About Arithmetics and Kernels

Sylvain Pion

Abstract

Non-robustness problems are a well known issue when it comes to implementing computational geometry algorithms. In this talk we will present an overview of the general approach followed by CGAL to address them. We will then introduce the tools provided by CGAL to implement it. Finally, We will try to list some ideas that could be followed in the future to improve the situation.

1 Introduction

Most geometric algorithms are a mix of numerical and combinatorial computations. This specific nature is usually considered as the root of the non-robustness problems that arise when implementing them since the correctness of the combinatorial part is usually guaranteed by geometric theorems which are not verified by approximate numerical computations.

There are several solutions in the litterature to solve the non-robustness issues. On one side there are approaches which target specific algorithms to make them tolerant to most kinds of input, by making the algorithms not crash, even though they compute a non mathematically exact solution.

Given the number of algorithms in CGAL, it would have been difficult to address the issues one algorithm at a time that way. Hence we have considered the general approach known as the *Exact Geometric Computation* paradigm (EGC). At the beginning of the development of CGAL, it has been decided to parametrize all algorithms by a *traits class*, specifying the types of the geometric objects and the basic geometric primitives acting on them (predicates and constructions). This design decision allowed to decouple the implementation of the algorithms from the implementation of traits classes, which are encapsulating the non-robustness solutions in the EGC paradigm.

Traits classes also tend to contain functionalities which are re-usable between different algorithms, hence the union of their functionalities has been gathered in *kernels*.

CGAL had from the beginning a few families of kernels providing solutions to the non-robustness problems : `Cartesian<FT>` and `Homogeneous<RT>`. These allowed to support both fast/approximate computations, e.g. using `Cartesian<double>`, and exact/slower computations when plugging-in an exact number type, e.g. `Cartesian<Quotient<Gmpz> >`.

After that came the optimization phase, where users needed both robustness and efficiency. So we will describe next the various steps which are needed to implement EGC efficiently. EGC is not only about exact arithmetic operations needed to implement geometric primitives, since an important remark is that providing exact geometric primitives is enough to ensure that the algorithms are robust.

2 Arithmetic tools

2.1 Multiprecision

The basic tool to implement EGC is exact arithmetic. CGAL is interfaced to external libraries that provide exact or guaranteed multiprecision computations:

- `GMP` which provides multiprecision integers and rationals (we do not use the multiprecision floats of `MPFR` at the moment).
- `LEDA` which additionaly provides `real`, i.e. numbers incrementally constructed with the 4 basic operations plus the k-th root. The sign of such a number can be computed exactly, which guarantees exact implementation of predicates.
- `CORE` which is similar to `LEDA::real` but only supports the square root (not general k-th root), but supports a `rootOf` operator able to extract roots of a polynomial with integer coefficients.

CGAL also provides the `MP_Float` class which can compute exact polynomial expressions with double coefficients. Together with the generic `Quotient` class, it also provides exact rational computations.

Multiprecision computation is costly by nature, so in order to amortize its cost, we use arithmetic filters.

2.2 Filters

Efficient implementation of EGC also relies on filters : a way to compute with multiprecision only on demand, that is, when approximate, but certified, computation is not precise enough to guarantee the exact geometric primitives. CGAL provides interval arithmetic through the `Interval_nt` class, which is a basic tool to implement dynamic filters.

In order to make it easier to use, this filtering scheme is encapsulated in `Filtered_kernel<K>`, which is a kernel wrapper around `K` whose predicates are replaced by filtered exact versions. Each predicate of the kernel is thus wrapped using the generic functor adaptor `Filtered_predicate`. This scheme also allows to apply the filtering techniques to predicates of traits classes which are not in the kernel, or to user code.

Other filtering schemes are available, although not as general : `Filtered_exact` is the ancestor of `Filtered_kernel` but it provides a number type interface, `Fixed_precision_nt` provides static filters for the predicates

used by 2D and 3D Delaunay triangulations, `Static_filters` provides a similar functionality but is less constrained on the input data.

2.3 Constructions

CGAL also provides a tool to compute geometric constructions in a lazy manner, following the same filtering principles : `Lazy_exact_nt` is a number type storing a DAG of the computation, with an interval approximation. The DAG is used to re-evaluate the value using an exact number type when the approximation is not precise enough.

3 Kernel organization

The kernel gathers dozens of geometric primitives and types. Providing variants of all of these using different schemes would be a maintainance nightmare if done by hand and copy-paste. Therefore we are using generic programming techniques as much as possible.

The kernel provides 2 interfaces to the user : global functions like `CGAL::orientation(p, q, r)` which are convenient to use separately, and corresponding functors like `K::Orientation_2` which are more easily usable by generic algorithms, especially those of the STL.

Recent changes in the kernel are aiming at making the first interface (global functions) call the second (the functors)[1]. That way, it is easy to provide a new kernel only by its functors, and get the global function interface for free. This is what `Filtered_kernel` is doing for example. The advantage over the reverse solution is that it is easier to process all functors automatically using functor adaptors: `Filtered_predicate` mentionned previously, but also other tools like `Kernel_checker` which allows to run 2 kernels in parallel for debugging purposes, or we could think of a projection kernel (2D geometry from 3D points as if projected on one plane)...

4 Conclusion and Future work

Finally, here is a list of things which would be nice to have in future CGAL releases:

- Merge the functionality of `Static_filters` into `Filtered_kernel`.
- Improve the static filters so that they can be used with inputs which are not only `double`s.
- Provide a kernel with filtered constructions (similar to `Lazy_exact_nt` but with one node of the DAG per geometric construction, not per arithmetic operation).

[1] This is hopefully be completed for the next release

- Better support for filtered constructions in the kernel by (re-)introducing constructive predicates, which allow to store some intermediate computations (the difficult part is to nicely handle the filtering schemes which only deal with predicates).
- In the longer term, it would be nice to have a way to generate static filters for the whole kernel automatically. This requires static code analysis tools.