SERIE B — INFORMATIK

Smallest Enclosing Circles
An Exact and Generic Implementation in C++ *

Bernd Gärtner†
Sven Schönherr‡

B 98-04
April 1998

Abstract

We present a C++ implementation of an optimisation algorithm for computing the smallest (w.r.t. area) enclosing circle of a finite point set in the plane. The algorithm is implemented as a semi-dynamic data structure, thus allowing to insert points while maintaining the smallest enclosing circle. Following the *generic programming* paradigm, we use the template feature of C++ to provide generic code. The data structure is parameterized with a traits class, that defines the abstract interface between the optimisation algorithm and the primitives it uses. The interface of the data structure is compliant with the STL.

* supported by the ESPRIT IV LTR Project No. 21957 (CGAL).
† Institut für Theoretische Informatik, ETH Zürich, Haldeneggsteig 4, CH-8092 Zürich, Switzerland, e-mail: `gaertner@inf.ethz.ch`
‡ Institut für Informatik, Freie Universität Berlin, Takustr. 9, D-14195 Berlin, Germany, e-mail: `sven@inf.fu-berlin.de`

1 Introduction

We present a C++ implementation of an optimisation algorithm for computing the smallest (w.r.t. area) enclosing circle of a finite point set in the plane. The algorithm is implemented as a semi-dynamic data structure, thus allowing to insert points while maintaining the smallest enclosing circle. It is parameterized with a traits class [3], that defines the abstract interface between the optimisation algorithm and the primitives it uses. We provide a traits class implementation using the CGAL kernel [2] and, for ease-of-use, traits class adapters to user supplied point classes. The interface of the data structure is compliant with the STL [10, 9].

The presented code will be part of release 1.0 of CGAL, the Computational Geometry Algorithms Library [1].

The rest of the document is organized as follows. The algorithm is described in Section 2. Section 3 contains the specifications as they appear in the CGAL Reference Manual [4]. Section 4 gives the implementations. In Section 5 we provide a test program which performs some correctness checks. Finally the code files are created in Section 6. For a more detailed overview, see the table of contents starting on page 84.

2 The Algorithm

The implementation is based on an algorithm by Welzl [11], which we shortly describe now. The smallest (w.r.t. area) enclosing circle of a finite point set P in the plane, denoted by $mc(P)$, is built up incrementally, adding one point after another. Assume $mc(P)$ has been constructed, and we would like to obtain $mc(P \cup \{p\})$, p some new point. There are two cases: if p already lies inside $mc(P)$, then $mc(P \cup \{p\}) = mc(P)$. Otherwise p must lie on the boundary of $mc(P \cup \{p\})$ (this is proved in [11] and not hard to see), so we need to compute $mc(P, \{p\})$, the smallest circle enclosing P with p on the boundary. This is recursively done in the same manner. In general, for point sets P,B, define $mc(P, B)$ as the smallest circle enclosing P that has the points of B on the boundary (if defined). Although the algorithm finally delivers a circle $mc(P, \emptyset)$, it internally deals with circles that have a possibly nonempty set B. Here is the pseudo-code of Welzl's method. To compute $mc(P)$, it is called with the pair $(P, \emptyset)$, assuming that $P = \{p_1, \ldots, p_n\}$ is stored in a linked list.

$\texttt{mc}(P, B)$:
    $mc := \texttt{mc}(\emptyset, B)$
    IF $|B| = 3$ THEN RETURN mc
    FOR $i := 1$ TO n DO
        IF $p_i \not\in mc$ THEN
            $mc := \texttt{mc}(\{p_1, \ldots, p_{i-1}\}, B \cup \{p_i\})$
            move p_i to the front of P
        END
    END
    RETURN mc

Note the following: (a) $|B|$ is always bounded by 3, thus the computation of $mc(\emptyset, B)$ is easy. In our implementation, it is done by the private member function *compute_circle*. (b) One can check that the method maintains the invariant '$mc(P, B)$ exists'. This justifies termination if $|B| = 3$, because then $mc(P, B)$ must be the unique circle with the points of B on the boundary, and $mc(P, B)$ exists if and only if this circle contains the points of P. Thus, no subsequent in-circle tests are necessary anymore (for details see [11]). (c) points which are found to lie outside the current circle mc are considered 'important' and are moved to the front of the linked list that stores P. This is crucial for the method's efficiency.

It can also be advisable to bring P into random order before computation starts. There are 'bad' insertion orders which cause the method to be very slow – random shuffling gives these orders a very small probability.

3 Specifications

This section contains the specifications as they appear in the CGAL Reference Manual [4].

3.1 2D Smallest Enclosing Circle (`CGAL_Min_circle_2<Traits>`)

Definition

An object of the class `CGAL_Min_circle_2<Traits>` is the unique circle of smallest area enclosing a finite set of points in two-dimensional Euclidean plane $\mathbb{E}_2$. For a point set P we denote by $mc(P)$ the smallest circle that contains all points of P. Note that $mc(P)$ can be degenerate, i.e. $mc(P) = \emptyset$ if $P = \emptyset$ and $mc(P) = \{p\}$ if $P = \{p\}$.

An inclusion-minimal subset S of P with $mc(S) = mc(P)$ is called a *support set*, the points in S are the *support points*. A support set has size at most three, and all its points lie on the boundary of $mc(P)$. If $mc(P)$ has more than three points on the boundary, neither the support set nor its size are necessarily unique.

The underlying algorithm can cope with all kinds of input, e.g. P may be empty or points may occur more than once. The algorithm computes a support set S which remains fixed until the next insert or clear operation.

Note: In this release correct results are only guaranteed if exact arithmetic is used, see Section 3.3.

#include `<CGAL/Min_circle_2.h>`

Traits Class

The template parameter `Traits` is a traits class that defines the abstract interface between the optimisation algorithm and the primitives it uses. For example `Traits::Point` is a mapping on a point class. Think of it as 2D points in the Euclidean plane.

We provide a traits class implementation using the CGAL 2D kernel as described in Section 3.3. Traits class adapters to user supplied point classes are available, see Sections 3.4

and 3.5. Customizing own traits classes for optimisation algorithms can be done according to the requirements for traits classes listed in Section 3.8.

Types

`CGAL_Min_circle_2<Traits>:: Traits`

`typedef Traits::Point`	`Point;`	Point type.
`typedef Traits::Circle`	`Circle;`	Circle type.

The following types denote iterators that allow to traverse all points and support points of the smallest enclosing circle, resp. The iterators are non-mutable and their value type is `Point`. The iterator category is given in parentheses.

`CGAL_Min_circle_2<Traits>:: Point_iterator` (bidirectional).

`CGAL_Min_circle_2<Traits>:: Support_point_iterator` (random access).

Creation

A `CGAL_Min_circle_2<Traits>` object can be created from an arbitrary point set P and by specialized construction methods expecting no, one, two or three points as arguments. The latter methods can be useful for reconstructing $mc(P)$ from a given support set S of P.

```
template < class InputIterator >
CGAL_Min_circle_2<Traits>  min_circle( InputIterator first,
                                       InputIterator last,
                                       bool randomize = false,
                                       CGAL_Random& random = CGAL_random,
                                       Traits traits = Traits())
```

creates a variable `min_circle` of type `CGAL_Min_circle_2<Traits>`. It is initialized to $mc(P)$ with P being the set of points in the range [`first`,`last`). If `randomize` is `true`, a random permutation of P is computed in advance, using the random numbers generator `random`. Usually, this will not be necessary, however, the algorithm's efficiency depends on the order in which the points are processed, and a bad order might lead to extremely poor performance (see example below). *Precondition*: The value type of `first` and `last` is `Point`.

Note: In case a compiler does not support member templates yet, we provide specialized constructors instead. In the current release there are constructors for C arrays (using pointers as iterators), for the STL sequence containers `vector<Point>` and `list<Point>` and for the STL input stream iterator `istream_iterator<Point>`.

```
CGAL_Min_circle_2<Traits>  min_circle( Traits traits = Traits());
```

creates a variable `min_circle` of type `CGAL_Min_circle_2<Traits>`. It is initialized to $mc(\emptyset)$, the empty set.
Postcondition: `min_circle.is_empty()` = `true`.

`CGAL_Min_circle_2<Traits>  min_circle( Point p, Traits traits = Traits());`

creates a variable `min_circle` of type `CGAL_Min_circle_2<Traits>`. It is initialized to $mc(\{p\})$, the set $\{p\}$.
Postcondition: `min_circle.is_degenerate()` = `true`.

```
CGAL_Min_circle_2<Traits>  min_circle( Point p1,
                                       Point p2,
                                       Traits traits = Traits())
```

creates a variable `min_circle` of type `CGAL_Min_circle_2<Traits>`. It is initialized to $mc(\{p1, p2\})$, the circle with diameter equal to the segment connecting $p1$ and $p2$.

```
CGAL_Min_circle_2<Traits>  min_circle( Point p1,
                                       Point p2,
                                       Point p3,
                                       Traits traits = Traits())
```

creates a variable `min_circle` of type `CGAL_Min_circle_2<Traits>`. It is initialized to $mc(\{p1, p2, p3\})$.

Access Functions

`int` `min_circle.number_of_points()`

returns the number of points of `min_circle`, i.e. $|P|$.

`int` `min_circle.number_of_support_points()`

returns the number of support points of `min_circle`, i.e. $|S|$.

`Point_iterator` `min_circle.points_begin()`

returns an iterator referring to the first point of `min_circle`.

`Point_iterator` `min_circle.points_end()`

returns the corresponding past-the-end iterator.

`Support_point_iterator` `min_circle.support_points_begin()`

returns an iterator referring to the first support point of `min_circle`.

`Support_point_iterator` `min_circle.support_points_end()`

returns the corresponding past-the-end iterator.

`Point` `min_circle.support_point( int i)`

returns the `i`-th support point of `min_circle`. Between two modifying operations (see below) any call to `min_circle.support_point(i)` with the same `i` returns the same point.
Precondition:
$0 \leq i <$ `min_circle.number_of_support_points()`.

`Circle` `min_circle.circle()`

returns the current circle of `min_circle`.

Predicates

By definition, an empty `CGAL_Min_circle_2<Traits>` has no boundary and no bounded side, i.e. its unbounded side equals the whole plane $\mathbb{E}_2$.

`CGAL_Bounded_side` `min_circle.bounded_side( Point p)`

returns `CGAL_ON_BOUNDED_SIDE`, `CGAL_ON_BOUNDARY`, or `CGAL_ON_UNBOUNDED_SIDE` iff `p` lies properly inside, on the boundary, or properly outside of `min_circle`, resp.

`bool` `min_circle.has_on_bounded_side( Point p)`

returns `true`, iff `p` lies properly inside `min_circle`.

`bool` `min_circle.has_on_boundary( Point p)`

returns `true`, iff `p` lies on the boundary of `min_circle`.

`bool` `min_circle.has_on_unbounded_side( Point p)`

returns `true`, iff `p` lies properly outside of `min_circle`.

`bool` `min_circle.is_empty()`

returns `true`, iff `min_circle` is empty (this implies degeneracy).

`bool` `min_circle.is_degenerate()`

returns `true`, iff `min_circle` is degenerate, i.e. if `min_circle` is empty or equal to a single point, equivalently if the number of support points is less than 2.

Modifiers

New points can be added to an existing `min_circle`, allowing to build $mc(P)$ incrementally, e.g. if P is not known in advance. Compared to the direct creation of $mc(P)$, this is not much slower, because the construction method is incremental itself.

`void` `min_circle.insert( Point p)`

inserts `p` into `min_circle` and recomputes the smallest enclosing circle.

```
template < class InputIterator >
void                    min_circle.insert( InputIterator first,
                                           InputIterator last)
```

inserts the points in the range [`first`,`last`) into `min_circle` and recomputes the smallest enclosing circle by calling `insert(p)` for each point `p` in [`first`,`last`).
Precondition: The value type of `first` and `last` is `Point`.

Note: In case a compiler does not support member templates yet, we provide specialized `insert` functions instead. In the current release there are `insert` functions for C arrays (using pointers as iterators), for the STL sequence containers `vector<Point>` and `list<Point>` and for the STL input stream iterator `istream_iterator<Point>`.

`void` `min_circle.clear()`

deletes all points in `min_circle` and sets it to the empty set.
Postcondition: `min_circle.is_empty()` = `true`.

Validity Check

An object `min_circle` is valid, iff

- `min_circle` contains all points of its defining set P,
- `min_circle` is the smallest circle spanned by its support set S, and
- S is minimal, i.e. no support point is redundant.

Using the traits class implementation for the CGAL kernel with exact arithmetic as described in Section 3.3 guarantees validity of `min_circle`. The following function is mainly intended for debugging user supplied traits classes but also for convincing the anxious user that the traits class implementation is correct.

`bool` `min_circle.is_valid( bool verbose = false, int level = 0)`

returns `true`, iff `min_circle` is valid. If `verbose` is `true`, some messages concerning the performed checks are written to standard error stream. The second parameter `level` is not used, we provide it only for consistency with interfaces of other classes.

Miscellaneous

`Traits` `min_circle.traits()`

returns a const reference to the traits class object.

I/O

`ostream&` `ostream& os << min_circle`

writes `min_circle` to output stream `os`.
Precondition: The output operator is defined for `Point` (and for `Circle`, if pretty printing is used).

`istream&` `istream& is >> & min_circle`

reads `min_circle` from input stream `is`.
Precondition: The input operator is defined for `Point`.

#include `<CGAL/IO/Window_stream.h>`

`CGAL_Window_stream&` `CGAL_Window_stream& ws << min_circle`

writes `min_circle` to window stream `ws`.
Precondition: The window stream output operator is defined for `Point` and `Circle`.

See Also

`CGAL_Min_ellipse_2` [5], `CGAL_Min_circle_2_traits_2` (Section 3.3), `CGAL_Min_circle_2_adapterC2` (Section 3.4), `CGAL_Min_circle_2_adapterH2` (Section 3.5).

Implementation

We implement the algorithm of Welzl, with move-to-front heuristic [11]. If randomization is chosen, the creation time is almost always linear in the number of points. Access functions and predicates take constant time, inserting a point might take up to linear time, but substantially less than computing the new smallest enclosing circle from scratch. The clear operation and the check for validity each takes linear time.

Example

To illustrate the creation of `CGAL_Min_circle_2<Traits>` and to show that randomization can be useful in certain cases, we give an example.

```
#include <CGAL/Gmpz.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Point_2.h>
#include <CGAL/Min_circle_2_traits_2.h>
#include <CGAL/Min_circle_2.h>

typedef  CGAL_Gmpz                      NT;
typedef  CGAL_Homogeneous<NT>           R;
typedef  CGAL_Point_2<R>                Point;
typedef  CGAL_Min_circle_2_traits_2<R>  Traits;
typedef  CGAL_Min_circle_2<Traits>      Min_circle;

int main()
{
    int     n = 1000;
    Point*  P = new Point[ n];

    for ( int i = 0; i < n; ++i)
        P[ i] = Point( (i%2 == 0 ? i : -i), 0);
    // (0,0), (-1,0), (2,0), (-3,0), ...

    Min_circle  mc1( P, P+n);           // very slow
    Min_circle  mc2( P, P+n, true);     // fast

    delete[] P;
    return( 0);
}
```

3.2 2D Optimisation Circle (`CGAL_Optimisation_circle_2<R>`)

Definition

An object of the class `CGAL_Optimisation_circle_2<R>` is a circle in the two-dimensional Euclidean plane $\mathbb{E}_2$. Its boundary splits $\mathbb{E}_2$ into a bounded and an unbounded side. Note that the circle can be degenerated, i.e. it can be empty or contain only a single point. By definition, an empty `CGAL_Optimisation_circle_2<R>` has no boundary and no bounded side, i.e. its unbounded side equals the whole plane $\mathbb{E}_2$. A `CGAL_Optimisation_circle_2<R>` containing exactly one point p has no bounded side, its boundary is $\{p\}$, and its unbounded side equals $\mathbb{E}_2 \setminus \{p\}$.

`#include <CGAL/Optimisation_circle_2.h>`

Types

`typedef CGAL_Point_2<R>`	`Point;`	Point type.
`typedef R::FT`	`Distance;`	Distance type.

Creation

void circle.set()

sets circle to the empty circle.

void circle.set(CGAL_Point_2<R> p)

sets circle to the circle containing exactly p.

void circle.set(CGAL_Point_2<R> p, CGAL_Point_2<R> q)

sets circle to the circle with diameter $\overline{pq}$.
Precondition: p and q are distinct.

void circle.set(CGAL_Point_2<R> p,
CGAL_Point_2<R> q,
CGAL_Point_2<R> r)

sets circle to the unique circle through p,q,r.
Precondition: p, q, r are not collinear.

void circle.set(CGAL_Point_2<R> center,
R::FT squared_radius)

sets circle to the circle with center center and squared radius squared_radius.

Access Functions

CGAL_Point_2<R> circle.center()

returns the center of circle.

R::FT circle.squared_radius()

returns the squared radius of circle.

Equality Tests

bool circle == circle2

returns true, iff circle and circle2 are equal, i.e. if they have the same center and same squared radius.

bool circle != circle2

returns true, iff circle and circle2 are not equal.

Predicates

CGAL_Bounded_side   circle.bounded_side(CGAL_Point_2<R> p)

> returns CGAL_ON_BOUNDED_SIDE, CGAL_ON_BOUNDARY, or CGAL_ON_UNBOUNDED_SIDE iff p lies properly inside, on the boundary, or properly outside of circle, resp.

bool   circle.has_on_bounded_side(CGAL_Point_2<R> p)

> returns true, iff p lies properly inside circle.

bool   circle.has_on_boundary(CGAL_Point_2<R> p)

> returns true, iff p lies on the boundary of circle.

bool   circle.has_on_unbounded_side(CGAL_Point_2<R> p)

> returns true, iff p lies properly outside of circle.

bool   circle.is_empty()

> returns true, iff circle is empty (this implies degeneracy).

bool   circle.is_degenerate()

> returns true, iff circle is degenerate, i.e. if circle is empty or equal to a single point.

I/O

ostream&   ostream& os << circle

> writes circle to output stream os.

istream&   istream& is >> & circle

> reads circle from input stream is.

#include <CGAL/IO/Window_stream.h>

CGAL_Window_stream&   CGAL_Window_stream& ws << circle

> writes circle to window stream ws.

3.3 Traits Class Implementation using the two-dimensional CGAL Kernel (`CGAL_Min_circle_2_traits_2<R>`)

Definition

The class `CGAL_Min_circle_2_traits_2<R>` interfaces the 2D optimisation algorithm for smallest enclosing circles with the CGAL 2D kernel.

`#include <CGAL/Min_circle_2_traits_2.h>`

Types

```
typedef CGAL_Point_2<R>                 Point;

typedef CGAL_Optimisation_circle_2<R>   Circle;
```

Creation

```
CGAL_Min_circle_2_traits_2<R>  traits;

CGAL_Min_circle_2_traits_2<R>  traits( CGAL_Min_circle_2_traits_2<R>);
```

Operations

```
CGAL_Orientation    traits.orientation( Point p, Point q, Point r)
```

returns `CGAL_orientation( p, q, r)`.

See Also

`CGAL_Min_circle_2` (Section 3.1), `CGAL_Min_circle_2_adapterC2` (Section 3.4), `CGAL_Min_circle_2_adapterH2` (Section 3.5), Requirements of Traits Classes for 2D Smallest Enclosing Circle (Section 3.8).

Example

See example for `CGAL_Min_circle_2` (Section 3.1).

3.4 Traits Class Adapter for 2D Smallest Enclosing Circle to 2D Cartesian Points (`CGAL_Min_circle_2_adapterC2<PT,DA>`)

Definition

The class `CGAL_Min_circle_2_adapterC2<PT,DA>` interfaces the 2D optimisation algorithm for smallest enclosing circles with the point class `PT`. The data accessor `DA` [6] is used to access the x- and y-coordinate of `PT`, i.e. `PT` is supposed to have a Cartesian representation of its coordinates.

`#include <CGAL/Min_circle_2_adapterC2.h>`

Types

`CGAL_Min_circle_2_adapterC2<PT,DA>:: DA`	Data accessor for Cartesian coordinates.
`CGAL_Min_circle_2_adapterC2<PT,DA>:: Point`	Point type.
`CGAL_Min_circle_2_adapterC2<PT,DA>:: Circle`	Circle type.

Creation

```
CGAL_Min_circle_2_adapterC2<PT,DA>  adapter( DA da = DA());
CGAL_Min_circle_2_adapterC2<PT,DA>  adapter(
                                        CGAL_Min_circle_2_adapterC2<PT,DA>);
```

Operations

```
CGAL_Orientation  adapter.orientation( Point p, Point q, Point r)
```

returns `CGAL_LEFTTURN`, `CGAL_COLLINEAR`, or `CGAL_RIGHTTURN` iff `r` lies properly to the left, on, or properly to the right of the oriented line through `p` and `q`, resp.

See Also

`CGAL_Min_circle_2` (Section 3.1), `CGAL_Min_circle_2_traits_2` (Section 3.3), `CGAL_Min_circle_2_adapterH2` (Section 3.5), Requirements of Traits Class Adapters to 2D Cartesian Points (Section 3.6).

Example

The following example illustrates the use of the traits class adapters with your own point class. For the sake of simplicity, we expect a point class with Cartesian `double` coordinates and access functions `x()` and `y()`. Based on this we show how to implement and use data accessors.

Note: In this release correct results are only guaranteed if exact arithmetic is used, so this example (using inexact floating-point arithmetic) is only intended to illustrate the techniques.

```
#include <CGAL/Min_circle_2_adapterC2.h>
#include <CGAL/Min_circle_2.h>

// your own point class (Cartesian)
class PtC {
    // ...
  public:
    PtC( double x, double y);
    double  x( ) const;
    double  y( ) const;
    // ...
```

```
};

// the data accessor for PtC
class PtC_DA {
  public:
    typedef  double  FT;
    void  get( const PtC& p, double& x, double& y) const {
        x = p.x(); y = p.y();
    }
    double  get_x( const PtC& p) const { return( p.x()); }
    double  get_y( const PtC& p) const { return( p.y()); }
    void  set( PtC& p, double x, double y) const { p = PtC( x, y); }
};

// some typedefs
typedef  CGAL_Min_circle_2_adapterC2 < PtC, PtC_DA >  AdapterC;
typedef  CGAL_Min_circle_2 < AdapterC >              Min_circle;

// do something with Min_circle
Min_circle   mc( /*...*/ );
```

3.5 Traits Class Adapter for 2D Smallest Enclosing Circle to 2D Homogeneous Points (CGAL_Min_circle_2_adapterH2<PT,DA>)

Definition

The class CGAL_Min_circle_2_adapterH2<PT,DA> interfaces the 2D optimisation algorithm for smallest enclosing circles with the point class PT. The data accessor DA [6] is used to access the hx-, hy- and hw-coordinate of PT, i.e. PT is supposed to have a homogeneous representation of its coordinates.

#include <CGAL/Min_circle_2_adapterH2.h>

Types

CGAL_Min_circle_2_adapterH2<PT,DA>:: DA	Data accessor for homogeneous coordinates.
CGAL_Min_circle_2_adapterH2<PT,DA>:: Point	Point type.
CGAL_Min_circle_2_adapterH2<PT,DA>:: Circle	Circle type.

Creation

```
CGAL_Min_circle_2_adapterH2<PT,DA>  adapter( DA da = DA());

CGAL_Min_circle_2_adapterH2<PT,DA>  adapter(
                                        CGAL_Min_circle_2_adapterH2<PT,DA>);
```

Operations

`CGAL_Orientation` `adapter.orientation( Point p, Point q, Point r)`

returns `CGAL_LEFTTURN`, `CGAL_COLLINEAR`, or `CGAL_RIGHTTURN` iff `r` lies properly to the left, on, or properly to the right of the oriented line through `p` and `q`, resp.

See Also

`CGAL_Min_circle_2` (Section 3.1), `CGAL_Min_circle_2_traits_2` (Section 3.3), `CGAL_Min_circle_2_adapterC2` (Section 3.4), Requirements of Traits Class Adapters to 2D Homogeneous Points (Section 3.7).

Example

The following example illustrates the use of the traits class adapters with your own point class. For the sake of simplicity, we expect a point class with homogeneous `int` coordinates and access functions `hx()`, `hy()`, and `hw()`. Based on this we show how to implement and use data accessors.

Note: In this release correct results are only guaranteed if exact arithmetic is used, so this example (using integer arithmetic with possible overflows) is only intended to illustrate the techniques.

```
#include <CGAL/Min_circle_2_adapterH2.h>
#include <CGAL/Min_circle_2.h>

// your own point class (homogeneous)
class PtH {
    // ...
  public:
    PtH( int hx, int hy, int hw);
    int  hx( ) const;
    int  hy( ) const;
    int  hw( ) const;
    // ...
};

// the data accessor for PtH
class PtH_DA {
  public:
    typedef  int  RT;
    void  get( const PtH& p, int& hx, int& hy, int& hw) const {
        hx = p.hx(); hy = p.hy(); hw = p.hw();
    }
    int  get_x( const PtH& p) const { return( p.hx()); }
    int  get_y( const PtH& p) const { return( p.hy()); }
    int  get_w( const PtH& p) const { return( p.hw()); }
    void  set( PtH& p, int hx, int hy, int hw) const { p = PtH( hx, hy, hw); }
};
```

```
// some typedefs
typedef  CGAL_Min_circle_2_adapterH2< PtH, PtH_DA >  AdapterH;
typedef  CGAL_Min_circle_2< AdapterH >               Min_circle;

// do something with Min_circle
Min_circle   mc( /*...*/ );
```

3.6 Requirements of Traits Class Adapters to 2D Cartesian Points

The family of traits class adapters `..._adapterC2` to 2D Cartesian points is parameterized with a point type `PT` and a data accessor `DA` [6]. The latter defines the coordinates-based interface between the traits class adapter and the point type. The following requirements catalog lists the primitives, i.e. types, member functions etc., that must be defined for classes `PT` and `DA` that can be used to parameterize `..._adapterC2`.

Point Type (PT)

`PT p;`		Default constructor.
`PT p( PT);`		Copy constructor.
`PT&`	`p = q`	Assignment.
`bool`	`p == q`	Equality test.

The following I/O operators are only needed, if the corresponding I/O operators of the optimisation algorithm are used.

`ostream&`	`ostream& os << p`	writes `p` to output stream `os`.
`istream&`	`istream& is >> &p`	reads `p` from input stream `is`.

Read/Write Data Accessor (DA)

`DA:: FT`		The number type `FT` has to fulfill the requirements of a CGAL field type.
`DA da;`		Default constructor.
`DA da( DA);`		Copy constructor.
`void`	`da.get( Point p, FT& x, FT& y)`	returns the Cartesian coordinates of `p` in `x` and `y`, resp.
`FT`	`da.get_x( Point p)`	returns the Cartesian x-coordinate of `p`.
`FT`	`da.get_y( Point p)`	returns the Cartesian y-coordinate of `p`.
`void`	`da.set( Point& p, FT x, FT y)`	sets `p` to the point with Cartesian coordinates `x` and `y`.

3.7 Requirements of Traits Class Adapters to 2D Homogeneous Points

The family of traits class adapters `..._adapterH2` to 2D homogeneous points is parameterized with a point type `PT` and a data accessor `DA` [6]. The latter defines the coordinates-based interface between the traits class adapter and the point type. The following requirements catalog lists the primitives, i.e. types, member functions etc., that must be defined for classes `PT` and `DA` that can be used to parameterize `..._adapterH2`.

Point Type (`PT`)

`PT p;`		Default constructor.
`PT p( PT);`		Copy constructor.
`PT&`	`p = q`	Assignment.
`bool`	`p == q`	Equality test.

The following I/O operators are only needed, if the corresponding I/O operators of the optimisation algorithm are used.

`ostream&`	`ostream& os << p`	writes `p` to output stream `os`.
`istream&`	`istream& is >> &p`	reads `p` from input stream `is`.

Read/Write Data Accessor (`DA`)

`DA:: RT`		The number type `RT` has to fulfill the requirements of a CGAL ring type.
`DA da;`		Default constructor.
`DA da( DA);`		Copy constructor.
`void`	`da.get( Point p, RT& hx, RT& hy, RT& hw)`	returns the homogeneous coordinates of `p` in `hx`, `hy` and `hw`, resp.
`RT`	`da.get_hx( Point p)`	returns the homogeneous x-coordinate of `p`.
`RT`	`da.get_hy( Point p)`	returns the homogeneous y-coordinate of `p`.
`RT`	`da.get_hw( Point p)`	returns the homogeneous w-coordinate of `p`.
`void`	`da.set( Point& p, RT hx, RT hy, RT hw)`	sets `p` to the point with homogeneous coordinates `hx`, `hy` and `hw`.

3.8 Requirements of Traits Classes for 2D Smallest Enclosing Circle

The class template `CGAL_Min_circle_2` is parameterized with a `Traits` class which defines the abstract interface between the optimisation algorithm and the primitives it uses. The following requirements catalog lists the primitives, i.e. types, member functions etc., that must be defined for a class that can be used to parameterize `CGAL_Min_circle_2`. A traits class implementation using the CGAL 2D kernel is available and described in Section 3.3. In addition, we provide traits class adapters to user supplied point classes, see Sections 3.4 and 3.5. Both, the implementation and the adapters, can be used as a starting point for customizing own traits classes, e.g. through derivation and specialization.

Traits Class (`Traits`)

Definition

A class that satisfies the requirements of a traits class for `CGAL_Min_circle_2` must provide the following primitives.

Types

`Traits:: Point`	The point type must provide default and copy constructor, assignment and equality test.
`Traits:: Circle`	The circle type must fulfill the requirements listed below in the next section.

In addition, if I/O is used, the corresponding I/O operators for `Point` and `Circle` have to be provided, see topic **I/O** in Section 3.1.

Variables

`Circle`	`circle;`	The actual circle. This variable is maintained by the algorithm, the user should neither access nor modify it directly.

Creation

Only default and copy constructor are required. Note that further constructors can be provided.

`Traits traits;`	A default constructor.
`Traits traits( Traits);`	A copy constructor.

Operations

The following predicate is only needed, if the member function `is_valid` of `CGAL_Min_circle_2` is used.

`CGAL_Orientation` `traits.orientation( Point p, Point q, Point r)`

returns constants `CGAL_LEFTTURN`, `CGAL_COLLINEAR`, or `CGAL_RIGHTTURN` iff `r` lies properly to the left, on, or properly to the right of the oriented line through `p` and `q`, resp.

Circle Type (`Circle`)

Definition

An object of the class `Circle` is a circle in two-dimensional Euclidean plane $\mathbb{E}_2$. Its boundary splits the plane into a bounded and an unbounded side. By definition, an empty `Circle` has no boundary and no bounded side, i.e. its unbounded side equals the whole plane $\mathbb{E}_2$. A `Circle` containing exactly one point p has no bounded side, its boundary is $\{p\}$, and its unbounded side equals $\mathbb{E}_2 \setminus \{p\}$.

Types

`Circle:: Point` Point type.

The following type is only needed, if the member function `is_valid` of `CGAL_Min_circle_2` is used.

`Circle:: Distance` Distance type. The function `squared_radius` (see below) returns an object of this type.

Creation

`void` `circle.set()`

sets `circle` to the empty circle.

`void` `circle.set( Point p)`

sets `circle` to the circle containing exactly {`p`}.

`void` `circle.set( Point p, Point q)`

sets `circle` to the circle with diameter $\overline{\texttt{pq}}$. The algorithm guarantees that `set` is never called with two equal points.

`void` `circle.set( Point p, Point q, Point r)`

sets `circle` to the circle through `p`, `q`, and `r`. The algorithm guarantees that `set` is never called with three collinear points.

Predicates

`bool` `circle.has_on_unbounded_side( Point p)`

returns `true`, iff `p` lies properly outside of `circle`.

Each of the following predicates is only needed, if the corresponding predicate of `CGAL_Min_circle_2` is used.

`CGAL_Bounded_side` `circle.bounded_side( Point p)`

returns `CGAL_ON_BOUNDED_SIDE`, `CGAL_ON_BOUNDARY`, or `CGAL_ON_UNBOUNDED_SIDE` iff `p` lies properly inside, on the boundary, or properly outside of `circle`, resp.

`bool` `circle.has_on_bounded_side( Point p)`

returns `true`, iff `p` lies properly inside `circle`.

`bool` `circle.has_on_boundary( Point p)`

returns `true`, iff `p` lies on the boundary of `circle`.

`bool` `circle.is_empty()`

returns `true`, iff `circle` is empty (this implies degeneracy).

`bool` `circle.is_degenerate()`

returns `true`, iff `circle` is degenerate, i.e. if `circle` is empty or equal to a single point.

Additional Operations for Checking

The following operations are only needed, if the member function `is_valid` of `CGAL_Min_circle_2` is used.

`bool` `circle == circle2`

returns `true`, iff `circle` and `circle2` are equal.

`Point` `circle.center()`

returns the center of `circle`.

`Distance` `circle.squared_radius()`

returns the squared radius of `circle`.

I/O

The following I/O operators are only needed, if the corresponding I/O operators of `CGAL_Min_circle_2` are used.

`ostream&` `ostream& os << circle`

writes `circle` to output stream `os`.

`istream&` `istream& is >> & circle`

reads `circle` from input stream `is`.

`CGAL_Window_stream&`

`CGAL_Window_stream& ws << circle`

writes `circle` to window stream `ws`.

4 Implementations

4.1 Class Template `CGAL_Min_circle_2<Traits>`

First, we declare the class template `CGAL_Min_circle_2`.

Min_circle_2 declaration [1] $\equiv$ {

```
template < class _Traits >
class CGAL_Min_circle_2;
```

}

This macro is invoked in definition 59.

The actual work of the algorithm is done in the private member functions `mc` and `compute_circle`. The former directly realizes the pseudo-code of $\texttt{mc}(P, B)$, the latter solves the basic case $\texttt{mc}(\emptyset, B)$, see Section 2.

Workaround: The GNU compiler (g++ 2.7.2[.x]) does not accept types with scope operator as argument type or return type in class template member functions. Therefore, all member functions are implemented in the class interface.

The class interface looks as follows.

Min_circle_2 interface [2] $\equiv$ {

```
template < class _Traits >
class CGAL_Min_circle_2 {
  public:
    Min_circle_2 public interface [3]

  private:
    // private data members
    Min_circle_2 private data members [4]
```

```
    // copying and assignment not allowed!
    CGAL_Min_circle_2( const CGAL_Min_circle_2<_Traits>&);
    CGAL_Min_circle_2<_Traits>&
        operator = ( const CGAL_Min_circle_2<_Traits>&);
```

dividing line [68]

```
// Class implementation
// ====================

  public:
    // Access functions and predicates
    // -------------------------------
```

Min_circle_2 access functions 'number_of_...' [10]

Min_circle_2 predicates 'is_...' [13]

Min_circle_2 access functions [11]

Min_circle_2 predicates [14]

```
  private:
    // Private member functions
    // ------------------------
```

Min_circle_2 private member function 'compute_circle' [28]

Min_circle_2 private member function 'mc' [29]

```
  public:
    // Constructors
    // ------------
```

Min_circle_2 constructors [7]

```
    // Destructor
    // ----------
```

Min_circle_2 destructor [9]

```
    // Modifiers
    // ---------
```

Min_circle_2 modifiers [15]

```
    // Validity check
    // --------------
```

Min_circle_2 validity check [18]

```
    // Miscellaneous
```

```
        // -------------
        Min_circle_2 miscellaneous [25]
    };
}
```
This macro is invoked in definition 59.

4.1.1 Public Interface

The functionality is described and documented in the specification section, so we do not comment on it here.

Min_circle_2 public interface [3] ≡ {

```
    // types
    typedef           _Traits                        Traits;
    typedef typename  _Traits::Point                 Point;
    typedef typename  _Traits::Circle                Circle;
    typedef typename  list<Point>::const_iterator    Point_iterator;
    typedef           const Point *                  Support_point_iterator;

    /**************************************************************
    WORKAROUND: The GNU compiler (g++ 2.7.2[.x]) does not accept types
    with scope operator as argument type or return type in class template
    member functions. Therefore, all member functions are implemented in
    the class interface.

    // creation
    CGAL_Min_circle_2( const Point*  first,
                       const Point*  last,
                       bool          randomize = false,
                       CGAL_Random&  random    = CGAL_random,
                       const Traits& traits    = Traits());
    CGAL_Min_circle_2( list<Point>::const_iterator first,
                       list<Point>::const_iterator last,
                       bool          randomize = false,
                       CGAL_Random&  random    = CGAL_random,
                       const Traits& traits    = Traits());
    CGAL_Min_circle_2( istream_iterator<Point,ptrdiff_t> first,
                       istream_iterator<Point,ptrdiff_t> last,
                       bool          randomize = false,
                       CGAL_Random&  random    = CGAL_random,
                       const Traits& traits    = Traits())
    CGAL_Min_circle_2( const Traits& traits = Traits());
    CGAL_Min_circle_2( const Point&  p,
                       const Traits& traits = Traits());
    CGAL_Min_circle_2( const Point&  p,
                       const Point&  q,
                       const Traits& traits = Traits());
```

```
    CGAL_Min_circle_2( const Point&  p1,
                       const Point&  p2,
                       const Point&  p3,
                       const Traits& traits = Traits());
    ~CGAL_Min_circle_2( );

    // access functions
    int  number_of_points        ( ) const;
    int  number_of_support_points( ) const;

    Point_iterator  points_begin( ) const;
    Point_iterator  points_end  ( ) const;

    Support_point_iterator  support_points_begin( ) const;
    Support_point_iterator  support_points_end  ( ) const;

    const Point&  support_point( int i) const;

    const Circle&  circle( ) const;

    // predicates
    CGAL_Bounded_side  bounded_side( const Point& p) const;
    bool  has_on_bounded_side      ( const Point& p) const;
    bool  has_on_boundary          ( const Point& p) const;
    bool  has_on_unbounded_side    ( const Point& p) const;

    bool  is_empty     ( ) const;
    bool  is_degenerate( ) const;

    // modifiers
    void  insert( const Point& p);
    void  insert( const Point* first,
                  const Point* last );
    void  insert( list<Point>::const_iterator first,
                  list<Point>::const_iterator last );
    void  insert( istream_iterator<Point,ptrdiff_t> first,
                  istream_iterator<Point,ptrdiff_t> last );
    void  clear( );

    // validity check
    bool  is_valid( bool verbose = false, int level = 0) const;

    // miscellaneous
    const Traits&  traits( ) const;
    ********************************************************************/
}
```

This macro is invoked in definition 2.

4.1.2 Private Data Members

First, the traits class object is stored.

Min_circle_2 private data members [4] $+\equiv$ {
```
    Traits        tco;                  // traits class object
```
}

This macro is defined in definitions 4, 5, and 6.
This macro is invoked in definition 2.

The points of P are internally stored as a linked list that allows to bring points to the front of the list in constant time. We use the sequence container `list` from STL [10].

Min_circle_2 private data members [5] $+\equiv$ {
```
    list<Point>   points;               // doubly linked list of points
```
}

This macro is defined in definitions 4, 5, and 6.
This macro is invoked in definition 2.

The support set S of at most three support points is stored in an array `support_points`, the actual number of support points is given by `n_support_points`. During the computations, the set of support points coincides with the set B appearing in the pseudo-code for `mc`(P, B), see Section 2.

Workaround: The array of support points is allocated dynamically, because the SGI compiler (mipspro CC 7.1) does not accept a static array here.

Min_circle_2 private data members [6] $+\equiv$ {
```
    int           n_support_points;       // number of support points
    Point*        support_points;         // array of support points
```
}

This macro is defined in definitions 4, 5, and 6.
This macro is invoked in definition 2.

Finally, the actual circle is stored in a variable `circle` provided by the traits class object, by the end of computation equal to $mc(P)$. During computation, `tco.circle` equals the circle mc appearing in the pseudo-code for `mc`(P, B), see Section 2.

4.1.3 Constructors and Destructor

We provide several different constructors, which can be put into two groups. The constructors in the first group, i.e. the more important ones, build the smallest enclosing circle $mc(P)$ from a point set P, given by a begin iterator and a past-the-end iterator. Usually, this is implemented as a single member template, but in case a compiler does not support member templates yet, we provide specialized constructors for C arrays (using pointers as iterators), for STL sequence containers `vector<Point>` and `list<Point>` and for the STL input stream iterator `istream_iterator<Point>`. Actually, the constructors for a C array and a `vector<point>` are the same, since the random access iterator of `vector<Point>` is implemented as `Point*`.

All constructors of the first group copy the points into the internal list `points`. If randomization is demanded, the points are copied to a vector and shuffled at random, before being copied to **points**. Finally the private member function mc is called to compute $mc(P) = mc(P, \emptyset)$.

Min_circle_2 constructors [7] $+\equiv$ {

```
#ifndef CGAL_CFG_NO_MEMBER_TEMPLATES

    // STL-like constructor (member template)
    template < class InputIterator >
    CGAL_Min_circle_2( InputIterator first,
                       InputIterator last,
                       bool          randomize = false,
                       CGAL_Random&  random    = CGAL_random,
                       const Traits& traits    = Traits())
        : tco( traits)
    {
        // allocate support points' array
        support_points = new Point[ 3];

        // range not empty?
        if ( first != last) {

            // store points
            if ( randomize) {

                // shuffle points at random
                vector<Point> v( first, last);
                random_shuffle( v.begin(), v.end(), random);
                copy( v.begin(), v.end(), back_inserter( points)); }
            else
                copy( first, last, back_inserter( points)); }

        // compute mc
        mc( points.end(), 0);
    }

#else

    // STL-like constructor for C array and vector<Point>
    CGAL_Min_circle_2( const Point*  first,
                       const Point*  last,
                       bool          randomize = false,
                       CGAL_Random&  random    = CGAL_random,
                       const Traits& traits    = Traits())
        : tco( traits)
    {
        // allocate support points' array
        support_points = new Point[ 3];
```

```
    // range not empty?
    if ( ( last-first) > 0) {

        // store points
        if ( randomize) {

            // shuffle points at random
            vector<Point> v( first, last);
            random_shuffle( v.begin(), v.end(), random);
            copy( v.begin(), v.end(), back_inserter( points)); }
        else
            copy( first, last, back_inserter( points)); }

    // compute mc
    mc( points.end(), 0);
}

// STL-like constructor for list<Point>
CGAL_Min_circle_2( list<Point>::const_iterator first,
                   list<Point>::const_iterator last,
                   bool          randomize = false,
                   CGAL_Random&  random    = CGAL_random,
                   const Traits& traits    = Traits())
    : tco( traits)
{
    // allocate support points' array
    support_points = new Point[ 3];

    // compute number of points
    list<Point>::size_type n = 0;
    CGAL__distance( first, last, n);
    if ( n > 0) {

        // store points
        if ( randomize) {

            // shuffle points at random
            vector<Point> v;
            v.reserve( n);
            copy( first, last, back_inserter( v));
            random_shuffle( v.begin(), v.end(), random);
            copy( v.begin(), v.end(), back_inserter( points)); }
        else
            copy( first, last, back_inserter( points)); }

    // compute mc
    mc( points.end(), 0);
}
```

```
        // STL-like constructor for istream_iterator<Point>
        CGAL_Min_circle_2( istream_iterator<Point,ptrdiff_t>  first,
                           istream_iterator<Point,ptrdiff_t>  last,
                           bool          randomize = false,
                           CGAL_Random&  random    = CGAL_random,
                           const Traits& traits    = Traits())
            : tco( traits)
        {
            // allocate support points' array
            support_points = new Point[ 3];

            // range not empty?
            if ( first != last) {

                // store points
                if ( randomize) {

                    // shuffle points at random
                    vector<Point> v;
                    copy( first, last, back_inserter( v));
                    random_shuffle( v.begin(), v.end(), random);
                    copy( v.begin(), v.end(), back_inserter( points)); }
                else
                    copy( first, last, back_inserter( points)); }

            // compute mc
            mc( points.end(), 0);
        }

    #endif // CGAL_CFG_NO_MEMBER_TEMPLATES
}
```

This macro is defined in definitions 7 and 8.
This macro is invoked in definition 2.

The remaining constructors are actually specializations of the previous ones, building the smallest enclosing circle for up to three points. The idea is the following: recall that for any point set P there exists $S \subseteq P$, $|S| \leq 3$ with $mc(S) = mc(P)$ (in fact, such a set S is determined by the algorithm). Once S has been computed (or given otherwise), $mc(P)$ can easily be reconstructed from S in constant time. To make this reconstruction more convenient, a constructor is available for each size of $|S|$, ranging from 0 to 3. For $|S| = 0$, we get the default constructor, building $mc(\emptyset)$.

Min_circle_2 constructors [8] $+\equiv$ {

```
    // default constructor
    inline
    CGAL_Min_circle_2( const Traits& traits = Traits())
        : tco( traits), n_support_points( 0)
```

```
{
    // allocate support points' array
    support_points = new Point[ 3];

    // initialize circle
    tco.circle.set();

    CGAL_optimisation_postcondition( is_empty());
}

// constructor for one point
inline
CGAL_Min_circle_2( const Point& p, const Traits& traits = Traits())
    : tco( traits), points( 1, p), n_support_points( 1)
{
    // allocate support points' array
    support_points = new Point[ 3];

    // initialize circle
    support_points[ 0] = p;
    tco.circle.set( p);

    CGAL_optimisation_postcondition( is_degenerate());
}

// constructor for two points
inline
CGAL_Min_circle_2( const Point& p1,
                   const Point& p2,
                   const Traits& traits = Traits())
    : tco( traits)
{
    // allocate support points' array
    support_points = new Point[ 3];

    // store points
    points.push_back( p1);
    points.push_back( p2);

    // compute mc
    mc( points.end(), 0);
}

// constructor for three points
inline
CGAL_Min_circle_2( const Point& p1,
                   const Point& p2,
```

```
                       const Point& p3,
                       const Traits& traits = Traits())
        : tco( traits)
    {
        // allocate support points' array
        support_points = new Point[ 3];

        // store points
        points.push_back( p1);
        points.push_back( p2);
        points.push_back( p3);

        // compute mc
        mc( points.end(), 0);
    }
}
```

This macro is defined in definitions 7 and 8.
This macro is invoked in definition 2.

The destructor only frees the memory of the support points' array.

Min_circle_2 destructor [9] $\equiv$ {

```
    inline
    ~CGAL_Min_circle_2( )
    {
        // free support points' array
        delete[] support_points;
    }
}
```

This macro is invoked in definition 2.

4.1.4 Access Functions

These functions are used to retrieve information about the current status of the `CGAL_Min_circle_2<Traits>` object. They are all very simple (and therefore `inline`) and mostly rely on corresponding access functions of the data members.

First, we define the `number_of_...` methods.

Min_circle_2 access functions 'number_of_...' [10] $\equiv$ {

```
    // #points and #support points
    inline
    int
    number_of_points( ) const
    {
        return( points.size());
    }
```

```
    inline
    int
    number_of_support_points( ) const
    {
        return( n_support_points);
    }
}
```
This macro is invoked in definition 2.

Then, we have the access functions for points and support points.

Min_circle_2 access functions [11] $+\equiv$ {
```
    // access to points and support points
    inline
    Point_iterator
    points_begin( ) const
    {
        return( points.begin());
    }

    inline
    Point_iterator
    points_end( ) const
    {
        return( points.end());
    }

    inline
    Support_point_iterator
    support_points_begin( ) const
    {
        return( support_points);
    }

    inline
    Support_point_iterator
    support_points_end( ) const
    {
        return( support_points+n_support_points);
    }

    // random access for support points
    inline
    const Point&
    support_point( int i) const
    {
        CGAL_optimisation_precondition(
            ( i >= 0) && ( i <  number_of_support_points()));
```

```
        return( support_points[ i]);
    }
}
```

This macro is defined in definitions 11 and 12.
This macro is invoked in definition 2.

Finally, the access function `circle`.

Min_circle_2 access functions [12] $+\equiv$ {

```
    // circle
    inline
    const Circle&
    circle( ) const
    {
        return( tco.circle);
    }
}
```

This macro is defined in definitions 11 and 12.
This macro is invoked in definition 2.

4.1.5 Predicates

The predicates `is_empty` and `is_degenerate` are used in preconditions and postconditions of some member functions. Therefore we define them `inline` and put them in a separate macro.

Min_circle_2 predicates 'is_...' [13] $\equiv$ {

```
    // is_... predicates
    inline
    bool
    is_empty( ) const
    {
        return( number_of_support_points() == 0);
    }

    inline
    bool
    is_degenerate( ) const
    {
        return( number_of_support_points() <  2);
    }
}
```

This macro is invoked in definition 2.

The remaining predicates perform in-circle tests, based on the corresponding predicates of class `Circle`.

Min_circle_2 predicates [14] $\equiv$ {

```
// in-circle test predicates
inline
CGAL_Bounded_side
bounded_side( const Point& p) const
{
    return( tco.circle.bounded_side( p));
}

inline
bool
has_on_bounded_side( const Point& p) const
{
    return( tco.circle.has_on_bounded_side( p));
}

inline
bool
has_on_boundary( const Point& p) const
{
    return( tco.circle.has_on_boundary( p));
}

inline
bool
has_on_unbounded_side( const Point& p) const
{
    return( tco.circle.has_on_unbounded_side( p));
}
```

}

This macro is invoked in definition 2.

4.1.6 Modifiers

There is another way to build up $mc(P)$, other than by supplying the point set P at once. Namely, $mc(P)$ can be built up incrementally, adding one point after another. If you look at the pseudo-code in the introduction, this comes quite naturally. The modifying method `insert`, applied with point p to a `CGAL_Min_circle_2<Traits>` object representing $mc(P)$, computes $mc(P \cup \{p\})$, where work has to be done only if p lies outside $mc(P)$. In this case, $mc(P \cup \{p\}) = mc(P, \{p\})$ holds, so the private member function `mc` is called with support set $\{p\}$. After the insertion has been performed, p is moved to the front of the point list, just like in the pseudo-code in Section 2.

Min_circle_2 modifiers [15] $+\equiv$ {

```
void
insert( const Point& p)
{
```

```
        // p not in current circle?
        if ( has_on_unbounded_side( p)) {

            // p new support point
            support_points[ 0] = p;

            // recompute mc
            mc( points.end(), 1);

            // store p as the first point in list
            points.push_front( p); }
        else

            // append p to the end of the list
            points.push_back( p);
    }
}
```

This macro is defined in definitions 15, 16, and 17.
This macro is invoked in definition 2.

Inserting a range of points is done by a single member template. In case a compiler does not support this yet, we provide specialized `insert` functions for C arrays (using pointers as iterators), for STL sequence containers `vector<Point>` and `list<Point>` and for the STL input stream iterator `istream_iterator<Point>`. Actually, the `insert` function for a C array and a `vector<point>` are the same, since the random access iterator of `vector<Point>` is implemented as `Point*`.

The following `insert` functions perform a call `insert(p)` for each point `p` in the range [`first`,`last`).

Min_circle_2 modifiers [16] $+\equiv$ {

```
    #ifndef CGAL_CFG_NO_MEMBER_TEMPLATES

        template < class InputIterator >
        void
        insert( InputIterator first, InputIterator last)
        {
            for ( ; first != last; ++first)
                insert( *first);
        }

    #else

        inline
        void
        insert( const Point* first, const Point* last)
        {
            for ( ; first != last; ++first)
                insert( *first);
        }
```

```
        inline
        void
        insert( list<Point>::const_iterator first,
                list<Point>::const_iterator last )
        {
            for ( ; first != last; ++first)
                insert( *first);
        }

        inline
        void
        insert( istream_iterator<Point,ptrdiff_t>  first,
                istream_iterator<Point,ptrdiff_t>  last )
        {
            for ( ; first != last; ++first)
                insert( *first);
        }

    #endif // CGAL_CFG_NO_MEMBER_TEMPLATES
}
```

This macro is defined in definitions 15, 16, and 17.
This macro is invoked in definition 2.

The member function `clear` deletes all points from a `CGAL_Min_circle_2<Traits>` object and resets it to the empty circle.

Min_circle_2 modifiers [17] $+\equiv$ {

```
    void
    clear( )
    {
        points.erase( points.begin(), points.end());
        n_support_points = 0;
        tco.circle.set();
    }
}
```

This macro is defined in definitions 15, 16, and 17.
This macro is invoked in definition 2.

4.1.7 Validity Check

A `CGAL_Min_circle_2<Traits>` object can be checked for validity. This means, it is checked whether (a) the circle contains all points of its defining set P, (b) the circle is the smallest circle spanned by its support set, and (c) the support set is minimal, i.e. no support point is redundant. The function `is_valid` is mainly intended for debugging user supplied traits classes but also for convincing the anxious user that the traits class implementation is correct. If `verbose` is `true`, some messages concerning the performed checks are written to standard error stream. The second parameter `level` is not used, we provide it only for consistency with interfaces of other classes.

Min_circle_2 validity check [18] $\equiv$ {

```
bool
is_valid( bool verbose = false, int level = 0) const
{
    CGAL_Verbose_ostream verr( verbose);
    verr << endl;
    verr << "CGAL_Min_circle_2<Traits>::" << endl;
    verr << "is_valid( true, " << level << "):" << endl;
    verr << "  |P| = " << number_of_points()
         << ", |S| = " << number_of_support_points() << endl;

    // containment check (a)
    Min_circle_2 containment check [19]

    // support set checks (b)+(c)
    Min_circle_2 support set checks [20]

    verr << "  object is valid!" << endl;
    return( true);
}
```

}

This macro is invoked in definition 2.

The containment check (a) is easy to perform, just a loop over all points in `points`.

Min_circle_2 containment check [19] $\equiv$ {

```
verr << "  a) containment check..." << flush;
Point_iterator point_iter;
for ( point_iter  = points_begin();
      point_iter != points_end();
      ++point_iter)
    if ( has_on_unbounded_side( *point_iter))
        return( CGAL__optimisation_is_valid_fail( verr,
                    "circle does not contain all points"));
verr << "passed." << endl;
```

}

This macro is invoked in definition 18.

To check the support set (b) and (c), we distinguish four cases with respect to the number of support points, which may range from 0 to 3.

Min_circle_2 support set checks [20] $\equiv$ {

```
verr << "  b)+c) support set checks..." << flush;
switch( number_of_support_points()) {

  case 0:
    Min_circle_2 check no support point [21]
    break;
```

```
    case 1:
      Min_circle_2 check one support point [22]
      break;

    case 2: {
      Min_circle_2 check two support points [23]  }
      break;

    case 3: {
      Min_circle_2 check three support points [24]  }
      break;

    default:
      return( CGAL__optimisation_is_valid_fail( verr,
                    "illegal number of support points, \
                     not between 0 and 3."));
  };
  verr << "passed." << endl;
}
```

This macro is invoked in definition 18.

The case of no support point happens if and only if the defining point set P is empty.

Min_circle_2 check no support point [21] $\equiv$ {

```
  if ( ! is_empty())
      return( CGAL__optimisation_is_valid_fail( verr,
                    "P is nonempty, \
                     but there are no support points."));
}
```

This macro is invoked in definition 20.

If the smallest enclosing circle has one support point p, it must be equal to that point, i.e. its center must be p and its radius 0.

Min_circle_2 check one support point [22] $\equiv$ {

```
  if ( ( circle().center() != support_point( 0)    ) ||
       ( ! CGAL_is_zero( circle().squared_radius())) )
      return( CGAL__optimisation_is_valid_fail( verr,
                    "circle differs from the circle \
                     spanned by its single support point."));
}
```

This macro is invoked in definition 20.

In case of two support points p, q, these points must form a diameter of the circle. The support set $\{p, q\}$ is minimal if and only if p, q are distinct.

The diameter property is checked as follows. If p and q both lie on the circle's boundary and if p, q (knowing they are distinct) and the circle's center are collinear, then p and q form a diameter of the circle.

Min_circle_2 check two support points [23] ≡ {
```
    const Point& p( support_point( 0)),
                 q( support_point( 1));

    // p equals q?
    if ( p == q)
        return( CGAL__optimisation_is_valid_fail( verr,
                    "the two support points are equal."));

    // segment(p,q) is not diameter?
    if ( ( ! has_on_boundary( p)                              ) ||
         ( ! has_on_boundary( q)                              ) ||
         ( tco.orientation( p, q,
                            circle().center()) != CGAL_COLLINEAR) )
        return( CGAL__optimisation_is_valid_fail( verr,
                    "circle does not have its \
                     two support points as diameter."));
```
}
This macro is invoked in definition 20.

If the number of support points is three (and they are distinct and not collinear), the circle through them is unique, and must therefore equal the current circle stored in `circle`. It is the smallest one containing the three points if and only if the center of the circle lies inside or on the boundary of the triangle defined by the three points. The support set is minimal only if the center lies properly inside the triangle.

Both triangle properties are checked by comparing the orientations of three point triples, each containing two of the support points and the center of the current circle, resp. If one of these orientations equals `CGAL_COLLINEAR`, the center lies on the boundary of the triangle. Otherwise, if two triples have opposite orientations, the center is not contained in the triangle.

Min_circle_2 check three support points [24] ≡ {
```
    const Point& p( support_point( 0)),
                 q( support_point( 1)),
                 r( support_point( 2));

    // p, q, r not pairwise distinct?
    if ( ( p == q) || ( q == r) || ( r == p))
        return( CGAL__optimisation_is_valid_fail( verr,
                    "at least two of the three \
                     support points are equal."));

    // p, q, r collinear?
    if ( tco.orientation( p, q, r) == CGAL_COLLINEAR)
        return( CGAL__optimisation_is_valid_fail( verr,
                    "the three support points are collinear."));
```

```
    // current circle not equal the unique circle through p,q,r ?
    Circle c( circle());
    c.set( p, q, r);
    if ( circle() != c)
        return( CGAL__optimisation_is_valid_fail( verr,
                    "circle is not the unique circle \
                     through its three support points."));

    // circle's center on boundary of triangle(p,q,r)?
    const Point& center( circle().center());
    CGAL_Orientation o_pqz = tco.orientation( p, q, center);
    CGAL_Orientation o_qrz = tco.orientation( q, r, center);
    CGAL_Orientation o_rpz = tco.orientation( r, p, center);
    if ( ( o_pqz == CGAL_COLLINEAR) ||
         ( o_qrz == CGAL_COLLINEAR) ||
         ( o_rpz == CGAL_COLLINEAR) )
        return( CGAL__optimisation_is_valid_fail( verr,
                    "one of the three support points is redundant."));

    // circle's center not inside triangle(p,q,r)?
    if ( ( o_pqz != o_qrz) || ( o_qrz != o_rpz) || ( o_rpz != o_pqz))
        return( CGAL__optimisation_is_valid_fail( verr,
                    "circle's center is not in the \
                     convex hull of its three support points."));
}
```

This macro is invoked in definition 20.

4.1.8 Miscellaneous

The member function `traits` returns a const reference to the traits class object.

Min_circle_2 miscellaneous [25] $\equiv$ {

```
    inline
    const Traits&
    traits( ) const
    {
        return( tco);
    }
}
```

This macro is invoked in definition 2.

4.1.9 I/O

Min_circle_2 I/O operators declaration [26] $\equiv$ {

```
template < class _Traits >
ostream& operator << ( ostream& os,
                       const CGAL_Min_circle_2<_Traits>& mc);

template < class _Traits >
istream& operator >> ( istream& is,
                       CGAL_Min_circle_2<_Traits>& mc);
```

}

This macro is invoked in definition 59.

Min_circle_2 I/O operators [27] $\equiv$ {

```
template < class _Traits >
ostream&
operator << ( ostream& os,
              const CGAL_Min_circle_2<_Traits>& min_circle)
{
    typedef typename  CGAL_Min_circle_2<_Traits>::Point  Point;

    switch ( CGAL_get_mode( os)) {

      case CGAL_IO::PRETTY:
        os << endl;
        os << "CGAL_Min_circle_2( |P| = "
           << min_circle.number_of_points()
           << ", |S| = "
           << min_circle.number_of_support_points() << endl;
        os << "  P = {" << endl;
        os << "    ";
        copy( min_circle.points_begin(), min_circle.points_end(),
              ostream_iterator<Point>( os, ",\n    "));
        os << "}" << endl;
        os << "  S = {" << endl;
        os << "    ";
        copy( min_circle.support_points_begin(),
              min_circle.support_points_end(),
              ostream_iterator<Point>( os, ",\n    "));
        os << "}" << endl;
        os << "  circle = " << min_circle.circle() << endl;
        os << ")" << endl;
        break;

      case CGAL_IO::ASCII:
        copy( min_circle.points_begin(), min_circle.points_end(),
              ostream_iterator<Point>( os, "\n"));
```

```
          break;

        case CGAL_IO::BINARY:
          copy( min_circle.points_begin(), min_circle.points_end(),
                ostream_iterator<Point>( os));
          break;

        default:
          CGAL_optimisation_assertion_msg( false,
                                           "CGAL_IO::mode invalid!");
          break; }

      return( os);
  }

  template < class Traits >
  istream&
  operator >> ( istream& is, CGAL_Min_circle_2<Traits>& min_circle)
  {
      switch ( CGAL_get_mode( is)) {

        case CGAL_IO::PRETTY:
          cerr << endl;
          cerr << "Stream must be in ascii or binary mode" << endl;
          break;

        case CGAL_IO::ASCII:
        case CGAL_IO::BINARY:
          typedef typename  CGAL_Min_circle_2<Traits>::Point   Point;
          typedef           istream_iterator<Point,ptrdiff_t>  Is_it;
          min_circle.clear();
          min_circle.insert( Is_it( is), Is_it());
          break;

        default:
          CGAL_optimisation_assertion_msg( false,
                                           "CGAL_IO::mode invalid!");
          break; }

      return( is);
  }
}
```

This macro is invoked in definition 60.

4.1.10 Private Member Function `compute_circle`

This is the method for computing the basic case $\texttt{mc}(\emptyset, B)$, the set B given by the first `n_support_points` in the array `support_points`. It is realized by a simple case analysis, noting that $|B| \leq 3$.

Min_circle_2 private member function 'compute_circle' [28] $\equiv$ {

```
    // compute_circle
    inline
    void
    compute_circle( )
    {
        switch ( n_support_points) {
          case 3:
            tco.circle.set( support_points[ 0],
                            support_points[ 1],
                            support_points[ 2]);
            break;
          case 2:
            tco.circle.set( support_points[ 0], support_points[ 1]);
            break;
          case 1:
            tco.circle.set( support_points[ 0]);
            break;
          case 0:
            tco.circle.set( );
            break;
          default:
            CGAL_optimisation_assertion( ( n_support_points >= 0) &&
                                         ( n_support_points <= 3) ); }
    }
}
```

This macro is invoked in definition 2.

4.1.11 Private Member Function `mc`

This function computes the general circle $mc(P, B)$, where P contains the points in the range [`points.begin()`,`last`) and B is given by the first `n_sp` support points in the array `support_points`. The function is directly modeled after the pseudo-code above.

Min_circle_2 private member function 'mc' [29] $\equiv$ {

```
    void
    mc( const Point_iterator& last, int n_sp)
    {
        // compute circle through support points
        n_support_points = n_sp;
        compute_circle();
```

```
        if ( n_sp == 3) return;

        // test first n points
        list<Point>::iterator  point_iter( points.begin());
        for ( ; last != point_iter; ) {
            const Point& p( *point_iter);

            // p not in current circle?
            if ( has_on_unbounded_side( p)) {

                // recursive call with p as additional support point
                support_points[ n_sp] = p;
                mc( point_iter, n_sp+1);

                // move current point to front
                if ( point_iter != points.begin()) {    // p not first?
                    points.push_front( p);
                    points.erase( point_iter++); }
                else
                    ++point_iter; }
            else
                ++point_iter; }
    }
}
```

This macro is invoked in definition 2.

4.2 Class Template `CGAL_Optimisation_circle_2<R>`

First, we declare the class template `CGAL_Optimisation_circle_2`.

Optimisation_circle_2 declaration [30] $\equiv$ {

```
    template < class _R >
    class CGAL_Optimisation_circle_2;
}
```

This macro is invoked in definition 61.

Workaround: The GNU compiler (g++ 2.7.2[.x]) does not accept types with scope operator as argument type or return type in class template member functions. Therefore, all member functions are implemented in the class interface.

The class interface looks as follows.

Optimisation_circle_2 interface [31] $\equiv$ {

```
    template < class _R >
    class CGAL_Optimisation_circle_2 {
      public:
        Optimisation_circle_2 public interface [32]
```

```
    private:
      // private data members
      Optimisation_circle_2 private data members [33]

  dividing line [68]

  // Class implementation
  // ====================

    public:
      // Set functions
      // -------------
      Optimisation_circle_2 set functions [34]

      // Access functions
      // ----------------
      Optimisation_circle_2 access functions [35]

      // Equality tests
      // --------------
      Optimisation_circle_2 equality tests [36]

      // Predicates
      // ----------
      Optimisation_circle_2 predicates [37]
  };
}
```

This macro is invoked in definition 61.

4.2.1 Public Interface

The functionality is described and documented in the specification section, so we do not comment on it here.

Optimisation_circle_2 public interface [32] ≡ {

```
  // types
  typedef           _R                R;
  typedef           CGAL_Point_2<R>   Point;
  typedef typename  _R::FT            Distance;

  /**********************************************************************
  WORKAROUND: The GNU compiler (g++ 2.7.2[.x]) does not accept types
  with scope operator as argument type or return type in class template
  member functions. Therefore, all member functions are implemented in
  the class interface.

  // creation
  void  set( );
```

```
    void  set( const Point& p);
    void  set( const Point& p, const Point& q);
    void  set( const Point& p, const Point& q, const Point& r);
    void  set( const Point& center, const Distance& squared_radius);

    // access functions
    const Point&     center        ( ) const;
    const Distance&  squared_radius( ) const

    // equality tests
    bool  operator == ( const CGAL_Optimisation_circle_2<R>& c) const;
    bool  operator != ( const CGAL_Optimisation_circle_2<R>& c) const;

    // predicates
    CGAL_Bounded_side  bounded_side( const Point& p) const;
    bool  has_on_bounded_side      ( const Point& p) const;
    bool  has_on_boundary          ( const Point& p) const;
    bool  has_on_unbounded_side    ( const Point& p) const;

    bool  is_empty     ( ) const;
    bool  is_degenerate( ) const;
    *****************************************************************/
}
```

This macro is invoked in definition 31.

4.2.2 Private Data Members

The circle is represented by its center and squared radius.

Optimisation_circle_2 private data members [33] $\equiv$ {

```
    Point     _center;
    Distance  _squared_radius;
}
```

This macro is invoked in definition 31.

4.2.3 Set Functions

We provide set functions taking zero, one, two, or three boundary points and another set function taking a center point and a squared radius.

Optimisation_circle_2 set functions [34] $\equiv$ {

```
    inline
    void
    set( )
    {
        _center         =  Point( CGAL_ORIGIN);
```

```
        _squared_radius = -Distance( 1);
    }

    inline
    void
    set( const Point& p)
    {
        _center         = p;
        _squared_radius = Distance( 0);
    }

    inline
    void
    set( const Point& p, const Point& q)
    {
        _center         = CGAL_midpoint( p, q);
        _squared_radius = CGAL_squared_distance( p, _center);
    }

    inline
    void
    set( const Point& p, const Point& q, const Point& r)
    {
        _center         = CGAL_circumcenter( p, q, r);
        _squared_radius = CGAL_squared_distance( p, _center);
    }

    inline
    void
    set( const Point& center, const Distance& squared_radius)
    {
        _center         = center;
        _squared_radius = squared_radius;
    }
}
```

This macro is invoked in definition 31.

4.2.4 Access Functions

These functions are used to get the current center point or squared radius, resp.

Optimisation_circle_2 access functions [35] ≡ {

```
    inline
    const Point&
    center( ) const
    {
        return( _center);
    }
```

```
    inline
    const Distance&
    squared_radius( ) const
    {
        return( _squared_radius);
    }
}
```
This macro is invoked in definition 31.

4.2.5 Equality Tests

Optimisation_circle_2 equality tests [36] ≡ {
```
    bool
    operator == ( const CGAL_Optimisation_circle_2<R>& c) const
    {
        return( ( _center          == c._center        ) &&
                ( _squared_radius  == c._squared_radius) );
    }

    bool
    operator != ( const CGAL_Optimisation_circle_2<R>& c) const
    {
        return( ! operator==( c));
    }
}
```
This macro is invoked in definition 31.

4.2.6 Predicates

The following predicates perform in-circle tests and check for emptiness and degeneracy, resp.

Optimisation_circle_2 predicates [37] ≡ {
```
    inline
    CGAL_Bounded_side
    bounded_side( const Point& p) const
    {
        return( CGAL_static_cast( CGAL_Bounded_side,
                    CGAL_sign( CGAL_squared_distance( p, _center)
                               - _squared_radius)));
    }

    inline
    bool
    has_on_bounded_side( const Point& p) const
    {
```

```
        return( CGAL_squared_distance( p, _center) < _squared_radius);
    }

    inline
    bool
    has_on_boundary( const Point& p) const
    {
        return( CGAL_squared_distance( p, _center) == _squared_radius);
    }

    inline
    bool
    has_on_unbounded_side( const Point& p) const
    {
        return( _squared_radius < CGAL_squared_distance( p, _center));
    }

    inline
    bool
    is_empty( ) const
    {
        return( CGAL_is_negative( _squared_radius));
    }

    inline
    bool
    is_degenerate( ) const
    {
        return( ! CGAL_is_positive( _squared_radius));
    }
}
```
This macro is invoked in definition 31.

4.2.7 I/O

Optimisation_circle_2 I/O operators declaration [38] $\equiv$ {

```
    template < class _R >
    ostream&
    operator << ( ostream& os, const CGAL_Optimisation_circle_2<_R>& c);

    template < class _R >
    istream&
    operator >> ( istream& is, CGAL_Optimisation_circle_2<_R>      & c);
}
```
This macro is invoked in definition 61.

Optimisation_circle_2 I/O operators [39] ≡ {

```
template < class _R >
ostream&
operator << ( ostream& os, const CGAL_Optimisation_circle_2<_R>& c)
{
    switch ( CGAL_get_mode( os)) {

      case CGAL_IO::PRETTY:
        os << "CGAL_Optimisation_circle_2( "
           << c.center() << ", "
           << c.squared_radius() << ')';
        break;

      case CGAL_IO::ASCII:
        os << c.center() << ' ' << c.squared_radius();
        break;

      case CGAL_IO::BINARY:
        os << c.center();
        CGAL_write( os, c.squared_radius());
        break;

      default:
        CGAL_optimisation_assertion_msg( false,
                                         "CGAL_IO::mode invalid!");
        break; }

    return( os);
}

template < class _R >
istream&
operator >> ( istream& is, CGAL_Optimisation_circle_2<_R>& c)
{
    typedef typename  CGAL_Optimisation_circle_2<_R>::Point     Point;
    typedef typename  CGAL_Optimisation_circle_2<_R>::Distance
                                                              Distance;

    switch ( CGAL_get_mode( is)) {

      case CGAL_IO::PRETTY:
        cerr << endl;
        cerr << "Stream must be in ascii or binary mode" << endl;
        break;

      case CGAL_IO::ASCII: {
        Point     center;
```

```
          Distance  squared_radius;
          is >> center >> squared_radius;
          c.set( center, squared_radius); }
          break;

        case CGAL_IO::BINARY: {
          Point     center;
          Distance  squared_radius;
          is >> center;
          CGAL_read( is, squared_radius);
          c.set( center, squared_radius); }
          break;

        default:
          CGAL_optimisation_assertion_msg( false,
                                           "CGAL_IO::mode invalid!");
          break; }

      return( is);
  }
}
```

This macro is invoked in definition 62.

4.3 Class Template `CGAL_Min_circle_2_traits_2<R>`

First, we declare the class templates `CGAL_Min_circle_2_traits_2` and `CGAL_Min_circle_2`.

Min_circle_2_traits_2 declarations [40] $\equiv$ {

```
  template < class _Traits >
  class CGAL_Min_circle_2_traits_2;

  template < class _R >
  class CGAL_Min_circle_2;
}
```

This macro is invoked in definition 63.

Since the actual work of the traits class is done in the nested type `Circle`, we implement the whole class template in its interface.

The variable `circle` containing the current circle is declared `private` to disallow the user from directly accessing or modifying it. Since the algorithm needs to access and modify the current circle, it is declared `friend`.

Min_circle_2_traits_2 interface and implementation [41] $\equiv$ {

```
  template < class _R >
  class CGAL_Min_circle_2_traits_2 {
    public:
```

```
        // types
        typedef  _R                             R;
        typedef  CGAL_Point_2<R>                Point;
        typedef  CGAL_Optimisation_circle_2<R>  Circle;

    private:
        // data members
        Circle  circle;                                 // current circle

        // friends
        friend  class CGAL_Min_circle_2< CGAL_Min_circle_2_traits_2<R> >;

      public:
        // creation (use default implementations)
        // CGAL_Min_circle_2_traits_2( );
        // CGAL_Min_circle_2_traits_2(
        //     CGAL_Min_circle_2_traits_2<R> const&);

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

This macro is invoked in definition 63.

4.4 Class Template `CGAL_Min_circle_2_adapterC2<PT,DA>`

First, we declare the class templates `CGAL_Min_circle_2`, `CGAL_Min_circle_2_adapterC2` and `CGAL__Min_circle_2_adapterC2__Circle`.

Min_circle_2_adapterC2 declarations [42] $\equiv$ {

```
    template < class _Traits >
    class CGAL_Min_circle_2;

    template < class _PT, class _DA >
    class CGAL_Min_circle_2_adapterC2;

    template < class _PT, class _DA >
    class CGAL__Min_circle_2_adapterC2__Circle;
}
```

This macro is invoked in definition 64.

The actual work of the adapter is done in the nested class `Circle`. Therefore, we implement the whole adapter in its interface.

The variable `circle` containing the current circle is declared `private` to disallow the user from directly accessing or modifying it. Since the algorithm needs to access and modify the current circle, it is declared `friend`.

Min_circle_2_adapterC2 interface and implementation [43] ≡ {

```
template < class _PT, class _DA >
class CGAL_Min_circle_2_adapterC2 {
  public:
    // types
    typedef  _PT  PT;
    typedef  _DA  DA;

    // nested types
    typedef  PT                                            Point;
    typedef  CGAL__Min_circle_2_adapterC2__Circle<PT,DA>  Circle;

  private:
    DA      dao;                              // data accessor object
    Circle  circle;                           // current circle
    friend
        class CGAL_Min_circle_2< CGAL_Min_circle_2_adapterC2<PT,DA> >;

  public:
    // creation
    Min_circle_2_adapterC2 constructors [44]

    // operations
    Min_circle_2_adapterC2 operations [45]
};
```

}

This macro is invoked in definition 64.

4.4.1 Constructors

Min_circle_2_adapterC2 constructors [44] ≡ {

```
CGAL_Min_circle_2_adapterC2( const DA& da = DA())
    : dao( da), circle( da)
{ }
```

}

This macro is invoked in definition 43.

4.4.2 Operations

Min_circle_2_adapterC2 operations [45] ≡ {

```
CGAL_Orientation
orientation( const Point& p, const Point& q, const Point& r) const
```

```
    {
        typedef  typename _DA::FT  FT;

        FT  px;
        FT  py;
        FT  qx;
        FT  qy;
        FT  rx;
        FT  ry;

        dao.get( p, px, py);
        dao.get( q, qx, qy);
        dao.get( r, rx, ry);

        return( CGAL_static_cast( CGAL_Orientation,
                    CGAL_sign(    ( px-rx) * ( qy-ry)
                                - ( py-ry) * ( qx-rx))));
    }
}
```

This macro is invoked in definition 43.

4.4.3 Nested Type Circle

Min_circle_2_adapterC2 nested type 'Circle' [46] $\equiv$ {

```
    template < class _PT, class _DA >
    class CGAL__Min_circle_2_adapterC2__Circle {
      public:
        // typedefs
        typedef  _PT  PT;
        typedef  _DA  DA;

        typedef  typename _DA::FT  FT;

      private:
        // data members
        DA  dao;                                // data accessor object

        FT  center_x;                           // center's x-coordinate
        FT  center_y;                           // center's y-coordinate
        FT  sqr_rad;                            // squared radius

        // private member functions
        FT
        sqr_dist( const FT& px, const FT& py,
                  const FT& qx, const FT& qy) const
        {
            FT  dx( px - qx);
```

```
        FT  dy( py - qy);
        return( dx*dx + dy*dy);
    }

  public:
    // types
    typedef  PT  Point;
    typedef  FT  Distance;

    // creation
    CGAL__Min_circle_2_adapterC2__Circle( const DA& da) : dao( da) { }

    void  set( )
    {
        center_x =  FT( 0);
        center_y =  FT( 0);
        sqr_rad  = -FT( 1);
    }

    void  set( const Point& p)
    {
        dao.get( p, center_x, center_y);
        sqr_rad = FT( 0);
    }

    void  set( const Point& p, const Point& q)
    {
        FT  px;
        FT  py;
        FT  qx;
        FT  qy;

        dao.get( p, px, py);
        dao.get( q, qx, qy);

        center_x = ( px+qx) / FT( 2);
        center_y = ( py+qy) / FT( 2);
        sqr_rad  = sqr_dist( px, py, center_x, center_y);
    }

    void  set( const Point& p, const Point& q, const Point& r)
    {
        FT  px;
        FT  py;
        FT  qx;
        FT  qy;
        FT  rx;
```

```
        FT  ry;

        dao.get( p, px, py);
        dao.get( q, qx, qy);
        dao.get( r, rx, ry);

        FT  qx_px( qx - px);
        FT  qy_py( qy - py);
        FT  rx_px( rx - px);
        FT  ry_py( ry - py);

        FT  p2   ( px*px + py*py);
        FT  q2_p2( qx*qx + qy*qy - p2);
        FT  r2_p2( rx*rx + ry*ry - p2);
        FT  denom( ( qx_px*ry_py - rx_px*qy_py) * FT( 2));

        center_x = ( q2_p2*ry_py - r2_p2*qy_py) / denom;
        center_y = ( r2_p2*qx_px - q2_p2*rx_px) / denom;
        sqr_rad  = sqr_dist( px, py, center_x, center_y);
    }

    // predicates
    CGAL_Bounded_side
    bounded_side( const Point& p) const
    {
        FT  px;
        FT  py;
        dao.get( p, px, py);
        return( CGAL_static_cast( CGAL_Bounded_side, CGAL_sign(
            sqr_dist( px, py, center_x, center_y) - sqr_rad)));
    }

    bool
    has_on_bounded_side( const Point& p) const
    {
        FT  px;
        FT  py;
        dao.get( p, px, py);
        return( sqr_dist( px, py, center_x, center_y) < sqr_rad);
    }

    bool
    has_on_boundary( const Point& p) const
    {
        FT  px;
        FT  py;
        dao.get( p, px, py);
```

```
    return( sqr_dist( px, py, center_x, center_y) == sqr_rad);
}

bool
has_on_unbounded_side( const Point& p) const
{
    FT  px;
    FT  py;
    dao.get( p, px, py);
    return( sqr_rad < sqr_dist( px, py, center_x, center_y));
}

bool
is_empty( ) const
{
    return( CGAL_is_negative( sqr_rad));
}

bool
is_degenerate( ) const
{
    return( ! CGAL_is_positive( sqr_rad));
}

// additional operations for checking
bool
operator == (
    const CGAL__Min_circle_2_adapterC2__Circle<_PT,_DA>& c) const
{
    return( ( center_x == c.center_x) &&
            ( center_y == c.center_y) &&
            ( sqr_rad  == c.sqr_rad ) );
}

Point
center( ) const
{
    Point  p;
    dao.set( p, center_x, center_y);
    return( p);
}

const Distance&
squared_radius( ) const
{
    return( sqr_rad);
}
```

```
// I/O
friend
ostream&
operator << ( ostream& os,
    const CGAL__Min_circle_2_adapterC2__Circle<_PT,_DA>& c)
{
    switch ( CGAL_get_mode( os)) {

      case CGAL_IO::PRETTY:
        os << "CGAL_Min_circle_2_adapterC2::Circle( "
           << c.center_x << ", "
           << c.center_y << ", "
           << c.sqr_rad  << ')';
        break;

      case CGAL_IO::ASCII:
        os << c.center_x << ' ' << c.center_y << ' ' << c.sqr_rad;
        break;

      case CGAL_IO::BINARY:
        CGAL_write( os, c.center_x);
        CGAL_write( os, c.center_y);
        CGAL_write( os, c.sqr_rad);
        break;

      default:
        CGAL_optimisation_assertion_msg(
            false, "CGAL_IO::mode invalid!");
        break; }

    return( os);
}

friend
istream&
operator >> ( istream& is,
              CGAL__Min_circle_2_adapterC2__Circle<_PT,_DA>& c)
{
    switch ( CGAL_get_mode( is)) {

      case CGAL_IO::PRETTY:
        cerr << endl;
        cerr << "Stream must be in ascii or binary mode" << endl;
        break;

      case CGAL_IO::ASCII:
```

```
                is >> c.center_x >> c.center_y >> c.sqr_rad;
                break;

              case CGAL_IO::BINARY:
                CGAL_read( is, c.center_x);
                CGAL_read( is, c.center_y);
                CGAL_read( is, c.sqr_rad);
                break;

              default:
                CGAL_optimisation_assertion_msg(
                    false, "CGAL_IO::mode invalid!");
                break; }

            return( is);
        }
    };
}
```

This macro is invoked in definition 64.

4.5 Class Template `CGAL_Min_circle_2_adapterH2<PT,DA>`

First, we declare the class templates `Min_circle_2`, `CGAL_Min_circle_2_adapterH2` and `CGAL__Min_circle_2_adapterH2__Circle`.

Min_circle_2_adapterH2 declarations [47] ≡ {

```
    template < class _Traits >
    class CGAL_Min_circle_2;

    template < class _PT, class _DA >
    class CGAL_Min_circle_2_adapterH2;

    template < class _PT, class _DA >
    class CGAL__Min_circle_2_adapterH2__Circle;
}
```

This macro is invoked in definition 65.

The actual work of the adapter is done in the nested class `Circle`. Therefore, we implement the whole adapter in its interface.

The variable `circle` containing the current circle is declared `private` to disallow the user from directly accessing or modifying it. Since the algorithm needs to access and modify the current circle, it is declared `friend`.

Min_circle_2_adapterH2 interface and implementation [48] ≡ {

```
    template < class _PT, class _DA >
    class CGAL_Min_circle_2_adapterH2 {
      public:
```

```
        // types
        typedef  _PT  PT;
        typedef  _DA  DA;

        // nested types
        typedef  PT                                          Point;
        typedef  CGAL__Min_circle_2_adapterH2__Circle<PT,DA>  Circle;

      private:
        DA      dao;                            // data accessor object
        Circle  circle;                         // current circle
        friend
            class CGAL_Min_circle_2< CGAL_Min_circle_2_adapterH2<PT,DA> >;

      public:
        // creation
        Min_circle_2_adapterH2 constructors [49]

        // operations
        Min_circle_2_adapterH2 operations [50]
    };
}
```

This macro is invoked in definition 65.

4.5.1 Constructors

Min_circle_2_adapterH2 constructors [49] $\equiv$ {

```
    CGAL_Min_circle_2_adapterH2( const DA& da = DA())
        : dao( da), circle( da)
    { }
}
```

This macro is invoked in definition 48.

4.5.2 Operations

Min_circle_2_adapterH2 operations [50] $\equiv$ {

```
    CGAL_Orientation
    orientation( const Point& p, const Point& q, const Point& r) const
    {
        typedef  typename _DA::RT  RT;

        RT  phx;
        RT  phy;
        RT  phw;
        RT  qhx;
        RT  qhy;
```

```
        RT  qhw;
        RT  rhx;
        RT  rhy;
        RT  rhw;

        dao.get( p, phx, phy, phw);
        dao.get( q, qhx, qhy, qhw);
        dao.get( r, rhx, rhy, rhw);

        return( CGAL_static_cast( CGAL_Orientation,
            CGAL_sign(   ( phx*rhw - rhx*phw) * ( qhy*rhw - rhy*qhw)
                       - ( phy*rhw - rhy*phw) * ( qhx*rhw - rhx*qhw))));
    }
}
```

This macro is invoked in definition 48.

4.5.3 Nested Type `Circle`

Min_circle_2_adapterH2 nested type 'Circle' [51] ≡ {

```
    template < class _PT, class _DA >
    class CGAL__Min_circle_2_adapterH2__Circle {
      public:
        // typedefs
        typedef  _PT  PT;
        typedef  _DA  DA;

        typedef  typename _DA::RT            RT;
        typedef           CGAL_Quotient<RT>  FT;

      private:
        // data members
        DA  dao;                                // data accessor object

        RT  center_hx;                          // center's hx-coordinate
        RT  center_hy;                          // center's hy-coordinate
        RT  center_hw;                          // center's hw-coordinate
        FT  sqr_rad;                            // squared radius

        // private member functions
        FT
        sqr_dist( const RT& phx, const RT& phy, const RT& phw,
                  const RT& qhx, const RT& qhy, const RT& qhw) const
        {
            RT  dhx( phx*qhw - qhx*phw);
            RT  dhy( phy*qhw - qhy*phw);
            RT  dhw( phw*qhw);
            return( FT( dhx*dhx + dhy*dhy, dhw*dhw));
        }
```

```
public:
  // types
  typedef  PT  Point;
  typedef  FT  Distance;

  // creation
  CGAL__Min_circle_2_adapterH2__Circle( const DA& da) : dao( da) { }

  void  set( )
  {
      center_hx =  RT( 0);
      center_hy =  RT( 0);
      center_hw =  RT( 1);
      sqr_rad   = -FT( 1);
  }

  void  set( const Point& p)
  {
      dao.get( p, center_hx, center_hy, center_hw);
      sqr_rad = FT( 0);
  }

  void  set( const Point& p, const Point& q)
  {
      RT  phx;
      RT  phy;
      RT  phw;
      RT  qhx;
      RT  qhy;
      RT  qhw;

      dao.get( p, phx, phy, phw);
      dao.get( q, qhx, qhy, qhw);
      center_hx = ( phx*qhw + qhx*phw);
      center_hy = ( phy*qhw + qhy*phw);
      center_hw = ( phw*qhw * RT( 2));
      sqr_rad   = sqr_dist( phx, phy, phw,
                            center_hx, center_hy, center_hw);
  }

  void  set( const Point& p, const Point& q, const Point& r)
  {
      RT  phx;
      RT  phy;
      RT  phw;
      RT  qhx;
      RT  qhy;
```

```
    RT  qhw;
    RT  rhx;
    RT  rhy;
    RT  rhw;

    dao.get( p, phx, phy, phw);
    dao.get( q, qhx, qhy, qhw);
    dao.get( r, rhx, rhy, rhw);

    RT  qhx_phx( qhx*phw - phx*qhw);
    RT  qhy_phy( qhy*phw - phy*qhw);    // denominator: qhw*phw

    RT  rhx_phx( rhx*phw - phx*rhw);
    RT  rhy_phy( rhy*phw - phy*rhw);    // denominator: rhw*phw

    RT  phw2( phw*phw);
    RT  qhw2( qhw*qhw);
    RT  rhw2( rhw*rhw);

    RT  p2( phx*phx + phy*phy);         // denominator: phw2

    RT  q2_p2( ( qhx*qhx + qhy*qhy) * phw2 - p2 * qhw2);
                                        // denominator: qhw2*phw2

    RT  r2_p2( ( rhx*rhx + rhy*rhy) * phw2 - p2 * rhw2);
                                        // denominator: rhw2*phw2

    center_hx = q2_p2*rhy_phy * rhw - r2_p2*qhy_phy * qhw;
    center_hy = r2_p2*qhx_phx * qhw - q2_p2*rhx_phx * rhw;
    center_hw = ( qhx_phx*rhy_phy - rhx_phx*qhy_phy)
                  * phw*qhw*rhw * RT( 2);
    sqr_rad   = sqr_dist( phx, phy, phw,
                          center_hx, center_hy, center_hw);
}

// predicates
CGAL_Bounded_side
bounded_side( const Point& p) const
{
    RT  phx;
    RT  phy;
    RT  phw;
    dao.get( p, phx, phy, phw);
    return( CGAL_static_cast( CGAL_Bounded_side,
        CGAL_sign( sqr_dist( phx, phy, phw,
                             center_hx, center_hy, center_hw)
                   - sqr_rad)));
}
```

```
bool
has_on_bounded_side( const Point& p) const
{
    RT  phx;
    RT  phy;
    RT  phw;
    dao.get( p, phx, phy, phw);
    return( sqr_dist( phx, phy, phw,
                      center_hx, center_hy, center_hw) < sqr_rad);
}

bool
has_on_boundary( const Point& p) const
{
    RT  phx;
    RT  phy;
    RT  phw;
    dao.get( p, phx, phy, phw);
    return( sqr_dist( phx, phy, phw,
                center_hx, center_hy, center_hw) == sqr_rad);
}

bool
has_on_unbounded_side( const Point& p) const
{
    RT  phx;
    RT  phy;
    RT  phw;
    dao.get( p, phx, phy, phw);
    return( sqr_rad < sqr_dist( phx, phy, phw,
                                center_hx, center_hy, center_hw));
}

bool
is_empty( ) const
{
    return( CGAL_is_negative( sqr_rad));
}

bool
is_degenerate( ) const
{
    return( ! CGAL_is_positive( sqr_rad));
}

// additional operations for checking
bool
```

```
operator == (
    const CGAL__Min_circle_2_adapterH2__Circle<_PT,_DA>& c) const
{
    return( ( center_hx*c.center_hw == c.center_hx*center_hw) &&
            ( center_hy*c.center_hw == c.center_hy*center_hw) &&
            ( sqr_rad  == c.sqr_rad ) );
}

Point
center( ) const
{
    Point  p;
    dao.set( p, center_hx, center_hy, center_hw);
    return( p);
}

const Distance&
squared_radius( ) const
{
    return( sqr_rad);
}

// I/O
friend
ostream&
operator << ( ostream& os,
    const CGAL__Min_circle_2_adapterH2__Circle<_PT,_DA>& c)
{
    switch ( CGAL_get_mode( os)) {

      case CGAL_IO::PRETTY:
        os << "CGAL_Min_circle_2_adapterH2::Circle( "
           << c.center_hx << ", "
           << c.center_hy << ", "
           << c.center_hw << ", "
           << c.sqr_rad   << ')';
        break;

      case CGAL_IO::ASCII:
        os << c.center_hx << ' '
           << c.center_hy << ' '
           << c.center_hw << ' '
           << c.sqr_rad;
        break;

      case CGAL_IO::BINARY:
        CGAL_write( os, c.center_hx);
        CGAL_write( os, c.center_hy);
```

```
            CGAL_write( os, c.center_hw);
            CGAL_write( os, c.sqr_rad);
            break;

          default:
            CGAL_optimisation_assertion_msg(
                false, "CGAL_IO::mode invalid!");
            break; }

        return( os);
    }

    friend
    istream&
    operator >> ( istream& is,
                  CGAL__Min_circle_2_adapterH2__Circle<_PT,_DA>& c)
    {
        switch ( CGAL_get_mode( is)) {

          case CGAL_IO::PRETTY:
            cerr << endl;
            cerr << "Stream must be in ascii or binary mode" << endl;
            break;

          case CGAL_IO::ASCII:
            is >> c.center_hx
               >> c.center_hy
               >> c.center_hw
               >> c.sqr_rad;
            break;

          case CGAL_IO::BINARY:
            CGAL_read( is, c.center_hx);
            CGAL_read( is, c.center_hy);
            CGAL_read( is, c.center_hw);
            CGAL_read( is, c.sqr_rad);
            break;

          default:
            CGAL_optimisation_assertion_msg(
                false, "CGAL_IO::mode invalid!");
            break; }

        return( is);
    }
};
}
```
This macro is invoked in definition 65.

5 Tests

We test CGAL_Min_circle_2 with the traits class implementation based on the two-dimensional CGAL kernel, using exact arithmetic, i.e. Cartesian representation with number type CGAL_Quotient<CGAL_Gmpz> and homogeneous representation with number type CGAL_Gmpz.

Min_circle_2 test (includes and typedefs) [52] ≡ {

```
#include <CGAL/Cartesian.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Min_circle_2.h>
#include <CGAL/Min_circle_2_traits_2.h>
#include <CGAL/Min_circle_2_adapterC2.h>
#include <CGAL/Min_circle_2_adapterH2.h>
#include <CGAL/IO/Verbose_ostream.h>
#include <assert.h>
#include <string.h>
#include <fstream.h>

#ifdef CGAL_USE_LEDA_FOR_OPTIMISATION_TEST
#  include <CGAL/leda_integer.h>
   typedef  leda_integer                      Rt;
   typedef  CGAL_Quotient< leda_integer >     Ft;
#else
#  include <CGAL/Gmpz.h>
   typedef  CGAL_Gmpz                         Rt;
   typedef  CGAL_Quotient< CGAL_Gmpz >        Ft;
#endif

typedef  CGAL_Cartesian< Ft >                 RepC;
typedef  CGAL_Homogeneous< Rt >               RepH;
typedef  CGAL_Min_circle_2_traits_2< RepC >   TraitsC;
typedef  CGAL_Min_circle_2_traits_2< RepH >   TraitsH;
```

}

This macro is invoked in definition 66.

The command line option -verbose enables verbose output.

Min_circle_2 test (verbose option) [53] ≡ {

```
bool  verbose = false;
if ( ( argc > 1) && ( strcmp( argv[ 1], "-verbose") == 0)) {
    verbose = true;
    --argc;
    ++argv; }
```

}

This macro is invoked in definition 66.

5.1 Code Coverage

We call each function of class `CGAL_Min_circle_2<Traits>` at least once to ensure code coverage.

Min_circle_2 test (code coverage) [54] ≡ {

```
    cover_Min_circle_2( verbose, TraitsC(), Rt());
    cover_Min_circle_2( verbose, TraitsH(), Rt());
```

}

This macro is invoked in definition 66.

Min_circle_2 test (code coverage test function) [55] ≡ {

```
    template < class Traits, class RT >
    void
    cover_Min_circle_2( bool verbose, const Traits&, const RT&)
    {
        typedef  CGAL_Min_circle_2< Traits >  Min_circle;
        typedef  Min_circle::Point            Point;
        typedef  Min_circle::Circle           Circle;

        CGAL_Verbose_ostream verr( verbose);

        // generate 'n' points at random
        const int    n = 20;
        CGAL_Random  random_x, random_y;
        Point        random_points[ n];
        int          i;
        verr << n << " random points from [0,128)^2:" << endl;
        for ( i = 0; i < n; ++i)
            random_points[ i] = Point( RT( random_x( 128)),
                                       RT( random_y( 128)));
        if ( verbose)
            for ( i = 0; i < n; ++i)
                cerr << i << ": " << random_points[ i] << endl;

        // cover code
        verr << endl << "default constructor...";
        {
            Min_circle  mc;
            bool  is_valid = mc.is_valid( verbose);
            bool  is_empty = mc.is_empty();
            assert( is_valid);
            assert( is_empty);
        }

        verr << endl << "one point constructor...";
        {
```

```
    Min_circle  mc( random_points[ 0]);
    bool  is_valid      = mc.is_valid( verbose);
    bool  is_degenerate = mc.is_degenerate();
    assert( is_valid);
    assert( is_degenerate);
}

verr << endl << "two points constructor...";
{
    Min_circle  mc( random_points[ 1],
                    random_points[ 2]);
    bool  is_valid = mc.is_valid( verbose);
    assert( is_valid);
    assert( mc.number_of_points() == 2);
}

verr << endl << "three points constructor...";
{
    Min_circle  mc( random_points[ 3],
                    random_points[ 4],
                    random_points[ 5]);
    bool  is_valid = mc.is_valid( verbose);
    assert( is_valid);
    assert( mc.number_of_points() == 3);
}

verr << endl << "Point* constructor...";
Min_circle  mc( random_points, random_points+9);
{
    Min_circle  mc2( random_points, random_points+9, true);
    bool  is_valid  = mc .is_valid( verbose);
    bool  is_valid2 = mc2.is_valid( verbose);
    assert( is_valid);
    assert( is_valid2);
    assert( mc .number_of_points() == 9);
    assert( mc2.number_of_points() == 9);
    assert( mc.circle() == mc2.circle());
}

verr << endl << "list<Point>::const_iterator constructor...";
{
    Min_circle  mc1( mc.points_begin(), mc.points_end());
    Min_circle  mc2( mc.points_begin(), mc.points_end(), true);
    bool  is_valid1 = mc1.is_valid( verbose);
    bool  is_valid2 = mc2.is_valid( verbose);
    assert( is_valid1);
    assert( is_valid2);
```

```
        assert( mc1.number_of_points() == 9);
        assert( mc2.number_of_points() == 9);
        assert( mc.circle() == mc1.circle());
        assert( mc.circle() == mc2.circle());
    }

    verr << endl << "#points already called above.";

    verr << endl << "points access already called above.";

    verr << endl << "support points access...";
    {
        Point  support_point;
        Min_circle::Support_point_iterator
            iter( mc.support_points_begin());
        for ( i = 0; i < mc.number_of_support_points(); ++i, ++iter) {
            support_point = mc.support_point( i);
            assert( support_point == *iter); }
        Min_circle::Support_point_iterator
            end_iter( mc.support_points_end());
        assert( iter == end_iter);
    }

    verr << endl << "circle access already called above...";

    verr << endl << "in-circle predicates...";
    {
        Point              p;
        CGAL_Bounded_side  bounded_side;
        bool               has_on_bounded_side;
        bool               has_on_boundary;
        bool               has_on_unbounded_side;
        for ( i = 0; i < 9; ++i) {
            p = random_points[ i];
            bounded_side          = mc.bounded_side( p);
            has_on_bounded_side   = mc.has_on_bounded_side( p);
            has_on_boundary       = mc.has_on_boundary( p);
            has_on_unbounded_side = mc.has_on_unbounded_side( p);
        assert( bounded_side != CGAL_ON_UNBOUNDED_SIDE);
        assert( has_on_bounded_side || has_on_boundary);
        assert( ! has_on_unbounded_side); }
    }

    verr << endl << "is_... predicates already called above.";

    verr << endl << "single point insert...";
    mc.insert( random_points[ 9]);
```

```
{
    bool  is_valid = mc.is_valid( verbose);
    assert( is_valid);
    assert( mc.number_of_points() == 10);
}

verr << endl << "Point* insert...";
mc.insert( random_points+10, random_points+n);
{
    bool  is_valid = mc.is_valid( verbose);
    assert( is_valid);
    assert( mc.number_of_points() == n);
}

verr << endl << "list<Point>::const_iterator insert...";
{
    Min_circle  mc2;
    mc2.insert( mc.points_begin(), mc.points_end());
    bool  is_valid = mc2.is_valid( verbose);
    assert( is_valid);
    assert( mc2.number_of_points() == n);

    verr << endl << "clear...";
    mc2.clear();
          is_valid = mc2.is_valid( verbose);
    bool  is_empty = mc2.is_empty();
    assert( is_valid);
    assert( is_empty);
}

verr << endl << "validity check already called several times.";

verr << endl << "traits class access...";
{
    Traits  traits( mc.traits());
}

verr << endl << "I/O...";
{
    verr << endl << "  writing `test_Min_circle_2.ascii'...";
    ofstream os( "test_Min_circle_2.ascii");
    CGAL_set_ascii_mode( os);
    os << mc;
}
{
    verr << endl << "  writing `test_Min_circle_2.pretty'...";
    ofstream os( "test_Min_circle_2.pretty");
```

```
            CGAL_set_pretty_mode( os);
            os << mc;
        }
        {
            verr << endl << "  writing `test_Min_circle_2.binary'...";
            ofstream os( "test_Min_circle_2.binary");
            CGAL_set_binary_mode( os);
            os << mc;
        }
        {
            verr << endl << "  reading `test_Min_circle_2.ascii'...";
            Min_circle mc_in;
            ifstream is( "test_Min_circle_2.ascii");
            CGAL_set_ascii_mode( is);
            is >> mc_in;
            bool    is_valid = mc_in.is_valid( verbose);
            assert( is_valid);
            assert( mc_in.number_of_points() == n);
            assert( mc_in.circle() == mc.circle());
        }
        verr << endl;
    }
}
```

This macro is invoked in definition 66.

5.2 Traits Class Adapters

We define two point classes (one with Cartesian, one with homogeneous representation) and corresponding data accessors.

Min_circle_2 test (point classes) [56] $\equiv$ {

```
    // 2D Cartesian point class
    class MyPointC2 {
      public:
        typedef  ::Ft  FT;
      private:
        FT _x;
        FT _y;
      public:
        MyPointC2( ) { }
        MyPointC2( const FT& x, const FT& y) : _x( x), _y( y) { }

        const FT&  x( ) const { return( _x); }
        const FT&  y( ) const { return( _y); }

        bool
        operator == ( const MyPointC2& p) const
```

```
    {
        return( ( _x == p._x) && ( _y == p._y));
    }

    friend
    ostream&
    operator << ( ostream& os, const MyPointC2& p)
    {
        return( os << p._x << ' ' << p._y);
    }

    friend
    istream&
    operator >> ( istream& is, MyPointC2& p)
    {
        return( is >> p._x >> p._y);
    }
};

// 2D Cartesian point class data accessor
class MyPointC2DA {
  public:
    typedef  ::Ft  FT;

    const FT&  get_x( const MyPointC2& p) const { return( p.x()); }
    const FT&  get_y( const MyPointC2& p) const { return( p.y()); }

    void
    get( const MyPointC2& p, FT& x, FT& y) const
    {
        x = get_x( p);
        y = get_y( p);
    }

    void
    set( MyPointC2& p, const FT& x, const FT& y) const
    {
        p = MyPointC2( x, y);
    }
};

// 2D homogeneous point class
class MyPointH2 {
  public:
    typedef  ::Rt  RT;
  private:
    RT _hx;
```

```
    RT _hy;
    RT _hw;
  public:
    MyPointH2( ) { }
    MyPointH2( const RT& hx, const RT& hy, const RT& hw = RT( 1))
        : _hx( hx), _hy( hy), _hw( hw) { }

    const RT&  hx( ) const { return( _hx); }
    const RT&  hy( ) const { return( _hy); }
    const RT&  hw( ) const { return( _hw); }

    bool
    operator == ( const MyPointH2& p) const
    {
        return(    ( _hx*p._hw == p._hx*_hw)
                && ( _hy*p._hw == p._hy*_hw));
    }

    friend
    ostream&
    operator << ( ostream& os, const MyPointH2& p)
    {
        return( os << p._hx << ' ' << p._hy << ' ' << p._hw);
    }

    friend
    istream&
    operator >> ( istream& is, MyPointH2& p)
    {
        return( is >> p._hx >> p._hy >> p._hw);
    }
};

// 2D homogeneous point class data accessor
class MyPointH2DA {
  public:
    typedef  ::Rt  RT;

    const RT&  get_hx( const MyPointH2& p) const { return( p.hx()); }
    const RT&  get_hy( const MyPointH2& p) const { return( p.hy()); }
    const RT&  get_hw( const MyPointH2& p) const { return( p.hw()); }

    void
    get( const MyPointH2& p, RT& hx, RT& hy, RT& hw) const
    {
        hx = get_hx( p);
        hy = get_hy( p);
```

```
            hw = get_hw( p);
        }

        void
        set( MyPointH2& p, const RT& hx, const RT& hy, const RT& hw) const
        {
            p = MyPointH2( hx, hy, hw);
        }
    };
}
```

This macro is invoked in definition 66.

To test the traits class adapters we use the code coverage test function.

Min_circle_2 test (adapters test) [57] ≡ {

```
    typedef  CGAL_Min_circle_2_adapterC2<MyPointC2,MyPointC2DA> AdapterC2;
    typedef  CGAL_Min_circle_2_adapterH2<MyPointH2,MyPointH2DA> AdapterH2;
    cover_Min_circle_2( verbose, AdapterC2(), Rt());
    cover_Min_circle_2( verbose, AdapterH2(), Rt());
}
```

This macro is invoked in definition 66.

5.3 External Test Sets

In addition, some data files can be given as command line arguments. A data file contains pairs of `int`s, namely the x- and y-coordinates of a set of points. The first number in the file is the number of points. A short description of the test set is given at the end of each file.

Min_circle_2 test (external test sets) [58] ≡ {

```
    while ( argc > 1) {

        typedef  CGAL_Min_circle_2< TraitsH >  Min_circle;
        typedef  Min_circle::Point             Point;
        typedef  Min_circle::Circle            Circle;

        CGAL_Verbose_ostream verr( verbose);

        // read points from file
        verr << endl << "input file: `" << argv[ 1] << "'" << flush;

        list<Point>  points;
        int          n, x, y;
        ifstream     in( argv[ 1]);
        in >> n;
        assert( in);
        for ( int i = 0; i < n; ++i) {
```

```
            in >> x >> y;
            assert( in);
            points.push_back( Point( x, y)); }

        // compute and check min_circle
        Min_circle  mc2( points.begin(), points.end());
        bool  is_valid = mc2.is_valid( verbose);
        assert( is_valid);

        // next file
        --argc;
        ++argv; }
}
```
This macro is invoked in definition 66.

6 Files

6.1 Min_circle_2.h

include/CGAL/Min_circle_2.h [59] $\equiv$ {

Min_circle_2 header [70] ('`include/CGAL/Min_circle_2.h`')

```
#ifndef CGAL_MIN_CIRCLE_2_H
#define CGAL_MIN_CIRCLE_2_H

// Class declaration
// =================
```
Min_circle_2 declaration [1]

```
// Class interface
// ===============
// includes
#ifndef CGAL_RANDOM_H
#  include <CGAL/Random.h>
#endif
#ifndef CGAL_OPTIMISATION_ASSERTIONS_H
#  include <CGAL/optimisation_assertions.h>
#endif
#ifndef CGAL_OPTIMISATION_BASIC_H
#  include <CGAL/optimisation_basic.h>
#endif
#ifndef CGAL_PROTECT_LIST_H
#  include <list.h>
#endif
#ifndef CGAL_PROTECT_VECTOR_H
#include <vector.h>
```

```
#endif
#ifndef CGAL_PROTECT_ALGO_H
#include <algo.h>
#endif
#ifndef CGAL_PROTECT_IOSTREAM_H
#include <iostream.h>
#endif
```

Min_circle_2 interface [2]

```
// Function declarations
// =====================
// I/O
// ---
```

Min_circle_2 I/O operators declaration [26]

```
#ifdef CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION
#  include <CGAL/Min_circle_2.C>
#endif

#endif // CGAL_MIN_CIRCLE_2_H
```

end of file line [69]

}

This macro is attached to an output file.

6.2 Min_circle_2.C

include/CGAL/Min_circle_2.C [60] $\equiv$ {

Min_circle_2 header [70] (‘include/CGAL/Min_circle_2.C’)

```
// Class implementation (continued)
// ================================
// I/O
// ---
```

Min_circle_2 I/O operators [27]

end of file line [69]

}

This macro is attached to an output file.

6.3 Optimisation_circle_2.h

include/CGAL/Optimisation_circle_2.h [61] $\equiv$ {

Optimisation_circle_2 header [71] (‘include/CGAL/Optimisation_circle_2.h’)

```
#ifndef CGAL_OPTIMISATION_CIRCLE_2_H
#define CGAL_OPTIMISATION_CIRCLE_2_H

// Class declaration
// =================
```
Optimisation_circle_2 declaration [30]

```
// Class interface
// ===============
// includes
#ifndef CGAL_POINT_2_H
#  include <CGAL/Point_2.h>
#endif
#ifndef CGAL_BASIC_CONSTRUCTIONS_2_H
#  include <CGAL/basic_constructions_2.h>
#endif
#ifndef CGAL_SQUARED_DISTANCE_2_H
#  include <CGAL/squared_distance_2.h>
#endif
```

Optimisation_circle_2 interface [31]

```
// Function declarations
// =====================
// I/O
// ---
```
Optimisation_circle_2 I/O operators declaration [38]

```
#ifdef CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION
#  include <CGAL/Optimisation_circle_2.C>
#endif

#endif // CGAL_OPTIMISATION_CIRCLE_2_H
```

end of file line [69]

}

This macro is attached to an output file.

6.4 Optimisation_circle_2.C

include/CGAL/Optimisation_circle_2.C [62] $\equiv$ {

Optimisation_circle_2 header [71] (‘`include/CGAL/Optimisation_circle_2.C`’)

```
// Class implementation (continued)
// ================================
// includes
#ifndef CGAL_OPTIMISATION_ASSERTIONS_H
```

```
    #  include <CGAL/optimisation_assertions.h>
    #endif

    // I/O
    // ---
```
Optimisation_circle_2 I/O operators [39]

end of file line [69]

}

This macro is attached to an output file.

6.5 Min_circle_2_traits_2.h

include/CGAL/Min_circle_2_traits_2.h [63] ≡ {

Min_circle_2 header [70] (‘`include/CGAL/Min_circle_2_traits_2.h`’)

```
    #ifndef CGAL_MIN_CIRCLE_2_TRAITS_2_H
    #define CGAL_MIN_CIRCLE_2_TRAITS_2_H

    // Class declarations
    // ==================
```
Min_circle_2_traits_2 declarations [40]

```
    // Class interface and implementation
    // ==================================
    // includes
    #ifndef CGAL_POINT_2_H
    #  include <CGAL/Point_2.h>
    #endif
    #ifndef CGAL_OPTIMISATION_CIRCLE_2_H
    #  include <CGAL/Optimisation_circle_2.h>
    #endif
    #ifndef CGAL_PREDICATES_ON_POINTS_2_H
    #  include <CGAL/predicates_on_points_2.h>
    #endif
```
Min_circle_2_traits_2 interface and implementation [41]

```
    #endif // CGAL_MIN_CIRCLE_2_TRAITS_2_H
```
end of file line [69]

}

This macro is attached to an output file.

6.6 Min_circle_2_adapterC2.h

include/CGAL/Min_circle_2_adapterC2.h [64] ≡ {

Min_circle_2 header [70] (‘`include/CGAL/Min_circle_2_adapterC2.h`’)

```
#ifndef CGAL_MIN_CIRCLE_2_ADAPTERC2_H
#define CGAL_MIN_CIRCLE_2_ADAPTERC2_H

// Class declarations
// ==================
```

Min_circle_2_adapterC2 declarations [42]

```
// Class interface and implementation
// ==================================
// includes
#ifndef CGAL_BASIC_H
#  include <CGAL/basic.h>
#endif
#ifndef CGAL_OPTIMISATION_ASSERTIONS_H
#  include <CGAL/optimisation_assertions.h>
#endif
```

Min_circle_2_adapterC2 interface and implementation [43]

```
// Nested type ‘Circle’
```

Min_circle_2_adapterC2 nested type ‘Circle’ [46]

```
#endif // CGAL_MIN_CIRCLE_2_ADAPTERC2_H
```

end of file line [69]

}

This macro is attached to an output file.

6.7 Min_circle_2_adapterH2.h

include/CGAL/Min_circle_2_adapterH2.h [65] ≡ {

Min_circle_2 header [70] (‘`include/CGAL/Min_circle_2_adapterH2.h`’)

```
#ifndef CGAL_MIN_CIRCLE_2_ADAPTERH2_H
#define CGAL_MIN_CIRCLE_2_ADAPTERH2_H

// Class declarations
// ==================
```

Min_circle_2_adapterH2 declarations [47]

```
// Class interface and implementation
// ==================================
```

```
// includes
#ifndef CGAL_BASIC_H
#  include <CGAL/basic.h>
#endif
#ifndef CGAL_OPTIMISATION_ASSERTIONS_H
#  include <CGAL/optimisation_assertions.h>
#endif
```

Min_circle_2_adapterH2 interface and implementation [48]

```
// Nested type 'Circle'
```
Min_circle_2_adapterH2 nested type 'Circle' [51]

```
#endif // CGAL_MIN_CIRCLE_2_ADAPTERH2_H
```

end of file line [69]

}

This macro is attached to an output file.

6.8 test_Min_circle_2.C

test/Optimisation/test_Min_circle_2.C [66] $\equiv$ {

Min_circle_2 header [70] ('test/optimisation/test_Min_circle_2.C')

Min_circle_2 test (includes and typedefs) [52]

```
// code coverage test function
// ---------------------------
```
Min_circle_2 test (code coverage test function) [55]

```
// point classes for adapters test
// -------------------------------
```
Min_circle_2 test (point classes) [56]

```
// main
// ----
int
main( int argc, char* argv[])
{
    // command line options
    // --------------------
    // option '-verbose'
```
Min_circle_2 test (verbose option) [53]

```
    // code coverage
    // -------------
```
Min_circle_2 test (code coverage) [54]

```
        // adapters test
        // -------------
```
Min_circle_2 test (adapters test) [57]

```
        // external test sets
        // ------------------
```
Min_circle_2 test (external test sets) [58]

```
        return( 0);
    }
```

end of file line [69]

}

This macro is attached to an output file.

File Header

A formatted file header allows easy identification of the files. It is parameterized with the title of the implementation, the product file name, the source file name, the author name, and the RCS variables *Revision* and *Date* of the source file.

file header [67] (◇9)**ZM** ≡ {

```
// ============================================================================
//
// Copyright (c) 1997,1998 The CGAL Consortium
//
// This software and related documentation is part of an INTERNAL
// release of the Computational Geometry Algorithms Library (CGAL).
// It is not intended for general use.
//
// ----------------------------------------------------------------------------
//
// release       : $CGAL_Revision: CGAL-wip $
// release_date  : $CGAL_Date$
//
// file          : ◇2
// source        : web/◇4.aw
// revision      : ◇8
// revision_date : ◇9
// package       : $CGAL_Package: ◇3 WIP $
// author(s)     : ◇5
//                 ◇6
//
// coordinator   : ◇7
//
// implementation: ◇1
```

```
    // ==================================================================
}
```

This macro is invoked in definitions 70 and 71.

dividing line [68] **ZM** $\equiv$ {

```
    // =============================================================
}
```

This macro is invoked in definitions 2 and 31.

end of file line [69] **ZM** $\equiv$ {

```
    // ===== EOF ===================================================
}
```

This macro is invoked in definitions 59, 60, 61, 62, 63, 64, 65, and 66.

Min_circle_2 header [70] ($\diamond$1)**M** $\equiv$ {

file header [67] (‘2D Smallest Enclosing Circle’,‘$\diamond$1’,
‘Optimisation’,‘Optimisation/Min_circle_2’,
‘Sven Schönherr <sven@inf.fu-berlin.de>’,
‘Bernd Gärtner’,
‘ETH Zurich (Bernd Gärtner <gaertner@inf.ethz.ch>)’,
‘$Revision: 4.1 $’,‘$Date: 1998/03/30 14:21:06 $’)

}

This macro is invoked in definitions 59, 60, 63, 64, 65, and 66.

Optimisation_circle_2 header [71] ($\diamond$1)**M** $\equiv$ {

file header [67] (‘2D Optimisation Circle’,‘$\diamond$1’,
‘Optimisation’,‘Optimisation/Min_circle_2’,
‘Sven Schönherr <sven@inf.fu-berlin.de>’,
‘Bernd Gärtner’,
‘ETH Zurich (Bernd Gärtner <gaertner@inf.ethz.ch>)’,
‘$Revision: 4.1 $’,‘$Date: 1998/03/30 14:21:06 $’)

}

This macro is invoked in definitions 61 and 62.

References

[1] The CGAL project. URL `http://www.cs.inf.ruu.nl/CGAL/`.

[2] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL kernel: A basis for geometric computation. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry – Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 191–202. Springer Verlag, 1996.

[3] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the Computational Geometry Algorithms Library. Research Report MPI-I-98-1-007, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, Feb. 1998. URL `http://data.mpi-sb.mpg.de/reports/`.

[4] B. Gärtner, M. Hoffmann, and S. Schönherr. Geometric Optimisation. In H. Brönnimann, S. Schirra, and R. Veltkamp, editors, *CGAL Reference Manual, Part 2: Basic Library*. 1998. CGAL R1.0. URL `http://www.cs.ruu.nl/CGAL`. to appear.

[5] B. Gärtner and S. Schönherr. Smallest enclosing ellipses – an exact and generic implementation. Serie B – Informatik B 98-04, Freie Universität Berlin, Germany, Apr. 1998. URL `http://www.inf.fu-berlin.de/inst/pubs/`.

[6] D. Kühl and K. Weihe. Data access templates. *C++ Report*, June 1997.

[7] S. Meyers. *Effective C++*. Addison-Wesley, 1992.

[8] S. Meyers. *More Effective C++*. Addison-Wesley, 1996.

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

[10] A. Stepanov and M. Lee. The Standard Template Library, Oct. 1995. URL `http://www.cs.rpi.edu/~musser/doc.ps`.

[11] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and New Trends in Computer Science*, volume 555 of *Lecture Notes in Computer Science*, pages 359–370. Springer Verlag, 1991.

Contents