Nr.: FIN-003-2010

Template Little Numbers: A Toolkit for More Userfriendly Effcient Exact Geometric Computation

Jan Tusch

Arbeitsgruppe Algorithmische Geometrie

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Technical report

Nr.: FIN-003-2010

Template Little Numbers: A Toolkit for More Userfriendly Effcient Exact Geometric Computation

Jan Tusch

Arbeitsgruppe Algorithmische Geometrie

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG)

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Jan Tusch
Postfach 4120
39016 Magdeburg
E-Mail: tusch@isg.cs.uni-magdeburg.de

http://www.cs.uni-magdeburg.de/Technical_reports.html
Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 15.02.2010

Bezug: Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

Template Little Numbers: A Toolkit for More Userfriendly Efficient Exact Geometric Computation[*]

Jan Tusch

September 24, 2009

Abstract

Template Little Numbers (`TLN`) is a prototypical `C++` toolkit to support exact geometric computation, similar to Fortune and van Wyk's `LN` [4] toolkit. However, `TLN` uses expression template techniques and thus, in contrast to `LN`, does not require separate preprocessing of the `C++` source code. We present design and implementation of `TLN` and compare the efficiency of implementations of geometric predicates derived from `TLN` with other approaches.

1 Introduction

Geometric algorithms are usually designed in a theoretical model of computation such as the Real-RAM [9] which assumes that all computations with real numbers are exact. Algorithms implemented by directly replacing the arithmetic of the model with floating-point arithmetic may not yield correct results. They even may crash or loop [18, 10]. Exact geometric computing (EGC) is one way to avoid problems caused by this discrepancy between a theoretical model of computation in which the algorithms are proven to be correct, and the physical platforms on which the algorithms are implemented on the other hand. The idea is to use extended precision arithmetic such that the algorithm behaves as if all computations were exact. By using EGC we make sure that all decisions done in the control flow of an algorithm are correct.

A straightforward way to implement EGC is to use software packages providing multiprecision arithmetic. Such libraries provide number types, that can store values of arbitrary precision (ignoring the limitation of a limited amount of memory etc.). Thus, if we replace the built-in bounded precision hardware number types by such software number types the programmer can more or less directly translate the description in the theoretical model of computation to a physical implementation of a geometric algorithm.

Many algorithms in computational geometry evaluate the relative position of two or more geometric objects, such as points, lines and circles etc. Depending on the outcome of testing for a particular configuration of these objects different branches of execution are pursued. Many tests in geometric computation can be formulated as a predicate that returns the sign of a polynomial in the coordinates of the involved geometric objects.

When implementing an algorithm to compute the sign of such polynomials, the programmer is facing the following dilemma: machine arithmetic using hardware supported number types is fast but error-prone where software multiprecision computations are exact but slow. Fortunately, many platforms implement floating-point arithmetic conforming to the IEEE 754 standard, see [5] and the references cited therein. Here, each basic operation, such as $+, -, *$ introduces a bounded relative

[*]Partially supported by DFG grant SCHI 858/1-1

error of the computed result. Hence, if the absolute value of the result computed using floating-point arithmetic is large enough we can trust the sign of the result.

In this way we can avoid the expensive multiprecision computation by checking against an error bound. This technique is called *numerical filtering* in general, or *floating-point filter* when floating-point arithmetic is used. If the filter fails to certify the correctness of the sign we resort to some sort of exact computation. A floating-point filter is called *static* if it does not depend on the values of the variables of the polynomial. Since, the error made in each machine operation is a relative error this requires an upper bound on the absolute value of the variables. Then the error-bound only depends on the sequence of operations used in the computation. On the contrary, a *dynamic* filter uses the values of the variable together with the floating-point approximation to compute the error bound. A hybrid approach using both methods is called *semi-static* filter.

To apply these concepts in practice, tools [4, 2] have been developed that transform critical sections of the source code into code providing correct decisions using floating-filters and multiprecision arithmetic for the `C++` language. These tools are preprocessors searching the source code for special statements that the programmer uses to annotate the critical sections of the code. We show that reliable geometric predicates including numerical filtering and exact arithmetic can be realized user-friendly without external tools by using the `C++` Expression Templates of `TLN`.

`TLN` is a prototypic implementation based on the concepts described in [4]. It provides a user-friendly way to reliably determine the sign of polynomial expressions in `C++`. The expressions can contain the usual $+, -, *$ operations and variables. The type of a variable is the template class `integer<L>`, where L is a constant that bounds the bitlength of the variable of this type. In other words, the absolute value of any variable of type `integer<L>` is less than 2^L. Thus, all variables must be integers for which the maximum absolute values must be known at compile time. These maximum values are used in a static filter which is more efficacious when the actual input values are close to this upper bound.

The programmer uses `integer<L>` as if it were a multiprecision number type. Polynomial expressions over variables of this type should be passed directly to the `sign(...)` function. This is due to a limitation in the `C++` language, that does not allow for automatic type deduction of the result type in assignments.

The rest of the paper is organized as follows: After reviewing similar tools in section 2, we give a brief introduction to `C++` Expression Templates. We describe many details of the `TLN` implementation in section 4. In section 5 we present experimental results demonstrating the efficiency of `TLN` compared to off-the-shelf multiprecision number types. We conclude with a discussion and mention perspectives on how to extend and improve our prototype implementation in section 6.

2 Related Work

Fortune and van Wyk where among the first to demonstrate the effectiveness of numeric filters when they developed `LN` – a preprocessor that generates code for geometric integer predicates and `C++`-classes representing constructed geometric objects, e.g. intersection points [4]. Using `LN` requires to learn a macro-language that the preprocessor can understand. `TLN`, however, solely uses `C++` language features to trigger the generation of exact evaluation and filter code. Clearly, `C++` expression templates cannot compete with the algorithmic power offered by a preprocessor, but provide an easy to use user-interface.

`EXPCOMP` – another preprocessor in the spirit of `LN` was developed by Burnikel, Funke and Seel [2]. Again the user has to annotate the source code to obtain exact results. `EXPCOMP` can generate static and semi-static filters for integer and floating-point expressions.

Another interesting code generator for exact predicates was developed by Nanevski, Blelloch and Harper [8]. They describe how to generalize the approach of Shewchuk [11] to evaluate the polynomial expression adaptively. That is, the floating-point approximation is refined in several stages by reusing the results from the previous stage until the sign can be certified.

A comparison of our method with the results produced by the above tools is not possible, since to our knowledge, none of the above code generators is publicly available.

3 C++ Expression Templates

C++ templates were born in the beginning of the 90s – primarily for the purpose of generic programming. Further development led, among others to the Standard Template Library emerging from the evolving capabilities of C++ templates, see [13, 12, 14] for the early history of the C++ language. In the mid 90s, Todd Veldhuizen and David Vandevoorde independently discovered the potential of using template techniques to speed up computations compared to conventional ways to code expression evaluation in the C++ language [17, 15, 16].

A celebrated feature of C++ is operator overloading. Operator overloading is useful because the developer can write programs using notation closer to the target domain, e.g. algebra. In this way custom types look like types built into the language. For example, a multiprecision C++ libraries overloads the operators corresponding to the supported operation on its number types to increase readability of the code. However, this user-friendliness comes at the price of additional overhead during execution of such code. For example consider an extended precision data type storing an integer with two times the precision of a `double` variable. We store the high and low bits of the number in two separate `double` variables:

```
struct DoubleDouble {
  ...
  double high, low;
  ...
};
```

To make the type user-friendly one would overload the usual operators on such number type:

```
DoubleDouble operator+ (const DoubleDouble& a,
                        const DoubleDouble& b)
{
   DoubleDouble result;
   result.high = a.high + b.high;
   result.low  = a.low  + b.low;
   return result;
}
```

Now, suppose given three numbers `a,b` and `c` of type `DoubleDouble`. For the statement

```
DoubleDouble d = (a + b) + c;
```

the compiler generates the following sequence of operations

1. Evaluate `a+b` and store the result in a temporary variable, say `tmp` of type `DoubleDouble`
2. Evaluate `tmp+c` and store the result in a temporary variable of type `DoubleDouble`
3. Call the assignment operator of `DoubleDouble` (straightforward, omitted here) for `d` and this temporary variable

Thus, to add six numbers we have allocated two temporary objects which can be a dramatic slowdown if the objects are larger. The idea to avoid this overhead is *lazy evaluation*.

The problem is that the assignment operator does not know how the object on the right of the `=` sign (RHS) is created. The solution is to evaluate RHS within the assignment using Expression

Templates. To achieve lazy evaluation we record the construction of RHS in a binary expression type tree. For a binary operation we have a template class

```
template <class L, class R, class Op>
struct BinXpr {
  const L& l;
  const R& r;
  inline BinXpr(const L& ll, const R& rr): l(ll), r(rr) {}
};
```

and a class representing the operation +:

```
template struct plus {
  static inline
  double apply(double l, double r) {
    return l + r;
  }
};
```

The +-operator is overloaded to return the corresponding `BinXpr<...>` type object using partial template specialization. We omit the details here. As a consequence the assignment in our example is now instantiated with a RHS of type

```
BinXpr< BinXpr< DoubleDouble,
                DoubleDouble> , plus >,
        DoubleDouble , plus >
```

What is left is to implement the assignment by evaluating the `high` and `low` part of the result of the expression:

```
template <class Expression>
DoubleDouble& DoubleDouble::operator= (const Expression& e){
  high = evalHigh(e);
  low = evalLow(e);
}
```

The actual evaluation is implemented recursively using the function template

```
template <class L, class R, class Op>
static inline double evalHigh(const BinXpr<L,R,Op>& e){
  return Op::apply( e.l, e.r );
}
```

for binary expressions. Recursion stops when a leaf of the expression type tree is reached. Therefore, we give a specialization for the type `DoubleDouble`:

```
template <>
static inline double evalHigh(const DoubleDouble& d){
  return d.high;
}
```

Similarly `evalLow` is implemented. The careful reader may insist that instead of creating two temporary objects the compiler creates a tree of nested types: So what is the benefit of doing all this? Most likely the compiler will inline all the function calls including constructors and operators because they are very short and simple. When a function is inlined, the compiler, instead of copying the arguments on the execution stack and making a function call, virtually pastes the code of the function body at the place of the call. Thus, the resulting code will be broken down to something like

```
high = (a.high + b.high ) + c.high;
low  = (a.low  + b.low  ) + c.low;
```

Clearly, the benefit we obtain depends on the code optimization algorithms implemented in the compiler, i.e., the strategy on when to inline function calls. Experiments have shown, that vector and matrix expressions evaluated with C++ Expression Templates can be evaluated efficiently [17] as long as the objects are not too large, e.g., when the maximum inlining depth of the compiler is reached.

4 The TLN Library

The design of TLN was mainly pointed to two objectives: user-friendliness and performance. However, sometimes these goals are competing in the sense that, an increase in performance can only be achieved by providing additional information.

The user interface consists of two main parts:

- function and number types used for sign computation
- selection of filter and evaluation algorithms

The latter is implemented by a so-called policy class that can be passed to the sign evaluation function. The policy class contains type definitions of classes responsible for the numerical filtering and exact evaluation of the code. By providing her own policy class, the user can thereby exchange parts of the sign evaluation procedure. For example, the user in this way can deactivate the filter by changing the filter type of the policy class. Currently, only the exact evaluation algorithm described below is available. We provide reasonable defaults to satisfy the programmer who does not want to care about the details of the implementation, as long as the implementation is reliable and sufficiently efficient. The function and number types used for sign computation are described in the following.

The simplest polynomial expressions are variables and constants, each having a maximum bitlength that is known at compile time. These variables are instances of the template

```
  template <unsigned int BIT_LEN >
  struct integer {
    ...
    typedef double NT;
    NT value;
    ....
  };
```

where BIT_LEN is an upper bound on the number of bits of the value of the variable.

The computation of the sign of a polynomial over such variables is triggered by the function template tln::sign(...). The actual sign computation is performed by the Filter class defined in the provided Policy class.

```
template <typename Exp, typename Policy>
  TLN_INLINE int sign(const Exp& e, const Policy&){

    typedef typename expression_traits<Exp>::type Expression;
    typedef typename Policy::Filter               Filter;

    return Filter::sign(e,policy);
  }
```

If only the polynomial expression to be evaluated is given, we use a default policy to compute the sign:

```
template <typename Exp>
TLN_INLINE int sign(const Exp& e){
  ...
  typedef typename expression_traits<Exp>::category Expr_Category;
  typedef tln_default_eval_policy<Expr_Category> Policy;

  return sign(e,Policy());
}
```

Finally, we give an example of a common predicate in 2-dimensional computational geometry. Given three points p, q and r with integer coordinates of bitlength at most 50, we determine on which side of directed line from p to q the point r is located:

```
  const tln::integer<50> px = p.x();
  const tln::integer<50> py = p.y();
  const tln::integer<50> qx = q.x();
  const tln::integer<50> qy = q.y();
  const tln::integer<50> rx = r.x();
  const tln::integer<50> ry = r.y();

  int d = tln::sign( (qx-px)*(ry-py) - (rx-px)*(qy-py) );

  std::cout << "r is ";
  switch(d){
    case  1: std::cout << "right" << std::endl;
    case  0: std::cout << "on"   << std::endl;
    case -1: std::cout << "left"    << std::endl;
  };
```

The above example will be used in the following to illustrate most of the implementation details.

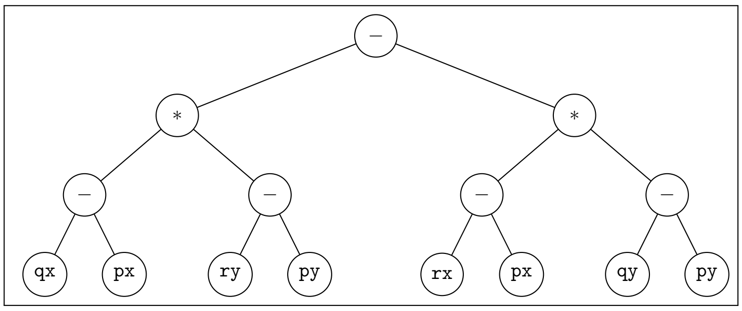
4.1 Implementation Details

The exact sign computation is executed on expression trees:

- Each variable is an expression.
- If T_a and T_b are expressions, then $T_a \circ T_b$ with $\circ \in \{ +, -, * \}$ and $-T_a$ are expressions.

The leaves of an expression tree are thus variables and the internal nodes are operations. For our example we have the expression tree shown at the top of Figure 1. To obtain such an expression tree at compile time we provide `C++`-operators to construct the tree in a bottom-up fashion. According to the definition of an expression we have two types of operations: (1) binary operations and (2) unary operations. A binary operation `a` $\circ$ `b`, where `a` and `b` are of type `A` and `B`, respectively, is represented by the template class as already described in section 3

```
template <typename A, typename B, typename Op>
 struct BinXpr {
   typedef BinXpr<A,B,Op> type;
   typedef Op             op_type;
   typedef A              a_type;
```



```
tln::BinXpr<
  tln::BinXpr<
    tln::BinXpr< tln::integer<50>, tln::integer<50>, tln::diff >,
    tln::BinXpr< tln::integer<50>, tln::integer<50>, tln::diff >,
    tln::prod >,
  tln::BinXpr<
    tln::BinXpr< tln::integer<50>, tln::integer<50>, tln::diff >,
    tln::BinXpr< tln::integer<50>, tln::integer<50>, tln::diff >,
    tln::prod >,
  tln::diff >
```

Figure 1: Expression tree (top) for the 2-dimensional orientation test and the corresponding C++ template type (bottom) representing this tree.

```
    typedef B              b_type;

    TLN_INLINE
    BinXpr(const a_type& a_, const b_type& b_): a(a_), b(b_){}

    const a_type& a;
    const b_type& b;
  };
```

Note that we store only `const` references to the subexpressions since the compiler may thus avoid the actual instantiation of an object of the constructed tree. In this way, each operation is represented by a C++-type. For addition we implemented

```
struct plus {
  template <typename T>
  TLN_INLINE static T apply(const T& a, const T& b) {
    return a + b;
  }
};
```

The static member function `apply` is used to compute the result of the operation using the built-in type `T`. Its result will be used in the static filter.

To construct an expression tree we overload the corresponding C++-operators. For example, the type of the sum of two expressions is obtained by:

```
template <typename A, typename B>
TLN_INLINE BinXpr< typename tln::expression_traits<A>::type,
                   typename tln::expression_traits<B>::type,plus >
```

```
operator+ (const A& a, const B& b)
{
   return BinXpr< typename tln::expression_traits<A>::type,
                  typename tln::expression_traits<B>::type,
                  plus > (a,b);
}
```

The class template `expression_traits` is used to make some type checking. We thereby make sure that we use the operator for valid subexpressions only. Usually `expression_traits<A>::type` has type `A`. Similarly we provide a class template `UnaryXpr` for unary expression and the classes `diff`, `prod` and `negate` together with their corresponding `C++`-operators for subtraction, multiplication and negation, respectively.

For the example predicate , the compiler instantiates the `sign` function with an argument of the type given at the bottom of Figure 1.

We will now see how we can compute a static error bound used in the floating-point filter for such expressions.

4.1.1 Computing Approximation and Error Bounds

An approximation for a given expression is computed using the `C++` built-in double precision floating-point type `double`. Remember, that we want to avoid the creation of an object of the given expression type. Hence, instead of writing a member function of the `BinXpr` class, we implemented a static function template `approximateResult(...)`, for which we give specializations depending on the structure of the expression. The leaves of the expression tree store the exact value of the variable in a `double`. The approximation is obtained by using this value, which is in fact exact. For internal nodes (operations) of the expression tree we recursively compute the approximate result for the subtrees and apply the operator to the result(s). Finally, we obtain the approximate result of the expression as if it would have been evaluated using `double`s.

```
template <typename Expr> TLN_INLINE
static double approximateResult(const Expr& e);

template <unsigned int BIT_LEN> TLN_INLINE
static double approximateResult(const integer<BIT_LEN>& e){
  return e.value;
}

template <typename A, typename B, typename Op > TLN_INLINE
static double approximateResult(const BinXpr<A,B,Op>& e){
  return Op::apply(approximateResult(e.a), approximateResult(e.b));
}

template <typename E, typename Op > TLN_INLINE
static double approximateResult(const UnaryXpr<E,Op>& e){
  return Op::apply(approximateResult(e.e));
}
```

Beside the approximate value of an expression E we obtain an upper bound `MaxError` on the error made during its computation. `MaxError` however depends on the operation and the maximum bitlength `MaxBitlen` of the operands. The required quantities are defined recursively as given in Table 1.

To let the compiler compute `MaxError` for a given expression type, we use a constant propagation mechanism together with in-class static member initialization. We demonstrate the application of

E	`MaxBitlen`(E)
$-F$	`MaxBitlen`(F)
`integer<N>`	N
$F \pm G$	$1 + \max\{$ `MaxBitlen`(F) , `MaxBitlen`(G) $\}$
$F * G$	`MaxBitlen`$(F) +$ `MaxBitlen`(G)

E	`MaxError`(E)
$-F$	`MaxError`(F)
`integer<N>`	0 if $N \leq 53$
$F \pm G$	`MaxError`(F) + `MaxError`(G) + $2^{\texttt{MaxBitlen}(F \pm G)-53}$
$F * G$	$2^{\texttt{MaxBitlen}(G)}$`MaxError`$(F)$ + $2^{\texttt{MaxBitlen}(F)}$`MaxError`$(G)$ $+2^{\texttt{MaxBitlen}(F*G)-53}$

Table 1: Rules for the computation of `MaxBitlen` and `MaxError` for the various expression types.

this technique to the computation of a static constant `double` value of 2^n for $0 \leq n \leq n_{\texttt{max}}$, where $n_{\texttt{max}}$ is the maximum exponent of a double precision floating-point number. We start with the primary template, which recursively computes the necessary powers of two:

```
template <unsigned int N> struct P2 {
  static const double VALUE;
}
template <unsigned int N>
const double P2<N>::VALUE = P2<N/2>::VALUE * P2<N-N/2>::VALUE;
```

The base cases are covered by the following specializations:

```
template <> const double P2<0>::VALUE = 1.;

template <> const double P2<1>::VALUE = 2.;
```

We can easily apply this idea to the computation of `MaxBitlen` and `MaxError`. For example we have the following specializations for the `sum` operation:

```
template <typename A, typename B>
const int MaxBitlen< BinXpr<A,B,sum> >::VALUE =
TLN_MAX( MaxBitlen<A>::VALUE + 1, MaxBitlen<B>::VALUE + 1 );
```

similarly we implement `MaxError`:

```
template <typename A, typename B>
const double MaxError< BinXpr< A,B, sum > >::VALUE =
(MaxBitlen< BinXpr< A,B, sum > >::VALUE > 53)
   ? (((( MaxError<A>::VALUE + MaxError<B>::VALUE )
        * fp_traits<double>::RRE )
       + P2< MaxBitlen<BinXpr<A,B,sum> >::VALUE - 53>::VALUE )
      * fp_traits<double>::RRE )
   : 0.0 ;
```

Note, that the values given in Table 1 must be representable as a `double`. Their computation may thus cause rounding. Depending on the rounding mode the computed error bound may be to small. To avoid this, we encapsulate each error prone operation in a multiplication with relative rounding error `fp_traits<double>::RRE` with a value of $1 + 2^{2-D}$, where D is the number of mantissa digits of a `double`.

4.1.2 Exact Polynomial Sign Computation

If the static filter fails to verify the correctness of the sign of the approximation we must use some other arithmetic that can evaluate the expression reliably. We use the approach of [4] to implement exact arithmetic. A multiprecision integer a is stored as a $n+1$-digit 2^R-adic number as

$$a = \sum_{i=0}^{n} 2^{iR} a_i$$

This representation is called the *expansion* of a, which is called normalized, if $a_i < 2^R$ for all $i = 0, \ldots, n$. The sum (difference) of two expansions is the componentwise sum (difference) of the expansions of the operands. The product $c = a * b$ is computed by the so-called *school method* as

$$c_k = \sum_{i+j=k} a_i b_j \text{ for } k = 0, \ldots, n+m-2$$

where a and b are expansions of size n and m, respectively. Note, that neither the sum nor the product of two normalized expansions are necessarily normalized. In our implementation each components of the expansions is a `double` value. To obtain the exact result no intermediate operation must require more than the number of mantissa digits of a `double` to store its results. In [4] it was recommended to use a radix of $R = 23$. In this way we can make one multiplication and several additions without normalizing the intermediate results. We adhere to this suggestion in our prototype implementation.

Having fixed the radix R, there is also a fixed sequence of operations that is required to obtain the expansion of a given expression type. In the following we describe the algorithm that derives this sequence from the expression type given at compile time.

Static Storage Layout:
To evaluate the polynomial we use a static array of `double`s that holds the components of the multiprecision integer representation of a given expression. All algorithms involved in the computation of the exact result operate on parts of this array. For a given binary expression we recursively evaluate the subexpressions and then apply the current operation to the results. For each subexpression we designate a portion of the array available for their own evaluation. We reuse parts of the array, since the results of the subexpressions are not used after evaluation of the current expression. For example, when adding two expansions the result overlaps the longer of the the two expansions and the shorter is added in-place, thereby avoiding to copy the leading components of the longer operand.

Unrolled Arithmetic:
The main advantage of using expression templates is the ability to communicate the structure of subsequences of the evaluation code to the compiler. Each operation, such as computing the sum of two expansions is split into small blocks of code, that is repeated for a fixed number of times. Each block is an `inline` function template. During optimization, the compiler decides whether the function is inlined. If so, we obtain a sequence of code without any overhead for function calls, memory management or looping.

Thus, we leave it to the compiler to generate code that is most efficient, at least w.r.t. to its optimization algorithms. We demonstrate this technique on the example of computing the sum of two expansions a and b. Thereto, suppose a has N and b has M components with $N \geq M$, respectively. As mentioned above, we add b in-place to a. We have a template class

```
template <int M>  struct PolySum {};
```

where M is the number of components of b that has to be added to a. The recursion is as follows:

```
template <int M>
struct PolySum {
  template <typename S>
  TLN_INLINE static void eval(S* const a, S* const b){
      PolySum<M-1>::eval(a,b);
      a[M-1] += b[M-1];
   }
};
```

,i.e., first add the the other components and finally add the M-th component of b to the M-th component of a. The template parameter `S` is the number type used for the components of the expansions. In our current implementation this is always `double`. Finally, if there is nothing to add we stop, as realized with the following specialization:

```
template <>
struct PolySum <0> {
  template <typename S>
  TLN_INLINE static void eval(S* const a, S* const b){ }
};
```

In this way we implemented all arithmetic on our expansions including, multiplication and normalization. By inspection of the generated code we observed, that almost all the function calls are inlined by the compiler. At least for the expressions we used in the runtime tests. See section ?? for details.

Normalization Policy:
In order to keep all computations exact no intermediate result must exceed the precision of the `double` type. However, the bitlength of the numbers is increasing in each computation step. Therefore, we have to reduce the bitlength of the components of the expansion at some point during the evaluation.

Note that normalization is a relatively expensive operation, since we use a `fmod(...)` library call to compute the remainder of the `double` components on division by 2^R. Hence, we want to avoid as many unnecessary normalizations as possible.

We use the following simple strategy to apply normalizations to the operands of an operation: For each expansion we maintain the maximum bitlength of its components according to Table 2. If the maximum component bitlength of the result after evaluation is too large we normalize both operands. Then the value given in Table 2 is replaced by the radix R. Sometimes it is not sufficient

c	`MaxCompBitlen`(c)
$-a$	`MaxCompBitlen`(a)
`integer<N>`	N
$a \pm b$	$1 + \max\{$ `MaxCompBitlen`(a) , `MaxCompBitlen`(b) $\}$
$a * b$	`MaxCompBitlen`(a) + `MaxCompBitlen`(b) $+\lfloor \log_2 \min\{\ n, m\ \} \rfloor$

Table 2: Rules for computation of the maximum component bitlength of expansions. a and b are expansions with n and m components, respectively.

to normalize two expansion before multiplication. To see this, suppose we are multiplying two normalized expansions $a = a_0, \ldots, a_n$ and $b_0, \ldots, b_m$ then the maximum component bitlength according to Table 2 is $L = 2R + \lfloor \log_2 \min\{\ n, m\ \} \rfloor$. However, since R is fixed L only depends on the sizes of the expansions. Let D be the number of mantissa bits of the data type `double`. To have exact intermediate results we need $L \leq D$. Thus, one of the components to be multiplied must have less than 2^{D-2R+1} components to obtain exact intermediate results. For $D = 53$ and $R = 23$ we obtain a maximum expansion size of the shorter operand in any multiplication of 255 which corresponds to a number with $D2^{D-2R+1} = 5.865$ bits. Roughly speaking, for 53-bit integer inputs we can multiply two polynomials each having a degree up to 110. To see this, consider the simple polynomial $(a^k)^2$, where a is a 53 bit integer. The expansion of a^k has $\lceil \frac{kD}{R} \rceil$ components. For the inequality

$$\left\lceil \frac{kD}{R} \right\rceil < 2^{D-2R+1}$$

with $R = 23$ and $D = 53$ we get that $k < 111$.

For small degree polynomials used in computational geometry, this limitation can be neglected. We decided not to implement the normalization within operations in our prototype. To do this, we will have to redesign the current rather simple structure of the evaluation part of the library. However, by using so-called *compile time assertions*, we let the compiler issues an error if the user evaluates expressions exceeding the above limit.

Sign Determination:
After we have computed the full expansion of a given expression the sign of the number represented by this expansion is computed by first normalizing the expansion and then searching for the most significant non-zero component and report the sign of this component. If all components are zero, the sign is also zero.

Finally, we return to our orientation test example which is evaluated as follows:

1. Compute the floating-point approximation
2. Compare with the error bound computed at compile-time and return the sign if certain.
3. Evaluate the leaves. Compute the differences using doubles, since the result fits into 53 bits.
4. Normalize the results. This gives four expansions with three components each.
5. Multiply the two pairs of expansions. This gives two five-component expansions with a maximum bitlength of 48 in each component.
6. Compute the difference of these expansions.
7. Normalizing the result gives six components.
8. Search for the most significant non-zero digit of this expansion and return the sign.

Inspection of the binary produced by `gcc-4.2` revealed, that all functions were inlined, including constructors of the expressions and all evaluation functions.

5 Experimental Results

To evaluate the performance of `TLN` we measured the running time of some common geometric predicates in two and tree-dimensional computational geometry. The tested predicates are: 2- and 3-dimensional orientation test, incircle and insphere test. Each predicate evaluates the sign of a determinant – a polynomial in the coordinates of the input points. Each polynomial is written as a single `C++` expression.

5.1 Test Data and Environment

To evaluate the efficacy and the costs of the filter we tested the predicates on random and almost degenerate input data using point generators of CGAL. Random input for the predicates with point coordinates with a maximum bitlength of L is generated using `CGAL::Random_points_in_square_2` and `CGAL::Random_ points_in_cube_3` which produce points in a cube centered at the origin with corners at $\pm 2^L - 1$ for two- and three-dimensional predicates, respectively. The generated points with `double` coordinates were then rounded to the next smaller integer using `std::floor(...)`.

To obtain a test set of almost degenerate data we first generate a random segment, circle, plane or sphere and then use the corresponding point generators of CGAL to get points close to these geometric objects. Again, the coordinates are rounded to the next smaller integer. To be more precise, we generate nearly degenerate input configurations for each predicate as follows:

- 2-*dimensional orientation test:*
  - Fix `CGAL::Random_points_in_square_2` a and b.
  - Generate $3n$ points on the line segment from a to b with `CGAL::Random_points_on_segment_2`.
- *incircle test:*
  - Fix a circle c centered at the origin with radius chosen randomly from the interval $[L, 2^L - 1]$.
  - Generate $4n$ points on c with `CGAL::Random_points_on_circle_2`.
- 3-*dimensional orientation test:*
  - Fix a sphere s centered at the origin with radius chosen randomly from the interval $[L, 2^L - 1]$.
  - Fix three distinct random points a, b, c on s using `CGAL::Random_points_on_sphere_3`
  - Generate $4n$ points as the orthogonal projection of random points on s onto the plane through a, b and c.
- *insphere test:*
  - Fix a sphere s centered at the origin with radius chosen randomly from the interval $[L, 2^L - 1]$.
  - Generate $5n$ points on s with `CGAL::Random_points_on_sphere_3`.

In the experiments we used $n = 1.000$ for a single data set and compute the average running time over 10 different sets, each set being evaluated 100 (2d-orientation, incircle) and 10 (3d-orientation, insphere) times, respectively, to get timing results in a reasonable resolution.

The experiments were run under Debian Linux operating an Intel®Pentium®4 CPU at 2.4 GHz. We used `gcc` version 4.2.4 with optimization level 2 to compile the test code. The following third party libraries were used: CGAL (version 3.3.1, including CORE), LEDA (free-version 6.0) and GMP (version 4.2.4).

5.2 Results

To assess the efficiency of the code generated by the compiler we compare the running times of the `TLN` predicates with predicates using off-the-shelf multiprecision number types. For each predicate we have a fixed `C++` expression that has to be evaluated for the given input points to determine

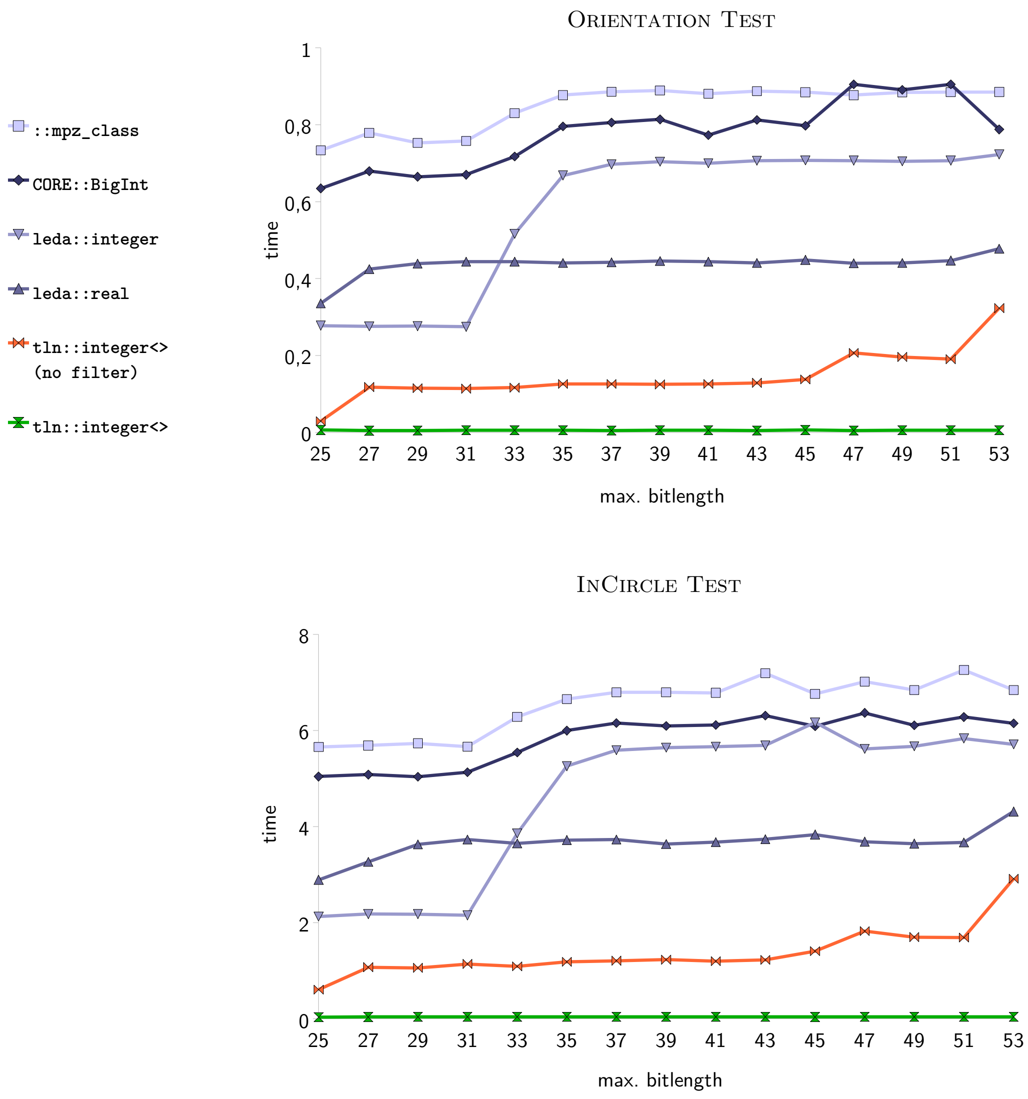


Figure 2: Running times for 2-dimensional geometric predicates on random input.

their relative geometric positioning. The time measured includes the conversion of the coordinates to the number type used to evaluate the sign of the expression and the actual computation of the sign. In the tests we used the following external off-the-shelf `C++` multiprecision number types : `::mpz_class` (GMP Library), `CORE::BigInt` [3, 6], `leda::integer`, `leda::real` [7, 1]. `TLN` was used in one version with a static filter (`tln:integer<>`) and without filtering in a second version (`tln:integer<> (no filter)`).

The results for random input for the two and three-dimensional predicates are given in Figure 2 and 3, respectively. Figures 4 and 5 show the results for nearly degenerate input as described in the previous section.

We will restrict our discussion to the comparison of the `TLN` number types to the external types.

In the filtered case our `tln:integer<>` is up to 100 times faster than the other methods. `tln:integer<> (no filter)` is still roughly twice as fast as the fastest external number type. With increasing maximum bitlength we also see an increase in the running time. The increase appears at certain key positions which are determined by both, the predicate and the radix used for representing the expansions. On these positions the number of components of the result or an intermediate result changes. This leads to an increasing number of machine operations and normalizations required for exact evaluation of the expression.

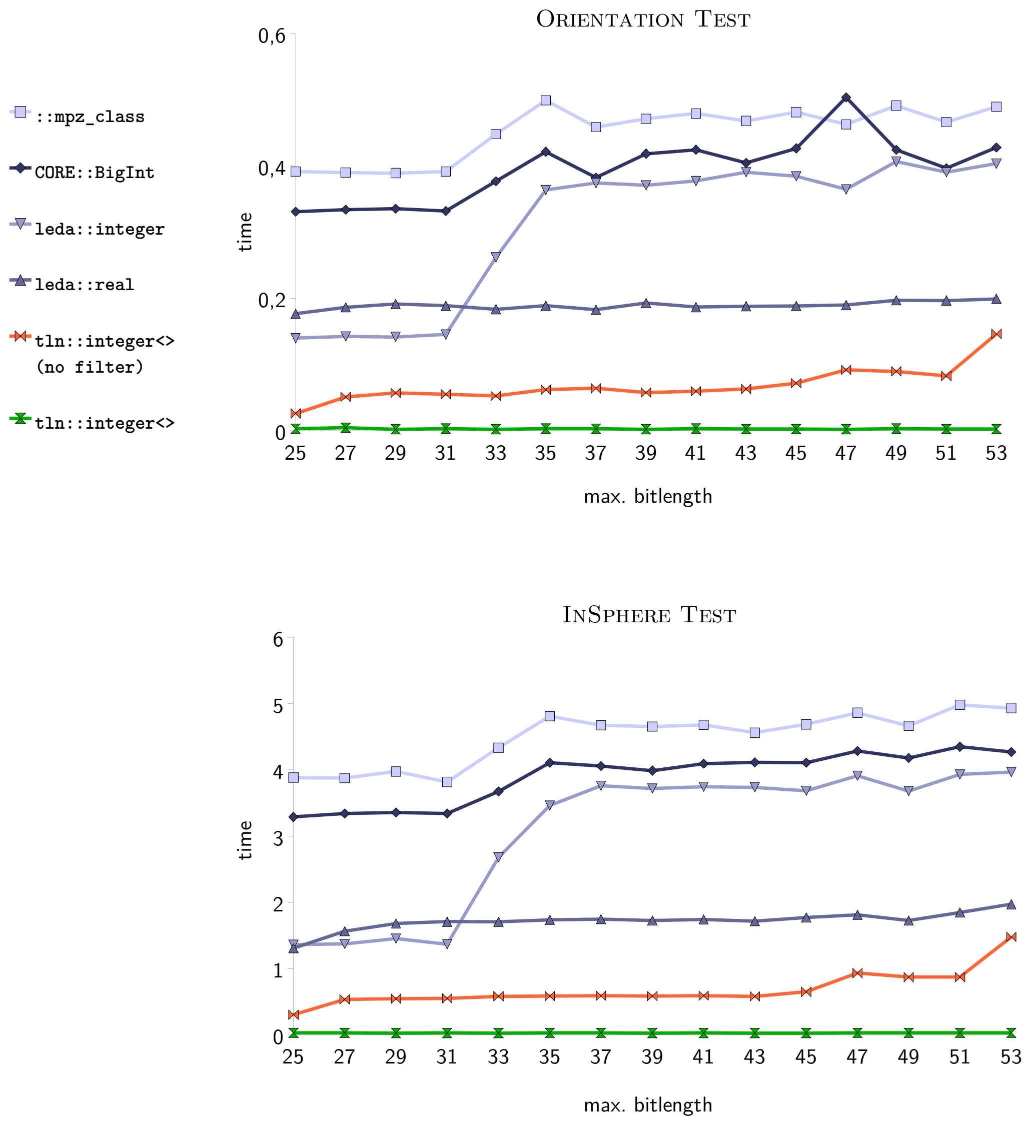


Figure 3: Running times for 3-dimensional geometric predicates on random input.

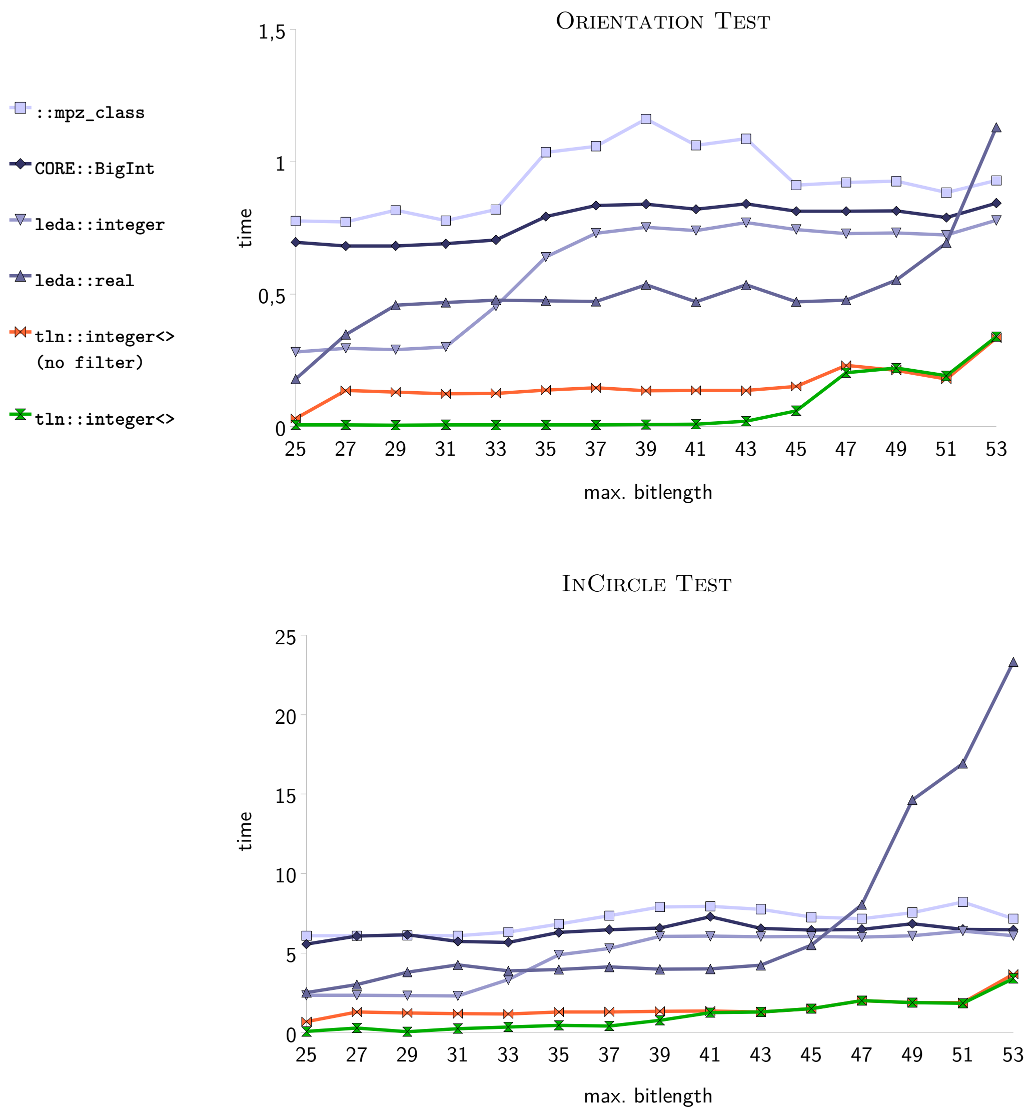


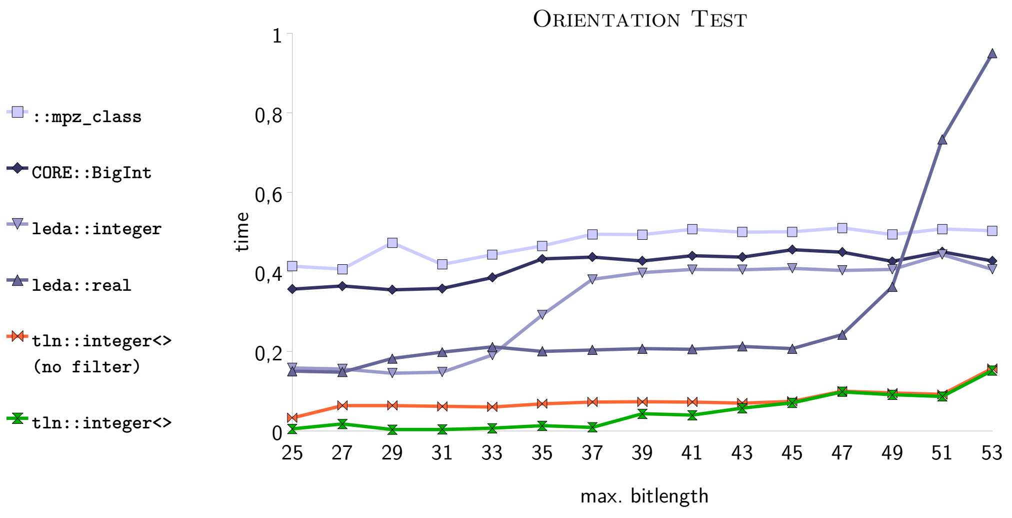
Figure 4: Running times for 2-dimensional geometric predicates on nearly degenerate input.

For example, consider times shown for the two-dimensional orientation test in Figures 2 and 4. We see an increase between a maximum bitlength of 25 and 27 since we cannot multiply 27 bit numbers without exceeding the precision of a `double`. The next increase happens close to 46 bits because whenever the maximum bitlength some intermediate result reaches a multiple of the radix (which is 23) we need one more component to store the corresponding expansion. Finally, we see an increase at full `double` precision of 53 bits. This is due to the fact, that the initial differences are not exact. Thus each operand (input variables) must be normalized, which doubles the number of required normalizations at this level compared to the 52-bit version.

The results for nearly degenerate input given in Figures 4 and 5 show that the static filter is still efficacious for smaller maximum bitlength values. However, we believe that this is rather a deficit of our method to generate these inputs. On the the other hand we take it as an evidence for the hypotheses, that nearly degenerate input occurs only rarely in applications.

6 Discussion and Further Work

Our prototype implementation of `TLN` has proven to generate efficient predicates for computational geometry in two and three-dimensional space. The inputs are restricted to integers for which a



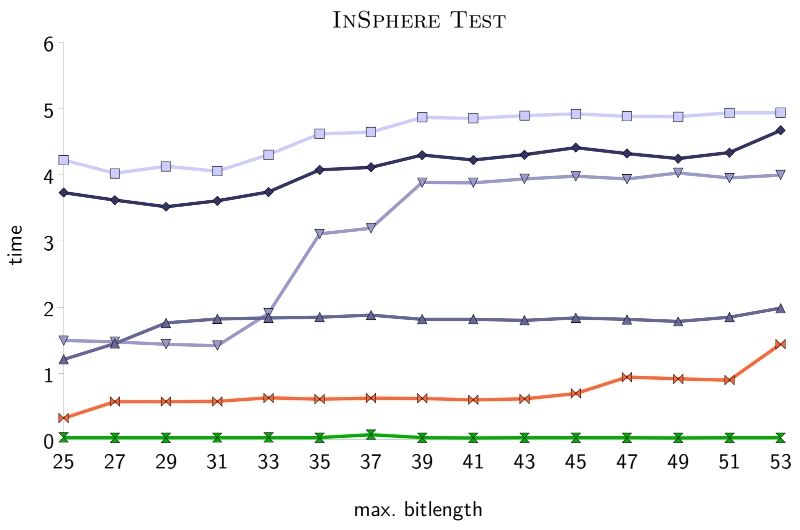


Figure 5: Running times for 3-dimensional geometric predicates on nearly degenerate input.

maximum bitlength is known in advance. The currently used static filter however, is only efficacious if the values of the inputs are close to the corresponding maximum value. In applications, this assumption in general does not hold. To improve the filtering efficacy we plan to implement the semi-static filter used by `EXPCOMP` in a future version of `TLN`. Moreover, it is possible to allow floating-point inputs for the predicates for which this filter works well, as demonstrated in [2]. But at the moment it is not clear how to evaluate polynomials with fractional inputs. One option is to represent the values as rational numbers, i.e, as the quotient of two integers and then use algorithms similar to the current implementation. But we believe, that this is not very efficient, since the number of machine operations required for the exact evaluation grows significantly. Another approach could use the arithmetic proposed by Shewchuk [11] in a static form or even his adaptive evaluation of the predicates as proposed in [8].

For integer inputs we have in mind to implement a sequence of static filters to make the predicates adaptive to some extent. The idea is to compute only a portion of the most significant bits, i.e., a few components of the result during exact evaluation of the expression together with a static error bound that can be derived from the omitted parts of the input. The problem is to find reasonable rules for general expressions about how many stages of filtering are advisable and how many bits are to be computed in the stages. With the current exact evaluation algorithms we would add many normalizations in each stage. But normalizations are expensive compared to other machine operations.

Another important aspect for the usability of `TLN` is portability. In the current status the prototype was not tested on other platforms or compilers other then Linux/`gcc`. Portable template programming, in particular meta-programming, is a subtle task, since the set of supported features of the `C++` standard varies drastically between compilers. For example Microsoft®Visual C++®.NET does not support partial template specialization which is quite a limitation when doing meta-programming. Fortunately, many `boost` libraries including `boost::mpl` can cope with many of these limitations and work well on many platforms. Thus, using `boost` we can improve the portability of `TLN` in a future version.

References

[1] Christoph Burnikel, Rudolf Fleischer, Stefan Funke, Kurt Mehlhorn, Stefan Schirra, and Susanne Schmitt. The leda class **real** number – extended version. Technical Report ECG-TR-363110-01, MPI Saarbrücken, 2005.

[2] Christoph Burnikel, Stefan Funke, and Michael Seel. Exact geometric predicates using cascaded computation. In *SCG '98: Proceedings of the fourteenth annual symposium on Computational geometry*, pages 175–183, New York, NY, USA, 1998. ACM.

[3] *The CORE Library project.* `http://cs.nyu.edu/exact/core_pages/`.

[4] Steven Fortune and Christopher J. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. Graph.*, 15(3):223–248, 1996.

[5] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[6] Vijay Karamcheti, Chen Li, Igor Pechtchanski, and Chee-Keng Yap. A core library for robust numeric and geometric computation. In *15th ACM Symposium on Computational Geometry (SCG'99)*, pages 351–359, New York, NY, USA, 1999. ACM.

[7] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, Cambridge, UK, 2000.

[8] Aleksandar Nanevski, Guy Blelloch, and Robert Harper. Automatic generation of staged geometric predicates. *Higher Order Symbol. Comput.*, 16(4):379–400, 2003.

[9] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry - An Introduction.* Springer, 1985.

[10] Stefan Schirra. Robustness and precision issues in geometric computation. In Jörg-Rüdiger Sack and Jorge Urrutia, editors, *Handbook of Computational Geometry*, chapter 14, pages 597–632. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.

[11] Johnathan Richard Shewchuk. Robust adaptive floating-point geometric predicates. In *SCG '96: Proceedings of the twelfth annual symposium on Computational geometry*, pages 141–150, New York, NY, USA, 1996. ACM.

[12] Bjarne Stroustrup. A history of c++: 1979–1991. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 271–297, New York, NY, USA, 1993. ACM.

[13] Bjarne Stroustrup. *The design and evolution of C++.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[14] Bjarne Stroustrup. Evolving a language in and for the real world: C++ 1991-2006. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 4–1–4–59, New York, NY, USA, 2007. ACM.

[15] David Vandevoorde. *C++ Solutions: Companion to the C++ Programming Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[16] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[17] Todd Veldhuizen. Expression templates. pages 475–487, 1996.

[18] C. K. Yap. Robust geometric computation. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 35, pages 653–668. CRC Press LLC, Boca Raton, FL, 1997.