Technical report

Nr.: FIN-001-2010

On the Design and Implementation of a Generic Number Type for Real Algebraic Number Computations Based on Expression Dags

Marc Mörig, Ivo Rössling, Stefan Schirra

Arbeitsgruppe Algorithmische Geometrie

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Nr.: FIN-001-2010

On the Design and Implementation of a Generic Number Type for Real Algebraic Number Computations Based on Expression Dags

Marc Mörig, Ivo Rössling, Stefan Schirra

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
Marc Mörig
Postfach 4120
39016 Magdeburg
E-Mail: moerig@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

On the Design and Implementation of a Generic Number Type for Real Algebraic Number Computations Based on Expression Dags *

Marc Mörig Ivo Rössling Stefan Schirra

Department of Simulation and Graphics,
Otto-von-Guericke University Magdeburg, Germany,
moerig@isg.cs.uni-magdeburg.de

October 01, 2009

Abstract

We report on the design and implementation of a number type called `Real_algebraic`. This number type allows us to compute the correct sign of arithmetic expressions involving the operations $\pm, \cdot, /, \sqrt[d]{\ }$. The sign computation is always correct and, in this sense, not subject to rounding errors. We focus on modularity and use generic programming techniques to make key parts of the implementation easily exchangeable. Thus our design allows for easily performing experiments with different implementations or thereby to tailor the number type for specific tasks. For many problems in computational geometry instantiations of our number type `Real_algebraic` are a user-friendly alternative for implementing the exact geometric computation paradigm in order to abandon numerical robustness problems.

1 Introduction

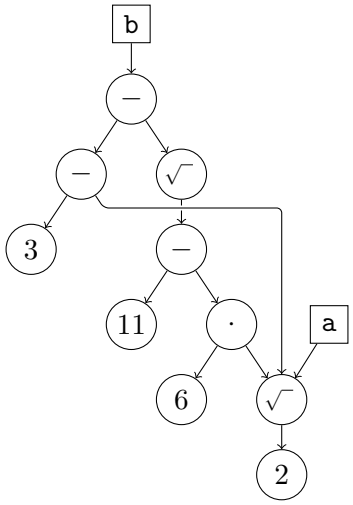
Recording the computation history of a numerical value in an expression tree, more precisely, in an expression dag, allows us to recompute the value at a later stage of a program in a different way. For example, the created expression dags allow for adaptive lazy evaluation: First, we compute a crude numerical approximation only. If the current approximation does not suffice anymore, we can use the expression dag to iteratively compute better and better approximations.

A typical application of this scheme is verified sign computation. If the actual numerical value is far away from zero, rough numerical approximations suffice to compute the correct sign. Only if numerical values are close to zero, high precision computation is needed. By correct sign computations we assure correct control flow. This is the crucial idea of the so-called exact geometric computation paradigm [26]. Regarding control flow, we ensure that the implementation behaves like its theoretical counterpart, thereby ensuring correct combinatorics, whereas numerical values might still be inaccurate. However, the potential inaccuracy never leads to wrong or even contradictory decisions.

For the sake of ease-of-use recording computation history and adaptive lazy evaluation is wrapped in a number type. In programming languages providing operator overloading, such a number type can then be used like built-in number types. A user need not care about any implementation details in order to get verified signs. There are certainly other techniques for verified sign computations, which lead to more efficient code. However, applying these techniques requires a deep understanding of the underlying theory and thus they are much less user-friendly.

*Supported by DFG grant SCHI 858/1-1

$$3 - \sqrt{2} - \sqrt{11 - 6\sqrt{2}}$$



⟨*simple expression*⟩≡

```
typedef Real_algebraic<...> NT;
NT a = sqrt(NT(2));
NT b = 3 - a - sqrt(11 - 6*a);
```

Figure 1: An expression, a corresponding DAG, and the code leading to this DAG.

We present generic design and basic implementation of a number type based on expression dags. We call it `Real_algebraic`. The number type allows you to exactly compute the sign of arithmetic expressions involving the operations $\pm, \cdot, /, \sqrt[d]{}$.

Since `Real_algebraic` is based on expression DAGs we start by reviewing the fundamentals of this technique. A number is represented as a handle into an directed acyclic graph or short DAG that records the creation history of that number. Creating a DAG representation for another, explicitly represented number simply creates a DAG node storing it. An arithmetic operation creates a new node, storing the type of the operation and pointers to the operands. A handle to the new node is returned. Each handle and each node corresponds to a unique arithmetic expression which in turn corresponds to a unique real algebraic number. The converse is not true, each real algebraic number is representable by many expressions (or none) and for an expression there may be several DAGs representing it. Most notable is the possibility to represent two identical subexpressions with just one subDAG. The actual structure of the DAG depends on the way it is created. See Figure 1 for an example DAG.

As said above, the reason for representing a number as an expression DAG is to allow for lazy adaptive sign computation, especially in geometric computing. The obvious alternative is to compute a more explicit "exact" representation with each arithmetic operation instead of just creating a DAG. For example, in case of algebraic numbers one could compute a polynomial with enclosing interval with each arithmetic operation, in case of rational numbers one could represent them by a quotient, where both numerator and denominator are represented by arbitrary precision integers. However, these approaches are not adaptive, i.e., the cost of sign computations with these approach does not really reflect the difficulty involved. Many sign computations in typical geometric computations in practice are simple however, meaning that the number whose sign must be determined has a relatively large absolute value. Then the sign can be determined by computing a crude approximation. Representing a number as a DAG allows us to iteratively compute approximations of increasing accuracy until the sign is know. This make the algorithm adaptive, its running time depends also on the absolute value of the number whose sign must be computed, the worst case is only attained if this number is actually zero.

We associate a DAG node v with the real algebraic number it represents. To compute the sign of v the expression corresponding to the subDAG below v is evaluated. Using software floating-point arithmetic with arbitrary precision an interval I containing v is computed. If zero is not contained in I the sign of v is known. Otherwise the expression is reevaluated with higher precision to reduce the size of I. Reducing the size of I repeatedly may however not suffice to compute the sign of v. If v is actually zero, this can only be detected if I contains zero only. In the presence of division and root

operations this may never occur. This case is resolved with the help of so called separation bounds. A separation bound consists of a set of rules that allows to compute a positive number $\mathrm{sep}(v)$ that is a lower bound on $|v|$ unless v is zero. We can conclude that v is zero if I is a subset of $[-\mathrm{sep}(v), \mathrm{sep}(v)]$.

There are many ways to refine this rough scheme and there is no obvious best variant. Thus, a lot of algorithm engineering is needed in order to detect superior and discard inferior methods. Our generic implementation is not the first number type based on expression DAGs. The number type `CORE::Expr` [12, 4] has been developed by Chee Yap and his co-workers at New York University, and the number type `leda::real` [5, 14] has been developed as part of the LEDA library. Our generic implementation subsumes both as special cases. Finding improvements over the existing implementations is the main motivation for our work.

1.1 Policy based class design

There is almost no modularity in `leda::real`, the implementation consists of two classes, one for the handle type and another one for the DAG node type. `Core::Expr` uses polymorphism to represent the different DAG node types corresponding to different operations. Since version 2 the software floating-point arithmetic, the separation bound and the floating-point filter used in the sign computation algorithm are exchangeable. The `CORE` library provides however only one choice for each of the three modules.

Our new number type `Real_algebraic` allows the exact sign computation for a subset of the real algebraic numbers. Any `int` or `double` is a `Real_algebraic` and `Real_algebraic` is closed under the operations $\pm, \cdot, /$ and $\sqrt[d]{\ }$. While maintaining generality, user friendliness and efficiency our focus in the design has been on flexibility. Our implementation is separated into several modules, each captures a certain algorithmic aspect. This enhances the maintainability of our implementation. We use generic programming techniques to make those modules easily replaceable. So in fact we do not provide a single new number type but a whole set of different but related number types. This allows us to perform experiments to further increase the efficiency. As a side effect it allows the user to set up a variant of `Real_algebraic` that best fits her needs.

The goal of policy based class design is to find relevant design decisions, things that can be implemented in different ways or involve a trade off then factor them out into separate, independent modules and allow to exchange them. This allows to evaluate the design decision or postpone the decision to the user.

We use generic programming based on the `template` feature of `C++` to make parts of the implementation exchangeable. Functions or classes, then called function templates and class templates are implemented depending on a so called template parameter – a placeholder for some type that must be specified or substituted at compile time. The template imposes certain requirements on the type to be substituted, both syntactical and semantical. The entity of those requirements is called the *Concept* for the type to be replaced. Since there is no direct language support for concepts, they have to be documented well. Types that actually fulfill a concept are said to be a `Model` of this concept. The main advantage of generic programming is that templates, once the parameters have been specified, yield code that is as efficient as a hand written version.

According to Alexandrescu [1], a policy is a concept that encapsulates a certain behavioral aspect of an algorithm or class. A small example given by him is that of a *CreationPolicy*. This policy depends on a template parameter `T` itself and therefore should more accurately be called a policy template. *CreationPolicy* provides a single member function `Create()` that shall return a pointer to a new object of type `T`. Different models of *CreationPolicy* can implement `Create()` differently, they can for example create an object using `new` or alternatively `malloc()` and placement `new`. A class based on one or more policies is called a host class. Host classes have to be customized at compile time by selecting an appropriate model for each policy. If a host class is based on several policies, it is important that those are orthogonal, meaning that models for those policies are freely combinable. It is for example not advisable to combine a *CreationPolicy* model that uses `new` with

a *DestructionPolicy* model that uses `free()`. In fact handling creation and destruction by different policies is a bad idea.

The setup of one of more policies controlling the behavior of a host class is a variant of the strategy design pattern [8] but with an emphasis on the point that the strategy or policy is supplied through a template parameter and not a virtual base class that concrete policies must implement. The downside is that a policy can not be exchanged at runtime. The advantage is, that policies can exercise a very fine-grained control efficiently. Consider an aspect that can not be represented by a single function, but affects small pieces of code at many places in the implementation. Using object oriented techniques to factor out this aspect would require a virtual function call at each affected place. When the policy is a template parameter there are no virtual function calls. In fact if the functions provided by the policy are small, each call may be inlined, resulting in code that is as efficient as hand written code.

1.2 Design Overview

We already reviewed the basic ingredients for arithmetic with expression DAGs. Creating a DAG node for an arithmetic operation is not free of charge. In some cases it may be faster to actually perform the operation and compute an exact, explicit representation for the result. The *LocalPolicy* provides a strategy to postpone or avoid the creation of DAG nodes by performing operations directly if possible. Complementary is the *ExpressionDagPolicy* that handles all operations on the DAG. The *DataMediator* provides conversion of numbers from the *LocalPolicy* to the *ExpressionDagPolicy*. All three are combined in the host class `Real_algebraic` that implements the handle to a DAG node. In Section 2 we have a more detailed look at *LocalPolicy*, *DataMediator* and how they interact with *ExpressionDagPolicy* inside `Real_algebraic`. We shortly introduce two *LocalPolicy* models. Section 3 is dedicated to another *LocalPolicy* that represents a number as a sum of `doubles`.

Only one *ExpressionDagPolicy* model `Original_leda_expression_dags` is implemented, it follows the sign computation algorithm from `leda::real` very closely. Key tools any sign computation algorithm has to use are exchangeable in `Original_leda_expression_dags`, but the algorithm itself is monolithic. First there is the *ApproximationPolicy* that provides arbitrary precision software floating-point arithmetic. Second there is the *SeparationBound* providing an implementation of a separation bound. Finally we have the *FilterPolicy* that provides a floating-point filter based on hardware floating-point numbers. Such a filter is not necessary for a sign computation algorithm, but can speed it up in many cases. These three parts are in correspondence to the modules that are exchangeable in `CORE::Expr`. We discuss the *ExpressionDagPolicy*, our model `Original_leda_expression_dags` and its three policies in Section 4. Approaches to further modularize and improve the sign computation algorithm are discussed in Section 5.

2 LocalPolicy and Real_algebraic

The class template `Real_algebraic` is a host class that depends on three policies, the *LocalPolicy*, the *ExpressionDagPolicy* and the *DataMediator*. The basic strategy of expression DAG based number types is to record the structure of an arithmetic expression as a DAG. More precisely each number is represented by a handle to a DAG node. This node may either store the number directly or is labeled with an operation and stores pointers to the operands which are again DAG nodes, confer Figure 1. Maintaining this representation is simple, when creating a number through a constructor call or through an operation, we create a new node with the appropriate data and return a handle to it. The class template `Real_algebraic` effectively implements the handle, the creation of DAG nodes and all other operations on the DAG like sign computation are handled by the *ExpressionDagPolicy*.

Creating a DAG node is not free from charge. It requires to allocate dynamic memory for the node and the data inside the node must be initialized. To reduce the cost of creating nodes `Real_algebraic` as well as `leda::real` and `CORE::Expr` use a memory manager specifically tuned to the frequent

allocation and deallocation of small, fixed size chunks of memory. Nevertheless adding two simple `doubles` requires to create and initialize three nodes. In many cases it might be faster to compute the result of an operation explicitly and exactly and only resort to expression DAGs if this is not possible. Of course this requires an explicit representation of the operands. Performing some initial operations explicitly before starting to create a DAG will also reduce the size of the DAG and therefore reduce the running time for sign computation.

The *LocalPolicy* is designed to enable this. It reserves a small memory chunk inside the handle to store a number. This chunk typically has a fixed size to avoid dynamic memory allocation. If the number represented by a handle is directly stored in this memory chunk we say it is represented locally. Therefore the number represented by a `Real_algebraic` may be represented locally or by a DAG node or both. When a number is created through a constructor we represent it locally if possible. If all arguments to an operation are represented locally and the result can be computed exactly and it fits into the reserved memory chunk we simply store it locally again and avoid the creation of a DAG node. If any of this is not the case we resort to creating a DAG. This might require to first transform the local representation of the operands into a DAG node representation which is done by the *DataMediator*. Then the *ExpressionDagPolicy* is used to perform the operation as usual.

A *LocalPolicy* provides storage for a number, constructors, member functions for the operations $\pm, \cdot, /, \sqrt[d]{}$ and for sign computation. It knows internally if it currently stores a number, but does not expose this knowledge explicitly. Instead all functions simply return whether they could be performed locally or not. Here are two examples:

⟨*LocalPolicy*⟩≡
```
bool local_multiplication(const LocalPolicy a, const LocalPolicy b);
```

shall compute a local representation for the product of a and b. Returns `true` upon success, i.e., iff the product could be computed locally.

⟨*LocalPolicy*⟩+≡
```
bool local_sign(int& s);
```

shall set s to the sign of the locally represented value. Returns `true` upon success i.e., iff the sign could be computed locally. Constructors can not return whether their argument is locally representable. The alternative is to first create a *LocalPolicy* using the default constructor and then set it's value with some member function but this might be inefficient, so we choose a different solution.

⟨*LocalPolicy*⟩+≡
```
LocalPolicy(double d);
```

shall create a `LocalPolicy` that represents d, if d can be locally represented. If creation succeeded can be checked afterwards.

⟨*LocalPolicy*⟩+≡
```
bool local_creation_succeeded(double d);
```

returns true if the `LocalPolicy` represents d, under the precondition, that it has been constructed from d. For a variety of local representations this pair of functions allows an efficient implementation of the creation process. `local_creation_succeeded()` may compare d with the stored number, or return a flag that has been set in the constructor, or simply always return `true`, e.g., if a `double` can always be represented locally. Which implementation is best of course depends on the specific number type used for local representation.

The *ExpressionDagPolicy* provides an interface similar to that of *LocalPolicy*, but without the option to refuse to perform some operation. We discuss the *ExpressionDagPolicy* in more detail in Section 4. The *DataMediator* plays a central role in the collaboration of *LocalPolicy* and *ExpressionDagPolicy*. One of the goals of policy based design and modularization in general is to create orthogonal policies. With *LocalPolicy* and *ExpressionDagPolicy* this is not easily possible. We would like to allow virtually any kind of number representation inside a *LocalPolicy* and we have to provide

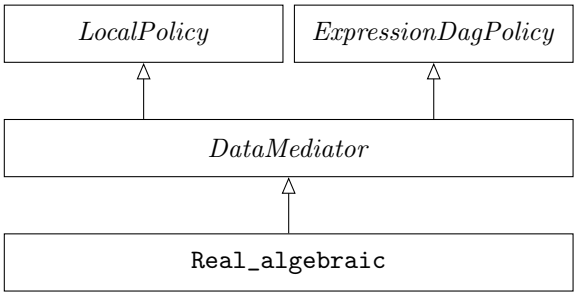


Figure 2: Class hierarchy of `Real_algebraic` and its immediate policies.

a way to convert this representation into an expression DAG. The *ExpressionDagPolicy* provides creation of a DAG node from `int` and `double`, furthermore it computes with a software floating-point number type `AT` provided by *ApproximationKernel*, therefore we can easily allow the creation of a node from `AT` too. A natural solution to decouple *LocalPolicy* and *ExpressionDagPolicy* is to let the *LocalPolicy* provide its value as `AT` and then create a DAG node from this representation. This is however restricting the *LocalPolicy*, since any locally representable value must be exactly convertible to `AT`. It prohibits e.g., *LocalPolicy* models that store some type of quotient to support division. To overcome the restrictiveness of `AT` one could introduce a more general intermediate representation, that ultimately should be able to represent any number that can be locally represented by any *LocalPolicy* model. This leads to an intermediate representation that is probably DAG based itself. Converting twice, first to a complex intermediate representation and then to the final DAG representation will in any case be inefficient.

Our solution is to implement the conversion method separately in a *DataMediator*. A *DataMediator* model depends on both the *LocalPolicy* model and *ExpressionDagPolicy* model it is made for and gets privileged access to both. It encapsulates the dependencies between both policies, thereby decoupling them. Of course this might require a quadratic number of *DataMediator*s. On the other hand the conversion method depends more on the *LocalPolicy* than the *ExpressionDagPolicy*. Many *DataMediator*s will convert the local data to an `int`, `double` or `AT` and create a single DAG node storing it. Those can in fact be generalized on and used with any *ExpressionDagPolicy*.

We compose the host class `Real_algebraic`, that implements the handle to a DAG node from its three policies by means of inheritance, confer Figure 2. This serves two purposes. First it aggregates the data fields from the *LocalPolicy* for storing a local representation and the data fields from the *ExpressionDagPolicy*. Second it allows the *DataMediator* privileged access to those data fields. The *DataMediator* is allowed to access those fields to implement an efficient conversion from the local representation to an expression DAG representation. The class `Real_algebraic` itself however is implemented by means of its policies only. We give some sample code from its implementation.

⟨*Real_ algebraic*⟩≡
```
friend Real_algebraic<Policies>
operator*(const Real_algebraic<Policies>& a,
          const Real_algebraic<Policies>& b){
  Real_algebraic<Policies> c;
  if(!c.local_multiplication(a,b)){
    a.create_dag_from_local_data();
    b.create_dag_from_local_data();
    c.expression_dag_multiplication(a,b);
  }
  return c;
}
```

implements the multiplication by means of its policies. `local_multiplication()` is provided by the *LocalPolicy*, `expression_dag_multiplication()` by the *ExpressionDagPolicy*. The function

`create_dag_from_local_data()` checks first if there already is an expression DAG representation before before using the *DataMediator* to create one. All other arithmetic operations and comparison operators are implemented analogously.

⟨*Real_ algebraic*⟩+≡
```
Real_algebraic(const double d):DataMediator(d){
  assert(isfinite(d));
  if(!LocalPolicy::local_creation_succeeded(d)){
    ExpressionDagPolicy::create_rep(d);
  }
};
```

creates a new `Real_algebraic` with value d. We pass the Argument through the *DataMediator* to the *LocalPolicy*, the *ExpressionDagPolicy* is default constructed and initialized only if no local representation is possible. If the return value of `local_creation_succeeded()` is fixed at compile time, the compiler may remove the branch and part of the code.

We have implemented two *LocalPolicy* models that reproduce already existing strategies. The first one is `No_local_data`, following the strategy of `CORE::Expr`. It does not attempt to represent any number locally, all operations are immediately forwarded to the *ExpressionDagPolicy*. The corresponding *DataMediator* `Jobless_mediator` is usable with any *ExpressionDagPolicy* and does nothing.

The *LocalPolicy* `Local_double_with_interval_check` reproduces the behavior of `leda::real`. `Local_double_with_interval_check` uses a single `double` for local representation. Any of the operations $\pm, \cdot, /, \sqrt[d]{}$ is performed with `double` interval arithmetic. If the resulting interval is a singleton the result is locally representable. The corresponding `Local_double_to_expression_dag_mediator` creates a DAG node from the locally stored `double` and is usable with any *ExpressionDagPolicy*.

The motivation for `Local_double_with_interval_check` is as follows. Most input data consists of small 32 bit `int`s or `double`s. Converting such an `int` into a `double` leaves quite some bits unused and it is likely that a couple of additions, subtractions and even multiplications can be performed before the available precision becomes insufficient. For `double` input a similar argument holds but since more bits are used initially, the strategy will be less effective. In Section 3 we present a *LocalPolicy* that is designed to allow some more operations before for `double` input before a DAG must be created. In Section 3.5 we compare some of our local policies with each other as well as `leda::real` and `CORE::Expr`.

3 A LocalPolicy based on sums of doubles

The *LocalPolicy* model `Local_double_sum` represents a number as a sum of `double`s. More precisely it stores a sequence of `double`s representing their sum. Operations that can be performed locally are negation, addition, subtraction and multiplication. Of course also the sign of a sum can be computed. This suffices for most applications in computational geometry. Other arithmetical operations are not supported but directly forwarded to the *ExpressionDagPolicy*. The number of summands currently stored is called the length or actual length of the sum. To avoid dynamic memory management we limit the maximum length of a sum at compile time and perform operations only if the length of the result does not exceed the maximum length. This leads to a tradeoff between the maximum length and the ability to postpone DAG construction. Increasing the maximum length will avoid more DAG constructions but on the other hand increase the size of a `Real_algebraic` handle in memory.

At the core of all operations on sums of `double`s are so called error-free transformations. Error-free transformations transform an arithmetic expression involving floating-point numbers into a mathematically equivalent expression that is more suited for a particular purpose, e.g., sign computation. Let $\oplus$ and $\odot$ denote floating-point addition and multiplication respectively. For example, $a + b$ can

be transformed into $c^{\text{hi}} + c^{\text{lo}}$, such that $a \oplus b = c^{\text{hi}}$ and $a + b = c^{\text{hi}} + c^{\text{lo}}$. Note that c^{lo} is the rounding error involved in computing $a \oplus b$. Efficient algorithms for performing this transformation have been devised for IEEE 754 [19] compliant arithmetic with exact rounding to nearest. TWOSUM(a, b), due to Knuth [13], uses six floating-point additions and subtractions to perform this transformation, FASTTWOSUM(a, b), due to Dekker [7], requires $|a| \geq |b|$, but uses only three operations. The transformations are error-free unless overflow occurs. Analogously, TWOPRODUCT(a, b), due to Veltkamp and Dekker [7] computes floating-point values c^{hi} and c^{lo} with $a \odot b = c^{\text{hi}}$ and $a \cdot b = c^{\text{hi}} + c^{\text{lo}}$. TWOPRODUCT uses 17 floating-point operations and is error-free, unless overflow or underflow occurs.

Error-free transformations allow us to implement the required operations on sums of `doubles`, but are susceptible to the floating-point exceptions overflow and underflow. The sum or difference of two sums of length m and n will have length at most $m + n$, using TWOSUM the product of two sums of length m and n can be transformed into a sum of length at most $2mn$. Both upper bounds hold for any method to compute the sum, difference or product and are attained in general. The length of sums will grow with operations and the maximum length for sums controls a tradeoff between size of a `Real_algebraic` handle and the effectiveness of `Local_double_sum` to postpone DAG construction. With this in mind we identified four orthogonal design decisions which are reflected in four policies that govern the behavior of our implementation.

- *DoubleSumMaxLength* simply provides the maximum length for sums.
- *DoubleSumOperations* provides the operations we can perform with sums, namely addition, subtraction, multiplication, sign computation and compression. A compression transforms a sum into an equivalent sum with smaller or equal length.
- *DoubleSumProtection* provides a way to handle overflow or underflow should it occur in one of the operations.
- *DoubleSumCompression* decides when and how to apply the compression algorithm provided by *DoubleSumOperations*.

We implemented several models for these policies and performed experiments to evaluate them. Section 3.1, Section 3.2 and Section 3.3 are dedicated to the operations policy, the protection policy and the compression policy respectively. In Section 3.4 we show how the policies are combined to a working implementation in the host class `Local_double_sum`. We describe our experiments in in Section 3.5.

3.1 Operations

One of the operations *DoubleSumOperations* must provide is the exact sign computation for a sum of `doubles`. We evaluated the performance of several exact sign of sum algorithms when used to implement geometric predicates [16]. Our first model `Double_sum_plain_operations` is consequently based on winner of this study, the SIGNK algorithm.

SIGNK is based on compensated summation, a well known approach to increase the accuracy of floating-point summation [11]. For $i = 2, \ldots, n$ we compute TWOSUM(a_{i-1}, a_i), replace a_{i-1} by c^{lo} and a_i by c^{hi} and eliminate zeros on the fly. This leaves the value of the sum unchanged but we may end up with fewer, namely n' summands. We call this step a TWOSUM sweep. Then we sum up a_1 to $a_{n'}$ with ordinary floating-point addition to an approximation s and use an error bound by Ogita et al. [18] to verify the sign of s. When the sign can not be verified we re-iterate the algorithm.

The TWOSUM sweep works similar to a bubblesort, sorting increasingly for absolute value. It does however not swap adjacent numbers but places the results of TWOSUM in the correct order. After some iterations we have $a_{i-1} \oplus a_i = a_i$ for $i = 2, \ldots, n$ and the TWOSUM operations will not lead to changes any more. We show that in this this case the error bound suffices to verify the sign and the

algorithm terminates [16]. However even after only a few sweeps the summands with higher indices will tend to be more significant and the error bound is likely to verify the sign.

The remaining operations are designed to support this property: The compression algorithm performs a TWOSUM sweep, addition and subtraction copy and mix the summands of the operands, placing the summands from one operand at the even positions and the summands from the other operand at the odd positions in the new sequence. Mixing is important as it keeps the more significant summands at the top of the sequence, if those summands have different signs, they will cancel out in the first sweep after the operation. Without mixing, the next TWOSUM sweep will basically operate on the same sequence of summands as in the operands and it will probably take much more sweeps to restore a rough order. The multiplication performs TWOPRODUCT for each pair of summands and stores c^{hi} in the top half of the new sequence of summands and c^{lo} in the lower half. The upper bounds $m + n$ and $2mn$ for the length of a sum, difference and product of two sums of length m and n are always attained.

Since the first step of SIGNK is always a TWOSUM sweep, one could perform one such sweep at the end of each arithmetical operation. This would increase the cost for operations, reduce the length of sums and reduce the cost for sign computation. Our other *DoubleSumOperations* model `Double_sum_expansion_zeroelim_operations` follows this idea through. Based on work by Priest [20], Shewchuk [24] has given algorithms to compute with sums of `double`s.

Expansion operations maintain what Shewchuk has called a strongly nonoverlapping expansion. The relevant mantissa of a `double` is the sequence of bits from its lowest nonzero bit to its largest nonzero bit. Two `double`s overlap if their relevant mantissa overlap, when properly aligned, otherwise they are non-overlapping. Two non-overlapping `double`s are adjacent if their relevant mantissa are adjacent. A sequence of `double`s is a strongly nonoverlapping expansion, when the summands are ordered increasingly by absolute value, no two summands overlap, each summand is adjacent to at most one other summand and if two summands are adjacent both are a power of two. Shewchuk allows zero summands anywhere in the sequence, in which case it is not necessarily ordered increasingly by absolute value. We do not allow zeros, unless the only summand is zero.

The structure of an expansion is as if a TWOSUM sweep has been performed on the sum. This is not evident from the definition but can be seen from the algorithms for computing with expansions [24]. They have an even better property: the sign of the sum is always the sign of a_n and can be read off directly. Our implementation is based on code provided by Shewchuk on the web [25], we implemented missing functionality following suggestions from his paper [24, section 2.8].

In our study of exact sign of sum algorithms for geometric predicates [16] we also included a competitor that straightforwardly evaluates predicates using expansions. This approach was not among the best ones. We turned it into an operations policy model nevertheless because it is diametral to plain sum operations and puts an emphasis on keeping the length of sums short. The arithmetical operations perform some work to keep the resulting sums short, which is not done at all with plain sums. Consequently the upper bounds $m + n$ and $2mn$ are rarely attained. Performing a compression this way inside the operations might well be better than performing it outside using a special compression algorithm and lead to a better overall performance.

Considering the design one could break up the operations policy into even smaller parts and have a separate policy for each of the five operations. The SIGNK algorithm can be used with any sum of `double`s and hence any set of arithmetical operations. Splitting the operations policy would simplify experiments combining the SIGNK algorithm with other implementations of the arithmetical operations. On the other hand these policies would not be completely orthogonal. The expansion operations maintain crucial invariants and are only useful in this combination. Increasing the number of policies from four to eight impairs the usability of our design, especially if policies can not be combined freely. Therefore we opted for a single policy for all operations.

3.2 Protection

Since all our operations policies are based on error-free transformations and those are truly error free only if neither overflow nor underflow occurs, we provide a way to handle those errors. According to the IEEE 754 standard we can check for overflow and underflow after the fact. To ensure we can do this all operations must always terminate, even in case of overflow or underflow. Three models are provided:

- `Double_sum_no_protection` does not detect any exceptions.
- `Double_sum_warning_protection` detects overflow and underflow and calls an error handler.
- `Double_sum_restoring_protection` makes a backup copy of any sum that is to be overwritten by an operation. If overflow or underflow are detected after the operation this sum is restored and we return that the operation could not be performed locally. Then a DAG node will be created from the still correct sum.

Restoring protection is actually what we want, overflow or underflow are invisible to the user and do not harm the correctness of our computation. The other models exist mostly to evaluate the cost of protection.

3.3 Compression

The sum and difference of two sums with m and n summands may take $m+n$ summands, the product may even take $2mn$ summands. Thus the actual length of sums grows with every operation. Since the maximum length is limited but we want to delay DAG node creation as long as possible, we try to reduce the actual length using the compression algorithm provided by the operations policy. The compression policy decides when to apply it. We provide four models:

- `Double_sum_no_compression` triggers no compression at all.
- `Double_sum_lazy_compression` compresses the operands of an operation once before the operation takes place. Compression is triggered only if the result might exceed the maximum length.
- `Double_sum_lazy_aggressive_compression` does the same, but keeps on compressing as long as the length decreases.
- `Double_sum_permanent_compression` compresses the result of each operation once.

Since our two models of the operations policy perform different amounts of compression inside the arithmetical operations we expect that different compression policies will be optimal for them.

3.4 Implementation

To obtain a `Local_double_sum` variant one has to first collect a set of policies and pass them as template parameter to the host class `Local_double_sum`. This generates the class hierarchy show in Figure 3 which reflects some dependencies between the policies. `Double_sum_storage` provides the storage for the sum, that is a field of `double`s and an `int` for the length. All other policies have access to the storage. The compression policy calls functions from the operations policy and the protection policy when compressing operands. The host class `Local_double_sum` uses functionality from all its policies, combining them to a working *LocalPolicy*.

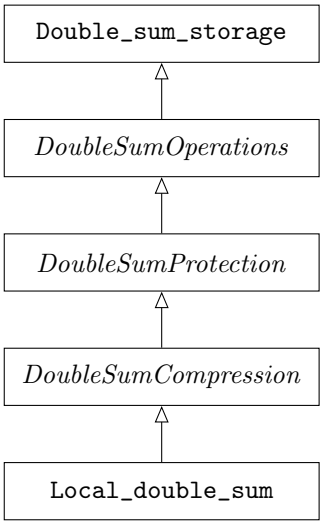


Figure 3: Class hierarchy of double sum policies.

The function `local_multiplication()` is one of the functions required in a local policy. Its implementation in `Local_double_sum` is entirely based on the four policies. The placeholders ⟨*compress operands*⟩ and ⟨*perform protected multiplication*⟩ represent code chunks given further below.

⟨*local multiplication*⟩≡

```
bool local_multiplication(const Local_double_sum& a,
                          const Local_double_sum& b){
  assert(this != &a); assert(this != &b);
  assert(Base::length == 0);
  if(a.length > 0 && b.length > 0){

    ⟨compress operands⟩

    if(predictor(a.length,b.length) <= MaxLength::value){

      ⟨perform protected multiplication⟩

      assert(0 <= Base::length && Base::length <= MaxLength::value);
    }
  }
  return static_cast<bool>(Base::length);
}
```

A length of zero implies that no sum is stored and of course we can perform the multiplication locally only if both operands store a sum. We compress the operands and perform the actual multiplication only if the final result is guaranteed not to exceed the maximum number of summands.

⟨*compress operands*⟩≡

```
multiplication_length_predictor predictor;
Base::compress_operands(a,b,predictor);
```

The class `multiplication_length_predictor` is a model of the `C++` Standard Template Library [2] concept *AdaptableBinaryFunction* and simply computes $f(n, m) = 2mn$. By passing it to the `compress_operands()` function from the compression policy, we can have a single function for multiplication and addition/subtraction, where we pass an *AdaptableBinaryFunction* computing $f(n, m) = m + n$.

⟨*perform protected multiplication*⟩≡

```
typedef typename Base::template
Protector<    Base::Tag_multiplication_needs_protection::value
           || Base::Tag_compress_result_needs_protection::value
         > Protector;

Protector p(*this);
Base::multiplication(a,b);
Base::compress_result();
Base::restore(p);
```

The protection policy provides a nested type `Protector<bool>` where `Protector<false>` does nothing and `Protector<true>` provides protection as described in Section 3.2. In general in any operation provided by the operations policy overflow or underflow may occur, leading to a wrong result. Therefore the operations policy additionally provides a tag, a simple nested type with value `true` or `false`, that indicates whether overflow or underflow may actually occur in an operation. To avoid unnecessary protection we select a protector based on the tags for `multiplication()` and `compress_results()`. For example with plain sum operations and any compression policy except permanent compression, the addition does not need any compression.

Addition and subtraction as well as sign computation are implemented analogously. Negation is always a simple summand wise copy and negation. The other arithmetical operations can not be performed with sums of `double`s and simply return `false`.

3.5 Experiments

`Local_double_sum` is designed to to speed up the evaluation of small polynomial expressions for `double` input within the `Real_algebraic` number type. Therefore we decided to evaluate our implementations using CGAL's [6] Delaunay triangulation algorithm that uses the 2D orientation and incircle predicate. Of those the incircle predicate is arithmetically more demanding. We use CGAL's `Simple_cartesian` kernel. The `Cartesian` kernel uses reference counting and is generally better suited for number types with a larger memory requirement, as our `Real_algebraic` with `Local_double_sum` will be. For the Delaunay triangulation algorithm however there is no difference since it does not copy points or numbers.

In order to force the Delaunay triangulation algorithm to perform more difficult incircle tests we generate test data that contains points almost on a circle with no other points in its interior: First, we create a set $\mathcal{D}$ of disks with a random radius and place a certain percentage f of the points (almost) on the boundary of their union, $\mathrm{bd}(\cup\mathcal{D})$. Next, the remaining points are generated uniformly in the complement of the disks. All points are generated inside the unit circle. In order to get nearly degenerate point sets we use exact arithmetic to compute a point on a circular arc of $\mathrm{bd}(\cup\mathcal{D})$ and then round it to a nearby floating-point point closest to the circular arc. For $f \in \{0\%, 25\%, 50\%, 75\%\}$ we generate 25 point sets with 5000 points each and measure the average running time. Sample input sets are shown in Figure 4.

We ran experiments on two platforms. A notebook with an Intel Core 2 Duo T5500 processor with 1.66 Ghz, using `g++ 4.3.2`, CGAL 3.3.1 and LEDA 5.2. and a Sun Blade Station 1000 with 0.9 Ghz, using `g++ 3.4.4`, CGAL 3.3.1 and LEDA 6.2. When ranking different variants of `Local_double_sum` by measured running time their relative order was invariant with respect to f and with respect to the platform. In fact the measured running time itself is nearly invariant with respect to f. For two variants this can be seen in Figure 8. Hence all other figures show only results from the Intel platform and for $f = 25\%$.

For `Local_double_sum` we have four policies each with at least two models, so the parameter space to search for an optimal variant is rather large. We ran some initial experiments fixing three policies and varying the fourth to determine a good choice for this policy. Then we replaced the previously

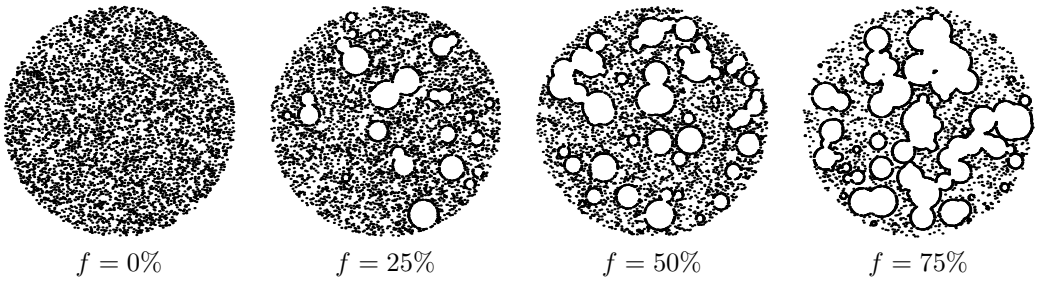


Figure 4: sample input data sets

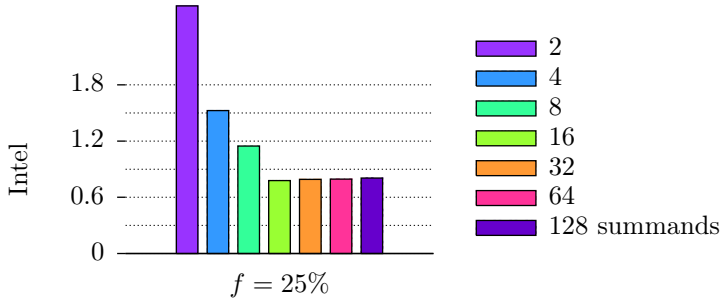


Figure 5: plain sum, restoring protection, lazy aggressive compression

fixed policies by choices that performed well and repeated the experiments, finally resulting in those experiments shown here. To complete the `Real_algebraic` we chose `Original_leda_expression_dags` as *ExpressionDagPolicy* an *ApproximationPolicy* based on `leda::bigfloat`, a *FilterPolicy* based on `leda::interval` and `Bfmss2_separation_bound` as *SeparationBound*, confer Section 4. The expression DAG computation of our `Real_algebraic` therefore comes close to `leda::real`. As *DataMediator* we use `Local_double_sum_to_expression_dag_mediator`. It computes the sum exactly using the arbitrary precision software floating-point type from *ApproximationPolicy* and creates a node from the result. If the sums consists of only a single `double` it directly creates a DAG node for this `double`.

First we were interested in the impact of the maximum length of a sum on the performance. We fixed policies to use a plain sum, restoring protection and lazy aggressive compression. We varied the maximum length from 2 to 128. Looking at Figure 5 it can be seen that with an increasing maximum length, the computation time decreases, reaches a minimum at 16 summands and then remains constant. We conjecture that at this point no DAG nodes are created at all and all work is done by the *LocalPolicy*, at least for the majority of predicate evaluations. It is quite surprising that the minimum is already attained at such a small number of summands. Considering the growth of sums, the incircle predicate as implemented by CGAL, may require up to 1152 summands for the final result. We observed already in our study of exact sign of sum algorithms [16] that often 96 summand suffice when eliminating zeros in the first stage of the predicate. Compressing sums and eliminating zeros in later stages obviously reduces the number of summands required even more.

Next we evaluate the cost of protecting against overflow and underflow. Since both models of the operations policy require protection for different operations, we use the expansion operations with lazy compression and 16 summands and the plain sum operations with lazy aggressive compression and 16 summands and combine both sets with each of the protection policies. Figure 6 shows the results of our experiments. Just resetting and checking the floating-point exception flags increases the running time for plain sums by a factor of two and by a factor of four for expansions. The additional cost

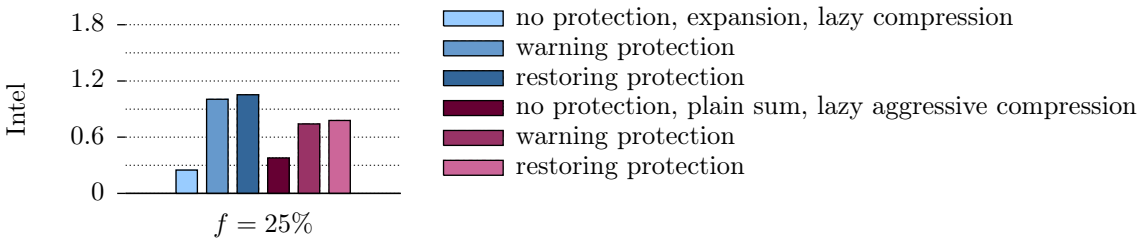


Figure 6: 16 summands, two combinations of operations and compression

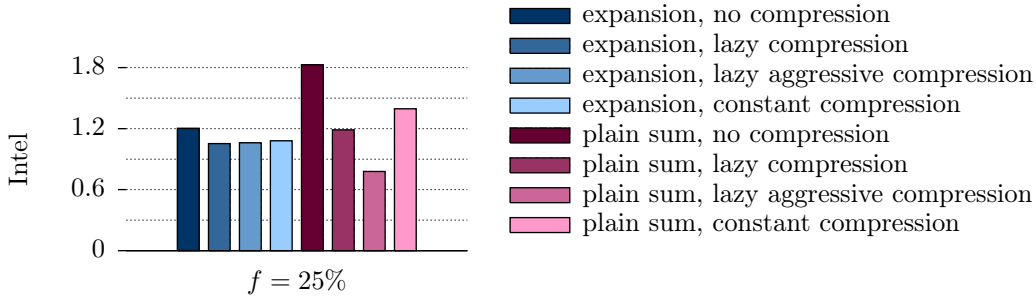


Figure 7: 16 summands, restoring protection

for making a backup copy in the restoring protection policy is negligible compared to this increase. For plain sums addition and subtraction only copy summands, and hence need no protection against overflow and underflow. For expansions only the sign computation needs no protection. Addition and subtraction are however used more often, so for expansions the floating-point exceptions are checked more often. The result is, that while expansions are the better choice without effective protection they are slower than plain sums when protection is used.

Next we evaluate the effect of different compression policies, again with both operation policies. We combine plain sum operations with a maximum length of 16 summands and restoring protection as well as expansion operations with a maximum length of 16 summands and restoring protection. Both sets are combined with each compression policy. Figure 7 shows the results.

No kind of compression is performed inside plain sum operations, so all policies that trigger additional compression increase the performance of plain sums. The permanent compression policy, however, triggers a compression directly after each operation. Then addition and subtraction must be protected which is very expensive as we have seen before. The best result is provided by lazy aggressive compression, the running time is nearly halved compared to no compression. The arithmetic operations on expansion perform some compression themselves. This is quite effective: as can be seen the running time decreases only slightly when performing additional compression. There is not much difference between lazy, lazy aggressive and permanent compression.

Finally we compare `Local_double_sum` with other approaches. We choose the best variant based on plain sums and the best variant based on expansions with restoring protection. As competitors we choose `No_local_data` and `Local_double_with interval_check` introduced in Section 2. Those two local policies resemble the behavior of `CORE::Expr` and `leda::real` respectively, which we also

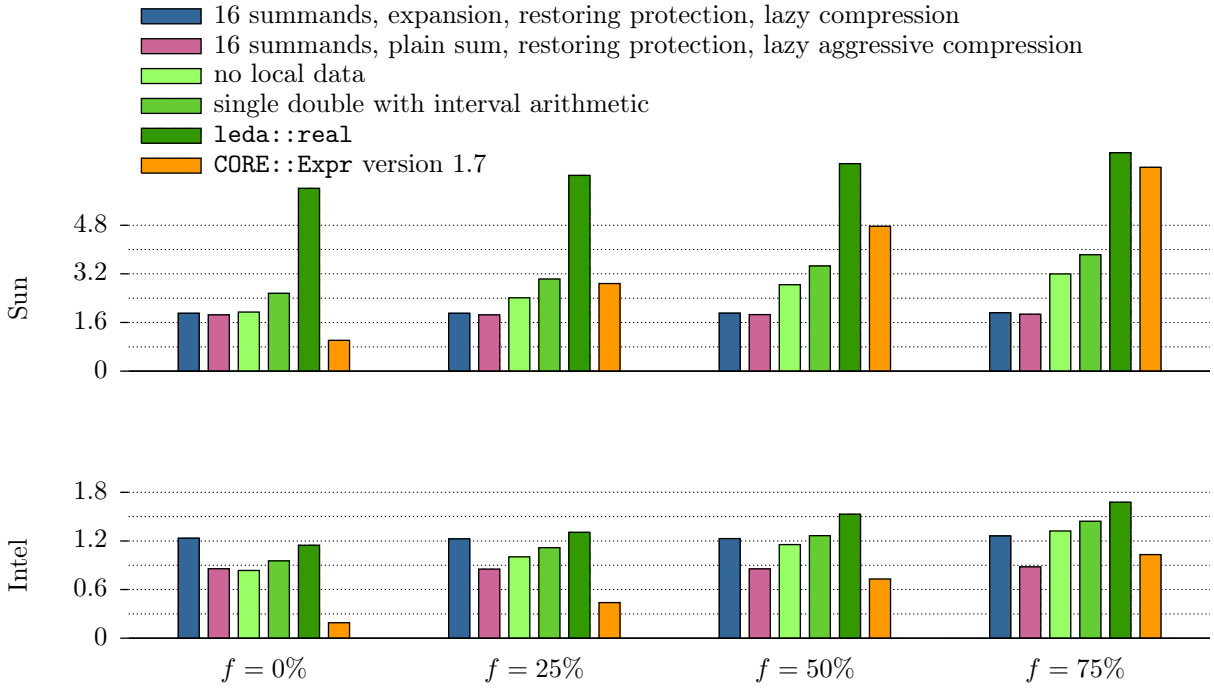


Figure 8: comparison with other approaches

add to the set of competitors. Note that `CORE::Expr` uses MPFR [17] for arbitrary precision floating-point computations while `leda::real` and our implementations use `leda::bigfloat`. Furthermore the expression DAG computations of all our variants closely resemble the behavior of `leda::real`. The results are shown in Figure 8.

Unlike the other competitors, both `Local_double_sum` variants are invariant to the amount of nearly degenerate predicate evaluations. Therefore they are a good choice for input sets where many such evaluations occur but less favorable for randomly distributed input. Compared to the other approaches `CORE::Expr` is very fast for randomly distributed input, especially on the Intel platform. It is in fact so fast that none of our local policies could improve the performance of `CORE::Expr` for this type of input.

Storing no local data is also always a bit better than storing a single `double` and using interval arithmetic to check the exactness of operations. We already hinted at the end of Section 2 that the latter strategy is more suitable for small precision integer input than `double`s. But this strategy, together with the other policies we use for our experiments form a `Real_algebraic` variant that comes very close to `leda::real`. The experiments show that no performance is lost by our modularization efforts. `leda::real` performs especially bad on the Sun platform and we can only guess the causes.

Going back to `Local_double_sum` most of the running time is spend on protecting against overflow and underflow, see again Figure 6. Without effective protection `Local_double_sum` makes `Real_algebraic` competitive to `CORE::Expr` for randomly distributed input and clearly superior for input with many nearly degenerate predicate evaluations. The goal therefore must be to reduce the cost for protection. One possible way is to base an operations policy on ESSA, another algorithm for exactly computing the sign of a sum of `double`s [21] that we also examine in our study [16]. Although it was shown not to be competitive there, ESSA is immune to overflow and underflow and can be used to implement an operations policy where only the multiplication must be protected. Another approach is to avoid overflow or underflow in the first place. This requires to check the operands before each operation. If some summands are so large or small that floating-point exceptions might occur we simply return that no local operation is possible and create a DAG node. What "large" or

"small" means depends however on the operation so these checks must occur in the operations policy, rendering the the protection policy obsolete. In the light of our experiments above this might well be faster than checking for floating-point exceptions after the fact. Altogether basing a *LocalPolicy* on sums of `double`s is a promising approach whose full potential has yet to be exploited.

4 ExpressionDagPolicy

The class `Real_algebraic`, among other things, acts as a handle to a DAG node. The handle part, as well as all operations on the DAG are inherited from the *ExpressionDagPolicy* and then used to implement the arithmetical operations and comparison operators, confer Section 2 where some code from `Real_algebraic` is shown. Some of the functions provided by the expression DAG policy are

⟨*ExpressionDagPolicy*⟩≡
```
bool expression_dag_is_initialized();
```

returns whether a representation as DAG node is available.

⟨*ExpressionDagPolicy*⟩+≡
```
void expression_dag_multiplication(const ExpressionDagPolicy a,
                                   const ExpressionDagPolicy b);
```

turns the *ExpressionDagPolicy* into a handle to a DAG node that represents the product of a and b. Both a and b must already be represented as DAG node. This function allows a variety of implementations, which may include restructuring the DAG.

⟨*ExpressionDagPolicy*⟩+≡
```
void expression_dag_sign(int& s);
```

sets s to the sign of the number represented by the DAG node. A DAG node representation must be available.

These three functions represent the two tasks of the expression DAG policy: creating nodes representing operations and sign computation. We provide a single model for *ExpressionDagPolicy* called `Original_leda_expression_dags` only. It comes with an associated class `sdag_node` that actually implements a DAG node and the algorithm for sign computation.

We describe how `Original_leda_expression_dags` implements the sign computation algorithm that we sketched in the introduction. Sign computation actually starts when a node v is created. Each node stores a dynamic hardware floating-point filter. This filter represents an interval I' containing v and allows to compute the filter for a new node from the operands. This is done every time a new node is created. Note that when dividing by a node whose interval contains zero, the resulting interval must represent the whole real line and will be meaningless. There are other cases that render the filter meaningless too. Since there are many ways to actually implement such a filter we made it exchangeable as *FilterPolicy*.

When the sign of a node v is requested, first the already available interval I' is checked. If zero is not contained in I' the sign of v is known. Otherwise the algorithm needs for each node z below v an upper bound on $|z|$, and if z is a divisor or radicand in some operation also a positive lower bound on $|z|$.

To compute these bounds the DAG below v is traversed by a depth-first-search, processing in postorder. For each node z a software floating-point approximation $\hat{z}$ and an absolute error $\mathtt{e}_z$ are computed and stored in the node, such that $|z - \hat{z}| \leq \mathtt{e}_z$. For example in case of a multiplication $z = x \cdot y$, we compute $\hat{z} = \hat{x} \odot \hat{y}$ with p bit precision and in round-to-nearest. The relative error of this multiplication is 2^{-p}. The initial computation is done with $p = 53$ bits, so the approximation is about as accurate as the one from the floating-point filter. The error $\mathtt{e}_z$ can be computed using the following estimate [5], where $y_{\text{high}} = |\hat{y}| + \mathtt{e}_y$ is an upper bound on $|y|$.

$$\begin{aligned} \mathtt{e}_z &= |\hat{z} - z| \\ &\leq |\hat{z} - \hat{x}\hat{y}| + |\hat{x} \cdot (\hat{y} - y)| + |y \cdot (\hat{x} - x)| \\ &\leq 2^{-p}|\hat{x}\hat{y}| + |\hat{x}|\mathtt{e}_y + y_{\text{high}}\mathtt{e}_x \end{aligned} \tag{1}$$

An upper bound of the right hand side is computed using low precision and directed rounding modes. If z is a divisor or radicand, the whole sign algorithm is applied recursively to z, resulting in an approximation and error that are good enough to provide a positive lower bound on $|z|$. Apart from the fact that the software floating-point arithmetic will not overflow or at least has a much larger exponent range than `double`, these are the cases where the new approximation is significantly better than the floating-point filter.

When this initial step has been done for the whole subDAG below v it results in an interval $I = [\hat{v} - \mathtt{e}_v, \hat{v} + \mathtt{e}_v]$ containing v and it is checked whether I contains zero. If this is still the case precision driven arithmetic is used to improve the interval I for v. Precision driven arithmetic allows to prescribe the error $\mathtt{e}_z$ of an approximation $\hat{z}$ before it is computed. The procedure works recursively and requires to recompute the approximations of the children x and y with some prescribed error, it stops at the leafs of the DAG where the error is always zero. Assume $z = x \cdot y$ and that we want to guarantee $\mathtt{e}_z < B$. Looking at Equation (1) this can be done in the following way: First recursively recompute $\hat{x}$, enforcing $\mathtt{e}_x < B/4y_{\text{high}}$, then recompute $\hat{y}$, enforcing $\mathtt{e}_y < B/4|\hat{x}|$ and finally compute $\hat{z}$ from $\hat{x}$ and $\hat{y}$ with p bit precision in round-to-nearest, such that $2^{-p} < B/2|\hat{x}\hat{y}|$. Note how the usage of $|\hat{x}|$ to bound $\mathtt{e}_y$ makes it mandatory to first recompute $\hat{x}$. This is the case for all binary operations: the error bound implies an order in which the children must be recomputed. When $\hat{z}$ has been computed, $\mathtt{e}_z$ is simply set to either B or in case $\mathtt{e}_x$ and $\mathtt{e}_y$ are zero and the recomputation of $\hat{z}$ from $\hat{x}$ and $\hat{y}$ is exact, to zero. Many implementations of software floating-point arithmetic exist that can be used in the sign computation algorithm. Therefore we made the arithmetic exchangeable as *ApproximationPolicy*. We discuss some details of the interface in Section 4.2.

Precision driven arithmetic not only allows to compute arbitrarily accurate approximations of v, but also to prescribe the accuracy before the computation is started. Starting with the initially computed error $\mathtt{e}_v$, the approximation $\hat{v}$ is recomputed iteratively such that in the i-th iteration $\mathtt{e}_v$ is decreased by a factor of $2^{-27 \cdot 2^i}$. Asymptotically, the number of correct bits in $\hat{v}$ is doubled with every iteration. This does not suffice to compute the sign of v in case v is actually zero. We use a separation bound $\text{sep}(v)$ to detect those cases. The iteration stops when $2\mathtt{e}_v < \text{sep}(v)$. The sign computation algorithm is adaptive, by computing several approximations of increasing quality it will terminate quickly if v is far from zero. If v is actually zero the worst case running time is attained which strongly depends on the separation bound. Many separation bounds are known and for several of them there exist classes of expressions where they are the best known separation bound. Therefore we also made the separation bound exchangeable as *SeparationBound*.

Not only is `Original_leda_expression_dags` parameterized by *ApproximationPolicy*, *SeparationBound* and *FilterPolicy*, but these three policies provide basic tools required in any algorithm for adaptive sign computation on expression dags.

4.1 FilterPolicy

The *FilterPolicy* is not actually required for sign computation but it can speed it up for easy cases, adding to the adaptivity of the sign computation. The interface provided by *FilterPolicy* is one for hardware floating-point interval arithmetic. An object represents an interval, the arithmetic operations $\pm, \cdot, /, \sqrt[d]{\ }$ are available as well as several auxiliary functions. As of now *FilterPolicy* provides the same interface as `leda::interval`. LEDA provides three different implementations of interval arithmetic having this interface. Other models could be implemented based on the BOOST interval library [3]. The dynamic floating-point filter from [5] does not need specific rounding modes

to be set and hence might be faster to compute at the expense of less precise intervals. This is the filter used in `CORE::Expr`.

4.2 ApproximationPolicy

The approximation policy provides a software floating-point type *Approximation* as well as approximate arithmetical operations on this type. Furthermore it provides associated types *Exponent*, used to represent the exponent stored with an approximation, *Precision* representing the number of bits of an approximation and *RoundingMode* representing rounding modes. We provide two models `Mpfr_approximation_policy` based on MPFR [17] and `Leda_approximation_policy` based on `leda::bigfloat` [14]. The arithmetic operations have the following interface:

⟨*ApproximationPolicy*⟩≡
```
static bool mul(Approximation& c,
                const Approximation& a,
                const Approximation& b,
                const Precision p,
                const RoundingMode rm);
```

computes $a \cdot b$, rounds the result to at least p bits according to the rounding mode and stores it in c. Returns `true` if no rounding occurred, that is if $c = a \cdot b$. a, b and c may refer to the same variable.

With the exception that we do not require that the result is rounded to exactly p bits, this is probably the most natural interface for arbitrary precision floating-point arithmetic, but `leda::bigfloat` and MPFR provide a different interface. `leda::bigfloat` returns the result, MPFR writes it into a variable passed by reference. More importantly however arithmetic operations in MPFR lack the argument for the precision of the operation. Instead each variable has an inherent precision and the current precision of the result variable is used for the operation.

Both `leda::bigfloat` and MPFR allow the result variable to equal one or both of the argument variables. This allows to reduce the number of auxiliary variables in many cases, saving some memory management. The MPFR interface however prevents operations where the result and one of the operand variables are the same and the result precision is different from the operand precision. One can increase the operand precision and therefore the result precision before the operation takes place, but reducing it is not possible without creating an auxiliary variable. For this reason we allow the result to be rounded to at least p bits and compute with higher precision in this case.

Next to the arithmetic operations, there are several auxiliary functions, e.g., to reduce the precision of an approximation by rounding, to return the sign of an approximation or to convert an approximation into a `double`. The binary logarithm is frequently used in `Original_leda_expression_dags`, e.g., to avoid expensive operations in the error computation.

⟨*ApproximationPolicy*⟩+≡
```
static Exponent floor_log2(const Approximation& a);
static Exponent  ceil_log2(const Approximation& a);
```

return an integer lower and upper bound on $\log_2 |a|$, for $a \neq 0$, respectively. Both bounds are almost optimal in that they may differ by at most one.

Depending on the normal form used for a software floating-point type, one of the bounds can be computed optimally more efficiently. If the mantissa is in $[1, 2)$, simply return the stored exponent as an optimal upper bound, if the mantissa is an integer, return the stored exponent plus the precision of the mantissa as an optimal lower bound. Computing the other bound optimally requires to check if the mantissa has exactly one nonzero bit. Our interface allows to compute one bound from the other, with non-optimal results only if the number is a power of two.

4.3 SeparationBound

The term "separation bound" is used for the conditional lower bound sep(v) for a node v but it also for the algorithm used to compute sep(v). In the latter sense known separation bounds consist of a set of parameters for a node and a set of rules, how to update the parameters from a nodes children. Our *SeparationBound* policy is a class that stores these parameters and implements the rules.

⟨*SeparationBound*⟩≡
```
void set(const double x);
```

initializes the parameters for a node that represents x, i.e., a leaf node in the DAG.

⟨*SeparationBound*⟩+≡
```
void multiplication(const SeparationBound& a,
                    const SeparationBound& b,
                    const Parameter& D);
```

computes and stores parameters for this node from the parameters of a and b using the rules for multiplication. D must be an upper bound on the algebraic degree of the node.

⟨*SeparationBound*⟩+≡
```
Parameter bound(const Parameter& D)const;
```

returns the actual separation bound sep(v) for this node. D must be an upper bound on the algebraic degree of the node.

A *SeparationBound* does not implement the traversal algorithm required to compute the bound, it must be provided by the class utilizing it. The algebraic degree is a quantity that plays an important role in many separation bounds, unfortunately in the presence of common subexpressions computing it recursively for each node from its children leads to weak degree bounds. Therefore it must be provided to the separation bound. We provide several models that implement different bounds known from the literature: `Bfmss2_separation_bound` [22], `DM_separation_bound`, `LY_separation_bound` [15] and `Sekigawa_separation_bound` [23].

5 Future Improvements

One of the goals of our research is to improve the efficiency of expression DAG based number types. The main working point that still has to be addressed is the sign computation algorithm. The algorithm in `Original_leda_expression_dags` closely resembles the one from `leda::real`. In `CORE::Expr` a different although similar algorithm is implemented [4]. Precision driven arithmetic requires that some data is available from the child nodes, e.g., in Equation (1) an upper bound on $|y|$ is required. `Original_leda_expression_dags` performs an initial step to ensure this data is available. In `CORE::Expr` this data is is requested from the child node as it is needed. Then the child node has several options to provide the data e.g., from the floating-point filter or from the current approximation but ultimately must trigger more computations on its subDAG. For example the sign function of a multiplication node simply returns the product of the signs of its children. Altogether in `CORE::Expr` there are six functions, returning the sign, an upper and lower bound, guiding the precision driven arithmetic and approximating a node with a prescribed absolute or relative error that all may call each other recursively. This approach tries to reduce the amount of floating-point operations, however at the cost of more DAG traversal. An optimal implementation of sign computation is as of yet unknown.

5.1 Sign Computation and Common Subexpressions

One idea to improve the efficiency of sign computation is to restructure the DAG before starting to compute. Higham [9, 10] analyzes several summation methods for fixed precision floating-point arithmetic. Some of those methods use a static summation order while others are dynamic i.e.,

they depend on intermediate results. Higham examines the accuracy of those methods and makes suggestions which methods to use. When computing with precision driven arithmetic, the accuracy of any intermediate result is fixed before it is computed. Our hope is that those methods which lead to high accuracy when using a fixed precision will lead to small precision requirement in case of fixed accuracy.

When computing a sum inside a program with a simple loop, the expression DAG will take the form of a linear list. Restructuring the sum by simple application of associativity or commutativity can lead to a more balanced DAG or a DAG where the intermediate nodes have a small absolute value, resembling the methods considered by Higham. We made some initial experiments by checking the identity for the geometric sum

$$\sum_{i=0}^{n-1} r^i = \frac{1-r^n}{1-r}. \tag{2}$$

We manually rearranged the sum on the left hand side and soon observed large differences in the running time. Finally we reduced the phenomenon to the following two functions.

⟨*geometric sum*⟩≡
```
template <class NT>
void geometric_sum_slow(const int n){
  NT s = 0;
  NT r(1.2398793486823876843);
  NT ri = 1;
  for(int i=0;i<n;i++){
    s  = s + ri;
    ri = ri * r;
  }

  NT sprime = ( NT(1)-ri )/( NT(1)-r );
  assert(s == sprime);
}
```

⟨*geometric sum*⟩+≡
```
template <class NT>
void geometric_sum_fast(const int n){
  NT s = 0;
  NT r(1.2398793486823876843);
  NT ri = 1;
  for(int i=0;i<n;i++){
    s  = s + ri;
    ri = ri * r;
  }

  NT sprime = ( NT(1)-ri )/( NT(1)-r );
  assert(sprime == s);
}
```

Note that they differ in the last line only! In one DAG s is the left child of the root node and s' is the right child, while its vice versa in the other DAG. Here are some measured running times for those functions when called with $n = 64$.

```
geometric_sum_fast(64) ... (0.02)
geometric_sum_slow(64) ... (0.14)
```

Our design allows us to easily measure the amount of work being done by the software floating-point arithmetic. The approximation policy `Approximation_policy_statistics` wraps around any other approximation policy and keeps a record of all calls. To a multiplication of two numbers with precision p_1 and p_2, rounding the result to precision p_3, we assigned the cost $c \cdot (p_1 + p_2) \log(p_1 + p_1)$. The histogram in Figure 9 shows how often a multiplication with a certain cost is performed for both of the functions above and $n = 64$.

The effect that the cost for some sign computation changes drastically with small code changes makes the behavior of `Real_algebraic` unpredictable. We would like to attain the smaller running time for either way to assert Equation (2).

To investigate the cause for the difference we recall how precision driven arithmetic in works. To recompute the approximation $\hat{z}$ of some node z with prescribed absolute error $\mathtt{e}_z$, first the approximation of its right child x must be recomputed with some prescribed error $\mathtt{e}_x$, then the left child y is handled. This is the source of the difference in running times. Consider some node u which is a descendant of both x and y. The algorithm reaches u for the first time coming from x and recomputes $\hat{u}$ with some error $\mathtt{e}_u$. Later it may arrive at u again, coming from y. Now an approximation with error $\mathtt{e}'_u$ is requested. If $\mathtt{e}_u < \mathtt{e}'_u$ then $\hat{u}$ can be used as is, but if not we need to recompute $\hat{u}$ and all its descendants! This is what occurs in `geometric_sum_slow()` while in `geometric_sum_fast()`

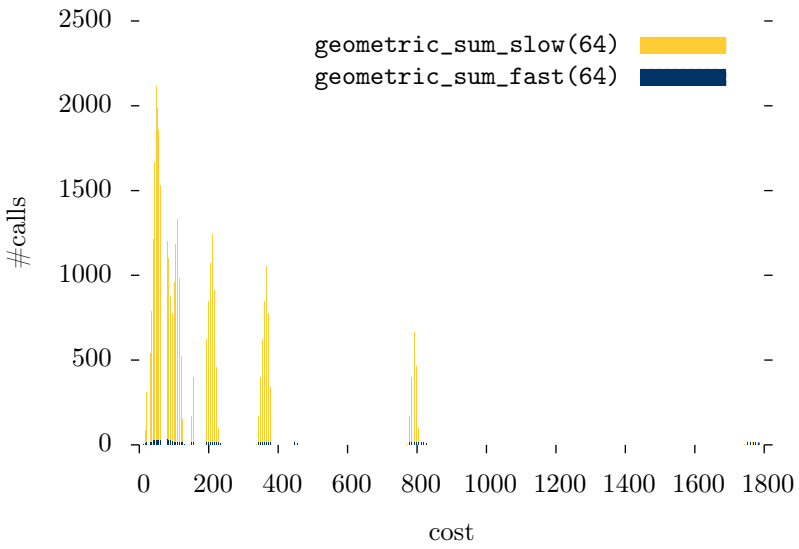


Figure 9: Cost for all floating-point multiplications arising from checking Equation (2).

we are lucky and compute common subnodes at the first time with sufficient accuracy for all later requests.

While commutativity allows to interchange x and y and hence recompute either $\hat{x}$ or $\hat{y}$ first, it is not clear how to choose. Neither is clear if the choice can be made in a way that avoids recomputation for any node in the DAG. Instead the solution is to refine Equation (1):

$$\begin{aligned} \mathtt{e}_z &\leq 2^{-p}|\hat{x}\hat{y}| + |\hat{x}|\mathtt{e}_y + y_{\text{high}}\mathtt{e}_x \\ &\leq 2^{-p}|\hat{x}\hat{y}| + (|\hat{x}| + \mathtt{e}_x)\mathtt{e}_y + y_{\text{high}}\mathtt{e}_x \\ &= 2^{-p}|\hat{x}\hat{y}| + x_{\text{high}}\mathtt{e}_y + y_{\text{high}}\mathtt{e}_x + \mathtt{e}_x\mathtt{e}_y \end{aligned} \tag{3}$$

Now we can select $\mathtt{e}_x$ and $\mathtt{e}_y$ such that $\mathtt{e}_x < B/4y_{\text{high}}$, $\mathtt{e}_y < B/4x_{\text{high}}$, and $\mathtt{e}_x\mathtt{e}_y < B/4$. Then we can recompute $\hat{x}$ and $\hat{y}$. This can now be done in parallel or at a later time, i.e., after other parents of x and y have registered their accuracy requirement on x and y. Finally we recompute $\hat{z}$ with precision p such that $2^{-p} < B/4|\hat{x}\hat{y}|$ to ensure $\mathtt{e}_z < B$.

Yap [27] has given estimates similar to Equation (3) for the operations $\pm, \cdot, /, \sqrt{}$, assuming $\pm, \cdot$ are performed exactly. It is not hard to generalize them to approximate arithmetic and derive an estimate for $\sqrt[d]{}$. For / and $\sqrt[d]{}$ lower bounds on the denominator and radicand are needed, for these nodes we can not avoid to compute their sign in the initialization step as before. A node v can be approximated with some prescribed error as follows. First all descendants u of v are visited in topological order, propagating the accuracy requirement downwards. Then they are visited in reverse topological order and $\hat{u}$ is recomputed if necessary.

Compared to the original approach in `Original_leda_expression_dags` this algorithm minimizes the number of node reevaluations. There is only a difference however if there are common subexpressions and the DAG is not actually a tree. Furthermore there is a certain overhead for computing the topological order of the nodes. Therefore experimentation is needed to evaluate both approaches. But even if the DAG is a tree, this scheme allows to handle several nodes in parallel. This is another approach that should be followed and evaluated.

5.2 Modularization of the sign computation algorithm

To facilitate the implementation of several similar or totally unrelated sign computation algorithms we would like to modularize it more. We already made the basic tools for such an algorithm exchangeable, but there are still many options in the algorithm itself. Some of them affect the data representation in

a node. The new sign computation scheme from the previous section requires a field in each node that stores the accuracy requirement for a node. The algorithm from `Original_leda_expression_dags` does not require such a field since a node is directly recomputed when the accuracy requirement changes. Both algorithms employ different traversal strategies. The algorithm used in `CORE::Expr` is again totally different. On the other hand all three algorithms can use the same set of basic data inside a node and the same error estimate Equation (3). We intend to modularize the sign computation in the following way. First algorithms are separated from the DAG implementation. The DAG implementation is only used to store data and provides some basic functions to access and manipulate the stored data.

All algorithms traverse the DAG in some way and perform operations on each visited node and in some cases its children. To make algorithms or parts of them reusable we intend to implement them by means of visitors [8] that are applied to each node of a DAG by some traversal algorithm. A policy to compute a separation bound then would consist of a set of parameters that must be added as data fields to the DAG node, a traversal algorithm, that might as well need additional data in the DAG node and a set of visitors that compute the parameters for each node type. A visitor may use the default interface of a node as well as the data fields / interface that was added to make the visitor applicable. This kind of modularization separates the traversal algorithms from the operations that are performed on a node, making them reusable but at the cost of higher code complexity.

6 Conclusion

Our new flexible allows one to easily create different instantiations of `Real_algebraic` and to compare them experimentally. We demonstrate the feasibility of our design by implementing multiple models for most of the concepts arising in our design. Experiments show that we achieve this flexibility at no extra cost: `Real_algebraic` is as efficient as previous, less flexible approaches when assembled analogously to its static predecessors. With `Local_double_sum` implemented and evaluated several new strategies to increase efficiency by postponing DAG creation. Our new approaches are promising yet not fully exploited. Future work will address expression DAG evaluation algorithms.

References

[1] A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[2] M. H. Austern. *Generic Programming and the STL.* Addison-Wesley, 1998.

[3] boost `C++` Libraries. `http://www.boost.org/`, 2009. Version 1.40.0.

[4] H. Brönnimann, Z. Du, S. Pion, and J. Yu. Transcendental and algebraic computation made easy: Redesign of core library. `http://www.cs.nyu.edu/exact/doc/easyCore.pdf`, December 2006.

[5] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Efficient exact geometric computation made easy. In *15th ACM Symposium on Computational Geometry (SCG'99)*, pages 341–350, New York, NY, USA, 1999. ACM.

[6] CGAL: Computational Geometry Algorithms Library. `http://www.cgal.org/`, 2009. Version 3.4.

[7] T. J. Dekker. A floating-point technique for extending the available precision. *Num. Math.*, 18(2):224–242, 1971.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] N. J. Higham. The accuracy of floating point summation. *SIAM J. Sci. Comput*, 14(4):783–799, July 1993.

[10] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2. edition, 2002.

[11] W. Kahan. Further remarks on reducing truncation errors. *Comm. of the ACM*, 8(1):40, 1965.

[12] V. Karamcheti, C. Li, I. Pechtchanski, and C.-K. 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.

[13] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art Of Computer Programming*. Addison-Wesley, 3. edition, 1997.

[14] LEDA: Library of Efficient Data Structures and Algorithms. http://www.algorithmic-solutions.com/, 2009. Version 6.2.1.

[15] C. Li and C. Yap. A new constructive root bound for algebraic expressions. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 496–505, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[16] M. Mörig and S. Schirra. On the design and performance of reliable geometric predicates using error-free transformations and exact sign of sum algorithms. In *19th Canadian Conference on Computational Geometry (CCCG'07)*, pages 45–48, August 2007.

[17] MPFR: A multiple precision floating-point library. http://www.mpfr.org/, 2009. Version 2.4.1.

[18] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988, 2005.

[19] M. L. Overton. *Numerical Computing with IEEE Floating-Point Arithmetic*. SIAM, 2001.

[20] D. M. Priest. *On properties of floating point arithmetics: numerical stability and the cost of accurate computations*. PhD thesis, University of California at Berkeley, 1992.

[21] H. Ratschek and J. G. Rokne. Exact computation of the sign of a finite sum. *Appl. Math. Computation*, 99(2-3):99–127, 1999.

[22] S. Schmitt. Improved separation bounds for the diamond operator. Report ECG-TR-363108-01, Effective Computational Geometry for Curves and Surfaces, Sophia Antipolis, FRANCE, 2004.

[23] H. Sekigawa. Using interval computation with the mahler measure for zero determination of algebraic numbers. *Josai University Information Sciences Research*, 9(1):83–99, 1998.

[24] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete and Computational Geometry*, 18(3):305–363, 1997.

[25] J. R. Shewchuk. Companion web page to [24], 1997. http://www.cs.cmu.edu/~quake/robust.html.

[26] C.-K. Yap. Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7(1-2):3–23, 1997.

[27] C.-K. Yap. *On Guaranteed Accuracy Computation*, chapter 12, pages 322–373. World Scientific, 2004.