrangements* package provides an *inserter* operator (<<), which inserts an arrangement object into an output stream, and an *extractor* operator (>>), which extracts an arrangement object from an input stream. The arrangement is written using a simple predefined plain-text format that encodes the arrangement topology, as well as all geometric entities associated with vertices and edges.

The ability to use the input and output operators requires that the **Point_2** type and the **X_monotone_curve_2** type defined by the traits class both support the << and >> operators. Only traits classes that handle linear objects are guaranteed to provide these operators for the geometric types they define. Thus, you can safely write and read arrangements of line segments, polylines, or unbounded linear objects.⁸



Example: The example below constructs the arrangement depicted on Page 27 and writes it to an output file. It also demonstrates how to reread the arrangement from a file.

```
// File: ex_io.cpp

#include <fstream>

#include <CGAL/basic.h>
#include <CGAL/IO/Arr_iostream.h>

#include "arr_inexact_construction_segments.h"
#include "arr_print.h"

int main()
{
    // Construct the arrangement.
    Point          p1(1, 3), p2(3, 5), p3(5, 3), p4(3, 1);
    Segment        s1(p1, p2), s2(p2, p3), s3(p3, p4), s4(p4, p1), s5(p1, p3);

    Arrangement    arr1;
    Halfedge_handle e1 = arr1.insert_in_face_interior(s1, arr1.unbounded_face());
    Vertex_handle   v1 = e1->source();
    Vertex_handle   v2 = e1->target();
    Halfedge_handle e2 = arr1.insert_from_left_vertex(s2, v2);
    Vertex_handle   v3 = e2->target();
    Halfedge_handle e3 = arr1.insert_from_right_vertex(s3, v3);
    Vertex_handle   v4 = e3->target();
    Halfedge_handle e4 = arr1.insert_at_vertices(s4, v4, v1);
    Halfedge_handle e5 = arr1.insert_at_vertices(s5, v1, v3);

    // Write the arrangement to a file.
    std::cout << "Writing" << std::endl;
    print_arrangement_size(arr1);
}
```

⁸Traits classes that handle non-linear objects use algebraic-number types. The inserter (<<) and extractor (>>) operators for these non-linear objects can be provided only if these operators are available for algebraic numbers.


```

std::ofstream out_file("arr_ex_io.dat");
out_file << arr1;
out_file.close();

// Read the arrangement from the file.
Arrangement arr2;
std::ifstream in_file("arr_ex_io.dat");
in_file >> arr2;
in_file.close();
std::cout << "Reading" << std::endl;
print_arrangement_size(arr2);

return 0;
}

```

The inserter and extractor operators utilize the free functions `write()` and `read()`. These functions use a *formatter* object, which defines the I/O format. Both `read()` and `write()` functions use the `Arr_text_formatter` formatter class, which ignores auxiliary data that might be attached to the arrangement features. If you wish to write or read arrangements extended with auxiliary data, use other plain-text formats, such as Postscript, XML, and IPE,⁹ or even use binary formats, you must call the `read()` and `write()` functions and pass an appropriate formatter. Section 6.2 describes how you can write or read an arrangement with auxiliary data stored with its features. The *2D Arrangements* package also comes with formatters that write and read arrangements that maintain cross-mappings between input curves and the arrangement edges they induce, referred to as arrangements-with-history objects; see Section 6.4 for details.

Output Qt-Widget Stream

The *2D Arrangements* package includes an interactive program that demonstrates its features. As mentioned in the preface of this book, this demonstration program is still based on QT 3, an older version of QT, and like many other CGAL programs based on QT 3, it uses a QT stream called `Qt_widget`. You can display the drawings of arrangements in a graphical window using `Qt_widget` streams just like the demonstration program does. All you need to do is follow the guidelines for handling `Qt_widget` objects, and apply the *inserter*, which inserts an arrangement into a `Qt_widget` stream to complete the drawing. The ability to use this output operator requires that the `Point_2` and `X_monotone_curve_2` types defined by the traits class both support the inserter (`<<`) operator that inserts the respective geometric object into a `Qt_widget` stream. The `Arr_rational_arc_traits_2` class template (see Section 5.4.3) and the `Arr_linear_traits_2` class template (see Section 5.2) currently do not provide this operator for the geometric types they define. Thus, only arrangements of line segments, polylines, or conic arcs can be drawn this way without additional code. The `<<` operator for polylines and conic arcs is defined in `CGAL/IO/Qt_widget_Polyline_2.h` and `CGAL/IO/Qt_widget_Conic_arc_2.h`, respectively. These files must be explicitly included to insert polylines or conic arcs into `Qt_widget` streams.

All CGAL programs based on QT Version 3, including the arrangement demonstration-program, were being ported to Version 4, the latest version of QT, at the time this book was written. In the new setup the visualization of two-dimensional CGAL objects is done with the QT Graphics View Framework. This framework enables managing and interacting with a large number of custom-made 2D graphical items, and provides a view widget for visualizing the items, with support for zooming and rotation.¹⁰

⁹<http://tclab.kaist.ac.kr/ipe/>.

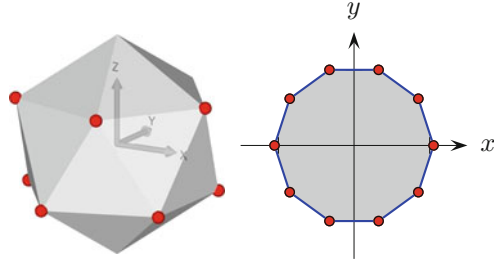
¹⁰See <http://doc.qt.nokia.com/latest/graphicsview.html> for more information on this framework. The package “CGAL and the QT Graphics View Framework” [66] provides the means to integrate CGAL and the Graphics View Framework.

2.3 Application: Obtaining Silhouettes of Polytopes

We conclude this chapter with a small application that obtains the silhouettes of bounded convex polyhedra in \mathbb{R}^3 , commonly referred to as convex *polytopes*. Given a convex polytope P , the program obtains the outline of the shadow of P cast on the xy -plane, where the scene is illuminated by a light source at infinity directed along the negative z -axis. The silhouette is represented as an arrangement that contains two faces—an unbounded face and a single hole inside the unbounded face. The silhouette is the outer boundary of the latter.

The figure to the right shows an icosahedron and its silhouette. The corresponding input file located in the example folder is called `icosahedron.dat`.

The program first constructs the input convex polytope and stores the result in a temporary object, the type of which is an instance of the CGAL `Polyhedron_3` class template. Then, it traverses the convex polytope facets. For each facet facing upwards (namely, whose outer normal has a positive z component), it traverses the edges on the facet boundary. The traversal of the facets and the traversal of the halfedges around each facet are done in a way similar to the traversals of arrangement cells as described in Section 2.2.1; see the Reference Manual for the exact interface of the `Polyhedron_3` class template. (Notice that the positive normal z -coordinate requirement rules out faces that are parallel to the z -axis.) Let e be the current polytope edge being processed, and let e' be the vertical projection of e (onto the xy -plane). The program inserts e' into the arrangement using one of the specialized insertion member-functions listed in Section 2.2.2. Once the traversal of the temporary polytope is completed, it is discarded. Finally, the program traverses the arrangement edges, and removes all edges that are not incident to the unbounded face.



We must ensure that each edge is inserted into the arrangement exactly once to avoid overlaps. To this end, we maintain a set E of handles to polytope edges. It contains the edges of the polytope, the projections of which have already been inserted into the arrangement. For each edge e being processed, if e is not in E , we insert the projection of e into the arrangement and we also insert e into E . We use the `std::set` data structure to maintain the set E . It requires the provision of a model of the `StrictWeakOrdering` STL concept that compares handles. We use the functor `Less_than_handle` listed below, and defined in the header file `Less_than_handle.h`. The functor compares the addresses of the handled objects. Thus, (different) handles to the same object are evaluated as equal. (The induced ordering is consistent, but arbitrary and irrelevant.)

```
struct Less_than_handle {
    template <typename Type>
    bool operator()(Type s1, Type s2) const { return (&(*s1) < &(*s2)); }
};
```

— advanced —

An alternative technique to avoid duplicate insertion is to extend each record of the polyhedron data structure that represents a halfedge with a Boolean value that indicates whether it has been inserted into the arrangement. This technique is more efficient, as it does not use the set auxiliary data structure. The technique used to extend the polyhedron data structure is similar to the technique used to extend the arrangement data structure, as both are based on the same halfedge data structure. The extending technique is discussed only in Chapter 6.

— advanced —

We must determine the appropriate insertion routines to insert the segments into the arrangement. To this end, we use yet another auxiliary data structure, namely a map, which maps polyhedron vertices to corresponding arrangement vertices. Before inserting a segment into the arrangement we search the map for the arrangement vertices that correspond to the segment endpoints, and dispatch the appropriate insertion routine.

————— *advanced* —————

An alternative technique is to extend each record of the polyhedron data structure that represents a vertex with the handle to the corresponding arrangement vertex. This field is initialized when the polyhedron vertex is processed for the first time, and is used during subsequent encounters with the vertex. Again, this technique is more efficient, as it does not use the map auxiliary data structure, but also more advanced. Moreover, the processing can be expedited further using the more efficient insertion methods that accept halfedges as operands.

————— *advanced* —————

The functor template **Arr_inserter**<**Polyhedron**, **Arrangement**> listed below, and defined in the header file **Arr_inserter.h**, is used to insert a segment, which is the projection of a polytope edge, into the arrangement. When it is instantiated its template parameters **Polyhedron** and **Arrangement** must be substituted with instances of the polyhedron and arrangement types, respectively. Its function operator accepts four parameters as follows: the target arrangement object, the polytope edge, and the two points that are the projection of the endpoints of the polytope edge onto the *xy*-plane.

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

#include "Less_than_handle.h"

template <typename Polyhedron, typename Arrangement> class Arr_inserter {
private:
    typedef typename Polyhedron::Halfedge_const_handle
        Polyhedron_halfedge_const_handle;
    typedef std::map<typename Polyhedron::Vertex_const_handle,
        typename Arrangement::Vertex_handle, Less_than_handle>
        Vertex_map;
    typedef typename Vertex_map::iterator
        Vertex_map_iterator;
    typedef typename Arrangement::Point_2
        Point_2;
    typedef typename Arrangement::X_monotone_curve_2
        X_monotone_curve_2;

    Vertex_map m_vertex_map;
    std::set<Polyhedron_halfedge_const_handle, Less_than_handle> m_edges;
    const typename Arrangement::Traits_2::Compare_xy_2 m_cmp_xy;
    const typename Arrangement::Traits_2::Equal_2 m_equal;

public:
    Arr_inserter(const typename Arrangement::Traits_2& traits) :
        m_cmp_xy(traits.compare_xy_2_object()),
        m_equal(traits.equal_2_object())
    {}

    void operator()(Arrangement& arr, Polyhedron_halfedge_const_handle he,
        Point_2& prev_arr_point, Point_2& arr_point)
    {
        // Avoid the insertion if he or its twin have already been inserted.
        if ((m_edges.find(he) != m_edges.end()) ||
            (m_edges.find(he->opposite()) != m_edges.end()))
            return;

        // Locate the arrangement vertices, which correspond to the projected
        // polyhedron vertices, and insert the segment corresponding to the
        // projected-polyhedron edge using the proper specialized insertion
    }
```

```

// function.
m_edges.insert(he);
X_monotone_curve_2 curve(prev_arr_point, arr_point);
Vertex_map_iterator it1 = m_vertex_map.find(he->opposite()->vertex());
Vertex_map_iterator it2 = m_vertex_map.find(he->vertex());
if (it1 != m_vertex_map.end()) {
    if (it2 != m_vertex_map.end())
        arr.insert_at_vertices(curve, (*it1).second, (*it2).second);
    else {
        typename Arrangement::Halfedge_handle arr_he =
            (m_cmp_xy(prev_arr_point, arr_point) == CGAL::SMALLER) ?
            arr.insert_from_left_vertex(curve, (*it1).second) :
            arr.insert_from_right_vertex(curve, (*it1).second);
        m_vertex_map[he->vertex()] = arr_he->target(); // map the new vertex
    }
}
else if (it2 != m_vertex_map.end()) {
    typename Arrangement::Halfedge_handle arr_he =
        (m_cmp_xy(prev_arr_point, arr_point) == CGAL::LARGER) ?
        arr.insert_from_left_vertex(curve, (*it2).second) :
        arr.insert_from_right_vertex(curve, (*it2).second);
    // map the new vertex.
    m_vertex_map[he->opposite()->vertex()] = arr_he->target();
}
else {
    typename Arrangement::Halfedge_handle arr_he =
        arr.insert_in_face_interior(curve, arr.unbounded_face());
    // map the new vertices.
    if (m_equal(prev_arr_point, arr_he->source()->point())) {
        m_vertex_map[he->opposite()->vertex()] = arr_he->source();
        m_vertex_map[he->vertex()] = arr_he->target();
    } else {
        m_vertex_map[he->opposite()->vertex()] = arr_he->target();
        m_vertex_map[he->vertex()] = arr_he->source();
    }
}
};

```

As we are interested in focusing on the arrangement construction and manipulation, and not on the polytope construction, we simplify the code that constructs the polytope, perhaps at the account of its performance. The program reads only the boundary points of the input polytope from a given input file and computes their convex hull instead of parsing an input file that contains a complete representation of the input polytope as a polyhedral mesh, for example.¹¹ In Chapter 8 we introduce a similar program that obtains the silhouettes of bounded polyhedra, which are not necessarily convex. In this case we are compelled to parse complete representations of the input polyhedra. Regardless of the construction technique, the normals to all facets are computed in a separate loop using the functor template **Normal_equation** listed below, and defined in the header file **Normal_equation.h**. An instance of the functor template is applied to each facet. It computes the cross product of two vectors that correspond to two adjacent edges on the boundary of the input facet, thus obtaining a normal to the facet underlying plane.

```
struct Normal_equation {
```

¹¹A polyhedral mesh representation consists of an array of boundary vertices and the set of boundary facets, where each facet is described by an array of indices into the vertex array.

```

template <typename Facet> typename Facet::Plane_3 operator()(Facet & f) {
    typename Facet::Halfedge_handle h = f.halfedge();
    return CGAL::cross_product(h->next()->vertex()->point() -
                               h->vertex()->point(),
                               h->next()->next()->vertex()->point() -
                               h->next()->vertex()->point());
}
};

```

By default, each record that represents a facet of the polytope is extended with the underlying plane of the facet. The plane is defined with four coefficients. You can override the default and store only the normal to the plane, which is defined by three coordinates instead of the plane four coefficients. In this application we are interested only in the normal to the plane. The listing of the main function follows.

```

// File: ex_polytope_projection.cpp

#include <set>
#include <map>

#include "arr_inexact_construction_segments.h"
#include "read_objects.h"
#include "arr_print.h"
#include "Normal_equation.h"
#include "Arr_inserter.h"

#include <CGAL/convex_hull_3.h>
#include <CGAL/Polyhedron_traits_with_normals_3.h>

typedef CGAL::Polyhedron_traits_with_normals_3<Kernel> Polyhedron_traits;
typedef CGAL::Polyhedron_3<Polyhedron_traits> Polyhedron;
typedef Kernel::Point_3 Point_3;

int main(int argc, char* argv[])
{
    // Read a sequence of 3D points.
    const char* filename = (argc > 1) ? argv[1] : "polytope.dat";
    std::list<Point_3> points;
    read_objects<Point_3>(filename, std::back_inserter(points));

    // Construct the polyhedron.
    Polyhedron polyhedron;
    CGAL::convex_hull_3(points.begin(), points.end(), polyhedron);
    // Compute the normals to all polyhedron facets.
    std::transform(polyhedron.facets_begin(), polyhedron.facets_end(),
                   polyhedron.planes_begin(), Normal_equation());

    // Construct the projection: go over all polyhedron facets.
    Traits traits;
    Arrangement arr(&traits);
    Kernel kernel;
    Kernel::Compare_z_3 cmp_z = kernel.compare_z_3_object();
    Kernel::Construct_translated_point_3 translate =
        kernel.construct_translated_point_3_object();
    Point_3 origin = kernel.construct_point_3_object()(CGAL::ORIGIN);

```

```

Arr_inserter<Polyhedron, Arrangement> arr_inserter(traits);
Polyhedron::Facet_const_iterator it;
for (it = polyhedron.facets_begin(); it != polyhedron.facets_end(); ++it) {
    // Discard facets whose normals have a non-positive z-components.
    if (cmp_z(translate(origin, it->plane()), origin) != CGAL::LARGER)
        continue;

    // Traverse the halfedges along the boundary of the current facet.
    Polyhedron::Halfedge_around_facet_const_circulator hit=it->facet_begin();
    const Point_3& prev_point = hit->vertex()->point();
    Point prev_arr_point = Point(prev_point.x(), prev_point.y());
    for (++hit; hit != it->facet_begin(); ++hit) {
        const Point_3& point = hit->vertex()->point();
        Point arr_point = Point(point.x(), point.y());
        arr_inserter(arr, hit, prev_arr_point, arr_point);
        prev_arr_point = arr_point;
    }
    const Point_3& point = hit->vertex()->point();
    Point arr_point = Point(point.x(), point.y());
    arr_inserter(arr, hit, prev_arr_point, arr_point);
    prev_arr_point = arr_point;
}
polyhedron.clear();

// Remove internal edges.
Face_handle unb_face = arr.unbounded_face();
Arrangement::Edge_iterator eit;
for (eit = arr.edges_begin(); eit != arr.edges_end(); ++eit) {
    Halfedge_handle he = eit;
    if ((he->face() != unb_face) && (he->twin()->face() != unb_face))
        arr.remove_edge(eit);
}

print_arrangement(arr);
return 0;
}

```

The program described above can be optimized, but even in its current state it is relatively efficient due to the good performance of the specialized insertion-functions. In addition, it is simple and, like all other programs presented in this book, it is robust and it produces exact results. It demonstrates well the use of the specialized insertion-functions.

As you might have noticed, the code above contains a call to an instance of a generic function-template called `read_objects()`. It reads the description of geometric objects from a file and constructs them. It accepts the name of an input file that contains the plain-text description of the geometric objects and an output iterator for storing the newly constructed objects. When the function is instantiated, the first template parameter, namely `Type`, must be substituted with the type of objects to read. It is assumed that an extractor operator (`>>`) that extracts objects of the given type from the input stream is available. The listing of the function template, which is defined in the file `read_objects.h`, is omitted here.

2.4 Bibliographic Notes and Remarks

The doubly-connected edge list (DCEL) as we use it in the *2D Arrangements* package enhances the structure described by de Berg et al. [45, Chapter 2]. They, in turn describe a variant of a structure

originally suggested by Muller and Preparata [163]. Many structures to describe two-dimensional subdivisions were proposed over the years. The DCEL we use in particular is an evolution of the halfedge data structure (HDS) designed and implemented by Kettner [129]. This implementation of HDS is provided through the *Halfedge Data Structures* package of CGAL [131], and is directly used by the *3D Polyhedral Surfaces* package of CGAL [130]. We only use limited facilities provided by the *Halfedge Data Structures* package. An overview and comparison of different data structures together with a thorough description of the design of the HDS implemented in CGAL can be found in [129]. These structures are generalized by a topological model, called *combinatorial maps*, which enables the representation of subdivided objects in any fixed dimension; see, e.g., [146].

There is an alternative, very different, way to represent two-dimensional arrangements, via *trapezoidal decomposition* (a.k.a., vertical decomposition). We study this alternative in Section 3.6.

The preliminary design of CGAL's *2D Arrangements* package is reported in [69, 107, 108]. Over the years it has been significantly modified and enhanced. The major innovations in the current design are described in [77, 213].

2.5 Exercises

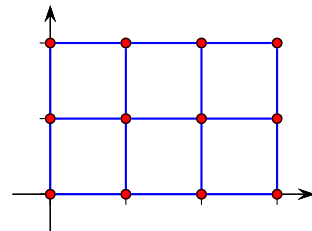
- 2.1 Write a function template that accepts a face (which is not necessarily convex) of an arrangement induced by line segments, and returns a point located inside the face. Use the following prototype:

```
template <typename Arrangement, typename Kernel>
typename Kernel::Point_2
point_in_face(Arrangement::Face_const_handle f,
              const Kernel& ker);
```

with the following preconditions: (i) The traits class used supports line segments, (ii) the types `Arrangement::Traits_2::Point_2` and `Kernel::Point_2` are convertible to one another, and (iii) the `Kernel` has a nested functor called `Construct_midpoint_2`, the function operator of which accepts two points p and q and returns the midpoint of the segment pq .

- 2.2 Optimize the program coded in the files `polytope_projection.cpp` and `Arr_inserter.h` as much as possible. Use the specialized insertion-methods that accept handles to halfedges as operands instead of the insertion methods that accept handles to vertices.
- 2.3 In this exercise you are asked to gradually develop an interactive system that creates and modifies an arrangement. Optionally, equip the system with the ability to render the arrangement in a dedicated window, adding visual capabilities.

- (a) Write a function that accepts two positive integers m and n , and constructs a grid-like arrangement of $m \times n$ squared faces in the first quadrant of the Cartesian plane. The figure to the right depicts an arrangement of 3×2 squared faces. Use the specialized insertion methods that accept handles to vertices as operands to insert all the segments except the first one.



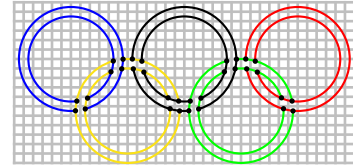
- (b) Apply an optimization to the function above, by using the specialized insertion methods that accept handles to halfedges as operands to insert all the segments except the first one.
- (c) Complete the development of the interactive system that creates and modifies an arrangement. When the system is ready to process a command, it prompts the user. The user as a response may issue one of the following commands:
- c** m n — create an arrangement of $m \times n$ squared faces in the first quadrant of the Cartesian plane.

- i** $m\ n$ — increase the number of columns by m and the number of rows by n .
- d** $m\ n$ — decrease the number of columns by m and the number of rows by n .
- m** $m\ n$ — multiply the number of columns and rows by m and n , respectively.
- /** $m\ n$ — divide the number of columns and rows by m and n , respectively.
- t** $x\ y$ — translate the arrangement by the vector (x, y) . Use the `modify_vertex()` and `modify_edge()` methods to perform the operation.
- s** $x\ y$ — scale the arrangement by the scaling factors x and y .
- h** $x\ y$ — add the vector (ix, jy) to every grid point $p_{i,j}$.

- 2.4 Given a set S of points in the plane such that no three of them lie on a common line, the triangulation of S is created by adding a maximal set of pairwise interior-disjoint segments connecting pairs of points in S . The result is an arrangement of segments whose vertices are the points in S , and which subdivides the convex hull of S into triangles. Write a program that reads an arrangement of line segments from a file and decides whether it is a triangulation of the vertices of the arrangement.

Remark: After reading the arrangement file, check that no three vertices of the arrangement lie on a line. You can do this naively in time that is cubic in the number of vertices. Carrying out this test efficiently is not trivial, and you will learn how to do it when you read Chapter 4.

- 2.5 Develop a function that constructs an arrangement of five interlocking rings that form the symbol of the Olympic Games, as illustrated in the figure to the right. Each ring must be represented as two polylines approximating the inner and outer circles of the ring. The number of segments that compose a polyline is $\lceil sr \rceil$, where r is the radius and s is some rational scale; two pairs, $\langle s_1, r_1 \rangle$ and $\langle s_2, r_2 \rangle$, are passed as input parameters to the function for the two polylines.



Develop a program that renders the arrangement that represents the symbol of the Olympic Games in a dedicated window with graphics. Assuming that the arrangement is displayed on a raster screen with a specific pixel resolution, the program should accept the window width and height, and the approximate length of a rendered segment that composes a polyline that represents a circle, given in terms of pixels. (Essentially, this length determines the quality of the rendered circles.)

Let r be the radius of a circle and let ℓ be the length of a segment of the polyline that represents the circle in terms of pixels. Set the rational scale that governs the number of segments that compose the polyline to $s = \frac{2\pi r}{\ell}$.

CGAL Arrangements and Their Applications

A Step-by-Step Guide

Fogel, E.; Halperin, D.; Wein, R.

2012, XIX, 293 p., Hardcover

ISBN: 978-3-642-17282-3