SERIE B — INFORMATIK

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

Bernd Gärtner†
Sven Schönherr‡

B 98-05
April 1998

Abstract

We present a C++ implementation of an optimisation algorithm for computing the smallest (w.r.t. area) enclosing ellipse of a finite point set in the plane. We obtain an exact solution by using Welzl's method [14] together with the primitives as described in [6, 7]. The algorithm is implemented as a semi-dynamic data structure, thus allowing to insert points while maintaining the smallest enclosing ellipse. 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 ellipse of a finite point set in the plane. We implement the algorithm of Welzl, with move-to-front heuristic [14], using the primitives as described in [6, 7], resulting in an exact solution. The algorithm is realized as a semi-dynamic data structure, thus allowing to insert points while maintaining the smallest enclosing ellipse. It is parameterized with a traits class [4], that defines the abstract interface between the optimisation algorithm and the primitives it uses. We provide a traits class implementation using the CGAL kernel [3] and, for ease-of-use, traits class adapters to user supplied point classes. The interface of the data structure is compliant with the STL [13, 12].

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 gives a brief introduction to conics. Section 4 contains the specifications as they appear in the CGAL Reference Manual [5]. Section 5 gives the implementations. In Section 6 we provide a test program which performs some correctness checks. Finally the code files are created in Section 7. For a more detailed overview, see the table of contents starting on page 148.

2 The Algorithm

The implementation is based on an algorithm by Welzl [14], which we shortly describe now. The smallest (w.r.t. area) enclosing ellipse of a finite point set P in the plane, denoted by $me(P)$, is built up incrementally, adding one point after another. Assume $me(P)$ has been constructed, and we would like to obtain $me(P \cup \{p\})$, p some new point. There are two cases: if p already lies inside $me(P)$, then $me(P \cup \{p\}) = me(P)$. Otherwise p must lie on the boundary of $me(P \cup \{p\})$ (this is proved in [14] and not hard to see), so we need to compute $me(P, \{p\})$, the smallest ellipse enclosing P with p on the boundary. This is recursively done in the same manner. In general, for point sets P,B, define $me(P, B)$ as the smallest ellipse enclosing P that has the points of B on the boundary (if defined). Although the algorithm finally delivers a ellipse $me(P, \emptyset)$, it internally deals with ellipses that have a possibly nonempty set B. Here is the pseudo-code of Welzl's method. To compute $me(P)$, it is called with the pair $(P, \emptyset)$, assuming that $P = \{p_1, \ldots, p_n\}$ is stored in a linked list.

```
me(P, B):
    me := me(∅, B)
    IF |B| = 5 THEN RETURN me
    FOR i := 1 TO n DO
        IF p_i ∉ me THEN
            me := me({p_1, ..., p_{i-1}}, B ∪ {p_i})
            move p_i to the front of P
        END
    END
    RETURN me
```

Note the following: (a) $|B|$ is always bounded by 5, thus the computation of $me(\emptyset, B)$ is easy. In our implementation, it is done by the private member function *compute_ellipse*. (b) One can check that the method maintains the invariant '$me(P, B)$ exists'. This justifies termination if $|B| = 5$, because then $me(P, B)$ must be the unique ellipse with the points of B on the boundary, and $me(P, B)$ exists if and only if this ellipse contains the points of P. Thus, no subsequent in-ellipse tests are necessary anymore (for details see [14]). (c) points which are found to lie outside the current ellipse *me* 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 Conics

For a given real vector $\mathcal{R} = (r, s, t, u, v, w)$, a conic $\mathcal{C} = \mathcal{C}(\mathcal{R})$ is the set of *homogeneous* points $p = (x, y, h), h \neq 0$ satisfying

$$\mathcal{R}(p) := rx^2 + sy^2 + txy + uxh + vyh + wh^2 = 0, \tag{1}$$

equivalently

$$(x, y, h) \begin{pmatrix} 2r & t & u \\ t & 2s & v \\ u & v & 2w \end{pmatrix} \begin{pmatrix} x \\ y \\ h \end{pmatrix} = 0. \tag{2}$$

$\mathcal{R}$ is called a *representation* of $\mathcal{C}$. Note that the homogeneous point (x, y, h) corresponds to the Cartesian point $(x/h, y/h)$ in the plane. Also, any Cartesian point $p = (x, y)$ can be identified with the homogeneous point $(x, y, 1)$, in which case (1) assumes the form

$$\mathcal{R}(p) = rx^2 + sy^2 + txy + ux + vy + w = 0, \tag{3}$$

equivalently

$$(x, y) \begin{pmatrix} 2r & t \\ t & 2s \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + (2u, 2v) \begin{pmatrix} x \\ y \end{pmatrix} + 2w = 0. \tag{4}$$

Thus, under the condition $h \neq 0$, homogeneous and Cartesian representation are equivalent, and we frequently switch between them. [1]

$\mathcal{C}$ is called *trivial* if $\mathcal{C} = \mathbb{R}^2$ which is equivalent to $\mathcal{R} = 0$. $\mathcal{C}$ is *empty* if $\mathcal{C} = \emptyset$, i.e. if (1) has no real solutions x, y, h with $h \neq 0$ (which happens e.g. in case of $\mathcal{R} = (1, 1, 0, 0, 0, 1)$, or $\mathcal{R} = (0, 0, 0, 0, 0, 1)$).

[1] The reason for scaling equations (2) and (4) by a factor of 2 is purely technical – we want to argue with division-free terms.

$\mathcal{C}$ is invariant under scaling its representation $\mathcal{R}$ by any nonzero factor. This means, a conic has five degrees of freedom, and in fact it holds that any five points uniquely determine a nontrivial conic passing through the points. Some care is in place: this does *not* mean that five points uniquely determine the conic's representation, up to scaling. For example, the x-axis is a conic uniquely determined by any five points on it, but as a representation we may choose $\{y = 0\}$ or $\{y^2 = 0\}$. Recall that we have already seen two representations of the empty conic which are not multiples of each other.

3.1 Conic Types

The number

$$\det(\mathcal{R}) := \det \begin{pmatrix} 2r & t \\ t & 2s \end{pmatrix} \tag{5}$$

determines the type of $\mathcal{C}(\mathcal{R})$. If $\det(\mathcal{R}) > 0$, $\mathcal{C}$ is an *ellipse*, if $\det(\mathcal{R}) < 0$, we get a *hyperbola*, and for $\det(\mathcal{R}) = 0$, a *parabola* is obtained. While the trivial conic is a degenerate parabola equal to the whole plane, any nontrivial one consists of at most two simple curves. As a special case, there is a conic $\mathcal{C} = \{p\}$ for any point $p = (x_0, y_0)$. It can be specified as

$$\mathcal{C} = \{(x, y) \mid (x - x_0)^2 + (y - y_0)^2 = 0\},$$

and a possible representation is $\mathcal{R} = (1, 1, 0, -2x_0, -2y_0, x_0^2 + y_0^2)$. This implies $\det(\mathcal{R}) = 4$, so $\mathcal{C}$ is a degenerate ellipse, see Section 3.3.

Note that $\det(\mathcal{R}) > 0$ implies $r, s > 0$ or $r, s < 0$ which is equivalent to

$$M := \begin{pmatrix} 2r & t \\ t & 2s \end{pmatrix} \tag{6}$$

being positive definite ($x^T M x > 0$ for $x \neq 0$) or negative definite ($x^T M x < 0$ for $x \neq 0$). In case of $\det(\mathcal{R}) < 0$, M is indefinite, meaning that $x^T M x$ assumes positive and negative values.

3.2 Symmetry Properties

If the conic $\mathcal{C}(\mathcal{R})$ is not a parabola, the matrix

$$M = \begin{pmatrix} 2r & t \\ t & 2s \end{pmatrix}$$

is regular, and $\mathcal{C}$ has a unique center of symmetry c, given as

$$c = -M^{-1} \begin{pmatrix} u \\ v \end{pmatrix}.$$

With this definition, (4) can alternatively been written as

$$(p - c)^T M (p - c) + 2w - c^T M c = 0, \tag{7}$$

$p = (x, y)^T$, from which the symmetry is obvious. In case of a parabola, we get an axis of symmetry.

3.3 Orientation and Degeneracy

An *oriented conic* is a pair $\mathcal{C}_\mathcal{R} = (\mathcal{C}(\mathcal{R}), \mathcal{R})$, i.e. a conic with a particular representation. $\mathcal{C}_\mathcal{R}$ subdivides $\mathbb{R}^2 \backslash \mathcal{C}(\mathcal{R})$ into a *positive* side, formed by the set of points such that $\mathcal{R}(p) > 0$, and a *negative* side ($\mathcal{R}(p) < 0$). Replacing $\mathcal{R}$ with $-\mathcal{R}$ leads to an oriented conic with positive and negative sides interchanged. This concept of assigning positive and negative sides is purely algebraic.

In addition, there is a geometric way of assigning sides to an oriented conic which does not depend on $\mathcal{R}$ (like an oriented circle has a bounded side, independent from its orientation). To this end, we define the *convex side* of a conic $\mathcal{C}$ as the the union of the convex connected components of $\mathbb{R}^2 \setminus \mathcal{C}$. The *non-convex side* is then just the union of the non-convex components. Figure 1 depicts the convex sides of an ellipse, hyperbola and parabola, labeled with the letter 'c'.

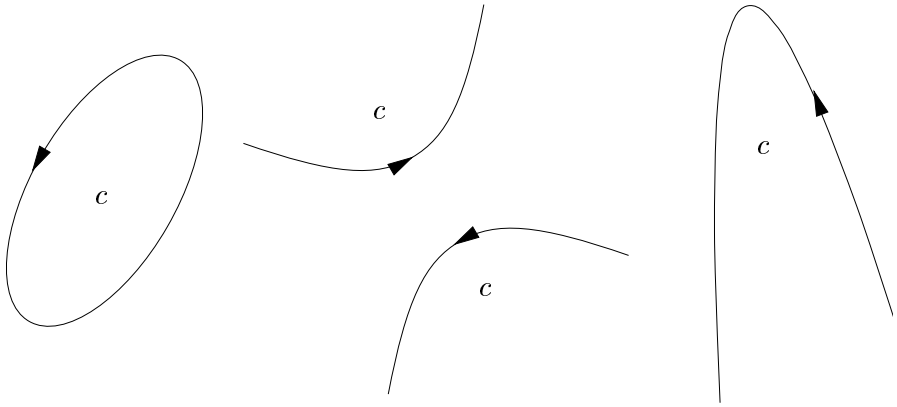

Figure 1: Ellipse, hyperbola and parabola with convex sides

A conic is defined to be *degenerate* if either its convex side or its non-convex side is empty. Let us discuss the possibilities for this. First, the trivial conic is degenerate, with both sides being empty. The empty conic is degenerate, with the non-convex side being empty. An ellipse is degenerate if and only if it consists of just one point (and so the convex side is empty). A hyperbola is degenerate if and only if it contains its center of symmetry c. In this case, the hyperbola is a pair of lines crossing at c, and so the non-convex side is empty. A degenerate parabola is either a pair of parallel lines or just one line. In both cases, the non-convex side is empty.

In the non-degenerate case, the classification of points by positive and negative side coincides with the one by convex and non-convex side. In the degenerate case, this exactly holds if positive or negative side disappear, like for a degenerate ellipse (but not for a degenerate hyperbola).

An oriented conic $\mathcal{C}_\mathcal{R}$ is said to have positive (negative) orientation, if and only if the convex side coincides with the positive (negative) side. If neither is the case, the orientation is zero. Thus, a degenerate ellipse has nonzero orientation, but a degenerate hyperbola has not.

While it is clear that positive and negative side of a conic are only defined with re-

spect to some representation, it is interesting to note that even the partition of $\mathbb{R}^2 \setminus \mathcal{C}_\mathcal{R}$ into positive and negative side does in general depend on $\mathcal{R}$. Coming back to the conic $\mathcal{C} = \{y = 0\} = \{y^2 = 0\}$, the first representation of it leads to nonempty positive and negative side, while in the second one, the negative side is empty.

3.4 Ellipses and the Volume Formula

The volume of an ellipse $\mathcal{E}$, $\mathrm{Vol}(\mathcal{E})$, is defined as the area of its convex side. If $\mathcal{E}$ is non-degenerate and presented in center form (7), consider the matrix

$$A := M/(2w - c^T M c).$$

It is easy to see that A is invariant under scaling $\mathcal{R}$ by any nonzero factor. The following holds.

Lemma 3.1

$$\mathrm{Vol}(\mathcal{E}) = \frac{\pi}{\sqrt{\det(A)}}.$$

For this note that $\det(M) > 0$ implies $\det(A) > 0$.

4 Specifications

This section contains the specifications as they appear in the CGAL Reference Manual [5].

4.1 2D Smallest Enclosing Ellipse (`CGAL_Min_ellipse_2<Traits>`)

Definition

An object of the class `CGAL_Min_ellipse_2<Traits>` is the unique ellipse of smallest area enclosing a finite set of points in two-dimensional euclidean space $\mathbb{E}_2$. For a point set P we denote by $me(P)$ the smallest ellipse that contains all points of P. Note that $me(P)$ can be degenerate, i.e. $me(P) = \emptyset$ if $P = \emptyset$, $me(P) = \{p\}$ if $P = \{p\}$, and $me(P) = \{(1-\lambda)p + \lambda q \mid 0 \le \lambda \le 1\}$ if $P = \{p, q\}$.

An inclusion-minimal subset S of P with $me(S) = me(P)$ is called a *support set*, the points in S are the *support points*. A support set has size at most five, and all its points lie on the boundary of $me(P)$. If $me(P)$ has more than five 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 4.3.

```
#include <CGAL/Min_ellipse_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 4.3. Traits class adapters to user supplied point classes are available, see Sections 4.4 and 4.5. Customizing own traits classes for optimisation algorithms can be done according to the requirements for traits classes listed in Section 4.8.

Types

`CGAL_Min_ellipse_2<Traits>::Traits`

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

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

`CGAL_Min_ellipse_2<Traits>::Point_iterator`	(bidirectional).
`CGAL_Min_ellipse_2<Traits>::Support_point_iterator`	(random access).

Creation

A `CGAL_Min_ellipse_2<Traits>` object can be created from an arbitrary point set P and by specialized construction methods expecting no, one, two, three, four or five points as arguments. The latter methods can be useful for reconstructing $me(P)$ from a given support set S of P.

```
template < class InputIterator >
CGAL_Min_ellipse_2<Traits>  min_ellipse( InputIterator first,
                                         InputIterator last,
                                         bool randomize = false,
                                         CGAL_Random& random = CGAL_random,
                                         Traits traits = Traits())
```

creates a variable `min_ellipse` of type `CGAL_Min_ellipse_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_ellipse_2<Traits>  min_ellipse( Traits traits = Traits());
```

creates a variable `min_ellipse` of type `CGAL_Min_ellipse_2<Traits>`. It is initialized to $me(\emptyset)$, the empty set.
Postcondition: `min_ellipse.is_empty()` = `true`.

```
CGAL_Min_ellipse_2<Traits>  min_ellipse( Point p,
                                         Traits traits = Traits())
```

creates a variable `min_ellipse` of type `CGAL_Min_ellipse_2<Traits>`. It is initialized to $me(\{p\})$, the set $\{p\}$.
Postcondition: `min_ellipse.is_degenerate()` = `true`.

```
CGAL_Min_ellipse_2<Traits>  min_ellipse( Point p,
                                         Point q,
                                         Traits traits = Traits())
```

creates a variable `min_ellipse` of type `CGAL_Min_ellipse_2<Traits>`. It is initialized to $me(\{p, q\})$, the set $\{(1-\lambda)p + \lambda q \mid 0 \leq \lambda \leq 1\}$.
Postcondition: `min_ellipse.is_degenerate()` = `true`.

```
CGAL_Min_ellipse_2<Traits>  min_ellipse( Point p1,
                                         Point p2,
                                         Point p3,
                                         Traits traits = Traits())
```

creates a variable `min_ellipse` of type `CGAL_Min_ellipse_2<Traits>`. It is initialized to $me(\{p1, p2, p3\})$.

```
CGAL_Min_ellipse_2<Traits>  min_ellipse( Point p1,
                                         Point p2,
                                         Point p3,
                                         Point p4,
                                         Traits traits = Traits())
```

creates a variable `min_ellipse` of type `CGAL_Min_ellipse_2<Traits>`. It is initialized to $me(\{p1, p2, p3, p4\})$.

```
CGAL_Min_ellipse_2<Traits>  min_ellipse( Point p1,
                                         Point p2,
                                         Point p3,
                                         Point p4,
                                         Point p5,
                                         Traits traits = Traits())
```

creates a variable `min_ellipse` of type `CGAL_Min_ellipse_2<Traits>`. It is initialized to $me(\{p1, p2, p3, p4, p5\})$.

Access Functions

`int` `min_ellipse.number_of_points()`

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

`int` `min_ellipse.number_of_support_points()`

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

`Point_iterator` `min_ellipse.points_begin()`

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

`Point_iterator` `min_ellipse.points_end()`

returns the corresponding past-the-end iterator.

`Support_point_iterator` `min_ellipse.support_points_begin()`

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

`Support_point_iterator` `min_ellipse.support_points_end()`

returns the corresponding past-the-end iterator.

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

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

`Ellipse` `min_ellipse.ellipse()`

returns the current ellipse of `min_ellipse`.

Predicates

By definition, an empty `CGAL_Min_ellipse_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_ellipse.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_ellipse`, resp.

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

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

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

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

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

returns `true`, iff `p` lies outside of `min_ellipse`.

`bool` `min_ellipse.is_empty()`

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

`bool` `min_ellipse.is_degenerate()`

returns `true`, iff `min_ellipse` is degenerate, i.e. if `min_ellipse` is empty, equal to a single point or equal to a segment, equivalently if the number of support points is less than 3.

Modifiers

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

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

inserts `p` into `min_ellipse` and recomputes the smallest enclosing ellipse.

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

inserts the points in the range [`first`,`last`) into `min_ellipse` and recomputes the smallest enclosing ellipse 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_ellipse.clear()`

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

Validity Check

An object `min_ellipse` is valid, iff

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

Note: In this release only the first item is considered by the validity check.

Using the traits class implementation for the CGAL kernel with exact arithmetic as described in Section 4.3 guarantees validity of `min_ellipse`. 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_ellipse.is_valid( bool verbose = false, int level = 0)`

returns `true`, iff `min_ellipse` contains all points of its defining set P. 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_ellipse.traits()`

returns a const reference to the traits class object.

I/O

`ostream&` `ostream& os` << `min_ellipse`

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

`istream&` `istream& is` >> `& min_ellipse`

reads `min_ellipse` 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_ellipse

writes min_ellipse to window stream ws.
Precondition: The window stream output operator is defined for Point and Ellipse.

See Also

CGAL_Min_circle_2 [8], CGAL_Min_ellipse_2_traits_2 (Section 4.3), CGAL_Min_ellipse_2_adapterC2 (Section 4.4), CGAL_Min_ellipse_2_adapterH2 (Section 4.5).

Implementation

We implement the algorithm of Welzl, with move-to-front heuristic [14], using the primitives as described in [6, 7]. 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 ellipse from scratch. The clear operation and the check for validity each takes linear time.

Example

To illustrate the creation of CGAL_Min_ellipse_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_ellipse_2_traits_2.h>
#include <CGAL/Min_ellipse_2.h>

typedef  CGAL_Gmpz                        NT;
typedef  CGAL_Homogeneous<NT>             R;
typedef  CGAL_Point_2<R>                  Point;
typedef  CGAL_Min_ellipse_2_traits_2<R>   Traits;
typedef  CGAL_Min_ellipse_2<Traits>       Min_ellipse;

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_ellipse  me1( P, P+n);           // very slow
    Min_ellipse  me2( P, P+n, true);     // fast
```

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

4.2 2D Optimisation Ellipse (CGAL_Optimisation_ellipse_2<R>)

Definition

An object of the class CGAL_Optimisation_ellipse_2<R> is an ellipse 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 ellipse can be degenerated, i.e. it can be empty, equal to a single point or equal to a segment. By definition, an empty CGAL_Optimisation_ellipse_2<R> has no boundary and no bounded side, i.e. its unbounded side equals the whole plane $\mathbb{E}_2$. A CGAL_Optimisation_ellipse_2<R> equal to a single point p or equal to a segment s, resp., has no bounded side, its boundary is $\{p\}$ or s, resp., and its unbounded side equals $\mathbb{E}_2 \setminus \{p\}$ or $\mathbb{E}_2 \setminus s$, resp.

#include <CGAL/Optimisation_ellipse_2.h>

Types

typedef CGAL_Point_2<R> Point; Point type.

Creation

void ellipse.set()

sets ellipse to the empty ellipse.

void ellipse.set(CGAL_Point_2<R> p)

sets ellipse to the ellipse equal to the single point p.

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

sets ellipse to the ellipse equal to the segment $\overline{pq}$.
Precondition: p and q are distinct.

void ellipse.set(CGAL_Point_2<R> p1,
CGAL_Point_2<R> p2,
CGAL_Point_2<R> p3)

sets ellipse to the ellipse of smallest area through p1, p2 and p3.
Precondition: p1, p2, p3 are not collinear.

void ellipse.set(CGAL_Point_2<R> p1,

```
                    CGAL_Point_2<R> p2,
                    CGAL_Point_2<R> p3,
                    CGAL_Point_2<R> p4)
```

sets `ellipse` to the ellipse of smallest area through `p1`, `p2`, `p3` and `p4`.
Precondition: `p1`, `p2`, `p3`, `p4` are in convex position.

`void` `ellipse.set( CGAL_Point_2<R> p1,`
`CGAL_Point_2<R> p2,`
`CGAL_Point_2<R> p3,`
`CGAL_Point_2<R> p4,`
`CGAL_Point_2<R> p5)`

sets `ellipse` to the unique ellipse through `p1`, `p2`, `p3`, `p4` and `p5`.
Precondition: There exists an ellipse through `p1`, `p2`, `p3`, `p4`, `p5`.

Access Functions

Equality Tests

`bool` `ellipse == ellipse2`

returns `true`, iff `ellipse` and `ellipse2` are equal.

`bool` `ellipse != ellipse2`

returns `true`, iff `ellipse` and `ellipse2` are not equal.

Predicates

`CGAL_Bounded_side` `ellipse.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 `ellipse`, resp.

`bool` `ellipse.has_on_bounded_side( CGAL_Point_2<R> p)`

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

`bool` `ellipse.has_on_boundary( CGAL_Point_2<R> p)`

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

`bool` `ellipse.has_on_unbounded_side( CGAL_Point_2<R> p)`

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

`bool` `ellipse.is_empty()`

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

`bool` `ellipse.is_degenerate()`

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

I/O

`ostream&` `ostream& os << ellipse`

writes `ellipse` to output stream `os`.

`istream&` `istream& is >> & ellipse`

reads `ellipse` from input stream `is`.

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

`CGAL_Window_stream&` `CGAL_Window_stream& ws << ellipse`

writes `ellipse` to window stream `ws`.

4.3 Traits Class Implementation using the two-dimensional CGAL Kernel (`CGAL_Min_ellipse_2_traits_2<R>`)

Definition

The class `CGAL_Min_ellipse_2_traits_2<R>` interfaces the 2D optimisation algorithm for smallest enclosing ellipses with the CGAL 2D kernel.

```
#include <CGAL/Min_ellipse_2_traits_2.h>
```

Types

```
typedef CGAL_Point_2<R>                 Point;
typedef CGAL_Optimisation_ellipse_2<R>  Ellipse;
```

Creation

```
CGAL_Min_ellipse_2_traits_2<R>  traits;
CGAL_Min_ellipse_2_traits_2<R>  traits( CGAL_Min_ellipse_2_traits_2<R>);
```

See Also

CGAL_Min_ellipse_2 (Section 4.1), CGAL_Min_ellipse_2_adapterC2 (Section 4.4), CGAL_Min_ellipse_2_adapterH2 (Section 4.5), Requirements of Traits Classes for 2D Smallest Enclosing Ellipse (Section 4.8).

Example

See example for CGAL_Min_ellipse_2 (Section 4.1).

4.4 Traits Class Adapter for 2D Smallest Enclosing Ellipse to 2D Cartesian Points (CGAL_Min_ellipse_2_adapterC2<PT,DA>)

Definition

The class CGAL_Min_ellipse_2_adapterC2<PT,DA> interfaces the 2D optimisation algorithm for smallest enclosing ellipses with the point class PT. The data accessor DA [9] 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_ellipse_2_adapterC2.h>
```

Types

`CGAL_Min_ellipse_2_adapterC2<PT,DA>::DA`	Data accessor for Cartesian coordinates.
`CGAL_Min_ellipse_2_adapterC2<PT,DA>::Point`	Point type.
`CGAL_Min_ellipse_2_adapterC2<PT,DA>::Ellipse`	Ellipse type.

Creation

```
CGAL_Min_ellipse_2_adapterC2<PT,DA>  adapter( DA da = DA());
CGAL_Min_ellipse_2_adapterC2<PT,DA>  adapter(
                                       CGAL_Min_ellipse_2_adapterC2<PT,DA>);
```

See Also

CGAL_Min_ellipse_2 (Section 4.1), CGAL_Min_ellipse_2_traits_2 (Section 4.3), CGAL_Min_ellipse_2_adapterH2 (Section 4.5), Requirements of Traits Class Adapters to 2D Cartesian Points (Section 4.6).

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_ellipse_2_adapterC2.h>
#include <CGAL/Min_ellipse_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_ellipse_2_adapterC2 < PtC, PtC_DA >  AdapterC;
typedef  CGAL_Min_ellipse_2 < AdapterC >               Min_ellipse;

// do something with Min_ellipse
Min_ellipse   me( /*...*/ );
```

4.5 Traits Class Adapter for 2D Smallest Enclosing Ellipse to 2D Homogeneous Points (`CGAL_Min_ellipse_2_adapterH2<PT,DA>`)

Definition

The class `CGAL_Min_ellipse_2_adapterH2<PT,DA>` interfaces the 2D optimisation algorithm for smallest enclosing ellipses with the point class `PT`. The data accessor `DA` [9] 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_ellipse_2_adapterH2.h>`

Types

`CGAL_Min_ellipse_2_adapterH2<PT,DA>::DA`	Data accessor for homogeneous coordinates.

`CGAL_Min_ellipse_2_adapterH2<PT,DA>::Point` Point type.

`CGAL_Min_ellipse_2_adapterH2<PT,DA>::Ellipse` Ellipse type.

Creation

```
CGAL_Min_ellipse_2_adapterH2<PT,DA>  adapter( DA da = DA());
CGAL_Min_ellipse_2_adapterH2<PT,DA>  adapter(
                                       CGAL_Min_ellipse_2_adapterH2<PT,DA>);
```

See Also

`CGAL_Min_ellipse_2` (Section 4.1), `CGAL_Min_ellipse_2_traits_2` (Section 4.3), `CGAL_Min_ellipse_2_adapterC2` (Section 4.4), Requirements of Traits Class Adapters to 2D Homogeneous Points (Section 4.7).

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_ellipse_2_adapterH2.h>
#include <CGAL/Min_ellipse_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_ellipse_2_adapterH2< PtH, PtH_DA >  AdapterH;
typedef  CGAL_Min_ellipse_2< AdapterH >               Min_ellipse;

// do something with Min_ellipse
Min_ellipse   me( /*...*/ );
```

4.6 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` [9]. 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`.

4.7 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` [9]. 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`.

4.8 Requirements of Traits Classes for 2D Smallest Enclosing Ellipse

The class template `CGAL_Min_ellipse_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_ellipse_2`. A traits class implementation using the CGAL 2D kernel is available and described in Section 4.3. In addition, we provide traits class adapters to user supplied point classes, see Sections 4.4 and 4.5. 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_ellipse_2` must provide the following primitives.

Types

`Traits::Point`	The point type must provide default and copy constructor, assignment and equality test.
`Traits::Ellipse`	The ellipse 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 `Ellipse` have to be provided, see topic **I/O** in Section 4.1.

Variables

`Ellipse ellipse;`	The actual ellipse. 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.

Ellipse Type (`Ellipse`)

Definition

An object of the class `Ellipse` is an ellipse in two-dimensional euclidean plane $\mathbb{E}_2$. Its boundary splits the plane into a bounded and an unbounded side. By definition, an empty `Ellipse` has no boundary and no bounded side, i.e. its unbounded side equals the whole plane $\mathbb{E}_2$.

Types

`Ellipse:: Point` Point type.

Creation

`void` `ellipse.set()`

sets `ellipse` to the empty ellipse.

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

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

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

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

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

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

`void` `ellipse.set( Point p, Point q, Point r, Point s)`

sets `ellipse` to the smallest ellipse through `p`, `q`, `r`, and `s`. The algorithm guarantees that this ellipse exists.

```
void                ellipse.set( Point p,
                                 Point q,
                                 Point r,
                                 Point s,
                                 Point t)
```

sets `ellipse` to the unique conic through `p`, `q`, `r`, `s`, and `t`. The algorithm guarantees that this conic is an ellipse.

Predicates

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

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

Each of the following predicates is only needed, if the corresponding predicate of `CGAL_Min_ellipse_2` is used.

`CGAL_Bounded_side` `ellipse.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 `ellipse`, resp.

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

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

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

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

`bool` `ellipse.is_empty()`

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

`bool` `ellipse.is_degenerate()`

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

I/O

The following I/O operators are only needed, if the corresponding I/O operators of `CGAL_Min_ellipse_2` are used.

`ostream&` `ostream& os << ellipse`

writes `ellipse` to output stream `os`.

`istream&` `istream& is >> & ellipse`

reads `ellipse` from input stream `is`.

`CGAL_Window_stream&`

`CGAL_Window_stream& ws << ellipse`

writes `ellipse` to window stream `ws`.

5 Implementations

5.1 Class Template `CGAL_Min_ellipse_2<Traits>`

First, we declare the class template `CGAL_Min_ellipse_2`.

Min_ellipse_2 declaration [1] ≡ {
```
template < class _Traits >
class CGAL_Min_ellipse_2;
```
}
This macro is invoked in definition 150.

The actual work of the algorithm is done in the private member functions `me` and `compute_ellipse`. The former directly realizes the pseudo-code of `me`(P, B), the latter solves the basic case `me`$(\emptyset, B)$, 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_ellipse_2 interface [2] ≡ {
```
template < class _Traits >
class CGAL_Min_ellipse_2 {
  public:
    Min_ellipse_2 public interface [3]

  private:
    // private data members
    Min_ellipse_2 private data members [4]

    // copying and assignment not allowed!
    CGAL_Min_ellipse_2( const CGAL_Min_ellipse_2<_Traits>&);
    CGAL_Min_ellipse_2<_Traits>&
        operator = ( const CGAL_Min_ellipse_2<_Traits>&);

dividing line [163]

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

  public:
    // Access functions and predicates
    // -------------------------------
    Min_ellipse_2 access functions 'number_of_...' [10]

    Min_ellipse_2 predicates 'is_...' [13]
```

```
    Min_ellipse_2 access functions [11]

    Min_ellipse_2 predicates [14]

  private:
    // Private member functions
    // ------------------------
    Min_ellipse_2 private member function 'compute_ellipse' [24]

    Min_ellipse_2 private member function 'me' [25]

  public:
    // Constructors
    // ------------
    Min_ellipse_2 constructors [7]

    // Destructor
    // ----------
    Min_ellipse_2 destructor [9]

    // Modifiers
    // ---------
    Min_ellipse_2 modifiers [15]

    // Validity check
    // --------------
    Min_ellipse_2 validity check [18]

    // Miscellaneous
    // -------------
    Min_ellipse_2 miscellaneous [21]
};
}
```

This macro is invoked in definition 150.

5.1.1 Public Interface

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

Min_ellipse_2 public interface [3] $\equiv$ {

```
  // types
  typedef           _Traits                      Traits;
  typedef typename  _Traits::Point               Point;
  typedef typename  _Traits::Ellipse             Ellipse;
  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_ellipse_2( const Point*  first,
                    const Point*  last,
                    bool          randomize = false,
                    CGAL_Random&  random    = CGAL_random,
                    const Traits& traits    = Traits());
CGAL_Min_ellipse_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_ellipse_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_ellipse_2( const Traits& traits = Traits());
CGAL_Min_ellipse_2( const Point&  p,
                    const Traits& traits = Traits());
CGAL_Min_ellipse_2( const Point&  p,
                    const Point&  q,
                    const Traits& traits = Traits());
CGAL_Min_ellipse_2( const Point&  p1,
                    const Point&  p2,
                    const Point&  p3,
                    const Traits& traits = Traits());
~CGAL_Min_ellipse_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 Ellipse&  ellipse( ) 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.

5.1.2 Private Data Members

First, the traits class object is stored.

Min_ellipse_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 [13].

Min_ellipse_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 five 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 `me`(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_ellipse_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 ellipse is stored in a variable `ellipse` provided by the traits class object, by the end of computation equal to $me(P)$. During computation, `tco.ellipse` equals the ellipse me appearing in the pseudo-code for `me`(P, B), see Section 2.

5.1.3 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 ellipse $me(P)$ 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 me is called to compute $me(P) = me(P, \emptyset)$.

Min_ellipse_2 constructors [7] $+\equiv$ {

```
#ifndef CGAL_CFG_NO_MEMBER_TEMPLATES

    // STL-like constructor (member template)
    template < class InputIterator >
    CGAL_Min_ellipse_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[ 5];
```

```
        // 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 me
        me( points.end(), 0);
    }

#else

    // STL-like constructor for C array and vector<Point>
    CGAL_Min_ellipse_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[ 5];

        // 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 me
        me( points.end(), 0);
    }
```

```
// STL-like constructor for list<Point>
CGAL_Min_ellipse_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[ 5];

    // 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 me
    me( points.end(), 0);
}

// STL-like constructor for istream_iterator<Point>
CGAL_Min_ellipse_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[ 5];

    // 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 me
        me( 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 ellipse for up to five points. The idea is the following: recall that for any point set P there exists $S \subseteq P$, $|S| \leq 5$ with $me(S) = me(P)$ (in fact, such a set S is determined by the algorithm). Once S has been computed (or given otherwise), $me(P)$ 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 5. For $|S| = 0$, we get the default constructor, building $me(\emptyset)$.

Min_ellipse_2 constructors [8] $+\equiv$ {

```
// default constructor
inline
CGAL_Min_ellipse_2( const Traits& traits = Traits())
    : tco( traits), n_support_points( 0)
{
    // allocate support points' array
    support_points = new Point[ 5];

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

    CGAL_optimisation_postcondition( is_empty());
}

// constructor for one point
inline
CGAL_Min_ellipse_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[ 5];

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

    CGAL_optimisation_postcondition( is_degenerate());
}

// constructor for two points
inline
CGAL_Min_ellipse_2( const Point& p,
                    const Point& q,
                    const Traits& traits = Traits())
    : tco( traits)
{
    // allocate support points' array
    support_points = new Point[ 5];

    // store points
    points.push_back( p);
    points.push_back( q);

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

// constructor for three points
inline
CGAL_Min_ellipse_2( const Point& p1,
                    const Point& p2,
                    const Point& p3,
                    const Traits& traits = Traits())
    : tco( traits)
{
    // allocate support points' array
    support_points = new Point[ 5];

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

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

```
    // constructor for four points
    inline
    CGAL_Min_ellipse_2( const Point& p1,
                        const Point& p2,
                        const Point& p3,
                        const Point& p4,
                        const Traits& traits = Traits())
        : tco( traits)
    {
        // allocate support points' array
        support_points = new Point[ 5];

        // store points
        points.push_back( p1);
        points.push_back( p2);
        points.push_back( p3);
        points.push_back( p4);

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

    // constructor for five points
    inline
    CGAL_Min_ellipse_2( const Point& p1,
                        const Point& p2,
                        const Point& p3,
                        const Point& p4,
                        const Point& p5,
                        const Traits& traits = Traits())
        : tco( traits)
    {
        // allocate support points' array
        support_points = new Point[ 5];

        // store points
        points.push_back( p1);
        points.push_back( p2);
        points.push_back( p3);
        points.push_back( p4);
        points.push_back( p5);

        // compute me
        me( 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_ellipse_2 destructor [9] ≡ {
```
    inline
    ~CGAL_Min_ellipse_2( )
    {
        // free support points' array
        delete[] support_points;
    }
}
```
This macro is invoked in definition 2.

5.1.4 Access Functions

These functions are used to retrieve information about the current status of the `CGAL_Min_ellipse_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_ellipse_2 access functions 'number_of_...' [10] ≡ {
```
    // #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_ellipse_2 access functions [11] + ≡ {
```
    // 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 `ellipse`.

Min_ellipse_2 access functions [12] $+\equiv$ {

```
    // ellipse
    inline
    const Ellipse&
    ellipse( ) const
    {
        return( tco.ellipse);
    }
}
```

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

5.1.5 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_ellipse_2 predicates 'is_...' [13] ≡ {

```
// 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-ellipse tests, based on the corresponding predicates of class `Ellipse`.

Min_ellipse_2 predicates [14] ≡ {

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

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

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

```
    inline
    bool
    has_on_unbounded_side( const Point& p) const
    {
        return( tco.ellipse.has_on_unbounded_side( p));
    }
}
```
This macro is invoked in definition 2.

5.1.6 Modifiers

There is another way to build up $me(P)$, other than by supplying the point set P at once. Namely, $me(P)$ 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_ellipse_2<Traits>` object representing $me(P)$, computes $me(P \cup \{p\})$, where work has to be done only if p lies outside $me(P)$. In this case, $me(P \cup \{p\}) = me(P, \{p\})$ holds, so the private member function `me` 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_ellipse_2 modifiers [15] $+\equiv$ {

```
    void
    insert( const Point& p)
    {
        // p not in current ellipse?
        if ( has_on_unbounded_side( p)) {

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

            // recompute me
            me( 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_ellipse_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_ellipse_2<Traits>` object and resets it to the empty ellipse.

Min_ellipse_2 modifiers [17] $+\equiv$ {
```
    void
    clear( )
    {
        points.erase( points.begin(), points.end());
        n_support_points = 0;
        tco.ellipse.set();
    }
}
```
This macro is defined in definitions 15, 16, and 17.
This macro is invoked in definition 2.

5.1.7 Validity Check

A `CGAL_Min_ellipse_2<Traits>` object can be checked for validity. This means, it is checked whether (a) the ellipse contains all points of its defining set P, (b) the ellipse is the smallest ellipse spanned by its support set, and (c) the support set is minimal, i.e. no support point is redundant. (*Note:* (b) and (c) are not yet implemented. Instead we check if the support set lies on the boundary of the ellipse.) 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_ellipse_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_ellipse_2<Traits>::" << endl;
        verr << "is_valid( true, " << level << "):" << endl;
        verr << "  |P| = " << number_of_points()
             << ", |S| = " << number_of_support_points() << endl;

        // containment check (a)
        Min_ellipse_2 containment check [19]

        // support set checks (b)+(c) (not yet implemented)

        // alternative support set check
        Min_ellipse_2 support set check [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_ellipse_2 containment check [19] ≡ {
```
    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,
                        "ellipse does not contain all points"));
    verr << "passed." << endl;
}
```
This macro is invoked in definition 18.

The alternative support set check is easy to perform, just a loop over all support points in `support_points`.

Min_ellipse_2 support set check [20] ≡ {
```
    verr << "  +) support set check..." << flush;
    Support_point_iterator support_point_iter;
    for ( support_point_iter  = support_points_begin();
          support_point_iter != support_points_end();
          ++support_point_iter)
        if ( ! has_on_boundary( *support_point_iter))
            return( CGAL__optimisation_is_valid_fail( verr,
                        "ellipse does not have all \
                         support points on the boundary"));
    verr << "passed." << endl;
}
```
This macro is invoked in definition 18.

5.1.8 Miscellaneous

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

Min_ellipse_2 miscellaneous [21] ≡ {
```
    inline
    const Traits&
    traits( ) const
    {
        return( tco);
```

```
    }
}
```
This macro is invoked in definition 2.

5.1.9 I/O

Min_ellipse_2 I/O operators declaration [22] ≡ {
```
    template < class _Traits >
    ostream& operator << ( ostream& os,
                           const CGAL_Min_ellipse_2<_Traits>& me);

    template < class _Traits >
    istream& operator >> ( istream& is,
                           CGAL_Min_ellipse_2<_Traits>& me);
}
```
This macro is invoked in definition 150.

Min_ellipse_2 I/O operators [23] ≡ {
```
    template < class _Traits >
    ostream&
    operator << ( ostream& os,
                  const CGAL_Min_ellipse_2<_Traits>& min_ellipse)
    {
        typedef typename  CGAL_Min_ellipse_2<_Traits>::Point  Point;

        switch ( CGAL_get_mode( os)) {

          case CGAL_IO::PRETTY:
            os << endl;
            os << "CGAL_Min_ellipse_2( |P| = "
               << min_ellipse.number_of_points()
               << ", |S| = "
               << min_ellipse.number_of_support_points() << endl;
            os << "  P = {" << endl;
            os << "    ";
            copy( min_ellipse.points_begin(), min_ellipse.points_end(),
                  ostream_iterator<Point>( os, ",\n    "));
            os << "}" << endl;
            os << "  S = {" << endl;
            os << "    ";
            copy( min_ellipse.support_points_begin(),
                  min_ellipse.support_points_end(),
                  ostream_iterator<Point>( os, ",\n    "));
            os << "}" << endl;
            os << "  ellipse = " << min_ellipse.ellipse() << endl;
            os << ")" << endl;
            break;
```

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

        case CGAL_IO::BINARY:
          copy( min_ellipse.points_begin(), min_ellipse.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_ellipse_2<Traits>& min_ellipse)
  {
      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_ellipse_2<Traits>::Point  Point;
          typedef           istream_iterator<Point,ptrdiff_t>  Is_it;
          min_ellipse.clear();
          min_ellipse.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 151.

5.1.10 Private Member Function `compute_ellipse`

This is the method for computing the basic case $\texttt{me}(\emptyset, B)$, 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 $|B| \leq 5$.

Min_ellipse_2 private member function 'compute_ellipse' [24] $\equiv$ {

```
// compute_ellipse
inline
void
compute_ellipse( )
{
    switch ( n_support_points) {
      case 5:
        tco.ellipse.set( support_points[ 0],
                         support_points[ 1],
                         support_points[ 2],
                         support_points[ 3],
                         support_points[ 4]);
        break;
      case 4:
        tco.ellipse.set( support_points[ 0],
                         support_points[ 1],
                         support_points[ 2],
                         support_points[ 3]);
        break;
      case 3:
        tco.ellipse.set( support_points[ 0],
                         support_points[ 1],
                         support_points[ 2]);
        break;
      case 2:
        tco.ellipse.set( support_points[ 0], support_points[ 1]);
        break;
      case 1:
        tco.ellipse.set( support_points[ 0]);
        break;
      case 0:
        tco.ellipse.set( );
        break;
      default:
        CGAL_optimisation_assertion( ( n_support_points >= 0) &&
                                     ( n_support_points <= 5) ); }
}
```

}

This macro is invoked in definition 2.

5.1.11 Private Member Function `me`

This function computes the general ellipse $me(P, B)$, 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_ellipse_2 private member function 'me' [25] $\equiv$ {

```
void
me( const Point_iterator& last, int n_sp)
{
    // compute ellipse through support points
    n_support_points = n_sp;
    compute_ellipse();
    if ( n_sp == 5) 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 ellipse?
        if ( has_on_unbounded_side( p)) {

            // recursive call with p as additional support point
            support_points[ n_sp] = p;
            me( 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.

5.2 Class Template `CGAL_Optimisation_ellipse_2<R>`

First, we declare the class template `CGAL_Optimisation_ellipse_2`.

Optimisation_ellipse_2 declaration [26] $\mathbf{M} \equiv$ {

```
template < class _R >
class CGAL_Optimisation_ellipse_2;
```

```
    class ostream;
    class istream;
    class CGAL_Window_stream;
}
```
This macro is invoked in definitions 152 and 155.

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_ellipse_2 interface [27] ≡ {
```
    template < class _R >
    class CGAL_Optimisation_ellipse_2 {
        friend  ostream&  operator << CGAL_NULL_TMPL_ARGS (
            ostream&, const CGAL_Optimisation_ellipse_2<_R>&);
        friend  istream&  operator << CGAL_NULL_TMPL_ARGS (
            istream&, CGAL_Optimisation_ellipse_2<_R> &);
        friend  CGAL_Window_stream& operator << CGAL_NULL_TMPL_ARGS (
            CGAL_Window_stream&, const CGAL_Optimisation_ellipse_2<_R>&);
      public:
        Optimisation_ellipse_2 public interface [28]

      private:
        // private data members
        Optimisation_ellipse_2 private data members [29]

    dividing line [163]

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

      public:
        // Set functions
        // -------------
        Optimisation_ellipse_2 set functions [33]

        // Access functions
        // ----------------
        Optimisation_ellipse_2 access functions [34]

        // Equality tests
        // --------------
        Optimisation_ellipse_2 equality tests [35]

        // Predicates
        // ----------
```

```
        Optimisation_ellipse_2 predicates [36]
    };
}
```
This macro is invoked in definition 152.

5.2.1 Public Interface

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

Optimisation_ellipse_2 public interface [28] $\equiv$ {

```
    // types
    typedef           _R               R;
    typedef           typename _R::RT  RT;
    typedef           typename _R::FT  FT;
    typedef           CGAL_Point_2<R>  Point;
    typedef           CGAL_Conic_2<R>  Conic;

    /*************************************************************************
    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& p1, const Point& p2, const Point& p3);
    void  set( const Point& p1, const Point& p2,
               const Point& p3, const Point& p4);
    void  set( const Point& p1, const Point& p2,
               const Point& p3, const Point& p4, const Point& p5);

    // access functions
    int  number_of_boundary_points()

    // equality tests
    bool  operator == ( const CGAL_Optimisation_ellipse_2<R>& e) const;
    bool  operator != ( const CGAL_Optimisation_ellipse_2<R>& e) 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 27.

5.2.2 Private Data Members

The representation of the ellipse depends on the number of given boundary points, stored in `n_boundary_points`.

Optimisation_ellipse_2 private data members [29] $+\equiv$ {
```
    int    n_boundary_points;           // number of boundary points
```
}
This macro is defined in definitions 29, 30, 31, and 32.
This macro is invoked in definition 27.

In the degenerate cases with zero to two boundary points, the given points are stored directly in `boundary_point1` and `boundary_point2`, resp.

Optimisation_ellipse_2 private data members [30] $+\equiv$ {
```
    Point  boundary_point1, boundary_point2;  // two boundary points
```
}
This macro is defined in definitions 29, 30, 31, and 32.
This macro is invoked in definition 27.

Given three or five points, the ellipse is represented as a conic, using the class `CGAL_Conic_2<R>`. The case with four boundary points is the most complicated one, since in general a direct representation with one conic has irrational coordinates [7]. Therefore the ellipse is represented implicitly as a linear combination of two conics.

Optimisation_ellipse_2 private data members [31] $+\equiv$ {
```
    Conic  conic1, conic2;                   // two conics
```
}
This macro is defined in definitions 29, 30, 31, and 32.
This macro is invoked in definition 27.

Finally, in the case of four boundary points, we need the gradient vector of the linear combination for the volume derivative in the in-ellipse test.

Optimisation_ellipse_2 private data members [32] $+\equiv$ {
```
    RT     dr, ds, dt, du, dv, dw;           // the gradient vector
```
}
This macro is defined in definitions 29, 30, 31, and 32.
This macro is invoked in definition 27.

5.2.3 Set Functions

We provide set functions taking zero, one, two, three, four or five boundary points. They all set the variable to the smallest ellipse through the given points. *Note:* The set function

taking five boundary points only uses the fifth point from its input together with the two internally represented conics to compute the ellipse. The algorithm in Section 2 guarantees that this set function is only called an ellipse that already have the first four points as its boundary points.

Optimisation_ellipse_2 set functions [33] $\equiv$ {

```
inline
void
set( )
{
    n_boundary_points = 0;
}

inline
void
set( const Point& p)
{
    n_boundary_points = 1;
    boundary_point1   = p;
}

inline
void
set( const Point& p, const Point& q)
{
    n_boundary_points = 2;
    boundary_point1   = p;
    boundary_point2   = q;
}

inline
void
set( const Point& p1, const Point& p2, const Point& p3)
{
    n_boundary_points = 3;
    conic1.set_ellipse( p1, p2, p3);
}

inline
void
set( const Point& p1, const Point& p2,
     const Point& p3, const Point& p4)
{
    n_boundary_points = 4;
    Conic::set_two_linepairs( p1, p2, p3, p4, conic1, conic2);
    dr = RT( 0);
    ds = conic1.r() * conic2.s() - conic2.r() * conic1.s(),
```

```
        dt = conic1.r() * conic2.t() - conic2.r() * conic1.t(),
        du = conic1.r() * conic2.u() - conic2.r() * conic1.u(),
        dv = conic1.r() * conic2.v() - conic2.r() * conic1.v(),
        dw = conic1.r() * conic2.w() - conic2.r() * conic1.w();
    }

    inline
    void
    set( const Point&, const Point&,
         const Point&, const Point&, const Point& p5)
    {
        n_boundary_points = 5;
        conic1.set( conic1, conic2, p5);
        conic1.analyse();
    }
}
```

This macro is invoked in definition 27.

5.2.4 Access Functions

Optimisation_ellipse_2 access functions [34] $\equiv$ {

```
    inline
    int
    number_of_boundary_points( ) const
    {
        return( n_boundary_points);
    }
}
```

This macro is invoked in definition 27.

5.2.5 Equality Tests

Optimisation_ellipse_2 equality tests [35] $\equiv$ {

```
    bool
    operator == ( const CGAL_Optimisation_ellipse_2<R>& e) const
    {
        if ( n_boundary_points != e.n_boundary_points)
            return( false);

        switch ( n_boundary_points) {
          case 0:
            return( true);
          case 1:
            return( boundary_point1 == e.boundary_point1);
          case 2:
            return(    (    ( boundary_point1 == e.boundary_point1)
```

```
                        && ( boundary_point2 == e.boundary_point2))
                    || (    ( boundary_point1 == e.boundary_point2)
                        && ( boundary_point2 == e.boundary_point1)));
          case 3:
          case 5:
            return( conic1 == e.conic1);
          case 4:
            return(    (    ( conic1 == e.conic1)
                        && ( conic2 == e.conic2))
                    || (    ( conic1 == e.conic2)
                        && ( conic2 == e.conic1)));
          default:
            CGAL_optimisation_assertion(    ( n_boundary_points >= 0)
                                         && ( n_boundary_points <= 5)); }
        // keeps g++ happy
        return( false);
    }

    inline
    bool
    operator != ( const CGAL_Optimisation_ellipse_2<R>& e) const
    {
        return( ! operator == ( e));
    }
}
```

This macro is invoked in definition 27.

5.2.6 Predicates

The following predicates perform in-ellipse tests and check for emptiness and degeneracy, resp. The way to evaluate the in-ellipse test depends on the number of boundary points and is realised by a case analysis. Again, the case with four points is the most difficult one.

Optimisation_ellipse_2 predicates [36] $\equiv$ {

```
    inline
    CGAL_Bounded_side
    bounded_side( const Point& p) const
    {
        switch ( n_boundary_points) {
          case 0:
            return( CGAL_ON_UNBOUNDED_SIDE);
          case 1:
            return( ( p == boundary_point1) ?
                          CGAL_ON_BOUNDARY : CGAL_ON_UNBOUNDED_SIDE);
          case 2:
            return(    ( p == boundary_point1)
```

```
                || ( p == boundary_point2)
                || ( CGAL_are_ordered_along_line(
                          boundary_point1, p, boundary_point2)) ?
                     CGAL_ON_BOUNDARY : CGAL_ON_UNBOUNDED_SIDE);
      case 3:
      case 5:
        return( conic1.convex_side( p));
      case 4: {
        Conic c;
        c.set( conic1, conic2, p);
        c.analyse();
        if ( ! c.is_ellipse()) {
            c.set_ellipse( conic1, conic2);
            c.analyse();
            return( c.convex_side( p)); }
        else {
            int tau_star = -c.vol_derivative( dr, ds, dt, du, dv, dw);
            return( CGAL_static_cast( CGAL_Bounded_side,
                                      CGAL_sign( tau_star))); } }
      default:
        CGAL_optimisation_assertion( ( n_boundary_points >= 0) &&
                                     ( n_boundary_points <= 5) ); }
    // keeps g++ happy
    return( CGAL_Bounded_side( 0));
}

inline
bool
has_on_bounded_side( const Point& p) const
{
    return( bounded_side( p) == CGAL_ON_BOUNDED_SIDE);
}

inline
bool
has_on_boundary( const Point& p) const
{
    return( bounded_side( p) == CGAL_ON_BOUNDARY);
}

inline
bool
has_on_unbounded_side( const Point& p) const
{
    return( bounded_side( p) == CGAL_ON_UNBOUNDED_SIDE);
}
```

```
    inline
    bool
    is_empty( ) const
    {
        return( n_boundary_points == 0);
    }

    inline
    bool
    is_degenerate( ) const
    {
        return( n_boundary_points < 3);
    }
}
```
This macro is invoked in definition 27.

5.2.7 I/O

Optimisation_ellipse_2 I/O operators declaration [37] ≡ {
```
    template < class _R >
    ostream&
    operator << ( ostream& os, const CGAL_Optimisation_ellipse_2<_R>& e);

    template < class _R >
    istream&
    operator >> ( istream& is, CGAL_Optimisation_ellipse_2<_R>      & e);
}
```
This macro is invoked in definition 152.

Optimisation_ellipse_2 I/O operators [38] ≡ {
```
    template < class _R >
    ostream&
    operator << ( ostream& os, const CGAL_Optimisation_ellipse_2<_R>& e)
    {
        const char* const  empty       = "";
        const char* const  pretty_head = "CGAL_Optimisation_ellipse_2( ";
        const char* const  pretty_sep  = ", ";
        const char* const  pretty_tail = ")";
        const char* const  ascii_sep   = " ";

        const char*  head = empty;
        const char*  sep  = empty;
        const char*  tail = empty;

        switch ( CGAL_get_mode( os)) {
          case CGAL_IO::PRETTY:
            head = pretty_head;
```

```
          sep  = pretty_sep;
          tail = pretty_tail;
          break;
        case CGAL_IO::ASCII:
          sep  = ascii_sep;
          break;
        case CGAL_IO::BINARY:
          break;
        default:
          CGAL_optimisation_assertion_msg( false,
                                           "CGAL_IO::mode invalid!");
          break; }

    os << head << e.n_boundary_points;
    switch ( e.n_boundary_points) {
      case 0:
        break;
      case 1:
        os << sep << e.boundary_point1;
        break;
      case 2:
        os << sep << e.boundary_point1
           << sep << e.boundary_point2;
        break;
      case 3:
      case 5:
        os << sep << e.conic1;
        break;
      case 4:
        os << sep << e.conic1
           << sep << e.conic2;
        break; }
    os << tail;

    return( os);
}

template < class _R >
istream&
operator >> ( istream& is, CGAL_Optimisation_ellipse_2<_R>& e)
{
    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:
            CGAL_read( is, e.n_boundary_points);
            switch ( e.n_boundary_points) {
              case 0:
                break;
              case 1:
                is >> e.boundary_point1;
                break;
              case 2:
                is >> e.boundary_point1
                   >> e.boundary_point2;
                break;
              case 3:
              case 5:
                is >> e.conic1;
                break;
              case 4:
                is >> e.conic1
                   >> e.conic2;
                break; }
            break;

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

        return( is);
    }
}
```

This macro is invoked in definition 153.

5.3 Class Template `CGAL_Conic_2<R>`

An object of the class `CGAL_Conic_2<R>` stores an oriented conic $\mathcal{C}_\mathcal{R} = (\mathcal{C}(\mathcal{R}), \mathcal{R})$ by its representation $\mathcal{R}$. The template parameter `R` is the representation type (Cartesian or homogeneous). In the sequel, a conic always means an oriented conic.

Conic_2 declaration [39] $\equiv$ {

```
    template < class _R >
    class CGAL_Conic_2;
}
```

This macro is invoked in definition 154.

The class `CGAL_Conic_2<R>` has several member functions, some of which can be considered general purpose (and are declared public), others are more specialized and needed only by

the class template `CGAL_Optimisation_ellipse_2<R>` – they appear as private methods. As a general rule, a method only qualifies for the public domain if it can be specified without referring to particular values of $\mathcal{R}$, in other words, if its behavior only depends on the equivalence class of all positive multiples of $\mathcal{R}$. For example, this holds for the sign of $\mathcal{R}(p)$, p some point, but not for the value of $\mathcal{R}(p)$. As usual, there is an exception to this rule: we have public methods to retrieve the components of the representation $\mathcal{R}$.

All calls to member functions are directly mapped to calls of corresponding methods declared by the conic class `R::Conic_2` of the representation type `R`.

Conic_2 interface and implementation [40] $\equiv$ {

```
template < class _R >
class CGAL_Conic_2 : public _R::Conic_2 {

    friend  class CGAL_Optimisation_ellipse_2<_R>;

    friend  CGAL_Window_stream&  operator << CGAL_NULL_TMPL_ARGS (
        CGAL_Window_stream&, const CGAL_Optimisation_ellipse_2<_R>&);

  public:
    // types
    typedef  _R                    R;
    typedef  typename _R::RT       RT;
    typedef  typename _R::FT       FT;
    typedef  typename _R::Conic_2  Conic_2;

    // construction
    Conic_2 constructors [41]

    // general access
    Conic_2 general access methods [42]

    // type related access
    Conic_2 type access methods [45]

    // orientation related access
    Conic_2 orientation access methods [47]

    // comparisons
    Conic_2 comparison methods [51]

    // set methods
    Conic_2 set methods [52]

  private:
    Conic_2 private methods [58]
};
}
```

This macro is invoked in definition 155.

5.3.1 Construction

We have a default constructor and a constructor to initialize a conic at coordinate level, providing the components of a representation $\mathcal{R}$. The orientation of the conic is determined by $\mathcal{R}$, as described in Section 3.3.

Conic_2 constructors [41] $\equiv$ {

```
CGAL_Conic_2 ()
{}

CGAL_Conic_2 (RT r, RT s, RT t, RT u, RT v, RT w)
    : R::Conic_2 (r, s, t, u, v, w)
{}
```

}

This macro is invoked in definition 40.

5.3.2 General Access

We provide access to the coordinates of the representation $\mathcal{R}$. Even if $\mathcal{C}_{\mathcal{R}}$ was constructed from some specific characteristic vector, the return coordinates do not necessarily belong to this vector (but at least to a nonnegative multiple of it). For example, if the coordinates are from some integer domain, the implementation might decide to divide all of them by their gcd to get smaller numbers. (In this implementation, this does not happen.)

Conic_2 general access methods [42] $+\equiv$ {

```
RT r () const
{
    return Conic_2::r();
}

RT s () const
{
    return Conic_2::s();
}

RT t () const
{
    return Conic_2::t();
}

RT u () const
{
    return Conic_2::u();
}

RT v () const
{
```

```
        return Conic_2::v();
    }

    RT w () const
    {
        return Conic_2::w();
    }

}
```
This macro is defined in definitions 42 and 43.
This macro is invoked in definition 40.

We can obtain the center of symmetry of $\mathcal{C}_\mathcal{R}$. Precondition is that $\mathcal{C}$ is not a parabola.

Conic_2 general access methods [43] $+\equiv$ {
```
    CGAL_Point_2<R> center () const
    {
        return Conic_2::center();
    }

}
```
This macro is defined in definitions 42 and 43.
This macro is invoked in definition 40.

The symmetry axis of a parabola and the two axes of an ellipse resp. a hyperbola require irrational coordinates in general, therefore they cannot be added here without further assumptions about the representation type `R`.

5.3.3 Type Related Access

Here we have access methods and predicates related to the type of $\mathcal{C}_\mathcal{R}$. We can either directly retrieve the type or ask whether $\mathcal{C}_\mathcal{R}$ is of some specific type. To realize the former, we provide a suitable enumeration type.

Conic_type declaration [44] $\equiv$ {
```
    enum CGAL_Conic_type
    {
        CGAL_HYPERBOLA = -1,
        CGAL_PARABOLA,
        CGAL_ELLIPSE
    };

}
```
This macro is invoked in definition 154.

Conic_2 type access methods [45] $+\equiv$ {
```
    CGAL_Conic_type conic_type () const
    {
        return Conic_2::conic_type();
    }
```

```
    bool is_hyperbola () const
    {
        return Conic_2::is_hyperbola();
    }

    bool is_parabola () const
    {
        return Conic_2::is_parabola();
    }

    bool is_ellipse () const
    {
        return Conic_2::is_ellipse();
    }

}
```

This macro is defined in definitions 45 and 46.
This macro is invoked in definition 40.

We have three more predicates that – in combination with the above – yield a finer access to the type, namely a test for emptiness, triviality and degeneracy, where both the empty and the trivial conic are also degenerate.

Conic_2 type access methods [46] $+\equiv$ {

```
    bool is_empty () const
    {
        return Conic_2::is_empty();
    }

    bool is_trivial () const
    {
        return Conic_2::is_trivial();
    }

    bool is_degenerate () const
    {
        return Conic_2::is_degenerate();
    }

}
```

This macro is defined in definitions 45 and 46.
This macro is invoked in definition 40.

5.3.4 Orientation Related Access

We can retrieve the orientation of a conic as defined in Section 3.3. A notable difference to the class `CGAL_Circle_2<R>` is that the orientation can be zero.

Conic_2 orientation access methods [47] $+\equiv$ {

```
CGAL_Orientation orientation () const
{
    return Conic_2::orientation ();
}
```

}

This macro is defined in definitions 47, 48, and 50.
This macro is invoked in definition 40.

The following methods classify points according to the positive and negative side of $\mathcal{C}_{\mathcal{R}}$. We can either retrieve the oriented side directly or ask whether the point lies on some specific side.

Conic_2 orientation access methods [48] $+\equiv$ {

```
CGAL_Oriented_side oriented_side (const CGAL_Point_2<R> &p) const
{
    return Conic_2::oriented_side (p);
}

bool has_on_positive_side (const CGAL_Point_2<R> &p) const
{
    return Conic_2::has_on_positive_side (p);
}

bool has_on_negative_side (const CGAL_Point_2<R> &p) const
{
    return Conic_2::has_on_negative_side (p);
}

bool has_on_boundary (const CGAL_Point_2<R> &p) const
{
    return Conic_2::has_on_boundary (p);
}

bool has_on (const CGAL_Point_2<R> &p) const
{
    return Conic_2::has_on (p);
}
```

}

This macro is defined in definitions 47, 48, and 50.
This macro is invoked in definition 40.

Then we have the classification according to convex and non-convex side. Because this is a natural generalization of the classification according to bounded and unbounded side, we 'extend' the type `CGAL_Bounded_side`.

Convex_side declaration [49] ≡ {
```
    typedef CGAL_Bounded_side CGAL_Convex_side;
    const CGAL_Convex_side CGAL_ON_CONVEX_SIDE = CGAL_ON_BOUNDED_SIDE;
    const CGAL_Convex_side CGAL_ON_NONCONVEX_SIDE =
                                                CGAL_ON_UNBOUNDED_SIDE;
}
```
This macro is invoked in definition 154.

Conic_2 orientation access methods [50] + ≡ {
```
    CGAL_Convex_side convex_side (const CGAL_Point_2<R> &p) const
    {
        return Conic_2::convex_side (p);
    }

    bool has_on_convex_side (const CGAL_Point_2<R> &p) const
    {
        return Conic_2::has_on_convex_side (p);
    }

    bool has_on_nonconvex_side (const CGAL_Point_2<R> &p) const
    {
        return Conic_2::has_on_nonconvex_side (p);
    }
}
```
This macro is defined in definitions 47, 48, and 50.
This macro is invoked in definition 40.

5.3.5 Comparison Methods

We provide tests for equality and inequality of two conics.

Conic_2 comparison methods [51] ≡ {
```
    bool operator == ( const CGAL_Conic_2<_R>& c) const
    {
        return Conic_2::operator == ( (Conic_2)c);
    }

    bool operator != ( const CGAL_Conic_2<_R>& c) const
    {
        return( ! operator == ( c));
    }
}
```
This macro is invoked in definition 40.

5.3.6 Public Set Methods

Apart from the most basic way of constructing an oriented conic from the coordinates of a representation, there are methods at a higher level, building a conic from points or

other conics. Such methods appear here; they are not given as constructors, but as 'set' functions that modify an already existing conic. But of course, there is also a set function that works on the coordinate level.

Conic_2 set methods [52] $+\equiv$ {

```
    void set (RT r, RT s, RT t,
              RT u, RT v, RT w)
    {
        Conic_2::set (r, s, t, u, v, w);
    }
```

}

This macro is defined in definitions 52, 53, 54, 55, 56, and 57.
This macro is invoked in definition 40.

Opposite conic. We can obtain the conic $\mathcal{C}_{-\mathcal{R}}$ of opposite orientation, having positive and negative sides interchanged (convex and non-convex side stay the same, of course). Note that if $\mathcal{C}_{\mathcal{R}}$ has zero orientation, so has $\mathcal{C}_{-\mathcal{R}}$.

Conic_2 set methods [53] $+\equiv$ {

```
    void set_opposite ()
    {
        Conic_2::set_opposite();
    }
```

}

This macro is defined in definitions 52, 53, 54, 55, 56, and 57.
This macro is invoked in definition 40.

Pair of lines through four points. The method `set_linepair` builds the conic equal to union of the lines $\overline{p_1p_2}$ and $\overline{p_3p_4}$. This is either a degenerate hyperbola in case the two lines are not parallel, or a degenerate parabola. The precondition is that $p_1 \neq p_2$ and $p_3 \neq p_4$. The positive side is the region to the left resp. to the right of both oriented lines $\overline{p_1p_2}, \overline{p_3p_4}$.

Conic_2 set methods [54] $+\equiv$ {

```
    void
    set_linepair (const CGAL_Point_2<R> &p1, const CGAL_Point_2<R> &p2,
                  const CGAL_Point_2<R> &p3, const CGAL_Point_2<R> &p4)
    {
        Conic_2::set_linepair (p1, p2, p3, p4);
    }
```

}

This macro is defined in definitions 52, 53, 54, 55, 56, and 57.
This macro is invoked in definition 40.

Smallest ellipse through three points. The following method `set_ellipse` generates the ellipse of smallest volume through p_1, p_2 and p_3. This is at the same time the unique ellipse through these points whose center is equal to the center of gravity of the points. Precondition is that p_1, p_2, p_3 are not collinear. The orientation of the ellipse is the orientation of the point triple.

Conic_2 set methods [55] $+\equiv$ {

```
void
set_ellipse (const CGAL_Point_2<R> &p1,
             const CGAL_Point_2<R> &p2,
             const CGAL_Point_2<R> &p3)
{
    Conic_2::set_ellipse (p1, p2, p3);
}
```

}
This macro is defined in definitions 52, 53, 54, 55, 56, and 57.
This macro is invoked in definition 40.

Some ellipse through four points in convex position. Another `set_ellipse` method constructs an ellipse from four given points, that are assumed to be in convex position (if not, the result will still be some conic through the points, but not an ellipse). The result will be just some ellipse through the points; in particular, it will usually *not* be the smallest one passing through the points. The orientation of the ellipse can be specified and defaults to positive.

Conic_2 set methods [56] $+\equiv$ {

```
void
set_ellipse (const CGAL_Point_2<R> &p1, const CGAL_Point_2<R> &p2,
             const CGAL_Point_2<R> &p3, const CGAL_Point_2<R> &p4,
             CGAL_Orientation o = CGAL_POSITIVE)
{
    Conic_2::set_ellipse (p1, p2, p3, p4, o);
}
```

}
This macro is defined in definitions 52, 53, 54, 55, 56, and 57.
This macro is invoked in definition 40.

Unique conic through five points. The method `set` generates the unique nontrivial conic containing the points p_1, p_2, p_3, p_4, p_5. Precondition is that all points are distinct. The orientation can be specified but is automatically set to zero if the resulting conic has zero orientation.

Conic_2 set methods [57] $+\equiv$ {

```
void set (const CGAL_Point_2<R> &p1, const CGAL_Point_2<R> &p2,
          const CGAL_Point_2<R> &p3, const CGAL_Point_2<R> &p4,
          const CGAL_Point_2<R> &p5,
```

```
            CGAL_Orientation o = CGAL_POSITIVE)
    {
        Conic_2::set (p1, p2, p3, p4, p5, o);
    }
}
```

This macro is defined in definitions 52, 53, 54, 55, 56, and 57.
This macro is invoked in definition 40.

5.3.7 Private Methods

Linear combination of conics. The linear combination of two conics $\mathcal{C}_{\mathcal{R}_1}$ and $\mathcal{C}_{\mathcal{R}_2}$ with coefficients a_1, a_2 is the conic with representation $\mathcal{R} := a_1\mathcal{R}_1 + a_2\mathcal{R}_2$. This is not a geometric operation, because the result depends on $\mathcal{R}_1, \mathcal{R}_2$ and not only on $\mathcal{C}_{\mathcal{R}_1}$ and $\mathcal{C}_{\mathcal{R}_2}$.

Conic_2 private methods [58] $+\equiv$ {

```
    void set_linear_combination (
        const RT &a1, const CGAL_Conic_2<R> &c1,
        const RT &a2, const CGAL_Conic_2<R> &c2)
    {
        Conic_2::set_linear_combination (a1, c1, a2, c2);
    }

}
```

This macro is defined in definitions 58, 59, 60, 61, 62, and 63.
This macro is invoked in definition 40.

Two pairs of lines through four points. The following method constructs two line-pairs through four given points p_1, p_2, p_3, p_4, assumed to be in convex position. If the points are enumerated in clockwise or counterclockwise order, these will be the line-pairs $\overline{p_1p_2} \cup \overline{p_3p_4}$ and $\overline{p_2p_3} \cup \overline{p_4p_1}$. Otherwise, points are renamed first to achieve (counter)clockwise orientation. The two resulting conics are passed to the method by reference, and their orientations depend on the points as described in the method `set_linepair`. If p_1, p_2, p_3, p_4 are not in convex position, the method still computes two line-pairs through the points, but it is not specified which ones. This is a static method.

Conic_2 private methods [59] $+\equiv$ {

```
    static void set_two_linepairs (const CGAL_Point_2<R> &p1,
                                   const CGAL_Point_2<R> &p2,
                                   const CGAL_Point_2<R> &p3,
                                   const CGAL_Point_2<R> &p4,
                                   CGAL_Conic_2<R> &pair1,
                                   CGAL_Conic_2<R> &pair2)
    {
        Conic_2::set_two_linepairs (p1, p2, p3, p4, pair1, pair2);
    }

}
```

This macro is defined in definitions 58, 59, 60, 61, 62, and 63.
This macro is invoked in definition 40.

Some ellipse from two pairs of lines. We have a method to construct an ellipse through p_1, p_2, p_3, p_4 in convex position, using two line-pairs through the points as obtained from a call to the method `set_two_linepairs` with parameters p_1, p_2, p_3, p_4. The orientation of the ellipse is not specified. In case the argument conics have not been constructed by the method `set_two_linepairs` in the described way, the resulting conic is unspecified.

Conic_2 private methods [60] $+\equiv$ {

```
void set_ellipse (const CGAL_Conic_2<R> &pair1,
                  const CGAL_Conic_2<R> &pair2)
{
    Conic_2::set_ellipse (pair1, pair2);
}
```

}

This macro is defined in definitions 58, 59, 60, 61, 62, and 63.
This macro is invoked in definition 40.

Conic from two conics and a point. Assuming that we already have two conics intersecting in four points, the following alternative `set` method is more efficient in constructing the unique nontrivial conic through them and another point p. It accepts two conics $\mathcal{C}_{\mathcal{R}_1}$ and $\mathcal{C}_{\mathcal{R}_2}$ and a point p, and computes some conic $\mathcal{C}_{\mathcal{R}}$ that goes through $\mathcal{C}_{\mathcal{R}_1} \cap \mathcal{C}_{\mathcal{R}_2} \cup \{p\}$. The method may construct the trivial conic. This happens if one of $\mathcal{C}_{\mathcal{R}_1}$ and $\mathcal{C}_{\mathcal{R}_2}$ is already trivial, $\mathcal{C}(\mathcal{R}_1) = \mathcal{C}(\mathcal{R}_2)$ holds, or if $p \in \mathcal{C}_{\mathcal{R}_1} \cap \mathcal{C}_{\mathcal{R}_2}$. In case of nonzero orientation, the actual orientation of $\mathcal{C}_{\mathcal{R}}$ is unspecified.

Conic_2 private methods [61] $+\equiv$ {

```
void set (const CGAL_Conic_2<R> &c1, const CGAL_Conic_2<R> &c2,
          const CGAL_Point_2<R> &p)
{
    Conic_2::set( c1, c2, p);  analyse();
}
```

}

This macro is defined in definitions 58, 59, 60, 61, 62, and 63.
This macro is invoked in definition 40.

Volume derivative of an ellipse. Given an ellipse $\mathcal{E}$ with representation $\mathcal{R}$ and some vector $\partial\mathcal{R} := (\partial r, \partial s, \partial t, \partial u, \partial v, \partial w)$, define

$$\mathcal{E}(\tau) = \mathcal{C}_{\mathcal{R}+\tau\partial\mathcal{R}}.$$

For small values of τ, $\mathcal{E}(\tau)$ is still an ellipse. The method `vol_derivative` computes the sign of

$$\frac{\partial}{\partial\tau} \mathrm{Vol}(\mathcal{E}(\tau)) \,|_{\tau=0} \ ,$$

i.e. decides how the volume develops when going from $\mathcal{E}(0)$ 'in direction' $\partial\mathcal{R}$. If $\mathcal{E}$ is not an ellipse, the result is meaningless.

Conic_2 private methods [62] $+\equiv$ {

```
CGAL_Sign vol_derivative (RT dr, RT ds,
                          RT dt, RT du,
                          RT dv, RT dw) const
{
    return Conic_2::vol_derivative (dr, ds, dt, du, dv, dw);
}
```

}

This macro is defined in definitions 58, 59, 60, 61, 62, and 63.
This macro is invoked in definition 40.

A related method computes the value τ^* such that $\mathrm{Vol}(\mathcal{E}(\tau))$ assumes its minimum over the set $T = \{\tau \mid \mathcal{E}(\tau) \text{ is an ellipse}\}$. Precondition is that this minimum exists. If so, τ^* is a local extremum of the volume function. τ^* might be irrational, but because the value is only used for drawing the ellipse $\mathcal{E}(\tau^*)$, a `double`-approximation suffices. As before, if $\mathcal{E}$ is not an ellipse, the result is meaningless.

Conic_2 private methods [63] $+\equiv$ {

```
double vol_minimum (RT dr, RT ds,
                    RT dt, RT du,
                    RT dv, RT dw) const
{
    return Conic_2::vol_minimum (dr, ds, dt, du, dv, dw);
}
```

}

This macro is defined in definitions 58, 59, 60, 61, 62, and 63.
This macro is invoked in definition 40.

5.3.8 I/O

Conic_2 I/O routines [64] $\equiv$ {

```
template< class _R>
ostream& operator << ( ostream& os, const CGAL_Conic_2<_R>& c)
{
    return( os << c.r() << ' ' << c.s() << ' ' << c.t() << ' '
               << c.u() << ' ' << c.v() << ' ' << c.w());
}
```

}

This macro is invoked in definition 155.

5.4 Class Templates `CGAL_ConicCPA2<PT,DA>` and `CGAL_ConicHPA2<PT,DA>`

The two classes described in this section realize the functionality of the general class `Conic_2<R>`, distinguished between Cartesian and homogeneous representation. Unlike it is practice in the CGAL kernel, the template parameters are not field type and/or ring type but point type and data accessor type. The reason is that the classes described here are

also used to build traits class adapters for the class CGAL_Min_ellipse_2<Traits>, where the adapters themselves are templatized with a point type and a data accessor type.

To realize this scheme, the representation type R of the class Conic_2<R> is enhanced with a data accessor class – it will be the canonical one, realizing access to the coordinates of a CGAL_Point_2<R>.

ConicCPA2 declaration [65] ≡ {
```
template < class PT, class DA>
class CGAL_ConicCPA2;

template < class PT, class DA>
class CGAL__Min_ellipse_2_adapterC2__Ellipse;
```
}
This macro is invoked in definition 157.

ConicHPA2 declaration [66] ≡ {
```
template < class PT, class DA>
class CGAL_ConicHPA2;

template < class PT, class DA>
class CGAL__Min_ellipse_2_adapterH2__Ellipse;
```
}
This macro is invoked in definition 156.

5.4.1 Public Interface

The interfaces look similar to the interface of the class CGAL_Conic_2<R> in the sense that for any public (private) member function of CGAL_Conic_2<R>, we have a corresponding public (private) member function here. These are the *high-level* member functions.

In addition, there are private data members to store the conic representation and a couple of additional *low-level* protected member functions. The reason for making them protected is that we do not want them to show up explicitly in the interface of the class CGAL_Conic_2<R>, but on the other hand, friends of CGAL_Conic_2<R> should be able to use them.

Note: We implement the classes in their interface to cope with insufficiencies of the GNU compiler concerning scope operators in typenames. Because the implementations of most member functions are very similar for both representations, we always write them down in parallel.

ConicCPA2 interface and implementation [67] ≡ {
```
template < class _PT, class _DA>
class CGAL_ConicCPA2
{
  public:
```

```
    // types
    typedef           _PT      PT;
    typedef           _DA      DA;
    typedef  typename _DA::RT  RT;

  private:
    // friends
    friend class CGAL_Conic_2< CGAL_Homogeneous<RT> >;
    friend class CGAL__Min_ellipse_2_adapterC2__Ellipse<PT,DA>;

    // data members
    ConicCPA2 private data members [69]

    // private member functions
    ConicCPA2 private member functions [101]

  protected:
    // protected member functions
    ConicCPA2 protected member functions [71]

  public:
    // public member functions
    ConicCPA2 public member functions [83]
};
}
```

This macro is invoked in definition 157.

ConicHPA2 interface and implementation [68] $\equiv$ {

```
template < class _PT, class _DA>
class CGAL_ConicHPA2
{
  public:
    // types
    typedef           _PT      PT;
    typedef           _DA      DA;
    typedef  typename _DA::FT  FT;

  private:
    // friends
    friend class CGAL_Conic_2< CGAL_Cartesian<FT> >;
    friend class CGAL__Min_ellipse_2_adapterH2__Ellipse<PT,DA>;

    // data members
    ConicHPA2 private data members [70]

    // private member functions
    ConicHPA2 private member functions [102]
```

```
    protected:
      // protected member functions
      ConicHPA2 protected member functions [72]

    public:
      // public member functions
      ConicHPA2 public member functions [84]
  };
}
```
This macro is invoked in definition 156.

5.4.2 Private Data Members

An oriented conic $\mathcal{C}_\mathcal{R}$ is stored by its representation $\mathcal{R}$ and certain *derived data*. These data are

- the type of $\mathcal{C}_\mathcal{R}$,
- the orientation of $\mathcal{C}_\mathcal{R}$,
- degeneracy information, consisting of three flags indicating whether $\mathcal{C}_\mathcal{R}$ is empty, trivial or degenerate.

Although type, orientation and degeneracy information can be retrieved from $\mathcal{R}$, it is more efficient to store them, because for example, repeated convex side tests on the same conic but with different points access these data over and over again.

ConicCPA2 private data members [69] $\equiv$ {
```
  DA                  dao;
  RT                  _r, _s, _t, _u, _v, _w;
  CGAL_Conic_type     type;
  CGAL_Orientation    o;
  bool                empty, trivial, degenerate;
}
```
This macro is invoked in definition 67.

ConicHPA2 private data members [70] $\equiv$ {
```
  DA                  dao;
  FT                  _r, _s, _t, _u, _v, _w;
  CGAL_Conic_type     type;
  CGAL_Orientation    o;
  bool                empty, trivial, degenerate;
}
```
This macro is invoked in definition 68.

5.4.3 Low-level Private Member Functions

Let's start with the low-level members that do not have a counterpart in the interface of the class `CGAL_Conic_2<R>`.

Determinant. The function `det` just computes $\det(\mathcal{R}) = 4rs - t^2$, and as mentioned in Section 3.1, this value determines the type of the conic.

ConicCPA2 protected member functions [71] $+\equiv$ {

```
RT det () const
{
    return RT(4)*s()*r() - t()*t();
}
```

}
This macro is defined in definitions 71, 73, and 81.
This macro is invoked in definition 67.

ConicHPA2 protected member functions [72] $+\equiv$ {

```
FT det () const
{
    return FT(4)*s()*r() - t()*t();
}
```

}
This macro is defined in definitions 72, 74, and 82.
This macro is invoked in definition 68.

Conic analysis. This method is the most important low-level method. It initializes the derived data from the representation $\mathcal{R}$, by first determining the conic's type and then handling the three possible types in a `case` statement.

ConicCPA2 protected member functions [73] $+\equiv$ {

```
void analyse( )
{
    RT d = det();
    type = (CGAL_Conic_type)(CGAL_sign(d));
    switch (type) {
    case CGAL_HYPERBOLA:
        {
            analyse hyperbola, homogeneous case [75]
        }
        break;
    case CGAL_PARABOLA:
        {
            analyse parabola, homogeneous case [79]
        }
```

```
            break;
        case CGAL_ELLIPSE:
            {
                analyse ellipse, homogeneous case [77]
            }
            break;
        }
    }

}
```

This macro is defined in definitions 71, 73, and 81.
This macro is invoked in definition 67.

ConicHPA2 protected member functions [74] $+\equiv$ {

```
    void analyse( )
    {
        FT d = det();
        type = (CGAL_Conic_type)(CGAL_sign(d));
        switch (type) {
        case CGAL_HYPERBOLA:
            {
                analyse hyperbola, cartesian case [76]
            }
            break;
        case CGAL_PARABOLA:
            {
                analyse parabola, cartesian case [80]
            }
            break;
        case CGAL_ELLIPSE:
            {
                analyse ellipse, cartesian case [78]
            }
            break;
        }
    }

}
```

This macro is defined in definitions 72, 74, and 82.
This macro is invoked in definition 68.

Let us first deal with the case where $\mathcal{C}_\mathcal{R}$ is a hyperbola or ellipse. Then we have seen in Section 3.2 that $\mathcal{C}(\mathcal{R})$ has a center of symmetry c and can be written in the form

$$\{p \mid (p-c)^T M(p-c) + 2w - c^T Mc = 0\},$$

Moreover, $\mathcal{C}(\mathcal{R})$ is degenerate if the center lies on the conic. This is the case if and only if $z := 2w - c^T Mc = 0$. To compute this value z, we go back to the formulas of Section 3.2,

where we have seen that

$$c = -M^{-1} \begin{pmatrix} u \\ v \end{pmatrix},$$

therefore

$$\begin{aligned} c^T M c &= (u,v) M^{-1} \begin{pmatrix} u \\ v \end{pmatrix} \\ &= \frac{1}{\det(M)} (u,v) \begin{pmatrix} 2s & -t \\ -t & 2r \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} \\ &= \frac{1}{\det(M)} (2u^2 s + 2v^2 r - 2uvt). \end{aligned}$$

This means,

$$z = 2\left(w - \frac{1}{\det(M)}(u^2 s + v^2 r - uvt)\right). \tag{8}$$

To avoid divisions, we consider the value $z' = \det(M)z/2$ which satisfies

$$z' = \det(M)w - u^2 s - v^2 r + uvt.$$

z' has the same sign as z in case of an ellipse, and opposite sign in case of a hyperbola.

To proceed further, we note that only a parabola can be trivial. Moreover, a hyperbola cannot be empty (the matrix M is indefinite, therefore $x^T M x$ assumes arbitrary real values). In case of an ellipse, M is either positive or negative definite, meaning $r > 0$ or $r < 0$. In the former case, the ellipse is empty if and only if $z > 0$, in the latter case, $z < 0$ leads to an empty ellipse. Summarizing, the ellipse is empty iff $rz > 0$, equivalently $rz' > 0$

Now consider orientation. A hyperbola is in positive orientation if and only if its center is on the negative side, equivalently $z < 0$, or $z' > 0$. Similarly, the orientation is negative in case of $z' < 0$. For $z = z' = 0$, the hyperbola is degenerate, and so its orientation is zero.

A non-degenerate ellipse, on the other hand, has positive orientation if and only if its center is in the positive side, equivalently, if $z > 0$, or $z' > 0$. This is equivalent to M being negative definite, or $r < 0$. In case of the degenerate ellipse $\mathcal{E} = \{c\}$, we have $z = z' = 0$, and the orientation is positive if and only if the negative side is nonempty (the positive side must agree with the empty convex side). As before, this is equivalent to M being negative definite, or $r < 0$.

In contrast to this, the empty ellipse has positive orientation if and only if the negative side is empty (the positive side must agree with the convex side, which is the whole plane in this case). For this, we get the equivalent condition $r > 0$.

analyse hyperbola, homogeneous case [75] $\equiv$ {

```
    trivial = empty = false;
    RT z_prime = d*w() - u()*u()*s() - v()*v()*r() + u()*v()*t();
    o = (CGAL_Orientation)(CGAL_sign (z_prime));
    degenerate = (o == CGAL_ZERO);
```

}

This macro is invoked in definition 73.

analyse hyperbola, cartesian case [76] ≡ {

```
    trivial = empty = false;
    FT z_prime = d*w() - u()*u()*s() - v()*v()*r() + u()*v()*t();
    o = (CGAL_Orientation)(CGAL_sign (z_prime));
    degenerate = (o == CGAL_ZERO);
}
```

This macro is invoked in definition 74.

analyse ellipse, homogeneous case [77] ≡ {

```
    trivial = false;
    RT z_prime = d*w() - u()*u()*s() - v()*v()*r() + u()*v()*t();
    if (CGAL_is_positive (r())) {
        empty = CGAL_is_positive(CGAL_sign (z_prime));
        empty ? o = CGAL_POSITIVE : o = CGAL_NEGATIVE;
    } else {
        empty = CGAL_is_negative(CGAL_sign (z_prime));
        empty ? o = CGAL_NEGATIVE : o = CGAL_POSITIVE;
    }
    degenerate = empty || CGAL_is_zero (z_prime);
}
```

This macro is invoked in definition 73.

analyse ellipse, cartesian case [78] ≡ {

```
    trivial = false;
    FT z_prime = d*w() - u()*u()*s() - v()*v()*r() + u()*v()*t();
    if (CGAL_is_positive (r())) {
        empty = CGAL_is_positive(CGAL_sign (z_prime));
        empty ? o = CGAL_POSITIVE : o = CGAL_NEGATIVE;
    } else {
        empty = CGAL_is_negative(CGAL_sign (z_prime));
        empty ? o = CGAL_NEGATIVE : o = CGAL_POSITIVE;
    }
    degenerate = empty || CGAL_is_zero (z_prime);
}
```

This macro is invoked in definition 74.

In the parabola case, we proceed as follows, first observing that $\det(M) = 0$ implies $r, s \geq 0$ or $r, s \leq 0$. Assume that $r \neq 0$. Then the conic equation (1) can be solved for x, obtaining

$$x = \frac{-ty - u \pm \sqrt{2(tu - 2rv)y + u^2 - 4rw}}{2r}.$$

The parabola is non-degenerate exactly if the factor $(tu - 2rv)$ is nonzero, where we obtain a curved object. For $tu = 2rv$, the parabola is either empty (this happens for $u^2 < 4rw$ when the radicant becomes negative), a single line (if $u^2 = 4rw$), or a pair of lines (for $u^2 > 4rw$). Let us treat the degenerate case first.

In the empty case, the parabola has only one nonempty side, and the orientation is determined by w. $w > 0$ means positive orientation (because then the point $(0, 0)$ is on the

positive side, which therefore equals the convex side in this case), $w < 0$ means negative orientation. Because of $u^2 < 4rw$, the case $w = 0$ cannot occur.

In case of $u^2 = 4rw$, the conic is given by

$$\mathcal{R}(p) = r(x + \frac{ty + u}{2r})^2,$$

and the orientation is positive if and only if $r > 0$ (in which case every point $p = (x, y)$ is on the positive side, equivalently on the convex side).

For $u^2 > 4rw$, we get a pair of parallel lines, of zero orientation (because both positive and negative sides are nonempty, while the non-convex side is empty).

We can argue completely similar in the case $s \neq 0$. Here we get a degeneracy exactly if $tv = 2su$, and the discriminant $v^2 - 4sw$ determines whether the parabola is empty, equal to one line or to a pair of lines.

If $r = s = 0$ (implying $t = 0$), we have the trivial conic if all other parameters are also zero. The trivial conic has always positive orientation (because both positive and convex side are empty). If $u = v = 0$ but $w \neq 0$, we get the empty conic, where the orientation is given by w. In any other case, we obtain a single line, this time with zero orientation, because it has nonempty positive and negative side but empty non-convex side.

Now consider the non-degenerate case. We claim that the orientation is positive if and only if $r, s \leq 0$. To see this, note that in this case, the parabola can be written in the form

$$\mathcal{R}(p) = -(\sqrt{-r}x - \sqrt{-s}y)^2 + ux + vy + w.$$

$\mathcal{R}(p)$ is a concave function which implies that if $\mathcal{R}(p_1), \mathcal{R}(p_2) > 0$, then also $\mathcal{R}(p) > 0$, p a convex combination of p_1, p_2. This means that the positive side is a convex set, thus equal to the convex side.

analyse parabola, homogeneous case [79] $\equiv$ {

```
if (!CGAL_is_zero (r())) {
    trivial         = false;
    degenerate      = (t()*u() == RT(2)*r()*v());
    if (degenerate) {
        CGAL_Sign discr = (CGAL_Sign)
                          CGAL_sign(u()*u()-RT(4)*r()*w());
        switch (discr) {
            case CGAL_NEGATIVE:
                empty = true;
                o = (CGAL_Orientation)(CGAL_sign (w()));
                break;
            case CGAL_ZERO:
                empty = false;
                o = (CGAL_Orientation)(CGAL_sign (r()));
                break;
            case CGAL_POSITIVE:
                empty = false;
```

```
                o = CGAL_ZERO;
                break;
            }
        } else {
            empty = false;
            o = (CGAL_Orientation)(-CGAL_sign (r()));
        }
    } else if (!CGAL_is_zero (s())) {
        trivial         = false;
        degenerate      = (t()*v() == RT(2)*s()*u());
        if (degenerate) {
            CGAL_Sign discr = (CGAL_Sign)
                              CGAL_sign(v()*v()-RT(4)*s()*w());
            switch (discr) {
                case CGAL_NEGATIVE:
                    empty = true;
                    o = (CGAL_Orientation)(CGAL_sign (w()));
                    break;
                case CGAL_ZERO:
                    empty = false;
                    o = (CGAL_Orientation)(CGAL_sign (s()));
                    break;
                case CGAL_POSITIVE:
                    empty = false;
                    o = CGAL_ZERO;
                    break;
            }
        } else {
            empty = false;
            o = (CGAL_Orientation)(-CGAL_sign (s()));
        }
    } else { // r=0, s=0
        degenerate      = true;
        bool uv_zero    = CGAL_is_zero (u()) && CGAL_is_zero (v());
        trivial         = uv_zero && CGAL_is_zero (w());
        empty           = uv_zero && !trivial;
        if (empty)
            o = (CGAL_Orientation)(CGAL_sign (w()));
        else if (trivial)
            o = CGAL_POSITIVE;
        else
            o = CGAL_ZERO;
    }
}
```

This macro is invoked in definition 73.

analyse parabola, cartesian case [80] $\equiv$ {

```
if (!CGAL_is_zero (r())) {
    trivial         = false;
    degenerate      = (t()*u() == FT(2)*r()*v());
    if (degenerate) {
        CGAL_Sign discr = (CGAL_Sign)
                          CGAL_sign(u()*u()-FT(4)*r()*w());
        switch (discr) {
            case CGAL_NEGATIVE:
                empty = true;
                o = (CGAL_Orientation)(CGAL_sign (w()));
                break;
            case CGAL_ZERO:
                empty = false;
                o = (CGAL_Orientation)(CGAL_sign (r()));
                break;
            case CGAL_POSITIVE:
                empty = false;
                o = CGAL_ZERO;
                break;
        }
    } else {
        empty = false;
        o = (CGAL_Orientation)(-CGAL_sign (r()));
    }
} else if (!CGAL_is_zero (s())) {
    trivial         = false;
    degenerate      = (t()*v() == FT(2)*s()*u());
    if (degenerate) {
        CGAL_Sign discr = (CGAL_Sign)
                          CGAL_sign(v()*v()-FT(4)*s()*w());
        switch (discr) {
            case CGAL_NEGATIVE:
                empty = true;
                o = (CGAL_Orientation)(CGAL_sign (w()));
                break;
            case CGAL_ZERO:
                empty = false;
                o = (CGAL_Orientation)(CGAL_sign (s()));
                break;
            case CGAL_POSITIVE:
                empty = false;
                o = CGAL_ZERO;
                break;
        }
    } else {
        empty = false;
```

```
            o = (CGAL_Orientation)(-CGAL_sign (s()));
        }
    } else { // r=0, s=0
        degenerate      = true;
        bool uv_zero    = CGAL_is_zero (u()) && CGAL_is_zero (v());
        trivial         = uv_zero && CGAL_is_zero (w());
        empty           = uv_zero && !trivial;
        if (empty)
            o = (CGAL_Orientation)(CGAL_sign (w()));
        else if (trivial)
            o = CGAL_POSITIVE;
        else
            o = CGAL_ZERO;
    }
}
```

This macro is invoked in definition 74.

Conic evaluation. For $\mathcal{C}_\mathcal{R}$ given by $R = (r, s, t, u, v, w)$ and a point $p = (x, y, h)$, the homogeneous evaluation returns the value $\mathcal{R}(p) = rx^2 + sy^2 + txy + uxh + vyh + wh^2$.

ConicCPA2 protected member functions [81] $+\equiv$ {

```
    RT evaluate (const PT &p) const
    {
        RT x, y, h;
        dao.get (p, x, y, h);
        return  r()*x*x + s()*y*y + t()*x*y + u()*x*h + v()*y*h + w()*h*h;
    }

}
```

This macro is defined in definitions 71, 73, and 81.
This macro is invoked in definition 67.

The Cartesian version is obtained for $h = 1$, i.e. it computes the value $\mathcal{R}(p) = rx^2 + sy^2 + txy + ux + vy + w$.

ConicHPA2 protected member functions [82] $+\equiv$ {

```
    FT evaluate (const PT &p) const
    {
        FT x, y;
        dao.get (p, x, y);
        return r()*x*x + s()*y*y + t()*x*y + u()*x + v()*y + w();
    }

}
```

This macro is defined in definitions 72, 74, and 82.
This macro is invoked in definition 68.

5.4.4 Construction

The construction from the representation proceeds by first setting the vector components and then analysing the conic to obtain the derived data.

ConicCPA2 public member functions [83] $+\equiv$ {

```
CGAL_ConicCPA2 ( const DA& da = DA()) : dao( da) { }

CGAL_ConicCPA2 ( RT r, RT s, RT t, RT u, RT v, RT w,
                 const DA& da = DA())
    : dao( da), _r(r), _s(s), _t(t), _u(u), _v(v), _w(w)
{
    analyse();
}
```

}

This macro is defined in definitions 83, 85, 87, 89, 91, 93, 95, 97, 99, 117, 119, 121, 123, 125, and 127.
This macro is invoked in definition 67.

ConicHPA2 public member functions [84] $+\equiv$ {

```
CGAL_ConicHPA2 ( const DA& da = DA()) : dao( da) { }

CGAL_ConicHPA2 ( FT r, FT s, FT t, FT u, FT v, FT w,
                 const DA& da = DA())
    : dao( da), _r(r), _s(s), _t(t), _u(u), _v(v), _w(w)
{
    analyse();
}
```

}

This macro is defined in definitions 84, 86, 88, 90, 92, 94, 96, 98, 100, 118, 120, 122, 124, 126, and 128.
This macro is invoked in definition 68.

5.4.5 Data Accessor

ConicCPA2 public member functions [85] $+\equiv$ {

```
const DA&  da() const
{
    return dao;
}
```

}

This macro is defined in definitions 83, 85, 87, 89, 91, 93, 95, 97, 99, 117, 119, 121, 123, 125, and 127.
This macro is invoked in definition 67.

ConicHPA2 public member functions [86] $+\equiv$ {

```
const DA&  da() const
{
```

```
        return dao;
    }

}
```

This macro is defined in definitions 84, 86, 88, 90, 92, 94, 96, 98, 100, 118, 120, 122, 124, 126, and 128.
This macro is invoked in definition 68.

5.4.6 General Access

The coordinate access is straightforward.

ConicCPA2 public member functions [87] $+\equiv$ {

```
    RT r() const { return _r;}
    RT s() const { return _s;}
    RT t() const { return _t;}
    RT u() const { return _u;}
    RT v() const { return _v;}
    RT w() const { return _w;}

}
```

This macro is defined in definitions 83, 85, 87, 89, 91, 93, 95, 97, 99, 117, 119, 121, 123, 125, and 127.
This macro is invoked in definition 67.

ConicHPA2 public member functions [88] $+\equiv$ {

```
    FT r() const { return _r;}
    FT s() const { return _s;}
    FT t() const { return _t;}
    FT u() const { return _u;}
    FT v() const { return _v;}
    FT w() const { return _w;}

}
```

This macro is defined in definitions 84, 86, 88, 90, 92, 94, 96, 98, 100, 118, 120, 122, 124, 126, and 128.
This macro is invoked in definition 68.

To obtain the center, recall from Section 3.2 that it is given by the point

$$c = -M^{-1} \begin{pmatrix} u \\ v \end{pmatrix} = -\frac{1}{\det(M)} \begin{pmatrix} 2s & -t \\ -t & 2r \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = -\frac{1}{\det(M)} \begin{pmatrix} 2s \cdot u - t \cdot v \\ 2r \cdot v - t \cdot u \end{pmatrix}.$$

In the homogeneous representation, the value $-\det(M)$ serves as the h-component of the center, in the Cartesian representation, we divide by it.

ConicCPA2 public member functions [89] $+\equiv$ {

```
    PT center () const
    {
        CGAL_kernel_precondition (type != CGAL_PARABOLA);
        PT p;
        RT two = RT(2);
```

```
        dao.set( p, two*s()*u() - t()*v(), two*r()*v() - t()*u(), -det());
        return p;
    }

}
```

This macro is defined in definitions 83, 85, 87, 89, 91, 93, 95, 97, 99, 117, 119, 121, 123, 125, and 127.
This macro is invoked in definition 67.

ConicHPA2 public member functions [90] $+\equiv$ {

```
    PT center () const
    {
        CGAL_kernel_precondition (type != CGAL_PARABOLA);
        PT p;
        FT two = FT(2);
        FT div = -det();
        dao.set( p, (two*s()*u() - t()*v()) / div,
                    (two*r()*v() - t()*u()) / div);
        return p;
    }

}
```

This macro is defined in definitions 84, 86, 88, 90, 92, 94, 96, 98, 100, 118, 120, 122, 124, 126, and 128.
This macro is invoked in definition 68.

5.4.7 Type Related Access

Because the conic stores its type and degeneracy information, this is straightforward.

ConicCPA2 public member functions [91] $+\equiv$ {

```
    CGAL_Conic_type conic_type () const
    {
        return type;
    }

    bool is_hyperbola () const
    {
        return (type == CGAL_HYPERBOLA);
    }

    bool is_parabola () const
    {
        return (type == CGAL_PARABOLA);
    }

    bool is_ellipse () const
    {
        return (type == CGAL_ELLIPSE);
    }
```

```
    bool is_empty () const
    {
        return empty;
    }

    bool is_trivial () const
    {
        return trivial;
    }

    bool is_degenerate () const
    {
        return degenerate;
    }

}
```

This macro is defined in definitions 83, 85, 87, 89, 91, 93, 95, 97, 99, 117, 119, 121, 123, 125, and 127.
This macro is invoked in definition 67.

ConicHPA2 public member functions [92] $+\equiv$ {

```
    CGAL_Conic_type conic_type () const
    {
        return type;
    }

    bool is_hyperbola () const
    {
        return (type == CGAL_HYPERBOLA);
    }

    bool is_parabola () const
    {
        return (type == CGAL_PARABOLA);
    }

    bool is_ellipse () const
    {
        return (type == CGAL_ELLIPSE);
    }

    bool is_empty () const
    {
        return empty;
    }

    bool is_trivial () const
    {
```

```
        return trivial;
    }

    bool is_degenerate () const
    {
        return degenerate;
    }

}
```
This macro is defined in definitions 84, 86, 88, 90, 92, 94, 96, 98, 100, 118, 120, 122, 124, 126, and 128.
This macro is invoked in definition 68.

5.4.8 Orientation Related Access

ConicCPA2 public member functions [93] $+\equiv$ {
```
    CGAL_Orientation orientation () const
    {
        return o;
    }

}
```
This macro is defined in definitions 83, 85, 87, 89, 91, 93, 95, 97, 99, 117, 119, 121, 123, 125, and 127.
This macro is invoked in definition 67.

ConicHPA2 public member functions [94] $+\equiv$ {
```
    CGAL_Orientation orientation () const
    {
        return o;
    }

}
```
This macro is defined in definitions 84, 86, 88, 90, 92, 94, 96, 98, 100, 118, 120, 122, 124, 126, and 128.
This macro is invoked in definition 68.

The orientation queries just evaluate the conic at the given point, using the private method `evaluate`. Recall that p is in the positive resp. negative side iff $\mathcal{R}(p) > 0$ resp. $\mathcal{R}(p) < 0$.

ConicCPA2 public member functions [95] $+\equiv$ {
```
    CGAL_Oriented_side oriented_side (const PT& p) const
    {
        return (CGAL_Oriented_side)(CGAL_sign (evaluate (p)));
    }

    bool has_on_positive_side (const PT& p) const
    {
        return (CGAL_is_positive (evaluate(p)));
    }

    bool has_on_negative_side (const PT& p) const
```

```
    {
        return (CGAL_is_negative (evaluate(p)));
    }

    bool has_on_boundary (const PT& p) const
    {
       return (CGAL_is_zero (evaluate(p)));
    }

    bool has_on (const PT& p) const
    {
       return (CGAL_is_zero (evaluate(p)));
    }

}
```

This macro is defined in definitions 83, 85, 87, 89, 91, 93, 95, 97, 99, 117, 119, 121, 123, 125, and 127.
This macro is invoked in definition 67.

ConicHPA2 public member functions [96] $+\equiv$ {

```
    CGAL_Oriented_side oriented_side (const PT& p) const
    {
        return (CGAL_Oriented_side)(CGAL_sign (evaluate (p)));
    }

    bool has_on_positive_side (const PT& p) const
    {
        return (CGAL_is_positive (evaluate(p)));
    }

    bool has_on_negative_side (const PT& p) const
    {
        return (CGAL_is_negative (evaluate(p)));
    }

    bool has_on_boundary (const PT& p) const
    {
       return (CGAL_is_zero (evaluate(p)));
    }

    bool has_on (const PT& p) const
    {
       return (CGAL_is_zero (evaluate(p)));
    }

}
```

This macro is defined in definitions 84, 86, 88, 90, 92, 94, 96, 98, 100, 118, 120, 122, 124, 126, and 128.
This macro is invoked in definition 68.

Then we have the convex side queries. Under nonzero orientation, the side is determined

by a conic evaluation. If the orientation is zero, we know that the non-convex side is empty, see Section 3.3.

ConicCPA2 public member functions [97] $+\equiv$ {

```
CGAL_Convex_side convex_side (const PT &p) const
{
    switch (o) {
    case CGAL_POSITIVE:
        return (CGAL_Convex_side)(-CGAL_sign (evaluate (p)));
    case CGAL_NEGATIVE:
        return (CGAL_Convex_side)( CGAL_sign (evaluate (p)));
    case CGAL_ZERO:
        return (CGAL_Convex_side)(
                    -CGAL_sign (CGAL_abs (evaluate(p))));
    }
    // keeps g++ happy
    return( CGAL_Convex_side( 0));
}

bool has_on_convex_side (const PT &p) const
{
    return (convex_side (p) == CGAL_ON_CONVEX_SIDE);
}

bool has_on_nonconvex_side (const PT &p) const
{
    return (convex_side (p) == CGAL_ON_NONCONVEX_SIDE);
}

}
```

This macro is defined in definitions 83, 85, 87, 89, 91, 93, 95, 97, 99, 117, 119, 121, 123, 125, and 127.
This macro is invoked in definition 67.

ConicHPA2 public member functions [98] $+\equiv$ {

```
CGAL_Convex_side convex_side (const PT &p) const
{
    switch (o) {
    case CGAL_POSITIVE:
        return (CGAL_Convex_side)(-CGAL_sign (evaluate (p)));
    case CGAL_NEGATIVE:
        return (CGAL_Convex_side)( CGAL_sign (evaluate (p)));
    case CGAL_ZERO:
        return (CGAL_Convex_side)(
                    -CGAL_sign (CGAL_abs (evaluate(p))));
    }
    // keeps g++ happy
    return( CGAL_Convex_side( 0));
}
```

```
    bool has_on_convex_side (const PT &p) const
    {
        return (convex_side (p) == CGAL_ON_CONVEX_SIDE);
    }

    bool has_on_nonconvex_side (const PT &p) const
    {
        return (convex_side (p) == CGAL_ON_NONCONVEX_SIDE);
    }

}
```

This macro is defined in definitions 84, 86, 88, 90, 92, 94, 96, 98, 100, 118, 120, 122, 124, 126, and 128.
This macro is invoked in definition 68.

5.4.9 Comparison Methods

We provide tests for equality and inequality of two conics.

ConicCPA2 public member functions [99] $+\equiv$ {

```
    bool operator == ( const CGAL_ConicCPA2<_PT,_DA>& c) const
    {
        // find coefficient != 0
        RT  factor1;
        if ( ! CGAL_is_zero( r())) factor1 = r(); else
        if ( ! CGAL_is_zero( s())) factor1 = s(); else
        if ( ! CGAL_is_zero( t())) factor1 = t(); else
        if ( ! CGAL_is_zero( u())) factor1 = u(); else
        if ( ! CGAL_is_zero( v())) factor1 = v(); else
        if ( ! CGAL_is_zero( w())) factor1 = w(); else
        CGAL_optimisation_assertion_msg( false, "all coefficients zero");

        // find coefficient != 0
        RT  factor2;
        if ( ! CGAL_is_zero( c.r())) factor2 = c.r(); else
        if ( ! CGAL_is_zero( c.s())) factor2 = c.s(); else
        if ( ! CGAL_is_zero( c.t())) factor2 = c.t(); else
        if ( ! CGAL_is_zero( c.u())) factor2 = c.u(); else
        if ( ! CGAL_is_zero( c.v())) factor2 = c.v(); else
        if ( ! CGAL_is_zero( c.w())) factor2 = c.w(); else
        CGAL_optimisation_assertion_msg( false, "all coefficients zero");

        return(    ( r()*factor2 == c.r()*factor1)
                && ( s()*factor2 == c.s()*factor1)
                && ( t()*factor2 == c.t()*factor1)
                && ( u()*factor2 == c.u()*factor1)
                && ( v()*factor2 == c.v()*factor1)
```

```
                && ( w()*factor2 == c.w()*factor1));
    }

}
```

This macro is defined in definitions 83, 85, 87, 89, 91, 93, 95, 97, 99, 117, 119, 121, 123, 125, and 127.
This macro is invoked in definition 67.

ConicHPA2 public member functions [100] $+\equiv$ {

```
    bool operator == ( const CGAL_ConicHPA2<_PT,_DA>& c) const
    {
        // find coefficient != 0
        FT  factor1;
        if ( ! CGAL_is_zero( r())) factor1 = r(); else
        if ( ! CGAL_is_zero( s())) factor1 = s(); else
        if ( ! CGAL_is_zero( t())) factor1 = t(); else
        if ( ! CGAL_is_zero( u())) factor1 = u(); else
        if ( ! CGAL_is_zero( v())) factor1 = v(); else
        if ( ! CGAL_is_zero( w())) factor1 = w(); else
        CGAL_optimisation_assertion_msg( false, "all coefficients zero");

        // find coefficient != 0
        FT  factor2;
        if ( ! CGAL_is_zero( c.r())) factor2 = c.r(); else
        if ( ! CGAL_is_zero( c.s())) factor2 = c.s(); else
        if ( ! CGAL_is_zero( c.t())) factor2 = c.t(); else
        if ( ! CGAL_is_zero( c.u())) factor2 = c.u(); else
        if ( ! CGAL_is_zero( c.v())) factor2 = c.v(); else
        if ( ! CGAL_is_zero( c.w())) factor2 = c.w(); else
        CGAL_optimisation_assertion_msg( false, "all coefficients zero");

        return(    ( r()*factor2 == c.r()*factor1)
                && ( s()*factor2 == c.s()*factor1)
                && ( t()*factor2 == c.t()*factor1)
                && ( u()*factor2 == c.u()*factor1)
                && ( v()*factor2 == c.v()*factor1)
                && ( w()*factor2 == c.w()*factor1));
    }

}
```

This macro is defined in definitions 84, 86, 88, 90, 92, 94, 96, 98, 100, 118, 120, 122, 124, 126, and 128.
This macro is invoked in definition 68.

5.4.10 Private Methods

A main difference between the public and the private set methods is that the private ones do not analyse the conic. If a conic is constructed in a sequence of calls to set functions, it is more efficient to do an analysis only once for the final result, rather than analysing any

intermediate result. Therefore, the private set functions also do not allow to specify an orientation, because in order to orient the conic, an analysis would be necessary. Under this scheme, caution is in place, of course: a conic constructed from a private set method is not fully initialized, and calls to type and orientation related access functions return undefined results. `friend` classes need to care for this, whenever they call a private set method.

Linear combination of conics. *ConicCPA2 private member functions* [101] $+\equiv$ {

```
    void
    set_linear_combination (const RT &a1, const CGAL_ConicCPA2<PT,DA> &c1,
                            const RT &a2, const CGAL_ConicCPA2<PT,DA> &c2)
    {
        _r = a1 * c1.r() + a2 * c2.r();
        _s = a1 * c1.s() + a2 * c2.s();
        _t = a1 * c1.t() + a2 * c2.t();
        _u = a1 * c1.u() + a2 * c2.u();
        _v = a1 * c1.v() + a2 * c2.v();
        _w = a1 * c1.w() + a2 * c2.w();
    }

}
```

This macro is defined in definitions 101, 105, 107, 109, 111, and 114.
This macro is invoked in definition 67.

ConicHPA2 private member functions [102] $+\equiv$ {

```
    void
    set_linear_combination (const FT &a1, const CGAL_ConicHPA2<PT,DA> &c1,
                            const FT &a2, const CGAL_ConicHPA2<PT,DA> &c2)
    {
        _r = a1 * c1.r() + a2 * c2.r();
        _s = a1 * c1.s() + a2 * c2.s();
        _t = a1 * c1.t() + a2 * c2.t();
        _u = a1 * c1.u() + a2 * c2.u();
        _v = a1 * c1.v() + a2 * c2.v();
        _w = a1 * c1.w() + a2 * c2.w();
    }

}
```

This macro is defined in definitions 102, 106, 108, 110, 112, and 115.
This macro is invoked in definition 68.

Two pairs of lines through four points. Here is the method to get two line-pairs from four given points. It just brings the points into (counter)clockwise order and then calls the `set_linepair` method for their two conic arguments, supplying the points in the right order. The reordering needs access to the orientations of certain point triples. For

points $p_i = (x_i, y_i, h_i)$, $i = 1 \ldots 3$, this orientation is given by the sign of the determinant

$$\det \begin{pmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ h_1 & h_2 & h_3 \end{pmatrix},$$

see Lemma 5.1 below. The following macros define this determinant (the Cartesian version is obtained by setting $h_i = 1, i = 1 \ldots 3$).

h_orientation [103] ($\diamond$3)$\mathbf{M} \equiv$ {

```
(CGAL_Orientation)(CGAL_sign
  (-h◇1*x◇3*y◇2+h◇3*x◇1*y◇2
   +h◇1*x◇2*y◇3-h◇2*x◇1*y◇3
   +h◇2*x◇3*y◇1-h◇3*x◇2*y◇1))
```

}

This macro is invoked in definitions 105, 105, and 105.

c_orientation [104] ($\diamond$3)$\mathbf{M} \equiv$ {

```
(CGAL_Orientation)(CGAL_sign
  (-x◇3*y◇2+x◇1*y◇2
   +x◇2*y◇3-x◇1*y◇3
   +x◇3*y◇1-x◇2*y◇1))
```

}

This macro is invoked in definitions 106, 106, and 106.

ConicCPA2 private member functions [105] $+\equiv$ {

```
static void set_two_linepairs (const PT &p1,
                               const PT &p2,
                               const PT &p3,
                               const PT &p4,
                               CGAL_ConicCPA2<PT,DA> &pair1,
                               CGAL_ConicCPA2<PT,DA> &pair2)
{
    RT x1, y1, h1, x2, y2, h2, x3, y3, h3, x4, y4, h4;
    const DA& da = pair1.da();
    da.get (p1, x1, y1, h1);
    da.get (p2, x2, y2, h2);
    da.get (p3, x3, y3, h3);
    da.get (p4, x4, y4, h4);

    CGAL_Orientation side1_24 = h_orientation [103] ('2', '4', '1'),
                    side3_24 = h_orientation [103] ('2', '4', '3');
    if (side1_24 != side3_24) {
        // (counter)clockwise order
        pair1.set_linepair (p1, p2, p3, p4);
        pair2.set_linepair (p2, p3, p4, p1);
    } else {
        CGAL_Orientation side1_32 = h_orientation [103] ('3', '2', '1');
```

```
            if (side1_32 != side3_24) {
                // p1, p2 need to be swapped
                pair1.set_linepair (p2, p1, p3, p4);
                pair2.set_linepair (p1, p3, p4, p2);
            } else {
                // p2, p3 need to be swapped
                pair1.set_linepair (p1, p3, p2, p4);
                pair2.set_linepair (p3, p2, p4, p1);
            }
        }
    }

}
```

This macro is defined in definitions 101, 105, 107, 109, 111, and 114.
This macro is invoked in definition 67.

ConicHPA2 private member functions [106] $+\equiv$ {

```
    static void set_two_linepairs (const PT &p1,
                                   const PT &p2,
                                   const PT &p3,
                                   const PT &p4,
                                   CGAL_ConicHPA2<PT,DA> &pair1,
                                   CGAL_ConicHPA2<PT,DA> &pair2)
    {
        FT x1, y1, x2, y2, x3, y3, x4, y4;
        const DA& da = pair1.da();
        da.get (p1, x1, y1);
        da.get (p2, x2, y2);
        da.get (p3, x3, y3);
        da.get (p4, x4, y4);

        CGAL_Orientation side1_24 = c_orientation [104] ('2', '4', '1'),
                         side3_24 = c_orientation [104] ('2', '4', '3');
        if (side1_24 != side3_24) {
            // (counter)clockwise order
            pair1.set_linepair (p1, p2, p3, p4);
            pair2.set_linepair (p2, p3, p4, p1);
        } else {
            CGAL_Orientation side1_32 = c_orientation [104] ('3', '2', '1');
            if (side1_32 != side3_24) {
                // p1, p2 need to be swapped
                pair1.set_linepair (p2, p1, p3, p4);
                pair2.set_linepair (p1, p3, p4, p2);
            } else {
                // p2, p3 need to be swapped
                pair1.set_linepair (p1, p3, p2, p4);
                pair2.set_linepair (p3, p2, p4, p1);
            }
```

```
        }
    }

}
```
This macro is defined in definitions 102, 106, 108, 110, 112, and 115.
This macro is invoked in definition 68.

Some ellipse from two pairs of lines. Assuming that we have constructed two line-pairs $\mathcal{C}_{\mathcal{R}_1}, \mathcal{C}_{\mathcal{R}_2}$ using the method `set_two_linepairs` with points p_1, p_2, p_3, p_4 in convex position, an ellipse through the points can be obtained as a linear combination $\mathcal{C}_{\mathcal{R}}$, $\mathcal{R} = \lambda\mathcal{R}_1 + \mu\mathcal{R}_2$, where

$$\begin{aligned} \lambda &= \det(\mathcal{R}_2) - 2(r_1 s_2 + r_2 s_1) + t_1 t_2, \\ \mu &= \det(\mathcal{R}_1) - 2(r_1 s_2 + r_2 s_1) + t_1 t_2, \end{aligned}$$

$r_i, \ldots, w_i$ the components of $\mathcal{R}_i, i = 1 \ldots 2$.

ConicCPA2 private member functions [107] $+\equiv$ {
```
    void set_ellipse (const CGAL_ConicCPA2<PT,DA> &pair1,
                      const CGAL_ConicCPA2<PT,DA> &pair2)
    {
        RT b = RT(2) * (pair1.r() * pair2.s() + pair1.s() * pair2.r()) -
               pair1.t() * pair2.t();
        set_linear_combination (pair2.det()-b, pair1,
                                pair1.det()-b, pair2);
    }

}
```
This macro is defined in definitions 101, 105, 107, 109, 111, and 114.
This macro is invoked in definition 67.

ConicHPA2 private member functions [108] $+\equiv$ {
```
    void set_ellipse (const CGAL_ConicHPA2<PT,DA> &pair1,
                      const CGAL_ConicHPA2<PT,DA> &pair2)
    {
        FT b = FT(2) * (pair1.r() * pair2.s() + pair1.s() * pair2.r()) -
               pair1.t() * pair2.t();
        set_linear_combination (pair2.det()-b, pair1,
                                pair1.det()-b, pair2);
    }

}
```
This macro is defined in definitions 102, 106, 108, 110, 112, and 115.
This macro is invoked in definition 68.

Conic from two conics and a point. If $\mathcal{C}_{\mathcal{R}_1}, \mathcal{C}_{\mathcal{R}_2}$ are two conics, p some point, it is easy to see that the conic $\mathcal{C}_{\mathcal{R}}$ with

$$\mathcal{R} = \mathcal{R}_2(p)\mathcal{R}_1 - \mathcal{R}_1(p)\mathcal{R}_2$$

is a conic containing the set $(\mathcal{C}_{\mathcal{R}_1} \cap \mathcal{C}_{\mathcal{R}_2}) \cup \{p\}$. Exactly this conic is constructed here.

ConicCPA2 private member functions [109] $+\equiv$ {

```
    void set (const CGAL_ConicCPA2<PT,DA> &c1,
              const CGAL_ConicCPA2<PT,DA> &c2,
              const PT &p)
    {
        set_linear_combination (c2.evaluate(p), c1, -c1.evaluate(p), c2);
    }

}
```

This macro is defined in definitions 101, 105, 107, 109, 111, and 114.
This macro is invoked in definition 67.

ConicHPA2 private member functions [110] $+\equiv$ {

```
    void set (const CGAL_ConicHPA2<PT,DA> &c1,
              const CGAL_ConicHPA2<PT,DA> &c2,
              const PT &p)
    {
        set_linear_combination (c2.evaluate(p), c1, -c1.evaluate(p), c2);
    }

}
```

This macro is defined in definitions 102, 106, 108, 110, 112, and 115.
This macro is invoked in definition 68.

Volume derivative of an ellipse. Let $r(\tau), s(\tau), t(\tau), u(\tau), v(\tau), w(\tau)$ denote the parameters of $\mathcal{E}\tau$). Omitting the parameter τ for the sake of readability, we have

$$\begin{aligned} r &= r_0 + \tau\partial r, \\ s &= s_0 + \tau\partial s, \\ t &= t_0 + \tau\partial t, \\ u &= u_0 + \tau\partial u, \\ v &= v_0 + \tau\partial v, \\ w &= w_0 + \tau\partial w, \end{aligned}$$

$r_0, \ldots, w_0$ the representation of $\mathcal{E}$, $\partial r, \ldots, \partial w$ the given values $\partial\mathcal{R}$.

Recall that

$$\mathrm{Vol}(\mathcal{E}(\tau)) = \pi/\sqrt{\det(M/(2w - c^T Mc))}.$$

This implies

$$\operatorname{sgn}\left(\frac{\partial}{\partial\tau}\,\mathrm{Vol}(\mathcal{E}(\tau))\big|_{\tau=0}\right) = -\operatorname{sgn}\left(\frac{\partial}{\partial\tau}\,\det(M/(2w - c^T Mc))\big|_{\tau=0}\right).$$

We have $\det(M/(2w - c^T Mc)) = d/z^2$, where

$$\begin{aligned} d &= \det(M) = 4rs - t^2, \\ z &= 2w - c^T Mc = 2\left(w - \frac{1}{d}(u^2 s - uvt + v^2 r)\right), \end{aligned}$$

see also (8). Hence

$$\frac{d}{z^2} = \frac{d^3}{4q^2}, \quad q = dw - u^2 s + uvt - v^2 r.$$

It follows that

$$\operatorname{sgn}\left(\frac{\partial}{\partial\tau}\operatorname{Vol}(\mathcal{E}(\tau))\right) = -\operatorname{sgn}\left(\frac{d^2(3d'q - 2dq')}{4q^3}\right) = -\operatorname{sgn}\left(3d'q - 2dq'\right)\operatorname{sgn}(q).$$

Consider the term $3d'q - 2dq' = 0$. We have

$$\begin{aligned} d &= a_2\tau^2 + a_1\tau + a_0, \\ q &= b_3\tau^3 + b_2\tau^2 + b_1\tau + b, \end{aligned}$$

where

$$\begin{aligned} a_2 &= 4\partial r\partial s - \partial t^2, \\ a_1 &= 4r_0\partial s + 4s_0\partial r - 2t_0\partial t, \\ a_0 &= 4r_0 s_0 - t_0^2, \\ b_3 &= (4\partial r\partial s - \partial t^2)\partial w - \partial u^2\partial s - \partial v^2\partial r + \partial t\partial u\partial v, \\ b_2 &= (4r_0\partial s + 4\partial r s_0 - 2t_0\partial t)\partial w + (4\partial r\partial s - \partial t^2)w_0 - 2u_0\partial u\partial s \\ &\quad -\partial u^2 s_0 - 2v_0\partial v\partial r - \partial v^2 r_0 + (u_0\partial v + \partial u v_0)\partial t + \partial u\partial v t_0, \\ b_1 &= (4r_0 s_0 - t_0^2)\partial w + (4r_0\partial s + 4\partial r s_0 - 2t_0\partial t)w_0 - u_0^2\partial s \\ &\quad -2u_0\partial u s_0 - v_0^2\partial r - 2v_0\partial v r_0 + u_0 v_0\partial t + (u_0\partial v + \partial u v_0)t_0, \\ b_0 &= (4r_0 s_0 - t_0^2)w_0 - u_0^2 s_0 - v_0^2 r_0 + u_0 v_0 t_0 \end{aligned}$$

Furthermore,

$$3d'q - 2dq' = c_3\tau^3 + c_2\tau^2 + c_1\tau + c_0,$$

with

$$\begin{aligned} c_3 &= -3a_1 b_3 + 2a_2 b_2, \\ c_2 &= -6a_0 b_3 - a_1 b_2 + 4a_2 b_1, \\ c_1 &= -4a_0 b_2 + a_1 b_1 + 6a_2 b_0, \\ c_0 &= -2a_0 b_1 + 3a_1 b_0. \end{aligned}$$

The desired sign of the volume derivative is obtained by evaluating the expression

$$-\operatorname{sgn}(3d'q - 2dq')\operatorname{sgn}(q)$$

at $\tau = 0$, where we get $3d'q - 2dq' = c_0$. Moreover, since $q = \det(M)(2w - c^T Mc)/2$ with $\det(M) > 0$, the sign of q is positive if and only if $2w - c^T Mc$ is positive, equivalently if the ellipse $\mathcal{E}(0)$ is in positive orientation (see also the description of the method `analyse`). This means, in case of positive orientation, the sign of $-\operatorname{sgn}(3d'q - 2dq')\operatorname{sgn}(q)$ equals the sign of $-c_0$, otherwise, we return the sign of c_0. To compute c_0, we only need the values a_1, a_0, b_1, b_0 from above.

ConicCPA2 private member functions [111] $+\equiv$ {

```
CGAL_Sign vol_derivative (RT dr, RT ds, RT dt,
                          RT du, RT dv, RT dw) const
{
    RT a1 = RT(4)*r()*ds+RT(4)*dr*s()-RT(2)*t()*dt,
       a0 = RT(4)*r()*s()-t()*t(),
       b1 = (RT(4)*r()*s()-t()*t())*dw+(RT(4)*r()*ds+RT(4)*dr*s()-
            RT(2)*t()*dt)*w()-u()*u()*ds -
            RT(2)*u()*du*s()-v()*v()*dr-RT(2)*v()*dv*r()+u()*v()*dt+
            (u()*dv+du*v())*t(),
       b0 = (RT(4)*r()*s()-t()*t())*w()
            -u()*u()*s()-v()*v()*r()+u()*v()*t(),
       c0 = -RT(2)*a0*b1 + RT(3)*a1*b0;

    return CGAL_Sign (-CGAL_sign (c0)*o);
}
```

}

This macro is defined in definitions 101, 105, 107, 109, 111, and 114.
This macro is invoked in definition 67.

ConicHPA2 private member functions [112] $+\equiv$ {

```
CGAL_Sign vol_derivative (FT dr, FT ds, FT dt,
                          FT du, FT dv, FT dw) const
{
    FT a1 = FT(4)*r()*ds+FT(4)*dr*s()-FT(2)*t()*dt,
       a0 = FT(4)*r()*s()-t()*t(),
       b1 = (FT(4)*r()*s()-t()*t())*dw+(FT(4)*r()*ds+FT(4)*dr*s()-
            FT(2)*t()*dt)*w()-u()*u()*ds -
            FT(2)*u()*du*s()-v()*v()*dr-FT(2)*v()*dv*r()+u()*v()*dt+
            (u()*dv+du*v())*t(),
       b0 = (FT(4)*r()*s()-t()*t())*w()
      -u()*u()*s()-v()*v()*r()+u()*v()*t(),
       c0 = -FT(2)*a0*b1 + FT(3)*a1*b0;

    return CGAL_Sign (-CGAL_sign (c0)*o);
}
```

}

This macro is defined in definitions 102, 106, 108, 110, 112, and 115.
This macro is invoked in definition 68.

To find the value τ^* such that $E(\tau^*)$ is the ellipse of smallest volume, we apply the Cardano formula to find the roots of the polynomial $p(\tau) = c_3\tau^3 + c_2\tau^2 + c_1\tau + c_0 = 0$. This is done by the function `CGAL_solve_cubic` which returns all (at most three) real roots. We then select the one which leads to largest (positive) volume.

Here is an outline of the Cardano formula, for the polynomial $p(\tau)$, assuming that $c_3 \neq 0$.

1. Divide by c_3 to normalize the equation, leading to
$$p(\tau) = \tau^3 + \gamma_2\tau^2 + \gamma_1\tau + \gamma_0, \quad \gamma_i := c_i/c_3,\ i = 0, \ldots, 2.$$

2. Eliminate the quadratic term by substituting $\tau := x - \gamma_2/3$. This leads to
$$p(x) = x^3 + ax + b, \quad a = \gamma_1 - \frac{\gamma_2^2}{3},\ b = \frac{2}{27}\gamma_2^3 - \frac{1}{3}\gamma_1\gamma_2 + \gamma_0.$$
If $a = 0$, p has only one real root, namely $x_1 = \sqrt[3]{-b}$, so let's assume $a \neq 0$.

3. Define
$$D := (a/3)^3 + (b/2)^2$$
and let u be any number such that
$$u^3 = -b/2 + \sqrt{D}.$$
Note that u^3 is a solution of the quadratic equation
$$x^2 + bx - \left(\frac{a}{3}\right)^3 = 0.$$
This means, if $a \neq 0$, then also $u \neq 0$. If the discriminant D is negative, u is a complex number, otherwise it must be chosen as the real number
$$u := \sqrt[3]{-b/2 + \sqrt{D}}.$$

4. Let u be of the form $u = u_R - u_I\mathbf{i}$ (we possibly have $u_I = 0$). p has always a real root, given by
$$x_1 = u_R\left(1 - \frac{a}{3\|u\|^2}\right).$$

5. If $D > 0$, the two other roots are complex. If $D \leq 0$, p has two more real roots, given by
$$\begin{aligned} x_2 &= -\frac{1}{2}\left(1 - \frac{a}{3\|u\|^2}\right)(u_R - u_I\sqrt{3}), \\ x_3 &= -\frac{1}{2}\left(1 - \frac{a}{3\|u\|^2}\right)(u_R + u_I\sqrt{3}). \end{aligned}$$
If $D = 0$, we have $u_I = 0$ and these two roots coincide.

It follows that if $D \geq 0$, the roots of p can be found by using only $\sqrt{\ }$ and $\sqrt[3]{\ }$ operations over the real numbers. If $D < 0$, it can be shown that this is not possible, and we need to approximate the values u_R, u_I in some other way. For this, we need to solve the equation
$$u^3 = C := -b/2 + i\sqrt{-D}$$
for u. Expressing C in polar coordinates (r, ϕ) we get
$$\begin{aligned} r &= \|C\| = \sqrt{b^2/4 - D} = \sqrt{-(a/3)^3}, \\ \cos\phi &= -\frac{b/2}{\|C\|}, \end{aligned}$$

where $0 \leq \phi < \pi$ because of $\sqrt{-D} > 0$. Therefore, a possible choice for u is $u = (r', \phi')$ with

$$\begin{aligned} r' &= \sqrt{-a/3}, \\ \phi' &= \operatorname{acos}\left(-\frac{b/2}{\|C\|}\right)/3. \end{aligned}$$

This gives

$$\begin{aligned} u_R &= r' \cos\phi', \\ u_I &= r' \sin\phi', \end{aligned}$$

Note that this implies $\|u\|^2 = -a/3$, showing that the roots of $p(x)$ assume the form

$$\begin{aligned} x_1 &= 2u_R, \\ x_2 &= -(u_R - u_I\sqrt{3}), \\ x_3 &= -(u_R + u_I\sqrt{3}). \end{aligned}$$

If u^3 is real, however, this is not true, and we really need to evaluate the factor

$$\alpha := \left(1 - \frac{a}{3\|u\|^2}\right) = \left(1 - \frac{a}{3u^2}\right)$$

in this case to obtain the roots of $p(x)$. In any case, we originally wanted to have the roots of $p(\tau)$. Using the substitution formula $\tau = x - \gamma_2/3$, these roots are given by $\tau_i = x_i - \gamma_2/3, i = 1, \ldots, 3$.

The function `CGAL_solve_cubic` returns the number of distinct real roots of $p(\tau) = c_3\tau^3 + c_2\tau^2 + c_1\tau + c_0 = 0$ and stores them consecutively in `r1`, `r2` and `r3`. Precondition is that $p(\tau)$ is not a constant function.

function CGAL_solve_cubic [113] $\mathbf{Z} \equiv \{$

```
int CGAL_solve_cubic (double c3, double c2, double c1, double c0,
                      double &r1, double &r2, double &r3)
{
    if (c3 == 0.0) {
        // quadratic equation
        if (c2 == 0) {
            // linear equation
            CGAL_kernel_precondition (c1 != 0);
            r1 = -c0/c1;
            return 1;
        }
        double D = c1*c1-4*c2*c0;
        if (D < 0.0)
            // only complex roots
            return 0;
        if (D == 0.0) {
```

```
        // one real root
        r1 = -c1/(2.0*c2);
        return 1;
    }
    // two real roots
    r1 = (-c1 + sqrt(D))/(2.0*c2);
    r2 = (-c1 - sqrt(D))/(2.0*c2);
    return 2;
}

// cubic equation
// define the gamma_i
double g2 = c2/c3,
       g1 = c1/c3,
       g0 = c0/c3;

// define a, b
double a  = g1 - g2*g2/3.0,
       b  = 2.0*g2*g2*g2/27.0 - g1*g2/3.0 + g0;

if (a == 0) {
    // one real root
    r1 = cbrt(-b) - g2/3.0;
    return 1;
}

// define D
double D  = a*a*a/27.0 + b*b/4.0;
if (D >= 0.0) {
    // real case
    double u = cbrt(-b/2.0 + sqrt(D)),
           alpha = 1.0 - a/(3.0*u*u);
    if (D == 0) {
        // two distinct real roots
        r1 =  u*alpha - g2/3.0;
        r2 =  -0.5*alpha*u - g2/3.0;
        return 2;
    }
    // one real root
    r1 = u*alpha - g2/3.0;
    return 1;
}
// complex case
double r_prime   = sqrt(-a/3),
       phi_prime = acos (-b/(2.0*r_prime*r_prime*r_prime))/3.0,
       u_R       = r_prime * cos (phi_prime),
       u_I       = r_prime * sin (phi_prime);
```

```
        // three distinct real roots
        r1 = 2.0*u_R - g2/3.0;
        r2 = -u_R + u_I*sqrt(3.0) - g2/3.0;
        r3 = -u_R - u_I*sqrt(3.0) - g2/3.0;
        return 3;
    }

}
```

This macro is invoked in definition 154.

Here comes the actual computation of the volume minimum. To this end, we compute the coefficients c_3, c_2, c_1, c_0 and find the roots of $p(\tau)$. Among the roots we then select the one which leads to the smallest volume, equivalently to the largest value of $\det(M/(2w - c^T Mc)) = d^3/4q^2$. The coefficients of d and q have been computed before in order to find the roots, so we can directly evaluate the determinant, using `double` approximations of the coefficients.

ConicCPA2 private member functions [114] $+\equiv$ {

```
    double vol_minimum (RT dr, RT ds, RT dt, RT du, RT dv, RT dw) const
    {
        RT a2 = RT(4)*dr*ds-dt*dt,
           a1 = RT(4)*r()*ds+RT(4)*dr*s()-RT(2)*t()*dt,
           a0 = RT(4)*r()*s()-t()*t(),
           b3 = (RT(4)*dr*ds-dt*dt)*dw-du*du*ds-dv*dv*dr+du*dv*dt,
           b2 = (RT(4)*r()*ds+RT(4)*dr*s()-RT(2)*t()*dt)*dw+
                (RT(4)*dr*ds-dt*dt)*w()-RT(2)*u()*du*ds-du*du*s()-
                RT(2)*v()*dv*dr-dv*dv*r()+(u()*dv+du*v())*dt+du*dv*t(),
           b1 = (RT(4)*r()*s()-t()*t())*dw+(RT(4)*r()*ds+RT(4)*dr*s()-
                RT(2)*t()*dt)*w()-u()*u()*ds -
                RT(2)*u()*du*s()-v()*v()*dr-RT(2)*v()*dv*r()+u()*v()*dt+
                (u()*dv+du*v())*t(),
           b0 = (RT(4)*r()*s()-t()*t())*w()
                -u()*u()*s()-v()*v()*r()+u()*v()*t(),
           c3 = -RT(3)*a1*b3 + RT(2)*a2*b2,
           c2 = -RT(6)*a0*b3 - a1*b2 + RT(4)*a2*b1,
           c1 = -RT(4)*a0*b2 + a1*b1 + RT(6)*a2*b0,
           c0 = -RT(2)*a0*b1 + RT(3)*a1*b0;

        if (c0 == 0) return 0;  // E(0) is the smallest ellipse

        double roots[3];
        int nr_roots = CGAL_solve_cubic
                             (CGAL_to_double(c3), CGAL_to_double(c2),
                              CGAL_to_double(c1), CGAL_to_double(c0),
                              roots[0], roots[1], roots[2]);
        CGAL_kernel_precondition (nr_roots > 0); // minimum exists
        return CGAL_best_value (roots, nr_roots,
                              CGAL_to_double(a2), CGAL_to_double(a1),
```

```
                                CGAL_to_double(a0), CGAL_to_double(b3),
                                CGAL_to_double(b2), CGAL_to_double(b1),
                                CGAL_to_double(b0));
    }

}
```

This macro is defined in definitions 101, 105, 107, 109, 111, and 114.
This macro is invoked in definition 67.

ConicHPA2 private member functions [115] $+\equiv$ {

```
    double vol_minimum (FT dr, FT ds, FT dt, FT du, FT dv, FT dw) const
    {
        FT a2 = FT(4)*dr*ds-dt*dt,
           a1 = FT(4)*r()*ds+FT(4)*dr*s()-FT(2)*t()*dt,
           a0 = FT(4)*r()*s()-t()*t(),
           b3 = (FT(4)*dr*ds-dt*dt)*dw-du*du*ds-dv*dv*dr+du*dv*dt,
           b2 = (FT(4)*r()*ds+FT(4)*dr*s()-FT(2)*t()*dt)*dw+
                (FT(4)*dr*ds-dt*dt)*w()-FT(2)*u()*du*ds-du*du*s()-
                FT(2)*v()*dv*dr-dv*dv*r()+(u()*dv+du*v())*dt+du*dv*t(),
           b1 = (FT(4)*r()*s()-t()*t())*dw+(FT(4)*r()*ds+FT(4)*dr*s()-
                FT(2)*t()*dt)*w()-u()*u()*ds -
                FT(2)*u()*du*s()-v()*v()*dr-FT(2)*v()*dv*r()+u()*v()*dt+
                (u()*dv+du*v())*t(),
           b0 = (FT(4)*r()*s()-t()*t())*w()
                -u()*u()*s()-v()*v()*r()+u()*v()*t(),
           c3 = -FT(3)*a1*b3 + FT(2)*a2*b2,
           c2 = -FT(6)*a0*b3 - a1*b2 + FT(4)*a2*b1,
           c1 = -FT(4)*a0*b2 + a1*b1 + FT(6)*a2*b0,
           c0 = -FT(2)*a0*b1 + FT(3)*a1*b0;

           // Is E(0) is the smallest ellipse?
           if ( CGAL_is_zero( c0)) return 0;

           double roots[3];
           int nr_roots = CGAL_solve_cubic
                               (CGAL_to_double(c3), CGAL_to_double(c2),
                                CGAL_to_double(c1), CGAL_to_double(c0),
                                roots[0], roots[1], roots[2]);
           CGAL_kernel_precondition (nr_roots > 0); // minimum exists
           return CGAL_best_value (roots, nr_roots,
                                CGAL_to_double(a2), CGAL_to_double(a1),
                                CGAL_to_double(a0), CGAL_to_double(b3),
                                CGAL_to_double(b2), CGAL_to_double(b1),
                                CGAL_to_double(b0));
    }

}
```

This macro is defined in definitions 102, 106, 108, 110, 112, and 115.
This macro is invoked in definition 68.

The function `CGAL_best_root` returns the value in its argument array which leads to the largest determinant d^3/q^2. A precondition was that an ellipse of smallest volume exists, so the largest determinant must be positive.

function CGAL_best_value [116] $\equiv$ {

```
double CGAL_best_value (double *values, int nr_values,
                        double a2, double a1, double a0,
                        double b3, double b2, double b1, double b0)
{
    bool det_positive = false;
    double d, q, max_det = 0.0, det, best;
    for (int i=0; i<nr_values; ++i) {
        double x = values[i];
        d = (a2*x+a1)*x+a0;
        q = ((b3*x+b2)*x+b1)*x+b0;
        det = d*d*d/(q*q);
        if (det > 0.0)
            if (!det_positive || (det > max_det)) {
                max_det = det;
                best = x;
                det_positive = true;
            }
    }
    CGAL_kernel_precondition (det_positive);
    return best;
}
```

}
This macro is invoked in definition 154.

5.4.11 Public Set Methods

Here is the set method at coordinate level.

ConicCPA2 public member functions [117] $+\equiv$ {

```
void set (RT r_, RT s_, RT t_, RT u_, RT v_, RT w_)
{
    _r = r_; _s = s_; _t = t_; _u = u_; _v = v_; _w = w_;
    analyse();
 }
```

}
This macro is defined in definitions 83, 85, 87, 89, 91, 93, 95, 97, 99, 117, 119, 121, 123, 125, and 127.
This macro is invoked in definition 67.

ConicHPA2 public member functions [118] $+\equiv$ {

```
void set (FT r_, FT s_, FT t_, FT u_, FT v_, FT w_)
{
    _r = r_; _s = s_; _t = t_; _u = u_; _v = v_; _w = w_;
```

```
        analyse();
    }

}
```

This macro is defined in definitions 84, 86, 88, 90, 92, 94, 96, 98, 100, 118, 120, 122, 124, 126, and 128.
This macro is invoked in definition 68.

Opposite conic. The method `set_opposite` just flips the representation $\mathcal{R}$ and the orientation, all other derived data are taken over.

ConicCPA2 public member functions [119] $+\equiv$ {

```
    void set_opposite ()
    {
        _r = -r(); _s = -s(); _t = -t(); _u = -u(); _v = -v(); _w = -w();
        o = CGAL_opposite(orientation());
    }

}
```

This macro is defined in definitions 83, 85, 87, 89, 91, 93, 95, 97, 99, 117, 119, 121, 123, 125, and 127.
This macro is invoked in definition 67.

ConicHPA2 public member functions [120] $+\equiv$ {

```
    void set_opposite ()
    {
        _r = -r(); _s = -s(); _t = -t(); _u = -u(); _v = -v(); _w = -w();
        o = CGAL_opposite(orientation());
    }

}
```

This macro is defined in definitions 84, 86, 88, 90, 92, 94, 96, 98, 100, 118, 120, 122, 124, 126, and 128.
This macro is invoked in definition 68.

Pair of lines through four points. Given $p_1 = (x_1, y_1, h_1)$, $p_2 = (x_2, y_2, h_2)$, $p_3 = (x_3, y_3, h_3)$, $p_4 = (x_4, y_4, h_4)$, we develop a formula for representing the pair of lines $\overline{p_1p_2}$, $\overline{p_3p_4}$ (forming a degenerate hyperbola) in the form of (1). To this end, let $p = (x, y, h)$ be any point and define

$$[p_i, p_j, p] := \det \begin{pmatrix} x_i & x_j & x \\ y_1 & y_2 & y \\ h_1 & h_2 & h \end{pmatrix}. \tag{9}$$

It is well known that $[p_i, p_j, p]$ records the orientation of the point triple: let ℓ be the oriented line through p_i and p_j. Then the following holds.

Lemma 5.1

$$p \text{ lies} \left\{ \begin{array}{c} \textit{to the left of} \\ \textit{on} \\ \textit{to the right of} \end{array} \right\} \ell \Leftrightarrow [p_i, p_j, p] \left\{ \begin{array}{c} > 0 \\ = 0 \\ < 0 \end{array} \right\}.$$

In particular, p lies on $\overline{p_1p_2} \cup \overline{p_3p_4}$ iff $[p_1, p_2, p][p_3, p_4, p] = 0$, and this expression turns out to be of the form (1), where `Maple` gives us the concrete values of r, s, t, u, v, w. Note that we must have $p_1 \neq p_2$ and $p_3 \neq p_4$ to obtain reasonable results (and this was a precondition).

$$\begin{aligned}
r &= (y_1h_2 - h_1y_2)(y_3h_4 - h_3y_4),\\
s &= (h_1x_2 - x_1h_2)(h_3x_4 - x_3h_4),\\
t &= (h_1x_2 - x_1h_2)(y_3h_4 - h_3y_4) + (y_1h_2 - h_1y_2)(h_3x_4 - x_3h_4),\\
u &= (-y_1x_2 + x_1y_2)(y_3h_4 - h_3y_4) + (y_1h_2 - h_1y_2)(-y_3x_4 + x_3y_4),\\
v &= (-y_1x_2 + x_1y_2)(h_3x_4 - x_3h_4) + (h_1x_2 - x_1h_2)(-y_3x_4 + x_3y_4),\\
w &= (-y_1x_2 + x_1y_2)(-y_3x_4 + x_3y_4).
\end{aligned}$$

ConicCPA2 public member functions [121] $+\equiv$ {

```
void
set_linepair (const PT &p1, const PT &p2, const PT &p3, const PT &p4,
              const DA &da = DA())
{
    RT x1, y1, h1, x2, y2, h2, x3, y3, h3, x4, y4, h4;
    da.get (p1, x1, y1, h1);
    da.get (p2, x2, y2, h2);
    da.get (p3, x3, y3, h3);
    da.get (p4, x4, y4, h4);

    // precondition: p1 != p2, p3 != p4
    CGAL_kernel_precondition
        ( ((x1*h2 != x2*h1) || (y1*h2 != y2*h1)) &&
          ((x3*h4 != x4*h3) || (y3*h4 != y4*h3)) );

    RT h1x2_x1h2 = h1*x2-x1*h2;
    RT h3x4_x3h4 = h3*x4-x3*h4;
    RT y1h2_h1y2 = y1*h2-h1*y2;
    RT y3h4_h3y4 = y3*h4-h3*y4;
    RT x1y2_y1x2 = x1*y2-y1*x2;
    RT x3y4_y3x4 = x3*y4-y3*x4;

    _r = y1h2_h1y2 * y3h4_h3y4;
    _s = h1x2_x1h2 * h3x4_x3h4;
    _t = h1x2_x1h2 * y3h4_h3y4 + y1h2_h1y2 * h3x4_x3h4;
    _u = x1y2_y1x2 * y3h4_h3y4 + y1h2_h1y2 * x3y4_y3x4;
    _v = x1y2_y1x2 * h3x4_x3h4 + h1x2_x1h2 * x3y4_y3x4;
    _w = x1y2_y1x2 * x3y4_y3x4;

    analyse();
}
```

}

This macro is defined in definitions 83, 85, 87, 89, 91, 93, 95, 97, 99, 117, 119, 121, 123, 125, and 127.
This macro is invoked in definition 67.

For the Cartesian representation we proceed completely similar, replacing values $h_1, \ldots, h_4$ by 1.

ConicHPA2 public member functions [122] $+\equiv$ {

```
void
set_linepair (const PT &p1, const PT &p2, const PT &p3, const PT &p4,
              const DA &da = DA())
{
    FT x1, y1, x2, y2, x3, y3, x4, y4;
    da.get (p1, x1, y1);
    da.get (p2, x2, y2);
    da.get (p3, x3, y3);
    da.get (p4, x4, y4);

    // precondition: p1 != p2, p3 != p4
    CGAL_kernel_precondition
        ( ((x1 != x2) || (y1 != y2)) &&
          ((x3 != x4) || (y3 != y4)) );

    FT x2_x1 = x2-x1;
    FT x4_x3 = x4-x3;
    FT y1_y2 = y1-y2;
    FT y3_y4 = y3-y4;
    FT x1y2_y1x2 = x1*y2-y1*x2;
    FT x3y4_y3x4 = x3*y4-y3*x4;

    _r = y1_y2 * y3_y4;
    _s = x2_x1 * x4_x3;
    _t = x2_x1 * y3_y4 + y1_y2 * x4_x3;
    _u = x1y2_y1x2 * y3_y4 + y1_y2 * x3y4_y3x4;
    _v = x1y2_y1x2 * x4_x3 + x2_x1 * x3y4_y3x4;
    _w = x1y2_y1x2 * x3y4_y3x4;

    analyse();
}
```

}

This macro is defined in definitions 84, 86, 88, 90, 92, 94, 96, 98, 100, 118, 120, 122, 124, 126, and 128.
This macro is invoked in definition 68.

Smallest ellipse through three points. Given $p_1 = (x_1, y_1, h_1)$, $p_2 = (x_2, y_2, h_2)$, and $p_3 = (x_3, y_3, h_3)$, we give a formula for representing the ellipse of smallest volume containing p_1, p_2, and p_3 in the form of (1). For this, we use the following well-known formula for this ellipse in case of Cartesian points.

Lemma 5.2 *Let q_1, q_2, q_3 be non-collinear Cartesian points. Then the smallest ellipse containing q_1, q_2, q_3 can be written as the set of points $q = (x, y)$ satisfying*

$$(q - c)^T M (q - c) = 1, \tag{10}$$

where

$$c = \frac{1}{3}\sum_{i=1}^{3} q_i, \quad M^{-1} = \frac{2}{3}\sum_{i=1}^{3}(q_i - c)(q_i - c)^T.$$

To apply this Lemma to points (p_1, p_2, p_3), we define $q_i = (x_i/h_i, y_i/h_i)$, $i = 1, \ldots, 3$ and observe that (10) can be written as

$$q^T M q - 2q^T M c + c^T M c - 1 = 0,$$

from which a representation in form of (4) is obtained via

$$\begin{pmatrix} 2r & t \\ t & 2s \end{pmatrix} := M, \quad \begin{pmatrix} u \\ v \end{pmatrix} := -Mc, \quad 2w := c^T M c - 1.$$

Using `Maple` [2], we get the following values for $\mathcal{R} = (r, s, t, u, v, w)$.

$$\begin{aligned}
r &= 3(y_1^2h_2^2h_3^2 - y_1h_2h_3^2y_2h_1 - y_1h_2^2h_3y_3h_1 + y_2^2h_1^2h_3^2 - y_2h_1^2h_3y_3h_2 + y_3^2h_1^2h_2^2)/d,\\
s &= 3(x_1^2h_2^2h_3^2 - x_1h_2h_3^2x_2h_1 - x_1h_2^2h_3x_3h_1 + x_2^2h_1^2h_3^2 - x_2h_1^2h_3x_3h_2 + x_3^2h_1^2h_2^2)/d,\\
t &= 3(-2x_1h_2^2h_3^2y_1 + x_1h_2h_3^2y_2h_1 + x_1h_2^2h_3y_3h_1 + x_2h_1h_3^2y_1h_2\\
&\quad - 2x_2h_1^2h_3^2y_2 + x_2h_1^2h_3y_3h_2 + x_3h_1h_2^2y_1h_3 + x_3h_1^2h_2y_2h_3 - 2x_3h_1^2h_2^2y_3)/d,\\
u &= -3(y_2^2h_1^2h_3x_3 - y_2h_1^2h_3y_3x_2 - y_2h_1^2y_3h_2x_3 + y_3^2h_1h_2^2x_1\\
&\quad + y_3^2h_1^2h_2x_2 + y_1^2h_2h_3^2x_2 + y_1^2h_2^2h_3x_3 - y_1h_2h_3^2y_2x_1\\
&\quad - y_1h_3^2y_2h_1x_2 - y_1h_2^2h_3y_3x_1 - y_1h_2^2y_3h_1x_3 + y_2^2h_1h_3^2x_1)/d,\\
v &= -3(-x_1h_2h_3^2y_1x_2 - x_1h_2^2h_3y_1x_3 + x_1^2h_2h_3^2y_2 - x_1h_3^2y_2h_1x_2\\
&\quad + x_1^2h_2^2h_3y_3 - x_1h_2^2y_3h_1x_3 + x_2^2h_1h_3^2y_1 - x_2h_1^2h_3y_2x_3\\
&\quad + x_2^2h_1^2h_3y_3 - x_2h_1^2y_3h_2x_3 + x_3^2h_1h_2^2y_1 + x_3^2h_1^2h_2y_2)/d,\\
w &= 3(x_2h_3x_3h_2y_1^2 - x_1h_2h_3x_3y_1y_2 - x_2h_1h_3x_3y_1y_2 + x_3^2h_1h_2y_1y_2\\
&\quad - x_1h_2h_3x_2y_1y_3 + x_2^2h_1h_3y_1y_3 - x_2h_1x_3h_2y_1y_3 + x_1h_3x_3h_1y_2^2\\
&\quad + x_1^2h_2h_3y_2y_3 - x_1h_3x_2h_1y_2y_3 - x_1h_2x_3h_1y_2y_3 + x_1h_2x_2h_1y_3^2)/d,
\end{aligned}$$

where

$$\begin{aligned}
\delta &= \left(\det\begin{pmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ h_1 & h_2 & h_3 \end{pmatrix}\right)^2\\
&= (-h_1x_3y_2 + x_1y_2h_3 + h_1x_2y_3 - x_1y_3h_2 + h_2x_3y_1 - x_2y_1h_3)^2.
\end{aligned}$$

After precomputing the values

$$x_i^2, y_i^2, x_ih_i, y_ih_i, h_i^2,$$

for $i = 1, \ldots, 3$, the components of the vector $\delta\mathcal{R}/3$ are easy to obtain. Note that this vector is a legal representation of the ellipse if $\delta \neq 0$. This is the case if and only if p_1, p_2, p_3 are non-collinear, see Lemma 5.1. Moreover, one can show that the formulas above determine an ellipse of negative orientation, regardless of the point triple orientation. This means, if the orientation was positive, we still need to flip the representation.

ConicCPA2 public member functions [123] $+\equiv$ {

```
void set_ellipse (const PT &p1, const PT &p2, const PT &p3)
{
    RT x1, y1, h1, x2, y2, h2, x3, y3, h3;
    dao.get (p1, x1, y1, h1);
    dao.get (p2, x2, y2, h2);
    dao.get (p3, x3, y3, h3);

    // precondition: p1, p2, p3 not collinear
    RT det = -h1*x3*y2+h3*x1*y2+h1*x2*y3-h2*x1*y3+h2*x3*y1-h3*x2*y1;
    CGAL_kernel_precondition (!CGAL_is_zero (det));

    RT x1x1 = x1*x1, y1y1 = y1*y1,
       x2x2 = x2*x2, y2y2 = y2*y2,
       x3x3 = x3*x3, y3y3 = y3*y3,  // x_i^2, y_i^2
       x1h1 = x1*h1, y1h1 = y1*h1,
       x2h2 = x2*h2, y2h2 = y2*h2,
       x3h3 = x3*h3, y3h3 = y3*h3,  // x_i h_i, y_i h_i
       h1h1 = h1*h1,
       h2h2 = h2*h2,
       h3h3 = h3*h3,                // h_i^2
       two = RT(2);                 // 2

    _r = y1y1*h2h2*h3h3 - y1h1*y2h2*h3h3 - y1h1*h2h2*y3h3 +
         h1h1*y2y2*h3h3 - h1h1*y2h2*y3h3 + h1h1*h2h2*y3y3;

    _s = x1x1*h2h2*h3h3 - x1h1*x2h2*h3h3 - x1h1*h2h2*x3h3 +
         h1h1*x2x2*h3h3 - h1h1*x2h2*x3h3 + h1h1*h2h2*x3x3;

    _t = -two*x1*y1*h2h2*h3h3 + x1h1*y2h2*h3h3 + x1h1*h2h2*y3h3 +
             y1h1*x2h2*h3h3 -two*h1h1*x2*y2*h3h3 + h1h1*x2h2*y3h3 +
             y1h1*h2h2*x3h3 + h1h1*y2h2*x3h3 -two*h1h1*h2h2*x3*y3;

    _u = -(h1h1*y2y2*x3h3 - h1h1*x2*y2*y3h3 - h1h1*y2h2*x3*y3 +
               x1h1*h2h2*y3y3 + h1h1*x2h2*y3y3 +y1y1*x2h2*h3h3 +
               y1y1*h2h2*x3h3 - x1*y1*y2h2*h3h3 - y1h1*x2*y2*h3h3 -
               x1*y1*h2h2*y3h3 - y1h1*h2h2*x3*y3 + x1h1*y2y2*h3h3);

    _v = -(h1h1*x2x2*y3h3 - h1h1*x2*y2*x3h3 - h1h1*x2h2*x3*y3 +
               y1h1*h2h2*x3x3 + h1h1*y2h2*x3x3 +x1x1*y2h2*h3h3 +
               x1x1*h2h2*y3h3 - x1*y1*x2h2*h3h3 - x1h1*x2*y2*h3h3 -
               x1*y1*h2h2*x3h3 - x1h1*h2h2*x3*y3 + y1h1*x2x2*h3h3);

    _w = y1y1*x2h2*x3h3 - x1*y1*y2h2*x3h3 - y1h1*x2*y2*x3h3 +
         y1h1*y2h2*x3x3 - x1*y1*x2h2*y3h3 + y1h1*x2x2*y3h3 -
         y1h1*x2h2*x3*y3 + x1h1*y2y2*x3h3 + x1x1*y2h2*y3h3 -
         x1h1*x2*y2*y3h3 - x1h1*y2h2*x3*y3 + x1h1*x2h2*y3y3;
```

```
        type = CGAL_ELLIPSE;
        degenerate = trivial = empty = false;
        o = CGAL_NEGATIVE;
        if (CGAL_is_positive (det)) set_opposite ();
    }

}
```

This macro is defined in definitions 83, 85, 87, 89, 91, 93, 95, 97, 99, 117, 119, 121, 123, 125, and 127.
This macro is invoked in definition 67.

As before, the Cartesian version is obtained by setting h_1, h_2, h_3 to 1.

ConicHPA2 public member functions [124] $+\equiv$ {

```
    void set_ellipse (const PT &p1, const PT &p2, const PT &p3)
    {
        FT x1, y1, x2, y2, x3, y3;
        dao.get (p1, x1, y1);
        dao.get (p2, x2, y2);
        dao.get (p3, x3, y3);

        // precondition: p1, p2, p3 not collinear
        FT det = -x3*y2+x1*y2+x2*y3-x1*y3+x3*y1-x2*y1;
        CGAL_kernel_precondition (!CGAL_is_zero (det));

        FT x1x1 = x1*x1, y1y1 = y1*y1,
           x2x2 = x2*x2, y2y2 = y2*y2,
           x3x3 = x3*x3, y3y3 = y3*y3,  // x_i^2, y_i^2
           two = FT(2);

        _r = y1y1 - y1*y2 - y1*y3 +
             y2y2 - y2*y3 + y3y3;

        _s = x1x1 - x1*x2 - x1*x3 +
             x2x2 - x2*x3 + x3x3;

        _t = -two*x1*y1 + x1*y2 + x1*y3 +
                 y1*x2 -two*x2*y2 + x2*y3 +
                 y1*x3 + y2*x3 -two*x3*y3;

        _u = -(y2y2*x3 - x2*y2*y3 - y2*x3*y3 +
                   x1*y3y3 + x2*y3y3 + y1y1*x2 +
                   y1y1*x3 - x1*y1*y2 - y1*x2*y2 -
                   x1*y1*y3 - y1*x3*y3 + x1*y2y2);

        _v = -(x2x2*y3 - x2*y2*x3 - x2*x3*y3 +
                   y1*x3x3 + y2*x3x3 + x1x1*y2 +
                   x1x1*y3 - x1*y1*x2 - x1*x2*y2 -
                   x1*y1*x3 - x1*x3*y3 + y1*x2x2);
```

```
        _w = y1y1*x2*x3 - x1*y1*y2*x3 - y1*x2*y2*x3 +
             y1*y2*x3x3 - x1*y1*x2*y3 + y1*x2x2*y3 -
             y1*x2*x3*y3 + x1*y2y2*x3 + x1x1*y2*y3 -
             x1*x2*y2*y3 - x1*y2*x3*y3 + x1*x2*y3y3;

        type = CGAL_ELLIPSE;
        degenerate = trivial = empty = false;
        o = CGAL_NEGATIVE;
        if (CGAL_is_positive (det)) set_opposite();
    }

}
```

This macro is defined in definitions 84, 86, 88, 90, 92, 94, 96, 98, 100, 118, 120, 122, 124, 126, and 128.
This macro is invoked in definition 68.

Some ellipse through four points in convex position. This method builds on the private method to obtain an ellipse from two pairs of lines through the four points. For constructing this pair, we also have a method available.

ConicCPA2 public member functions [125] $+\equiv$ {

```
    void set_ellipse (const PT &p1, const PT &p2,
                      const PT &p3, const PT &p4,
                      CGAL_Orientation _o = CGAL_POSITIVE)
    {
        CGAL_ConicCPA2<PT,DA> pair1, pair2;
        set_two_linepairs (p1, p2, p3, p4, pair1, pair2);
        set_ellipse (pair1, pair2);
        analyse();
        if (o != _o) set_opposite();
    }

}
```

This macro is defined in definitions 83, 85, 87, 89, 91, 93, 95, 97, 99, 117, 119, 121, 123, 125, and 127.
This macro is invoked in definition 67.

ConicHPA2 public member functions [126] $+\equiv$ {

```
    void set_ellipse (const PT &p1, const PT &p2,
                      const PT &p3, const PT &p4,
                      CGAL_Orientation _o = CGAL_POSITIVE)
    {
        CGAL_ConicHPA2<PT,DA> pair1, pair2;
        set_two_linepairs (p1, p2, p3, p4, pair1, pair2);
        set_ellipse (pair1, pair2);
        analyse();
        if (o != _o) set_opposite();
    }

}
```

This macro is defined in definitions 84, 86, 88, 90, 92, 94, 96, 98, 100, 118, 120, 122, 124, 126, and 128.
This macro is invoked in definition 68.

Unique conic through five points. Using the previously defined methods, we implement the method to compute the unique nontrivial conic through five given points p_1, p_2, p_3, p_4, p_5. For this, we first compute the two conics $\mathcal{C}_1 = \overline{p_1p_2} \cup \overline{p_3p_4}$ and $\mathcal{C}_2 = \overline{p_1p_4} \cup \overline{p_2p_3}$, using the `set_linepair` method. This gives two conics having the points p_1, p_2, p_3, p_4 in common. It follows that any linear combination of them goes through p_1, p_2, p_3, p_4 as well. A particular linear combination is given by

$$\mathcal{C} := \mathcal{C}_2(p_5)\mathcal{C}_1 - \mathcal{C}_1(p_5)\mathcal{C}_2, \tag{11}$$

and it has the property that $\mathcal{C}(p_5) = 0$, i.e. $\mathcal{C}$ goes through $p_1, \ldots, p_5$. In case all points are distinct, this is the unique nontrivial conic through the points.

ConicCPA2 public member functions [127] $+\equiv$ {

```
void set (const PT &p1, const PT &p2, const PT &p3, const PT &p4,
          const PT &p5, CGAL_Orientation _o = CGAL_POSITIVE)
{
    CGAL_ConicCPA2<PT,DA> c1; c1.set_linepair (p1, p2, p3, p4);
    CGAL_ConicCPA2<PT,DA> c2; c2.set_linepair (p1, p4, p2, p3);
    set_linear_combination (c2.evaluate (p5), c1,
                            -c1.evaluate (p5), c2);
    analyse();
    // precondition: all points distinct <=> conic nontrivial
    CGAL_kernel_precondition (!is_trivial());
    if (o != _o) set_opposite();
}
```

}

This macro is defined in definitions 83, 85, 87, 89, 91, 93, 95, 97, 99, 117, 119, 121, 123, 125, and 127.
This macro is invoked in definition 67.

ConicHPA2 public member functions [128] $+\equiv$ {

```
void set (const PT &p1, const PT &p2, const PT &p3, const PT &p4,
          const PT &p5, CGAL_Orientation _o = CGAL_POSITIVE)
{
    CGAL_ConicHPA2<PT,DA> c1; c1.set_linepair (p1, p2, p3, p4);
    CGAL_ConicHPA2<PT,DA> c2; c2.set_linepair (p1, p4, p2, p3);
    set_linear_combination (c2.evaluate (p5), c1,
                            -c1.evaluate (p5), c2);
    analyse();
    // precondition: all points distinct <=> conic nontrivial
    CGAL_kernel_precondition (!is_trivial());
    if (o != _o) set_opposite();
}
```

}

This macro is defined in definitions 84, 86, 88, 90, 92, 94, 96, 98, 100, 118, 120, 122, 124, 126, and 128.
This macro is invoked in definition 68.

5.4.12 I/O

ConicCPA2 I/O routines [129] ≡ {

```
template< class _PT, class _DA>
ostream& operator << ( ostream& os, const CGAL_ConicCPA2<_PT,_DA>& c)
{
    return( os << c.r() << ' ' << c.s() << ' ' << c.t() << ' '
               << c.u() << ' ' << c.v() << ' ' << c.w());
}

template< class _PT, class _DA>
istream& operator >> ( istream& is, CGAL_ConicCPA2<_PT,_DA>& c)
{
    typedef           CGAL_ConicCPA2<_PT,_DA>  Conic;
    typedef  typename _DA::RT                  RT;

    RT  r, s, t, u, v, w;
    is >> r >> s >> t >> u >> v >> w;
    c.set( r, s, t, u, v, w);

    return( is);
}
}
```

This macro is invoked in definition 157.

ConicHPA2 I/O routines [130] ≡ {

```
template< class _PT, class _DA>
ostream& operator << ( ostream& os, const CGAL_ConicHPA2<_PT,_DA>& c)
{
    return( os << c.r() << ' ' << c.s() << ' ' << c.t() << ' '
               << c.u() << ' ' << c.v() << ' ' << c.w());
}

template< class _PT, class _DA>
istream& operator >> ( istream& is, CGAL_ConicHPA2<_PT,_DA>& c)
{
    typedef           CGAL_ConicHPA2<_PT,_DA>  Conic;
    typedef  typename _DA::FT                  FT;

    FT  r, s, t, u, v, w;
    is >> r >> s >> t >> u >> v >> w;
    c.set( r, s, t, u, v, w);

    return( is);
}
}
```

This macro is invoked in definition 156.

5.5 Class Template `CGAL_Min_ellipse_2_traits_2<R>`

First, we declare the class templates `CGAL_Min_ellipse_2_traits_2` and `CGAL_Min_ellipse_2`.

Min_ellipse_2_traits_2 declarations [131] ≡ {
```
template < class _Traits >
class CGAL_Min_ellipse_2;

template < class _R >
class CGAL_Min_ellipse_2_traits_2;
```
}
This macro is invoked in definition 158.

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

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

Min_ellipse_2_traits_2 interface and implementation [132] ≡ {
```
template < class _R >
class CGAL_Min_ellipse_2_traits_2 {
  public:
    // types
    typedef  _R                                R;
    typedef  CGAL_Point_2<R>                   Point;
    typedef  CGAL_Optimisation_ellipse_2<R>    Ellipse;

private:
    // data members
    Ellipse  ellipse;                                  // current ellipse

    // friends
    friend class CGAL_Min_ellipse_2< CGAL_Min_ellipse_2_traits_2<R> >;

  public:
    // creation (use default implementations)
    // CGAL_Min_ellipse_2_traits_2( );
    // CGAL_Min_ellipse_2_traits_2(
           CGAL_Min_ellipse_2_traits_2<R> const&);
};
```
}
This macro is invoked in definition 158.

5.6 Class Template `CGAL_Min_ellipse_2_adapterC2<PT,DA>`

First, we declare class templates `CGAL_Min_ellipse_2`, `CGAL_Min_ellipse_2_adapterC2` and `CGAL__Min_ellipse_2_adapterC2__Ellipse`.

Min_ellipse_2_adapterC2 declarations [133] ≡ {
```
template < class _Traits >
class CGAL_Min_ellipse_2;

template < class _PT, class _DA >
class CGAL_Min_ellipse_2_adapterC2;

template < class _PT, class _DA >
class CGAL__Min_ellipse_2_adapterC2__Ellipse;
```
}
This macro is invoked in definition 159.

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

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

Min_ellipse_2_adapterC2 interface and implementation [134] ≡ {
```
template < class _PT, class _DA >
class CGAL_Min_ellipse_2_adapterC2 {
  public:
    // types
    typedef  _PT  PT;
    typedef  _DA  DA;

    // nested types
    typedef  PT                                                  Point;
    typedef  CGAL__Min_ellipse_2_adapterC2__Ellipse<PT,DA>  Ellipse;

  private:
    DA      dao;                              // data accessor object
    Ellipse ellipse;                          // current ellipse
    friend
      class CGAL_Min_ellipse_2< CGAL_Min_ellipse_2_adapterC2<PT,DA> >;

  public:
    // creation
    Min_ellipse_2_adapterC2 constructors [135]

    // operations
    Min_ellipse_2_adapterC2 operations [136]
};
```
}
This macro is invoked in definition 159.

5.6.1 Constructors

Min_ellipse_2_adapterC2 constructors [135] ≡ {
```
CGAL_Min_ellipse_2_adapterC2( const DA& da = DA())
    : dao( da), ellipse( da)
{ }
```
}
This macro is invoked in definition 134.

5.6.2 Operations

Min_ellipse_2_adapterC2 operations [136] ≡ {
```
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 134.

5.6.3 Nested Type `Ellipse`

Min_ellipse_2_adapterC2 nested type 'Ellipse' [137] ≡ {
```
template < class _PT, class _DA >
class CGAL__Min_ellipse_2_adapterC2__Ellipse {
  public:
    // typedefs
    typedef  _PT  PT;
    typedef  _DA  DA;
```

```
  typedef           CGAL_ConicCPA2< PT, DA>  CT;
  typedef  typename _DA::FT                  FT;

private:
  // data members
  int  n_boundary_points;           // number of boundary points
  PT   boundary_point1, boundary_point2;  // two boundary points
  CT   conic1, conic2;                    // two conics
  FT   dr, ds, dt, du, dv, dw;            // the gradient vector

public:
  // types
  typedef  PT  Point;

  // creation
  CGAL__Min_ellipse_2_adapterC2__Ellipse( const DA& da)
      : conic1( da), conic2( da)
  { }

  void
  set( )
  {
      n_boundary_points = 0;
  }

  void
  set( const Point& p)
  {
      n_boundary_points = 1;
      boundary_point1   = p;
  }

  void
  set( const Point& p, const Point& q)
  {
      n_boundary_points = 2;
      boundary_point1   = p;
      boundary_point2   = q;
  }

  void
  set( const Point& p1, const Point& p2, const Point& p3)
  {
      n_boundary_points = 3;
      conic1.set_ellipse( p1, p2, p3);
  }
```

```
void
set( const Point& p1, const Point& p2,
     const Point& p3, const Point& p4)
{
    n_boundary_points = 4;
    CT::set_two_linepairs( p1, p2, p3, p4, conic1, conic2);
    dr = FT( 0);
    ds = conic1.r() * conic2.s() - conic2.r() * conic1.s(),
    dt = conic1.r() * conic2.t() - conic2.r() * conic1.t(),
    du = conic1.r() * conic2.u() - conic2.r() * conic1.u(),
    dv = conic1.r() * conic2.v() - conic2.r() * conic1.v(),
    dw = conic1.r() * conic2.w() - conic2.r() * conic1.w();
}

void
set( const Point&, const Point&,
     const Point&, const Point&, const Point& p5)
{
    n_boundary_points = 5;
    conic1.set( conic1, conic2, p5);
    conic1.analyse();
}

// predicates
CGAL_Bounded_side
bounded_side( const Point& p) const
{
    switch ( n_boundary_points) {
      case 0:
        return( CGAL_ON_UNBOUNDED_SIDE);
      case 1:
        return( ( p == boundary_point1) ?
                      CGAL_ON_BOUNDARY : CGAL_ON_UNBOUNDED_SIDE);
      case 2:
        return(    ( p == boundary_point1)
                || ( p == boundary_point2)
                || ( CGAL_are_ordered_along_lineC2(
                       boundary_point1, p,
                       boundary_point2, conic1.da())) ?
                       CGAL_ON_BOUNDARY : CGAL_ON_UNBOUNDED_SIDE);
      case 3:
      case 5:
        return( conic1.convex_side( p));
      case 4: {
        CT c( conic1.da());
        c.set( conic1, conic2, p);
        c.analyse();
```

```
        if ( ! c.is_ellipse()) {
            c.set_ellipse( conic1, conic2);
            c.analyse();
            return( c.convex_side( p)); }
        else {
            int tau_star = -c.vol_derivative( dr, ds, dt,
                                              du, dv, dw);
            return( CGAL_static_cast( CGAL_Bounded_side,
                                      CGAL_sign( tau_star))); } }
      default:
        CGAL_optimisation_assertion(
            ( n_boundary_points >= 0) &&
            ( n_boundary_points <= 5) ); }

    // keeps g++ happy
    return( CGAL_Bounded_side( 0));
}

bool
has_on_bounded_side( const Point& p) const
{
    return( bounded_side( p) == CGAL_ON_BOUNDED_SIDE);
}

bool
has_on_boundary( const Point& p) const
{
    return( bounded_side( p) == CGAL_ON_BOUNDARY);
}

bool
has_on_unbounded_side( const Point& p) const
{
    return( bounded_side( p) == CGAL_ON_UNBOUNDED_SIDE);
}

bool
is_empty( ) const
{
    return( n_boundary_points == 0);
}

bool
is_degenerate( ) const
{
    return( n_boundary_points < 3);
}
```

```
// additional operations for checking
bool
operator == (
  const CGAL__Min_ellipse_2_adapterC2__Ellipse<_PT,_DA>& e) const
{
    if ( n_boundary_points != e.n_boundary_points)
        return( false);

    switch ( n_boundary_points) {
      case 0:
        return( true);
      case 1:
        return( boundary_point1 == e.boundary_point1);
      case 2:
        return(    (    ( boundary_point1 == e.boundary_point1)
                     && ( boundary_point2 == e.boundary_point2))
                || (    ( boundary_point1 == e.boundary_point2)
                     && ( boundary_point2 == e.boundary_point1)));
      case 3:
      case 5:
        return( conic1 == e.conic1);
      case 4:
        return(    (    ( conic1 == e.conic1)
                     && ( conic2 == e.conic2))
                || (    ( conic1 == e.conic2)
                     && ( conic2 == e.conic1)));
      default:
        CGAL_optimisation_assertion(
            ( n_boundary_points >= 0) &&
            ( n_boundary_points <= 5) ); }

    // keeps g++ happy
    return( false);
}

// I/O
friend
ostream&
operator << ( ostream& os,
    const CGAL__Min_ellipse_2_adapterC2__Ellipse<_PT,_DA>& e)
{
    const char* const  empty       = "";
    const char* const  pretty_head =
                       "CGAL_Min_ellipse_2_adapterC2::Ellipse( ";
    const char* const  pretty_sep  = ", ";
    const char* const  pretty_tail = ")";
```

```
    const char* const  ascii_sep   = " ";

    const char*  head = empty;
    const char*  sep  = empty;
    const char*  tail = empty;

    switch ( CGAL_get_mode( os)) {
      case CGAL_IO::PRETTY:
        head = pretty_head;
        sep  = pretty_sep;
        tail = pretty_tail;
        break;
      case CGAL_IO::ASCII:
        sep  = ascii_sep;
        break;
      case CGAL_IO::BINARY:
        break;
      default:
        CGAL_optimisation_assertion_msg(
            false, "CGAL_IO::mode invalid!");
        break; }

    os << head << e.n_boundary_points;
    switch ( e.n_boundary_points) {
      case 0:
        break;
      case 1:
        os << sep << e.boundary_point1;
        break;
      case 2:
        os << sep << e.boundary_point1
           << sep << e.boundary_point2;
        break;
      case 3:
      case 5:
        os << sep << e.conic1;
        break;
      case 4:
        os << sep << e.conic1
           << sep << e.conic2;
        break; }
    os << tail;

    return( os);
}
```

```
        friend
        istream&
        operator >> ( istream& is,
                      CGAL__Min_ellipse_2_adapterC2__Ellipse<_PT,_DA>& e)
        {
            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:
                CGAL_read( is, e.n_boundary_points);
                switch ( e.n_boundary_points) {
                  case 0:
                    break;
                  case 1:
                    is >> e.boundary_point1;
                    break;
                  case 2:
                    is >> e.boundary_point1
                       >> e.boundary_point2;
                    break;
                  case 3:
                  case 5:
                    is >> e.conic1;
                    break;
                  case 4:
                    is >> e.conic1
                       >> e.conic2;
                    break; }
                break;

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

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

This macro is invoked in definition 159.

5.7 Class Template `CGAL_Min_ellipse_2_adapterH2<PT,DA>`

First, we declare class templates `CGAL_Min_ellipse_2`, `CGAL_Min_ellipse_2_adapterH2` and `CGAL__Min_ellipse_2_adapterH2__Ellipse`.

Min_ellipse_2_adapterH2 declarations [138] ≡ {
```
template < class _Traits >
class CGAL_Min_ellipse_2;

template < class _PT, class _DA >
class CGAL_Min_ellipse_2_adapterH2;

template < class _PT, class _DA >
class CGAL__Min_ellipse_2_adapterH2__Ellipse;
```
}
This macro is invoked in definition 160.

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

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

Min_ellipse_2_adapterH2 interface and implementation [139] ≡ {
```
template < class _PT, class _DA >
class CGAL_Min_ellipse_2_adapterH2 {
  public:
    // types
    typedef  _PT  PT;
    typedef  _DA  DA;

    // nested types
    typedef  PT                                                  Point;
    typedef  CGAL__Min_ellipse_2_adapterH2__Ellipse<PT,DA>  Ellipse;

  private:
    DA      dao;                              // data accessor object
    Ellipse ellipse;                          // current ellipse
    friend
      class CGAL_Min_ellipse_2< CGAL_Min_ellipse_2_adapterH2<PT,DA> >;

  public:
    // creation
    Min_ellipse_2_adapterH2 constructors [140]

    // operations
    Min_ellipse_2_adapterH2 operations [141]
};
```
}
This macro is invoked in definition 160.

5.7.1 Constructors

Min_ellipse_2_adapterH2 constructors [140] ≡ {
```
    CGAL_Min_ellipse_2_adapterH2( const DA& da = DA())
        : dao( da), ellipse( da)
    { }
}
```
This macro is invoked in definition 139.

5.7.2 Operations

Min_ellipse_2_adapterH2 operations [141] ≡ {
```
    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 139.

5.7.3 Nested Type `Ellipse`

Min_ellipse_2_adapterH2 nested type 'Ellipse' [142] ≡ {
```
    template < class _PT, class _DA >
    class CGAL__Min_ellipse_2_adapterH2__Ellipse {
      public:
        // typedefs
        typedef  _PT  PT;
        typedef  _DA  DA;
```

```
  typedef           CGAL_ConicHPA2< PT, DA>  CT;
  typedef  typename _DA::RT                  RT;

private:
  // data members
  int  n_boundary_points;           // number of boundary points
  PT   boundary_point1, boundary_point2;  // two boundary points
  CT   conic1, conic2;                    // two conics
  RT   dr, ds, dt, du, dv, dw;            // the gradient vector

public:
  // types
  typedef  PT  Point;

  // creation
  CGAL__Min_ellipse_2_adapterH2__Ellipse( const DA& da)
      : conic1( da), conic2( da)
  { }

  void
  set( )
  {
      n_boundary_points = 0;
  }

  void
  set( const Point& p)
  {
      n_boundary_points = 1;
      boundary_point1   = p;
  }

  void
  set( const Point& p, const Point& q)
  {
      n_boundary_points = 2;
      boundary_point1   = p;
      boundary_point2   = q;
  }

  void
  set( const Point& p1, const Point& p2, const Point& p3)
  {
      n_boundary_points = 3;
      conic1.set_ellipse( p1, p2, p3);
  }
```

```
void
set( const Point& p1, const Point& p2,
     const Point& p3, const Point& p4)
{
    n_boundary_points = 4;
    CT::set_two_linepairs( p1, p2, p3, p4, conic1, conic2);
    dr = RT( 0);
    ds = conic1.r() * conic2.s() - conic2.r() * conic1.s(),
    dt = conic1.r() * conic2.t() - conic2.r() * conic1.t(),
    du = conic1.r() * conic2.u() - conic2.r() * conic1.u(),
    dv = conic1.r() * conic2.v() - conic2.r() * conic1.v(),
    dw = conic1.r() * conic2.w() - conic2.r() * conic1.w();
}

void
set( const Point&, const Point&,
     const Point&, const Point&, const Point& p5)
{
    n_boundary_points = 5;
    conic1.set( conic1, conic2, p5);
    conic1.analyse();
}

// predicates
CGAL_Bounded_side
bounded_side( const Point& p) const
{
    switch ( n_boundary_points) {
      case 0:
        return( CGAL_ON_UNBOUNDED_SIDE);
      case 1:
        return( ( p == boundary_point1) ?
                       CGAL_ON_BOUNDARY : CGAL_ON_UNBOUNDED_SIDE);
      case 2:
        return(    ( p == boundary_point1)
                || ( p == boundary_point2)
                || ( CGAL_are_ordered_along_lineH2(
                       boundary_point1, p,
                       boundary_point2, conic1.da())) ?
                        CGAL_ON_BOUNDARY : CGAL_ON_UNBOUNDED_SIDE);
      case 3:
      case 5:
        return( conic1.convex_side( p));
      case 4: {
        CT c( conic1.da());
        c.set( conic1, conic2, p);
        c.analyse();
```

```
        if ( ! c.is_ellipse()) {
            c.set_ellipse( conic1, conic2);
            c.analyse();
            return( c.convex_side( p)); }
        else {
            int tau_star = -c.vol_derivative( dr, ds, dt,
                                              du, dv, dw);
            return( CGAL_static_cast( CGAL_Bounded_side,
                                      CGAL_sign( tau_star))); } }
      default:
        CGAL_optimisation_assertion(
            ( n_boundary_points >= 0) &&
            ( n_boundary_points <= 5) ); }

    // keeps g++ happy
    return( CGAL_Bounded_side( 0));
}

bool
has_on_bounded_side( const Point& p) const
{
    return( bounded_side( p) == CGAL_ON_BOUNDED_SIDE);
}

bool
has_on_boundary( const Point& p) const
{
    return( bounded_side( p) == CGAL_ON_BOUNDARY);
}

bool
has_on_unbounded_side( const Point& p) const
{
    return( bounded_side( p) == CGAL_ON_UNBOUNDED_SIDE);
}

bool
is_empty( ) const
{
    return( n_boundary_points == 0);
}

bool
is_degenerate( ) const
{
    return( n_boundary_points < 3);
}
```

```
// additional operations for checking
bool
operator == (
  const CGAL__Min_ellipse_2_adapterH2__Ellipse<_PT,_DA>& e) const
{
    if ( n_boundary_points != e.n_boundary_points)
        return( false);

    switch ( n_boundary_points) {
      case 0:
        return( true);
      case 1:
        return( boundary_point1 == e.boundary_point1);
      case 2:
        return(    (    ( boundary_point1 == e.boundary_point1)
                     && ( boundary_point2 == e.boundary_point2))
                || (    ( boundary_point1 == e.boundary_point2)
                     && ( boundary_point2 == e.boundary_point1)));
      case 3:
      case 5:
        return( conic1 == e.conic1);
      case 4:
        return(    (    ( conic1 == e.conic1)
                     && ( conic2 == e.conic2))
                || (    ( conic1 == e.conic2)
                     && ( conic2 == e.conic1)));
      default:
        CGAL_optimisation_assertion(
            ( n_boundary_points >= 0) &&
            ( n_boundary_points <= 5) ); }

    // keeps g++ happy
    return( false);
}

// I/O
friend
ostream&
operator << ( ostream& os,
    const CGAL__Min_ellipse_2_adapterH2__Ellipse<_PT,_DA>& e)
{
    const char* const  empty       = "";
    const char* const  pretty_head =
                       "CGAL_Min_ellipse_2_adapterH2::Ellipse( ";
    const char* const  pretty_sep  = ", ";
    const char* const  pretty_tail = ")";
```

```
    const char* const  ascii_sep   = " ";

    const char*  head = empty;
    const char*  sep  = empty;
    const char*  tail = empty;

    switch ( CGAL_get_mode( os)) {
      case CGAL_IO::PRETTY:
        head = pretty_head;
        sep  = pretty_sep;
        tail = pretty_tail;
        break;
      case CGAL_IO::ASCII:
        sep  = ascii_sep;
        break;
      case CGAL_IO::BINARY:
        break;
      default:
        CGAL_optimisation_assertion_msg(
            false, "CGAL_IO::mode invalid!");
        break; }

    os << head << e.n_boundary_points;
    switch ( e.n_boundary_points) {
      case 0:
        break;
      case 1:
        os << sep << e.boundary_point1;
        break;
      case 2:
        os << sep << e.boundary_point1
           << sep << e.boundary_point2;
        break;
      case 3:
      case 5:
        os << sep << e.conic1;
        break;
      case 4:
        os << sep << e.conic1
           << sep << e.conic2;
        break; }
    os << tail;

    return( os);
}
```

```
        friend
        istream&
        operator >> ( istream& is,
                      CGAL__Min_ellipse_2_adapterH2__Ellipse<_PT,_DA>& e)
        {
            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:
                CGAL_read( is, e.n_boundary_points);
                switch ( e.n_boundary_points) {
                  case 0:
                    break;
                  case 1:
                    is >> e.boundary_point1;
                    break;
                  case 2:
                    is >> e.boundary_point1
                       >> e.boundary_point2;
                    break;
                  case 3:
                  case 5:
                    is >> e.conic1;
                    break;
                  case 4:
                    is >> e.conic1
                       >> e.conic2;
                    break; }
                break;

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

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

This macro is invoked in definition 160.

6 Tests

We test `CGAL_Min_ellipse_2` with the traits class implementation for optimisation algorithms, using exact arithmetic, i.e. Cartesian representation with number type `CGAL_Quotient<CGAL_Gmpz>` or `CGAL_Quotient<integer>` and homogeneous representation with number type `CGAL_Gmpz` or `integer`.

Min_ellipse_2 test (includes and typedefs) [143] ≡ {

```
#include <CGAL/Cartesian.h>
#include <CGAL/Homogeneous.h>
#include <CGAL/Min_ellipse_2.h>
#include <CGAL/Min_ellipse_2_traits_2.h>
#include <CGAL/Min_ellipse_2_adapterC2.h>
#include <CGAL/Min_ellipse_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_ellipse_2_traits_2< RepC >   TraitsC;
typedef  CGAL_Min_ellipse_2_traits_2< RepH >   TraitsH;
```

}

This macro is invoked in definition 161.

The command line option `-verbose` enables verbose output.

Min_ellipse_2 test (verbose option) [144] ≡ {

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

}

This macro is invoked in definition 161.

6.1 Code Coverage

We call each function of class `CGAL_Min_ellipse_2<Traits>` at least once to ensure code coverage.

Min_ellipse_2 test (code coverage) [145] ≡ {

```
    cover_Min_ellipse_2( verbose, TraitsC(), Rt());
    cover_Min_ellipse_2( verbose, TraitsH(), Rt());
}
```

This macro is invoked in definition 161.

Min_ellipse_2 test (code coverage test function) [146] ≡ {

```
    template < class Traits, class RT >
    void
    cover_Min_ellipse_2( bool verbose, const Traits&, const RT&)
    {
        typedef  CGAL_Min_ellipse_2< Traits >   Min_ellipse;
        typedef  Min_ellipse::Point             Point;
        typedef  Min_ellipse::Ellipse           Ellipse;

        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_ellipse  me;
            bool  is_valid = me.is_valid( verbose);
            bool  is_empty = me.is_empty();
            assert( is_valid);
            assert( is_empty);
        }

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

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

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

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

verr << endl << "four points constructor...";
{
    Min_ellipse  me( random_points[ 6],
                     random_points[ 7],
                     random_points[ 8],
                     random_points[ 9]);
    bool  is_valid = me.is_valid( verbose);
    assert( is_valid);
    assert( me.number_of_points() == 4);
}

verr << endl << "five points constructor...";
{
    Min_ellipse  me( random_points[ 10],
                     random_points[ 11],
                     random_points[ 12],
                     random_points[ 13],
                     random_points[ 14]);
    bool  is_valid = me.is_valid( verbose);
    assert( is_valid);
    assert( me.number_of_points() == 5);
```

```
}

verr << endl << "Point* constructor...";
Min_ellipse  me( random_points, random_points+9);
{
    Min_ellipse  me2( random_points, random_points+9, true);
    bool  is_valid  = me .is_valid( verbose);
    bool  is_valid2 = me2.is_valid( verbose);
    assert( is_valid);
    assert( is_valid2);
    assert( me .number_of_points() == 9);
    assert( me2.number_of_points() == 9);
    assert( me.ellipse() == me2.ellipse());
}

verr << endl << "list<Point>::const_iterator constructor...";
{
    Min_ellipse  me1( me.points_begin(), me.points_end());
    Min_ellipse  me2( me.points_begin(), me.points_end(), true);
    bool  is_valid1 = me1.is_valid( verbose);
    bool  is_valid2 = me2.is_valid( verbose);
    assert( is_valid1);
    assert( is_valid2);
    assert( me1.number_of_points() == 9);
    assert( me2.number_of_points() == 9);
    assert( me.ellipse() == me1.ellipse());
    assert( me.ellipse() == me2.ellipse());
}

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

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

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

verr << endl << "ellipse access already called above...";
```

```
verr << endl << "in-ellipse 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          = me.bounded_side( p);
        has_on_bounded_side   = me.has_on_bounded_side( p);
        has_on_boundary       = me.has_on_boundary( p);
        has_on_unbounded_side = me.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...";
me.insert( random_points[ 9]);
{
    bool  is_valid = me.is_valid( verbose);
    assert( is_valid);
    assert( me.number_of_points() == 10);
}

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

verr << endl << "list<Point>::const_iterator insert...";
{
    Min_ellipse  me2;
    me2.insert( me.points_begin(), me.points_end());
    bool  is_valid = me2.is_valid( verbose);
    assert( is_valid);
    assert( me2.number_of_points() == n);

    verr << endl << "clear...";
    me2.clear();
```

```
            is_valid = me2.is_valid( verbose);
        bool  is_empty = me2.is_empty();
        assert( is_valid);
        assert( is_empty);
    }

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

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

    verr << endl << "I/O...";
    {
        verr << endl << "  writing ‘test_Min_ellipse_2.ascii’...";
        ofstream os( "test_Min_ellipse_2.ascii");
        CGAL_set_ascii_mode( os);
        os << me;
    }
    {
        verr << endl << "  writing ‘test_Min_ellipse_2.pretty’...";
        ofstream os( "test_Min_ellipse_2.pretty");
        CGAL_set_pretty_mode( os);
        os << me;
    }
    {
        verr << endl << "  writing ‘test_Min_ellipse_2.binary’...";
        ofstream os( "test_Min_ellipse_2.binary");
        CGAL_set_binary_mode( os);
        os << me;
    }
    {
        verr << endl << "  reading ‘test_Min_ellipse_2.ascii’...";
        Min_ellipse me_in;
        ifstream is( "test_Min_ellipse_2.ascii");
        CGAL_set_ascii_mode( is);
        is >> me_in;
        bool    is_valid = me_in.is_valid( verbose);
        assert( is_valid);
        assert( me_in.number_of_points() == n);
        assert( me_in.ellipse() == me.ellipse());
    }
    verr << endl;
}
}
```

This macro is invoked in definition 161.

6.2 Traits Class Adapters

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

Min_ellipse_2 test (point classes) [147] ≡ {

```
// 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 161.

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

Min_ellipse_2 test (adapters test) [148] $\equiv$ {

```
    typedef  CGAL_Min_ellipse_2_adapterC2< MyPointC2, MyPointC2DA >
                                                              AdapterC2;
    typedef  CGAL_Min_ellipse_2_adapterH2< MyPointH2, MyPointH2DA >
                                                              AdapterH2;
    cover_Min_ellipse_2( verbose, AdapterC2(), Rt());
    cover_Min_ellipse_2( verbose, AdapterH2(), Rt());
}
```

This macro is invoked in definition 161.

6.3 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_ellipse_2 test (external test sets) [149] ≡ {

```
while ( argc > 1) {

    typedef  CGAL_Min_ellipse_2< TraitsH >  Min_ellipse;
    typedef  Min_ellipse::Point             Point;
    typedef  Min_ellipse::Ellipse           Ellipse;

    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_ellipse
    Min_ellipse  me2( points.begin(), points.end());
    bool  is_valid = me2.is_valid( verbose);
    assert( is_valid);

    // next file
    --argc;
    ++argv; }
```

}

This macro is invoked in definition 161.

7 Files

7.1 Min_ellipse_2.h

include/CGAL/Min_ellipse_2.h [150] ≡ {

Min_ellipse_2 header [165] (`include/CGAL/Min_ellipse_2.h`)

```
#ifndef CGAL_MIN_ELLIPSE_2_H
#define CGAL_MIN_ELLIPSE_2_H

// Class declaration
// =================
```
Min_ellipse_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_ellipse_2 interface [2]

```
// Function declarations
// =====================
// I/O
// ---
```
Min_ellipse_2 I/O operators declaration [22]

```
#ifdef CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION
#  include <CGAL/Min_ellipse_2.C>
#endif

#endif // CGAL_MIN_ELLIPSE_2_H
```

end of file line [164]
}
This macro is attached to an output file.

7.2 Min_ellipse_2.C

include/CGAL/Min_ellipse_2.C [151] ≡ {
Min_ellipse_2 header [165] (‘include/CGAL/Min_ellipse_2.C’)

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

Min_ellipse_2 I/O operators [23]

end of file line [164]
}
This macro is attached to an output file.

7.3 Optimisation_ellipse_2.h

include/CGAL/Optimisation_ellipse_2.h [152] ≡ {
Optimisation_ellipse_2 header [166] (‘include/CGAL/Optimisation_ellipse_2.h’)

```
#ifndef CGAL_OPTIMISATION_ELLIPSE_2_H
#define CGAL_OPTIMISATION_ELLIPSE_2_H

// Class declaration
// =================
```

Optimisation_ellipse_2 declaration [26]

```
// Class interface
// ===============
// includes
#ifndef CGAL_POINT_2_H
#  include <CGAL/Point_2.h>
#endif
#ifndef CGAL_CONIC_2_H
#  include <CGAL/Conic_2.h>
#endif
#ifndef CGAL_OPTIMISATION_ASSERTIONS_H
#  include <CGAL/optimisation_assertions.h>
#endif
```

Optimisation_ellipse_2 interface [27]

```
// Function declarations
// =====================
// I/O
// ---
```
Optimisation_ellipse_2 I/O operators declaration [37]

```
#ifdef CGAL_CFG_NO_AUTOMATIC_TEMPLATE_INCLUSION
#  include <CGAL/Optimisation_ellipse_2.C>
#endif

#endif // CGAL_OPTIMISATION_ELLIPSE_2_H
```

end of file line [164]
}
This macro is attached to an output file.

7.4 Optimisation_ellipse_2.C

include/CGAL/Optimisation_ellipse_2.C [153] ≡ {
Optimisation_ellipse_2 header [166] (‘include/CGAL/Optimisation_ellipse_2.C’)

```
// Class implementation (continued)
// ================================
// includes
#ifndef CGAL_OPTIMISATION_ASSERTIONS_H
#  include <CGAL/optimisation_assertions.h>
#endif

// I/O
// ---
```
Optimisation_ellipse_2 I/O operators [38]

end of file line [164]
}
This macro is attached to an output file.

7.5 Conic_misc.h

Here we collect all types and functions that are independent of the representation type `R`. These are the declarations of the enumeration types `CGAL_Conic_type` and `CGAL_Convex_side`, and the functions in connection with the solution of cubic equations.

include/CGAL/Conic_misc.h [154] ≡ {
Conic_2 header [167] (‘include/CGAL/Conic_misc.h’)

```
#ifndef CONIC_MISC_H
```

```
#define CONIC_MISC_H
```

Conic_2 declaration [39]
Conic_type declaration [44]
Convex_side declaration [49]

function CGAL_best_value [116]
function CGAL_solve_cubic [113]

```
#endif // CONIC_MISC_H
```

end of file line [164]
}
This macro is attached to an output file.

7.6 Conic_2.h

This file contains the implementation of the class `CGAL_Conic_2<R>`. Depending on the loaded representation classes, the representation specific classes `CGAL_ConicCPA2<PT,DA>` and/or `CGAL_ConicHPA2<PT,DA>` are included before that.

include/CGAL/Conic_2.h [155] ≡ {
Conic_2 header [167] (‘`include/CGAL/Conic_2.h`’)

```
#ifndef CGAL_CONIC_2_H
#define CGAL_CONIC_2_H

#ifndef CGAL_REP_CLASS_DEFINED
#  error  no representation class defined
#endif // CGAL_REP_CLASS_DEFINED

#ifdef CGAL_CARTESIAN_H
#  include <CGAL/ConicHPA2.h>
#endif

#ifdef CGAL_HOMOGENEOUS_H
#  include <CGAL/ConicCPA2.h>
#endif
```

Optimisation_ellipse_2 declaration [26]

```
// Class interface and implementation
// ==================================
// includes
#ifndef CGAL_POINT_2_H
#  include <CGAL/Point_2.h>
#endif
```

Conic_2 interface and implementation [40]

```
// I/O
// ---
#ifndef CGAL_NO_OSTREAM_INSERT_CONIC_2
```
Conic_2 I/O routines [64]
```
#endif // CGAL_NO_OSTREAM_INSERT_CONIC_2

#endif // CGAL_CONIC_2_H
```

end of file line [164]

}

This macro is attached to an output file.

7.7 ConicCPA2.h

include/CGAL/ConicCPA2.h [157] ≡ {

Conic_2 header [167] (‘include/CGAL/ConicCPA2.h’)

```
#ifndef CGAL_CONICCPA2_H
#define CGAL_CONICCPA2_H

// Class declarations
// ==================
```
ConicCPA2 declaration [65]
```
// Class interface and implementation
// ==================================
// includes
#ifndef CGAL_CONIC_MISC_H
#  include <CGAL/Conic_misc.h>
#endif
#ifndef CGAL_OPTIMISATION_ASSERTIONS_H
#  include <CGAL/optimisation_assertions.h>
#endif
```

ConicCPA2 interface and implementation [67]

```
// I/O
// ---
#ifndef CGAL_NO_OSTREAM_INSERT_CONICCPA2
```
ConicCPA2 I/O routines [129]
```
#endif // CGAL_NO_OSTREAM_INSERT_CONICCPA2

#endif // CGAL_CONICCPA2_H
```

end of file line [164]
}
This macro is attached to an output file.

7.8 ConicHPA2.h

include/CGAL/ConicHPA2.h [156] ≡ {
Conic_2 header [167] (‘include/CGAL/ConicHPA2.h’)

```
#ifndef CGAL_CONICHPA2_H
#define CGAL_CONICHPA2_H

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

ConicHPA2 declaration [66]

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

ConicHPA2 interface and implementation [68]

```
// I/O
// ---
#ifndef CGAL_NO_OSTREAM_INSERT_CONICHPA2
```

ConicHPA2 I/O routines [130]

```
#endif // CGAL_NO_OSTREAM_INSERT_CONICHPA2

#endif // CGAL_CONICHPA2_H
```

end of file line [164]
}
This macro is attached to an output file.

7.9 Min_ellipse_2_traits_2.h

include/CGAL/Min_ellipse_2_traits_2.h [158] ≡ {
Min_ellipse_2 header [165] (‘include/CGAL/Min_ellipse_2_traits_2.h’)

```
#ifndef CGAL_MIN_ELLIPSE_2_TRAITS_2_H
```

```
#define CGAL_MIN_ELLIPSE_2_TRAITS_2_H

// Class declarations
// ==================
```
Min_ellipse_2_traits_2 declarations [131]

```
// Class interface and implementation
// ==================================
// includes
#ifndef CGAL_POINT_2_H
#  include <CGAL/Point_2.h>
#endif
#ifndef CGAL_OPTIMISATION_ELLIPSE_2_H
#  include <CGAL/Optimisation_ellipse_2.h>
#endif
```

Min_ellipse_2_traits_2 interface and implementation [132]

```
#endif // CGAL_MIN_ELLIPSE_2_TRAITS_2_H
```

end of file line [164]

}

This macro is attached to an output file.

7.10 Min_ellipse_2_adapterC2.h

include/CGAL/Min_ellipse_2_adapterC2.h [159] ≡ {

Min_ellipse_2 header [165] (‘include/CGAL/Min_ellipse_2_adapterC2.h’)

```
#ifndef CGAL_MIN_ELLIPSE_2_ADAPTERC2_H
#define CGAL_MIN_ELLIPSE_2_ADAPTERC2_H

// Class declarations
// ==================
```
Min_ellipse_2_adapterC2 declarations [133]

```
// Class interface and implementation
// ==================================
// includes
#ifndef CGAL_CONICHPA2_H
#  include <CGAL/ConicHPA2.h>
#endif
#ifndef CGAL_OPTIMISATION_ASSERTIONS_H
#  include <CGAL/optimisation_assertions.h>
#endif

template < class PT, class DA >
```

```
bool
CGAL_are_ordered_along_lineC2( const PT& p, const PT& q, const PT& r,
                               const DA& da)
{
    typedef  typename DA::FT  FT;

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

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

    // p,q,r collinear?
    if ( ! CGAL_is_zero( ( px-rx) * ( qy-ry) - ( py-ry) * ( qx-rx)))
        return( false);

    // p,q,r vertical?
    if ( px != rx)
        return(    ( ( px < qx) && ( qx < rx))
                || ( ( rx < qx) && ( qx < px)));
    else
        return(    ( ( py < qy) && ( qy < ry))
                || ( ( ry < qy) && ( qy < py)));
}
```

Min_ellipse_2_adapterC2 interface and implementation [134]

```
// Nested type 'Ellipse'
```

Min_ellipse_2_adapterC2 nested type 'Ellipse' [137]

```
#endif // CGAL_MIN_ELLIPSE_2_ADAPTERC2_H
```

end of file line [164]

}

This macro is attached to an output file.

7.11 Min_ellipse_2_adapterH2.h

include/CGAL/Min_ellipse_2_adapterH2.h [160] $\equiv$ {

Min_ellipse_2 header [165] (`include/CGAL/Min_ellipse_2_adapterH2.h`)

```
#ifndef CGAL_MIN_ELLIPSE_2_ADAPTERH2_H
```

```
#define CGAL_MIN_ELLIPSE_2_ADAPTERH2_H

// Class declarations
// ==================
```
Min_ellipse_2_adapterH2 declarations [138]

```
// Class interface and implementation
// ==================================
// includes
#ifndef CGAL_CONICCPA2_H
#  include <CGAL/ConicCPA2.h>
#endif
#ifndef CGAL_OPTIMISATION_ASSERTIONS_H
#  include <CGAL/optimisation_assertions.h>
#endif

template < class PT, class DA >
bool
CGAL_are_ordered_along_lineH2( const PT& p, const PT& q, const PT& r,
                               const DA& da)
{
    typedef  typename DA::RT  RT;

    RT  phx;
    RT  phy;
    RT  phw;
    RT  qhx;
    RT  qhy;
    RT  qhw;
    RT  rhx;
    RT  rhy;
    RT  rhw;

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

    // p,q,r collinear?
    if ( ! CGAL_is_zero(
                ( phx*rhw - rhx*phw) * ( qhy*rhw - rhy*qhw)
              - ( phy*rhw - rhy*phw) * ( qhx*rhw - rhx*qhw)))
        return( false);

    // p,q,r vertical?
    if ( phx*rhw != rhx*phw)
        return(    ( ( phx*qhw < qhx*phw) && ( qhx*rhw < rhx*qhw))
                || ( ( rhx*qhw < qhx*rhw) && ( qhx*phw < phx*qhw)));
```

```
        else
            return(    ( ( phy*qhw < qhy*phw) && ( qhy*rhw < rhy*qhw))
                    || ( ( rhy*qhw < qhy*rhw) && ( qhy*phw < phy*qhw)));
    }
```

Min_ellipse_2_adapterH2 interface and implementation [139]

```
    // Nested type 'Ellipse'
```

Min_ellipse_2_adapterH2 nested type 'Ellipse' [142]

```
    #endif // CGAL_MIN_ELLIPSE_2_ADAPTERH2_H
```

end of file line [164]

}

This macro is attached to an output file.

7.12 test_Min_ellipse_2.C

test/Optimisation/test_Min_ellipse_2.C [161] ≡ {

Min_ellipse_2 header [165] ('test/Optimisation/test_Min_ellipse_2.C')

Min_ellipse_2 test (includes and typedefs) [143]

```
    // code coverage test function
    // ---------------------------
```

Min_ellipse_2 test (code coverage test function) [146]

```
    // point classes for adapters test
    // -------------------------------
```

Min_ellipse_2 test (point classes) [147]

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

Min_ellipse_2 test (verbose option) [144]

```
        // code coverage
        // -------------
```

Min_ellipse_2 test (code coverage) [145]

```
        // adapters test
        // -------------
```

```
        Min_ellipse_2 test (adapters test) [148]

        // external test sets
        // ------------------
        Min_ellipse_2 test (external test sets) [149]

        return( 0);
    }

    end of file line [164]
}
```

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 [162] (◇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 165, 166, and 167.

dividing line [163] **ZM** ≡ {

```
// ============================================================
```

}

This macro is invoked in definitions 2 and 27.

end of file line [164] **ZM** ≡ {

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

}

This macro is invoked in definitions 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, and 161.

Min_ellipse_2 header [165] (⋄1)**M** ≡ {

file header [162] (‘2D Smallest Enclosing Ellipse’,‘⋄1’,
‘Optimisation’,‘Optimisation/Min_ellipse_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 14:21:11 $’)

}

This macro is invoked in definitions 150, 151, 158, 159, 160, and 161.

Optimisation_ellipse_2 header [166] (⋄1)**M** ≡ {

file header [162] (‘2D Optimisation Ellipse’,‘⋄1’,
‘Optimisation’,‘Optimisation/Min_ellipse_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 14:21:11 $’)

}

This macro is invoked in definitions 152 and 153.

Conic_2 header [167] (⋄1)**M** ≡ {

file header [162] (‘2D Conic’,‘⋄1’,
‘Optimisation’,‘Optimisation/Conic_2’,
‘Bernd Gärtner <gaertner@inf.ethz.ch>’,
‘Sven Schönherr’,
‘ETH Zurich (Bernd Gärtner <gaertner@inf.ethz.ch>)’,
‘$Revision: 4.1 $’,‘$Date: 1998/03/30 14:18:08 $’)

}

This macro is invoked in definitions 154, 155, 156, and 157.

References

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

[2] B. Char, K. Geddes, G. Gonnet, B. Leong, M. Monagan, and S. Watt. *Maple V Library Reference Manual.* Springer Verlag, 1991.

[3] 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.

[4] 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/`.

[5] 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.* Apr. 1998. CGAL R1.0.

[6] B. Gärtner and S. Schönherr. Exact primitives for smallest enclosing ellipses. In *Proc. 13th Annu. ACM Symp. on Computational Geometry*, pages 430–432, 1997.

[7] B. Gärtner and S. Schönherr. Smallest enclosing ellipses – fast and exact. Serie B – Informatik B 97-03, Freie Universität Berlin, Germany, June 1997. URL `http://www.inf.fu-berlin.de/inst/pubs/`.

[8] B. Gärtner and S. Schönherr. Smallest enclosing circles – 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/`.

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

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

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

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

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

[14] 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