A Generic Lazy Evaluation Scheme for Exact Geometric Computations

Sylvain Pion, Andreas Fabri

To cite this version:

Sylvain Pion, Andreas Fabri. A Generic Lazy Evaluation Scheme for Exact Geometric Computations. Science of Computer Programming, Elsevier, 2011, Special issue on library-centric software design (LCSD 2006), 76 (4), pp.307-323. 10.1016/j.scico.2010.09.003 . inria-00562300

HAL Id: inria-00562300

https://hal.inria.fr/inria-00562300

Submitted on 3 Feb 2011

A Generic Lazy Evaluation Scheme for Exact Geometric Computations

Sylvain Pion [1,*]

INRIA, Sophia Antipolis, France

Andreas Fabri [1]

GeometryFactory, Grasse, France

Abstract

We present a generic C++ design to perform exact geometric computations efficiently using lazy evaluations. Exact geometric computations are critical for the robustness of geometric algorithms. Their efficiency is also important for many applications, hence the need for delaying the costly exact computations at run time until they are actually needed, if at all. Our approach is generic and extensible in the sense that it is possible to make it a library that users can apply to their own geometric objects and primitives. It involves techniques such as generic functor-adaptors, static and dynamic polymorphism, reference counting for the management of directed acyclic graphs, and exception handling for triggering exact computations when needed. It also relies on multi-precision arithmetic as well as interval arithmetic. We apply our approach to the whole geometry kernel of CGAL.

Key words: computational geometry, exact geometric computation, numerical robustness, interval arithmetic, lazy evaluation, generic programming, C++, CGAL

* Corresponding author. Address: INRIA Sophia Antipolis-Méditerranée, BP 93, 06902 Sophia Antipolis cedex, France.
Email addresses: `Sylvain.Pion@sophia.inria.fr` (Sylvain Pion), `andreas.fabri@geometryfactory.com` (Andreas Fabri).

[1] This research is supported in part by the IST Programme of the EU as a Shared-cost RTD (FET Open) Project under Contract No IST-006413 (ACS - Algorithms for Complex Shapes).

1 Introduction

Computational geometry can be regarded as the art of designing algorithms and data structures that organize geometric data and extract structure out of it. The convex hull of a set of points, its Delaunay triangulation (see Figure 1) and Voronoi diagram are basic examples of such algorithms and data structures. They are usually described in the literature using the simple Real-RAM model, which assumes in particular that operations on real numbers are performed exactly and in constant time. Geometric algorithms are unique in the sense that they mix combinatorial techniques with numerical computations. Indeed, they build structures akin to graphs, and they handle point coordinates.

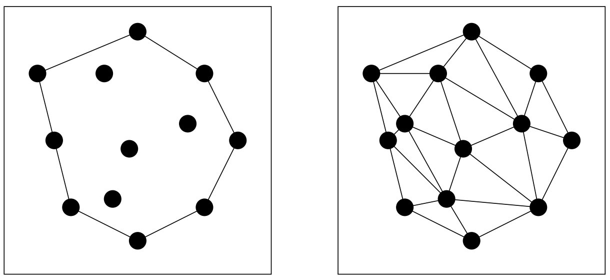
Fig. 1. Convex hull and Delaunay triangulation of a set of points in the plane.

This dual nature is at the root of many practical *non-robustness issues*, because approximate numerical computations violate key geometric properties, which in turn interferes with the discrete logic of combinatorial algorithms. Indeed, the crucial exactness assumption is not satisfied when floating-point arithmetic is used to implement operations on real numbers. With the increasing use of digital sensors like scanners or digital cameras in numerous areas, the amount of geometric data that needs to be processed is growing tremendously, making geometric algorithms more relevant than ever. This also pushes for a careful, efficient and general treatment of the numerical issues. In particular, the development of the CGAL library [1], a large collection of geometric algorithms implemented in C++, strongly expresses such a need.

The *exact geometric-computation* paradigm investigated by Yap [2] offers a solution with a wide practical applicability. This paradigm states that, in order to achieve robustness of an algorithm, it is enough to guarantee the exactness of the results of the *geometric predicates* that it evaluates. These predicates are the functions that take as input the numerical data and compute a discrete result, such as a Boolean value, that is used in the control flow of the algorithm to build its combinatorial output. A simple example of predicate is the lexicographic comparison of the coordinates of two points and a more interesting, but more involved example is the *orientation* predicate described below.

Guaranteeing the exactness of the predicates can be done using an exact arithmetic, which is implemented, for example, by multi-precision integers or rational numbers (also known as arbitrary precision arithmetic). Moreover, for efficiency reasons, tools for certified approximate computations based on floating-point arithmetic, like interval arithmetic, are also used as a first step in the evaluation of the predicates, since they allow amortizing the cost of exact arithmetic operations. Such techniques are known as *arithmetic filtering*, since the interval arithmetic step filters out the easy instances.

Naturally, it is common for geometric algorithms to call predicates over intermediate geometric objects that have been built from the input numerical data. A simple example of *geometric construction* is the computation of the intersection point of two segments specified by their end points. In this case, the exact geometric-computation paradigm implies that the intermediate geometric objects also need to be known exactly.

An efficient approach to implement such exact geometric constructions is to perform exact computations in a lazy way at the level of geometric objects, rather than at the underlying arithmetic level. It consists in computing a certified approximation of the objects using interval arithmetic and memorizing the necessary information such that a representation using exact arithmetic can be computed on demand, that is, when the exact computation of a predicate will later require it due to insufficient precision of the initial approximation. This approach is mentioned in [3] and an implementation is described in [4]. However, this implementation does not use the generic-programming paradigm, although the approach is fundamentally general. This missing part is the contribution of our paper.

In this paper we describe a generic design using C++ that allows these filtering and lazy exact computation techniques to apply to a large number of geometric primitives. In particular, we consider the complete geometry-kernel of CGAL.

The paper is organized as follows. Section 2 gives some background information about arithmetic tools and specific C++ features that are necessary for the detailed understanding of the paper. Section 3 describes the context and motivation in geometric computing in more detail, as well as the basics of a generic geometry-kernel allowing the use of different arithmetics and what can be done at this level. Section 4 presents our main design, namely, how geometric predicates, constructions and objects are adapted, while Section 5 illustrates how our scheme can be applied to the user own geometric objects and primitives. We also provide some benchmark data in Section 6 that confirms the benefit of our design and implementation. Finally, we list a few open questions related to our design in Section 7 and conclude with ideas for future work.

2 Prerequisites

This section introduces some auxiliary notions that help understanding the core of the paper.

2.1 Variadic C++ Templates

We use the C++ programming language in the examples of code throughout this paper. However, in order to improve clarity, we consider two small deviations. First, given that we use templates as a means to implement generic programming, we improve readability by omitting some uses of the `typename` keyword that are normally formally required when referring to a nested type in *dependant contexts*.

Moreover and more importantly, for the sake of genericity, we use functions of arbitrary arity. This is implementable in C++ by providing overloads up to a fixed arity, but for the purpose of simplifying the exposition, we use variadic templates [5]. This feature, which will most likely become part of C++0x, the next revision of the C++ standard, allows a template to take an unspecified number of parameters and to apply expressions to them repeatedly using an ellipsis "..." as the syntax. In order to avoid confusion, we use additional spaces ". . ." for normal (non-variadic) ellipses. The following example:

```
template < typename... Args >
int fog(Args ... args)
{ return f(g(args)...); }
```

is equivalent to the following set of overloads for any arity:

```
int fog()
{ return f(); }

template < typename A1 >
int fog(A1 a1)
{ return f(g(a1)); }

template < typename A1, typename A2 >
int fog(A1 a1, A2 a2)
{ return f(g(a1), g(a2)); }

. . .
```

Let us also introduce some related tools. First, the variadic template class `tuple` stores any number of objects of any types.

```
template < typename... Args >
class tuple;
```

Then, the `apply` function calls a functor to each element of a `tuple`, and returns the results in a `tuple` of the appropriate type. We do not specify here how to obtain this type, it is named `X...` in the following example.

```
template < typename F, typename... Args >
tuple<X...> apply(F f, tuple<Args...> args);
```

Finally, the `bind_tuple` function binds the first arguments of a functor `f` to values passed as a tuple. It returns a functor of type `G`, which simply forwards its arguments to `f` after those of the tuple.

```
template < typename F, typename... Args >
G bind_tuple(F f, tuple<Args...> args);
```

Here is an example putting all these tools together, where `apply` is called with a functor that doubles its argument, and `bind_tuple` is applied to a functor performing additions.

```
struct Doubles {
  template < typename T >
  T operator()(T t) const { return 2*t; }
};

tuple<int, double> t(2, 3.14);
tuple<int, double> u = apply(t, Doubles()); // u holds (4, 6.28)

struct Add {
  double operator()(int a, double b) const { return a+b; }
  double operator()(int a, double b, int c) const { return a+b+c; }
};

double v = bind_tuple(Add(), t)();  // v holds 5.14
double w = bind_tuple(Add(), u)(6); // w holds 22.28
```

2.2 Arithmetic Tools

We describe now the computer arithmetics that we use: floating-point arithmetic, interval arithmetic and multi-precision arithmetic.

2.2.1 Floating-Point Arithmetic

Floating-point arithmetic is the usual way to compute with approximations of real numbers on a computer. The IEEE-754 standard defines all the details of its behavior, and its correct use is described in a well known article by Goldberg [6]. We only recall the properties that matter here. The operations considered are the addition, subtraction, multiplication, division and square root. Floating-point values are mostly finite real numbers, but $+\infty$ and $-\infty$ are also representable.

First, arithmetic operations are necessarily approximate, which means that exactness is generally lost. This process is called *directed rounding*, as the result of operations can be seen as if they were performed exactly, but then rounded to one of the nearest floating-point values enclosing the exact value. By default, the nearest value is selected, this is rounding to nearest. The other relevant modes are rounding towards plus and minus infinity, which respectively select the representable floating-point value that is immediately smaller or larger than the exact result.

The *rounding error*, that is, the distance between the exact result of an operation and its floating-point equivalent, can be bounded, since it depends on the distance between consecutive floating-point values, which is specified by the standard. Its value is $0.5 \times \texttt{ulp}(x)$ where x is the result, and `ulp` is the unit in the last place function, which is the difference between x and the next floating-point value. In double precision, $\texttt{ulp}(1)$ equals 2^{-53}.

This property is interesting, but it exhibits a major problem when it comes to computation, which means composing operations. Indeed, considering the expression $a + b + c$, it is not possible to bound the rounding error as easily in terms of the floating-point result. Simply consider what happens when $a = 1$, $b = 2^{-60}$ and $c = -1$ to observe what is referred to as a catastrophic cancellation.

As a consequence, this model of approximate computation with real functions is hard to deal with when it comes to controlling the rounding errors. Instead, we present interval arithmetic, a tool adapted to solve this problem.

2.2.2 Interval Arithmetic

Interval arithmetic can be seen as an alternative to the floating-point model, but it can also be implemented easily on top of it. This concept has been introduced by Moore [7] as the basis for the field of interval analysis.

The idea is simple; an interval is used to represent the rounding error attached to a variable by guaranteeing that it contains its exact value. This key property is named the *containment property* and the arithmetic operations are carefully defined to propagate it. In practice, two floating-point numbers are used to encode the lower and upper bounds of the interval, and therefore an approximation of a real number x is represented by an interval $[\texttt{x.lower}, \texttt{x.upper}]$ which contains x. Note that the lower bound can equate $-\infty$ and the upper bound $+\infty$.

The arithmetic operations are typically implemented by making use of the rounding modes towards $+\infty$ and $-\infty$. For example, the addition of two intervals $x + y$ is computed as $[\texttt{x.lower}\underline{+}\texttt{y.lower}, \texttt{x.upper}\overline{+}\texttt{y.upper}]$, where $\underline{+}$ and $\overline{+}$ respectively denote the floating-point addition with rounding towards $-\infty$ and $+\infty$. Similarly, the subtraction $x - y$ is computed as $[\texttt{x.lower}\underline{-}\texttt{y.upper}, \texttt{x.upper}\overline{-}\texttt{y.lower}]$. The multiplication follows the same principle except that its definition requires a few case distinctions and is therefore less straightforward. Other functions can be defined and implemented similarly.

Due to this outward rounding, we can intuitively say that the width of the intervals will tend to grow as operations are accumulated. This can lead to a complete ineffectivity of the method for some algorithms which perform lots of such accumulations, but fortunately this is not a problem for many practical other cases.

In summary, it is possible to build an interval arithmetic on top of the floating-point one relatively simply and at a running time that is a few times slower.

2.2.2.1 Rounding Mode Changes We now consider a remark on the implementation. Current hardware is advantageous to the floating-point model, since it implements the basic operations directly, while interval arithmetic requires both twice the number of registers (for storing the two bounds) and frequent changes of the rounding mode in general. Most CPUs provide a way to change the rounding direction as a special register that can be modified by specific instructions. These rounding mode changes slow down the interval operations substantially. Fortunately, it is possible to remove some of them by making a few observations. The first is that it is sometimes possible, like in the common case of the interval addition, to replace an operation rounded towards $-\infty$ by one rounded towards $+\infty$. This gives $[-((-\texttt{x.lower})\overline{-}\texttt{y.lower}), \texttt{x.upper}\overline{+}\texttt{y.upper}]$. This allows the replacement of some rounding mode changes by some faster equivalent computations that can even sometimes be folded by the compiler. But the bulk of the potential improvements follows from the observation that consecutive interval operations can keep the rounding mode set towards $+\infty$, so that a block of code using only $+, -, \times$ and $\div$ can perform a single change of rounding mode at the beginning and one restoration at the end.

For the purpose of code illustration, we show a C++ class that performs this initialization in its constructor and the restoration in its destructor. We also show how to use it for defining the interval addition as an example.

```
class Protect_FPU_rounding {
  int bak;
public:
  Protect_FPU_rounding(int mode = FE_UPWARD)
    : bak(std::fegetround()) { std::fesetround(mode); }
  ~Protect_FPU_rounding() { std::fesetround(bak); }
};

Interval operator+(Interval a, Interval b) {
  Protect_FPU_rounding P;
  return Interval(-((-a.lower()) - b.lower()), a.upper() + b.upper());
}
```

A final note on this issue: besides the parameterization by the types of the bounds, it is useful to consider parameterizing an interval class by a Boolean flag specifying if the rounding mode has been set to $+\infty$ before executing the function. Indeed, this allows to skip the rounding mode change at each call by moving this responsibility higher up in the call tree.

2.2.2.2 Interval Comparisons For reasons of generic programming, it is also useful to consider the definition of comparison operators over intervals, corresponding to the comparison over real numbers. The semantic of a comparison between intervals being true (resp. false) is that it is true (resp. false) for all possible real values chosen in both interval arguments, otherwise it is indeterminate. The latter case cannot be returned through a Boolean value, but can be returned through a more complex type holding this uncertainty information, or more simply, by throwing an exception.

```
class Unsafe_comparison {};

bool operator<(Interval a, Interval b) {
  if (a.upper() <  b.lower()) return true;
  if (a.lower() >= b.upper()) return false;
  throw Unsafe_comparison();
}
```

2.2.3 Multi-Precision Arithmetic

Geometric data naturally has a numeric aspect since it models entities using approximations of real numbers. However, geometric computations also have a strong algebraic structure, and in order to compute algebraic expressions exactly, it is necessary to be able to compute without approximations. For this, we can use multi-precision arithmetic, which performs exact computations over the integers $\mathbb{Z}$ and the rational numbers $\mathbb{Q}$ (up to the available memory).

It is worth noting that the time complexity and memory size used by this model are typically here at least one to two orders of magnitude worse than for the approximate models described above. Libraries providing such functionality exist, for example the C library GMP [8].

3 Exact Geometric Computations and the CGAL Kernel

3.1 Exact Geometric Computations

As mentioned in the introduction, when programming geometric algorithms, we usually consider some low-level functions – called geometric *predicates* – that take some numerical data as input in the form of geometric objects such as points, and return a Boolean or an enumerated value. This discrete return value is then used to direct the flow of the program to construct some higher-level data structure such as a graph. Internally, these predicates typically perform comparisons of numerical values computed from the input. A classic example is the `orientation` predicate of three points in the plane, which returns an enumerated value that indicates whether the three points form a right turn, are collinear, or form a left turn (see Figure 2). Using Cartesian coordinates for the points, the orientation is computed as the sign (as a three-valued function: -1, 0, 1) of the following 3-dimensional determinant, which reduces to a 2-dimensional one:

$$\begin{vmatrix} 1 & 1 & 1 \\ p.x() & q.x() & r.x() \\ p.y() & q.y() & r.y() \end{vmatrix} = \begin{vmatrix} q.x() - p.x() & r.x() - p.x() \\ q.y() - p.y() & r.y() - p.y() \end{vmatrix}$$

Most predicates can be expressed as signs of polynomial expressions over the coordinates of the input points. Evaluating such a function with floating-point arithmetic is going to introduce rounding errors, which can lead to an approximate value having a sign that differs from the exact one. The impact of wrong signs on the geometric algorithms can be disastrous, as for example, it can break some invariants like planarity of a graph, or make the program loop forever or simply crash. Didactic examples of harmful consequences can be found in [9].

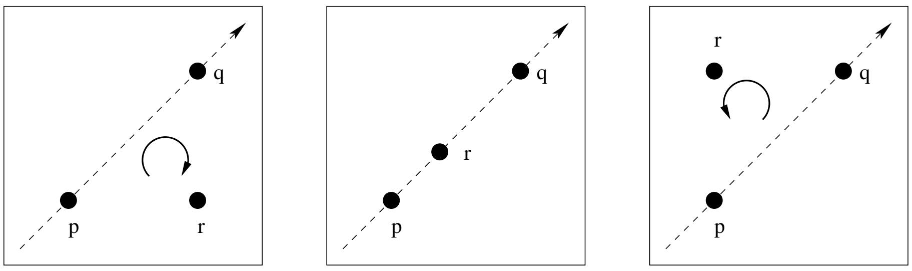

Fig. 2. The orientation predicate of 3 points p, q, r in the plane: right turn, collinear and left turn.

Operations computing new geometric objects, like the point at the intersection of two non-parallel lines, the circumcenter of three non-collinear points, or the midpoint of two points, are called geometric *constructions*. We use the term geometric *primitives* when referring to either predicates or constructions.

In order to tackle the non-robustness issues, many solutions have been proposed. We focus here on the *exact geometric-computation paradigm* [2], as it is a general solution. This paradigm states that in order to ensure the correct execution of the algorithms, it is enough that all predicates return exact results. Basically, this means that all comparisons of numerical values need to be performed exactly.

A natural way to perform the exact evaluation of predicates is to evaluate the numerical expressions using exact arithmetic. For example, since most computations are signs of polynomials, it is enough to use multi-precision rational arithmetic provided by libraries such as GMP [8]. Note that exact arithmetic is also available for all algebraic computations using libraries such as CORE [10] or LEDA [11], which is useful when doing geometry over curved objects. This solution works well, but it tends to be very slow.

We next describe the implementation of the geometry kernel of CGAL, which provides a generic implementation of these methods using C++ templates. We use it later to describe the design of our lazy exact evaluation framework.

3.2 The Geometry Kernel of CGAL

The geometric algorithms of CGAL are parameterized by the geometry they apply to. The geometry takes the form of a *kernel* [12,13] regrouping the types of the geometric objects such as points, segments and lines, as well as the geometric primitives operating on them, under the form of functors. The CGAL kernel provides over 100 predicates and 150 constructions. Hence, uniformity and genericity are crucial for both the interface and the maintenance.

CGAL provides several models of kernels. The basic families are the class templates Cartesian and Homogeneous, which are parameterized by the number type used to store the coordinates of the points and to compute arithmetic operations inside the primitives. They respectively differ in the Cartesian (such as (x, y) for a 2D point) or homogeneous (such as (hx, hy, hw), equivalent to the Cartesian $(hx/hw, hy/hw)$) representation of the points. Their interfaces are otherwise similar and look like the following:

```
template < typename NT >
struct Cartesian {
  // Number type used for the coordinates
  typedef NT           FT;

  // Geometric objects
  typedef . . .        Point_2;
  typedef . . .        Point_3;
  typedef . . .        Segment_2;
  . . .

  // Functors for predicates
  typedef . . .        Compare_x_2;
  typedef . . .        Orientation_2;
  . . .

  // Functors for constructions
  typedef . . .        Construct_midpoint_2;
  typedef . . .        Construct_circumcenter_2;
  . . .
};
```

Variants named Simple_cartesian and Simple_homogeneous are also provided, and they differ from the non-simple variants only in that they do not offer reference counting for the geometric objects.

These basic template models allow to use floating-point, multi-precision rational or interval arithmetic for example. More precisely, CGAL provides a hierarchy of concepts for the *number types* that describe the requirements on types to be valid number types for instantiating some kernel primitives. For example, some primitive might require a division or a square root, while most only need additions, subtractions, multiplications and comparisons. We refer the reader to the CGAL documentation for more details on this particular topic.

The functors are implemented in the following way (here the return type of the predicate is a three-valued enumerated type):

```
template < typename Kernel >
```

```
class Orientation_2 {
  typedef Kernel::Point_2     Point;
  typedef Kernel::FT          FT;
public:
  typedef CGAL::Orientation   result_type;

  result_type operator()(Point p, Point q, Point r) const {
    FT det = (q.x() - p.x()) * (r.y() - p.y())
           - (r.x() - p.x()) * (q.y() - p.y());
    if (det > 0) return LEFT_TURN;
    if (det < 0) return RIGHT_TURN;
    return COLLINEAR;
  }
};

template < typename Kernel >
class Construct_midpoint_2 {
  typedef Kernel::Point_2     Point;
public:
  typedef Point               result_type;

  result_type operator()(Point p, Point q) const {
    return Point( (p.x() + q.x()) / 2,
                  (p.y() + q.y()) / 2 );
  }
};
```

3.2.1 Conversions Between Objects of Different Kernels

As much as conversions between number types are useful, CGAL also provides tools to convert geometric objects between different kernels. We briefly present these below, as they are used later. A kernel converter is a functor whose function operator is overloaded for each object of the source kernel and returns the corresponding object of the target kernel. Such conversions may depend on the details of representation of the geometric objects, such as homogeneous versus Cartesian representation. CGAL provides such converters parameterized by converters between number types. Here is for example the converter between kernels of the `Cartesian` family:

```
template < typename K1, typename K2,
           typename NT_converter = Default_converter<K1::FT, K2::FT> >
class Cartesian_converter {
  NT_converter cv;

public:
  K2::Point_2 operator()(K1::Point_2 p) const {
    return K2::Point_2( cv(p.x()), cv(p.y()) );
  }
  . . .
};
```

CGAL also provides a way to find out the type of a geometric object (say a 3D segment) in a given kernel, given its type in another kernel and this second kernel. This is in practice the return type of the function operator of the kernel converter described above.

```
template < typename O1, typename K1, typename K2 >
struct Type_mapper {
  typedef . . .  type;
};
```

The current implementation works by specializing on all known kernel object types like `K1::Point_2` or `K1::Segment_3`. A more extensible approach could be sought, but this is not the main point of this paper.

Now that we have described all the tools, we can present the heart of our paper, the generic design of the lazy exact computation framework.

4 Design of the Lazy Exact Computation Framework

In this section, we describe the filtering and lazy exact computations techniques. We first devise the filtered predicates, then we propose a lazy exact number-type, which we then generalize to all geometric constructions. We also show how to put things together in geometry kernels.

4.1 Filtered Exact Predicates

Evaluating a filtered predicate means first calling the predicate instantiated with interval arithmetic [14]. If this fails because some interval is not precise enough, the predicate is reevaluated, this time instantiated with an exact number-type. As all predicates of a CGAL kernel are functors, we can use the following `Filtered_predicate` adaptor, which is itself parameterized by four functors: (i) the exact version of a predicate, which is using exact arithmetic such as rational numbers, (ii) the approximate version of the predicate, which is using interval arithmetic, (iii) a functor that converts the input geometric objects to their exact counterpart (say, points with rational coordinates), and

(iv) a functor that converts the input geometric objects to their enclosing interval equivalent.

```
template <typename Exact_predicate,
          typename Approximate_predicate,
          typename Converter_to_exact,
          typename Converter_to_approximate>
class Filtered_predicate {
  Exact_predicate             ex_p;
  Approximate_predicate       ap_p;
  Converter_to_exact          to_ex;
  Converter_to_approximate    to_ap;

public:

  typedef Exact_predicate::result_type  result_type;

  template <typename... Args>
  result_type operator()(const Args & ... args) const {
    try {
      Protect_FPU_rounding P(FE_UPWARD);
      return ap_p(to_ap(args)...);
    } catch (Interval_nt_advanced::unsafe_comparison) {
      Protect_FPU_rounding P(FE_TONEAREST);
      return ex_p(to_ex(args)...);
    }
  }
};
```

The class `Filtered_kernel` is then obtained from a kernel `K` by repeating the adaptation for each predicate of `K`. The geometric objects as well as the constructions remain unchanged.

```
template < typename K >
class Filtered_kernel {
  // Kernels used internally
  typedef Simple_cartesian<Interval_nt>  AK;
  typedef Simple_cartesian<Gmpq>         EK;

  // Kernel converters
  typedef Cartesian_converter<K, AK>     C2A;
  typedef Cartesian_converter<K, EK>     C2E;

public:

  // Geometric objects (unchanged)
  typedef K::Point_2                     Point_2;
  . . .
  // Functors providing (filtered exact) predicates
  typedef Filtered_predicate<EK::Compare_x_2,
                             AK::Compare_x_2,
                             C2E, C2A> Compare_x_2;
  . . .

  // Functors for constructions (unchanged)
  typedef K::Construct_midpoint_2        Construct_midpoint_2;
  . . .
};
```

4.2 Lazy Exact Numbers

In order to extend the filtering technique to exact constructions, we can implement a number type that evaluates computations using interval arithmetic. Moreover, it must remember the way it has been constructed by storing the history of operations as a directed acyclic graph (DAG) [15]. Figure 3 illustrates the history DAG of the expression $\sqrt{x} + \sqrt{y} - \sqrt{x + y + 2\sqrt{xy}}$.

When a comparison is performed on variables of this number type and the intervals overlap, the DAG is used to recompute the values of each node recursively with an exact multi-precision type, hence giving the exact result. CGAL provides such a lazy number-type called `Lazy_exact_nt<NT>` that is parameterized by the exact type used to perform the exact computations when needed (such as a rational number-type). This can be seen as a wrapper on top of its template parameter that delays the computations until they are needed, as they might not be needed at all.

We can compare the performance of this approach against a straightforward use of rational arithmetic. Such a test is described in detail in Section 6. In particular, a comparison of the first two rows of Table 1 indicates that the lazy exact number-type approach gains a factor of 8 to 10 in running time, while it causes a loss of a factor of 7 in memory use caused by the storage of the DAG. We next describe a way to recover from this loss and optimize the running time even further.

4.3 Optimizing by Regrouping Operations

There are mostly two aspects that can be optimized. First, a node of the DAG is created and the corresponding heap-memory allocated for each arithmetic

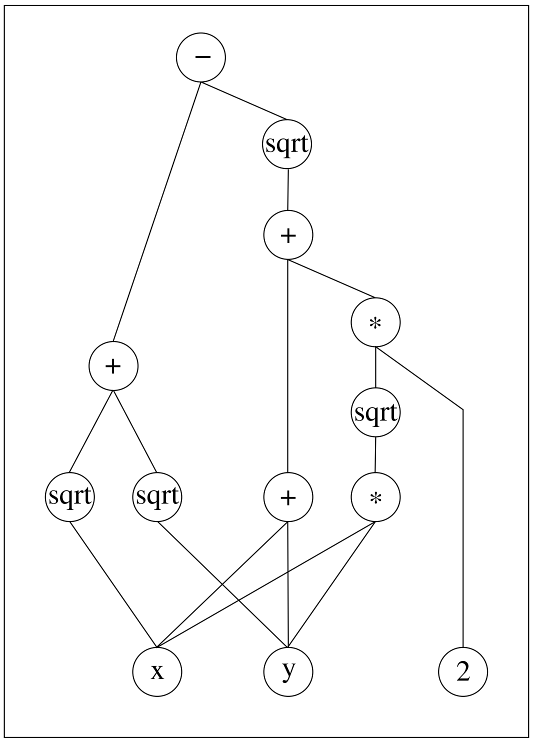

Fig. 3. Example DAG: $\sqrt{x} + \sqrt{y} - \sqrt{x + y + 2\sqrt{xy}}$.

operation. Secondly, as mentioned in Section 2.2.2.1, interval arithmetic benefits from the elimination of redundant rounding mode changes, which is made possible for consecutive operations. In both cases, regrouping operations in more complex functions would help. For example, if we consider the computation of the determinant in the `orientation` predicate, we see that seven operations are needed, but we could just as well use one function of arity six for the whole determinant. This would introduce only one DAG node and allow to perform the optimizations mentioned by amortizing the costs over more arithmetic operations.

These remarks have lead to a new scheme mentioned in [3], and the description of an implementation has also been proposed in [4]. The idea is to introduce a DAG at the geometric level, by considering geometric objects and primitives for the nodes, and not only numbers and arithmetic operations. The next section describes such an optimized setup. Our design differs from the one in [4] in that we followed the generic-programming paradigm and made extensive use of templates, to make it as easily extensible as possible.

4.4 Lazy Exact Geometric Objects

Performing lazy exact constructions means performing constructions with interval approximations, and storing the sequence of construction steps. When later a predicate applied to these approximations cannot return a result that is guaranteed to be correct, the sequence of construction steps is performed again, this time with an exact arithmetic. Now the predicate can be evaluated correctly.

The sequence of construction steps is stored in a DAG. Each node of the DAG stores (i) an approximation (`at`), (ii) a pointer to an exact result (`et`), (iii) a functor able to perform the same construction exactly (`ec`), that is, to construct `*et`, and (iv) reference-counted pointers to the lazy objects that were arguments to the function (`args...`). The out-degree of a node is therefore the arity of the function.

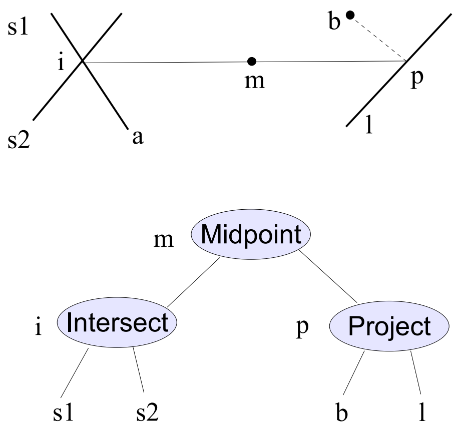

Fig. 4. The DAG represents the midpoint m of an intersection point i and the orthogonal projection p of a point b onto a line l. Testing whether a, m, and b are collinear may trigger an exact construction.

The example shown in Figure 4 illustrates that lazy objects can be of the same type, without being the result of the same sequence of constructions. a, b, i, m, and p are all point-ish. Therefore, we use a handle class template, `Lazy_exact`, storing a reference-counted pointer to a node of the DAG. The types of the nodes of the DAG are organized in a hierarchy using `Construction_base` as abstract base class. The reference-counting mechanism follows a design similar to the one described in [16].

We now explain some of the classes in Figure 5 in more detail.

`Lazy_exact` has `Lazy_exact_nt` as subclass, which provides arithmetic operations. Note that this framework handles arithmetic and geometric objects in a unified way. For example, a distance between geometric objects yields a lazy exact-number, and a lazy exact-number can become the coordinate of a

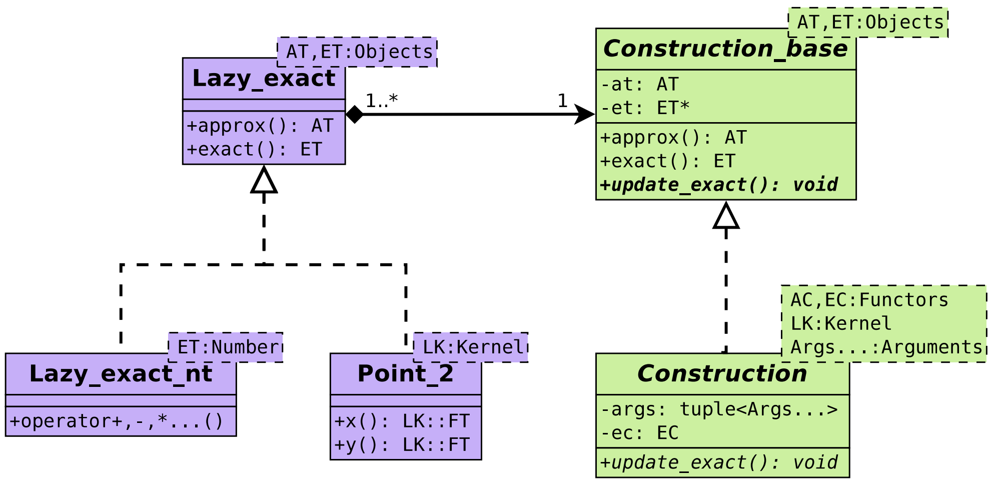

Fig. 5. UML diagram of the classes involved in the nodes of the DAG.

point. Therefore, geometric-object types such as `Point_2` are also subclasses of `Lazy_exact`.

The `Construction_base` abstract base class stores the approximation (`at`), and holds a pointer to the exact value (`*et`). Initially, this pointer is set to `NULL`, and it is the virtual member function `update_exact` that later may compute the exact value and then cache it. Here is an outline of this class:

```
template <typename AT, typename ET>
class Construction_base
  : public Reference_counter {
public:

  AT at;
  ET *et;

  Construction_base(const AT& at_)
    : at(at_), et(NULL) {}

  virtual void update_exact() = 0;
};
```

The subclass `Construction` is used for generic functions. This class template is parameterized by the types of (i) the approximate construction functor (`AC`), (ii) the exact construction functor (`EC`), (iii) the `Lazy_kernel` class (`LK`) described in more detail in Section 4.6, which provides the required conversion functors, and (iv) the arguments of the function (`Args...`). The arguments, which may be of arbitrary types, are stored in the class together with the exact version of the function. In the case of lazy exact geometric-objects or lazy exact numbers, the arguments are handles as described above.

```
template <typename AC, typename EC, typename LK, typename... Args>
```

```
class Construction
  : public Construction_base<AC::result_type, EC::result_type> {
  typedef AC                Approximate_construction;
  typedef EC                Exact_construction;
  typedef LK::C2E           To_exact_converter;
  typedef LK::C2A           To_approximate_converter;
  typedef LK::E2A           Exact_to_approximate_converter;
  typedef AC::result_type  AT;
  typedef EC::result_type  ET;

  tuple<Args...> args;
  EC ec;

public:

  void update_exact() {
    this->et = new ET(bind_tuple(ec, apply(C2E(), args)()));
    this->at = E2A()(*(this->et));
    args = tuple<Args...>(); // Prune dag
  }

  Construction(const AC & ac, const EC & ec_, const Args & ... a)
    : Construction_base<AT, ET>(ac(C2A()(a)...)), args(a...),
      ec(ec_) {}
};
```

The constructor stores the arguments. It then takes their approximations and calls the approximate version of the functor.

In case the exact version of the construction is needed, this gets computed in the `update_exact` method. The exact versions of the arguments are fetched, which in turn may trigger their exact computation if they are not already computed and cached. From the exact object one computes an approximate object directly again, as this may improve the precision of the approximation by avoiding any accumulation of past rounding errors and this can help in further uses of this lazy object.

Finally, the DAG is pruned. As the nodes of the DAG are reference counted, some of them may be deallocated by the pruning. Most often, the types forming `Args` are lazy exact objects. For performance reasons their default constructors generate handles to shared static nodes of the DAG.

The other derived classes store the leaves of the DAG. For efficiency considerations, there is a general purpose leaf class, but also more specialized ones, for example for creating a lazy exact number from an `int`.

So far we have only explained how lazy constructions are stored, but not how new nodes of the DAG are generated. The following functor adaptor maps the basic generic functors encoding the geometric constructions to their corresponding lazy versions.

```
template <typename LK, typename AC, typename EC>
class Lazy_construct {
  typedef LK                    Lazy_kernel;
  typedef AC                    Approximate_construction;
  typedef EC                    Exact_construction;
  typedef LK::AK                AK;
  typedef LK::EK                EK;
  typedef LK::E2A               E2A;
  typedef LK::C2E               C2E;
  typedef AC::result_type       AT;
  typedef EC::result_type       ET;
  typedef Lazy_exact<AT, ET, E2A> Handle;

  AC ac;
  EC ec;

public:

  typedef Type_mapper<AT, AK, LK>::type result_type;

  template <typename... Args>
  result_type operator()(const Args & ... args) const {
    try {
      Protect_FPU_rounding P(FE_UPWARD);
      return result_type(Handle(new Construction<AC, EC, LK, Args...>
                                (ac, ec, args...)));
    } catch (Interval_nt_advanced::unsafe_comparison) {
      Protect_FPU_rounding P(FE_TONEAREST);
      return result_type(Handle(new Construction<AC, EC, LK>
                                (ec(C2E()(args)...))));
    }
  }
};
```

The functor first tries to construct a new node of the DAG. If inside the approximate version of the construction an exception is thrown because of an interval arithmetic comparison failure, the exact version of the construction is called, and only a leaf node for the DAG is created.

4.5.1 *Special Case Handling*

The generic functor adaptor works out of the box for all functors that return geometric objects or numbers.

Functors returning objects that are not made lazy are easily handled exceptions. An example in CGAL is the functor that computes a bounding box with `double` coordinates around geometric objects, whose width is not required to be tight. As the intervals corresponding to the coordinates of the approximate geometric object are already 1-dimensional bounding boxes, we never have to resort to the exact geometric object. The functor adaptor is trivial.

Other functors return a polymorphic object. For example, the intersection of two segments may either be empty, a point or a segment. In order to store such a polymorphic result, CGAL offers a class `Object` capable of storing one object of any type. The problem we have to solve is that the lazy exact functor must not return a lazy exact `Object`, but instead must return an `Object` holding a lazy geometric object. This is solved by iterating over all CGAL kernel types, to try to cast, and if it works to construct the lazy geometric object and put it in an `Object` again. This functor appears as `Lazy_construct_returning_object` in the next code example.

Less trivial cases are functors that pass results of a computation back to reference parameters, or that write into output iterators. They need a special functor as well as special `Construction` classes. It is not hard to write them, but the problem is that they must be dispatched by hand, as we have no means of introspection.

4.6 *Lazy Exact Geometry Kernel*

We are ready to put all pieces together, by defining a new kernel that takes an approximate and an exact kernels as template arguments.

```
template < typename AK, typename EK,
           typename Kernel_converter = Default_converter<AK, EK> >
struct Lazy_kernel {

  // Kernel converters
  typedef Lazy_kernel<AK, EK>          LK;
  typedef Approx_converter<LK, AK>     C2A;
  typedef Exact_converter<LK, EK>      C2E;
  typedef Kernel_converter             E2A;

  // Geometric objects
  typedef Lazy_exact<AK::Point_2,   EK::Point_2>   Point_2;
  typedef Lazy_exact<AK::Segment_2, EK::Segment_2> Segment_2;
  . . .

  // Functors for predicates
  typedef Filtered_predicate<EK::Compare_x_2, AK::Compare_x_2,
                             C2E, C2A> Compare_x_2;
  . . .

  // Functors for constructions
  typedef Lazy_construct<LK, AK::Construct_midpoint_2,
                         EK::Construct_midpoint_2>
          Construct_midpoint_2;
  typedef Lazy_construct_returning_object<LK, AK::Intersection_2,
                                          EK::Intersection_2>
          Intersection_2;
  . . .
};
```

It can then be instantiated, for example, as:

```
typedef Lazy_kernel<Cartesian<Interval_nt>, Cartesian<Gmpq> >
        Lazy_cartesian_kernel;

typedef Lazy_kernel<Homogeneous<Interval_nt>, Homogeneous<Gmpz> >
        Lazy_homogeneous_kernel;
```

In the current implementation, we use the preprocessor to generate the typedefs from a list of types, and template specializations for dispatching the special cases. `Approx_converter` simply fetches the stored approximate object, while `Exact_converter` fetches the exact object, possibly triggering its computation.

5 Extensibility

We have to distinguish between different levels of extensibility.

When CGAL kernels get extended by geometric objects and constructions, changes are needed in the lazy construction framework if the new constructions have "new" interfaces, e.g., two output iterators, followed by two reference parameters to return a result. This would need a new node type for the DAG, a new functor, and hard wired dispatching in the lazy kernel. Otherwise, there is nothing special to do.

When the user wants to extend the lazy kernel with his own geometric objects and constructions, he first has to add them to the kernel that then gets into the lazy computation machinery, as described in [12]. Then, our statement in the previous paragraph applies.

The `Circular_kernel_2` class of CGAL [17,18] is an example for both. It extends the basic CGAL kernels by adding geometric types such as circular arcs, and it provides additional predicates and constructions around them, like circle instersections. A lazy exact variant of it is also provided by applying the generic filtered tools described in this paper.

6 Benchmarks

We present the results of two benchmarks, which show the benefit of our techniques. Firstly, we use a simple artificial algorithm and then a more complex and useful algorithm, the computation of the 3D flow complex. The benchmarks compare the running time and the memory consumption of various kernel choices.

Here is the description of our artificial algorithm:

- generate 2000 pairs of 2D points with random coordinates (using `drand48()`).
- construct 2000 segments out of these points.
- intersect all pairs of segments among these, and store the resulting intersection points.
- shuffle the resulting points
- iterate over consecutive triplets of these points, and compute the `orientation` predicate of these.

Table 1 provides the resulting data for a choice of four different kernels:

- `SC<Gmpq>` stands for the `Simple_cartesian` representation kernel instantiated with `Gmpq`, which is a C++ wrapper around the multi-precision rational number type provided by GMP,
- `SC<Lazy_exact_nt<Gmpq>>` uses the lazy exact evaluation mechanism at the arithmetic level,
- `Lazy_kernel<SC<Gmpq>>` is our approach for performing lazy exact evaluations at the geometric object level,
- `Lazy_kernel<SC<Gmpq>> (2)` is similar to the previous one, but it does not include the additional optimization that consists in eliminating rounding mode changes between consecutive interval computations,
- finally, `SC<double>` is the `Simple_cartesian` representation kernel instantiated with `double`. It is given for reference as it does not provide any guar-

antee of robustness, since it does not use any interval nor multi-precision arithmetic. It only shows what the optimal performance could be.

These benchmarks have been compiled using the Gnu `g++` compiler version 4.1 with the `-O2` optimization option. The code was executed on a Pentium-M laptop clocked at 1.7 GHz, equipped with 1 GB of RAM and 1 MB of cache, running the Fedora Core 3 Linux distribution. Timings are reported in seconds and memory usage in megabytes.

Kernel	Time	Memory
`SC<Gmpq>`	70	70
`SC<Lazy_exact_nt<Gmpq>>`	7.4	501
`Lazy_kernel<SC<Gmpq>> (2)`	3.6	64
`Lazy_kernel<SC<Gmpq>>`	2.8	64
`SC<double>`	0.72	8.3

Table 1
Benchmarks comparing different kernels on an artificial algorithm.

The results show that our approach wins almost a factor of 10 on memory over the basic lazy-evaluation scheme. It is also between 2 and 3 times faster. While it remains 4 times slower than the approximate floating-point evaluation, it is guaranteed for all cases.

The benchmarks also show the gain obtained thanks to the elimination of rounding mode changes, which is now allowed by the regrouping of operations on intervals.

Another data point illustrating the improvements is that we measured the number of Dag nodes allocated. For `SC<Lazy_exact_nt<Gmpq>>`, 29 million nodes were allocated, while for `Lazy_kernel<SC<Gmpq>>` only 2.5 million nodes were needed. So we have won a factor of more than 10, due to the regrouping allowed by our design.

Note that the algorithm uses random data, hence it does not produce many filter failures, so almost no exact evaluation is performed. Another thing worth noticing is that it uses relatively simple 2D primitives. More complex primitives, especially in higher dimensions, should render the method even more advantageous. Finally, real-world geometric applications tend to produce more combinatorial output, hence the relative runtime cost of primitives is smaller, so the slow down factor is lower in those cases.

We now show experiments on the computation of the flow complex, a 3D graph built on top of the 3D Delaunay triangulation, which is used, for example, in a 3D surface reconstruction algorithm. The flow complex algorithm is described in more detail in [19]. The data sets are composed of 12593 and 54811 points, respectively. They represent scanned objects and are referred to as `Mecanic` and `Fish`. The implementation is based on the 3D Delaunay triangulation of CGAL. The latter exclusively uses geometric predicates, and no construction, whereas the flow complex computation itself uses cascaded constructions that demonstrate our point (repeated projections of points on the segments representing the Voronoi edges). This benchmark has been run in the same conditions as the previous one, except that the system had 2GB of memory, a 3GHz processor and was running Linux Fedora Core 5. Table 2 illustrates a gain of a factor of 4 in memory, and about 30% in time. Note that the reference kernel `SC<double>` lead to crashes of the algorithm on both data sets due to numerical issues.

	Mecanic		Fish	
Kernel	Time	Memory	Time	Memory
`SC<Lazy_exact_nt<Gmpq>>`	13	413	32	1856
`Lazy_kernel<SC<Gmpq>>`	9.4	108	24	395

Table 2
Benchmarks comparing different kernels for the construction of the flow complex.

7 Open Design Questions

Here is a list of open questions related to our framework.

The first question concerns the regrouping of operations as described in Section 4.3. Our framework asks the user to pass functors specifying the level at which the regrouping of operations is made. In CGAL this is not a problem since the primary interface of the kernel towards the geometric algorithms is a list of functors. However it has the drawback of not being automatic. We can think of approaches based on expression templates [20] that would automatically detect sequences of operations and regroup them. Unfortunately, expression templates are limited to single statements and they tend to increase compilation times considerably. Could there be a way to extend the automatic regrouping to more than single statements? Maybe the `auto` keyword of C++0x will allow to propagate this through several statements? Or maybe the `axiom` feature of the `concept` extension of C++ [21] could be used to specify this kind of transformation.

Another question is whether similarly delayed computations are used in other areas, and if yes, whether it is possible to find out a common design, more general than the one we propose.

8 Conclusion and Future Work

In this paper we have presented a generic framework that implements lazy exact geometric computations, motivated by the needs for robustness and efficiency of geometric algorithms. This framework enables the delay of costly exact evaluations using multi-precision arithmetic when the faster interval arithmetic suffices.

The proposed design is easily extensible to new geometric primitives (predicates and constructions), as well as new geometric objects. It is based on a template family for representing lazy objects, as well as generic functor adaptors that produce them.

Our implementation is now part of CGAL public releases, and its entire geometry kernel already benefits from it.

Future work in this area will consist of various added special-case optimizations as well as generalizations. It is for example possible to refine the filtering scheme by growing the precision incrementally instead of switching directly to full multi-precision computation in case of insufficiency of precision of the intervals. We would also like to study the possibilities of merging the `Filtered_predicate` and `Lazy_construct` functor adaptors. Possible optimizations for specific cases also can be done, using faster schemes than interval arithmetic (so-called static filters). Moreover, the current way of providing a full kernel is by a list of types for the objects and functors, which is generated using of the preprocessor. We will therefore try to provide a better design on this particular point.

The authors wish to thank the anonymous reviewers for their careful work.

A Benchmark Code

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Lazy_kernel.h>
#include <CGAL/Gmpq.h>
#include <CGAL/Lazy_exact_nt.h>
#include <CGAL/intersections.h>
#include <CGAL/Timer.h>
#include <CGAL/Memory_sizer.h>
using namespace CGAL;

// Choose a kernel:
```

```
//typedef Simple_cartesian<Gmpq>                K;
//typedef Simple_cartesian<Lazy_exact_nt<Gmpq> > K;
//typedef Lazy_kernel<Simple_cartesian<Gmpq> >   K;
typedef Simple_cartesian<double>               K;

typedef K::Point_2    Point;
typedef K::Segment_2  Segment;

Point   random_point()
{ return Point(drand48(), drand48()); }

Segment random_segment()
{ return Segment(random_point(), random_point()); }

int main() {
  int loops = 2000, init_mem = Memory_sizer().virtual_size();
  Timer t; t.start();

  std::cout << "Generating initial random segments: "
            << loops << std::endl;
  std::vector<Segment> segments;
  for (int i = 0; i < loops; ++i)
    segments.push_back(random_segment());

  std::cout << "Counting intersections [brute force algorithm]: "
            << std::flush;
  std::vector<Point> points;
  for (int i = 0; i < loops-1; ++i)
    for (int j = i+1; j < loops; ++j) {
      Object obj = intersection(segments[i], segments[j]);
      if (const Point* pt = object_cast<Point>(&obj))
        points.push_back(*pt);
    }
  std::cout << points.size() << std::endl;

  // Shuffle the points, as consecutive points have good chance
  // to come from the same segments, hence filter failures in
  // orientation() later...
  std::random_shuffle(points.begin(), points.end());

  std::cout << "Performing orientation tests" << std::endl;
  int negative_ort = 0, positive_ort = 0, collinear_ort = 0;
  for (int i=0; i < points.size()-2; ++i) {
    Orientation o = orientation(points[i], points[i+1], points[i+2]);
    if (o < 0)      ++negative_ort;
    else if (o > 0) ++positive_ort;
    else            ++collinear_ort;
  }
  std::cout << "orientation results : (-) = " << negative_ort
                                   << "    (+) = " << positive_ort
                                   << "    (0) = " << collinear_ort
                                   << std::endl;

  t.stop();
  std::cout << "Total time   = " << t.time() << std::endl;
  std::cout << "Total memory = "
            << ((Memory_sizer().virtual_size() - init_mem) >>10)
            << " KB" << std::endl;
}
```

References

[1] CGAL User and Reference Manual, 3.5 Edition (2009).
URL
`http://www.cgal.org/Manual/3.5/doc_html/cgal_manual/packages.html`

[2] C. Yap, Robust geometric computation, in: J. E. Goodman, J. O'Rourke (Eds.), Handbook of Discrete and Computational Geometry, 2nd Edition, CRC Press LLC, Boca Raton, FL, 2004, Ch. 41.

[3] S. Pion, De la géométrie algorithmique au calcul géométrique, Thèse de doctorat en sciences, Université de Nice-Sophia Antipolis, France (1999).
URL `http://tel.archives-ouvertes.fr/tel-00011258/`

[4] S. Funke, K. Mehlhorn, Look – a lazy object-oriented kernel for geometric computation, Computational Geometry - Theory and Applications (CGTA) 22 (2002) 99–118.

[5] D. Gregor, J. Järvi, Variadic templates for C++, in: SAC '07: Proceedings of the 2007 ACM symposium on Applied computing, ACM, New York, NY, USA, 2007, pp. 1101–1108.

[6] D. Goldberg, What every computer scientist should know about floating-point arithmetic, ACM Computing Surveys 23 (1) (1991) 5–48.

[7] R. E. Moore, Interval Analysis, Prentice Hall, Englewood Cliffs, NJ, 1966.

[8] GMP, The GNU Multiple Precision Arithmetic Library.
URL `http://gmplib.org/`

[9] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, C. Yap, Classroom examples of robustness problems in geometric computations, Computational Geometry: Theory and Applications 40 (1) (2008) 61–78.
URL `http://hal.inria.fr/inria-00344310/`

[10] V. Karamcheti, C. Li, I. Pechtchanski, C. Yap, The CORE Library Project, 1st Edition (1999).
URL `http://www.cs.nyu.edu/exact/core/`

[11] C. Burnikel, K. Mehlhorn, S. Schirra, The LEDA class real number, Technical Report MPI-I-96-1-001, Max Planck Institut für Informatik, Saarbrücken, Germany (Jan. 1996).
URL `http://data.mpi-sb.mpg.de/internet/reports.nsf/NumberView/1996-1-001`

[12] S. Hert, M. Hoffmann, L. Kettner, S. Pion, M. Seel, An adaptable and extensible geometry kernel, Computational Geometry: Theory and Applications 38 (2007) 16–36.
URL `http://hal.inria.fr/inria-00344363/`

[13] H. Brönnimann, A. Fabri, G.-J. Giezeman, S. Hert, M. Hoffmann, L. Kettner, S. Schirra, S. Pion, 2D and 3D geometry kernel, in: CGAL Editorial Board (Ed.), CGAL User and Reference Manual, 3.5 Edition, 2009.
URL `http://www.cgal.org/Manual/3.5/doc_html/cgal_manual/packages.html#Pkg:Kernel23`

[14] H. Brönnimann, C. Burnikel, S. Pion, Interval arithmetic yields efficient dynamic filters for computational geometry, Discrete Applied Mathematics 109 (2001) 25–47.

[15] M. Benouamer, P. Jaillon, D. Michelucci, J.-M. Moreau, A lazy solution to imprecision in computational geometry, in: Proc. 5th Canad. Conf. Comput. Geom., 1993, pp. 73–78.

[16] L. Kettner, Reference counting in library design – optionally and with union-find optimization, in: Workshop on Library Centric Software Design, 2005.

[17] P. M. M. de Castro, S. Pion, M. Teillaud, 2D circular geometry kernel, in: CGAL Editorial Board (Ed.), CGAL User and Reference Manual, 3.5 Edition, 2009.
URL `http://www.cgal.org/Manual/3.5/doc_html/cgal_manual/packages.html#Pkg:CircularKernel2`

[18] I. Z. Emiris, A. Kakargias, S. Pion, M. Teillaud, E. P. Tsigaridas, Towards an open curved kernel, in: Proc. 20th Annu. Sympos. Comput. Geom., 2004, pp. 438–446.

[19] F. Cazals, A. Parameswaran, S. Pion, Robust construction of the three-dimensional flow complex, in: Proceedings of the 24th Annual Symposium Computational Geometry, 2008, pp. 182–191.
URL `http://hal.inria.fr/inria-00344962/`

[20] T. L. Veldhuizen, Expression templates, C++ Report 7 (5) (1995) 26–31, reprinted in C++ Gems, ed. Stanley Lippman.

[21] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. Dos Reis, A. Lumsdaine, Concepts: linguistic support for generic programming in C++, in: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, ACM, New York, NY, USA, 2006, pp. 291–310.