Algorithm Engineering for Expression Dag Based Number Types

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von Diplom-Mathematiker Marc Andreas Mörig

geb. am 19.08.1979 in Magdeburg

Gutachterinnen/Gutachter

Prof. Dr. Stefan Schirra
Prof. Dr. Chee K. Yap
Prof. Dr. Stefan Funke

Magdeburg, den 29.01.2015

$$\frac{12 + 144 + 20 + 3\sqrt{4}}{7} + 5 \times 11 = 9^2 + 0$$

A dozen, a gross, and a score
Plus three times the square root of four
Divided by seven
Plus five times eleven
Is nine squared and not a bit more.

Leigh Mercer [7]

Zusammenfassung. Geometrische Algorithmen werden für das Real RAM Modell entworfen, welches exakte Arithmetik mit reellen Zahlen in konstanter Zeit erlaubt. Wird dieses Modell durch approximative Hardwarearithmetik ersetzt, so kommt es zu Robustheitsproblemen. Implementierungen stürzen ab, berechnen falsche Ergebnisse, oder terminieren nicht. Ein direkter Ansatz diese Probleme zu umgehen ist exakte Arithmetik zu verwenden. Zahltypen mit beliebiger Präzision können numerische Berechnungen exakt oder mit sehr großer Genauigkeit durchführen, jedoch um den Preis einer deutlich höheren Laufzeit. Für geometrische Algorithmen genügt jedoch eine korrekte Vorzeichenberechnung.

Auf Ausdrucksbäumen basierende Zahltypen speichern arithmetische Ausdrücke in Form eines gerichteten azyklischen Graphens. Dies erlaubt es ihnen, einen Ausdruck mehrfach auszuwerten und so oftmals die Kosten hoher Präzision zu umgehen. Sie sind ein sehr allgemeines und benutzerfreundliches Werkzeug um robuste Implementierungen geometrischer Algorithmen zu erstellen. Wir berichten von unseren Versuchen die Effizienz dieses Ansatzes mit Hilfe von Algorithm Engineering zu verbessern. Unser Ziel ist es, uns den Laufzeiten von spezialisierten Lösungen anzunähern, ohne die Benutzerfreundlichkeit oder Allgemeingültigkeit zu opfern.

Wir leisten folgende Beiträge. Erstens stellen wir ein neues Design für auf Ausdrucksbäumen basierende Zahltypen vor. Unser Design nutzt C++ Templates, um wesentliche Komponenten der Implementierung austauschbar zu machen. Dabei identifizieren und separieren wir auch solche Komponenten die in anderen Zahltypen nicht austauschbar sind. Für alle Komponenten stellen wir verschiedene Implementierungen bereit und zeigen so die Realisierbarkeit unseres Designs. Mit Hilfe von Experimenten vergleichen wir verschiedene Optionen und untersuchen ihren Einfluss auf die Gesamteffizienz. Dabei betrachten wir nicht nur Systemressourcen sondern auch Statistiken über die interne Verwendung von Arithmetik. Varianten unseres Zahltyps übertreffen die Effizient anderer Zahltypen für viele geometrische Probleme.

Zweitens stellen wir neue und verbesserte Algorithmen vor, welche auf fehlerfreien Transformationen basieren. Fehlerfreie Transformationen erlauben es, den Fehler einer inexakten Gleitkommaoperation mit Hilfe einiger zusätzlicher Operationen zu rekonstruieren sowie Polynome über Gleitkommazahlen in Summen von Gleitkommazahlen zu transformieren. Wir stellen verbesserte Algorithmen für das Berechnen des Vorzeichens einer Summe von Gleitkommazahlen vor und zeigen wie diese für die Erstellung robuster Implementierungen geometrischer Algorithmen genutzt werden können. Weiterhin stellen wir neue Algorithmen für die effiziente Konvertierung von Summen von Gleitkommazahlen in konventionelle Zahldarstellungen vor. Diese Algorithmen erlauben es, zu mächtigeren Zahltypen zu wechseln, falls sich fehlerfreie Transformationen für eine Berechnung als unzureichend erweisen, ohne diese Berechnung neu starten zu müssen.

Drittens schlagen wir vor mit Hilfe fehlerfreier Transformationen den Aufbauen von Ausdrucksbäumen zu verzögern. Ausdrucksbäume aufzubauen verursacht maßgebliche Kosten, so dass arithmetische Operationen mit geringer Präzision möglicherweise schneller direkt durchgeführt werden können. Wir stellen ein Framework vor welches das Aufbauen von Ausdrucksbäumen verzögern und statt dessen Summen von Gleitkommazahlen berechnet und speichert. Unser Framework separiert fünf Komponenten einer solchen Strategie: grundlegende Arithmetik, maximale Anzahl von Summanden, Behandlung von Gleitkommaausnahmen, Reduktion der Anzahl von Summanden sowie Wechseln zu einer Darstellung als Ausdrucksbaum. Wir stellen verschiedene Implementierungen dieser Komponenten bereit und untersuchen ihren Einfluss auf die Gesamteffizienz.

ABSTRACT. Geometric algorithms are designed for the Real RAM model, which allows for exact arithmetic with real numbers in constant time. It is well known that replacing the Real RAM with imprecise hardware floating-point arithmetic leads to robustness issues. Implementations occasionally and unpredictably crash, compute incorrect results, or fail to terminate. A straightforward approach to overcome these problems is to implement the Real RAM. Software number types allow to perform numerical computations exactly or with very high precision, but at the cost of a large increase in running time. For geometric algorithms however, only exact sign computation is necessary.

Expression dag based number types record arithmetic expressions in form of a directed acyclic graph. This enables them to evaluate an expression multiple times and to avoid the cost of high precision computation in many cases. They are a general and user-friendly option to abandon numerical robustness problems in the implementation of geometric algorithms. In this thesis we report on our attempts to improve the performance of expression dag based number types by means of Algorithm Engineering. Our goal is to bring their performance closer to that of specialized solutions without sacrificing their usability or generality.

Our contributions are as follows. First, we present a new design for expression dag based number types. The design takes advantage of the C++ template mechanism, to make key components of expression dag based number types exchangeable. The design identifies and separates components which are not available or exchangeable in previous expression dag based number types. For most components, different options are implemented, showing the feasibility of our design. We compare different options for components experimentally, evaluating their influence on the total efficiency. Experiments do not only consider system resources, i.e., running time, but also statistics about internal usage of arithmetic. The most efficient variants of our number type outperform other number types on most geometric problems.

Second, we present new and improved algorithms based on error-free transformation. Error-free transformations allow to recover the exact rounding error arising in an otherwise inexact floating-point operation with the help of a few additional floating-point operations and to transform polynomial expressions of hardware floating-point numbers into sums of floating-point numbers exactly and very efficiently. We present improved algorithms for computing the sign of a sum of floating-point numbers and show how to use them for the robust implementation of geometric algorithms. Furthermore we present new algorithms for the efficient conversion of sums of floating-point numbers into more conventional number representations. These conversion algorithms allow to use software number types as fallback strategy if error-free transformations are insufficient for a numerical computation without restarting this computation from scratch.

Third and last, we propose to improve the performance of expression dag based number types by postponing dag creation with the help of error-free transformations. Creating an expression dag comes at a non-negligible cost and an arithmetic operation requiring only low precision might be faster to perform immediately. We present a framework to postpone dag creation by computing and storing sums of floating-point numbers instead. The framework separates five key components of such a strategy: basic arithmetic, maximum number of summands, handling floating-point exceptions, reducing the number of summands, and switching to an expression dag representation if necessary. We provide several implementations for each component and examine their influence on the total efficiency.

Contents

Chapter 1. Introduction 1
1.1. Geometric Problems and Predicates 6
1.2. Why Implementations Fail 11
1.3. Robust Geometric Computation 18
1.4. Robust Geometric Computing in Practice 26

Chapter 2. Previous Work 31
2.1. Expression Dag Based Number Types 31
2.2. Exact Floating-Point Computations 44

Chapter 3. *RealAlgebraic* – an expression dag based number-type 65
3.1. Policy Based Design 66
3.2. Expression Evaluation 74
3.3. A Case Study on Common Subexpressions 78
3.4. Experiments 83

Chapter 4. New and Improved Exact Floating-Point Algorithms 101
4.1. Exact Sign of Sum Computation 101
4.2. Expansion to Bigfloat Conversion 115

Chapter 5. Exact Floating-Point Algorithms in *RealAlgebraic* 137
5.1. Deferring Dag Construction 138
5.2. Experiments 150
5.3. Exact Floating-Point Algorithms for Dag Evaluation 159

Chapter 6. Conclusion 161

Bibliography 165

Appendix A. Complete Results from Experiments 171

CHAPTER 1

Introduction

Algorithm Engineering [**73**] is a discipline that attempts to close a growing gap between advances in the theoretical design of algorithms and the impact of these algorithms in practice. The design of algorithms in theoretical computer science considers simplified and abstract machine models. A usual assumption is for example that all basic operations take unit cost. The main design goals in theory are time and space efficiency, measured by asymptotic worst case analysis.

The results are often very sophisticated algorithms and data structures. However, a sophisticated algorithm may have only a small asymptotic advantage over a simpler one, and, due to having a larger constant factor arising from its intricacy, the advantage will only appear for unreasonably large input sizes. Sometimes the algorithms are ridiculously hard to implement or the constant factors hidden in the asymptotic analysis are too large to make an implementation worthwhile.

The assumptions made in the machine model and for the analysis are often unreasonable in practice. On modern hardware, operations are anything but unit-cost. Memory access, especially to the hard drive, is significantly slower than a simple arithmetic operation and algorithms should be designed taking this into account. Sometimes, an algorithm behaves much better in practice than predicted by the worst case analysis, due to the instances solved in practice having a different structure than the worst case assumed for the analysis. In this case a better analysis exploiting this structure might be possible.

These and other phenomena have been observed for quite a while and helped to create the discipline of Algorithm Engineering. Its goal is to create usable, efficient and robust implementations, by putting a larger focus on realistic computer models as well as the tuning and experimental analysis of algorithms. But Algorithm Engineering explicitly encompasses the design and analysis of algorithms too. Design and analysis precede implementation and experiments, but implementation and experiments can lead to findings that give rise to new ideas for the design and analysis. In this way, Algorithm Engineering mandates a cycle of on-going algorithmic improvement.

One gap between theory and practice can be found in computational geometry, where the implementation requires numerical computations that are rarely addressed in algorithm design. Geometric algorithms are usually designed in the *Real RAM model* [**19, 85**], which allows exact arithmetic with real numbers in constant time. In this model, the numerical computations required by geometric algorithms are

trivial. As soon as geometric algorithms are implemented, however, robustness problems surface. Naïve implementations occasionally crash, run forever, or compute incorrect results. It is well known, that these robustness problems stem from the straightforward replacement of the Real RAM with imprecise hardware arithmetic [**28, 97, 117**].

The distinguishing property of geometric algorithms is that they perform combinatorial computations that are guided by numerical computations. The main task of a geometric algorithm is to compute information about the global topology of the processed geometric objects. To do this, it repeatedly inspects the local topology of selected primitive geometric objects. The combinatorial part of the algorithm then integrates the local topology information into global topology knowledge. This part of the algorithm is usually considered in computational geometry theory.

To compute local topological information, numerical computations are used. *Primitive geometric objects* are objects of constant size, for example points, lines, circles, triangles, etc. Questions about the topology of constantly many primitive objects are called *geometric predicates*. Typical predicates are for example, whether two objects intersect or on which side of a line a point lies. Primitive geometric objects are represented by numerical data, i.e., point coordinates or coefficients in line equations. Answering a predicate is then equivalent to computing the sign of some arithmetic expression over numerical input data. In the Real RAM model, computing these signs is trivial, in practice numerical computations are either inaccurate or expensive in terms of running time. Predicate evaluation comprises the numerical part of a geometric algorithm implementation and guides the combinatorial part.

Many implementations of geometric algorithms still use hardware floating-point arithmetic as a replacement for the Real RAM. The now well known robustness problems arise from the imprecision necessarily inherent in this arithmetic. Depending on the complexity of the predicate, imprecise numerical computations more or less frequently return an incorrect sign. Thus, the combinatorial layer of the implementation will occasionally see wrong local topology. More grave is the fact, that the combinatorial layer will get information about local topology which is mutually contradictory! Due to one or more incorrect signs, there may be no global topology adhering to all the information given to the combinatorial layer. Not designed to handle inconsistent information, the algorithm will go into some undefined state and stop working correctly.

One solution to prevent these types of problems is to make sure all predicates return the correct answer at all times. This approach is called *Exact Geometric Computation* [**113**]. It ensures that the implementation is always in synchrony with its theoretical counterpart and hence works correctly. One approach to Exact Geometric Computation is providing a number type which guarantees correct sign computation. Software number types allow to compute a number correctly to the last bit, but at the cost of a much higher running time than hardware arithmetic. For sign computation, however, often approximate computations suffice. After all,

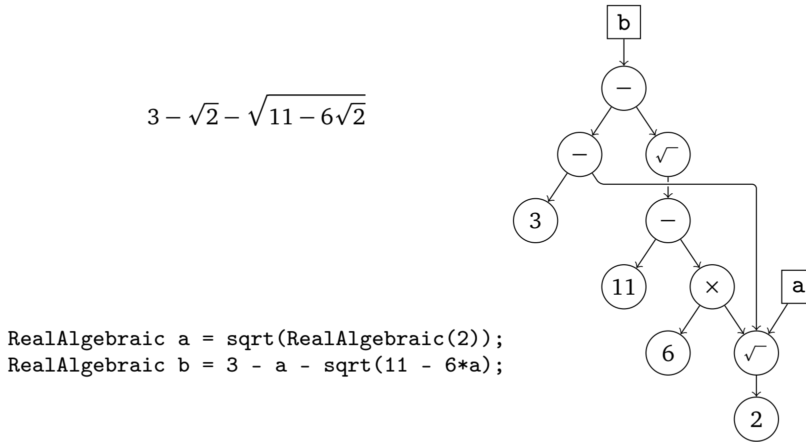


FIGURE 1.1. An expression, a corresponding dag and the code leading to this dag.

the hardware arithmetic does compute correct signs for most predicate evaluations. Otherwise geometric algorithms would not crash occasionally, but on every single instance.

One basic idea for efficiency improvement is to record arithmetic expressions and apply lazy evaluation to the sign computation. The recording is done in form of an *expression dag* , a directed acyclic graph, storing operands and operations as nodes. The sign computation is lazy in that the expression is evaluated not much more accurately than necessary. Typically, several approximate evaluations are made. If one evaluation does not verifiably compute the sign, another more accurate and more expensive evaluation follows. The final evaluation has to be exact to the extend that it guarantees to compute the sign correctly in any case. This strategy pays off for geometric applications because most predicate calls require only low accuracy computation.

Number types employing expression storage and lazy sign computation are called *expression dag based number types*. Examples are LEA [**3**], `CORE::Expr` [**119**], `leda::real` [**10**], and `Lazy_exact_nt` [**83**]. These number types have a couple of advantages over other approaches to Exact Geometric Computation. First of all, they are easy to use. Integrating exact sign computation into a number type allows to implement predicates exactly as they would be implemented in the Real RAM model. In some programming languages, like C++, operator overloading allows to use custom number types just like built-in number types. Then, expressions and formulas can be written down almost as in math textbooks. Expression dag based number types can be used straightforwardly but with the added benefit of abandoning all robustness and precision issues, cf. Figure 1.1.

Furthermore, many expression dag based number types are very general by handling *real algebraic numbers*. A real number α is called *algebraic* if it is the root of some polynomial with integer coefficients. The *degree* of an algebraic number is the smallest degree of any polynomial with integer coefficients having α as root. Any integer is a real algebraic number and real algebraic numbers are closed under field operations $(+, -, \times, /)$, radicals ($\sqrt[d]{}$), and more generally extracting a real root of a polynomial with real algebraic coefficients. The latter operation is called the *diamond operator* ($\diamond$). Current state of the art expression dag based number types implement algebraic numbers. Most support field operations and radicals, some even extracting roots from polynomials. This is remarkable since real algebraic numbers are the largest natural set of numbers for which algorithms for exact sign computation are known at all. It is, for example, open whether the sign of an expression can be computed, if the functions exp or log are admitted [**90**].

Finally, these number types are specifically designed to be efficient in geometric applications. The lazy approach to sign computation ensures that an expression is evaluated with the accuracy actually needed to compute the sign, not some a priori sufficient, high accuracy. The accuracy of an evaluation highly determines the running time. Therefore, in expression dag based number types, the running time for a sign computation is strongly, positively correlated to the difficulty of the sign computation. This behavior is called *adaptivity*. In comparison to hardware floating-point arithmetic, there is only a small overhead for all those predicate calls which would be solved correctly anyway and more running time is only spend on cases where it is worthwhile. This is in large contrast to traditional software number types, where the running time for an evaluation depends almost exclusively on the structure of the expression and is always large.

In terms of efficiency, however, expression dag based number types often fall short to more specialized approaches to implement geometric algorithms robustly. These approaches are applied not at the level of number types and arithmetic, but at the predicate or combinatorial level of an implementation. On the predicate level, for example, the expression to be evaluated is known and no explicit storage, allowing for multiple evaluations, is necessary. This and other a priori knowledge, not available at the arithmetic level, can be exploited to achieve more efficient robust implementations. Nevertheless, advanced and specialized solutions require more knowledge and more effort by the implementor, while using a number type is straightforward.

In this thesis we report on attempts to improve the performance of expression dag based number types by means of Algorithm Engineering. The goal is to bring their performance closer to that of specialized solutions, while keeping their advantages: ease of use and generality. The result is a new expression dag based number type *RealAlgebraic* written in `C++`. *RealAlgebraic* is available for download on the Internet [**89**] and contains all the techniques described in this work. Some parts of *RealAlgebraic* have gone through the Algorithm Engineering cycle several times,

where improvements were guided by experimental results. We describe the current state and discuss the history where appropriate. Due to the on-going development, experimental results in this thesis differ in some points from those presented in previous publications [**63, 64**].

Our contributions are as follows. First, we present a new design for expression dag based number types. The design takes advantage of the C++ template mechanism, to make key components of expression dag based number types exchangeable. In some parts, the design identifies and separates components which are not available or exchangeable in previous expression dag based number types. For most basic components, different options are implemented, showing the feasibility of the design. This, too, goes beyond previous implementations. We compare different options for components experimentally, evaluating their influence on the total efficiency. Experiments do not only consider system resources, i.e., running time, but also statistics about internal usage of arithmetic. Collecting these statistics is greatly simplified by the design. Finally, the design allows a user to adjust *RealAlgebraic* to her needs in terms of available third party libraries and type of application.

Second, we propose to improve the performance of expression dag based number types by postponing dag creation. Creating an expression dag comes at a non-negligible cost and an arithmetic operation requiring only low precision might be faster to perform immediately. So called *error-free transformations* allow to transform polynomial expressions of hardware floating-point numbers into sums of floating-point numbers exactly and very efficiently. Algorithms based on error-free transformations have been devised, e.g., for sign computation, very accurate arithmetic and exact arithmetic [**77, 93, 104**]. We present a framework, based on algorithms employing error-free transformations, to postpone dag creation by computing and storing sums of floating-point numbers instead. The framework separates five key components of such a strategy. These are (1) the basic algorithms to use, (2) a limit on the number of summands to store, (3) a strategy to handle floating-point exceptions which may render error-free transformations inexact, (4) a strategy to reduce the number of summands occasionally, and (5) a strategy to switch to an expression dag representation if necessary. For each component, different implementations are provided and we examine their influence on the total performance experimentally.

Third and last, we present three new and improved algorithms based on error-free transformation. The first, ESSA, is an improvement of an algorithm by Helmut Ratschek and Jon Rokne [**88**]. Though still rather slow compared to other exact sign of sum algorithms, this algorithm is very robust with respect to the floating-point environment. The second, SIGNK, combines techniques from Siegfried Rump and his co-workers [**77, 91**] into an efficient algorithm for computing the sign of a sum of floating-point numbers. The third, MONOTONIZE was designed from scratch and is a new tool for the efficient conversion of expansions, special sums of floating-point numbers arising from algorithms by Douglas Priest [**86**] and Jonathan

Shewchuk [**104**], into software floating-point numbers. The latter two algorithms are part of the framework for postponing dag creation in *RealAlgebraic*.

In the remaining parts of this chapter we examine more closely why and how floating-point based implementations of geometric algorithms fail. Then we discuss approaches to resolve these robustness problems. Most of them follow the Exact Geometric Computation paradigm but some provide robustness by other means.

1.1. Geometric Problems and Predicates

Geometric computation can be divided into three levels building upon each other. The bottom level provides numbers and arithmetic. The middle level, often called the *geometric kernel*, provides primitive geometric objects as well as operations on them. Basic operations on geometric objects are constructions, which create new geometric objects and predicates, which answer questions about the relation of geometric objects. A typical construction is the creation of the line defined by two different points, or its dual, computing the intersection point of two non-parallel lines. The top level finally consists of algorithms and data structures and is usually purely combinatorial.

In this section we discuss some classic problems in computational geometry frequently surfacing in this thesis. We will not discuss algorithms for these problems, but constructions and predicates that are associated with them, since these are the tasks that must be solved on the arithmetic level. In general, different algorithms for the same problem are based on the same set of predicates, though in some cases predicates are algorithm dependent. Looking at the predicates will give us some understanding of what arithmetic has to accomplish in order to provide Exact Geometric Computation.

A configuration of input objects that lets a predicate evaluate to zero, is called a *degenerate* configuration. What is considered degenerate is thus predicate dependent and in some cases algorithm dependent. A set of input objects containing no degenerate configuration is said to be in *general position*. As we will see, degenerate and nearly degenerate configurations are those which contribute most to robustness problems.

Convex Hull. Given n points in Euclidean space $\mathbb{R}^d$, the *convex hull* is the smallest convex subset of $\mathbb{R}^d$ containing all points. The convex hull is a convex polytope whose vertices are input points. The problem is thus to compute the combinatorial structure, i.e., the face lattice of the convex hull polytope. The problem is often considered in fixed, small dimension d.

A single predicate, the so called *orientation predicate* is needed for convex hull computation. Given points $p_1, \ldots, p_{d+1} \in \mathbb{R}^d$, it is equivalent to computing the sign

of the determinant

$$D_{\mathrm{O}d} = \begin{vmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,d} & 1 \\ p_{2,1} & p_{2,2} & \cdots & p_{2,d} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ p_{d,1} & p_{d,2} & \cdots & p_{d,d} & 1 \\ p_{d+1,1} & p_{d+1,2} & \cdots & p_{d+1,d} & 1 \end{vmatrix}. \tag{1.1}$$

The sign of $D_{\mathrm{O}d}$ is zero if all $d+1$ points are inside a $d-1$ dimensional hyperplane of $\mathbb{R}^d$. Otherwise, the sign of $D_{\mathrm{O}d}$ tells on which side of the hyperplane spanned by $p_1, \ldots, p_d$, the point p_{d+1} resides. The definition of side here depends on the orientation of $p_1, \ldots, p_d$ within the hyperplane spanned by them. Computing a d-dimensional convex hull needs the orientation predicate for all dimensions from 1 up to d. Note that the 1-dimensional orientation predicate is equivalent to coordinate comparison.

Lets make this more precise for two dimensions. Given three points p, q and r in the plane, the 2D orientation predicate determines the position of r relative to the oriented line $\vec{\ell}(p,q)$, passing first through p and then q. Let $p = (p_x, p_y)$, $q = (q_x, q_y)$ and $r = (r_x, r_y)$, then the predicate is tantamount to computing the sign of the determinant

$$D_{\mathrm{O}2} = \begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix} = \begin{vmatrix} p_x - r_x & p_y - r_y \\ q_x - r_x & q_y - r_y \end{vmatrix}. \tag{1.2}$$

The three points p, q and r are collinear if and only if $D_{\mathrm{O}2}$ is zero. Otherwise, r is to the left of $\vec{\ell}(p,q)$, if $D_{\mathrm{O}2}$ is greater than zero and r is to the right of $\vec{\ell}(p,q)$, if $D_{\mathrm{O}2}$ is smaller than zero.

VORONOI DIAGRAM, DELAUNAY TRIANGULATION. Let n points in $\mathbb{R}^d$ be given, and call them sites. The *Voronoi cell* of a site is the set of points in $\mathbb{R}^d$ not closer to any other site, measured by Euclidean distance. A Voronoi cell is a convex polyhedron and the intersection of two Voronoi cells is a face of both. The *Voronoi diagram* is the cell complex, partitioning whole $\mathbb{R}^d$, induced by all Voronoi cells. Its vertices are called *Voronoi vertices*. A Voronoi vertex has the same distance to all sites in whose Voronoi cell it is contained. The boundary of all Voronoi cells is called the *Voronoi skeleton*.

The *Delaunay diagram* of n sites in $\mathbb{R}^d$ is a cell complex partitioning the convex hull of the sites into convex polytopes. It is dual to the Voronoi Diagram in the following sense. Each full dimensional polytope P in the Delaunay diagram corresponds to a Voronoi vertex v. The vertices of P are exactly those sites, whose Voronoi cell contains v. If the sites are in general position, i.e., no $d+2$ sites are co-spherical, the Delaunay diagram is a simplicial triangulation, i.e., all polytopes are simplices. A *Delaunay triangulation* is a refinement of the Delaunay diagram obtained by triangulating all non-simplices in a compatible way.

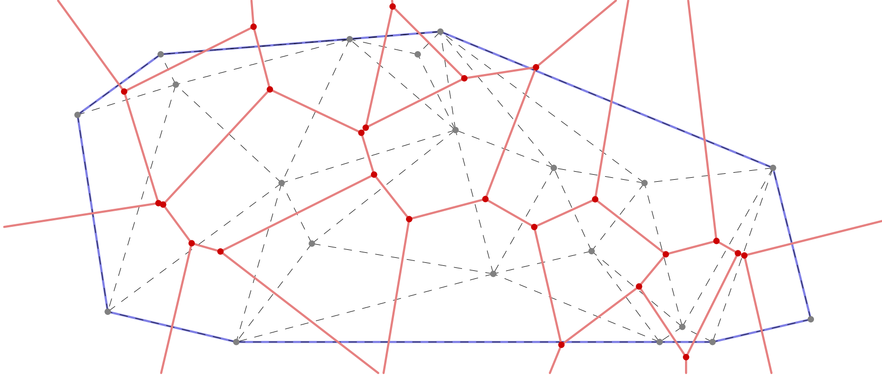


FIGURE 1.2. Convex hull ——, Delaunay triangulation - - -, and Voronoi diagram —•—, for a set of points • in the plane.

Computing the Voronoi diagram or Delaunay diagram is based on the orientation predicate and another predicate called the *insphere* or *incircle* predicate. Given $d+2$ points in $\mathbb{R}^d$, the insphere predicate checks whether one of the points is inside, on the boundary of, or outside the sphere spanned by the remaining $d+1$ points. To this end, each point $(p_1, p_2, \ldots, p_d)$ is lifted to the paraboloid

$$X_{d+1} = X_1^2 + X_2^2 + \ldots + X_d^2$$

in $d+1$ dimensional space. Then, the $d+1$ dimensional orientation test is performed on the lifted points.

Again, lets make this more precise for two dimensions. Given four points p, q, r, and s in the plane, the incircle test determines the position of s relative to the circle (possibly degenerated to a line) defined by p, q, and r. It is tantamount to computing the sign of the determinant

$$D_{\mathrm{IC}} = \begin{vmatrix} p_x & p_y & p_x^2+p_y^2 & 1 \\ q_x & q_y & q_x^2+q_y^2 & 1 \\ r_x & r_y & r_x^2+r_y^2 & 1 \\ s_x & s_y & s_x^2+s_y^2 & 1 \end{vmatrix} = \begin{vmatrix} p_x - s_x & p_y - s_y & (p_x-s_x)^2+(p_y-s_y)^2 \\ q_x - s_x & q_y - s_y & (q_x-s_x)^2+(q_y-s_y)^2 \\ r_x - s_x & r_y - s_y & (r_x-s_x)^2+(r_y-s_y)^2 \end{vmatrix}. \tag{1.3}$$

D_{IC} is zero if and only if all four points lie on a common circle. Otherwise, consider the circle boundary to be oriented, passing through p, q, and r in this order. Then, s is on the right side of the circle boundary, if D_{IC} is greater than zero, and on the left side, if D_{IC} is smaller than zero.

Thus, convex hull, Euclidean Voronoi diagram and Delaunay triangulation in d-dimensional space can all be computed using predicates which employ polynomial expressions of degree roughly d. The 2D and 3D orientation predicate and the incircle and insphere predicate are by far the most prominent predicates in the literature.

Arrangements. A set of curves in the plane induces a subdivision of the plane called an *arrangement*. The arrangement consists of vertices – intersection points of curves, edges – connected subsets of a curve separated by intersection points, and faces – connected subsets of the plane separated by curves. The goal is to compute the adjacency structure between these elements in form of a planar map. A large class of algorithms for computing arrangements are sweep line algorithms, traversing the plane from left to right and updating data-structures wherever the combinatorial structure changes. Hence, one basic predicate for these algorithms is to compare intersection points of curves by x-coordinate.

The basic case are arrangements of straight lines and line segments. Segments are usually defined by their endpoints, but it is not uncommon to represent lines in the same fashion. Let p,q,r, and s be four points in the plane and let

$$D_x = \begin{vmatrix} p_x q_y - p_y q_x & q_x - p_x \\ r_x s_y - r_y s_x & s_x - r_x \end{vmatrix}, \quad D_y = \begin{vmatrix} p_y - q_y & p_x q_y - p_y q_x \\ r_y - s_y & r_x s_y - r_y s_x \end{vmatrix},$$
$$D_w = \begin{vmatrix} p_y - q_y & q_x - p_x \\ r_y - s_y & s_x - r_x \end{vmatrix}. \tag{1.4}$$

Then, the lines $\ell(p,q)$ and $\ell(r,s)$ are parallel or identical, if $D_w = 0$. Otherwise, their intersection point is given by $t = (D_x/D_w, D_y/D_w)$. Whether the two segments $\overline{pq}$ and $\overline{rs}$ intersect, can be checked using the orientation test to determine the position of p and q relative to $\ell(r,s)$ and the position of r and s relative to $\ell(p,q)$. Of course, the intersection point is also given by t. Computing intersection points involves a division. In coordinate comparison the division may be avoided by rearranging, though some care must be taken, not to flip the sign. Thus, comparing the x-coordinates of two intersection points t and t' amounts to computing

$$\operatorname{sign}(D_w)\operatorname{sign}(D'_w)\operatorname{sign}(D_x D'_w - D'_x D_w).$$

Computing an arrangement of circles and circular arcs is arithmetically more demanding, since the intersection point of two circles has algebraic coordinates. Let

$$C_i : \ (X - p_i)^2 + (Y - q_i)^2 - r_i^2 = 0 \qquad \text{for } i = 1, 2,$$

be two circles, where we assume p_i, q_i and r_i are input numbers. The x-coordinates of their intersection points are the real roots of the polynomial

$$P : \ 4(u^2 + v^2)X^2 \ + 4(wu - 2v^2 p_2)X \ + w^2 - 4v^2(r_2^2 - p_2^2).$$

where

$$u = p_2 - p_1, \qquad v = q_1 - q_2, \qquad w = r_2^2 - r_1^2 + v^2 + p_1^2 - p_2^2.$$

The coefficients of P are polynomials of degree four in the input data. Now denote these coefficients by a, b, and c, i.e., let $P = aX^2 + bX + c$, then the roots of P are given by

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

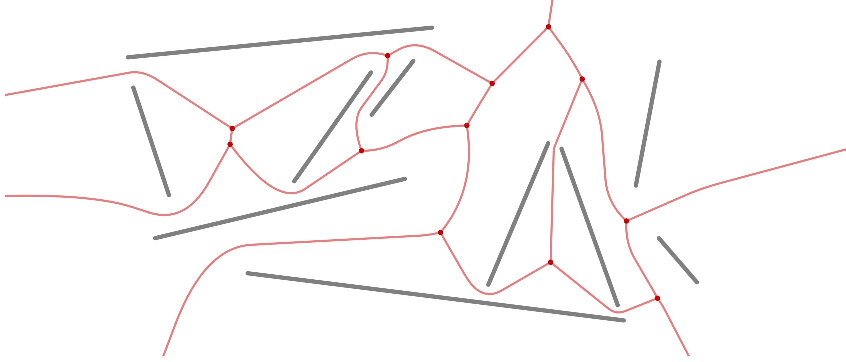


FIGURE 1.3. Voronoi diagram —•—, of line segments ——.

The sign of the polynomial discriminant $\Delta = b^2 - 4ac$ determines the number of real roots of P and hence the number of intersection points of C_1 and C_2. Comparing the x-coordinates of two intersection points is then equivalent to computing the sign of an expression of the form

$$\frac{-b' \pm \sqrt{b'^2 - 4a'c'}}{2a'} - \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{1.5}$$

VORONOI DIAGRAM OF SEGMENTS. Voronoi diagrams have been generalized in many ways, for example by considering a different measure of distance or by considering different types of sites. One generalization that has been studied extensively, is the Voronoi diagram of line segments, cf. Figure 1.3. Here, the sites are line segments and the distance to a site is the smallest Euclidean distance to any point in the site.

The predicates for Voronoi diagrams of segments again involve algebraic numbers. It suffices to consider lines and points in predicates, since the shortest distance to a segment is either the distance to one of the endpoints or the distance to the line supporting the segment. Predicates for computing the Voronoi diagram of segments have been analyzed by Christoph Burnikel in his thesis [**8**]. In [**14**], one incircle test is discussed in detail. Consider three sites, two of which are lines $\ell_i : a_i X + b_i Y + c_i = 0$ for $i = 1, 2$, while the third is a point $p = (0, 0)$, assumed to be the origin. The situation is illustrated in Figure 1.4. In general, there are two Voronoi vertices v for these sites. They have coordinates $v = (h_x/h_w, h_y/h_w)$, where

$$\begin{aligned}
h_x &= a_1c_2 + a_2c_1 \pm \sqrt{2c_1c_2(D_1D_2 + a_1a_2 - b_1b_2)},\\
h_y &= b_1c_2 + b_2c_1 \mp \operatorname{sign}(rs)\sqrt{2c_1c_2(D_1D_2 + b_1b_2 - a_1a_2)},\\
h_w &= D_1D_2 - a_1a_2b_1b_2.
\end{aligned}$$

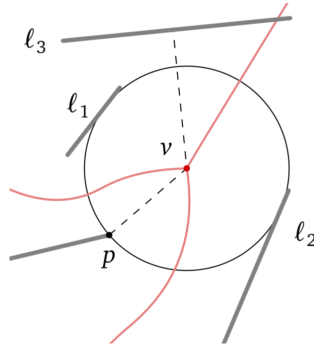


FIGURE 1.4. Incircle test for the Voronoi diagram of segments. Sites ℓ_1, ℓ_2 and p give rise to Voronoi vertex v. The distance from site ℓ_3 to v is checked.

and

$$s = b_1 D_2 - b_2 D_1, \qquad r = a_1 D_2 - a_2 D_1, \qquad D_i = \sqrt{a_i^2 + b_i^2}.$$

To test whether a third line $\ell_3 : a_3 X + b_3 Y + c_3 = 0$ is closer to, has the same distance to, or is further away from v, we can compare the squared distances from v to ℓ_3 and from v to p, i.e., we need to determine the sign of

$$\frac{(a_3 v_x + b_3 v_y + c_3)^2}{a_3^2 + b_3^2} - (v_x^2 + v_y^2). \tag{1.6}$$

This expression contains nested square roots and is therefore again a bit more complex than comparing coordinates of intersection points of circles.

1.2. Why Implementations Fail

Some failures of implementations of geometric algorithms have been reported in the literature. The LEDA book [**53, 58**] gives some instructional examples in section 9.6 and reports on experiments that fail for floating-point based implementations in sections 10.7 and 10.8. Shewchuk [**104**] discusses an example where the computation of a 2D Delaunay triangulation by divide and conquer fails.

Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra and Chee Yap [**51**] show how to create input data that lets an implementation fail for the problem of 2D convex hull and 3D Delaunay triangulation. After selecting an algorithm and a fixed implementation thereof, they examine the decisions the implementation will make for input points in a certain critical range of space. This allows them to select input points where these decisions are incorrect and furthermore lead to failures. Failures generated include computing output that is not intended, but also infinite looping, or the program crashing. Their examples are easy to understand and their methods can

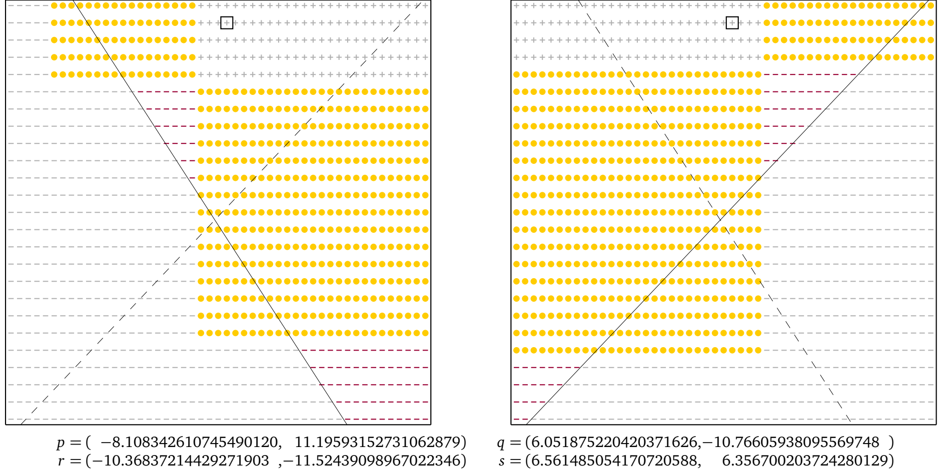


FIGURE 1.5. A case where the floating-point arithmetic gives inconsistent results: the intersection point is neither on $\ell(p,q)$ nor on $\ell(r,s)$.

be adapted to floating-point based implementations of algorithms for other geometric problems too.

A very common floating-point based implementation of the 2D orientation predicate Equation (1.2) is the straightforward one. Simply compute an approximation D'_{O2} of D_{O2} as

$$D'_{O2} = (p_x \ominus r_x) \otimes (q_y \ominus r_y) \ominus (q_x \ominus r_x) \otimes (p_y \ominus r_y), \tag{1.7}$$

where $\otimes$ and $\ominus$ are the imprecise floating-point multiplication and subtraction. The point r is distinguished in the computation of D'_{O2} and is called the pivot point. Many familiar mathematical properties like associativity or distributivity do not hold for floating-point operations. Therefore, while permuting p, q, and r will lead to sign changes only in D_{O2}, this is not the case for D'_{O2} in general.

LINE INTERSECTION. Lets have a look at the (in)consistency of floating-point based implementations of the 2D orientation predicate and computing the intersection point of two lines given by points. We start with four points $p,q,r,s \in \mathbb{R}^2$ with floating-point coordinates, compute the lines $\ell(p,q)$ and $\ell(r,s)$ and their intersection point $u = \ell(p,q) \cap \ell(r,s)$. Then we check whether u is on any of the lines. Stefan Schirra [**98**] reports that only about 20% of intersection points are reported to be on both lines in an experiment testing a large number of lines. Here, we look at two examples only, but examine the situation near the intersection point more closely. We visualize the results by actually checking for all points with floating-point coordinates near the exact intersection of both lines, whether they are on any of the two lines.

For our computations we use the C++ library CGAL [**16**], a major effort to create complete, correct, and efficient implementations of geometric algorithms. CGAL very

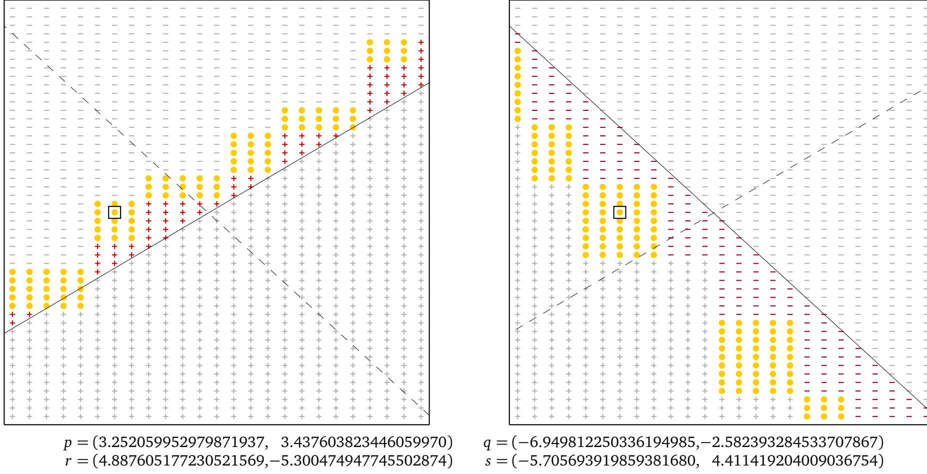
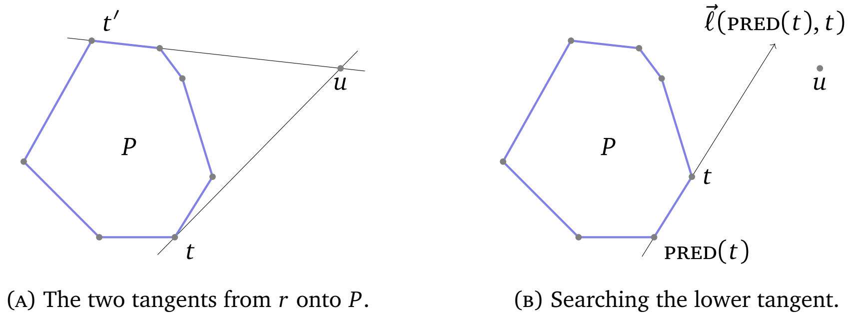


FIGURE 1.6. A case where the floating-point arithmetic gives consistent results: the intersection point is on both $\ell(p,q)$ and $\ell(r,s)$.

strictly follows the Exact Geometric Computation paradigm. A kernel in CGAL provides basic geometric objects and operations and thus corresponds directly to the middle level in geometric computation. The bottom level is provided by instantiating the kernel with a number type. We use CGAL's `Simple_cartesian` kernel, instantiated with `double` for floating-point based computations. The implementation of predicates and constructions in the kernel is treated as a black box. Other implementations will give different results, but as long as they use inexact floating-point based computations, the overall picture will be similar. To check the correctness of the floating-point based computations and to draw the images correctly, we use the `Simple_cartesian` kernel too, but instantiated with *RealAlgebraic*, the number type discussed in this thesis.

Images, showing the results are given in Figure 1.5 and Figure 1.6. In the left image, the line $\ell(p,q)$ is drawn solid, while $\ell(r,s)$ is drawn dashed. We closely zoom in on the intersection. For each point u with floating-point coordinates we classify if u is to the right (−), to the left (+), or on (●) $\vec{\ell}(p,q)$. Correctly classified points are drawn in grey, while misclassified points are colored (−, +, ●). The approximate intersection point, computed with floating-point arithmetic, is encased by a small box. In the right picture, the roles of $\ell(p,q)$ and $\ell(r,s)$ are reversed. Hence, on the right side all points are classified with respect to $\vec{\ell}(r,s)$ and the lines are used in different order to compute the intersection point.

Reversing the roles of $\ell(p,q)$ and $\ell(r,s)$ does not affect the approximate intersection point in both examples, i.e., CGAL's intersection point computation seems consistent in this regard. This is no surprise, since the straightforward floating-point



(A) The two tangents from r onto P.

(B) Searching the lower tangent.

FIGURE 1.7. Basic steps in a 2D convex hull computation.

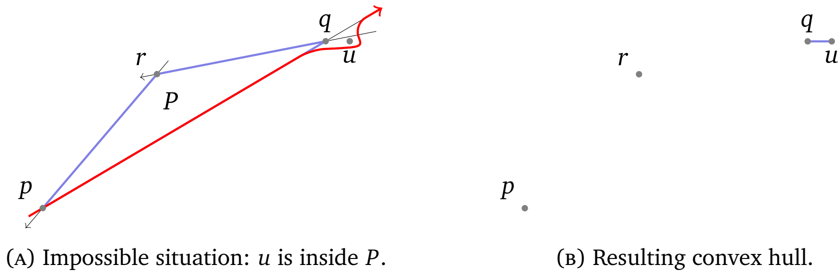
implementation of Equation (1.4) is still symmetric with respect to $\ell(p,q)$ and $\ell(r,s)$. In Figure 1.5, the approximate intersection point is correctly classified as left of both $\vec{\ell}(p,q)$ and $\vec{\ell}(r,s)$. Of course, mathematically, the intersection point of two lines is on both lines, so here the floating-point implementation is inconsistent. This simple example already indicates, that geometric algorithms that rely on this property may be hard to implemented with floating-point arithmetic!

The situation in Figure 1.5 could be improved by computing a better approximation of the intersection. There are points near the exact intersection that are classified to be on both lines. In Figure 1.6, the approximate intersection point of $\ell(p,q)$ and $\ell(r,s)$ is classified to be on both lines. Here, the floating-point arithmetic delivers consistent, if inexact, results by sheer coincidence. Points near the exact intersection are however classified to be not on any of the lines.

Both examples also show that the 2D orientation predicate delivers incorrect results for many points close to the lines. No points are classified correctly to be on a line and in fact almost no point is. Furthermore the topology of misclassified points is highly irregular. This is even more evident in the pictures by Kettner et al. [**51**], which show a larger section of points along a line. Nevertheless, for most points, i.e., all that are sufficiently far away from the line the orientation predicate decides correctly and the approximate intersection point is quite close to the exact intersection point. In general, floating-point arithmetic guarantees small errors and computes quite accurately. As we will see shortly, this observation can be quantified and exploited for efficient Exact Geometric Computation.

CONVEX HULL. Computing the convex hull of a set of points in the plane is one of the best studied problems in computational geometry. We will now discuss how a simple plane sweep algorithm for computing the convex hull can fail, when implemented with floating-point arithmetic.

The algorithm is a variant of Grahams scan [**19**] and proceeds as follows. We process the points one by one, in lexicographical order by increasing x and y-coordinate. We maintain the convex hull P of already processed points as the circular



(A) Impossible situation: u is inside P.

(B) Resulting convex hull.

FIGURE 1.8. A single incorrect result from the 2D orientation predicate can lead to catastrophic failure.

sequence of vertices in counterclockwise order along its boundary. Thus, when we arrive at a new point u, we have to update P. Since we process points from left to right, u is not contained in P. Consider the two tangents from u onto P, see Figure 1.7A. Each tangent touches exactly one or two vertices of P. Let t' be the vertex of P furthest from u that the upper tangent touches, and let t be the vertex of P furthest from u that the lower tangent touches. Note that $t = t'$ is possible in case P is a segment. In the sequence of vertices of P, we replace all vertices between t and t' with u.

How do we find t and t'? Let q be the point that was processed in the previous update step, then q is the rightmost vertex of P. We start with $t = q$ and check the position of u relative to $\vec{\ell}(\text{PRED}(t), t)$ with the 2D orientation predicate, see Figure 1.7 As long as u is not on the left side, we advance t to $\text{PRED}(t)$ and check the position of u again. The vertex t' can be found analogously. There is one exceptional case: if P is a segment we have to stop the search for t and t' after at most one step.

What can go wrong with this algorithm when a floating-point based orientation predicate is used? We have already seen that the CGAL implementation instantiated with `double` occasionally decides incorrectly. The search for t and t' may stop to soon, in which case the resulting polygon is not convex anymore. This may lead to incorrect output or more problems in later steps, since the correctness of the search for t and t' depends on P being convex! The search may also stop too late, in that case vertices of P are cut away and may not be contained in the end result.

Carrying the first case to the extreme, the search for both t' and t may stop at q. This can never occur geometrically, since for full dimensional P the upper and lower tangent touch P in different vertices. But it can occur with a single incorrect result from the 2D orientation predicate, as illustrated in Figure 1.8A. Here the current hull polygon P has vertices p, q and r and is about to be updated with point u. Figure 1.9 shows the lattice of points with floating-point coordinates close to q, among them u, and how the 2D orientation predicate classifies their position relative to $\vec{\ell}(p,q)$ and $\vec{\ell}(r,q)$. Point u is correctly classified to be right of $\vec{\ell}(r,q)$, but incorrectly classified to

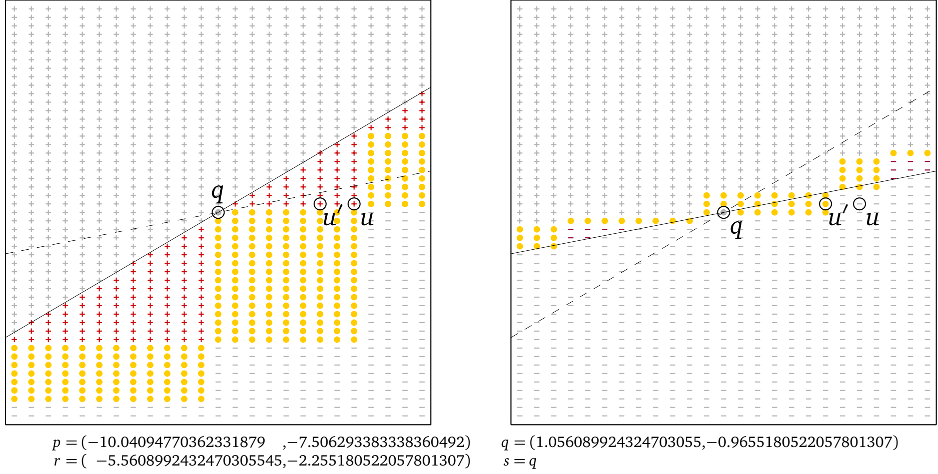


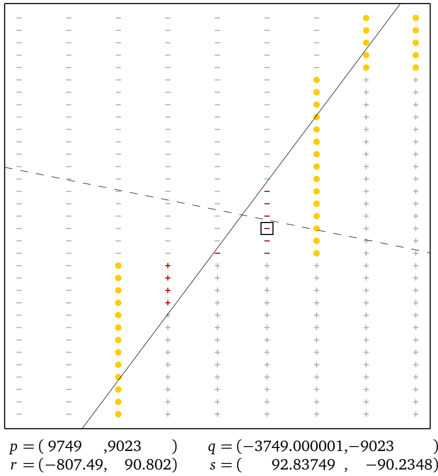
FIGURE 1.9. Zoom in on points q and u in Figure 1.8.

be left of $\vec{\ell}(p,q)$. Since we process points from left to right, u must be outside P, but the incorrect classification moves u inside P, at least from the view of the algorithm. We have $t = t' = q$ and all vertices between t and t' are to be replaced by u. The implementation may now traverse the sequence of hull vertices, starting at t, and remove vertices until t' is reached. This will remove all vertices except q. Then u is inserted, resulting in the hull polygon shown in Figure 1.8B.

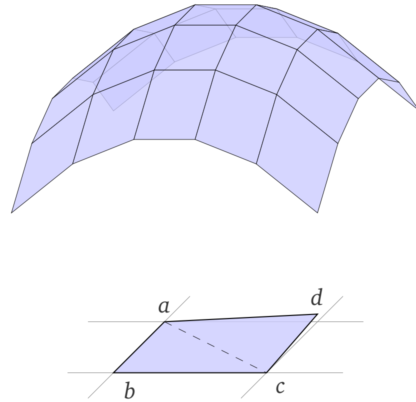
For another example, suppose we are updating the current hull P with the point u', three positions to the left from u in the lattice of floating-point points in Figure 1.9. The point u' is incorrectly classified to be on $\vec{\ell}(r,q)$, and incorrectly classified to be left of $\vec{\ell}(p,q)$. Hence, u' will be inserted into the sequence of vertices between q and r, resulting in a non-simple hull polygon.

The algorithm described above fails due to a single incorrect predicate conclusion, inconsistent with previous predicate decisions. Processing the points from left to right requires to sort them initially, and sorting uses coordinate comparison of points as a predicate. Sorting ensures that each point processed is outside the current hull, this property is used extensively by the algorithm. If an incorrect 2D orientation test places a point inside the current hull, this contradicts several previous predicate results from sorting and problems ensue. But ensuring correct predicate results ensures faithfulness to the actual input data and hence consistency. Together with the correctness of an algorithm, this guarantees correct output. This is the essence of Exact Geometric Computation.

INCONSISTENT INPUT. Floating-point based predicate implementations can lead to inconsistencies, but sometimes the inconsistencies already exist in the input data. Unless a line is in very special position, e.g., axis-parallel, almost no points with



(A) The intersection of $\ell(p,q)$ and $\ell(r,s)$ has no floating-point coordinates.



(B) A quadrilateral mesh and a non-planar quadrilateral.

FIGURE 1.10. Systems of equations almost never have floating-point solutions. As a result quadrilaterals in meshes almost never have coplanar vertices.

floating-point coordinates are exactly on that line. Hence the intersection of two lines is almost never a point with floating-point coordinates, cf. Figure 1.10A. Similarly, systems of (linear) equations almost never have a solution in floating-point numbers.

A mesh is a data structure used to model surfaces in three dimensional space. It is a collection of vertices, edges and faces. The adjacency structure between vertices, edges and faces makes up the combinatorial part of a mesh. The numerical data are usually the coordinates of the vertices. In a quadrilateral mesh, each face is a quadrilateral, enclosed by four edges and four vertices. Since three points determine a plane in three-dimensional space, the fourth point of a quadrilateral should be in the plane determined by the other three points. An interior vertex of a mesh is shared by at least three quadrilaterals. That means, if we fix the other vertices of the involved quadrilaterals to points with floating-point coordinates, this vertex should be at the intersection of three planes and hence is unlikely to have floating-point coordinates. The overwhelming majority of input data does however have floating-point coordinates, simply because this is the dominant numerical format available. As a consequence, four points forming a quadrilateral in a mesh are almost never coplanar. Figure 1.10B shows a quadrilateral mesh and a non-planar quadrilateral.

Implementations of geometric algorithms failing due to this type of inconsistency can not be repaired by Exact Geometric Computation. To determine the location of a point relative to a non-planar quadrilateral using the 3D orientation predicate, three

vertices from the quadrilateral must be used. The answer may differ, depending on which of the vertices are selected! Handling or removing inconsistencies in the input data is in general a very hard problem.

1.3. Robust Geometric Computation

The problems caused by inexact floating-point arithmetic, can be attacked on all three levels of geometric implementation. To ensure correct control flow, correct results from all predicate calls are necessary. This can be achieved on the arithmetic level by number types that compute exact numerical values or at least provide exact sign computation. On the level of geometric kernels, typically more efficient techniques are available, exploiting a priori knowledge not available on the arithmetic level. Finally, on the combinatorial level, one may redesign an algorithm to be able to cope with numerical inaccuracies. These approaches usually do not fall into the category of Exact Geometric Computation but still lead to robust implementations for some geometric algorithms. Of course, there also exist approaches that span multiple levels of geometric implementation.

1.3.1. Exact Arithmetic and Number Types. The Real RAM model allows for exact real arithmetic at unit cost, but many geometric problems require only integer, rational, or at most real algebraic numbers. For example, the set of integers, $\mathbb{Z}$, allows exact ring operations $(+, -, \times)$, while the set of rational numbers allows exact field operations $(+, -, \times, /)$. Exact numerical computation with software number types forms the backbone of Exact Geometric Computation, as a measure of last resort if more efficient approaches have failed to give correct results. Software number types are also used to compute approximations, if high *accuracy* is required. The term *accuracy* relates to how close an approximation is to its ideal mathematical counterpart.

Floating-point numbers reside between integer and rational numbers. They are important due to their omnipresence in hardware. Almost all input numbers are hardware integer or floating-point numbers. A binary floating-point number $f \neq 0$ consists of a *sign* $s \in \{-1, 1\}$, a sequence m of bits called *mantissa*, and an *exponent* $e \in \mathbb{Z}$. The number of bits in m is called the *precision* of f. By interpreting m as a binary number in a suitable way, e.g., as an integer, f is given as

$$f = s \times m \times 2^e.$$

Many different conventions to interpret or normalize m are in use. With $\mathscr{F}$, we denote the set of all floating-point numbers, i.e., all numbers representable in the described way, plus zero.

Floating-point numbers come in two flavors. The first are hardware floating-point numbers, usually following the IEEE 754 standard [**45, 46**]. They have a fixed precision and limited exponent range. Basic arithmetic operations $(+, -, \times, /, \sqrt[2]{\ })$ are performed approximately, but optimally. The result of an operation is a floating-point number closest to the true result. The other flavor are software floating-point

numbers. Up to memory limitations, they allow arbitrary precision and unlimited exponent range. Ignoring the limitations, as we will usually do in this thesis, software floating-point numbers are the subset of rational numbers whose denominator is a power of two. In this setting, ring operations $(+,-,\times)$ can be performed exactly and basic arithmetic operations (e.g., $+,-,\times,/,\sqrt[d]{\ },\ldots$) as well as important functions (e.g., $\exp,\log,\ldots$) can be performed approximately with arbitrary accuracy. The accuracy of a floating-point number is limited by its precision, but high precision does not guarantee high accuracy, as errors usually propagate in complex computations. Since most geometric predicates require to compute the sign of polynomial expressions in input values only, software floating-point numbers suffice for the correct implementation of many geometric algorithms. To distinguish between hardware and software floating-point numbers, from now on we call hardware floating-point numbers simply floating-point numbers and software floating-point numbers will be called *bigfloat* numbers.

Approaches to exact arithmetic with real algebraic numbers are less well known. A real algebraic number α can, for examples, be represented exactly by a tuple (p,I) where p is a polynomial having α as root and I is an isolating interval for α, i.e., α is the only root of p in I. An exact representation in the straightforward sense, i.e., as a decimal or binary number is not possible. This poses no problem for geometric applications, insofar it suffices to have a representation that is closed under the necessary arithmetic operations and allows to compute the sign of a number. For real algebraic numbers, both can be done using the representation as polynomial with isolating interval and other representations [**62, 114**].

Limits of Exact Arithmetic. All exact arithmetic, be it with rational, bigfloat or real algebraic numbers is essentially performed by reducing the computation to multiple operations with exact integer arithmetic. Efficient algorithms for exact integer arithmetic are known, see e.g., [**110**] and implementations of exact arithmetic are widely available [**37, 53, 71**]. Unlike hardware arithmetic, the few limitations of software number types are rarely relevant in practice. Using software arithmetic obliterates any robustness problems and nearly all Exact Geometric Computation solutions use them as a foundation around which more efficient solutions are build.

We can compute arbitrarily accurate approximations of nearly any number given by an expression, e.g., by using bigfloat arithmetic. But it is important to note, that this alone does not suffice to compute the sign of a number! Assume you are presented with a number x in form of a black box. The black box allows you to compute arbitrarily small intervals containing x, but will never give you an interval of zero width. If at some point the returned interval does not contain zero, the sign of x is known. If however x equals zero, you will never be able to detect this. Due to the problem of zero detection, not much is known beyond real algebraic numbers. While Chang et al. [**17**] show that a special geometric problem involving transcendental numbers can be solved exactly, it is in general unknown whether the sign of an expression can be computed, if the functions exp or log are admitted [**90**].

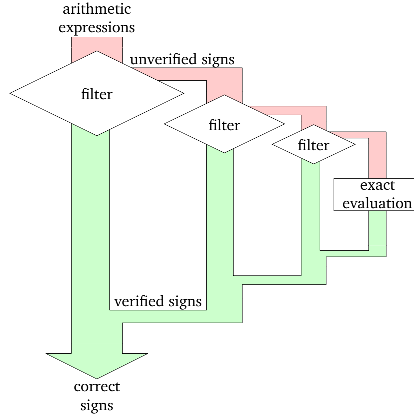


FIGURE 1.11. A cascade of arithmetic filters.

ARITHMETIC FILTERING. For the use in geometric algorithms, exact software arithmetic does have one big disadvantage. A single arithmetic operation is usually several orders of magnitude more expensive than the corresponding operation with hardware arithmetic. This is exacerbated by the fact that for most predicate evaluations, the hardware supported arithmetic is fully sufficient. This is evident in figures 1.5, 1.6, and 1.9, where points sufficiently far away from the line are always classified correctly by the 2D orientation predicate. Indeed, if this weren't the case, floating-point based implementations of geometric algorithms would not fail occasionally but be totally useless.

Relief from the high cost of exact arithmetic comes in the form of *arithmetic filters*. Instead of computing a value x exactly, an arithmetic filter computes an interval I that contains x. If zero is not contained in I, the sign of x is known, which is what we are interested in after all. If the sign of x remains unknown, it must be computed by other means. One might resort to using a better filter, which computes a tighter interval. This way, arithmetic filters may be cascaded, with a simple but fast filter first and more sophisticated and more expensive filters at later stages, cf. Figure 1.11. The last stage must consist of an approach that can recognize zero signs, e.g., by computing x exactly.

On the number type level, arithmetic filtering is synonymous to *interval arithmetic*. Given intervals I_x and I_y and an arithmetic operation $\circ$, we need to be able to compute an interval I such that

$$\{x \circ y \mid x \in I_x, y \in I_y\} \subseteq I. \tag{1.8}$$

Hence, if $x \in I_x$ and $y \in I_y$ then $x \circ y \in I$. This is called the inclusion property of interval arithmetic. There are two common interval representations. The first is by midpoint and radius of the interval, which can be interpreted as approximation $\hat{x}$

and corresponding error bound e_x for the unknown vale x. Approximation and error must satisfy $|x - \hat{x}| \leq \mathsf{e}_x$ and the sign of x is known if $|\hat{x}| > \mathsf{e}_x$. The second interval representation stores lower bound $\underline{x}$ and upper bound $\bar{x}$ of the interval. The bounds must satisfy $\underline{x} \leq x \leq \bar{x}$ and the sign of x is known if $0 \notin [\underline{x}, \bar{x}]$.

An arithmetic filter is called a *floating-point filter* if it is based on (hardware) floating-point arithmetic. The straightforward floating-point approximation $\hat{x}$ of an expression very often has the correct sign. All that is needed is an error bound e_x, that can be used to verify that the sign is indeed correct. Christoph Burnikel, Stefan Funke, and Michael Seel [**13**] show how to compute such an error bound with floating-point numbers. The challenge in this approach is to account for second order errors occurring in the computation of the bound. Hervé Brönnimann, Christoph Burnikel and Sylvain Pion [**5**] take the other approach and represent intervals by endpoints. They show how to implement interval arithmetic very precisely by using the directed rounding modes of floating-point arithmetic. For the practitioner, there are only minor differences between both approaches. The first approach is usually a bit faster, because it avoids switching the rounding mode, while the second produces more accurate intervals, since no second order error terms occur. It is not a priori clear, which is the better one for a particular application.

Expression dag Based Number Types. The lazy sign computation schemes of expression dag based number types are heavily based on arithmetic filters, both floating-point filters and bigfloat filters. Combining an exact number type with a floating-point filter alters the running time needed for an exact sign computation. If the filter succeeds to compute the sign, sign computation will be fast. If the filter fails, the sign computation will be only slightly slower than with the exact number type alone. Adding more filter stages, in general improves the adaptivity of a sign computation solution, but also increases the worst case running time which is reached by degenerate cases. The strategy of arithmetic filtering pays off for geometric computation because degenerate and nearly degenerate cases are usually rare. We discuss internals of expression dag based number types in much more detail in Section 2.1.

1.3.2. Exact Kernels and Predicates. On the level of geometric primitives, more efficient strategies for Exact Geometric Computation are available. These usually require more knowledge and work from the implementor, but can lead to largely improved performance with respect to techniques on the number type level.

Representation of Geometric Objects. For a start, one may modify the representation of basic geometric objects in a geometric kernel. One way to do this, is to use *homogeneous coordinate representation*. Homogeneous coordinates use an additional coordinate acting as common denominator to the other coordinates. A tuple

$$(h_1, \ldots, h_d, h_{d+1}) \in \mathbb{R}^{d+1} \qquad \text{with} \qquad h_{d+1} \neq 0$$

represents the Cartesian point with coordinates

$$(x_1,\dots,x_d) \in \mathbb{R}^d \qquad \text{where} \qquad x_i = h_i/h_{d+1}.$$

Tuples differing by a non-zero scalar represent the same Cartesian point and are considered equivalent. When $h_{d+1} = 0$ is allowed, homogeneous points span the projective space, which is quite different from Euclidean space and care must be taken to account for the differences [**107, 112**]. Homogeneous coordinate representation allows to avoid division in many geometric applications and hence may lead to more robust or more efficient implementations.

Another way to change the representation is, to lift the expression dag technique from the arithmetic level to the geometric kernel. When for example computing the intersection point p of two lines ℓ_1 and ℓ_2, a dag node for p is created, labeled as intersection node and storing references to ℓ_1 and ℓ_2. Additionally, a floating-point interval approximation for the coordinates of p is computed. This interval approximation is used for further computations, i.e., constructions and predicates involving p. Only if the interval representation turns out to be insufficient, the coordinates are computed exactly. A first implementation of this technique is described in [**32**], the more generic CGAL implementation is described in [**83**]. In comparison to expression dags at the arithmetic level, this approach has the advantage that significantly fewer intermediate results must be stored, which results in lower cost for dag construction and lower memory usage. Furthermore, arithmetic operations, first interval, then exact arithmetic, are performed en bloc, which allows further improvements.

GEOMETRIC FILTERING. Storing the creation history of a geometric object, in an expression dag or otherwise, allows to use this information later on in predicate implementations. Let again p be the intersection point of two lines ℓ_1 and ℓ_2. After computation of p, an algorithm may ask on which side of ℓ_1 the point p is located. If p still knows that it is the intersection of ℓ_1 and ℓ_2, the answer can be given without resorting to arithmetic. Note that this is a degenerate case which is expensive for arithmetic evaluation. Exploiting the creation history of geometric objects in predicate implementations is called *geometric filtering*. It has been developed for algorithms, where in a preprocessing step input objects are split into pieces. The splitting step results in many degenerate configurations, which are later caught by geometric filtering and would otherwise lead to expensive arithmetic evaluations [**50, 111**].

FLOATING-POINT FILTERS. When implementing geometric predicates, information not available on the arithmetic level can be exploited. First of all, the expression which must be evaluated is known. Thus it can be re-evaluated several times, without having to store it explicitly. Again, arithmetic operations on the same number type are performed en bloc. Often upper bounds on the input numbers are known. If, for example, the input numbers are hardware integers, a crude upper bound follows simply from the integer type. If no upper bound is known, it may be computed in a preprocessing step from input data.

On the arithmetic level, floating-point filters must be fully *dynamic*, i.e., the interval containing the true result is entirely computed at runtime. On the predicate level, floating-point filters often use the interval representation by midpoint and radius, respectively approximation and error bound. The approximation is almost always the straightforward floating-point evaluation of the expression, computed at runtime. But knowing the expression and/or upper bounds on the input numbers allows to compute an error bound partially or completely in advance. In a *static filter*, the error bound is entirely precomputed. Thus, a static filter takes the same runtime as a straightforward floating-point evaluation, can however not determine the sign for all predicate calls. In a *semi-static filter* the error bound is partially precomputed. The improved speed of static and semi-static filters comes at a cost. The error bounds are necessarily less sharp, hence the filter will verify the sign of an approximation less frequently than dynamic filters. For these reasons, state of the art predicate implementations use at least three stages. First, a static or semi static filter, then a dynamic filter, and finally exact arithmetic is employed. Static and semi-static filters are for example described in [**13, 23, 30, 104**].

Exact Arithmetic for Small Precision. Knowing the expression and upper bounds on the input, also allows to predict the arithmetic requirements for exact evaluation. If, for example, the expression is a polynomial of degree d and all input numbers are integers with b bits, then all intermediate results can be represented exactly using $d(b + O(1))$ bits of precision [**56**]. For many basic predicates, the requirements for exact arithmetic are quite low. The 2D orientation predicate determinant has a degree of 3. For 32 bit integer input, it can be evaluated exactly using a precision of 100 bits. Software number types tend to be optimized for much larger precision. Knowing the precision requirement in advance, allows to fine tune the necessary final exact evaluation step. One can save dynamic memory allocation and might use algorithms which are asymptotically suboptimal, but still faster for small precision. Fortune and van Wyk [**30**] develop small precision exact integer arithmetic, Shewchuk [**104**], based on work by Douglas Priest [**86**] and Theodorus Dekker [**20**], presents efficient exact ring operations for small precision over floating-point input. These form one set of algorithms used to defer expression dag creation in Chapter 5. We have a look at Shewchuk's work and the techniques it is based on in more detail in Section 2.2.

Based on his exact arithmetic operations, Shewchuk develops multi-stage 2D and 3D orientation predicate and insphere and incircle predicate implementations. The arithmetic operations allow to compute partial results without knowing the arguments completely. Partial result can be completed without recomputing the already known parts. This is exploited to efficiently compute approximations of the final result and corresponding error bounds with increasing quality. The predicates consist of four stages of increasing complexity and are really unique in that partial results from previous stages are reused and not recomputed in later stages. The

evaluation scheme and the error bounds are carefully crafted by hand and result in one of the most efficient predicate implementations to date.

Static Algebraic Predicates. Like rational arithmetic, exact arithmetic on algebraic numbers reduces a single operation to several exact operations in integers. In the case of predicates involving algebraic numbers of small degree, several authors perform this reduction manually, resulting in *static algebraic predicates*. Olivier Devillers, Alexandra Fronville, Bernard Mourrain, and Monique Teillaud [**22**] consider comparing the x-coordinates of intersection points of circles. More generally, Ioannis Emiris and Elias Tsigaridas [**27**] discuss approaches to compare the roots of polynomials of small degree. The basic approach is to unroll general algorithms for the computation with algebraic numbers to the specific case at hand. Using symbolic computation techniques, the authors precompute expressions for quantities occurring in these algorithms, like isolating intervals, resultants, or Sturm sequences. They further simplify these expressions by exploiting common sub-expressions and simple algebraic identities between them. The result are polynomial expressions of small degree in the input data. At runtime, only the sign of one or more of these expressions must be computed to determine the final comparison result. Furthermore, for the evaluation of an expression, the whole set of available techniques for adaptive evaluation may be emplaced, resulting in very efficient predicates.

Modular Arithmetic. Finally, *modular arithmetic* [**110**] may be used for sign computation of polynomial expressions over integers. The approach requires an upper bound on the final value of an expression, but not on intermediate results. A good bound can be induced by prepending modular arithmetic with a floating-point filter. Either the filter verifies the sign, or the value of the expression must be small. Thus, floating-point filter and modular arithmetic complement each other, with the floating-point filter providing smaller bounds for the modular arithmetic. To avoid high precision arithmetic, the expression is evaluated multiple times with relatively prime moduli, small enough to allow evaluation with hardware arithmetic. From the residues, the exact value may be reconstructed using software arithmetic, the sign, however, can be computed using hardware arithmetic alone [**6, 82**].

Automatic Predicate Generation. Designing and implementing efficient geometric predicates is a difficult task, mostly due to the error analysis that must be undertaken. To ease the implementation of multi stage geometric predicates, *expression compilers* have been devised. The compiler is fed with the expression, and possibly further information on the input numbers and generates the code to compute the sign of the expression for given input numbers correctly. For static and semi-static floating-point filters the compiler performs the necessary error analysis. Fortune and van Wyk [**30**] present an expression compiler, called LN for “little numbers”, based on their static floating-point filter and small precision exact arithmetic. The compiler generates `C++` code for both the filter stage and the exact evaluation. The exact evaluation code is completely unrolled, avoiding any memory allocation, function calls and even loops. Burnikel et al. [**13**] provide an expression compiler EXPCOMP

for their floating-point filter too. It generates C++ code and uses number types from the LEDA library for exact evaluation. Aleksandar Nanevski, Guy Blelloch, and Robert Harper [**76**] present an expression compiler that automatically generates multi-staged predicates following Shewchuk's scheme. The generated predicates however fall slightly short of the ones that were hand made by Shewchuk. Nevertheless, the compiler allows to generate more than the four predicates provided by Shewchuk. Expression compilers ease the generation and implementation of efficient geometric predicates, but they are restricted to a certain environment, i.e., the output language or presence of certain libraries.

1.3.3. Robustness on the Combinatorial Layer. There exists an abundance of techniques on the combinatorial level that allow geometric algorithms to cope with numerical imprecision, for an overview see [**97, 117**]. The approaches are often problem dependent and as diverse as the geometric problems they address. Very often, they change the notion of what is considered a solution to a problem. Output may lack certain topological properties exhibited by the exact solution, or the correct output for something different than the actual input is computed. However, they guarantee termination and that computed output is in some sense close to the exact solution. In this regard, they can be considered robust. By avoiding expensive exact arithmetic totally or at least more than exact approaches, they may potentially lead to very efficient implementations. In this section, we briefly present a selection of approaches on the combinatorial level that have been successfully applied to a variety of problems.

STRUCTURAL FILTERING. *Structural filtering* [**33, 75**], although working on the combinatorial level, is a technique which still belongs to Exact Geometric Computation. It applies to searching in data-structures, an important sub-step in many algorithms. A frequent search step in geometric algorithms is point location, where the part in a data-structure corresponding to a certain coordinate must be found. The idea of structural filtering is, to perform the search in a first step with inexpensive though inaccurate floating-point based predicates. Since these predicates return false answers rarely and only in nearly degenerate cases, the search should stop at or close to the true result in the data-structure. Then, exact predicates are used to verify and possibly correct the position in the data-structure.

CONTROLLED PERTURBATION. The technique of *controlled perturbation* [**41**], devised by Dan Halperin and co-workers, randomly perturbs, i.e., actually changes the input by a small amount and then attempts to compute the geometrically correct output for the modified input. The goal of the perturbation is to remove degenerate and nearly degenerate input configurations in such a way, that an arithmetic filter with predetermined accuracy will compute the correct sign for all predicate calls. If at some point the filter fails to determine a sign, the procedure is repeated. There is a tradeoff between the amount of perturbation and hence the quality of the result, the accuracy of the employed arithmetic filters and the probability that a restart is necessary. Controlled perturbation has been applied successfully to a variety of

problems, including arrangements of spheres and circles [**40, 41**] and Delaunay triangulations [**31**]. Kurt Mehlhorn, Ralf Osbild and Michael Sagraloff generalize controlled perturbation to a wide class of geometric algorithms [**59**].

Topology Oriented Implementation. The *topology-oriented approach* has been devised and popularized by Kokichi Sugihara and co-workers. They applied it successfully to many geometric problems, including the computation of convex hulls, Voronoi diagrams, Delaunay triangulations, and the intersection of polyhedra, see [**109**] for an overview of the technique and references to individual problems. The basic idea is to separate topology and combinatorics from imprecise numerics in order to avoid the inconsistency problem. Combinatorial invariants are determined, which hold for an exact solution to a geometric problem but can be enforced without resorting to numerical computation. Queries and updates to data-structures are designed to rely on these invariants only and to enforce them rigorously. Information from numerical computations may only guide these operations, combinatorial information is given strict precedence. The goal is to design an algorithm in such a way, that something combinatorially sound is computed, even if all numerical predicates give random answers. The algorithm shall give the correct answer in the limit, as the numerical precision tends to infinity. Since floating-point based predicates quite often return correct results, there is hope the computed result is close to the correct one. By design, the topology-oriented approach does not handle degeneracies. As an example, we briefly discuss an incremental algorithm for the two dimensional Voronoi diagrams of points [**108**]. The algorithm inserts sites one by one, maintaining the current Voronoi skeleton as a graph. The combinatorial invariants for the topology oriented approach are, that each site has exactly one connected Voronoi region, and two Voronoi regions share at most one edge. In an update step, the part of the old Voronoi skeleton covered by the new Voronoi cell is removed and replaced by the boundary of the new cell. The invariants are enforced by ensuring that the removed part always has a tree structure. Only the selection of the tree to be removed from the current Voronoi skeleton is guided by numerical computation.

Martin Held [**42**] reports on his implementation of an incremental algorithm for the computation of the Voronoi diagram of line segments. Computing the Voronoi diagram of line segments involves algebraic numbers and is hence quite challenging in terms of arithmetic and robustness. His code, called VRONI, is based on the topological approach and uses hardware floating-point arithmetic only. VRONI has been shown empirically to be both reliable and very fast. Its creation is a beautiful example for Algorithm Engineering.

1.4. Robust Geometric Computing in Practice

The Exact Geometric Computation paradigm has been been widely successful because it allows to implement algorithms as described in theory and leads to robust implementations. Other approaches to the robustness problem are less attractive as they require to re-invent algorithms or compute only approximate solutions. But, as

in the case of VRONI, they may still lead to very efficient implementations which are reliable in practice. Provable guarantees on robustness and output quality remain an issue.

LIBRARIES FOR EXACT GEOMETRIC COMPUTATION. Many geometric algorithms have been implemented successfully following the Exact Geometric Computation paradigm and are now available to the scientific community, industry, and the general public. Among the many approaches, the libraries LEDA [**53, 58**] and CGAL [**16**] stand out. Most of the techniques discussed in this chapter are part of one or both libraries. Both LEDA and CGAL can be considered a product of Algorithm Engineering in Computational Geometry.

LEDA is the first library to follow the Exact Geometric Computation paradigm rigorously. The decision for Exact Geometric Computation was made following disappointing results with floating-point based implementations and the finding that problem dependent approaches to robustness are unsuitable in a library, which should be open to many applications. LEDA contains an extensive collection of geometric algorithms and data-structures both in two and three dimensions. Tools for Exact Geometric Computation are provided, too. LEDA contains exact number types and interval arithmetic, as well as exact geometric kernels. Besides geometry, LEDA also provides an extensive set of combinatorial algorithms.

CGAL concentrates exclusively on geometry. It is the product of more than 15 years of work by the computational geometry community and now contains a much larger body of geometric algorithms than LEDA. CGAL is a more recent development, too, and could therefore take advantage of C++'s template feature which was not available when the development of LEDA started. By means of templates, CGAL very nicely separates the arithmetic layer, the layer of geometric primitives and the combinatorial layer. The three layers can be combined easily and quite independently. CGAL is not only the product of Algorithm Engineering but, due to its generic design, makes further Algorithm Engineering much easier and thereby possible. This thesis has profited immensely from CGAL, almost all experiments use kernels and geometric algorithms from CGAL.

COST OF EXACT GEOMETRIC COMPUTATION. Despite the apparent advantages of Exact Geometric Computation, practitioners still complain about its lack of efficiency. Often, they compare the efficiency to the speed of floating-point based implementations, but this comparison is not quite fair, since floating-point based implementations are not robust. Nearly all papers presenting new techniques for Exact Geometric Computation compare their new approach experimentally to other methods for performance. They often include straightforward floating-point based implementations too, since they provide some sort of lower bound. But, not surprisingly, as soon as problems are more challenging, the times for the floating-point based implementation are missing because of robustness problems. We now summarize some experimental results comparing Exact Geometric Computation approaches to each other and to other robust and non-robust approaches.

The input size of geometric algorithms, basis for asymptotic performance analysis, is usually the number of basic geometric input objects. The input size affects the number of predicate calls, not the time taken by a single predicate. Thus, exchanging predicate implementations in a geometric algorithm alters the running time roughly by a constant factor, independently of the input size. Assuming all predicates in an algorithm are sped up by a factor s, the algorithm will be sped up by a constant factor somewhat smaller than s, regardless of the input size. In case of adaptive predicates, the average speed up factor for a predicate depends on the number of nearly degenerate predicate evaluations. Hence, experiments are usually run for several inputs with varying number of degenerate configurations, but not so much for inputs of varying input size.

Predicates with Integer and Rational Numbers. Michael Karasick, Derek Lieber and Lee Nackman [**49**] are among the first to report on the use of exact arithmetic in geometric algorithms. They implement an algorithm for the computation of the 2D Delaunay triangulation of input points with floating-point coordinates. They report a slowdown factor of 10 000 when naïvely using rational arithmetic, compared to a floating-point based implementation. Clearly, with such a performance hit, one would abandon Exact Geometric Computation despite all problems with different approaches. Karasick et al. manage to improve the slowdown to a factor of 5, by using first filtering techniques in an adaptive scheme, and modifying the algorithm to use less arithmetic.

Several authors report on the efficiency of their predicate implementations for the computation of 2D and 3D Delaunay triangulations. Fortune and van Wyk [**29**] compute the 2D Delaunay triangulation of integer points. Compared to a floating-point based implementation, their predicates achieve a slight decrease in running time for input sets without degeneracies, for nearly co-circular points they report an increase in running time by 65%. Shewchuk [**104**] computes 2D and 3D Delaunay triangulation of points with floating-point coordinates. In the two dimensional case, the overhead for exactness is 8% to 30% for input sets with varying amounts of degenerate configurations, in the three dimensional case the overhead is between 30% for uniformly distributed points and a factor of 11 for co-spherical points. Devillers and Pion [**23**] compute the 3D Delaunay triangulation of points with floating-point coordinates for some uniformly at random and real world input data sets. For Shewchuk's predicates they report an increase in running time of 40% compared to floating-point predicates. Their own best exact predicates achieve an overhead of 8% to 25% only. Devillers and Pion include expression dag based number types `CORE::Expr`, `leda::real`, and `Lazy_exact_nt` in their experiments too. For random input data, `CORE::Expr` is the fastest among the three, while for real world data `Lazy_exact_nt` wins. Comparing to Shewchuk's adaptive predicates, the slowdown is at least a factor of 10 in any case.

The predicates for all problems mentioned so far consist of small polynomial expressions. Evidently, Exact Geometric Computation has been achieved for them

at an increased runtime cost well below a factor of two, except for cases with many degenerate input configurations. These are however the cases where Exact Geometric Computation is needed the most. This is a small price to pay for correct results and tolerable in all but the most runtime restricted applications. For these applications, expression dag based number types do not appear to be efficient.

PREDICATES WITH ALGEBRAIC NUMBERS. Christoph Burnikel, Rudolf Fleischer, Kurt Mehlhorn and Stefan Schirra [**10**] somewhat artificially design problems that require the power of expression dag based number types. For example, they first compute all intersection points induced by a set of lines or circles and then compute the convex hull of these intersection points. Here, `leda::real` outperforms both `CORE::Expr` and non-adaptive predicates based on exact number types. With respect to a floating-point based implementation, the slowdown is a factor of 15 to 20 for circles, and 25 to 60 for lines.

For problems involving algebraic numbers comparing exact approaches to non-robust floating-point based approaches is meaningless, even when the input data are hardware numbers. The robustness problems become so severe that no straightforward floating-point based implementation might work. Because exact approaches are still felt to be too slow though, it is very interesting to look for the most efficient exact approach.

Ioannis Emiris, Athanasios Kakargias, Sylvain Pion, Monique Teillaud, Elias Tsigaridas [**26**] present a new geometric kernel for the computation of arrangements of circles and circular arcs. To handle algebraic numbers occurring in this problem they either employ `CORE::Expr` or `leda::real`, or static algebraic predicates based on integer and rational number types [**22, 27**]. In their experiments, they compute arrangements for randomly generated input sets and structured input sets with many degeneracies. Among expression dag based number types, `leda::real` always outperforms `CORE::Expr`. However, `leda::real` is a factor of two to four slower than static algebraic predicates for an optimal choice of number type.

Menelaos Karavelas [**50**] reports on his implementation of segment Voronoi diagrams, another problem involving algebraic numbers. In his experiments he computes segment Voronoi diagrams for randomly generated data, structured data with many degeneracies and real world data. For predicates he uses `leda::real`, and alternatively static algebraic predicates based on rational numbers from GMP. Both variants are combined with a floating-point filter and achieve essentially the same running time. The reason might be, that the floating-point filter computes most of the signs.

More recent results involving expression dag based number types are presented by Martin Held and Willi Mann [**43**]. Among other problems, they examine the problem of computing the Voronoi diagram of segments, using robust and exact approaches. The first competitor is their own code VRONI, based on floating-point numbers as well as bigfloat numbers from MPFR. The MPFR based variant uses fixed precision which is higher than that of hardware arithmetic. It is therefore more

likely to produce correct topology and computes more accurate numerical data. The second competitor is the implementation of segment Voronoi diagrams by Karavelas, combined with CORE::Expr as exact number type. As input data, Held and Mann use a data base of roughly 20 000 input sets. The results are, that both Karavelas implementation with CORE::Expr, and VRONI based on MPFR are on average a factor of 50 – 80 slower than VRONI with floating-point based predicates. Furthermore, for Karavelas implementation, the running times are much more scattered and some outliers did not even finish computing in the time limit set for the experiments. A slowdown by a factor of 50 seems to be a hard price to pay for exactness or more accuracy.

These results might disencourage an implementor to use expression dag based number types, despite other advantages. For linear problems, Exact Geometric Computation seems achievable at a slowdown of less than two, while the cost for using expression dag based number types is much higher. For problems involving algebraic numbers, where it is reasonable to spend more time, static algebraic predicates appear to be the better choice. In some cases, non-exact but much faster implementations like VRONI may be used.

On the other hand, many of the experimental results given here are outdated. Linear problems rely on floating-point filters for speed. But floating-point filters are unlikely to have been improved since then or to be improved in the future. The reasons are, that they are quite simple and for easy predicates are so fast that there is just no room for improvement. Expression dag based number types however are internally quite complicated and it is reasonable to believe that progress can be made to narrow the gap.

The issue with floating-point filters for arithmetically more demanding problems is not lack of speed, but that floating-point filters fail more frequently and computation with software number types becomes necessary. But adaptivity beyond the limits of dynamic floating-point filters is only available in expression dag based number types. The running times achieved with static algebraic predicates are not so small, as to seem unreachable. The main goal of this thesis is, to improve expression dag based number types to close the performance gap to other, more specialized approaches.

CHAPTER 2

Previous Work

In this chapter we briefly discuss work by other authors directly relevant to the topics of this thesis. It is actually a chapter split in two. In the first part we closely examine the techniques needed to make expression dag based number types work. Then, we look more closely at some number types, and examine some strengths and weaknesses in their implementation. We concentrate on `CORE::Expr` and `leda::real` as they are most closely related to *RealAlgebraic*.

In the second part we discuss, how notoriously imprecise floating-point arithmetic can be used to design and implement numerical algorithms that compute very precise and even exact results. We start with some well known and less well known approaches to the analysis of floating-point arithmetic. Then we present error-free transformations, the basic toolkit for exact arithmetic based on floating-point numbers and finally some algorithms using them.

2.1. Expression Dag Based Number Types

An expression dag is a directed acyclic graph with labeled nodes. Nodes without outgoing edges are called *leaf* nodes. Leaf nodes are labeled with an integer or rational number. All other nodes are called *internal* nodes, they are labeled with an arithmetic operation. The number of outgoing edges of an internal node corresponds to the arity of the label. We call the direct successors of a node the *operands* of this node. An expression dag has exactly one node without ingoing edges, called the *root* node. We may however consider any node in a dag as a root node, by restricting the dag to the part reachable from this node. An expression dag represents an arithmetic expression in the straightforward way. A leaf node represents its label, and an internal node represents the expression obtained by applying its label to the expressions represented by its operands. Figure 1.1 in the previous chapter shows an expression dag and the represented expression. Note how the sub-expression $\sqrt{2}$, which occurs twice in the expression, is shared in the dag. Each dag node has a value, obtained by evaluating the expression it represents. The value is undefined if some undefined operation occurs, e.g., division by zero. We identify a dag node with both its expression and value.

Arithmetic operations we allow as label, are negation or unary minus ($-$), binary field operations, ($+, -, \times, /$), radicals of arbitrary degree ($\sqrt[d]{\ }$), and more generally extracting the i-th largest real root of a polynomial p. This is called the $\diamond$-operator

($\diamond(p,i)$). The operands for a $\diamond$ node are the coefficients of p. With this set of operations, the value of a dag node is either undefined or a real algebraic number. Sometimes the operands for the $\diamond$-operator are restricted to leaf nodes, i.e., integer or rational numbers. We call an expression *simple*, if it does not contain the $\diamond$-operator and we call it *division free*, if it does not contain a division. *General* expressions may contain all of the operations above.

2.1.1. Basic Techniques. Expression dag based number types record the creation history of a number as an expression dag. They defer numerical computations to the point where a user asks for the sign or approximate value of a number. Storing the expression allows to re-evaluate the expression. This has two main advantages. First, the number is known exactly. Although the representation as expression dag is rather implicit, any conceivable information about a number might be computed if necessary. Second, having said that, the sign or an approximation of the expression can be computed using as much numerical precision as necessary, but not more. Thus, expression dag based number types allow for adaptive sign computation.

The user does not need to care about the internals. He only sees variables, representing numbers, which he can manipulate in an accustomed way. In practice, this variable is a *handle*, pointing to a dag node and the dag rooted at this node represents the number. See again Figure 1.1, where two variables `a` and `b` are used in the code sample. From the users viewpoint, arithmetic operations simply return a new variable representing the result. Expression dag based number types mostly differ in how expression evaluation and sign computation are performed. We now collect some of the main techniques used.

STAGED FILTERS. The main reason for storing expressions is to defer high precision computations, since they are not necessary in most cases. But a user will rarely create a number without needing its sign or value at all. Therefore, each node v stores a dynamic floating-point filter or floating-point interval containing v. This filter is usually computed upon dag creation and the first means for sign computation. If the node is approximated more accurately in later stages, the filter may be updated. This allows to compute tighter floating-point intervals in arithmetic operations involving v later on.

Some number types inspect the dag structure to detect zero signs or equality of nodes, right after the floating-point filter and before switching to software number types. For example, two nodes are equal if they have the same address in memory. But it is also possible, to check the dag structure recursively. Leaf nodes are equal if they store the same number, and internal nodes are equal, if they have equal operands and are labeled with the same operation.

If the floating-point filter and other quick tests fail to give the correct sign or a sufficiently accurate approximation, software number types are used. One possibility is to compute the value of a node exactly using an appropriate number type. The alternative is, to continue computing approximately, using bigfloat arithmetic.

Approximation with Bigfloat Arithmetic. For each real number x and a fixed precision of p bits, there are two bigfloat numbers $\text{pred}(x)$ and $\text{succ}(x)$, such that

$$\text{pred}(x) \le x \le \text{succ}(x),$$

and no bigfloat number strictly between them. A bigfloat operation with p bit precision is rounding *faithfully*, if the result is as if first the exact result x is computed, and then rounded to $\text{pred}(x)$ or $\text{succ}(x)$. We assume that all bigfloat operations are rounding faithfully. A rounding mode additionally determines which of the two numbers to choose. Common rounding modes are for example rounding to nearest, or rounding upwards. While the first is slightly more accurate, the latter allows to compute rigorous bounds more easily. We can control the relative error of a bigfloat operation by specifying the precision p. Let $\mathbf{fl}(x)$ be a bigfloat number nearest to x, then

$$|x - \mathbf{fl}(x)| \le 2^{-p}|x| \qquad \text{and} \qquad |x - \mathbf{fl}(x)| \le 2^{-p}|\mathbf{fl}(x)|.$$

To keep track of the error accumulating in a series of bigfloat operations, arithmetic filters are used. Assume we use an interval representation based on midpoint and radius as filter. In this case, each node v stores an approximation $\hat{v}$ and an error bound e_v with

$$|v - \hat{v}| \le \mathsf{e}_v.$$

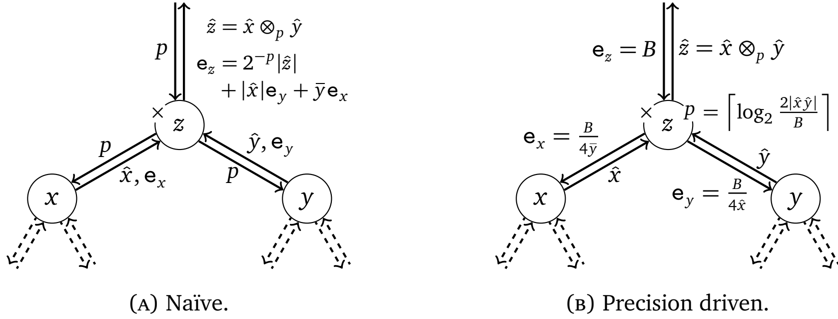
We may however easily switch between different interval representations by accepting some small additional error. For example, an upper bound $\bar{v}$ for $|v|$ can be computed by $\bar{v} = |\hat{v}| + \mathsf{e}_v$ with bigfloat arithmetic in any precision and rounding upwards.

We denote by $\otimes_p$ a bigfloat multiplication with p bit precision and faithful rounding. Let z be a multiplication node with operands x and y, i.e., $z = x \times y$. Recompute $\hat{z} = \hat{x} \otimes_p \hat{y}$. Then we have the following error estimate

$$\begin{aligned} |\hat{z} - z| &\le |\hat{z} - \hat{x}\hat{y}| + |\hat{x}(\hat{y} - y)| + |y(\hat{x} - x)| \\ &\le 2^{-p}|\hat{z}| + |\hat{x}|\mathsf{e}_y + \bar{y}\mathsf{e}_x \end{aligned} \tag{2.1}$$

We can compute an upper bound of the right hand side, again using bigfloat arithmetic, but now with small precision and rounding upwards. This gives us the error bound e_z. Similar error bounds exists for other arithmetic operations. By increasing the precision p and recomputing all nodes, arbitrarily accurate approximations and corresponding error bounds may be computed for a node. But there remain two questions: How to increase the precision? And how to detect zero numbers, since except for polynomial expressions, approximations can not be expected to become exact at any point.

Precision Driven Arithmetic. In the way just described, we can determine the accuracy of an approximation. *Precision driven arithmetic* allows to specify the accuracy of an approximation a priori, before it is recomputed. The scheme works recursively and in general requires to recompute the approximation of child nodes. It terminates at the leaves of the dag, where exact values are available. The technique was first proposed by Thomas Dubé and Chee Yap [**25**]. Here we give the error estimates and evaluation scheme for a multiplication node and absolute error, as



(A) Naïve. (B) Precision driven.

FIGURE 2.1. Improving the approximation of a multiplication node. Figures inspired by similar ones in [**115**]

presented in [**10**]. Schemes for more operations, both for relative error and absolute error are presented by Yap [**116**] and Du [**24**].

Let again z be a multiplication node with operands x and y. Analogously to Equation (2.1) we obtain

$$|\hat{z} - z| \leq 2^{-p}|\hat{x}\hat{y}| + |\hat{x}|e_y + \bar{y}e_x \tag{2.2}$$

This bound allows us to recompute $\hat{z}$ for given e_z, by bounding the terms on the right hand side appropriately. Assume we want to achieve $e_z = B$. First, we recompute $\hat{x}$, recursively enforcing $e_x \leq B/(4\bar{y})$ by precision driven arithmetic. This step requires an upper bound $\bar{y}$ on $|y|$, which we can easily compute using the current approximation of y. Afterwards, $\hat{x}$ is known and we recompute $\hat{y}$, enforcing $e_y \leq B/(4|\hat{x}|)$. Finally we set p, such that $2^{-p} \leq B/(2|\hat{x}\hat{y}|)$ and compute $\hat{z} = \hat{x} \otimes_p \hat{y}$. Then clearly $|\hat{z} - z| \leq B$ and we set $e_z = B$. See Figure 2.1 for comparison of a naïve improvement scheme and precision driven arithmetic.

Analogous schemes are known for other basic arithmetic operations and functions. In some cases, i.e., for division and radicals, they involve lower bounds. Having a lower bound on the absolute value of some node is equivalent to knowing its sign, so sign computation may be triggered recursively. Precision driven arithmetic allows to control the accuracy of an approximation to some expression in advance. This is in contrast to bigfloat interval arithmetic, which can compute arbitrarily accurate approximations and proper error bounds too. There, however, the accuracy is only known after the computation.

ZERO SEPARATION BOUNDS. If expressions are evaluated approximately only, *separation bounds* allow to identify nodes with a value of zero. A number $\xi(E)$ is a separation bound for some expression E, if

$$E \neq 0 \qquad \Rightarrow \qquad |E| \geq \xi(E).$$

If a separation bound is known, we can approximate E such that $2\mathtt{e}_E < \xi(E)$. Then, the sign of E is known because either $|\hat{E}| > \mathtt{e}_E$, which implies

$$\operatorname{sign}(E) = \operatorname{sign}(\hat{E}) \neq 0$$

or $|\hat{E}| \leq \mathtt{e}_E$ and

$$|E| \leq 2\mathtt{e}_E < \xi(E) \qquad \Rightarrow \qquad E = 0.$$

Actual separation bounds in use with expression dag based number types consist of a set of parameters for each dag node, among them the actual separation bound, and simple rules how to compute these parameters recursively from the operands of a node. They can be classified with respect to the type of expressions they support.

The parameters maintained in a separation bound for an expression E are typically bounds on quantities related to the *minimal polynomial* of the algebraic number α given by E. A polynomial $p \in \mathbb{Z}[X]$ is called a minimal polynomial for α, if $p(\alpha) = 0$ and the degree of p is minimal among all polynomials in $\mathbb{Z}[X]$ having α as root. Let

$$p(X) = \sum_{i=0}^{d} a_i X^i = a_d \prod_{i=1}^{d} (X - \alpha_i) \in \mathbb{Z}[X]$$

be a minimal polynomial of α, i.e., let $\alpha = \alpha_1$. The roots $\alpha_1, \ldots, \alpha_d$ of p are called *conjugates* of α. The proofs for separation bounds are mostly based on resultant calculus [**114**], which allows to compute the necessary polynomials. An upper bound $D(E)$ on the degree d of α occurs in nearly all known separation bounds. It is given by

$$D(E) = \prod_{\substack{v \text{ node in} \\ \text{dag for } E}} \deg(v) \qquad \text{where} \qquad \deg(v) = \begin{cases} d_i & \text{if } v = \sqrt[d_i]{} \\ \deg(p) & \text{if } v = \diamond(p, i) \\ 1 & \text{otherwise.} \end{cases} \tag{2.3}$$

A separation bound ξ *dominates* another separation bound ξ' for a class of expressions $\mathscr{E}$ ($\xi \succ \xi'$), if $\xi(E) \geq \xi'(E)$ for all $E \in \mathscr{E}$. A larger separation bound is clearly preferable, because it requires less accuracy from an approximation to compute a sign. Figure 2.2 gives an overview over known separation bounds and relations between them. A recent survey on separation bounds is given by Stefan Schirra [**99**].

Maurice Mignotte [**60, 61**] was the first to consider proving the equality of two algebraic numbers by approximation and separation bounds. He provides rules for maintaining upper bounds on the degree and *measure*

$$\mathscr{M}(\alpha) = |a_d| \prod_{i=1}^{d} \max\{1, |\alpha_i|\}$$

of a number α, for simple expressions. If $\alpha \neq 0$, then $\alpha > 1/\mathscr{M}(\alpha)$. We call this the *DM* or *degree-measure* bound. Chen Li and Chee Yap [**55**] improve the DM bound by computing tighter bounds on the degree using Equation (2.3) and add rules for the $\diamond$-operator restricted to integer coefficients. Hiroshi Sekigawa [**103**] improves upon

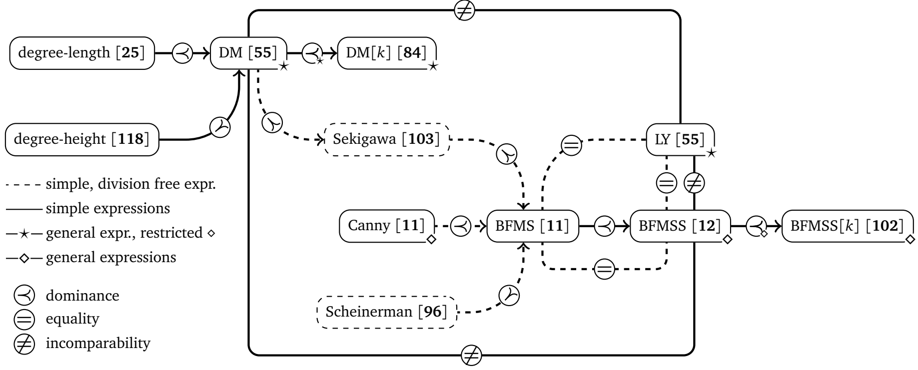


FIGURE 2.2. Relations between some known separation bounds.

the DM bound for simple, division free expressions, by maintaining upper bounds on $|a_d|$ and $\prod_{i=1}^{d} \max\{1, |\alpha_i|)\}$ separately.

The *BFMSS* bound by Christoph Burnikel, Stefan Funke, Kurt Mehlhorn, Stefan Schirra, and Susanne Schmitt [**12**] conceptually rewrites an expression E with value α into a quotient of two division free expressions. Let β be the value of the numerator in the rewritten expression, with minimal polynomial

$$q(X) = X^{d'} + \sum_{i=0}^{d'-1} b_i X^i = \prod_{i=1}^{d'} (X - \beta_i) \in \mathbb{Z}[X].$$

The conceptual rewriting rules guarantee, that the degree d' of β is at most the degree d of α. Let γ be the value of the denominator in the rewritten expression. The BFMSS bound maintains an upper bound $u(E)$ on the conjugates of β, and an upper bound $l(E)$ on the conjugates of γ. The rules for $u(E)$ and $l(E)$ are given in Table 2.1. If $\alpha \neq 0$, then

$$1 \leq |b_0| = |\beta_1 \beta_2, \ldots, \beta_d| \qquad \text{implies} \qquad |\alpha| = \frac{|\beta|}{|\gamma|} \geq \frac{1}{l(E)u(E)^{D(E)-1}}.$$

The BFMSS bound improves upon its predecessor, the BFMS bound [**11**] with a better conceptual rewriting rule, distributing the algebraic degree to either the numerator or denominator, not both. In the BFMS bound, the degree of β can only be bounded by $D(E)^2$. Together with the split rule for radicals, attributed to Chee Yap, the BFMSS bound dominates the BFMS bound. For simple, division free expressions, both bounds are equal.

The *LY* bound by Chen Li and Chee Yap [**55**] maintains an upper bound $U(E)$ on the conjugates of α, and an upper bound $\mathrm{lc}(E)$ on the leading coefficient $|a_d|$ of the

E	$u(E)$	$l(E)$
$E \in \mathbb{Z}$	$\lvert E\rvert$	1
$A \pm B$	$u(A)l(B)+u(B)l(A)$	$l(A)l(B)$
$A \times B$	$u(A)u(B)$	$l(A)l(B)$
A/B	$u(A)l(B)$	$l(A)u(B)$
$\sqrt[k]{A}$, $u(A) \geq l(A)$	$\sqrt[k]{u(A)l(A)^{k-1}}$	$l(A)$
$\sqrt[k]{A}$, $u(A) < l(A)$	$u(A)$	$\sqrt[k]{u(A)^{k-1}l(A)}$

TABLE 2.1. Rules for computing $u(E)$ and $l(E)$ for the BFMSS bound.

minimal polynomial of α. If $\alpha \neq 0$, then

$$1 \leq |a_0| = |a_d \alpha_1 \alpha_2, \ldots, \alpha_d| \qquad \text{implies} \qquad |\alpha| \geq \frac{1}{\mathrm{lc}(E)U(E)^{D(E)-1}}.$$

To allow computing $U(E)$ and $\mathrm{lc}(E)$ in the presence of division, the LY bound additionally maintains a lower bound on the conjugates, and upper bounds on tail coefficient and measure of the minimal polynomial. In one case, the LY bound resorts to the DM bound, though any separation bound may be used here. For simple, division free expressions, the LY bound and the BFMSS bound are equal.

Separation bounds are much worse for simple expressions than for division free expressions. The very common floating-point input, however, introduces division already at the leafs of a dag, even if the expression to evaluate is otherwise division free. Sylvain Pion and Chee Yap [**84**] develop a generic approach to overcome this issue. The basic idea is to factor a power of k from each number, before estimating the separation bound parameters. To this end, input numbers are scaled to the smallest integer, for computed numbers the corresponding power of k is computed recursively from the arguments like other separation bound parameters. If this computation can not be performed exactly, the error is moved into the original separation bound parameters. Pion and Yap apply their scheme to the DM and BFMSS bound and call the resulting bounds DM[k] and BFMSS[k]. Their new bounds dominate the old ones. In practice, one should use $k = 2$, or $k = 2$ and $k = 5$ simultaneously, because nearly all input numbers are binary or decimal. Experiments show that for floating-point input, the separation bounds computed by BFMSS[2] are almost as good as the BFMSS bound for integer input [**84**]. The BFMSS[2, 5] bound is not a large improvement over BFMSS[2]. Schmitt [**102**] extends the results of Pion and Yap to the $\diamond$-operator for the BFMSS bound.

For simple, division free expressions, the BFMSS[k] bound dominates all other known separation bounds. Most of the relevant proofs can be found in [**11, 55**]. A proof that BFMS dominates Sekigawa's bound is due to Stefan Schirra [**100**]. For simple expressions, the BFMSS bound, the LY bound, and the DM bound are

incomparable [**55**]. This means, there exist expressions where any of the three is the best known bound. The LY bound does however compute the DM bound anyway. Applications should therefore use either the BFMSS[2] bound alone, or the LY bound combined with the DM[2] bound, and the BFMSS[2] bound together.

Adaptive Evaluation. Separation bounds and precision driven arithmetic allow to evaluate an expression directly, with sufficient accuracy to determine its sign. In geometric applications this is however not the desired strategy. Instead, the expression is re-evaluated with gradually increasing accuracy, hoping that low accuracy approximations suffice to compute the sign. A common strategy is to ensure quadratic convergence at the root node. It is however by no means clear, that this is an optimal strategy. To avoid unnecessary work, the first step should enforce an error smaller than the one achieved by the floating-point filter.

Beyond the floating-point filter stage, high precision bigfloat arithmetic is the main cost factor. Lets assume, that quadratic convergence translates to doubling the precision in each node in each step. This is approximately true, once all nodes have been approximated reasonably well. Then, each approximation step costs at least twice the runtime of the previous step. The last approximation step, revealing the sign, dominates the running time. If the root node is zero, the separation bound determines the accuracy for the last step and thus the worst case running time. The total runtime is then twice the time it would have taken to approximate the root node directly to the accuracy required for the separation bound to strike.

If a node is not zero, however, the running time depends on how much this node is actually separated from zero. This is what occurs in practice, since expressions over floating-point numbers are almost never zero

Dag Node Management. A dag node may be referenced by several nodes or handles. This allows for example to share common sub-expressions within the same dag or between different dags. Usually it is up to the user to create dags in a way that sub-expressions are shared, see again Figure 1.1 for an example. Sharing common sub-expressions improves the performance because it reduces evaluation cost. To allow for multiple references, nodes are *reference counted*, each node remembers the number of other nodes or handles pointing to it. If the reference count of a node drops to zero, it releases itself. Thus, reference counting enables automatic memory management.

In the best case, the floating-point filter computes the sign for an expression. In a typical geometric algorithm, e.g., using the 2D orientation predicate, this means the dag is created, evaluated and immediately released. Dag nodes require small chunks of memory from the heap. Since the filter evaluation is very fast, the necessary memory management imposes a serious overhead. To reduce the overhead, expression dag based number types use *memory pools* for dag nodes. The idea of a memory pool is to allocate a large chunk of memory once and split it into small pieces, sufficient for dag nodes. Nodes are then allocated from and released to the pool. Since the

pool provides memory blocks of a fixed size only, this is much more efficient than memory management by the operating system.

When the value of a node becomes known exactly, the node may store this value and remove pointers to its operands, thus turning into a leaf node. This is known as dag *pruning*. Pruning reduces the size of the dag and releases memory.

2.1.2. Concrete Number Types. The idea to store expressions as means for adaptive sign computation was already mentioned by Fortune and van Wyk [**29**], but was not pursued further due to high running time cost. We now list some number types and discuss their specifics. Of those, only `Lazy_exact_nt`, `leda::real` and `CORE::Expr` are publicly available.

LEA**:** Mohand Benouamer, Philippe Jaillon, Dominique Michelucci, and Jean-Michel Moreau present the first expression dag based number type, which they call LEA for *lazy exact arithmetic* [**3**]. LEA supports rational arithmetic. The first evaluation stage for sign computation uses a floating-point filter, then the expression structure is inspected to detect equality cases. Finally, and if necessary, the expression in question is evaluated exactly using rational arithmetic. After exact evaluation, the dag is pruned.

`Lazy_exact_nt`**:** The number type `Lazy_exact_nt<NT>` by Sylvain Pion and Andreas Fabri [**83**] is a generic version of LEA. It is distributed as part of the CGAL library. `Lazy_exact_nt` must be instantiated with some exact number type `NT` used for the second evaluation stage. `Lazy_exact_nt` supports field operations and square roots, but at most the operations supported by `NT`. In contrast to LEA, `Lazy_exact_nt` does not exploit expression structure. But if a node is evaluated exactly, its floating-point interval representation is updated for later operations.

Both LEA and `Lazy_exact_nt` have the disadvantage of only two evaluation stages. The accuracy achievable with floating-point filters is limited and large expressions may categorically require higher accuracy for exact sign computation. Other number types go beyond the capabilities of LEA and `Lazy_exact_nt`, both in terms of functionality and adaptivity. They implement subsets of algebraic numbers and increase accuracy iteratively to provide adaptivity beyond the floating-point filter stage. Zero numbers are detected using separation bounds.

Sekigawa**:** Hiroshi Sekigawa [**103**] discusses the implementation of a number type supporting polynomial expressions over algebraic numbers. He uses bigfloat interval arithmetic and increases the accuracy of approximations by the naïve method.

`leda::real`**:** The number type `leda::real` [**10**] is part of the LEDA library [**53**] developed at Max Planck Institute Saarbrücken and commercially available from Algorithmic Solutions. The source code of an early version is publicly available in a technical report [**15**], more recent source code can be found at [**101**].

leda::real is the only number type to support general algebraic expressions with an unrestricted $\diamond$-operator.

The sign computation strategy of leda::real is very close to our description above. High precision approximations in leda::real are represented by a bigfloat interval with midpoint and radius. If the floating-point filter fails, leda::real first ensures that each node stores a valid interval. The interval representation is initialized from the floating-point filter if possible, i.e., if the floating-point filter does not represent the whole range of real numbers. Otherwise it is computed from the operands with low precision bigfloat arithmetic, based on Equation (2.1) and related error bounds. Initializing the interval representation ensures that a finite upper bound for each node is available. If the sign of the root node is still not known, precision driven arithmetic is performed, improving the root node until either the interval approximation does not contain zero or the separation bound is hit. The downwards propagation of precision is based on absolute error, again precisely as described above. When the approximation of a node is improved, its floating-point filter is updated. Furthermore it is checked, whether the node is now known exactly. If this is the case, the dag is pruned. leda::real uses the BFMSS[2] separation bound to detect zero numbers.

To possibly avoid expensive dag creation, a leda::real variable may store a floating-point number explicitly, instead of referring to a dag node. Using interval arithmetic, leda::real checks whether the result of an operation is still a floating-point number. If this is the case, dag creation is avoided.

CORE::Expr 1: The number type CORE::Expr is the product of on-going development efforts by Chee Yap and co-workers at New York University [**25, 118, 79, 48, 54, 18**]. We now discuss some internals of CORE::Expr 1. Version 1.7 is currently part of the CGAL library. The latest version 1.8 fixes some bugs, but is otherwise identical to 1.7. The development of CORE::Expr 1 was halted in favor of version 2, which we discuss further below. CORE::Expr 1 supports field operations, square roots and the $\diamond$-operator restricted to rational coefficients.

There are some differences between the sign computation strategy we describe above and the implementation of CORE::Expr. To represent approximations, CORE::Expr 1 uses a special type of bigfloat with associated error, which actually is an interval representation. Let $B = 2^c$ for some positive $c \in \mathbb{N}$. A triple $\langle m, r, e\rangle$ with $m \in \mathbb{Z}$, $r \in \mathbb{N}$ and $e \in \mathbb{Z}$ represents the interval

$$[(m-r)B^e, (m+r)B^e].$$

This is an interval representation with bigfloat midpoint and radius, such that the mantissae m and r share a common exponent e. Originally, CORE::Expr 1 is based on a concept of mixed absolute and relative accuracy. A number x is approximated to accuracy $[a, r]$, written $\hat{x} \cong x[a, r]$, if

$$|\hat{x} - x| \leq \max\{2^{-a}, 2^{-r}|x|\}.$$

Only the less sharp of both bounds has to be satisfied. It is possible to express a purely relative (resp. absolute) error bound by setting $a = \infty$ (resp. $r = \infty$). Originally, mixed accuracy was propagated in precision driven arithmetic, but newer versions of `CORE::Expr` 1 use absolute accuracy only.

When recomputing the approximation of a node, `CORE::Expr` 1 performs ring operations exactly. This can be much more expensive than approximate computation, because the cost depends on the output precision. The exact result has only slightly better accuracy than an appropriate approximate one, because the relevant error terms simply add up, see Equation (2.1). Furthermore, `CORE::Expr` does not set the error bound for a node, but simply computes with its bigfloat/interval representation. This may result in a slight improvement in known accuracy.

The sign computation framework on the other hand tries to minimize the usage of precision driven arithmetic and hence expensive bigfloat arithmetic. Beyond the floating-point filter there is an initialization routine, and precision driven arithmetic which call each other recursively. The initialization step is called first and tries to compute for each node z the sign as well as a lower bound $\mathrm{lMSB}(z)$ and an upper bound $\mathrm{uMSB}(z)$ on the most significant bit of z. More precisely

$$\mathrm{lMSB}(z) \leq \lfloor \log_2 |z| \rfloor \leq \mathrm{uMSB}(z)$$

must hold. This is effectively a one bit interval approximation not containing zero and provides the upper and lower bound needed for precision driven arithmetic. In most cases the interval and sign can be determined from the operand nodes recursively, for example $\mathrm{sign}(xy) = \mathrm{sign}(x)\,\mathrm{sign}(y)$. The major exception is an addition or subtraction node when no zero-free one bit interval approximation can be determined from the operands. In this case precision driven arithmetic is used to compute a sufficiently accurate approximation, however only after performing the initialization step on the operands of this node. Thus, precision driven arithmetic might start much lower than at the node whose sign is actually requested and will need less accuracy. To detect zero nodes, `CORE::Expr` 1 uses the maximum of the DM bound, the LY bound and the BFMSS[2, 5] bound.

`CORE::Expr` 2: In 2006, major redesign and improvement efforts on `CORE::Expr` were made [**24, 119**], resulting in a new version. The current version number is 2.1.1. `CORE::Expr` 2 supports field operations, radicals, and the $\diamond$-operator restricted to rational coefficients.

`CORE::Expr` 2 uses generic programming techniques, to make key parts of an implementation exchangeable. These parts are the floating-point filter, the separation bound, and an arithmetic kernel, providing approximate real number computation. For the floating-point filter parameter, one implementation based on the work by Burnikel et al. [**13**] is available. As separation bound, the BFMSS bound and the BFMSS[2, 5] bound are implemented and can be used. The arithmetic kernel effectively must provide arbitrary precision interval arithmetic.

One implementation is available, based on MPFR [**71**] bigfloat numbers and representing intervals by endpoints.

The use of precision driven arithmetic has been improved. Ring operations are now performed approximately, but error bounds are still computed, not set. The sign computation scheme is even lazier than in `CORE::Expr 1`. The initialization routine, computing sign(z), lMSB(z), and uMSB(z) is split into three individual parts, which may call each other recursively. This may avoid some unnecessary computations. Consider for example the lower bound lMSB(z). In `CORE::Expr 2` it is only computed when it is actually needed, i.e., requested from a node. The node has several means to compute the information, which are checked in order of expensiveness. It first checks the floating-point filter and then whether lMSB(z) is already cached. Then, the node checks whether a bigfloat approximation is available and can provide a lower bound. Only afterwards, the node tries to compute lMSB(z), by requesting the necessary information from its operands. As measure of last resort, the node is approximated using precision driven arithmetic until a lower bound is known. Data is only cached when it has been computed by expensive means, to avoid the overhead associated with caching.

In contrast to all other number types mentioned so far, `CORE::Expr 2` allows expressions involving transcendental constants and functions, e.g., π, exp and log. Precision driven arithmetic is extended to these functions and `CORE::Expr 2` allows to approximate transcendental expressions to any a priori absolute accuracy. Since the problem of detecting zero numbers is unsolved beyond algebraic numbers, `CORE::Expr 2` uses a cutoff bound for transcendental expressions, i.e., any number below some small ε is considered to be zero. `CORE::Expr 2` notifies the user if the correctness of computational results depend on the cutoff bound.

2.1.3. Possible Improvements. What can be improved, regarding the current state of the art implementations `leda::real` and `CORE::Expr`? Both number types have different strengths and weaknesses.

Not much can be done on the level of floating-point filters. Beyond that, bigfloat arithmetic is the main cost factor. Thus, an implementation should avoid bigfloat arithmetic as much as possible or reduce the precision of operations. The approach of `CORE::Expr`, to shift the starting point for precision driven arithmetic downwards in the dag is therefore reasonable. In applications however, the root node of a dag is usually an addition or subtraction node, see for example the predicates discussed in Section 1.1. If the floating-point filter fails on this node, it is very likely that precision driven arithmetic starts right there.

In `leda::real`, bigfloat arithmetic with low precision is used to compute error bounds, both a posteriori in an initialization step, and a priori for precision driven arithmetic. Compared to the benefits, the overhead of bigfloat arithmetic is especially large for small precision. Bigfloat arithmetic is used in this step basically because it

has a much large exponent range and will not overflow. Here it might be better to compute logarithmically, i.e., with one bit only and hence without bigfloat arithmetic, as is done for precision propagation in `CORE::Expr`.

Both versions of `CORE::Expr` compute an a posteriori error bound in precision driven arithmetic. This is unnecessary double work since the bound can only be slightly better than the a priori bound. Much worse is however that `CORE::Expr 2` represents intervals by endpoints! Assume we have approximated a node very accurately, e.g., with a few thousand correct bits. In the midpoint and error representation, the error term may have precision as small as one bit. To improve the node, one high precision operation for the midpoint and several small precision operations for the error bound are necessary. In the endpoint representation, however, both endpoints have high precision and nearly all bits of the endpoints are equal! Improving the approximation of the node requires two high precision bigfloat operations. All other things being equal, for high precision intervals, the endpoint representation is slower by a factor of two, compared to the midpoint and radius representation. Hence, clearly the latter should be used.

`leda::real` is based on the LEDA number type `leda::bigfloat`, but more efficient choices are available, especially the MPFR number type. Since different bigfloat number types have similar interfaces, the bigfloat arithmetic should be easily exchangeable in an expression dag based number type. Apart from allowing to choose an efficient one, this also allows a user to provide her own bigfloat arithmetic, i.e., one that fits into her application environment.

In `leda::real`, there is this interesting approach to defer dag creation by storing an exact floating-point number instead. Since the original problem is the inexactness of floating-point arithmetic, this imposes rather strong conditions on the class of expressions where this strategy is useful. The value of sub-expressions must be exactly computable with floating-point arithmetic, but for the whole expression, floating-point arithmetic should compute the wrong sign. Remember that a polynomial expression of degree d over integers with b bits precision can be evaluated exactly using $d(b + O(1))$ bits of precision. In general, this amount of precision is also necessary. Hence, for the first condition to hold, input numbers must have low precision, i.e., less than floating-point numbers can handle. It would be interesting to extend this approach and allow exact representations with more precision than a single floating-point number, to make the approach useful for input numbers with larger precision, e.g., floating-point numbers. But there is a caveat. The high cost and non-adaptiveness of exact representations and arithmetic are the reason for expression dag based number types in the first place. Hence this can only work if very fast exact arithmetic, tuned for low precision is available. Indeed, this arithmetic must be faster than, lets say dag creation and computation of the floating-point filter. We attempt to provide such an arithmetic based on fast and exact floating-point algorithms in this thesis. We will discuss the basics for the development of such algorithms in the next section.

2.2. Exact Floating-Point Computations

Every computer scientist knows or should know [**38**] that floating-point arithmetic is slightly imprecise. For many this may translate into a notion of fuzziness, floating-point computations are perceived as slightly unpredictable or even non-deterministic. The task of this section is to convince the reader of the contrary. Floating-point arithmetic following the IEEE 754 standard [**45**] or its recent revision the IEEE 754-2008 standard [**46**] is very well defined and has a precise notion of how numbers are represented and arithmetic operations are carried out. The standard allows to prove rigorous mathematical theorems about floating-point arithmetic and based on these, design very precise and even exact algorithms.

Equally important, floating-point arithmetic following the IEEE 754 standard is ubiquitous in hardware. This allows for efficient implementation nearly everywhere. In recent years, the degree of compatibility has improved. For example graphic cards, which are nowadays used for parallel general purpose computations, not only support floating-point arithmetic, but continuously implement more IEEE 754 features, e.g., denormalized numbers, directed rounding, and exception handling. Another example for improved compatibility is the $x86$ class of CPUs. These CPUs feature 80 bit extended registers, which compute with larger precision and larger exponent range than the default 64 bit floating-point numbers. While this increases the accuracy of results when applying floating-point arithmetic naïvely, it does prevent techniques like the ones we will discuss in this section. There are some means to improve the situation, but none of the available techniques truly gives the results required by the standard. With the advent of SSE floating-point instructions the situation has changed, as these behave exactly as the standard requires.

While compatible hardware is available, one nevertheless has to be careful that the programming environment does not mess up carefully designed algorithms. For $x86$ CPUs we have to make sure for example, that only the SSE floating-point instructions are used. Furthermore, nearly any class of CPUs has some kind of high speed floating-point mode, which sacrifices standard conformity. These modes must be avoided too. We need to make sure that compiler optimization, in particular constant folding and expression rearrangement, is only done using the true semantics of floating-point arithmetic at runtime and not that of real number arithmetic. Our experience with the gnu, SUN, and Intel C++ compiler is, that full compliance with the IEEE 754 standard is not the default behavior, but can be enforced easily with compiler options.

All features of floating-point arithmetic that we rely on in the following are already present in the older IEEE 754 standard. We will however use the updated terminology of the IEEE 754-2008 standard.

2.2.1. Notation and Basic Facts. Recall, that with $\mathscr{F}$ we denote the set of binary floating-point numbers, i.e., zero plus all real numbers $f \neq 0$ that can be decomposed into a sign $s \in \{-1, 1\}$, a mantissa $m \in \mathbb{N}$, and an exponent $e \in Z$, such

that

$$f = s \times m \times 2^e.$$

This is equivalent to the set of rational numbers, whose denominator is a power of two, i.e.,

$$\mathscr{F} = \bigcup_{\sigma \in 2^{\mathbb{Z}}} \sigma\mathbb{Z}.$$

Our goal is now to characterize the subset $\mathbb{F}$ of $\mathscr{F}$ that corresponds to the set of floating-point numbers defined by the IEEE 754 standard. We parameterize $\mathbb{F}$ with three quantities. The first is the precision or number of bits in the mantissa, p, though we use the equivalent quantity $\varepsilon_{\mathtt{m}} = 2^{-p}$. The other two parameters are the largest and the smallest representable power of two, τ and η. Then, some number from $\mathscr{F}$ is in $\mathbb{F}$, if its mantissa uses at most p bits and contains no bit larger than τ and no bit smaller than η. We formalize this in the following.

Definition 2.1. *The function* msb : $\mathbb{R} \to 2^{\mathbb{Z}}$ *with*

$$\operatorname{msb}(x) = \begin{cases} 2^{\lfloor \log_2 |x| \rfloor} & \text{for } x \neq 0 \\ 0 & \text{for } x = 0 \end{cases}$$

gives the most significant bit *for any real number x. The function* lsb : $\mathscr{F} \to 2^{\mathbb{Z}}$ *with*

$$\operatorname{lsb}(f) = \begin{cases} \max\{\sigma \mid f \in \sigma\mathbb{Z},\ \sigma \in 2^{\mathbb{Z}}\} & \text{for } f \neq 0 \\ \infty & \text{for } f = 0 \end{cases}$$

gives the least significant bit *of any floating-point number f.*

Siegfried Rump, Takeshi Ogita, and Shin'ichi Oishi [**94**] introduce the most significant bit under the name of ufp(x) for *unit in the first place* as a tool for the analysis of floating-point algorithms. The name is chosen complimentary to ulp(x) or *unit in the last place*, which is a classic quantity in the error analysis of floating-point algorithms. For a real number x, ulp(x) is the distance between the two floating-point numbers closest to x. The quantities ulp(x) and lsb(x) are different and we do not use ulp(x) in our analysis. Hence we settled for the names most significant bit and least significant bit. Using them, we can now define the set $\mathbb{F}$.

Definition 2.2. *Let $\eta, \varepsilon_{\mathtt{m}}, \tau$ be powers of two such that $0 < \eta < \varepsilon_{\mathtt{m}} < 1 < \tau$. Then let*

$$\begin{aligned} \tilde{\mathbb{F}} &= \{\, x \in \mathbb{R} \mid \operatorname{msb}(x) \leq \tfrac{1}{2}\varepsilon_{\mathtt{m}}^{-1} \operatorname{lsb}(x) \,\}, \\ \mathbb{F} &= \{\, f \in \tilde{\mathbb{F}} \mid \eta \leq \operatorname{lsb}(f),\ \operatorname{msb}(f) \leq \tau \,\}, \\ \overline{\mathbb{F}} &= \mathbb{F} \cup \{\pm\infty\}. \end{aligned}$$

We call $\mathbb{F}$ the set of floating-point numbers.

Although $\mathbb{F}$ depends on several parameters, we omit them because we use only one floating-point format at a time. The condition $0 < \eta < \varepsilon_{\mathtt{m}} < 1 < \tau$ is imposed to consider only reasonable sets of floating-point numbers. All binary floating-point

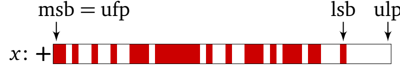


FIGURE 2.3. Visualization of floating-point numbers, location of $\mathrm{msb}(x)$, $\mathrm{lsb}(x)$, $\mathrm{ufp}(x)$, $\mathrm{ulp}(x)$.

formats defined by the IEEE 754 standard can be obtained from Definition 2.2 for certain values of η, $\varepsilon_{\mathtt{m}}$, and τ. Unless noted otherwise, our algorithms and analyses hold for any of them. Of special interest to us is the `binary64` format with $\eta = 2^{-1074}$, $\varepsilon_{\mathtt{m}} = 2^{-53}$ and $\tau = 2^{1023}$ that is in widespread use and that we also use in our implementations.

Next to η, $\varepsilon_{\mathtt{m}}$, and τ, there are other constants of interest. The number $2\tau(1-\varepsilon_{\mathtt{m}})$ is the largest number in $\mathbb{F}$. The number $\frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta$ is the smallest positive so called *normalized* number. A number $f \in \mathbb{F}$, with $\mathrm{msb}(f) < \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta$, is called *denormalized*. For a denormalized number, less than p bits beyond the most significant bit are available, due to the limitation by η. The name denormalized comes from their different hardware representation. Denormalized numbers also need special care in most error analyses. But their inclusion into the standard makes the definition above possible and allows for several nice results, which would be false otherwise. Floating-point arithmetic also encompasses special values $+\infty, -\infty$ and `nan`. They occur as the result of arithmetic operations in exceptional situations. We do not include them into $\mathbb{F}$ and never allow them as input to our algorithms. By definition $\mathbb{F} \subset \mathbb{R}$. We use the set $\tilde{\mathbb{F}}$ if we would like to ignore the limitations imposed by η and τ.

We frequently visualize floating-point numbers as in Figure 2.3. For these examples, we use the `binary64` format. For a floating-point number, we draw the mantissa, where zero bits are white and non-zero bits are colored. The exponent is only represented in relation to other numbers: if more than one number is in the figure, bits of equal magnitude are vertically aligned, see for example Figure 2.4.

Before turning to arithmetic over $\mathbb{F}$, we collect some properties of the most significant bit. The msb function is monotone for positive numbers and it is homogeneous for powers of two. For $x, y \in \mathbb{R}$ and $\sigma = 2^k$, $k \in \mathbb{Z}$ we have

$$|x| < |y| \;\Rightarrow\; \mathrm{msb}(x) \le \mathrm{msb}(y), \qquad\qquad \mathrm{msb}(\sigma x) = \sigma\,\mathrm{msb}(x).$$

Furthermore, we can bound any number from below and above by means of its most significant bit. For $x \in \mathbb{R}$,

$$\mathrm{msb}(x) \;\le\; |x| \;<\; 2\,\mathrm{msb}(x).$$

For $f \in \mathbb{F}$ we can improve the upper bound slightly to

$$\mathrm{msb}(f) \;\le\; |f| \;\le\; 2\,\mathrm{msb}(f)(1-\varepsilon_{\mathtt{m}}) \;<\; 2\,\mathrm{msb}(f).$$

We will use these properties frequently in our proofs.

Rounding and Arithmetic. We next discuss basic arithmetic operations over $\mathbb{F}$. Since the set $\mathbb{F}$ is finite, rounding is inevitable. Field operations $(+,-,\times,/)$ and the square root $(\sqrt[2]{\ })$ are performed as if first the exact result is computed and then rounded to a floating-point number according to some *rounding mode*. Essential to rounding are the notion of *predecessor* and *successor*.

Definition 2.3. *Let* $x \in \mathbb{R}$*, then* $\operatorname{pred}, \operatorname{succ} : \mathbb{R} \to \overline{\mathbb{F}}$ *with*

$$\operatorname{pred}(x) = \max\{f \in \overline{\mathbb{F}} \mid f < x\}$$
$$\operatorname{succ}(x) = \min\{f \in \overline{\mathbb{F}} \mid x < f\}$$

give the predecessor *and* successor *of* x.

A rounding mode is *faithful* if it rounds $x \in \mathbb{F}$ to itself and $x \in \mathbb{R}\backslash\mathbb{F}$ to either $\operatorname{pred}(x)$ or $\operatorname{succ}(x)$. All five IEEE 754 rounding modes are faithful. There are three directed rounding modes, *roundTowardPositive*, *roundTowardNegative*, and *roundTowardZero*, which round to the next floating-point number in direction towards $+\infty$, $-\infty$, and zero, respectively. The rounding modes *roundTiesToAway* and *roundTiesToEven* round to the closest floating-point number. If the number to be rounded is larger in absolute value than $2\tau(1-\varepsilon_{\mathtt{m}})$, the rounding is first performed with respect to $\tilde{\mathbb{F}}$. If the closest number in $\tilde{\mathbb{F}}$ is not an element of $\mathbb{F}$, then the result is $+\infty$ or $-\infty$, respectively. In case there is a tie, i.e., the number to be rounded is exactly between two floating-point numbers, roundTiesToAway rounds to the floating-point number farther away from zero. The rounding mode roundTiesToEven resolves ties by rounding to the floating-point number whose last bit is zero.

There are two exceptional situations when rounding. We say *underflow* occurs, when a real number x with $\operatorname{msb}(x) \leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta$ is rounded to different numbers in $\tilde{\mathbb{F}}$ and $\mathbb{F}$. We say *overflow* occurs, when a real number x with $\operatorname{msb}(x) > \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta$ is rounded to different numbers in $\tilde{\mathbb{F}}$ and $\mathbb{F}$. Both overflow and underflow signal a loss of accuracy. When rounding to a nearest floating-point number, overflow corresponds to rounding to $\pm\infty$ but for directed rounding this is not the case. The IEEE 754 standard provides means for the user to check or be notified if overflow or underflow occurs.

Another exceptional situation occurs if the mathematical result of an operation is not defined, for example in case of a division by zero. Then, the floating-point arithmetic returns one of the special values $\pm\infty$ or nan. The IEEE 754 standard attempts to define reasonable or expected results for these cases, even when $\pm\infty$ occur as operands. For example $-4/0 = -\infty$ and $-\infty \times -\infty = +\infty$. If no reasonable result can be given, the return value is nan. In particular, any operation involving a nan returns nan. Regarding the comparison of numbers, $\pm\infty$ behave as expected and any comparison involving nan evaluates to false.

In this thesis we distinguish between faithful rounding and rounding to nearest only. With $\tilde{\mathbf{fl}} : \mathbb{R} \to \overline{\mathbb{F}}$ we denote an arbitrary but fixed faithful rounding. All IEEE 754 rounding modes are a model for $\tilde{\mathbf{fl}}$. With $\mathbf{fl} : \mathbb{R} \to \overline{\mathbb{F}}$ we denote a function that rounds $x \in \mathbb{R}$ to a nearest floating-point number and breaks ties in a symmetric way,

i.e., such that $\mathbf{fl}(-x) = -\mathbf{fl}(x)$. The IEEE 754 rounding modes roundTiesToAway and roundTiesToEven are a model for **fl**. We denote floating-point operations with $\oplus, \ominus, \otimes, \oslash$, i.e., for $a, b \in \mathbb{F}$, $\circ \in \{+, -, \times, /\}$ and $a \circ b \in \mathbb{R}$ we have $a \circledcirc b = \tilde{\mathbf{fl}}(a \circ b)$ or $a \circledcirc b = \mathbf{fl}(a \circ b)$, depending on the rounding mode considered.

Properties of Faithful Rounding. Faithful rounding has the important property of being *monotone*. For $x, y \in \mathbb{R}$, $\tilde{\mathbf{fl}}$ satisfies

$$x \le y \;\Rightarrow\; \tilde{\mathbf{fl}}(x) \le \tilde{\mathbf{fl}}(y), \qquad \tilde{\mathbf{fl}}(x) < \tilde{\mathbf{fl}}(y) \;\Rightarrow\; x < y. \tag{2.4}$$

In other words, we never skip over a floating-point number in the process of rounding. One of the most basic tools for the analysis of floating-point operations is an estimate of the error of floating-point operations. For the following see for example [**44, 72, 80**]. Let $x \in \mathbb{R}$ and $f \in \mathbb{F}$ with $\tilde{\mathbf{fl}}(x) = f$, then

$$\begin{array}{llll} f \;=\; x(1+\delta_1)+\mu_1 \;=\; x/(1+\delta_2)+\mu_2 & & & \text{with} \\ |\delta_i| < 2\varepsilon_{\mathtt{m}}, & |\mu_i| < \eta, & \mu_i\delta_i = 0 & \text{for } i = 1, 2. \end{array} \tag{2.5}$$

Floating-point addition and subtraction have better properties than the other operations and they are also the operations we investigate the most. For a fixed rounding mode, the result of rounding $x \in \mathbb{R}$ to f depends on x only, hence

$$a \ominus -b \;=\; a \oplus b \;=\; b \oplus a \;=\; b \ominus -a.$$

Therefore, all properties of $\oplus$ also hold for $\ominus$ as long as no condition on the signs of the operands are imposed

To keep track of the lower bits arising in computations with floating-point numbers, we view them as element of a ring $\sigma\mathbb{Z}$, where σ is a power of two. A statement such as $f \in \sigma\mathbb{Z}$ tells us that $f = 0$ or $\sigma \le \mathrm{lsb}(f)$. Some useful properties are:

$$\mathbb{F} \subset \eta\mathbb{Z} \tag{2.6}$$

$$\sigma_1 \ge \sigma_2 \quad\Rightarrow\quad \sigma_1\mathbb{Z} \subseteq \sigma_2\mathbb{Z} \tag{2.7}$$

$$a, b \in \sigma\mathbb{Z} \quad\Rightarrow\quad a + b \in \sigma\mathbb{Z} \tag{2.8}$$

$$a, b \in \sigma\mathbb{Z} \cap \mathbb{F},\; u \oplus v \in \mathbb{F} \quad\Rightarrow\quad u \oplus v \in \sigma\mathbb{Z} \tag{2.9}$$

The first three are simply ring properties. Equation (2.9) follows from them, since in the process of rounding $u + v$ to $u \oplus v$, trailing bits are removed, i.e., either $u + v = 0$ or $\eta \le \sigma \le \mathrm{lsb}(u + v) \le \mathrm{lsb}(u \oplus v)$. To demonstrate the usefulness of the msb function in combination with Equation (2.9), we now show the well known result that addition and subtraction are exact in case the result falls in the range of denormalized numbers.

Lemma 2.4. *Let $a, b \in \mathbb{F}$ with $\mathrm{msb}(a + b) \le \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta$. Then $a + b \in \mathbb{F}$ and hence $a \oplus b = a + b$ for faithful rounding.*

Proof. We know $a, b \in \mathbb{F}$ and hence that $a + b \in \eta\mathbb{Z}$. If $a + b = 0$ the claim holds. Otherwise we have $\eta \le \mathrm{lsb}(a + b)$ and

$$\mathrm{msb}(a + b) \;\le\; \tfrac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta \;\le\; \tfrac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\,\mathrm{lsb}(a + b).$$

Furthermore $\operatorname{msb}(a+b) < \tau$ and hence $a+b \in \mathbb{F}$. □

Lemma 2.4 is one major reason for the inclusion of denormalized numbers into the IEEE 754 standard. It has some nice consequences. It implies for example that no non-zero number is ever rounded to zero in an addition or subtraction. Since any faithful rounding is monotone, it follows for $a, b \in \mathbb{F}$ that

$$\operatorname{sign}(a \oplus b) = \operatorname{sign}(a+b). \tag{2.10}$$

Furthermore, no underflow occurs for addition or subtraction and we get an improved version of Equation (2.5). If $a, b \in \mathbb{F}$ and no overflow occurs in computing $a \oplus b$, then

$$a \oplus b = (a+b)(1+\delta_1) = (a+b)/(1+\delta_2) \quad \text{with} \quad |\delta_i| < 2\varepsilon_{\mathtt{m}} \quad \text{for } i = 1,2. \tag{2.11}$$

Another important exactness result is due to Pat Sterbenz [**106**]. It gives a sufficient condition when the difference of two floating-point numbers with the same sign is free from rounding error.

Lemma 2.5 (Sterbenz).
Let $a, b \in \mathbb{F}$ with $a, b \geq 0$ and $\frac{1}{2}a \leq b \leq 2a$. Then $a-b \in \mathbb{F}$ and hence $a \ominus b = a-b$ for faithful rounding.

Note that Sterbenz Lemma holds unconditionally. It is not affected by underflow or overflow.

Properties of Rounding to Nearest. Rounding to nearest is a special case of faithful rounding, so all properties are preserved. We do however get smaller errors and error estimates. A classic error estimate is the following [**44, 72, 80**]. Let $x \in \mathbb{R}$ and $f \in \mathbb{F}$ with $\mathbf{fl}(x) = f$, then

$$f = x(1+\delta_1)+\mu_1 = x/(1+\delta_2)+\mu_2 \quad \text{with} \quad |\delta_i| \leq \varepsilon_{\mathtt{m}}, \quad |\mu_i| \leq \tfrac{1}{2}\eta, \quad \mu_i\delta_i = 0 \quad \text{for } i = 1,2. \tag{2.12}$$

In comparison to Equation (2.5) the error bounds are reduced by a factor of about two. The following theorem by Rump et al. [**94**] gives yet a better estimate.

Theorem 2.6. *Let $x \in \mathbb{R}$ and $f \in \mathbb{F}$ with $\mathbf{fl}(x) = f$.*
If $f \geq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta$, then

$$f = x+\delta \quad \textit{with} \quad |\delta| \leq \varepsilon_{\mathtt{m}} \operatorname{msb}(x) \leq \varepsilon_{\mathtt{m}} \operatorname{msb}(f). \tag{2.13}$$

If $f \leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta$, then

$$f = x+\mu \quad \textit{with} \quad |\mu| \leq \tfrac{1}{2}\eta. \tag{2.14}$$

The major difference between Equation (2.12) and Theorem 2.6 is the use of the most significant bit in the error bound, which can be an improvement by nearly a factor of two. Note that Equation (2.14) is only relevant for operations where underflow might have occurred and that it is a much worse bound than

Equation (2.13) in this case. In fact, the reason for having overflow and underflow exceptions is to identify the cases where Equation (2.13) may not hold.

Again, we can give a better error estimate for floating-point addition and subtraction since underflow never occurs. Furthermore, we can exclude the possibility of overflow by requiring that a result is a floating-point number. Let $a, b, a \oplus b \in \mathbb{F}$, where the addition is performed in rounding to nearest. Then

$$a \oplus b = a + b + \delta \qquad \text{with} \qquad |\delta| \le \varepsilon_{\mathtt{m}} \operatorname{msb}(a+b) \le \varepsilon_{\mathtt{m}} \operatorname{msb}(a \oplus b). \tag{2.15}$$

Both $\mathbb{F}$ and **fl** are symmetric with respect to zero, i.e., $\mathbb{F} = -\mathbb{F}$ and $\mathbf{fl}(-x) = -\mathbf{fl}(x)$ for $x \in \mathbb{R}$. Hence, for rounding to nearest we have

$$a \oplus b = -(-a \ominus b)$$

and similar equalities. For this reason we can often retreat to a special case, symmetric to the remaining cases, in proofs concerning addition and subtraction.

2.2.2. Error-Free Transformations. The name *error-free transformation* refers to algorithms that efficiently transform expressions of floating-point numbers into mathematically equivalent expressions [**77**]. They form the basic toolkit for exact floating-point algorithms or algorithms with increased precision. All error-free transformations presented here require rounding to nearest.

ADDITION AND SUBTRACTION. There are two algorithms TWOSUM and FASTTWOSUM, which transform a sum of two floating-point numbers into a new sum. This is done by recovering the exact error term arising in a floating-point addition.

Algorithm 2.7 (FASTTWOSUM).
Let $a, b \in \mathbb{F}$, with $|a| \ge |b|$ or $a = 0$. Compute $(x, y) \leftarrow$ FASTTWOSUM(a, b). If $x \in \mathbb{F}$, then $x = a \oplus b$ and $a + b = x + y$.

Algorithm 2.8 (TWOSUM).
Let $a, b \in \mathbb{F}$ and compute $(x, y) \leftarrow$ TWOSUM(a, b). If $x \in \mathbb{F}$, then $x = a \oplus b$ and $a + b = x + y$.

1: **procedure** FASTTWOSUM(a, b)
2: $x \leftarrow a \oplus b$
3: $b_v \leftarrow x \ominus a$
4: $y \leftarrow b \ominus b_v$
5: **return** (x, y)

1: **procedure** TWOSUM(a, b)
2: $x \leftarrow a \oplus b$
3: $b_v \leftarrow x \ominus a$
4: $b_r \leftarrow b \ominus b_v$
5: $a_v \leftarrow x \ominus b_v$
6: $a_r \leftarrow a \ominus a_v$
7: $y \leftarrow a_r \oplus b_r$
8: **return** (x, y)

Both algorithms compute the same result, though FASTTWOSUM requires the input summands to be ordered. Often it is known a priori, which summand will be the larger one. If this is not known, one may use TWOSUM, or one may swap the summands if necessary. On contemporary CPUs, using TWOSUM is then usually faster,

since checking and swapping summands involves a branch, which may significantly slow down the computation.

Note that the computed error term y equals the error term δ in Equation (2.15) and we have $|y| \leq \varepsilon_{\mathtt{m}} \operatorname{msb}(x)$. Quite remarkably, we can compute the error term using only two or five additional, ordinary floating-point operations. The only exception is the case, when overflow occurs in the computation of x. TwoSum and FastTwoSum are safe from overflow if $|x| \leq 2\tau(1-\varepsilon_{\mathtt{m}})$ or equivalently $x \in \mathbb{F}$.

A proof for the exactness of the error term in FastTwoSum was first given by Theodorus Dekker [**20**], though the algorithm was in use before, see e.g., [**47**]. Using Sterbenz Lemma, one can show that either $x = a+b$ or $b_v = x - a$. In the first case $b_v = b$ and $y = 0$. In the second case $y = b \ominus b_v = \mathbf{fl}(b-x+a)$. The proof finishes by observing that $a+b-(a\oplus b) \in \mathbb{F}$ and hence $y = b-x+a$. TwoSum basically applies FastTwoSum twice, reversing the roles of a and b. A proof for the exactness of the error term computed by TwoSum is due to Donald Knuth [**52**]. Both FastTwoSum and TwoSum are in danger of being messed up by the programming environment. For example in FastTwoSum one might conclude that

$$y = b - b_v = b - (x-a) = b - (a+b-a) = 0.$$

using the semantics of real arithmetic.

Multiplication. Similar to TwoSum and FastTwoSum, TwoProduct transforms the product of two floating-point numbers into a sum of two floating-point numbers. A basic sub-step in the TwoSum algorithm given below, is to split the operands into two numbers which can be multiplied exactly.

Algorithm 2.9 (split).
Let $a \in \mathbb{F}$ and compute $(a_{hi}, a_{lo}) \leftarrow \text{split}(a)$. Assume that no overflow occurs. Then $a = a_{hi} + a_{lo}$, $|a_{hi}| \geq |a_{lo}|$ and both can be represented using a mantissa of at most $\lfloor p/2 \rfloor$ bits.

Algorithm 2.10 (TwoProduct).
Let $a, b \in \mathbb{F}$ and compute $(x, y) \leftarrow \text{TwoProduct}(a, b)$. If neither overflow nor underflow occurs, then $x = a \otimes b$ and $ab = x + y$.

```
1: procedure split(a)
2:     c ← (2^⌈p/2⌉ + 1) ⊗ a
3:     a_big ← c ⊖ a
4:     a_hi ← c ⊖ a_big
5:     a_lo ← a ⊖ a_hi
6:     return (a_hi, a_lo)
```

```
1: procedure TwoProduct(a, b)
2:     x ← a ⊗ b
3:     (a_hi, a_lo) ← split(a)
4:     (b_hi, b_lo) ← split(b)
5:     e_1 ← x ⊖ (a_hi ⊗ b_hi)
6:     e_2 ← e_1 ⊖ (a_lo ⊗ b_hi)
7:     e_3 ← e_2 ⊖ (a_hi ⊗ b_lo)
8:     y ← (a_lo ⊗ b_lo) ⊖ e_3
9:     return (x, y)
```

The split subroutine is due to Dekker [**20**], who attributes TwoProduct to G. W. Veltkamp. In case p is odd, it may seem impossible to represent a p bit number as

the sum of two $\lfloor p/2 \rfloor$ bit numbers. The missing bit is however hidden in the sign of a_{lo}. Therefore, the products $a_{\mathrm{hi}} \otimes b_{\mathrm{hi}}$ etc. in TwoProduct are computed without rounding error. The subsequent subtractions in TwoProduct are exact, too.

TwoProduct needs 16 additional operations to compute the error term y. Again, y equals the error term δ in Equation (2.13) and we have $|y| \leq \varepsilon_{\mathrm{m}} \operatorname{msb}(x)$. Both, overflow and underflow may affect TwoProduct. The splitting step involves the constant $2^{\lceil p/2 \rceil}+1$, which is multiplied with both a and b. If overflow occurs anywhere, it occurs in the computation of c or x, too. Hence, TwoProduct is safe from overflow, if

$$\max\{|a|, 2^{\lceil p/2 \rceil} + 1\} \otimes \max\{|b|, 2^{\lceil p/2 \rceil} + 1\} \leq 2\tau(1 - \varepsilon_{\mathrm{m}}).$$

No underflow can occur in split, since $2^{\lceil p/2 \rceil}+1 \in \mathbb{Z}$ implies $\operatorname{lsb}((2^{\lceil p/2 \rceil}+1)a) \geq \operatorname{lsb}(a)$. But it may occur in any multiplication in TwoProduct itself. The exact product ab may have up to $2p$ bits. Hence, TwoProduct is safe from underflow, if

$$ab = 0 \qquad \text{or} \qquad |a||b| > \tfrac{1}{2}\varepsilon_{\mathrm{m}}^{-2}\eta.$$

The newer IEEE 754-2008 standard mandates the availability of a fused-multiply-add instruction **fma**(a, b, c), rounding $ab + c$ in one step to a floating-point number. Using **fma**, TwoProduct can be implemented as

```
1: procedure TwoProduct(a, b)
2:     x ← a ⊗ b
3:     y ← fma(a, b, −x)
4:     return (x, y)
```

Although a fused-multiply-add operation might be more costly than a standard binary floating-point operation, this TwoProduct implementation can be expected to be more efficient. Furthermore, it avoids the problem of overflow in split and hence increases the range of validity for TwoProduct. A TwoProduct based on **fma** is safe from overflow, if $|a| \otimes |b| \leq 2\tau(1 - \varepsilon_{\mathrm{m}})$.

TwoSum, FastTwoSum and TwoProduct have in common, that they compute their results with just a few ordinary floating-point operations. They do not involve any branches. Thus, they can easily be optimized by a compiler, e.g., using instruction level parallelism, and lead to efficient code. Proofs for the exactness of the error terms in TwoSum, FastTwoSum and TwoProduct can also be found in [**104**].

For each of the algorithms we gave sufficient and easily checkable conditions when they are safe from corruption by overflow and underflow. Why do we care so much for overflow and underflow? After all, they occur only for very small or very large input data and in many cases the input data can be scaled to avoid these problems. We do intend to integrate error-free transformations into an expression dag based number type. One of the main advantages of such a number type is user-friendliness. The user should get correct signs and approximations without caring about the internals. To achieve this goal, we need means to handle overflow

an underflow. A first step in this direction is to understand precisely when they may corrupt our data.

Auxiliary Functions. In some cases, we would like to actually compute msb(f), pred(f), or succ(f) for some $f \in \mathbb{F}$. There are several means to do this. For example in C/C++, the library function

```
double frexp(double f,int *e)
```

decomposes the floating-point number f into exponent e and mantissa m, allowing to compute msb(f). The library function

```
double nextafter( double f, double t )
```

returns the floating-point number next to f in direction towards t and hence allows to compute pred(f) and succ(f). An alternative is to interpret the memory representation of a floating-point number as unsigned integer and manipulate the bits directly. The IEEE 754 representation is such that for $f \geq 0$, the unsigned integer representation of the successor of f is simply the successor of the unsigned integer representation of f. The practicality of these methods depends however on the programming environment. For example, in our experience, `frexp` is prohibitively slow. Accessing the bit representation may not be possible in every programming language.

Luckily, msb(f), pred(f), and succ(f) can be computed with a few standard floating-point operations. To compute the most significant bit, Siegfried Rump [**93**] considers the computation

$$\varphi = (2\varepsilon_{\mathtt{m}})^{-1} + 1, \qquad q \leftarrow \varphi \otimes f, \qquad m \leftarrow |q \ominus (1 - \varepsilon_{\mathtt{m}}) \otimes q|$$

and shows that $m = \mathrm{msb}(f)$ for $f \in \mathbb{F}$, if no overflow occurs. If $f \leq 2\varepsilon_{\mathtt{m}}\tau$, this computation is safe from overflow, since (for $\varepsilon_{\mathtt{m}} \leq \frac{1}{4}$):

$$\varphi f \;\leq\; ((2\varepsilon_{\mathtt{m}})^{-1} + 1) 2\varepsilon_{\mathtt{m}}\tau \;=\; (1 + 2\varepsilon_{\mathtt{m}})\tau \;\leq\; 2\tau(1 - \varepsilon_{\mathtt{m}}) \;\in\; \mathbb{F}.$$

This implies $q \in \mathbb{F}$. If $f > 2\varepsilon_{\mathtt{m}}\tau$ we perform the same computation, but scaled down by $\varepsilon_{\mathtt{m}}$. But then $f' = \varepsilon_{\mathtt{m}} f \leq \varepsilon_{\mathtt{m}} 2\tau(1 - \varepsilon_{\mathtt{m}}) < 2\varepsilon_{\mathtt{m}}\tau$, so we always get a correct result.

How to compute predecessor and successor was shown by Siegfried Rump, Paul Zimmermann, Sylvie Boldo, Guillaume Melquiond [**92**]. For $f \in \mathbb{F}$, they first consider the computation

$$\begin{aligned} &\psi = \varepsilon_{\mathtt{m}}(1 + 2\varepsilon_{\mathtt{m}}), && e \leftarrow \psi \otimes |f| \oplus \eta, \\ &\underline{f} \leftarrow f \ominus e, && \bar{f} \leftarrow f \oplus e \end{aligned} \tag{2.16}$$

and show that with $\varepsilon_{\mathtt{m}} \leq \frac{1}{32}$

$$\underline{f} = \begin{cases} \mathrm{pred}(f) \\ \mathrm{pred}(\mathrm{pred}(f)) \end{cases} \qquad \text{and} \qquad \bar{f} = \begin{cases} \mathrm{succ}(f) & \text{for } |f| \notin [\frac{1}{2}, 2]\varepsilon_{\mathtt{m}}^{-1}\eta \\ \mathrm{succ}(\mathrm{succ}(f)) & \text{for } |f| \in [\frac{1}{2}, 2]\varepsilon_{\mathtt{m}}^{-1}\eta \end{cases}.$$

Hence, $\underline{f}$ and $\bar{f}$ are always a lower and upper bound on f and optimal except for a small range of numbers near $\frac{1}{2}\varepsilon_{\mathrm{m}}^{-1}\eta$. The constant η used in this computation incorporates a possible performance penalty. On most architectures, operations involving denormalized numbers are significantly slower than other operations. Thus one should avoid denormalized constants, at least unless input numbers are denormalized anyway. Thus, the final algorithm PredSucc, distinguishes between several cases. It handles large numbers by Equation (2.16), but with η removed. Numbers with $\mathrm{msb}(f) \leq \frac{1}{2}\varepsilon_{\mathrm{m}}^{-1}\eta$ are handled directly by adding or subtracting η. The remaining numbers are scaled upwards and again handled by Equation (2.16).

Algorithm 2.11 (msb).
Assume $\varepsilon_{\mathrm{m}} \leq \frac{1}{4}$. *Let* $f \in \mathbb{F}$ *and compute* $m \leftarrow \textsc{msb}(f)$. *Then* $m = \mathrm{msb}(f)$.

Algorithm 2.12 (PredSucc).
Assume $\varepsilon_{\mathrm{m}} \leq \frac{1}{32}$. *Let* $f \in \mathbb{F}$ *and compute* $(p,s) \leftarrow \textsc{PredSucc}(f)$. *Then* $p = \mathrm{pred}(f)$ *and* $s = \mathrm{succ}(f)$.

1: **procedure** msb(f)
2: $\quad\varphi \leftarrow (2\varepsilon_{\mathrm{m}})^{-1} + 1$
3: $\quad$**if** $|f| \leq 2\varepsilon_{\mathrm{m}}\tau$ **then**
4: $\quad\quad q \leftarrow \varphi \otimes f$
5: $\quad\quad m \leftarrow |q \ominus (1-\varepsilon_{\mathrm{m}}) \otimes q|$
6: $\quad$**else**
7: $\quad\quad q \leftarrow (\varepsilon_{\mathrm{m}}\varphi) \otimes f$
8: $\quad\quad m \leftarrow \varepsilon_{\mathrm{m}}^{-1}|q \ominus (1-\varepsilon_{\mathrm{m}}) \otimes q|$
9: $\quad$**return** m

1: **procedure** PredSucc(f)
2: $\quad\psi \leftarrow \varepsilon_{\mathrm{m}}(1+2\varepsilon_{\mathrm{m}}) = \mathrm{succ}(\varepsilon_{\mathrm{m}})$
3: $\quad$**if** $|f| \geq \frac{1}{2}\varepsilon_{\mathrm{m}}^{-2}\eta$ **then**
4: $\quad\quad e \leftarrow \psi \otimes |f|$
5: $\quad\quad p \leftarrow f \ominus e$
6: $\quad\quad s \leftarrow f \oplus e$
7: $\quad$**else if** $|f| < \varepsilon_{\mathrm{m}}^{-1}\eta$ **then**
8: $\quad\quad p \leftarrow f \ominus \eta$
9: $\quad\quad s \leftarrow f \oplus \eta$
10: $\quad$**else**
11: $\quad\quad F \leftarrow \varepsilon_{\mathrm{m}}^{-1} f$
12: $\quad\quad e \leftarrow \psi \otimes |F|$
13: $\quad\quad p \leftarrow \varepsilon_{\mathrm{m}}(F \ominus e)$
14: $\quad\quad s \leftarrow \varepsilon_{\mathrm{m}}(F \oplus e)$
15: $\quad$**return** (p,s)

Both msb and PredSucc require only a few floating-point operations and they might be adapted further to specific uses. For example, branches can be avoided if it is known that they are never called for large or small numbers.

2.2.3. Floating-Point Expansions. We ignore for a moment the limitations by η and τ, i.e., consider floating-point arithmetic over $\tilde{\mathbb{F}}$. Based on TwoSum, FastTwoSum, and TwoProduct, Dekker [**20**] develops a so called *double length* arithmetic. A number x is represented by two floating-point numbers x_{lo} and x_{hi} with $x = x_{\mathrm{lo}} + x_{\mathrm{hi}}$. The summands x_{lo} and x_{hi} satisfy

$$|x_{\mathrm{lo}}| \leq \frac{\varepsilon_{\mathrm{m}}}{1-\varepsilon_{\mathrm{m}}}|x|. \tag{2.17}$$

Note the similarity to the error bound on δ in Theorem 2.6. Equation (2.17) implies that there are approximately $2p$ bits available in x_{hi} and x_{lo} to represent x. But x might have more bits, namely if there is a gap of zero bits between x_{hi} and x_{lo}. Double length arithmetic increases the precision and hence the accuracy, compared to plain floating-point arithmetic. The sum and product of two plain floating-point numbers can be computed exactly, but operations on double length numbers still involve rounding error.

Douglas Priest [**86**] extends the idea of double length arithmetic by allowing an arbitrary number of summands. Based on Priests work, Jonathan Shewchuk [**104**] gives improved, i.e., faster algorithms.

Let $a, b \in \mathbb{F}$ with $|a| \leq |b|$. Then a and b are *non-overlapping* if $\mathrm{msb}(a) < \mathrm{lsb}(b)$. Otherwise a and b *overlap*. A sequence $e_1, e_2, \ldots, e_n \in \mathbb{F}$ is called an *expansion* if its elements are pairwise non-overlapping and they are ordered by increasing magnitude, except that any of the e_i may be zero. More formally

$$e_j \neq 0 \quad \Rightarrow \quad \mathrm{msb}(e_i) < \mathrm{lsb}(e_j) \quad \text{for } 1 \leq i < j \leq n \tag{2.18}$$

must hold. An expansion $e = e_1, e_2, \ldots, e_n$ represents the number

$$E = \sum_{i=1}^{n} e_i.$$

Therefore, we call the elements of an expansion *summands*. We call an expansion *zero-free* if either all summands are non-zero or the expansion consists of only one summand.

The term non-overlapping refers to how the mantissae of two floating-point numbers behave, when aligning bits by significance. When two numbers are non-overlapping, the relevant parts of their mantissae, i.e., the shortest subsequences containing all non-zero bits, do not overlap, when aligning bits by significance. Please see Figure 2.4 where examples for different types of expansions, are given. Expansions have some nice properties. In a zero-free expansion, the sign of the expansion is the sign of the most significant summand. By the non-overlapping property, non-zero bits in different summands do not interfere with each other. Hence, we can talk about the sequence of bits given by an expansion. Summands may however have different signs, so the exact binary representation of the represented number is not directly given.

Shewchuk also considers two other, slightly more restrictive types of expansions. Let $a, b \in \mathbb{F}$ with $|a| \leq |b|$ be non-overlapping. Then a and b are *adjacent*, if $2a$ overlaps b. Otherwise, a and b are *non-adjacent*. An expansion is called non-adjacent, if its summands are pairwise non-adjacent.

An expansion is *strongly non-overlapping* if each summand is adjacent to at most one other summand and each summand that is adjacent to some other summand is a power of two. Hence, in a strongly non-overlapping expansion, adjacent summands come in pairs and both summands are a power of two. This definition may seem a bit artificial, but turns out to be the correct invariant for the fastest addition

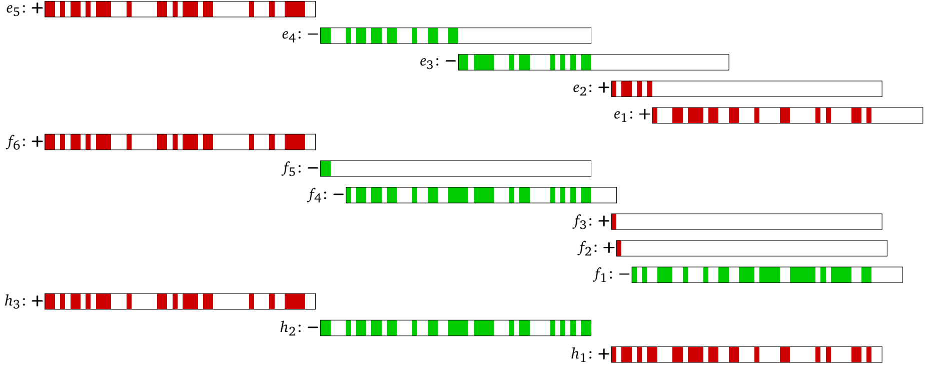


FIGURE 2.4. Several expansions representing the same number. Expansion e is non-overlapping but not non-adjacent. Expansion f is strongly non-overlapping. Expansion h is maximally non-overlapping and non-adjacent.

algorithm. Any non-adjacent expansion is also strongly non-overlapping and any strongly non-overlapping expansion is also non-overlapping.

Priest considers a different notion of non-overlappingness, which we call maximally non-overlapping. Let $a, b \in \mathbb{F}$ with $|a| \leq |b|$. Then a and b are *maximally non-overlapping* if $\operatorname{msb}(a) \leq \varepsilon_{\mathrm{m}} \operatorname{msb}(b)$. If a and b are maximally non-overlapping, then they are also non-overlapping, since

$$\operatorname{msb}(a) \leq \varepsilon_{\mathrm{m}} \operatorname{msb}(b) \leq \varepsilon_{\mathrm{m}} \tfrac{1}{2} \varepsilon_{\mathrm{m}}^{-1} \operatorname{lsb}(b) < \operatorname{lsb}(b).$$

An expansion $e = e_1, e_2, \ldots, e_n \in \mathbb{F}$ is maximally non-overlapping, if its summands are pairwise maximally non-overlapping. Clearly, a maximally non-overlapping expansion is also non-overlapping, but it need not be strongly non-overlapping or non-adjacent.

In a non-overlapping expansion, non-zero bits that fit into one summand may be spread over multiple summands. Maximally non-overlapping expansions enforce a more compact representation. The spacing between two maximally non-overlapping floating-point numbers a and b is best possible, there are at least $2p$ bits available in $a + b$.

Priest [**86**] considers floating-point arithmetic over $\tilde{\mathbb{F}}$ with faithful rounding, and shows how to perform basic error-free transformations in this setting. For floating-point arithmetic with rounding to nearest, his transformations simplify to FASTTWOSUM and TWOPRODUCT. He then presents algorithms for exact ring operations and arbitrarily accurate division over $\mathscr{F}$, based on these error-free transformations. He represents elements from $\mathscr{F}$ by zero-free, maximally non-overlapping expansions. A major component in his algorithms is a renormalization procedure which converts a sequence of floating-point numbers, ordered by magnitude and with only little overlap into a zero-free, maximally non-overlapping expansion.

Jonathan Shewchuk [**104**] gives improved, i.e., faster algorithms for exact ring operations over $\mathscr{F}$. His algorithms are based on TwoSum, FastTwoSum and TwoProduct and represent numbers by non-overlapping expansions. Shewchuk considers floating-point arithmetic with rounding to nearest and either tie-breaking to even or tie-breaking towards zero. Depending on the tie-breaking rule, his algorithms maintain different invariants for the processed expansions. The invariants are simpler for tie-breaking towards zero, but unfortunately this rule is not available in IEEE 754. Tie-breaking to even is however the default mode. An implementation of the algorithms is available at [**105**]. We present some of these algorithms now, since we use them in our attempts to defer dag creation in Section 5.1, where we also derive additional properties of them.

Expansion Arithmetic. Of course, a single floating-point number is an expansion, both non-adjacent and maximally non-overlapping. FastTwoSum, TwoSum, and TwoProduct produce maximally non-overlapping expansions, too. If ties are broken to even, then they also produce non-adjacent expansions.

Lemma 2.13. *Let (x, y) be the results from* FastTwoSum*,* TwoSum*, or* TwoProduct*, performed with tie-breaking to even. Then x and y are non-adjacent.*

Proof. We can assume that $y \neq 0$. Then, by Equation (2.13) we have $|y| \leq \varepsilon_{\mathtt{m}} \operatorname{msb}(x)$. If the inequality is strict, then x and y are non-adjacent. If however $|y| = \varepsilon_{\mathtt{m}} \operatorname{msb}(x)$, then the last bit in x must be zero, due to the tie-breaking rule. □

This observation plays a central role in the analysis of Shewchuk's algorithms. Shewchuk presents several algorithms for the addition of expansions, among them FastExpansionSum is the fastest.

Algorithm 2.14 (FastExpansionSum)**.**
We consider floating-point arithmetic over $\tilde{\mathbb{F}}$, with $\varepsilon_{\mathtt{m}} \leq \frac{1}{16}$, rounding to nearest and tie-breaking to even. Let $e = e_1, e_2, \ldots, e_m$ and $f = f_1, f_2, \ldots, f_n$ be two strongly non-overlapping expansions. Then FastExpansionSum *computes a strongly non-overlapping expansion $h = h_1, h_2, \ldots, h_{m+n}$ with*

$$\sum_{i=1}^{m} e_i + \sum_{i=1}^{n} f_i = \sum_{i=1}^{m+n} h_i.$$

FastExpansionSum *runs in time $O(m + n)$.*

```
1: procedure FastExpansionSum(e_1, e_2, ..., e_m, f_1, f_2, ..., f_n)
2:     Merge e_1, e_2, ..., e_m and f_1, f_2, ..., f_n into a single sequence g_1, g_2, ..., g_{m+n}
       of nondecreasing magnitude (except for stray zero summands).
3:     (Q, h_1) ← FastTwoSum(g_2, g_1)
4:     for i ← 3 to m + n do
5:         (Q, h_{i-1}) ← TwoSum(Q, g_i)
6:     h_{m+n} ← Q
7:     return (h_1, h_2, ..., h_{m+n})
```

The correctness proof for FastExpansionSum makes explicit use of the tie-breaking rule. Furthermore Shewchuk gives examples, where the input expansions are non-overlapping (resp. non-adjacent), but the output contains overlapping summands (resp. adjacent summands). It is however important to maintain some invariant since we want to apply algorithms iteratively. By choosing FastExpansionSum, we are thus limited to strongly non-overlapping expansions. It is straightforward to modify FastExpansionSum to produce zero-free output expansions, by only writing non-zero summands to h. Of course, FastExpansionSum can also compute the difference of two expansions, simply by negating one of the input expansions.

An alternative to FastExpansionSum is LinearExpansionSum, which maintains the non-overlapping property. LinearExpansionSum runs in time $O(m+n)$, too, but requires nine instead of six floating-point operations per summand. Finally, there is ExpansionSum, which maintains the non-overlapping property for any tie-breaking rule and the non-adjacent property, when tie-breaking to even is used. ExpansionSum does not involve any branches, since summands must not be ordered, but takes time $O(mn)$. Shewchuk suggests to use an unrolled form of ExpansionSum to create small expansions and FastExpansionSum with zero elimination for larger expansions. One can however not mix them arbitrarily, since ExpansionSum does not maintain the strongly non-overlapping property, even with tie-breaking to even.

The sum, or more precisely an expansion for the sum of k floating-point numbers can be computed using a divide and conquer approach in time $O(k \log k)$. First, recursively, compute two expansions for one half of the numbers each, then add these expansions using FastExpansionSum.

The basis for the multiplication of two expansions is ScaleExpansion, which allows to multiply an expansion with a single floating-point number.

Algorithm 2.15 (ScaleExpansion).
We consider floating-point arithmetic over $\tilde{\mathbb{F}}$, *with* $\varepsilon_{\mathtt{m}} \leq \frac{1}{16}$, *rounding to nearest and tie-breaking to even. Let* $e = e_1, e_2, \ldots, e_m$ *be a strongly non-overlapping expansion and* $f \in \tilde{\mathbb{F}}$. *Then* ScaleExpansion *computes a strongly non-overlapping expansion* $h = h_1, h_2, \ldots, h_{2m}$ *with*

$$f \times \sum_{i=1}^{m} e_i = \sum_{i=1}^{2m} h_i.$$

ScaleExpansion *runs in time* $O(m)$.

1: ***procedure*** ScaleExpansion(f, $e_1, e_2, \ldots, e_m$)
2: $(Q, h_1) \leftarrow$ TwoProduct(e_1, f)
3: ***for*** $i \leftarrow 2$ ***to*** m ***do***
4: $(T, t) \leftarrow$ TwoProduct(e_i, f)
5: $(Q, h_{2i-2}) \leftarrow$ TwoSum(Q, t)
6: $(Q, h_{2i-1}) \leftarrow$ FastTwoSum(T, Q)
7: $h_{2m} \leftarrow Q$
8: ***return*** $(h_1, h_2, \ldots, h_{2m})$

Thus, ScaleExpansion is compatible with FastExpansionSum. ScaleExpansion maintains the non-overlapping property for any tie-breaking rule and the non-adjacent property, when tie-breaking to even is used. To multiply two expansions, Shewchuk suggests to first multiply one expansion with each summand from the other expansion and add the resulting expansions by divide and conquer and FastExpansionSum. This allows to multiply two expansions with m and n summands in time $O(mn \log \min\{m,n\})$. The resulting expansion may have up to $2mn$ summands.

Of course, FastExpansionSum and ScaleExpansion may be used over $\mathbb{F}$, too. The results of ScaleExpansion are invalid, if overflow or underflow occurs. Due to Lemma 2.4, FastExpansionSum is not affected by underflow, but its results are invalid if overflow occurs. In Section 5.1 we give sufficient conditions for FastExpansionSum and ScaleExpansion to be free from overflow and underflow.

Compression. We already mentioned, that non-overlapping expansions may have significantly more summands than needed to represent a given number, cf. Figure 2.4. And indeed, ScaleExpansion and FastExpansionSum commonly output expansions, where many summands carry only a few non-zero bits, see for example the experiments in Section 4.2.3. An excessive number of summands, however, makes subsequent operations more expensive. The following algorithm Compress is designed to reduce the number of summands in an expansion, if possible.

Algorithm 2.16 (Compress).
We consider floating-point arithmetic over $\tilde{\mathbb{F}}$, with rounding to nearest and tie-breaking to even. Let $e = e_1, e_2, \ldots, e_m$ be a non-overlapping expansion. Then Compress *computes a non-adjacent expansion $f = f_1, f_2, \ldots, f_n$ with $n \leq m$, $f_i \neq 0$ for $1 \leq i \leq n$ and*

$$\sum_{i=1}^{m} e_i = \sum_{i=1}^{n} f_i, \qquad \left| \sum_{i=1}^{n-1} f_i \right| \leq 2\varepsilon_{\mathtt{m}} \operatorname{msb}(f_n).$$

Compress *runs in time $O(m)$.*

```
 1: procedure Compress(e_1, e_2, ..., e_m)
 2:     Q ← e_m
 3:     k ← m
 4:     for i ← m − 1 downto 1 do
 5:         (Q, q) ← FastTwoSum(Q, e_i)
 6:         if q ≠ 0 then
 7:             g_{k--} ← Q
 8:             Q ← q
 9:     g_k ← Q
10:     n ← 1
11:     for i ← k + 1 to m do
12:         (Q, q) ← FastTwoSum(g_i, Q)
13:         if q ≠ 0 then f_{n++} ← q
14:     f_n ← Q
15:     return (f_1, f_2, ..., f_n)
```

Note that it is not necessary to use three sequences e, f, and g to store the summands. Both f and g may be equal to e, i.e., e can be overwritten. COMPRESS maintains the non-overlapping property with any tie-breaking rule, but when tie-breaking to even is used, it converts a non-overlapping expansion into a non-adjacent expansion.

Unfortunately, Shewchuk does not discuss how much the number of summands might be reduced. But, following the correctness proof of COMPRESS, the intermediate sequence $g_k, g_{k+1}, \ldots, g_m$ satisfies $\text{msb}(g_i) \leq 2\varepsilon_{\mathtt{m}} \text{msb}(g_{i+1})$ for $k \leq i < m$. This is because in Line 5 we have $|q| \leq \varepsilon_{\mathtt{m}} \text{msb}(Q)$ and the subsequent addition of smaller, non-overlapping summands to q can increase q by at most a factor of two. The sequence $g_k, g_{k+1}, \ldots, g_m$ of summands is almost maximally non-overlapping. Hence we have $\text{msb}(g_k) \leq (2\varepsilon_{\mathtt{m}})^{m-k} \text{msb}(g_m)$ and get an upper bound on the number of summands of

$$m - k \leq \left\lceil \frac{\log_2(\text{msb}(g_k)/\,\text{msb}(g_m))}{p-1} \right\rceil. \tag{2.19}$$

The second loop in COMPRESS, computing f, does not increase the number of summands. Equation (2.19) relates the number of output summands to the number of bits required to represent e exactly. For floating-point arithmetic over $\mathbb{F}$, we get a static upper bound if we replace g_k with 2τ and g_m with η. For the `binary64` format, this yields an upper bound of 41 summands. Thus, COMPRESS may be used to avoid dynamic memory allocation. If a `binary64` expansion is larger than 41 summands, we may compress it. Then, no output expansion will ever have more than $2 \times 41 \times 41 = 3362$ summands. Better static or semi static bounds may be achieved if a priori information about input numbers and expressions is available. In Section 5.1 we show that COMPRESS is free from underflow and overflow for strongly non-overlapping expansions.

2.2.4. Accurate Floating-Point Summation. Using TWOPRODUCT, we can transform any polynomial expression over floating-point numbers exactly into a sum of floating-point numbers. This reduces the problem of computing the sign or an approximation of a polynomial expression to computing the sign or an approximation of a sum. Let $a_1, a_2, \ldots, a_k \in \mathbb{F}$, with $A = \sum_{i=1}^{k} a_i$. There are several interesting tasks, with increasing difficulty.

- Compute $\hat{A}$ and e_A, such that $|A - \hat{A}| \leq \mathsf{e}_A$.
- Compute $\text{sign}(A)$.
- Compute a faithful rounding $\tilde{\mathbf{fl}}(A)$.
- Compute $\mathbf{fl}(A)$, i.e., a floating-point number nearest to A.

For two numbers $a_1, a_2 \in \mathbb{F}$, we already know how to do any of these tasks. Let $(x, y) = \text{TWOSUM}(a_1, a_2)$, then we have

$$\hat{A} = x, \qquad \mathsf{e}_A = |y|, \qquad \text{sign}(A) = \text{sign}(x), \qquad \mathbf{fl}(A) = x.$$

Actually, it suffices to compute $x \leftarrow a_1 \oplus a_2$, since we can set $\mathsf{e}_A = \varepsilon_{\mathtt{m}}|x|$. But with TwoSum, we additionally have $a_1 + a_2 = x + y$ and no information about the original sum is lost. This observation is the key to accurate summation of more than two summands. Using error-free transformations, we can rewrite the sum repeatedly, while simultaneously simplifying the task of computing an accurate result. We now briefly present techniques and algorithms by other authors, that we use a starting point for our work or for comparison in experiments later on.

Expansions. Floating-point expansion allow to evaluate polynomial expressions exactly, so we may use them for summation, too. We may first transform a sequence $a_1, a_2, \ldots, a_k$ of floating-point numbers into a zero-free expansion $e_1, e_2, \ldots, e_m$, using the divide and conquer approach. The leading summand e_m carries the sign of the sum and an approximation and error bound are given by

$$\hat{A} = e_m, \qquad\qquad \mathsf{e}_A = |e_{m-1}| + \operatorname{lsb}(e_{m-1}).$$

This approximation may however be poor, since e_m may carry a single non-zero bit only. If we compress the expansion additionally, the same bounds hold, but then e_m is a faithful approximation of A.

ESSA. Helmut Ratschek and Jon Rokne [**88**] present an algorithm, called ESSA for "exact sign of sum algorithm", to compute the sign of a sum of floating-point values exactly. ESSA iteratively performs error-free transformations on the largest positive and the smallest negative number in the current sum, thereby decreasing the sum of the absolute values of the summands. The iteration continues until the sum vanishes, or the largest positive number clearly dominates the sum of negative ones, or vice versa. In Section 4.1.1 we show that instead of the error-free transformation used by Ratschek and Rokne, FastTwoSum may be used. This modification leads to a much improved variant, which we confirm theoretically and experimentally. The improved variant is still slower than alternative approaches, but has the unique property of being completely immune to both overflow and underflow.

Compensated Summation. A well known technique to improve the accuracy of floating-point summation is *compensated summation*. First, the summands a_1, a_2, …, a_n are added using an error-free transformation, e.g., TwoSum. Besides an approximation $\hat{a}$, this gives us $n - 1$ error terms. In a second step, the error terms are summed up first, and then added to $\hat{a}$, resulting in a better approximation $\hat{A}$. Compensated summation can be traced back at least to William Kahan [**47**]. When this paper was written, floating-point arithmetic was far from standardized and the error terms were inexact. Takeshi Ogita, Siegfried Rump and Shin'ichi Oishi [**77**] analyze compensated summation and prove an error bound e_A for $\hat{A}$, computable in floating-point arithmetic. In Section 4.1.2 we give an improved error bound based on work by Siegfried Rump [**91**] and show how to use compensated summation iteratively for exact sign computation.

AccSum. Let $a_1, a_2, \dots, a_k \in \mathbb{F}$ and $A = \sum_{i=1}^{k} a_i$. The AccSum algorithm by Rump, Ogita and Oishi [**94**] allows to compute a faithful approximation of A. AccSum requires floating-point arithmetic over $\mathbb{F}$ with rounding to nearest and tie-breaking to even. The number of summands is limited to k with $4(k+2)^2\varepsilon_{\mathtt{m}} < 1$.

Let $M \in \mathbb{N}$ with $k < 2^M$ and let σ be a power of two such that $|a_i| \le 2^{-M}\sigma$ for $1 \le i \le k$. Then AccSum performs the following extraction step.

```
1: procedure Extract(σ, a_1, a_2, ..., a_k)
2:     μ_0 ← 0
3:     for i ← 1 to k do
4:         b_i ← (σ ⊕ a_i) ⊖ a_i
5:         a'_i ← a_i ⊖ b_i
6:         μ_i ← μ_{i−1} ⊕ b_i
7:     return (μ_n, a'_1, a'_2, ..., a'_k)
```

Extract splits each summand a_i into two parts b_i and a_i' such that

$$a_i = b_i + a_i', \qquad b_i \in \varepsilon_{\mathtt{m}}\sigma\mathbb{Z}, \qquad |a_i'| \le \varepsilon_{\mathtt{m}}\sigma.$$

The constant σ is chosen such that $\mu = \sum_{i=1}^{n} b_i$ is a floating-point number and therefore $\mu = \mu_n$. Indeed, $\mu < \sigma$, and $\mu \in \varepsilon_{\mathtt{m}}\sigma\mathbb{Z}$, so for $\mu \neq 0$ we have $\operatorname{msb}(\mu) \le \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\operatorname{lsb}(\mu)$. Thus, AccSum transforms

$$\sum_{i=1}^{k} a_i \qquad \text{into} \qquad \mu + \sum_{i=1}^{k} a_i',$$

without rounding error. Now, two cases may occur. Either, μ is sufficiently large. Then the a_i' are added straightforwardly into an approximation $\hat{a}$ and $\mu \oplus \hat{a}$ is a faithful approximation of A. Or μ is small. Then, the extraction step is repeated, transforming

$$\mu + \sum_{i=1}^{k} a_i' \qquad \text{into} \qquad \mu + \nu + \sum_{i=1}^{k} a_i''.$$

But now, σ can be chosen such that $\mu' = \mu + \nu$ is a floating-point number, too. Hence we end up with

$$\mu' + \sum_{i=1}^{k} a_i'',$$

and again, AccSum decides based on μ' whether to stop or to continue. The hard part in the design of AccSum is to determine how to choose σ, as well as the threshold that separates large μ from small μ in a way that makes the outlined strategy work.

In [**95**], Rump, Ogita and Oishi present variants of AccSum for different tasks. For example to compute sign(A) only, some constants in AccSum may be relaxed, possibly allowing fewer extraction steps. Other variants allow to compute a floating-point approximation of A according to a rounding mode, i.e., the next larger, next smaller or nearest floating-point number. These variants are computationally more

expensive. Finally, there is also a variant that transforms a sum into a maximally non-overlapping expansion. Siegfried Rump [**93**] presents an improved variant called FastAccSum. While in AccSum, an extraction step for k summands takes $4k$ floating-point operations, FastAccSum achieves essentially the same with only $3k$ floating-point operations.

Application in Geometric Computing. Many authors have proposed to implement robust predicates for Computational Geometry based on error-free transformations or exact floating-point summation, see for example [**21, 39, 81, 87, 104**]. Possibly the earliest example is [**78**], which assumes the availability of an exact scalar product. These approaches lead to some of the fastest predicate implementations to date. The downside is, that they involve careful error analysis tuned to the expression and underlying method [**81, 104**]. Furthermore, the problem of overflow and underflow remains, though rarely relevant in practice.

But even if applied straightforwardly, exact arithmetic based on error-free transformations is faster than traditional software number types for geometric predicates. Software number types are asymptotically fast, but involve a constant overhead per operation. Algorithms based on error-free transformations usually have a worse asymptotic running time, but less constant overhead. For example, no initial conversion into a software number is necessary. This tradeoff pays off in geometric predicates, which require relatively low precision only.

In Chapter 4 we present improved algorithms for accurate floating-point summation. We implement geometric predicates based on our and other algorithms and evaluate their efficiency experimentally. We also give new algorithms for the efficient conversion of expansions to bigfloat numbers. In Chapter 5 we integrate algorithms based on error-free transformations into our number type *RealAlgebraic*. Our approaches are based on computing with expansions, as well as on accurate summation algorithms. We rigorously treat and handle limitations of these algorithms, e.g., underflow and overflow, to relieve the user from these problems and keep *RealAlgebraic* user-friendly.

CHAPTER 3

RealAlgebraic – an expression dag based number-type

In this chapter we report on our new number type, *RealAlgebraic*, based on the basic principles and algorithms described in Section 2.1. Our goal in the development of *RealAlgebraic* is to improve the efficiency of such number types, while maintaining their generality and usability.

From the basic approach of storing expressions to allow lazy adaptive evaluation many design choices arise, exemplified in the differences between `Lazy_exact_nt`, `leda::real`, and `CORE::Expr`. One has to choose for example a dynamic floating-point filter and an implementation of arbitrary precision arithmetic. Furthermore there is a large range of possible evaluation and sign computation schemes, even if one decides to base them on precision driven arithmetic.

In *RealAlgebraic* some of these design choices are captured and exposed in terms of exchangeable modules. For each choice, several options are implemented and can be chosen by the user. In this way, *RealAlgebraic* is not a single number type but actually a family of related number types. The goal of this approach is to allow and simplify experiments that let us determine the impact of certain design choices on the overall performance of the number type.

All *RealAlgebraic* variants provide the same interface and, with the exception of running time, behave the same from the users point of view. *RealAlgebraic* supports field operations and radicals of arbitrary degree, but not the $\diamond$-operator. It allows for exact sign computation and comparison results are always correct. Furthermore, *RealAlgebraic* allows the computation of arbitrarily accurate bigfloat approximations, where the user can specify the relative and absolute error a priori.

RealAlgebraic is implemented in `C++`. Through operator overloading, it may be used like any built-in number type, but with the added benefit of exactness. *RealAlgebraic* depends on three third party libraries, though not all must be available. A working number-type can be obtained if either LEDA [**53**] or both BOOST [**4**] and MPFR [**71**] are available. Since there are many choices to be made, we provide a default *RealAlgebraic* variant. It is provided under the name `Default_real_algebraic` in the file `Default_real_algebraic.hpp`. Including the file `real_algebraic_traits_for_cgal.hpp` makes any *RealAlgebraic* variant fit for usage with CGAL.

In this chapter we first present the software design of *RealAlgebraic*, i.e., how we made important parts of the implementation exchangeable and we present the options

we implemented for several of these parts. Then we discuss some improvements to our expression dag evaluation algorithms which, though small individually, together have a noticeable impact on running time. We identify a problem in current dag evaluation algorithms which negates the advantage of sharing common sub-expressions and present two possible improvements. Finally, we compare the efficiency of several *RealAlgebraic* variants in the context of various geometric algorithms. The default *RealAlgebraic* variant is the result of these experiments.

3.1. Policy Based Design

We use generic programming based on the template mechanism of `C++` to make the relevant parts of our implementation exchangeable. In generic programming, a *Concept* is an abstraction defined by a set of syntactical and semantical requirements. Concrete types that actually fulfill the requirements of a concept are called a `Model` of this concept.

According to Alexandrescu [**1**], a policy is a concept that encapsulates a certain behavioral aspect of an algorithm or class. He illustrates this on the task of creating a new object on the heap. The policy *CreationPolicy* depends on a template parameter `T` itself and mandates 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 create the object in a location raised from a memory pool by using placement `new`. Placement `new` performs no memory allocation and instead creates a new object in a memory location provided by the user. A class based on one or more policies is called a *host class*. Host classes have to be instantiated at compile time by selecting an appropriate model for each policy. If a host class is based on several policies, it is desirable 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 raises memory from a pool with a *DestructionPolicy* model that returns memory to the operating system. In fact, handling creation and destruction by different policies constitutes bad design.

The setup of one or more policies controlling the behavior of a host class is a variant of the strategy design pattern [**34**], 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. Providing policies as template parameters allows run-time efficient, very fine-grained control. An overview of our design in *RealAlgebraic* is given in Figure 3.1, showing all concepts, their models and relations between them.

Our main host class is `Real_algebraic`, which is parameterized by three policies. The *ExpressionDagPolicy* provides strategies for operations on the dag, i.e., creation of dag nodes, reference counting, dag evaluation and sign computation. The complementary *LocalPolicy* provides a strategy to postpone or avoid the creation of dag nodes, by using an explicit and exact number representation stored in the handle.

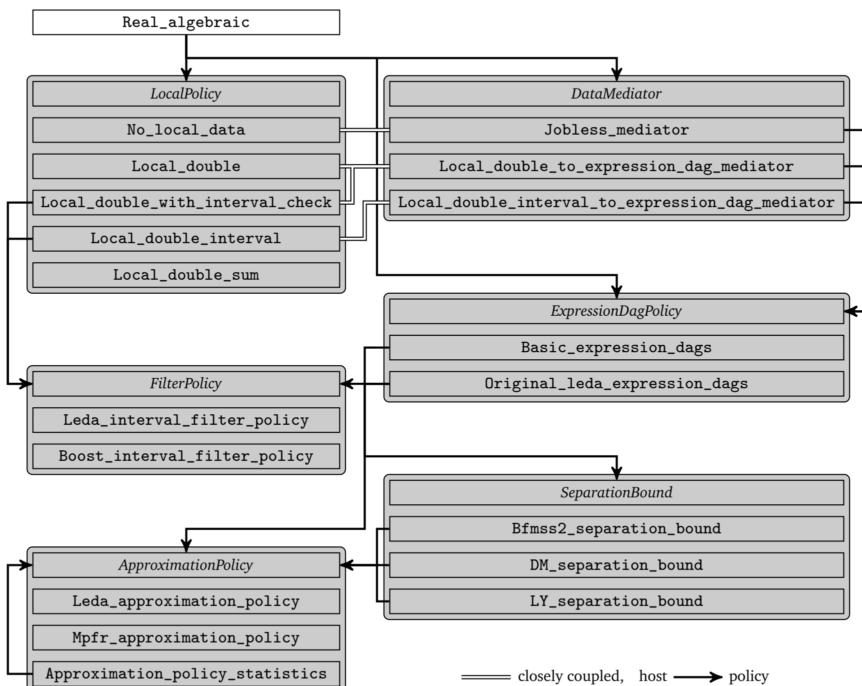


FIGURE 3.1. Concepts and models in *RealAlgebraic*.

The *DataMediator* provides conversion from the representation used by a *LocalPolicy* to a dag representation.

LocalPolicy. The main cost factors of computing with expression dags are dynamic memory management and bigfloat arithmetic. Computing even the first approximation of an expression is sometimes orders of magnitudes slower than evaluating the expression with floating-point arithmetic. Therefore it might be faster to compute the result of an arithmetic operation in some way exactly and store it in the handle, if possible, and resort to an expression dag only if this approach fails. If an exact value is stored in a handle we say it is represented locally. A *RealAlgebraic* may be represented locally or by an expression dag or both. In our implementation, the local representation in the handle is inherited from the *LocalPolicy* model given as parameter to `Real_algebraic`.

A *LocalPolicy* model must provide constructors, arithmetic operations and sign computation on the local representation. All these operations may signal failure, in which case they must be forwarded to the expression dag. A *LocalPolicy* model does not need to expose anything about the internal representation. Only the success or

failure of certain operations is of interest. A part of the required interface is shown below.

```
LocalPolicy(double d);
```

creates a new *LocalPolicy*. The created *LocalPolicy* represents d iff d can be represented locally.

```
bool local_creation_succeeded(double d);
```

returns if d is represented by `(*this)`, given that `(*this)` has been created from d.

```
bool local_multiplication(const LocalPolicy a,
                          const LocalPolicy b);
```

tries to compute a local representation for the product of a and b. If the product can be computed locally, a local representation is indeed computed and true is returned. Otherwise, false is returned.

```
bool local_sign(int& s);
```

tries to compute the sign of `(*this)` locally and stores it to s. If the sign can be computed, true is returned. Otherwise, false is returned.

Constructors cannot return a Boolean value indicating whether their argument is locally representable, so we allow to create a *LocalPolicy* right away and check afterwards if a local representation is possible. For a variety of local representations this interface allows for an efficient implementation of the creation process. To check for success after the creation, we may compare to the stored number, or return a flag that has been set by the constructor, or simply always return true, e.g., when all floating-point numbers are locally representable. Which implementation is best, depends on the specific number type used for local representation. We provide the following *LocalPolicy* models.

`No_local_data` (`nodaL`): Here the strategy is simply to not store any representation locally. All operations are immediately forwarded to the *ExpressionDagPolicy*. This simulates the behavior of both `CORE::Expr` versions where no attempts to defer dag creation are made.

`Local_double` (`doubL`): Stores one floating-point number plus a boolean value, which indicates whether this number is exact. Checks the exactness of arithmetic operations using error-free transformations. Supports $+, -, \times, /, \sqrt[2]{\ }$, but not $\sqrt[d]{\ }$.

`Local_double_with_interval_check` (`dwicL`): Stores the same data as `Local_double`, but checks the exactness of arithmetic operations using interval arithmetic, supplied by *FilterPolicy*. Supports $+, -, \times, /, \sqrt[d]{\ }$. This *LocalPolicy* models the behavior of `leda::real`, if instantiated with `Leda_interval_filter_policy` as *FilterPolicy*.

`Local_double_interval` (`dintL`): Stores a floating-point interval, based on *FilterPolicy* and always computes an interval enclosure, even if the interval is not a singleton. Forwards arithmetic operations to the *ExpressionDagPolicy*, if the

interval is not a singleton. Sign computation is however forwarded only, if the interval contains zero. Supports $+, -, \times, /, \sqrt[d]{\ }$.

For all *ExpressionDagPolicy* models currently available, the interval stored by `Local_double_interval` is identical to the one in the expression dag, so the avoidance of sign computation on the dag is nearly pointless. It would however be interesting, to combine this model with an expression dag that does not store a floating-point interval. This saves space in the dag node, but keeps a floating-point interval for all numbers which are still accessible to the user. It would further be interesting to extend the *DataMediator* concept to allow a reverse conversion and thus improvement of the local interval. This way, only those intervals are improved that might be useful later on, in contrast to improving all intervals in a dag.

`Local_double_sum` **(**`dsumL`**):** Stores a sequence $a_1, a_2, \ldots, a_n$ of floating-point numbers, representing their sum $\sum_{i=1}^{n} a_i$. Supports ring operations only and performs arithmetic operations and sign computation using algorithms based on error-free transformations. The behavior of `Local_double_sum` is governed by several further policies, fixing for example the number of summands or the set of arithmetic operations. Chapter 5 is dedicated to an in depth discussion of `Local_double_sum` and its policies.

DataMediator. If we do not succeed in performing a task locally, we resort to the expression dag. For an arithmetic operation, this may require to first transform the local representation of the operands into an expression dag representation. This conversion is done by a *DataMediator* model. The *DataMediator* plays a central role in orthogonalizing *LocalPolicy* and *ExpressionDagPolicy*. We would like to allow virtually any kind of exact number representation inside a *LocalPolicy* and we have to provide a way to convert this representation into an expression dag. Using a bigfloat type as intermediate representation is a straightforward option, but it restricts local representations to those exactly convertible to a bigfloat type and involves the possibly unnecessary cost for this conversion. Our solution is to provide the conversion method separately with a *DataMediator*. A *DataMediator* model may depend 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 separate *DataMediator* for each pair of *LocalPolicy* and *ExpressionDagPolicy*. In practice, the conversion method depends more on the *LocalPolicy* than the *ExpressionDagPolicy*. Many *DataMediator* models in our current design convert the local data to a floating-point number or bigfloat, and create a single dag node storing it. Those can in fact be used with any *ExpressionDagPolicy*.

Except for `Local_double_sum`, each *LocalPolicy* model above has a unique corresponding *DataMediator*, which can be found in Figure 3.1. Since these mediators need to convert at most a single floating-point number to an expression dag representation, they work with any *ExpressionDagPolicy*.

ExpressionDagPolicy. An *ExpressionDagPolicy* model provides operations on an expression dag, i.e., dag creation, sign computation and computation of arbitrarily accurate approximations. The key tools for these tasks are a floating-point filter, bigfloat arithmetic and a separation bound. We capture the requirements on these tools in three concepts, the *FilterPolicy*, the *ApproximationPolicy*, and the *SeparationBound*. Both our *ExpressionDagPolicy* models allow to interchange these three tools. Therefore, they do not provide a single expression dag implementation, but a family of closely related ones.

`Original_leda_expression_dags` **(**`oledaD`**):** This model is originally based on the source code for `leda::real` and implements the dag creation and evaluation strategy of `leda::real`, as described in Section 2.1.2. It does however incorporate the necessary changes to make floating-point filter, bigfloat arithmetic and separation bound exchangeable.

`Basic_expression_dags` **(**`basicD`**):** This model is similar to the one above, but includes some improvements to data storage and the sign computation algorithm. Most importantly it employs a new scheme to propagate the accuracy requirements in precision driven arithmetic. We discuss these improvements in Section 3.2.

FilterPolicy. The *FilterPolicy* concept specifies an interface for a dynamic floating-point filter or interval arithmetic and captures those features of interval arithmetic relevant for its use in *RealAlgebraic*. Any model for *FilterPolicy* represents an interval and provides arithmetic satisfying the interval inclusion property Equation (1.8). The interface to access or modify the interval supports two representation paradigms: midpoint and radius, or lower bound and upper bound. Depending on the actual representation, accessing or modifying the interval may therefore require floating-point computation. For this reason, the interface does not guarantee exact access to the interval, but guarantees the inclusion property only.

`Leda_interval_filter_policy` **(**`ledaF`**):** This model is based on interval arithmetic in LEDA, more precisely `leda::interval_bound_absolute`. This class is an implementation of the dynamic floating-point filter by Burnikel et al. [**13**] and represents intervals by midpoint and radius.

`Boost_interval_filter_policy` **(**`boostF`**):** This model is based on interval arithmetic in BOOST, more precisely `boost::numeric::interval<double>`. BOOST implements interval arithmetic as described by Brönnimann et al. [**5**]. Intervals are represented by upper and lower bound.

ApproximationPolicy. The *ApproximationPolicy* concept captures the functionality required from bigfloat arithmetic in *RealAlgebraic*. A model must provide a bigfloat type *Approximation* as well as approximate arithmetic operations on this type. The required interface for arithmetic operations is the following.

```
static bool mul(Approximation& c,
                const Approximation& a,
                const Approximation& b,
                const Precision p,
                const RoundingMode rm);
```

computes $a \times b$ rounded to at least p bit precision according to the rounding mode `rm` and stores it in c. If no rounding occurred, i.e., if $c = a \times b$, true is returned. Otherwise false is returned. The references a, b and c may refer to the same variable.

We do not require the result to be rounded to *exactly* p bits. In MPFR, precision is associated with variables, not operations, making it infeasible or at least inefficient to enforce rounding to exactly p bits, e.g., in case $c = a$.

Next to arithmetic operations, an *ApproximationPolicy* model must provide some auxiliary types and methods. Important is here the *Exponent* type, which arises whenever $\log|x|$ instead of $|x|$ is used in a computation. This is the case in the error bound computation for precision driven arithmetic and in the computation of separation bounds.

`Mpfr_approximation_policy` **(**`mpfrA`**):** Is based on the MPFR library [**71**]. In Section 4.2 we present new methods to convert expansions exactly into bigfloat numbers. It turns out, our new methods are faster even for a single floating-point number, see Section 4.2.3. Thus, `Mpfr_approximation_policy` uses the new conversion instead of the one provided by MPFR. The *Exponent* type is a hardware integer type and thus might wrap around and lead to inexact error bound computation on rare occasions.

`Leda_approximation_policy` **(**`ledaA`**):** Based on the `leda::bigfloat` number type from LEDA. For $\sqrt[d]{}$ with $d > 2$, this model does not support the required rounding modes but only guarantees the associated relative rounding error. It is hence not quite conforming to *ApproximationPolicy*, but for the actual usage of *ApproximationPolicy* inside current *ExpressionDagPolicy* models this poses no problem. The *Exponent* type is the exact number type `leda::integer`, therefore all error bound computations are safe.

`Approximation_policy_statistics`**:** The purpose of this model is to collect statistics about the usage of bigfloat arithmetic in *RealAlgebraic*. It is not really a *ApproximationPolicy* itself, but must be parameterized by another *ApproximationPolicy*, which then provides the actual functionality. For each type of arithmetic operation it collects a histogram of how often an operation of certain precision is performed.

The mantissa of a bigfloat number is usually stored as an array of integers, where each integer stores a part of the mantissa. These integers are called *words* or *limbs*. To make the histogram more compact, we measure precision not in the number of bits but in the number of limbs. For addition and subtraction we collect a single histogram but separate histograms for multiplication, division and radicals.

While this is certainly a very rough approximation of the running time spend for a single arithmetic operation, it does suffice for our purposes of gaining some insight into the usage of bigfloat arithmetic in *RealAlgebraic*. See Section 3.3 for how to use this model and Figure 3.5B for some collected statistics.

SeparationBound. The *SeparationBound* concept captures a separation bound. The following models are implemented.

`Bfmss2_separation_bound`: Implements the BFMSS[2] separation bound, as described by Pion and Yap [**84**].

`DM_separation_bound`: Implements the DM separation bound as described by Mignotte [**61, 12**]. Note that this bound can be significantly worse then the improved version by Li and Yap [**55**].

`LY_separation_bound`: Implements the new separation bound presented by Li and Yap [**55**], but uses `DM_separation_bound` internally, i.e., not the improved DM bound from [**55**].

MEMORY ALLOCATION. By default, *RealAlgebraic* allocates memory for dag nodes from a memory pool. It uses the pool implementation from either LEDA or BOOST, depending on availability. There is no corresponding concept or policy, the user may however select a specific pool implementation at compile time using preprocessor macros.

COMPOSING A *RealAlgebraic* VARIANT. To compose a *RealAlgebraic* variant, we have to instantiate `Real_algebraic` with its three policies, which may themselves be host classes depending on further polices. Instead of having in each host class one template parameter per policy, we decided to have a single template parameter only. The parameter has to be a class collecting all required policies as nested types. By feeding the collecting class itself to the models collected inside it, all necessary policies can be subsumed in a single class, resulting in a convenient way to create a *RealAlgebraic* variant.

We now show how to create the default *RealAlgebraic* variant and custom variants. We provide the policies for the default variant first in a templated class as shown below.

```
template <class Derived>
struct Default_real_algebraic_policies_base{
  typedef No_local_data                         LocalPolicy;
  typedef Jobless_mediator<Derived>             DataMediator;
  typedef Basic_expression_dags<Derived>        ExpressionDagPolicy;
  typedef Leda_interval_filter_policy           FilterPolicy;
  typedef Mpfr_approximation_policy             ApproximationPolicy;
  typedef Bfmss2_separation_bound<Derived>      SeparationBound;
};
```

While these are the preferred policies, as determined by our experiments in Section 3.4, the actual choice also depends on the availability of third party libraries BOOST, LEDA, and MPFR. The actual default collection of policies is created by deriving

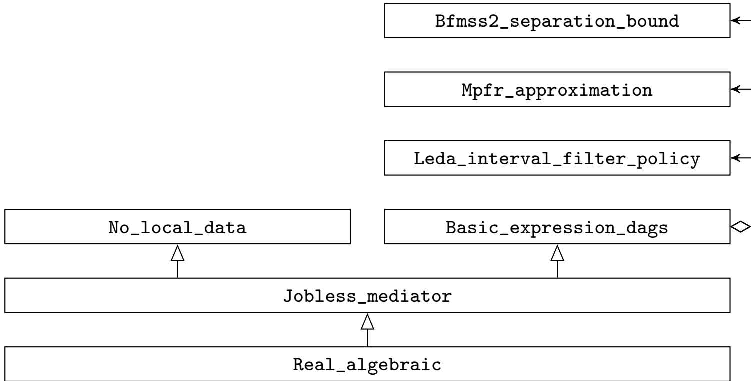


FIGURE 3.2. Collaboration of classes in `Default_real_algebraic`.

from the templated collection, providing the derived class as template parameter. In this way, each class depending on policies receives the collection as template parameter. This allows to create the default *RealAlgebraic* variant.

```
struct Default_real_algebraic_policies :
  public Default_real_algebraic_policies_base<
              Default_real_algebraic_policies > {};

typedef Real_algebraic<Default_real_algebraic_policies>
                                              Default_real_algebraic;
```

Instantiating `Real_algebraic` with a set of policies generates a small class hierarchy. The one generated for `Default_real_algebraic` is shown in Figure 3.2.

We could have provided a set of policies to `Real_algebraic` without using a templated base class, but doing so enables us to easily exchange one policy without having to specify the remaining ones. For example we might want to use `Local_double` as *LocalPolicy* but otherwise use the default parameters. Then we can do this in the following way.

```
struct Custom_policies :
  public Default_real_algebraic_policies_base<Custom_policies> {
  typedef Custom_policies                                  P;
  typedef Local_double<P>                                  LocalPolicy;
  typedef Local_double_to_expression_dag_mediator<P>    DataMediator;
};

typedef Real_algebraic<Custom_policies>        Custom_real_algebraic;
```

Comparison to Other Number Types. Our design goes beyond that of previous expression dag based number types. There is no genericity in `leda::real` or `CORE::Expr` 1. The number type `Lazy_exact_nt<NT>` adds adaptivity to any exact number type `NT` in a generic way. Our approach is however different. We make the inner parts of our number type exchangeable to better understand their interaction and influence on the performance, and to ultimately create better number types. Closest to *RealAlgebraic* comes `CORE::Expr` 2, which makes floating-point filter, separation bound and a so called arithmetic kernel exchangeable. While the first two parameters are in correspondence to our *FilterPolicy* and *SeparationBound* concepts, the arithmetic kernel of `CORE::Expr` 2 provides arbitrary precision interval arithmetic. In *RealAlgebraic* only the underlying bigfloat arithmetic is exchangeable, we leave the interval computation to an *ExpressionDagPolicy* model. Through *LocalPolicy*, we add the ability to combine different expression dag implementations with strategies to avoid dag creation, an option which is not available in any other number type.

3.2. Expression Evaluation

The general strategies for expression dag creation and expression evaluation in `Original_leda_expression_dags` and `Basic_expression_dags` are very similar to those in `leda::real`. But while `Original_leda_expression_dags` only involves changes necessary to make floating-point filter, bigfloat arithmetic and separation bound exchangeable, the strategy of `Basic_expression_dags` differs in a few more details. We now discuss these modifications.

Reorganization of Node Data. A common strategy for the storage of data in a dag node is, to only store the data absolutely necessary in the node itself and move all other data variables into another structure, created upon request later. For example, a bigfloat approximation is not always needed, but creating the variable holding it requires expensive dynamic memory allocation. It is thus moved to another structure and created later, if necessary.

In `Basic_expression_dags`, we have a common node type for all operations and only one additional structure. Both `leda::real` and `Original_leda_expression_dags` have two additional structures, one for the approximation and one for the separation bound. `CORE::Expr` 2 also has these two, plus a third one for caching the sign and bounds. In `Basic_expression_dags`, the additional structure is created when a bigfloat approximation is computed for the first time. While the separation bound might be needed only even later, the runtime for initializing the approximation easily dominates the cost for creating separation bound variables. No separation bound is computed at this point. By having only one structure, we need to store only one pointer in the node itself, reducing its size. We also moved a flag required for dag traversal out of the dag node into the extra structure.

The dag node itself contains the following variables only: The floating-point filter, one pointer for additional data, two pointers for operands, and two integers for the

reference count and node type. Four bits of the node type are used for the type flag, i.e., constant floating-point, constant bigfloat, or $+, -, \times, /, \sqrt[d]{\ }$, the remaining bits store d in case of a $\sqrt[d]{\ }$ node. Thus, on a 64 bit architecture a dag node has a size of 48 bytes, compared to 72 bytes in `Original_leda_expression_dags`. As long as no bigfloat operations are necessary, the memory requirement for a dag is minimized. This may help to improve the cache performance of an algorithm in this case, without noticeably slowing down the case where bigfloat operations are necessary.

Careful Floating-Point Filter Update. It is generally useful to update the floating-point filter stored in a dag node, once a high precision approximation has been computed, since this node may be used as operand in the creation of new nodes later on. This leads to a tighter interval in the new node and may avoid bigfloat operations. Of course, it is only useful to perform an update for nodes that are still directly accessible to the user, but this is expensive to check. Updating the floating-point filter involves bigfloat operations and has non-negligible cost.

We update the floating-point filter only, if this is reasonable considering the interval size. Let I be some interval with midpoint m and radius r, containing and representing some unknown number x. If r is on the order of the floating-point rounding error associated with m, i.e., $r \lesssim \varepsilon_{\mathtt{m}}|m|$, then it is very unlikely, that I can be improved. If I is represented by midpoint and radius, then x must be of the form $x = m + \delta$, with $|\delta| \ll r$. If I is represented by upper and lower bound, no improvement is possible. To allow for some widening in the conversion from bigfloat approximation to floating-point filter, we do not update intervals once $r \leq 4\varepsilon_{\mathtt{m}}|m|$.

Logarithmic Error Bound Computation. We need to compute several types of error bounds. In an initialization step, we report the error of an approximation bound bottom up, while in precision driven arithmetic we request accuracy top down, see Figure 2.1. The actual error bound formulas in these two cases are very similar. Finally, there is the separation bound computation. In all expression dag based number types, the separation bound is computed logarithmically, i.e., any parameter $x > 0$ is maintained as $\log x$ with the necessary changes in formulas.

In `Original_leda_expression_dags`, the bottom up error, as well as the top down accuracy requirement are computed directly, using bigfloat arithmetic with low precision. To minimize computational cost, the precision is selected such that all numbers may be stored using a single limb. The most expensive operations, i.e., division and radicals, are already avoided by computing logarithmically. In `Basic_expression_dags`, all top down accuracy requirements are computed logarithmically, avoiding any bigfloat operation.

By computing logarithmically we save some bigfloat operations, in a range of precision where the bigfloat arithmetic is exceptionally expensive compared to the accuracy it provides. By doing so, we get more restrictive accuracy requirements further down in the dag, which translate to larger precision for the bigfloat operations to compute approximations. Compared to direct error bound computation, the

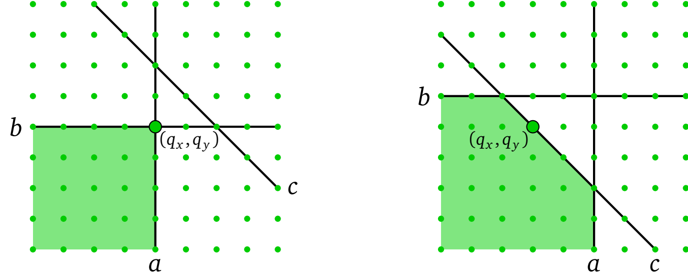


FIGURE 3.3. Two tiny integer programs.

necessary precision increases by one or two bits per stage, which is acceptable as long as the depth of the dag is not too large.

Bottom up error computation in the initialization step remains however direct, as this step is only used when the floating-point filter fails to provide a bound, e.g., when floating-point overflow occurred in the filter.

NEW ACCURACY PROPAGATION RULES. Besides computing accuracy requirements logarithmically, we also use a different set of rules to compute these accuracy requirements. Our new rules for accuracy propagation are a refinement of those used in `leda::real` [**15**] and `Original_leda_expression_dags`. They do resemble those by Chee Yap [**116**] and subsequently refined by Zilin Du [**24**] and used in `CORE::Expr`, by decoupling the actual recomputation of child nodes from computing accuracy requirements for them. The new rules remove the necessity implied by Equation (2.1) to recompute one child before computing the accuracy requirement for the other child. The new rules do however further increase the accuracy requirements on operands, for a result of the same quality. We present a new dag evaluation algorithm that benefits from the decoupling in Section 3.3.

As an example, we now derive the new rule for the improvement of multiplication nodes. Recall, that for each node v in the dag we maintain an approximation $\hat{v}$ and corresponding absolute error e_v with

$$|\hat{v} - v| \le \mathsf{e}_v$$

Let x, y, z be dag nodes with $z = xy$. We start with an error estimate analogous to Equation (2.2) but get rid of the dependency on $\hat{x}$, which forces us to recompute $\hat{x}$ first. As a result, we get an additional second order error term.

$$\begin{aligned} |\hat{z} - z| &\le |\hat{z} - \hat{x}\hat{y}| + |\hat{x}(\hat{y} - y)| + |y(\hat{x} - x)| \\ &\le 2^{-p}|\hat{x}\hat{y}| + (|x| + \mathsf{e}_x)\mathsf{e}_y + \bar{y}\mathsf{e}_x \\ &\le 2^{-p}|\hat{x}\hat{y}| + \bar{x}\mathsf{e}_y + \bar{y}\mathsf{e}_x + \mathsf{e}_x\mathsf{e}_y \end{aligned}$$

```
template <class NT>
bool geom_series_A(const NT r,
                   const int n){
  NT s=0, p=1;

  for(int i=0;i<n;i++){
    s = s + p;
    p = p * r;
  }

  NT t = ( 1 - p )/( 1 - r );
  return (t==s);
}
```

(A)

```
template <class NT>
bool geom_series_B(const NT r,
                   const int n){
  NT a=0, p=1;

  for(int i=0;i<n;i++){
    a = p + a;     //change here
    p = p * r;
  }

  NT b = ( 1 - p )/( 1 - r );
  return (a==b);
}
```

(B)

FIGURE 3.4. Two C++ functions for the verification of Equation (3.2).

We compute logarithmically, so for each node v, we actually have a variable q_v with $\mathsf{e}_v = 2^{q_v}$. To bound the right hand side by 2^{q_z}, we set q_x and q_y such that

$$\begin{aligned} q_x &\le q_z - 2 - \lceil \log_2 \bar{y} \rceil &=: a \\ q_y &\le q_z - 2 - \lceil \log_2 \bar{x} \rceil &=: b \\ q_x + q_y &\le q_z - 2 &=: c \end{aligned} \tag{3.1}$$

and recompute $\hat{x}$ and $\hat{y}$. Then we set

$$p = \max\{2,\ \lceil \log_2 |\hat{x}| \rceil + \lceil \log_2 |\hat{y}| \rceil + 2 - q_z\}$$

and compute $\hat{z} = \hat{x} \otimes_p \hat{y}$. Then

$$2^{-p} \le \frac{2^{q_z-2}}{|\hat{x}\hat{y}|}, \qquad \mathsf{e}_x \le \frac{2^{q_z-2}}{\bar{y}}, \qquad \mathsf{e}_y \le \frac{2^{q_z-2}}{\bar{x}}, \qquad \mathsf{e}_x \mathsf{e}_y \le 2^{q_z-2}$$

and hence $|\hat{z} - z| \le 2^{q_z}$. As final step, we set $\mathsf{e}_z = 2^{q_z}$. How do we select q_x and q_y, as large as possible, such that Equation (3.1) is satisfied? This is a very tiny integer optimization problem which takes only two combinatorially different configurations, shown in Figure 3.3. We select

$$\begin{aligned} q_x &= a & q_y &= b & &\text{if } \ a + b \le c \\ q_x &= \left\lfloor \frac{c + a - b}{2} \right\rfloor & q_y &= c - q_x & &\text{otherwise.} \end{aligned}$$

The remaining rules for $+, -, /$ and $\sqrt[d]{\ }$ can be found in the *RealAlgebraic* documentation [**89**]. Again, $/$ and $\sqrt[d]{\ }$ require a lower bound on divisor and radicand, respectively.

3.3. A Case Study on Common Subexpressions

In this section we study the behavior of expression dag based number types on expression dags with common sub-expression. If a sub-expression occurs multiple times in an expression, we may represent this sub-expression using a single dag only. This has several advantages, using Equation (2.3) it might for example allow to compute a better separation bound. The main advantage is however, that we need to evaluate the shared sub-expression only once and thus can reduce evaluation cost. Using an example, we show that current evaluation strategies may void this advantage and we propose improved evaluation strategies.

We first encountered the problem, when assembling test cases to check the correctness of the *RealAlgebraic* implementation. One of the test cases is the well known identity

$$\sum_{i=0}^{n-1} r^i = \frac{1-r^n}{1-r} \qquad \text{for } r \neq 1, \tag{3.2}$$

for geometric series. Using an expression dag based number type `NT`, and the code in Figure 3.4, we can verify this identity for any r and n. Here we always use $r = \sqrt{13}$. Note how the code reuses the sub-expression r^{i-1} for the computation of r^i. When running the code in Figure 3.4A for the default variant of *RealAlgebraic* and different values of n on the `descartes` platform from Section 3.4, we get the running times in the first line of the table below. For increasing n, the time increases moderately. If we however replace `(t==s)` in the last line of Figure 3.4A with `(s==t)` and repeat the experiment, we get the running times in the second line. Suddenly the running time explodes with increasing n. A small code change leads to very significant differences in the running time.

	n	128	256	512	1024
deterministic	`t==s`	0.00	0.00	0.00	0.03
RealAlgebraic	`s==t`	0.01	0.06	0.51	6.27

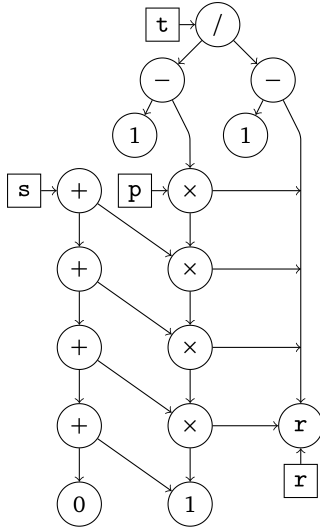
Why does this happen? Bigfloat arithmetic is the main cost factor in expression dag evaluation. Using our framework, it is easy to check, how much bigfloat arithmetic is used by *RealAlgebraic*. The class `Approximation_policy_statistics` counts how often a bigfloat operation of a certain precision is performed. The following code extends the default *RealAlgebraic* type for the collection of statistics.

```
struct Stat_policies :
  public Default_real_algebraic_policies_base<Stat_policies> {

  typedef Default_real_algebraic_policies_base<Stat_policies>   Base;
  typedef typename Base::ApproximationPolicy                     BAP;
  typedef Approximation_policy_statistics<BAP>   ApproximationPolicy;
};

typedef Real_algebraic<Stat_policies> Stat_real_algebraic;
```

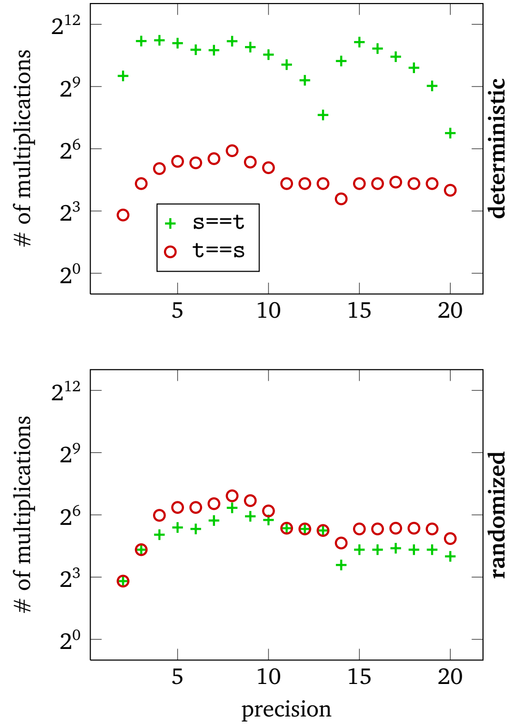
We repeat both experiments for $n = 128$ and check the amount of bigfloat arithmetic used. It turns out, that there is a large difference in the usage of multiplication, while the difference for $+, -, /, \sqrt[2]{\ }$ is negligible. The upper part of Figure 3.5B, labeled **deterministic**, shows how often a bigfloat multiplication of certain precision is performed when running the code in Figure 3.4A and its modification. It is clearly visible, that the problem is not more expensive bigfloat operations. Rather, significantly more operations but of the same precision are performed by the `(s==t)` variant. The y-axis is logarithmic, so the number of bigfloat multiplications roughly squares.

The question remains, why this occurs. Figure 3.5A shows the expression dag generated by the code in Figure 3.4A prior to the last line. The final comparison `(t==s)` creates a new root node with t as left child and s as right child and computes the sign of this node. For `(s==t)`, the same happens, but with s as left child and t as right child. In both cases, the sign of the root node is zero. This means, the sign will not be found by a filter stage, but the node has to be approximated repeatedly, until the separation bound is hit. Using $r = \sqrt{13}$, we also make sure, that no node becomes known exactly at some point. Lets consider a single step of precision driven arithmetic to improve the root node. It is significant, that precision driven arithmetic first recursively improves the left child, then the right child and finally recomputes the node itself, where left and right correspond to the drawing in Figure 3.5A. We already identified the multiplication nodes as central to the problem. Due to the order of traversal, in the case of `(t==s)`, we visit each multiplication node coming from above first. Later we arrive at this node again, coming from the left side. Upon the second arrival, we again request an approximation of certain accuracy from this node. If we are lucky, we need not recompute the approximation. This is what happens in case `(t==s)`. In case `(s==t)` we arrive from the left side first. Later we arrive at a node again, coming from above. This time the present accuracy is insufficient and we must recompute the approximation of this node and much worse, all nodes below. Thus, we have to perform quadratically many bigfloat multiplications, while in case `(t==s)` each node is recomputed only once.

In case `(s==t)`, we approximate each multiplication node several times, with increasing accuracy, until finally reaching the necessary maximal accuracy. This is however not visible in Figure 3.5B, only more operations are performed. This shows



(A) Generated dag for $n = 4$.



(B) Bigfloat multiplications for $n = 128$.

FIGURE 3.5. Dag structure and usage of bigfloat multiplication for *RealAlgebraic* and the code in Figure 3.4A.

that the already available approximation can only be slightly insufficient, making the difference disappear in the coarse statistics.

We can reproduce this behavior for other number types. Sharing a very similar evaluation strategy with *RealAlgebraic*, `leda::real` shows the same behavior on exactly the same example. For `CORE::Expr` 2, we have to make some changes, i.e., swap the operands in the addition. The resulting code is shown in Figure 3.4B. Running times from experiments with `leda::real` and `CORE::Expr` 2 are shown below. Again, a small code change leads to a large change in running time. Although we tried more variations of the code given here, we could not find one which reproduces this behavior for `CORE::Expr` 1. The precision driven arithmetic of `CORE::Expr` 1 performs ring operations exactly and computes the error bound for a node, instead of setting it to the requested value. Therefore, the actual error stored in a node may be slightly smaller than originally requested, which avoids the problem in our example. This is however no general solution, as the accuracy requirements of

parents may differ much more than can be bridged by a slightly better error bound computation.

	n	128	256	512	1024
	`t==s`	0.00	0.01	0.02	0.10
`leda::real`	`s==t`	0.07	0.37	2.87	31.59
	`a==b`	0.00	0.01	0.01	0.04
`CORE::Expr` 2	`b==a`	0.01	0.06	0.62	4.29

In the bad case, the evaluation strategy totally negates the advantage of sharing common sub-expressions. The dag is evaluated as if it were an expression tree, storing copies of an identical sub-expression for each reference to it. Furthermore, any dag sharing common sub-expressions is prone to this problem. If a node is referenced several times, there is a good chance that all parents have different accuracy requirements on this node. If we arrive from a parent with low requirements first, we must recompute the node upon a later arrival. Since the accuracy requirements propagated by precision driven arithmetic are usually tight, this will almost always trigger a recomputation of all descendent nodes. Even if the problem does not appear in cascaded form, we might easily loose a factor of two by going wrong just once near the root node. This happens for example if we run the code in Figure 3.4B with *RealAlgebraic*. Here, the operands for the addition are ordered favorably for *RealAlgebraic*, but the root node `(a==b)` or `(b==a)` decides whether each multiplication will be evaluated once or twice.

It is clearly undesirable, that the running time is this sensitive to small code changes. A user may rewrite a part of her code and end up with significantly worse performance, without any hint to the cause. If we compare two expression dag based number types, one of them might seem inefficient only because expressions are created in a way favoring the other number type. Of course, we would like to attain minimal runtime, independent of the dag structure.

So, how can this problem be avoided? The accuracy propagation scheme in `leda::real` in general requires to recompute the approximation of one child node to compute the accuracy requirement for the other child, cf. Section 2.1 and is therefore part of the problem. In Section 3.2 we lifted this requirement in nearly all cases. The exceptions are the divisor node in a division and the radicand node in a radical, for which a positive lower bound must be known. For these nodes it might be necessary, to compute a lower bound and hence the sign recursively, but only once. As soon as a lower bound is known, we can propagate the accuracy requirements to these nodes and their siblings directly.

The new error propagation scheme allows to recurse to any child first. It is however by no means clear how to choose an order to avoid recomputation for all nodes in the dag globally. We can however choose randomly. If we do this for *RealAlgebraic*, we get the running times shown in the table below. The corresponding usage

of bigfloat arithmetic for $n = 128$ is shown in the lower part of Figure 3.5B, labeled **randomized**. Both the running time and the number of bigfloat multiplications is significantly lower than in the deterministic (`s==t`) case, but still higher than in the deterministic (`t==s`) case.

	n	128	256	512	1024
randomized	`t==s`	0.00	0.00	0.01	0.06
RealAlgebraic	`s==t`	0.00	0.01	0.00	0.05

By randomizing dag traversal, the expected number of node re-evaluations in our example becomes linear. It suffices to consider the dag rooted at s for the analysis. Let $M(i)$ be the number of bigfloat multiplications triggered by the addition node with distance i to the root node. Then

$$M(i) = \begin{cases} i-1 & \text{if we traverse to the right first,} \\ i-1+M(i-1) & \text{otherwise.} \end{cases}$$

We can see by induction that the expected number of multiplications is bounded by $2(i-1)$. On average, we perform at most twice the optimal number of multiplications, which our experiments nicely confirm.

Randomization helps to avoid a traversal order which leads to quadratically many re-evaluations, provided there are only few such orders. In geometric applications, where we compute the sign of expressions with equal dag structure a few hundred times or more, the average running time using randomization will actually be realized. However, the average number of bigfloat operations in a randomized evaluation strategy is still larger than the minimal number and we would like to avoid even that.

The key to improvement is the observation, that the new scheme for precision propagation from Section 3.2 does not need to recompute approximations at all. A node can wait until all its parents have registered their accuracy requirements and all its children have recomputed their approximation. Only then it has to compute a new approximation. To this end, each node v stores a new information $\tilde{\mathtt{e}}_v$, which is the new error bound the approximation of v should satisfy. This allows us to improve the approximation of a node using the following algorithm.

Algorithm 3.1 (topological precision driven arithmetic)**.**
Let u be a dag node and $e > 0$. Then TopPrecDrivArith *computes an approximation $\hat{u}$ such that*

$$|u - \hat{u}| \leq e.$$

1: ***procedure*** TopPrecDrivArith (u, e)
2: $\quad \tilde{\mathtt{e}}_u \leftarrow \min\{e, \tilde{\mathtt{e}}_u\}$
3: $\quad$ *let D be the dag rooted at u*
4: $\quad$ ***for*** *all nodes $v \in D$ in topological order* ***do***
5: $\quad\quad$ ***if*** $\tilde{\mathtt{e}}_v < \mathtt{e}_v$ ***then***

6: *if a lower bound from any child w of v is required **then***
7: *compute sign and lower bound for w recursively*
8: ***for** all children w of v **do***
9: *compute error bound r to request from w*
10: $\tilde{\mathsf{e}}_w \leftarrow \min\{r, \tilde{\mathsf{e}}_w\}$
11: ***for** all nodes $v \in D$ in reverse topological order **do***
12: ***if** $\tilde{\mathsf{e}}_v < \mathsf{e}_v$ **then***
13: *compute necessary precision p and recompute $\hat{v}$*
14: $\mathsf{e}_v \leftarrow \tilde{\mathsf{e}}_v$

We only need to initialize $\tilde{\mathsf{e}}_v$ once, when v is created, since it never becomes larger than e_v. By processing nodes in topological order in the first stage, we ensure that upon arriving at a node v, all its parents have registered their accuracy requirements. At this point we have all the data necessary to compute the requirements from v to its children. In the second stage, processing nodes in reverse topological order ensures that the children of a node have already been re-evaluated with the necessary accuracy. Sorting the nodes of a dag topologically takes linear time in the size of the dag, so there is no asymptotic disadvantage compared to other traversal methods. The algorithm does however need to traverse the dag more often, so the involved constants will be larger.

To avoid the overhead for additional dag traversal, expression evaluation may be implemented in an introspective way [**74**]. Initially, straightforward precision driven evaluation is used, while counting the number of dag node evaluations. If the number of evaluation crosses some threshold, e.g., twice the number of dag nodes, evaluation switches to the new evaluation scheme.

A final advantage of the algorithm is, that it may be parallelized to the amount allowed by the expression dag. Any two nodes not connected by a directed path can be processed in parallel in both stages. This is especially interesting for the second stage, where expensive bigfloat operations are performed.

3.4. Experiments

We study the influence of different models and implementation parameters on the overall performance of *RealAlgebraic* by means of experiments. Our goal is to better understand the characteristics of our algorithms and to determine an efficient default *RealAlgebraic* variant. To this end, we run several geometric algorithms with different requirements on the arithmetic for randomly generated and structured input data.

An alternative to using geometric algorithms is to compare the performance for some geometric predicates or arithmetic expressions only. It is however difficult to generate realistic input for predicates only. For example, matrices filled straightforwardly from pseudorandom numbers exhibit a structure that makes computing the determinant numerically easier than truly random matrices [**35**]. Geometric algorithms usually create a certain distribution of predicate calls, which may be

hard to simulate or even understand. For example, a good algorithm computing the Delaunay triangulation of a set of points will almost exclusively perform incircle tests for points which are close to each other in the input set. Using geometric algorithms as test bed produces more meaningful results about the performance of different number types.

By a similar argument, one should favor input data from real applications over artificial data. In that regard, the experiments by Held and Mann [**43**] are exemplary, which test their algorithms on a set of approximately 20,000 data sets which were collected over time. Real world data is however notoriously hard to obtain. Even if one can run experiments with some real world data sets, it is unclear whether the results carry over to other data sets. To achieve representativeness, one needs input data from many different sources. We use artificially generated data, since we can easily create many data sets with different characteristics. This allows us to observe the performance of number types for both simple and more challenging input sets.

3.4.1. Experimental Setup. For our experiments we use algorithms from CGAL, solving four of the geometric problems introduced in Section 1.1. In CGAL, the three layers of geometric algorithm, geometric primitives and arithmetic are nicely separated. This allows us to run the same algorithm with different geometric primitives and with different number types, too.

Delaunay triangulation: We compute the Delaunay triangulation of points in the plane, using `Delaunay_triangulation_2` and the `Simple_cartesian` kernel. All predicates involve polynomial expressions only.

Segment intersection: We compute all intersection points among a set of line segments in the plane using `compute_intersection_points()` and both the `Simple_homogeneous` and `Simple_cartesian` kernel. In the homogeneous kernel, geometric primitives involve polynomial expressions only. The degree of these expressions is generally larger than of those needed for computing the 2D Delaunay triangulation. With the Cartesian kernel, geometric primitives involve rational expressions.

Arrangement of circles: We compute the arrangement of circles and line segments using `Arrangement_2` and the `Simple_cartesian` kernel. This problem involves algebraic numbers and geometric primitives which are not provided by the standard kernels in CGAL. Additional geometric primitives are provided to the arrangement implementation by a so-called *traits class*. We perform experiments for two different traits classes. The first is `Arr_circle_segment_traits_2`, it handles algebraic numbers by means of static algebraic predicates. Each predicate is reduced to the evaluation of several rational expressions. The second traits class we use is `Arr_circular_line_arc_traits_2`. For number types with field operations only, it provides static algebraic predicates, but if the number type supports square roots, straightforward expressions are used for predicate evaluation.

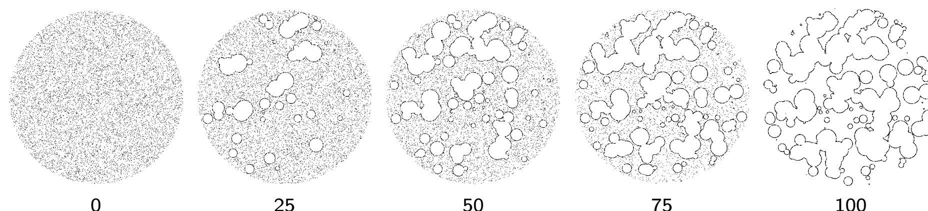


FIGURE 3.6. Input data for Delaunay triangulation.

Segment Voronoi Diagram: We compute the Voronoi diagram of line segments using `Segment_Delaunay_graph_hierarchy_2`. Again, the necessary geometric primitives are provided by a traits class. We use `Segment_Delaunay_graph_filtered_traits_without_intersections_2` when segments do not intersect in their interior, and `Segment_Delaunay_graph_filtered_traits_2` otherwise. Both provide static algebraic predicates for number types without square root and straightforward predicate implementations for number types with square root. If segments do not intersect in their interior, static algebraic predicates employ ring operations only, the straightforward predicates however always need a division operation. Both traits classes have a built-in dynamic floating-point filter. The geometric primitives for the Voronoi diagram of segments are numerically harder than those for the arrangement of circles, since they involve algebraic numbers of higher degree.

If we count using different kernels or traits classes, this gives us seven different geometric algorithms solving four geometric problems. Each of these algorithms has different requirements on the arithmetic. Two use ring operations, and two use field operations only. Three algorithms use field operations plus square root, but may be used in combination with more restricted number types, by means of static algebraic predicates.

INPUT DATA. Most input data sets are randomly generated. In some of those, input objects are uniformly distributed. This usually results in predicate calls which are numerically easy, i.e., may be decided by a floating-point filter. In other data sets, some structure is imposed to enforce a larger amount of degenerate or nearly degenerate predicate evaluations. Some data sets are completely structured without a random component.

We generate floating-point input data with 53 bits of precision, and, since small precision integer coordinates are common in real world applications, we also generate integer input data with 25 bits of precision. To this end, we first generate a data set with floating-point coordinates, then scale the data such that its bounding box coincides with $[-2^{24}, 2^{24}]^2$. Finally we round all numerical data to integers. Small precision integer data is more easy to handle numerically. For example, the straightforward floating-point implementation of the 2D orientation predicate, given

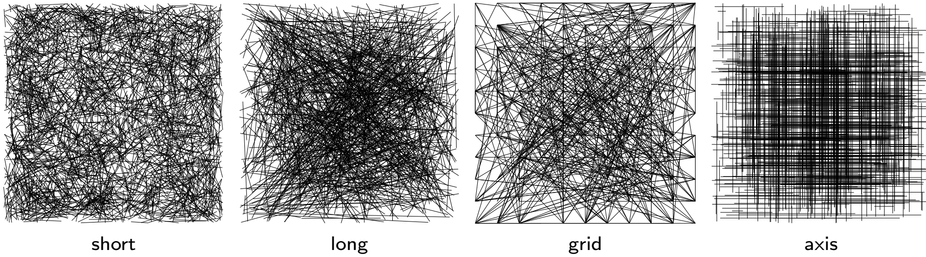


FIGURE 3.7. Input data for segment intersection.

in Equation (1.7), always computes the correct result for 25 bit integer input. We generate the following data sets.

Delaunay triangulation: For $p \in \{0, 25, 50, 75, 100\}$, we generate input data sets with 10 000 points each, of which p% are located almost on the boundary of a union of disks. The remaining points are placed uniformly at random outside the union of disk, no points are placed in the interior. With increasing p, this forces the Delaunay triangulation algorithm to perform more nearly degenerate incircle tests. Input data sets are shown in Figure 3.6.

Segment intersection: We generate four types of inputs sets, shown in Figure 3.7.

short: Sets of 2500 segments with endpoints uniformly distributed in a square. The length of segments is restricted to approximately one third of the side of the square.

long: Sets of 700 segments with endpoints uniformly distributed in a square.

grid: Sets of 500 segments with endpoints randomly placed on an 11×11 grid. Some intersection points will have the same x-coordinate, which is a degeneracy for the plane sweep algorithm that we use.

axis: Sets of 700 axis parallel segments. In this type of set, there are many degeneracies too, however all intersection points have a representation in input precision.

Arrangement of circles: We generate four types of data sets, shown in Figure 3.8.

rand: Sets of 500 line segments and 500 circles, randomly generated inside a square. Segment length and circle diameter are restricted to approximately one fourth of the side of the square.

gridrn: A set of 529 circles with centers on a 23×23 grid and radius of approximately twice the diagonal of the grid. There are many occurrences of four circles almost intersecting in a single point. The data set is slightly rotated, such that the intersection points are not all near the same x-coordinates. Due to numerical imprecision, rotating the data set also perturbs the input data and thus further relaxes the near degeneracies.

pack: An approximate circle packing of 2000 circles. We generate data sets incrementally using the Voronoi diagram of disks. After a small initial set of disks,

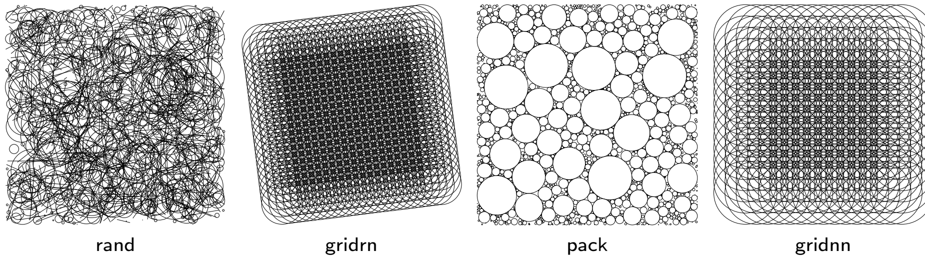


FIGURE 3.8. Input data for arrangements of circles and line segments.

we create alternately disks with random centers, touching at least one other disk and disks with their center on random Voronoi vertices, touching at least three other disks. Since we round data to floating-point numbers, circles in the final data set are unlikely to touch but will either miss or intersect slightly.

gridnn: A set of 225 circles with centers on a 15×15 grid and radius of approximately twice the diagonal of the grid. This data set is not rotated and therefore closer to being degenerate than the gridrn set. Furthermore, intersection points all occur near the same x-coordinate, which is another degeneracy for the plane sweep algorithm we use.

Segment Voronoi diagram: We generate four types of data sets, which are shown in Figure 3.9. In the first two, segments only intersect at endpoints, in the other two, true intersections occur.

mst: Sets of 5000 segments, which form the minimum spanning tree of a point set with 75% of points on the boundary of a union of disks. Most segments in this data set are very short.

sqrs: Sets of 785 axis parallel segments, which are edges of small squares, regularly placed on a 15×15 grid.

short: Sets of 500 segments with endpoints uniformly distributed in a square. The length of segments is restricted to approximately one third of the side of the square.

shoax: Sets of 200 short, axis parallel segments inside a square. The length of segments is restricted to approximately one third of the side of the square.

PLATFORMS. Comparing algorithms by measured running time is a tricky business. Apart from input data and in our case the geometric algorithm, also the compiler, third party libraries, and hardware have an influence on the running time. In general, small performance differences between algorithms may vanish or revert when the computing environment changes. To give our results more significance, we run experiments on three different platforms.

`descartes`: A desktop PC with an Intel Core i5-660 processor running at 3.33 GHz. All software is compiled using g++ 4.6.3.

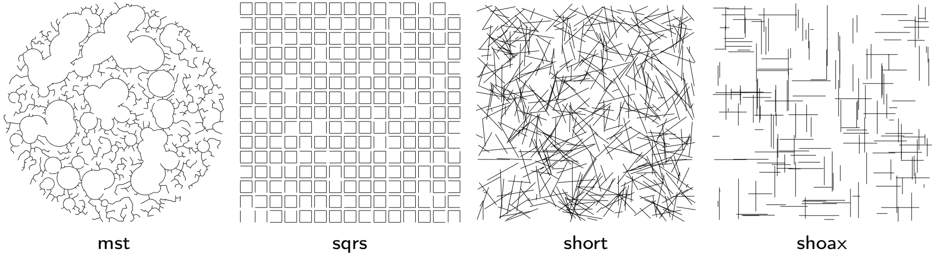


FIGURE 3.9. Input data for segment Voronoi diagram.

minkowski: A notebook with an Intel Core 2 Duo T5500 processor running at 1.66 GHz. All software is compiled using g++ 4.5.1.

thales: A Sun workstation with a SPARC-T3 processor running at 1.66 GHz. This platform comes with a hardware fused-multiply-add instruction, which we exploit in algorithms based on error-free transformations. All software is compiled using g++ 4.7.1. Due to compiler incompatibilities, we are unable to use a precompiled LEDA package on thales. The implementation of interval arithmetic in LEDA is however almost entirely based on header files, therefore Leda_interval_filter_policy is still available.

On all platforms we use the following library versions: GMP 5.0.5, MPFR 3.1.1, BOOST 1.50, CORE 2.1.1, CGAL 4.0.2, CORE 1.8 as shipped with CGAL, and LEDA 6.4. With the exception of LEDA, which comes in a precompiled package, we build all libraries and all experiments in release mode and optimization level -O3.

3.4.2. Results. To avoid combinatorial blowup, we do not perform experiments for all possible combinations of policies. Instead, we use Default_real_algebraic as baseline variant and only exchange one or two parameters at a time. Using the acronyms from Section 3.1, we label a number type with the parameters in which it differs from the baseline variant.

For each type of input set, we generate 25 sets and for each number type run the corresponding algorithm on all of them. We do this also in the two cases, where all 25 sets are equal. We measure and report the average running time. The complete set of results can be found in Appendix A. Here we show barcharts for a few selected data sets and algorithms only. Those are usually selected to show the general behavior, but often also include interesting exceptional cases.

MEMORY ALLOCATION. Exemplary results are shown in Figure 3.10. In most cases, using a memory pool instead of the default allocator improves the performance. The improvement is largest for computing the Delaunay triangulation and much lesser for the remaining algorithms. There are some cases, e.g., the degenerate gridrn dataset, where the default allocator gives better performance on both minkowski and descartes.

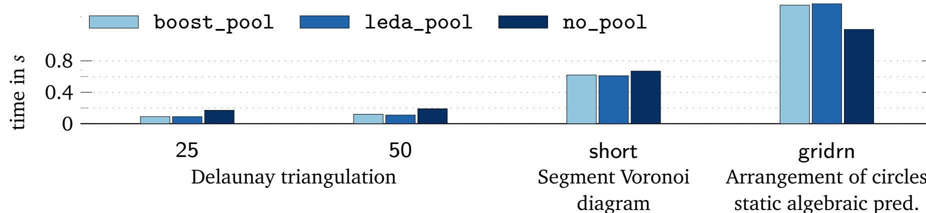


FIGURE 3.10. Effect of different strategies for dag node memory allocation on the running time. Floating-point data on `descartes`.

In general, the pool implementations from BOOST and LEDA perform about equally good. Results are comparable for integer and floating-point data as well as for different platforms, though with a larger advantage for memory pools on `thales`. We suggest to always use a memory pool for dag nodes, since the possible gain in performance is larger than the possible loss.

Our observation that the choice of memory allocation has a larger impact for geometric problems with simpler predicates is no surprise. With rising complexity, it is more likely that the floating-point filter fails, simultaneously predicates are more expensive to evaluate using software arithmetic. Hence a smaller fraction of the total running time is spend with memory allocation in the first place, leaving less room for improvement in total. Amdahl's law [**2**] relates speedup in a part of a program to total speedup of the program. It shows that total speedup is inherently limited unless the part that is improved takes a significant portion of the overall running time. Consider a program run, where the part to be optimized takes a fraction f of the running time. If this part runs a factor of s faster, the total speedup S_T of the program is given by

$$S_T = \frac{1}{(1-f)+f/s}.$$

No matter how large s is, the total speedup is limited to $1/(1-f)$. Conversely this means, that for computing a Delaunay triangulation about half of the running time is spend with memory allocation for dag nodes if the default allocator is used. Amdahl's law gives good reason to concentrate optimization efforts on those parts of a program which account for most of the running time. Which parts of *RealAlgebraic* take up most of the runtime depends however on the geometric problem and input data. We can see effects that relate to Amdahl's law in many of our experiments.

FilterPolicy. Between the two *FilterPolicy* models `boostF` and `ledaF` there is a time versus accuracy tradeoff. `boostF` computes tighter intervals, while `ledaF` is faster. *FilterPolicy* models are used for two different purposes in our implementation. One is as first evaluation stage in expression dag evaluation, the other is to check if operations can be performed locally, to defer dag creation. We therefore compare

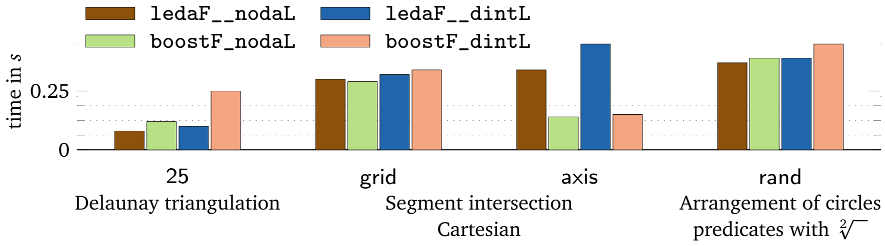


FIGURE 3.11. Effect of different *FilterPolicy* models on the running time. Floating-point data on `descartes`.

them in combination with two *LocalPolicy* models: `nodaL`, where only the first use case occurs, and `dintL`, where both use-cases may occur.

Selected results are shown in Figure 3.11. We get similar results in both use-cases, as well as for integer and floating-point data. With respect to algorithms and data sets, `ledaF` is the better choice for Delaunay triangulation and non-degenerate data sets in segment intersection, as well as for arrangement of circles with square root. `boostF` is the better choice for degenerate data sets in segment intersection and for arrangement of circles with static algebraic predicates. For the Voronoi diagram of segments, the winner depends on the platform. On `minkowski`, `ledaF` is the better variant, while on `descartes` and `thales`, `boostF` has a small advantage. This platform dependency is also visible for the other algorithms. While `ledaF` is clearly the overall best choice on `minkowski`, on the other platforms this is less clear.

We check the influence, the *FilterPolicy* model has on the usage of bigfloat arithmetic, a major cost factor in *RealAlgebraic*. Figure 3.12 shows bigfloat usage statistics for both filters in combination with `nodaL` and the problems selected for Figure 3.11. Note that both axes are logarithmic.

For Delaunay triangulation and the 25 data set, `boostF` reduces bigfloat usage more than `ledaF`, but not enough to achieve a running time advantage. Both filters are nearly equally effective, so the lower runtime cost of `ledaF` pays off. For Cartesian segment intersection and the `grid` data set, the differences between both filters are more significant and for the `axis` data set, `boostF` eliminates bigfloat operations completely. For more degenerate data sets, the better accuracy of `boostF` eliminates sufficiently more bigfloat operations to also gain a running time advantage. Hence it is the better choice for these data sets. For arrangements of circles based on predicates with square root and the `rand` data set, a surprising effect takes place. Here, `ledaF` eliminates bigfloat operations of low precision, while the number of operation with high precision is unchanged in comparison to `boostF`. Evidently, there are sub-expressions which `ledaF` can evaluate exactly, but `boostF` can not.

With only a few exceptions, the possible performance loss from choosing `boostF` over `ledaF` is larger then the possible performance loss from choosing `ledaF` over

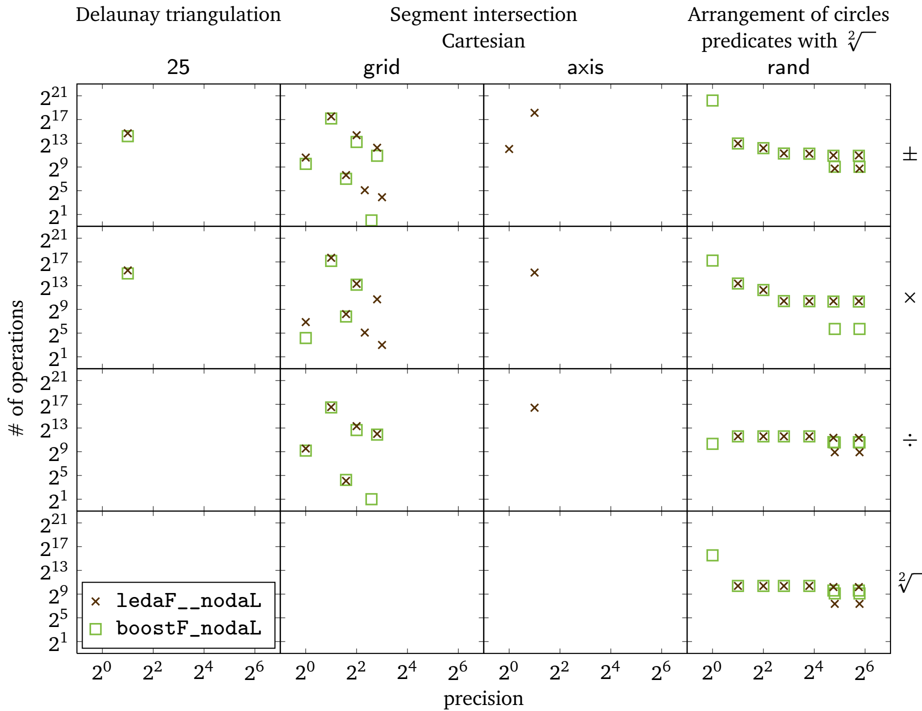


FIGURE 3.12. Usage of bigfloat arithmetic with different *FilterPolicy* models. Selected problems correspond to Figure 3.11.

`boostF`. Thus, we suggest to use `ledaF` or a dynamic floating-point filter based on similar techniques, unless one knows explicitly that data sets contain many degeneracies.

ApproximationPolicy. There are two *ApproximationPolicy* models, `mpfrA` and `ledaA`. To the set of competitors we add `pure_mpfrA`, which is a variant of `mpfrA` purely based on MPFR, i.e., one that does not employ the improved floating-point number to MPFR conversion. Exemplary results are shown in Figure 3.13.

Results are consistent for all platforms, algorithms and input sets. Not surprisingly, `mpfrA` is a large improvement over `ledaA`. The few cases where both perform equal, are those cases where no bigfloat arithmetic is used at all. Furthermore, `mpfrA` has a small but consistent advantage over `pure_mpfrA`. The advantage is larger for floating-point data and input sets with a larger amount of degeneracies, where more conversions occur. Regarding Amdahl's Law, it is surprising that something as fast as this number type conversion can have a visible effect. Based on our results, we suggest to use `mpfrA` in all cases.

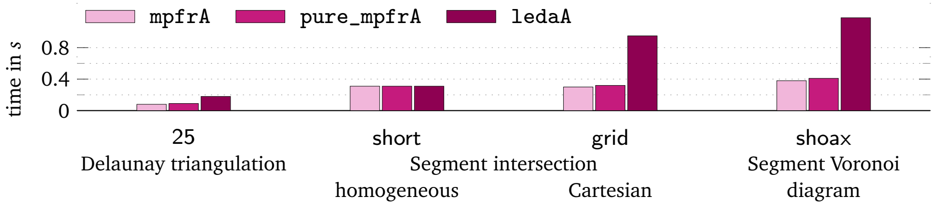


FIGURE 3.13. Effect of different *ApproximationPolicy* models on the running time. Floating-point data on `descartes`.

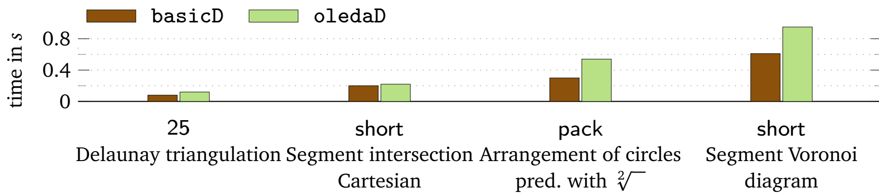


FIGURE 3.14. Effect of different *ExpressionDagPolicy* models on the running time. Floating-point data on `descartes`.

ExpressionDagPolicy. Selected results are shown in Figure 3.14, with corresponding bigfloat usage statistics in Figure 3.15. Of our two *ExpressionDagPolicy* models, `basicD` is consistently better than `oledaD` on all platforms, algorithms and input sets. It is interesting to see, that `basicD` is an improvement even for those problems which are solved exclusively without bigfloat arithmetic, e.g., Cartesian segment intersection on short data sets. The improvement in these cases originates in the reorganization of node data only! Reducing the node size does indeed improve cache performance.

In general, both *ExpressionDagPolicy* models utilize about the same amount of bigfloat arithmetic. The few small deviations, for example for multiplications on the pack dataset in Figure 3.15 are usually dominated by other more frequent or more expensive operations where the usage is equal for both *ExpressionDagPolicy* models. There is however one exception from this rule. With `oledaD`, by far the most operations are bigfloat additions of small precision. Due to the new implementation of precision propagation, `basicD` significantly reduces their number without increasing the number or precision of other bigfloat operations.

LocalPolicy. Selected results are shown in Figure 3.16. We first observe that `dwicL` and `dintL` nearly always perform worse than `doubL`. The reason is, that `doubL` performs an operation locally if and only if the corresponding floating-point operation is exact, while the interval representation of `dwicL` and `dintL` may let some exact operations go by undetected. Furthermore, error-free transformations are simply faster than interval operations.

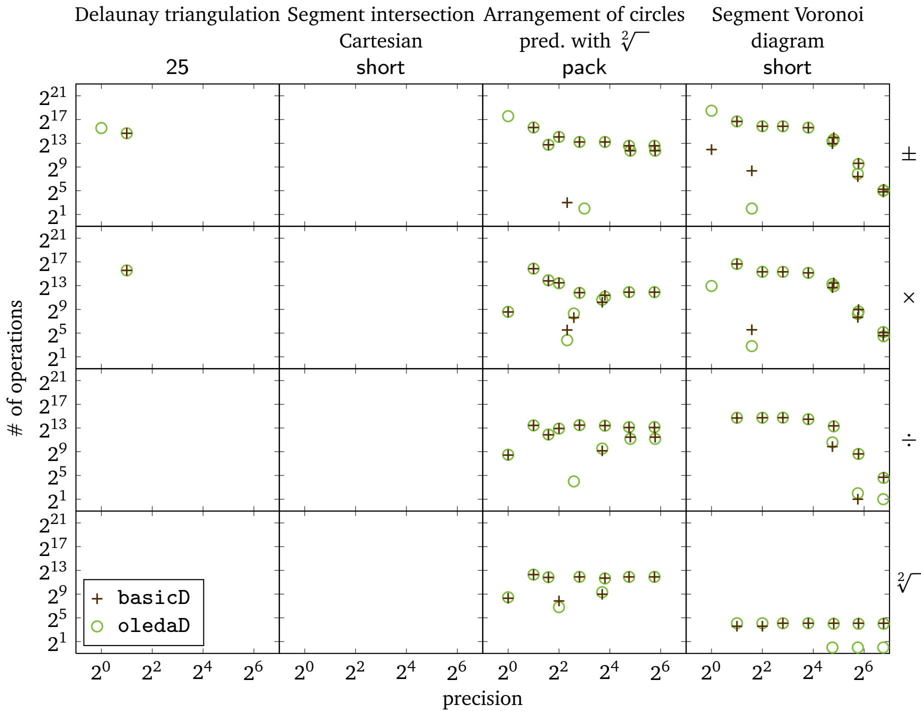


FIGURE 3.15. Usage of bigfloat arithmetic with different *ExpressionDagPolicy* models. Selected problems correspond to Figure 3.14.

Among the remaining three models, nodaL, doubL, and dsumL, there is unfortunately no unique best variant. Results are similar on all platforms but differ for integer and floating-point data. For most data sets with floating-point coordinates, dsumL is the best choice, while for most data sets with integer coordinates, doubL is the best choice. However, regardless of integer or floating-point input data, nodaL remains the best variant for data sets with few degeneracies. Following the basic assumption that degenerate cases are rare and the principle to choose a simple approach when in doubt, we suggest to use nodaL as base variant, but to consider as alternatives doubL for small precision integer data and dsumL for floating-point data, especially in case of degenerate input data.

How successful are different *LocalPolicy* models in reducing the number of bigfloat operations? Figure 3.17 shows that mostly operations of low precision are avoided. The reason is that these occur in the small subtrees whose creation may be avoided by *LocalPolicy* models. For floating-point data dsumL is much more effective in reducing bigfloat operations than doubL, which explains why it performs better there. On integer data however, the difference is smaller. Here the advantages of doubL, i.e.,

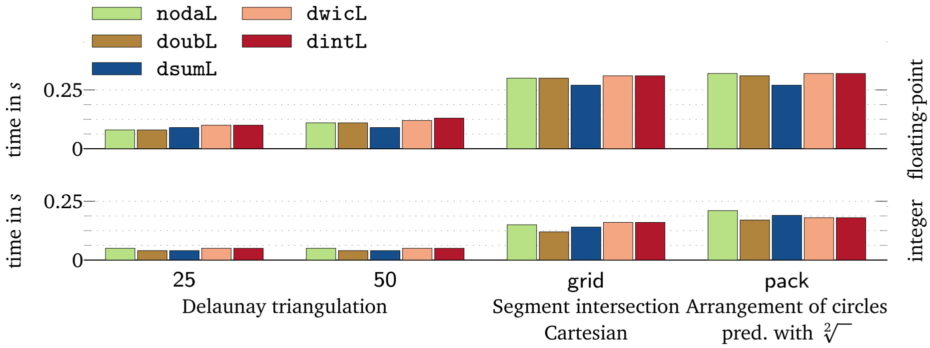


FIGURE 3.16. Effect of different *LocalPolicy* models on the running time.

faster computation of local operations and smaller memory footprint make it the better choice. Interestingly, for integer pack data sets, the number of high precision operations is affected, too, while the number of medium precision operations is unchanged.

GENERAL OBSERVATIONS. In general, the selection of a best model for each concept is independent of the platform, although some run time differences are more significant on `thales`. With the exception of *LocalPolicy*, there is also no difference between integer and floating-point data. The chosen models are combined into the *RealAlgebraic* variant `Default_real_algebraic`. No experiments were done comparing different separation bounds, since we have strong theoretical reason to use the `bfmssB` bound for all computations.

Not surprisingly, solving a certain problem on integer data is faster than solving the same problem on floating-point data. But our experiments also allow us to check whether it is more efficient to use homogeneous or Cartesian coordinate representation for computing intersections of segments and whether to use static algebraic predicates or predicates with square root for computing an arrangement of circles. Slightly generalized, the question is whether it is better to use straightforward predicate implementations, employing more involved arithmetic operations, or use more complicated predicate implementations based on simpler arithmetic operations. Restricted to *RealAlgebraic*, the answer is to use straightforward predicates with more involved arithmetic operations.

For those concepts, where the choice of model appears to be sensitive to the amount of degeneracy in the data set, static algebraic predicates appear to prefer models which were better suited for data with more degeneracies. For example, predicates with square roots always prefer the `ledaF` *FilterPolicy* model, while static algebraic predicates nearly always prefer `boostF`. Still, predicates with square root using `ledaF` are nearly always faster than static algebraic predicates with `boostF`. For *LocalPolicy* models a similar effect can be seen, although not quite as strong.

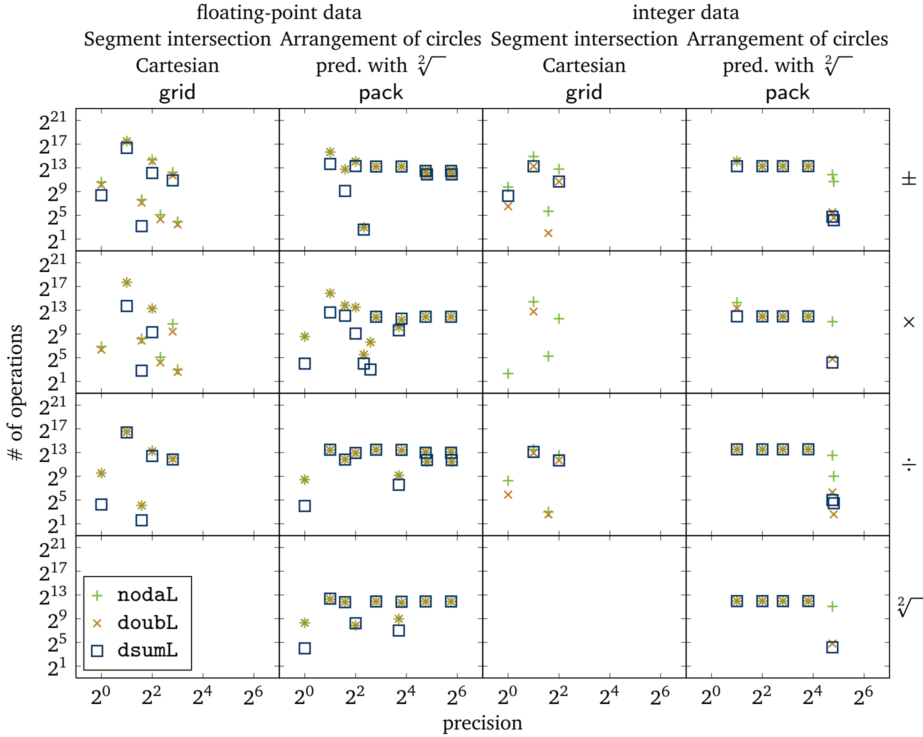


FIGURE 3.17. Usage of bigfloat arithmetic with different *LocalPolicy* models. Selected problems correspond to Figure 3.16.

RealAlgebraic is designed on the assumption, that degenerate and nearly degenerate predicate calls and expressions are rare. Our results suggest that static algebraic predicates transform non-degenerate, easy to evaluate expressions into harder, and closer to degenerate expressions. This would explain why *RealAlgebraic* performs worse on static algebraic predicates than predicates with square root.

3.4.3. Comparison to other Exact Geometric Computation Solutions. Using the same experimental setup as above, we compare three *RealAlgebraic* variants to other exact number types and to Exact Geometric Computation approaches on the level of geometric primitives. We use the `nodaL`, `doubL`, and `dsumL` variants, since the *LocalPolicy* model is the most undetermined choice in our experiments. Note that `nodaL` is equal to the default *RealAlgebraic* variant, while `doubL` and `dsumL` only differ in the *LocalPolicy* model from it. Competing number types are `leda::real`, `CORE::Expr 1`, and `CORE::Expr 2` as well as the following:

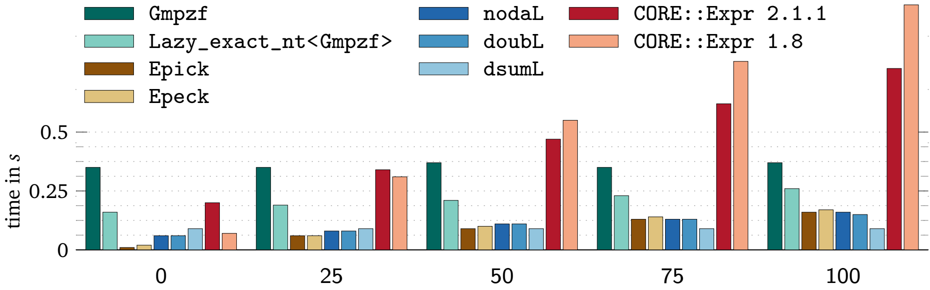


FIGURE 3.18. Delaunay triangulation, floating-point data sets on `descartes`.

`Gmpzf` and `Lazy_exact_nt<Gmpzf>`: `Gmpzf` is a number type implementing the ring $\mathscr{F}$ of bigfloat numbers. It is based on MPFR and part of CGAL.

`Gmpq` and `Lazy_exact_nt<Gmpq>`: `Gmpq` is a rational number type, providing exact field operations. Based on GMP and part of CGAL.

`EXT::real`: A variant of `leda::real`, build from source code available on the Internet [**9, 101**]. While we compile this number type ourselves, it still depends on precompiled bigfloat arithmetic and other code from LEDA.

Where possible, we also run our experiments using the following kernels from CGAL.

`Exact_predicates_inexact_constructions_kernel` (`Epick`): implements a three stage evaluation scheme for geometric predicates. The first stage is a semi static floating-point filter, followed by a dynamic filter. The final exact evaluation is done using `Gmpq`. As the name suggests, this kernel does not support geometric constructions, e.g., computing the intersection point of two segments. Arguments to predicates must be input numbers.

`Exact_predicates_exact_constructions_kernel` (`Epeck`): implements the expression dag technique on the level of geometric primitives, i.e., dag nodes correspond to geometric constructions or predicates [**32, 83**]. The first evaluation is done with a dynamic floating-point filter, an exact evaluation with `Gmpq` follows if necessary. A few selected predicates, e.g., the 2D and 3D orientation and insphere predicates additionally use a semi static filter.

The dynamic floating-point filter in CGAL is based on the same techniques as `boostF`. While we favor a different technique for *RealAlgebraic*, the slightly increased runtime and precision are well invested here, since the dynamic filter is usually prepended by a semi static filter of lower precision and runtime.

Selected results for floating-point data sets on `descartes` are shown in figures 3.18 – 3.21, complete results are given in Appendix A.

DELAUNAY TRIANGULATION. With the exception of `CORE::Expr` 1 on the 0 data set, all three *RealAlgebraic* variants are faster than any other number type, often by a factor of two or more. On integer data sets, both the `Epick` and `Epeck` kernel

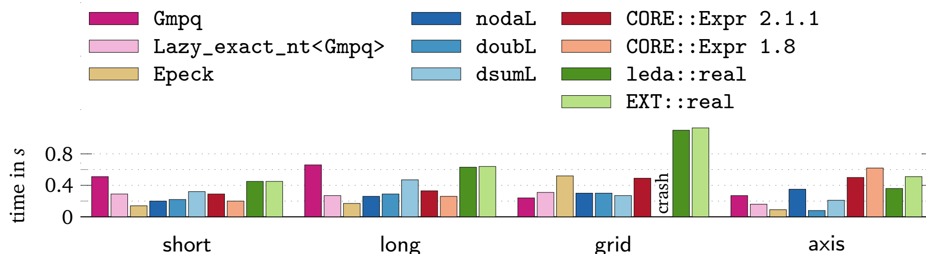


FIGURE 3.19. Segment intersection with Cartesian coordinate representation, floating-point data sets on descartes.

are far superior to *RealAlgebraic*, the Epick kernel by a factor of five to ten. On floating-point data, however dsumL is slightly more efficient on data sets containing many degeneracies, starting with the 50 data set. Due to the special construction of data sets, one can nicely see the adaptivity or non-adaptivity of the different approaches in Figure 3.18. dsumL behaves non-adaptive but is still quite fast and therefore a good choice for data sets with many degeneracies.

SEGMENT INTERSECTION. Our observation for *RealAlgebraic*, that Cartesian coordinate representation and usage of division is superior to homogeneous representation and avoidance of division, carries over to other number types. In particular, Gmpzf and Lazy_exact_nt<Gmpzf> with homogeneous coordinates are slower than Gmpq and respectively Lazy_exact_nt<Gmpq> with Cartesian coordinates, the grid data set with floating-point coordinates being a slight exception. We thus look at the results for Cartesian representation only.

There is no clear winner among the competing number types. For data sets short and long, nodaL and CORE::Expr 1 show the best performance. For the grid data set, the best choice is either Gmpq, doubL, or dsumL, depending on data precision and platform and for the axis data set, doubL is clearly the best. Of all number types, only doubL and Lazy_exact_nt<Gmpq> are within a factor of two of the best result for all data types and platforms. Of these two, doubL is usually the better one. If we ignore the axis data set, the other two *RealAlgebraic* variants are within a factor of two as well and nodaL is the overall best number type.

The Epeck kernel is usually faster than any number type for short, long and axis data sets, but significantly slower than some number types on the grid data set with floating-point coordinates. In any case, doubL is at most a factor of two slower than Epeck, and ignoring the axis data set, nodaL is at most 50% slower than Epeck.

ARRANGEMENTS OF CIRCLES. For arrangements of circles, our observation that it is better to use an available square root operation and to avoid static algebraic predicates does not carry over to other number types having an exact square root. Switching the traits class and hence the predicate type can lead to a speedup by a factor of two for one data set while leading to a simultaneous slowdown by the same

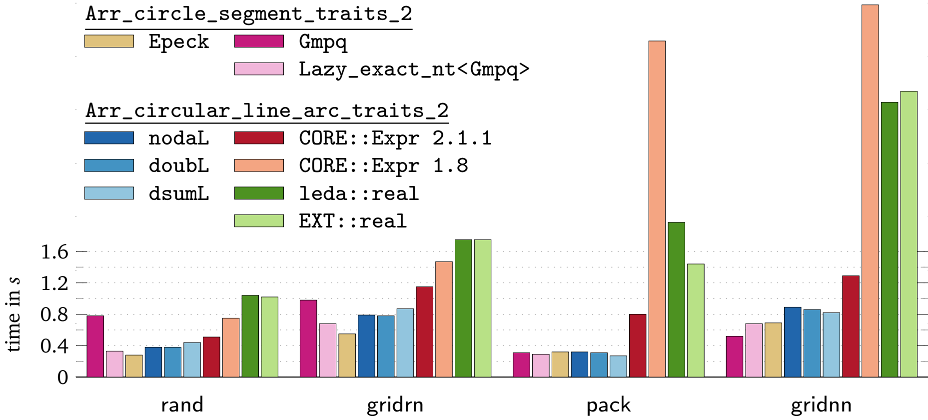


FIGURE 3.20. Arrangement of circles, floating-point data on `descartes`.

factor on another data set. But regardless of the type of predicates, all number types with square root are dominated by any of the three *RealAlgebraic* variants on all data sets and platforms.

For number types without square root or in combination with the `Epeck` kernel, both traits classes provide static algebraic predicates. Running times are generally close when they are instantiated with the same number type, though in most cases `Arr_circle_segment_traits_2` is the slightly better choice. With this traits class, `Gmpq` is the best number type for the highly degenerate gridnn data set but `Lazy_exact_nt<Gmpq>` is faster on all other data sets. The `Epeck` kernel is faster than number type solutions for rand and gridrn data sets on `descartes` and `minkowski` only.

In contrast to other number types, *RealAlgebraic* variants work better with the `Arr_circular_line_arc_traits_2` traits class. For the rand, gridrn, and pack data sets, all three variants are usually within 50% of the globally best variant including both traits classes and often better. In a few cases, `doubL` or `dsumL` are the best choice for integer or floating-point data, respectively. Only on the highly degenerate gridnn data set, the slowdown for *RealAlgebraic* variants compared to the best variant reaches a factor of two.

SEGMENT VORONOI DIAGRAM. For the mst data set, `Gmpzf` is usually the best variant, but all approaches using static algebraic predicates perform nearly equal. The number types using a square root operation are somewhat slower and nearly equal to each other, too. Inspection of statistics on bigfloat usage reveals, that relatively few bigfloat operations of low precision are performed for mst, i.e., most predicates are decided at an earlier evaluation stage. Hence the results only reflect the performance of the floating-point filter in the two types of predicate implementations.

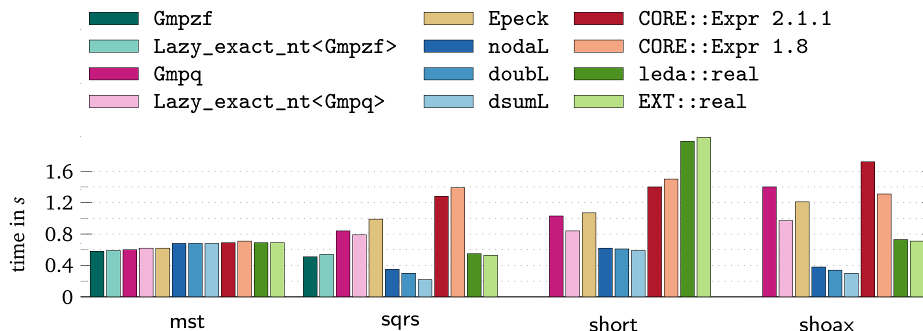


FIGURE 3.21. Segment Voronoi diagram, floating-point data on `descartes`.

For the other data sets sqrs, short, and shoax, `dsumL` is clearly the best approach, and with a few exceptions all three *RealAlgebraic* variants are better than any other number type or kernel. The differences here reflect the performance on degenerate and nearly degenerate predicate calls only, since the non-degenerate cases are again solved by the floating-point filter embodied in the traits class.

CONCLUSIONS. The tested *RealAlgebraic* variants, including the default variant `nodaL`, are in many cases more efficient than other exact number types. The reason for this is the combination of adaptive evaluation, memory manager, using efficient bigfloat arithmetic as backbone and an efficient representation of high precision interval approximations. Every other number type in the field of competitors lacks at least one of these features.

Number types without adaptive evaluation, i.e., `Gmpzf` and `Gmpq` handle data sets with many degeneracies well, but are inferior on non-degenerate input. The memory pool for dag nodes gives *RealAlgebraic* an advantage for all data sets free from any degeneracies. Only `CORE::Expr` 1 shows similar performance here, although `leda::real` and `EXT::real` use custom memory management, too. Adding memory management to `Lazy_exact_nt` should improve its performance significantly for non-degenerate input. On other input sets, the speed of the arbitrary precision arithmetic is decisive. Here, `Gmpzf` and `Gmpq` and their lazy variants give strong results and are therefore strong competitors to *RealAlgebraic*.

`CORE::Expr` 2 is the only number type similar in functionality to *RealAlgebraic* and using state of the art bigfloat arithmetic. `leda::real`, `EXT::real`, and `CORE::Expr` 1 all suffer from using outdated implementations. But `CORE::Expr` 2 is nearly always significantly slower than *RealAlgebraic* variants, and in many cases it is even slower than its ancestor `CORE::Expr` 1. For input sets without degeneracies, this can be attributed to the omission of memory management, but for input sets with many degeneracies a different reason must hold. We suspect, the issue is in the representation of high precision approximations. `CORE::Expr` 2 uses an interval

representation by endpoints. For high accuracy approximations, both endpoints carry nearly the same information, i.e., they are equal except for the last few bits. Much worse, this representation in general enforces two high precision operations to recompute an interval, whereas a midpoint and radius representation needs only one high precision operations plus a few operations with small precision.

The `Epick` and `Epeck` kernels are superior to number-type based solutions for the polynomial and rational problems they are designed for, although the *RealAlgebraic* variants `dsumL` and `doubL` lead to better performance on a few input sets with many degeneracies. The implementor of geometric algorithms should use a kernel based solution if possible. Not only will he save the implementation of geometric primitives, he will also gain much in terms of performance. Here, number type based solutions are only the second option.

In the realm of geometric computations involving algebraic numbers of small degree, static algebraic predicates are the strongest competitors to expression dag based number types supporting algebraic numbers. Static algebraic predicates were shown to have better performance for computing arrangements of circles [**22, 26**] and comparable performance for computing the Voronoi diagram of segments [**50**]. Owing to the substantial improvements in *RealAlgebraic*, expression dag based number types have caught up. Our experiments show, that *RealAlgebraic* variants are competitive to static algebraic predicates for arrangements of circles and clearly superior for computing Voronoi diagrams of segments. Our performance improvements also narrow the large performance gap reported by Held and Mann [**43**], between the Exact Geometric Computation approach and the topology oriented approach for Voronoi diagrams of segments.

CHAPTER 4

New and Improved Exact Floating-Point Algorithms

In this chapter we present some new and improved algorithms that are based on error-free transformation and work with sums of floating-point numbers. In the first part we present two algorithms for computing the sign of the sum of floating-point numbers exactly. We show that predicates based on error-free transformations and exact sign of sum algorithms can be more efficient than predicates based on software number types.

In the second part we present new algorithms for converting floating-point expansions and sums of floating-point numbers exactly and efficiently into bigfloat numbers. We present experiments showing the advantage of our algorithm over straightforward approaches.

4.1. Exact Sign of Sum Computation

4.1.1. Improvements on ESSA. Helmut Ratschek and Jon Rokne [**88**] present an algorithm, called ESSA for "exact sign of sum algorithm", to compute the sign of a sum of floating-point values exactly. They advocate the use of their algorithm in computational geometry, see [**87**] for an overview.

Let two sequences of positive floating-point numbers $a = a_1, \ldots, a_m$ and $b = b_1, \ldots, b_n$ be given. ESSA allows to compute the sign of

$$S = \sum_{i=1}^{m} a_i - \sum_{j=1}^{n} b_j.$$

We call a the positive summands and b the negative summands. We denote with $l = m + n$ the total number of summands. At any time, l is bounded by $2l\varepsilon_{\mathtt{m}} \leq 1$. Let us further assume, that at any time a_1 is the largest positive summand and b_1 the largest negative summand. To allow efficient access to a_1 and b_1, we maintain a and b as heap. Using an error-free transformation, ESSA iteratively computes two new summands x and y with $a_1 - b_1 = x + y$, but $|x| + |y| < a_1 + b_1$. The old summands a_1 and b_1 are removed from a and b and $|x|$ and $|y|$ are inserted into a and b, according to the sign of x and y respectively. This step is iterated until the sum vanishes, or a termination criterion shows that the positive summands dominate the negative ones, or vice versa. The overall number of summands never increases.

Original ESSA. The termination criterion and the error-free transformation used by Ratschek and Rokne are based on exponent extraction. Let F be the exponent of b_1 if we normalize the mantissa to a number in $[0.5, 1)$, i.e., $F = \lfloor \log_2 b_1 \rfloor + 1$. If $a_1 \geq n2^F$, the sign of S is known since

$$S = \sum_{i=1}^{m} a_i - \sum_{j=1}^{n} b_j \geq a_1 - nb_1 \geq n(2^F - b_1) > 0,$$

and original ESSA terminates in this case. Likewise, let E be the exponent of a_1. In case $b_1 \geq m2^E$, the sign is negative. If the sign can not be determined, the termination criterion allows to bound the maximal difference between E and F. Using the assumption $2l\varepsilon_{\mathtt{m}} \leq 1$, it can easily be shown that $|E - F| \leq p - 1$, which means that the mantissae of a_1 and b_1 overlap in at least one bit.

Lets turn to the error-tree transformation. If both a_1 and b_1 have the same exponent, we can compute their difference exactly, i.e. $x = a_1 \ominus b_1 = a_1 - b_1$ and $y = 0$. If $E > F$ and hence $a_1 > b_1$, we compute $x = a_1 \ominus u$ and $y = u \ominus b_1$, where $u = 2^{\lceil \log_2 b_1 \rceil}$. Since $1 \leq E - F \leq p - 1$, the operand u overlaps the mantissa of a_1, and therefore $x = a_1 - u$. Furthermore $\frac{1}{2}u \leq b_1 \leq u$ and therefore $y = u - b_1$ by Sterbenz Lemma. The case $F > E$ is handled analogously.

These two steps, the termination criterion and error-free transformation are applied repeatedly until the sign is known. Since a and b are maintained in a heap priority queue, there is an initialization cost of $O(l)$ and each iteration takes time $O(\log l)$. Ratschek and Rokne show the following bound on the number of iterations.

Theorem 4.1.
Run original ESSA *with initially* $l \leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}$ *summands. Then the algorithm terminates after at most* pl^2 *iterations.*

Marina Gavrilova, Dmitri Gavrilov and Jon Rokne [**36**] propose several modifications to the error-free transformation of original ESSA. In case $E > F$, they propose to split b_1 into b_{hi} and b_{lo} such that $b_1 = b_{\text{hi}} + b_{\text{lo}}$ and b_{hi} is basically the part of b_1 overlapping the mantissa of a_1. Then they compute $x = a_1 - b_{\text{hi}}$ and $y = b_{\text{lo}}$. The case $F > E$ is handled analogously. The variants differ in whether $b_{\text{hi}} > b_1$ or $b_{\text{hi}} < b_1$ and hence in the sign of b_{lo}. Compared to the original error-free transformation, the proposed modifications increase the amount of cancellation taking place in a single iteration and should therefore lead to faster termination.

Revised ESSA. The special operations on floating-point numbers needed by the original ESSA variants can be quite expensive. We will now present a new variant which we call revised ESSA, which uses ordinary floating-point arithmetic only in the termination criterion and FastTwoSum for the error free transformation. It thus maximizes the amount of cancellation in each step. Our variant keeps the main advantage of ESSA, which is robustness towards the floating-point environment. Like original ESSA, revised ESSA is immune to overflow and underflow and works in any IEEE 754 rounding mode. Hence it will always compute the correct sign, provided

the input numbers are real numbers. Furthermore, we are able to give an improved upper bound on the number of iterations. In experiments we can observe a reduced number of iterations, too [**67**]. For all floating-point operations, we assume faithful rounding only, in the following. Again, we start with the termination criterion, which checks if the sign can be determined.

Algorithm 4.2 (ESSA_SIGN).
Let a, b, l, and S be defined as above. Run ESSA_SIGN(a_1, m, b_1, n) with faithful rounding. Then ESSA_SIGN returns either UNKNOWN or the sign of S. Let $\alpha = \max\{a_1, b_1\}$ and $\beta = \min\{a_1, b_1\}$. If UNKNOWN is returned, then $\alpha < l\beta$.

1: ***procedure*** ESSA_SIGN(a_1, m, b_1, n)
2: ***if*** $m = 0$ ***and*** $n = 0$ ***then return*** 0
3: ***if*** $n = 0$ ***then return*** $+1$
4: ***if*** $m = 0$ ***then return*** -1
5: ***if*** $a_1 > n \otimes b_1$ ***then return*** $+1$
6: ***if*** $b_1 > m \otimes a_1$ ***then return*** -1
7: ***return*** UNKNOWN

PROOF. Consider Line 5 first. If ESSA_SIGN returns here, $a_1 > n \otimes b_1$ implies by monotonicity of rounding, that $a_1 > nb_1$. From there it follows that

$$S = \sum_{i=1}^{m} a_i - \sum_{j=1}^{n} b_j \geq a_1 - nb_1 > 0$$

and the correct sign is returned. An analogous claim holds for Line 6. Now we assume that ESSA_SIGN does not return in Line 5, and ignore the possibility of overflow for a moment. Then by Equation (2.5) we know that

$$a_1 \leq n \otimes b_1 = (1+\delta_1)nb_1 + \mu_1 \qquad |\delta_1| < 2\varepsilon_{\mathrm{m}}, \ |\mu_1| < \eta, \ \delta_1\mu_1 = 0.$$

We have $n \leq l - 1$ since $m > 0$ and $2l\varepsilon_{\mathrm{m}} \leq 1$, therefore

$$\begin{array}{lllllll} a_1 & < & (1+2\varepsilon_{\mathrm{m}})nb_1 & \leq & (1+2\varepsilon_{\mathrm{m}})(l-1)b_1 & < & lb_1 \quad \text{for } \mu_1 = 0 \\ a_1 & < & nb_1 + \eta & \leq & (l-1)b_1 + b_1 & = & lb_1 \quad \text{for } \delta_1 = 0. \end{array}$$

If an overflow occurs in the product $n \otimes b_1$, we still have $a_1 < nb_1 < lb_1$, since $nb_1 \leq a_1 \in \mathbb{F}$ implies that no overflow occurs. In any case $a_1 < lb_1$. From Line 6 we have analogously $b_1 < la_1$ and combining these two inequalities we have $\alpha < l\beta$. □

We perform the error-free transformation of $a_1 - b_1$ into $x + y$ using FASTTWOSUM. The correctness of FASTTWOSUM relies on the fact, that if $x = a \oplus b$ is computed with rounding to nearest, the exact rounding error $a + b - x$ is a floating-point number. For faithful rounding this is not necessarily true. However, $\alpha < l\beta$ from Algorithm 4.2 tells us that a_1 and b_1 overlap in at least one bit. This property allows us to show that the rounding error involved in $a_1 \ominus b_1$ is a floating-point number even for faithful rounding. Depending on whether $a_1 - b_1$ is rounded up or down to x, there are two possible outcomes for the pair (x, y). An example is given in Figure 4.1.

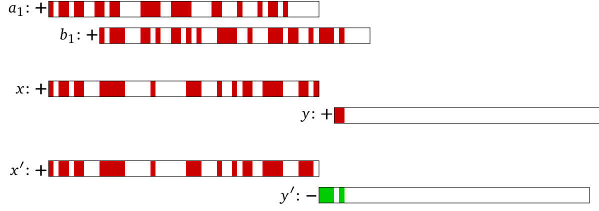


FIGURE 4.1. The two possible results from the error-free transformation $a_1 - b_1 = x + y$ in ESSA.

In our usage of FASTTWOSUM, we rearrange the error-term computation to avoid unary minus operations. The algorithm is given below. One might think of replacing FASTTWOSUM with TWOSUM to avoid the branch, but knowing which of a_1 or b_1 is larger is equivalent to knowing the sign of x by Equation (2.10). We need to know this sign anyway for insertion of x into the set of positive or negative summands.

Algorithm 4.3 (ESSA_DIFF).
Let a, b, l, and S be given as above. Compute $(x, y) \leftarrow$ ESSA_DIFF(a_1, b_1) with faithful rounding. Then $x + y = a_1 - b_1$. Let $\alpha = \max\{a_1, b_1\}$ and $\beta = \min\{a_1, b_1\}$. Then $|x| < \alpha$ and $|y| < 2\varepsilon_{\mathrm{m}} l\beta$. If ESSA_DIFF is performed with rounding to nearest, then $|y| < \varepsilon_{\mathrm{m}} l\beta$.

1: ***procedure*** ESSA_DIFF(a_1, b_1)
2: $\quad x \leftarrow a_1 \ominus b_1$
3: $\quad$ ***if*** $a_1 \geq b_1$ ***then***
4: $\quad\quad y \leftarrow (a_1 \ominus x) \ominus b_1$
5: $\quad$ ***else***
6: $\quad\quad y \leftarrow a_1 \ominus (b_1 \oplus x)$
7: $\quad$ ***return*** (x, y)

This already shows that ESSA terminates, since in each step a_1 and b_1 are replaced by numbers smaller in absolute value. Since positive floating-point numbers can not be arbitrarily small, sooner or later zero values must be produced which are not inserted into the sequences a or b again. The number of summands drops and eventually ESSA terminates.

PROOF. Let $s = a_1 - b_1$, $x = \tilde{\mathrm{fl}}(s) = a_1 \ominus b_1$ and $y = s - x$. We will first show that $y \in \mathbb{F}$ and then proceed to show that y is actually computed by ESSA_DIFF. Finally we prove the claimed inequalities.

We show $y \in \mathbb{F}$ first. Assume $y \neq 0$ and let

$$\sigma = \min\{\mathrm{lsb}(a_1), \mathrm{lsb}(b_1)\},$$

then we know by Equation (2.6) – Equation (2.9), that $a_1, b_1, s, x, y \in \sigma\mathbb{Z}$ and particularly $\sigma \leq \mathrm{lsb}(y)$. From Algorithm 4.2 we know that $\alpha < l\beta \leq \frac{1}{2}\varepsilon_{\mathrm{m}}^{-1}\beta$, so we

can conclude

$$\begin{aligned} \mathrm{msb}(\alpha) &\leq \tfrac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\,\mathrm{msb}(\beta) &\leq (\tfrac{1}{2}\varepsilon_{\mathtt{m}}^{-1})^2\,\mathrm{lsb}(\beta) \quad \text{and} \\ \mathrm{msb}(\alpha) &\leq \tfrac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\,\mathrm{lsb}(\alpha) &\leq (\tfrac{1}{2}\varepsilon_{\mathtt{m}}^{-1})^2\,\mathrm{lsb}(\alpha). \end{aligned}$$

The first of these two inequalities is the critical one. Combining them, we get

$$\mathrm{msb}(\alpha) \;\leq\; (\tfrac{1}{2}\varepsilon_{\mathtt{m}}^{-1})^2\sigma.$$

We have $|y| = |\delta_1||s|$ with $|\delta_1| < 2\varepsilon_{\mathtt{m}}$ by Equation (2.11). From $-b_1 < s < a_1$ follows $|s| < \alpha$. Combining all the collected inequalities, we have

$$\mathrm{msb}(y) \;\leq\; 2\varepsilon_{\mathtt{m}}\,\mathrm{msb}(s) \;\leq\; 2\varepsilon_{\mathtt{m}}\,\mathrm{msb}(\alpha) \;\leq\; 2\varepsilon_{\mathtt{m}}(\tfrac{1}{2}\varepsilon_{\mathtt{m}}^{-1})^2\sigma \;\leq\; \tfrac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\,\mathrm{lsb}(y).$$

Thus, we know that $y \in \tilde{\mathbb{F}}$ and with $\eta \leq \sigma \leq \mathrm{lsb}(y)$ and $\mathrm{msb}(y) \leq \mathrm{msb}(\alpha) \leq \tau$ it follows that $y \in \mathbb{F}$.

Now we show that y is indeed computed. We consider the case $a_1 \geq b_1$ first. In case $b_1 \geq \frac{1}{2}a_1$, by Sterbenz Lemma we have $x = a_1 - b_1$ and consequently $y = 0$. In the other case $b_1 < \frac{1}{2}a_1$, we have $\frac{1}{2}a_1 = a_1 - \frac{1}{2}a_1 < a_1 - b_1$ and with monotonicity of rounding it follows that $\frac{1}{2}a_1 \leq x$. Furthermore, we have $x \leq a_1$ and can apply Sterbenz Lemma to the computation $b_v \leftarrow a_1 \ominus x$, therefore $b_v = a_1 - x$. Finally, we already know, that $b_v - b_1 = a_1 - x - b_1$ is a floating-point number, so in the last step $y \leftarrow b_v \ominus b_1$ we have $y = b_v - b_1$. For the case $a_1 < b_1$ the proof proceeds analogously, after observing that $x < 0$ and $b_1 \oplus x = b_1 \ominus (-x)$.

It is worth noting that the discussion above implies that no overflow occurs, all claims hold unconditionally. Another way to see this is, that by $|s| < \alpha \leq 2\tau(1-\varepsilon_{\mathtt{m}})$ all computed quantities are smaller than $2\tau(1-\varepsilon_{\mathtt{m}})$ in absolute value and hence no overflow occurs.

Finally we turn to the bounds on $|x|$ and $|y|$. From $|s| < \alpha$ follows $|x| \leq \alpha$ with monotonicity of rounding. For $|y|$ we have with Equation (2.11)

$$|y| \;=\; |\delta_1||s| \;\leq\; 2\varepsilon_{\mathtt{m}}\alpha \;<\; 2\varepsilon_{\mathtt{m}} l\beta.$$

Now assume $|x| = \alpha$, then $|y| = \beta$, which is a contradiction to $2\varepsilon_{\mathtt{m}} l \leq 1$. Hence $|x| < \alpha$. When rounding to nearest, we can replace $2\varepsilon_{\mathtt{m}}$ by $\varepsilon_{\mathtt{m}}$ in the bound on $|y|$ by Equation (2.15). □

We will now give an upper bound on the number of iterations revised ESSA performs when run in rounding to nearest.

Theorem 4.4.
Run revised ESSA *with initially $l \leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}$ summands and rounding to nearest. Then the algorithm terminates after at most*

$$\chi(l)l^2 \qquad \textit{iterations, where} \qquad \chi(l) = \frac{1}{2}\left\lceil \frac{p + \log_2 l}{p - \log_2 l} \right\rceil.$$

Ratschek and Rokne show that their original ESSA terminates after at most pl^2 iterations. We remark that $\chi(l) < p$ for $l \leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}$ so our bound is indeed an improvement. Furthermore for IEEE 754 `binary64` arithmetic we have $\chi(l) \leq 1$ for $l \leq 2^{17}$, which should include nearly all applications.

Our proof of the bound closely follows the proof of the upper bound for the original ESSA by Ratschek and Rokne. But instead of working with the exponents of numbers we look at their actual values and exploit the better reduction of a single value in one iteration.

PROOF. For $l = 1$ the algorithm does not perform an iteration, so we can assume $l \geq 2$. We consider the input numbers $a_1, a_2, \ldots, a_m$ and $b_1, b_2, \ldots, b_n$ as a single sequence of variables $c_1, c_2, \ldots, c_l$. Let σ be the smallest non-zero bit in all summands and let it be attained by c_μ:

$$\sigma = \min\{\operatorname{lsb}(c_i) \mid 1 \leq i \leq l\} = \operatorname{lsb}(c_\mu)$$

It follows that $c_i \in \sigma\mathbb{Z}$ for $1 \leq i \leq l$. When we compute $x + y = c_i - c_j$ in an iteration, we replace $\alpha = \max\{c_i, c_j\}$ with $|x|$ and $\beta = \min\{c_i, c_j\}$ with $|y|$ in the sequence. Since x and y are computed using the operations $\oplus, \ominus$ only, after such an operation still $c_i \in \sigma\mathbb{Z}$ for $1 \leq i \leq l$. By replacing α and β with $|x|$ and $|y|$, the size of the elements in the sequence is gradually reduced. We denote by c_i^r the value of c_i after it has played r times the role of β. As soon as all elements are smaller than $\varepsilon_{\mathtt{m}}^{-1}\sigma$, all computations $x + y = c_i - c_j$ will result in $y = 0$. This holds in the case $c_i - c_j = 0$. Otherwise we have

$$\operatorname{msb}(c_i - c_j) \ \leq\ \max\{\operatorname{msb}(c_i), \operatorname{msb}(c_j)\} \ \leq\ \tfrac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\sigma \ \leq\ \tfrac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\operatorname{lsb}(c_i - c_j)$$

which shows $c_i - c_j \in \tilde{\mathbb{F}}$. Furthermore, we have $\eta \leq \sigma \leq \operatorname{lsb}(c_i - c_j)$ and $\operatorname{msb}(c_i - c_j) \leq \tau$, so $c_i - c_j \in \mathbb{F}$ and hence $x = c_i \ominus c_j = c_i - c_j$ and $y = 0$.

When an element plays the role of β in a computation, its absolute value is reduced by a factor of $\varepsilon_{\mathtt{m}} l$ in Algorithm 4.3. We will now count how often an element must play the role of β to be reduced to $\varepsilon_{\mathtt{m}}^{-1}\sigma$. We do not count when an element plays the role of α, since then another element plays the role of β. When an element plays the role of α its value is not increased so this does not invalidate the analysis.

Assume that at the start of the algorithm the sequence is ordered, i.e., $c_1 \geq c_2 \geq \ldots \geq c_l$. We can further assume, that the quotient of two consecutive numbers is not too large, precisely we assume

$$c_i \leq l\varepsilon_{\mathtt{m}}^{-1} c_{i+1}. \tag{4.1}$$

To justify this assumption, let ν be the lowest index such that the assumption is not fulfilled, i.e.,

$$c_\nu > l\varepsilon_{\mathtt{m}}^{-1} c_{\nu+1}. \tag{4.2}$$

Let $\hat{\sigma} = \min\{\operatorname{lsb}(c_i) \mid 1 \leq i \leq \nu\} = \operatorname{lsb}(c_{\hat{\mu}})$. The algorithm now reduces the c_i values, however as long as an operation $x + y = c_i - c_j$ only involves elements with $i, j \leq \nu$,

we have $c_i \in \hat{\sigma}\mathbb{Z}$ for $1 \le i \le \nu$. As long as values are non-zero, they stay larger than $lc_{\nu+1}$:

$$c_i^r \;\ge\; \hat{\sigma} = \mathrm{lsb}(c_{\hat{\mu}}) \;\ge\; 2\varepsilon_{\mathtt{m}}\,\mathrm{msb}(c_{\hat{\mu}}) \;>\; \varepsilon_{\mathtt{m}} c_{\hat{\mu}} \;\ge\; \varepsilon_{\mathtt{m}} c_\nu \;>\; lc_{\nu+1}.$$

There are now two possibilities:

- The algorithm only considers elements c_i, c_j with $i, j \le \nu$ until all these elements are zero and then proceeds with elements with larger index. In this case we can scale all c_i, $i > \nu$ by an appropriate power of two, such that $c_\nu \le l\varepsilon_{\mathtt{m}}^{-1} c_{\nu+1}$ but still $c_\nu > \varepsilon_{\mathtt{m}}^{-1} c_{\nu+1}$. This will not change the course of the algorithm.
- At some point the algorithm considers for the first time two elements c_i^r with $i \le \nu$ and $c_j = c_j^0$ with $j > \nu$. Then the algorithm terminates immediately, because $c_i^r > lc_{\nu+1} \ge lc_j$ but Algorithm 4.2 guarantees $c_i^r < lc_j$. This is not a worst case, the algorithm would perform more steps if the gap was smaller.

Having justified the assumption we proceed with counting how often c_i must play the role of β to be smaller than $\varepsilon_{\mathtt{m}}^{-1}\sigma$. We have $c_l \le c_\mu < \varepsilon_{\mathtt{m}}^{-1}\sigma$ and therefore

$$\begin{aligned} c_i^{k_i} &< (\varepsilon_{\mathtt{m}} l)^{k_i} c_i && \text{by Algorithm 4.3} \\ &\le (\varepsilon_{\mathtt{m}} l)^{k_i} (l\varepsilon_{\mathtt{m}}^{-1})^{l-i} c_l && \text{by Equation (4.1)} \\ &< (\varepsilon_{\mathtt{m}} l)^{k_i} (l\varepsilon_{\mathtt{m}}^{-1})^{l-i} \varepsilon_{\mathtt{m}}^{-1}\sigma \end{aligned}$$

so $c_i^{k_i} < \varepsilon_{\mathtt{m}}^{-1}\sigma$ if $(\varepsilon_{\mathtt{m}} l)^{k_i}(l\varepsilon_{\mathtt{m}}^{-1})^{l-i} \le 1$. This is the case if

$$k_i \ge (l-i)\left\lceil \frac{t + \log_2 l}{t - \log_2 l} \right\rceil = (l-i)2\chi(l).$$

The right hand side must be rounded up, because only an integral number of iterations can be performed. When all summands are smaller than $\varepsilon_{\mathtt{m}}^{-1}\sigma$, at most l additional iterations are required, so finally the total number of iterations is bounded by

$$l + \sum_{i=1}^{l} k_i \;=\; l + 2\chi(l)\sum_{i=1}^{l}(l-i) \;\le\; \chi(l)l^2 \qquad \text{for } l \ge 2,$$

which finishes the proof. □

In experiments we observed, that revised ESSA usually need less than l iterations, and fewer iterations than original ESSA. But we also construct examples, where revised ESSA needs more iterations than original ESSA [**67**].

4.1.2. The SIGNK Algorithm. In this section we present another algorithm, SIGNK to compute the sign of a sum exactly. The algorithm uses compensated summation, a well known technique to increase the accuracy of summation algorithms. Compensated summation can be traced back at least to Kahan [**47**]. Ogita, Rump and Oishi [**77**] analyze compensated summation and give an error bound that can be used to verify the sign of the computed approximation. Using a result by Rump [**91**],

we improve this error bound. Finally, we show how to use compensated summation iteratively for sign computation.

In the following we assume, that no overflow occurs and that the number of summands n is bounded by $\varepsilon_{\mathtt{m}} n < 1$. Consider the following algorithm, which computes the sum of n floating-point values straightforwardly.

1: **procedure** PLAINSUM$(p_1, \dots, p_n)$
2: $s \leftarrow p_1$
3: $\sigma \leftarrow |p_1|$
4: **for** $i \leftarrow 2$ to n **do**
5: $s \leftarrow s \oplus p_i$
6: $\sigma \leftarrow \sigma \oplus |p_i|$
7: **return** (s, σ)

The approximation s, computed by PLAINSUM satisfies the following standard error estimate, see e.g., [**44, 77**].

$$\left| s - \sum_{i=1}^{n} p_i \right| \leq \gamma_{n-1} \sum_{i=1}^{n} |p_i| \qquad \text{where} \qquad \gamma_n = \frac{n\varepsilon_{\mathtt{m}}}{1 - n\varepsilon_{\mathtt{m}}}. \tag{4.3}$$

Rump [**91**] improves upon this in several ways. Let s and σ be computed by PLAINSUM. Then

$$\left| s - \sum_{i=1}^{n} p_i \right| \leq (n-1) \otimes \left(\varepsilon_{\mathtt{m}} \otimes \operatorname{msb}(\sigma) \right). \tag{4.4}$$

Note that for this bound to be valid, both s and σ must be computed in the same order! The new bound is an improvement for several reasons. To actually compute an upper bound of the right hand side in Equation (4.3), we have to take care of rounding errors in the computation of the bound, e.g., resulting from replacing $\sum_{i=1}^{n} |p_i|$ with σ. The new bound can easily be computed. Furthermore, $\operatorname{msb}(\sigma)$ is up to a factor of two smaller than $\sigma \approx \sum_{i=1}^{n} |p_i|$, while $(n-1)\varepsilon_{\mathtt{m}}$ is similar in size to γ_{n-1} for $n << \varepsilon_{\mathtt{m}}^{-1}$.

The basis for compensated summation is the following algorithm, that transforms a sequence of floating-point numbers into a new sequence of floating-point numbers with the same sum.

1: **procedure** VECSUM$(p_1, \dots, p_n)$
2: $t \leftarrow p_1$
3: **for** $i \leftarrow 1$ to $n-1$ **do**
4: $(t, q_i) \leftarrow$ TWOSUM(t, p_{i+1})
5: $q_n \leftarrow t$
6: **return** $(q_1, \dots, q_n)$

The sequence $q_1, \dots, q_n$ computed by VECSUM satisfies $\sum_{i=1}^{n} q_i = \sum_{i=1}^{n} p_i$ by the properties of TWOSUM. Note that q_n is the plain floating-point approximation of the

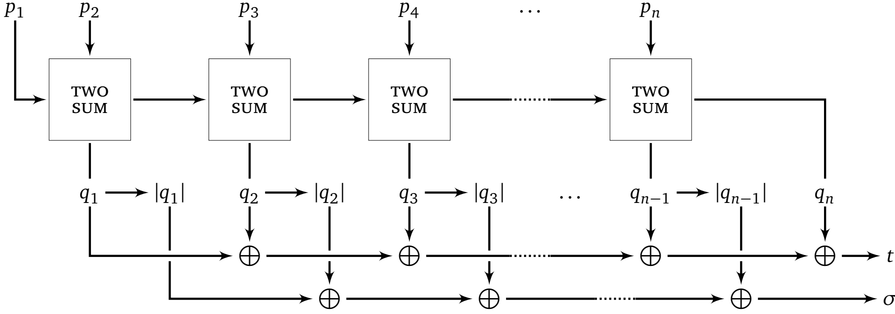


FIGURE 4.2. Flowchart for COMPENSATEDSUM.

sum $\sum_{i=1}^{n} p_i$. Ogita et al. [77] show that the error terms $q_1, \ldots, q_{n-1}$ satisfy

$$\sum_{i=1}^{n-1} |q_i| \leq \gamma_{n-1} \sum_{i=1}^{n} |p_i|, \tag{4.5}$$

i.e., they are much smaller in magnitude than the original summands. Compensated summation now first transforms the sum using VECSUM and then adds the resulting sequence with plain summation.

```
1: procedure COMPENSATEDSUM(p_1, ..., p_n)
2:     q_1, ..., q_n ← VECSUM(p_1, ..., p_n)
3:     (s, σ) = PLAINSUM(q_1, ..., q_{n-1})
4:     t ← q_n ⊕ s
5:     return (t, σ)
```

By using the error estimates Equation (2.12) and Equation (4.3), Ogita et al. prove the error bound

$$\left| t - \sum_{i=1}^{n} p_i \right| \leq \varepsilon_{\mathtt{m}} |t| + \gamma_{n-2} \sum_{i=1}^{n-1} |q_i| \tag{4.6}$$

and show how to compute it rigorously in floating-point arithmetic. Then, they use Equation (4.5) to get

$$\left| t - \sum_{i=1}^{n} p_i \right| \leq \varepsilon_{\mathtt{m}} |t| + \gamma_{n-2} \gamma_{n-1} \sum_{i=1}^{n} |p_i|.$$

In comparison to the standard error estimate Equation (4.3) for plain summation, this shows the improvement from using compensated summation. For $n << \varepsilon_{\mathtt{m}}^{-1}$ the result t is almost as if it were first computed with a precision of $2p$ bits and then rounded to p bits.

We give the following new error bound which improves upon Equation (4.6) in two ways. It may be up to a factor of two smaller and it can be computed more easily.

Our improvement simply stems from replacing Equation (2.12) and Equation (4.3) with Theorem 2.6 and Equation (4.4).

Lemma 4.5.
Let $n\varepsilon_{\mathtt{m}} < 1$ and $p_1, \ldots, p_n \in \mathbb{F}$. Compute $(t, \sigma) = \textsc{CompensatedSum}(p_1, \ldots, p_n)$. Assume no overflow occurs. Then

$$\left| t - \sum_{i=1}^{n} p_i \right| \leq \varepsilon_{\mathtt{m}} \otimes \operatorname{msb}(t) + (n-2) \otimes \big(\varepsilon_{\mathtt{m}} \otimes \operatorname{msb}(\sigma)\big).$$

Proof. We have $t = q_n + s + \delta$ with $|\delta| < \varepsilon_{\mathtt{m}} \operatorname{msb}(t)$ and therefore

$$\left| t - \sum_{i=1}^{n} p_i \right| \leq |\delta| + \left| s - \sum_{i=1}^{n-1} q_i \right|.$$

If $\operatorname{msb}(t) \leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta$, then $\delta = 0$ by Lemma 2.4, otherwise $\varepsilon_{\mathtt{m}} \operatorname{msb}(t) \in \mathbb{F}$. In any case $|\delta| < \varepsilon_{\mathtt{m}} \otimes \operatorname{msb}(t)$. The bound on the other term follows from the improved summation bound Equation (4.4). □

Since only the final addition is not a floating-point operation, we can compute the right hand side in Lemma 4.5 as

$$\Delta_t = (1 + 2\varepsilon_{\mathtt{m}}) \otimes \left\{ \varepsilon_{\mathtt{m}} \otimes \operatorname{msb}(t) \;\oplus\; (n-2) \otimes \big(\varepsilon_{\mathtt{m}} \otimes \operatorname{msb}(\sigma)\big) \right\}.$$

This is valid since $a + b \leq (a \oplus b)(1 + 2\varepsilon_{\mathtt{m}})$ implies $a \oplus b \leq (a \oplus b) \otimes (1 + 2\varepsilon_{\mathtt{m}})$ by monotonicity of rounding. Δ_t can be used to verify the sign of t, which is correct, if $|t| > \Delta_t$. This is the basis for algorithm SignK for computing the sign of a sum.

Algorithm 4.6 (SignK).
Let $p = p_1, p_2, \ldots, p_n \in \mathbb{F}$ and $n\varepsilon_{\mathtt{m}} < 1$, $\varepsilon_{\mathtt{m}} \leq \frac{1}{8}$. Assume rounding to nearest and that no overflow occurs. Then SignK$(p_1, p_2, \ldots, p_n)$ returns the sign of $\sum_{i=1}^{n} p_i$.

```
 1: procedure SignK(p_1, p_2, ..., p_n)
 2:     while n ≥ 2 do
 3:         p_1, ..., p_n ← VecSum(p_1, ..., p_n)
 4:         (s, σ) = PlainSum(p_1, ..., p_{n-1})
 5:         t ← p_n ⊕ s
 6:         Δ_t ← (1 + 2ε_m) ⊗ {ε_m ⊗ msb(t) ⊕ (n − 2) ⊗ (ε_m ⊗ msb(σ))}
 7:         if not  t ≤ Δ_t then return +1
 8:         if not −t ≤ Δ_t then return −1
 9:         eliminate zero summands in p
10:     if n = 0 then return 0
11:     return sign(p_1)
```

In SignK we iterate compensated summation until the sign of the approximation t can be verified using Δ_t, or at most one summand remains. In the actual implementation, summands are overwritten and all computations are moved into a single loop since this is more efficient. If overflow occurs, then t or Δ_t may become nan.

Any comparison involving `nan` evaluates to false, hence the sign check is written to terminate SIGNK in this case.

It is clear from the discussion above, that SIGNK returns the correct sign if no overflow occurs. It remains however to be shown, that SIGNK terminates! The key observation is, that repeated application of VECSUM modifies and in fact improves the sum until at some point the error bound will verify the sign.

Lemma 4.7. SIGNK *always terminates.*

PROOF. First, we show that SIGNK terminates if no overflow occurs. Repeated application of VECSUM transforms the sequence of summands $p_1, \ldots, p_n$ into a special form. We show, that at this point the error bound does indeed verify the sign of the approximation and hence SIGNK terminates.

Let $(x, y) = \text{TwoSum}(p_i, p_{i-1})$ as computed in SIGNK. Then $x \oplus y = x$, and $|y| \le \varepsilon_{\mathtt{m}} \operatorname{msb}(x)$, since y is the rounding error involved in the computation $p_i \oplus p_{i-1}$. Each TWOSUM operation in VECSUM replaces p_i by x and p_{i-1} by y and then continues. Hence, the algorithm moves the more significant parts of the sum towards p_n. This is very similar to bubblesort, where adjacent numbers are swapped if not in correct order. After some iterations of SIGNK, we have $p_i \oplus p_{i-1} = p_i$ for $1 < i \le n$ and VECSUM will not lead to changes any more. We now consider the computation of t and Δ_t in an iteration where none of the summands is changed. The sequence of summands satisfies

$$|p_i| \;\le\; \varepsilon_{\mathtt{m}} \operatorname{msb}(p_{i+1}) \qquad \text{for } 1 \le i < n.$$

SIGNK terminates for $n = 1$, therefore we have $n \ge 2$. Furthermore $|p_n| \ge \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta$. If not, then $|p_{n-1}| \le \varepsilon_{\mathtt{m}}|p_n| < \frac{1}{2}\eta$ and hence $p_{n-1} = 0$. This in turn implies $n = 1$, a case we just excluded. In the considered iteration, SIGNK computes $t = p_n$, $s = p_{n-1}$ and σ as the floating-point sum of $|p_1|, \ldots, |p_{n-1}|$. Let

$$\sigma_1 \;=\; |p_1| \qquad \text{and} \qquad \sigma_i \;=\; |p_i| \oplus \sigma_{i-1} \qquad \text{for } 1 < i < n.$$

We next show

$$\sigma_i \;\le\; 2 \operatorname{msb}(p_i), \qquad \text{for } 1 \le i < n$$

by induction on i. For $i = 1$, we have $\sigma_1 = |p_1| < 2 \operatorname{msb}(p_1)$, so the claim is true. For $i > 2$, first we have

$$\begin{aligned} |p_i| + \sigma_{i-1} &\le |p_i| + 2 \operatorname{msb}(p_{i-1}) \\ &\le 2(1 - \varepsilon_{\mathtt{m}}) \operatorname{msb}(p_i) + 2 \operatorname{msb}(\varepsilon_{\mathtt{m}} \operatorname{msb}(p_i)) \\ &= (2 - 2\varepsilon_{\mathtt{m}}) \operatorname{msb}(p_i) + 2\varepsilon_{\mathtt{m}} \operatorname{msb}(p_i) \\ &= 2 \operatorname{msb}(p_i). \end{aligned}$$

This in turn implies by monotonicity of rounding that

$$\sigma_i \;=\; |p_i| \oplus \sigma_{i-1} \;\le\; 2 \operatorname{msb}(p_i).$$

Altogether we have $\sigma = \sigma_{n-1} \leq 2\,\mathrm{msb}(p_{n-1})$. Note that the order of summation matters for this bound to be valid. Finally we show that $\Delta_t < |t|$ at the end of this iteration, and that SIGNK terminates. We compute

$$\Delta_t \;=\; (1+2\varepsilon_{\mathtt{m}}) \otimes \left\{\varepsilon_{\mathtt{m}} \otimes \mathrm{msb}(p_n) \;\oplus\; (n-2) \otimes \left(\varepsilon_{\mathtt{m}} \otimes \mathrm{msb}(\sigma)\right)\right\}.$$

We have

$$\varepsilon_{\mathtt{m}} \otimes \mathrm{msb}(p_n) \;=\; \varepsilon_{\mathtt{m}}\,\mathrm{msb}(p_n),$$

since $|p_n| > \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta$, and

$$(n-2) \otimes \left(\varepsilon_{\mathtt{m}} \otimes \mathrm{msb}(\sigma)\right) \;=\; (n-2)\left(\varepsilon_{\mathtt{m}} \otimes \mathrm{msb}(\sigma)\right),$$

since $\varepsilon_{\mathtt{m}} \otimes \mathrm{msb}(\sigma)$ is always a power of two. But for $\varepsilon_{\mathtt{m}} \otimes \mathrm{msb}(\sigma)$ we only have the upper bound $2\varepsilon_{\mathtt{m}}\,\mathrm{msb}(\sigma)$. The critical case is $\mathrm{msb}(\sigma) = \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta$, then $\varepsilon_{\mathtt{m}}\,\mathrm{msb}(\sigma) = \frac{1}{2}\eta$, which may be rounded to η. Compensating for the rounding error from the remaining floating-point operations, we have

$$\Delta_t \;\leq\; (1+2\varepsilon_{\mathtt{m}})(1+\varepsilon_{\mathtt{m}})^2\left(\varepsilon_{\mathtt{m}}\,\mathrm{msb}(p_n) + 2(n-2)\varepsilon_{\mathtt{m}}\,\mathrm{msb}(\sigma)\right),$$

and by $n\varepsilon_{\mathtt{m}} < 1$ and $\sigma \leq 2\,\mathrm{msb}(p_{n-1}) \leq 2\varepsilon_{\mathtt{m}}\,\mathrm{msb}(p_n)$ we arrive at

$$\begin{aligned}
\Delta_t \;&\leq\; (1+2\varepsilon_{\mathtt{m}})(1+\varepsilon_{\mathtt{m}})^2\left(\varepsilon_{\mathtt{m}}\,\mathrm{msb}(p_n) + 4\varepsilon_{\mathtt{m}}\,\mathrm{msb}(p_n)\right) \\
&=\; 5\varepsilon_{\mathtt{m}}(1+2\varepsilon_{\mathtt{m}})(1+\varepsilon_{\mathtt{m}})^2\,\mathrm{msb}(p_n) \\
&\leq\; \tfrac{5}{8} \times \tfrac{5}{4} \times \left(\tfrac{9}{8}\right)^2 \mathrm{msb}(t) && \text{for } \varepsilon_{\mathtt{m}} \leq \tfrac{1}{8} \\
&<\; |t|.
\end{aligned}$$

Hence, if no overflow occurs, SIGNK terminates.

Finally we discuss what happens if overflow occurs somewhere. If overflow occurs in the computation of t, then t will be `nan` in this or the next iteration and SIGNK will terminate. This happens because VECSUM recomputes t from the current iteration as new summand p_n in the next iteration. At least one of the new summands will become $\pm\infty$ and TWOSUM will produce `nan` as error term. Hence $t = \mathtt{nan}$ in the next iteration latest.

When overflow occurs in the computation of Δ_t, but not in the computation of t, then $\Delta_t = \infty$ and the sign of t is not verified. This poses a possible threat to termination. We have however shown $\Delta_t < |t|$ for a case that will always be reached if no overflow occurs in the computation of t. Hence, no overflow occurs in the computation of Δ_t in that case and SIGNK will terminate. □

We close with a note on an alternative version of SIGNK. It may seem wasteful to compute t and all necessary intermediate results in one iteration, only to throw them away and recompute them with VECSUM in the next iteration. Instead, one could perform VECSUM only and use the simple error bound Equation (4.4) after the first and Δ_t after subsequent iterations. The computation of t by plain summation, parallel to VECSUM, is however cheap due to instruction level parallelism. In the alternative approach, the first iteration is almost as expensive as a SIGNK iteration,

but gives a result and error bound of significantly lower quality. To get a result of the same quality and be able to use Δ_t, one has to iterate over all summands twice.

4.1.3. Predicates based on Exact Sign of Sum Algorithms. Following [**66**], we now present implementations of the 2D incircle and orientation predicates based on error-free transformations and exact sign of sum algorithms. We compare the efficiency of several predicate implementations by computing Delaunay triangulations.

We consider two strategies to transform the two predicates into a sum at runtime. One results in a fixed number of summands, the other results in a variable, but potentially smaller number of summands. We describe the strategy for the incircle predicate only, it is both more involved and arithmetically more expensive. For the orientation predicate, we use matching strategies.

`S384`**:** Given four points $p, q, r,$ and s the incircle predicate is tantamount to computing the sign of the determinant

$$D_{\mathrm{IC}} = \begin{vmatrix} p_x - s_x & p_y - s_y & (p_x - s_x)^2 + (p_y - s_y)^2 \\ q_x - s_x & q_y - s_y & (q_x - s_x)^2 + (q_y - s_y)^2 \\ r_x - s_x & r_y - s_y & (r_x - s_x)^2 + (r_y - s_y)^2 \end{vmatrix}.$$

Symbolically expanding D_{IC} results in a polynomial expression

$$r_x s_y p_x^2 - r_x q_y p_x^2 + s_x q_y p_x^2 - s_x r_y p_x^2 + q_x r_y p_x^2 - q_x s_y p_x^2 \pm \dots$$

consisting of 48 monomial summands of degree four. By applying TwoProduct six times, we transform each monomial into a sum of 8 summands, resulting in an overall sum for D_{IC} having 384 summands.

`S96:1152`**:** We perform six transformations like $(p_x^{\mathrm{hi}}, p_x^{\mathrm{lo}}) \leftarrow \mathrm{TwoSum}(p_x, -s_x)$, to get the determinant

$$D_{\mathrm{IC}} = \begin{vmatrix} p_x^{\mathrm{hi}} + p_x^{\mathrm{lo}} & p_y^{\mathrm{hi}} + p_y^{\mathrm{lo}} & (p_x^{\mathrm{hi}} + p_x^{\mathrm{lo}})^2 + (p_y^{\mathrm{hi}} + p_y^{\mathrm{lo}})^2 \\ q_x^{\mathrm{hi}} + q_x^{\mathrm{lo}} & q_y^{\mathrm{hi}} + q_y^{\mathrm{lo}} & (q_x^{\mathrm{hi}} + q_x^{\mathrm{lo}})^2 + (q_y^{\mathrm{hi}} + q_y^{\mathrm{lo}})^2 \\ r_x^{\mathrm{hi}} + r_x^{\mathrm{lo}} & r_y^{\mathrm{hi}} + r_y^{\mathrm{lo}} & (r_x^{\mathrm{hi}} + r_x^{\mathrm{lo}})^2 + (r_y^{\mathrm{hi}} + r_y^{\mathrm{lo}})^2 \end{vmatrix}.$$

Note that some of the $\square^{\mathrm{lo}}$ values might be zero, because the corresponding subtraction is exact. Since by Sterbenz Lemma a floating-point subtraction is exact if the operands differ by a factor of two in size at most, this is not unlikely! We transform

$$\begin{aligned} &(p_x^{\mathrm{hi}} + p_x^{\mathrm{lo}})^2 + (p_y^{\mathrm{hi}} + p_y^{\mathrm{lo}})^2 = \\ &p_x^{\mathrm{hi}} p_x^{\mathrm{hi}} + 2 p_x^{\mathrm{hi}} p_x^{\mathrm{lo}} + p_x^{\mathrm{lo}} p_x^{\mathrm{lo}} + p_y^{\mathrm{hi}} p_y^{\mathrm{hi}} + 2 p_y^{\mathrm{hi}} p_y^{\mathrm{lo}} + p_y^{\mathrm{lo}} p_y^{\mathrm{lo}} \end{aligned}$$

and analogous terms into a sum using TwoProduct, where a product incurring $\square^{\mathrm{lo}}$ is computed only if $\square^{\mathrm{lo}} \neq 0$. The resulting sum has between 4 and 12 summands. Then we use cofactor expansion on the last column. We transform

$$(q_x^{\mathrm{hi}} + q_x^{\mathrm{lo}})(r_y^{\mathrm{hi}} + r_y^{\mathrm{lo}}) - (q_y^{\mathrm{hi}} + q_y^{\mathrm{lo}})(r_x^{\mathrm{hi}} + r_x^{\mathrm{lo}})$$

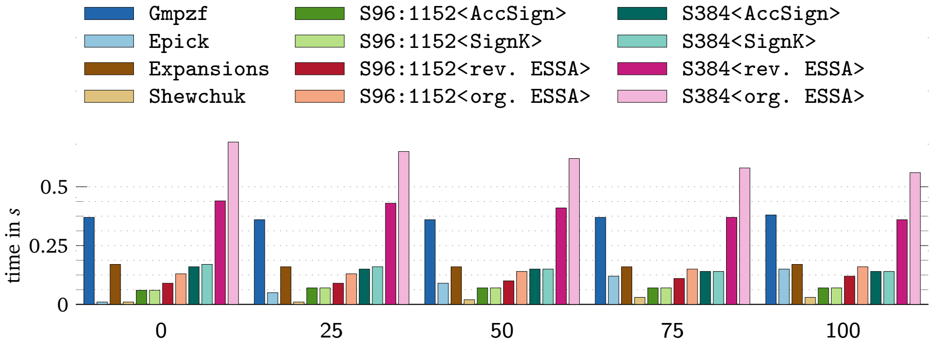


FIGURE 4.3. Delaunay triangulation, floating-point data sets on `descartes`.

and resembling expressions into a sum with between 4 and 16 summands. Each remaining product of two sums is then transformed into a single sum by applying TwoProduct to each pair of summands, resulting in a sum with between 32 to 384 summands. Overall, we get an expression with between 96 to 1152 summands.

We combine both methods to create a sum with four exact sign of sum algorithms. Those are AccSign [**95**], a variant of AccSum specialized for sign computation, the SignK algorithm, the original ESSA implementation by Ratschek and Rokne [**57**], and our revised ESSA variant. We compare the resulting predicate implementations with Shewchuk's adaptive predicates, as well as predicates evaluating the determinants straightforwardly using expansions [**104**]. For reference we also add the `Epick` kernel as well as `Gmpzf` to the list of competitors.

We use the setup from Section 3.4 and run experiments on the `descartes` platform, using floating-point data sets. Results are shown in Figure 4.3. Note that cross comparisons with Figure 3.18, and Table A.1 are valid.

Clearly, `S96:1152` has a significant advantage over `S384`. Closer inspection shows, that about 75% of sums of variable length have the minimum number of 96 summands and virtually all of them have at most the 384 summands of the fixed length sum [**66**]. This is caused by the good locality of the Delaunay triangulation algorithm. Nearly all incircle tests are performed with points close to each other, therefore nearly all of the initial transformations are exact, leading to short sums.

All predicates based on exact sign of sum algorithms show non-adaptive behavior and in case of ESSA even anti adaptive behavior. For SignK and AccSign, the explanation is that the generated sums are not complicated enough. Both algorithms always terminate after exactly one iteration of the main loop. For the same reason, there is also not much difference in running time between them. Our revised ESSA variant is clearly better than original ESSA, yet worse than other exact sign of sum algorithms, especially for longer sums.

The experiments show that our predicate implementations behave more like an exact evaluation of the expression. They are however faster than `Gmpzf` and, ignoring ESSA, faster than expansions, too. For data sets with many degeneracies, SIGNK on short sums is even faster than the adaptive `Epick` kernel. This suggests to use our predicates as last stage in an adaptive evaluation scheme, e.g., replacing the last stage in the `Epick` kernel. In [**66**] we show that this leads indeed to very fast predicates, challenging even Shewchuk's adaptive predicates. In Chapter 5 we show how to utilize the speed of exact computations based on error free transformations in expression dag based number types like *RealAlgebraic*.

4.2. Expansion to Bigfloat Conversion

An expansion $e = e_1, e_2, \ldots, e_n$ is a special sequence of floating-point numbers, representing the sum of its elements

$$E = \sum_{i=1}^{n} e_i.$$

In this section we address the problem of converting e exactly and efficiently into a bigfloat number, that is, we want to compute E. A straightforward approach is, to first convert each summand e_i into a bigfloat number and then compute the sum exactly. This approach however does not take into account the special properties of expansions, it works for any sum of floating-point numbers. Since summands in an expansion are non-overlapping, computing their sum should be easier. The main obstacle to better conversion algorithms is that summands may have different signs. We call an expansion where all summands have the same sign *monotone*. To convert a monotone expansion into a bigfloat number, it suffices to concatenate or join the mantissae of all summands appropriately, see Figure 4.4.

The first result of this section is an efficient algorithm based on error-free transformations for converting any expansion into a monotone, maximally non-overlapping expansion. In the important case of strongly non-overlapping expansions, the algorithm is free from overflow and underflow. This means it always works and no fall-back conversion strategy is needed. The second result are algorithms joining the mantissae of monotone `binary64` expansions into arbitrary precision mantissae with 32 bit and 64 bit limbs. These algorithms perform integer arithmetic only and are essentially branch free.

For both MPFR and `leda::bigfloat`, we implement two conversion strategies based on these algorithms. Experiments show that they are significantly more efficient than the straightforward approach.

4.2.1. Monotone Expansions. An expansion $e_1, e_2, \ldots, e_n$ is called *monotone*, if all summands are positive, or all summands are negative or it consists of one zero summand only. A monotone expansion is truly monotone in that summands are strictly ordered, increasing if the summands are positive, decreasing if they are negative. If e is monotone, it contains the non-zero bits of a binary representation of

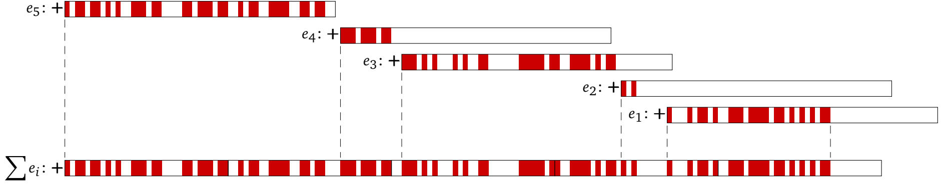


FIGURE 4.4. Joining the mantissae from a monotone expansion.

$E = \sum_{i=1}^{n} e_i$ explicitly, see again Figure 4.4. Monotone, maximally non-overlapping expansions are unique.

Lemma 4.8. *Let $e = e_1, e_2, \dots, e_n$ and $f = f_1, f_2, \dots, f_k$ be two monotone, maximal non-overlapping expansions which represent the same number, i.e., with $\sum_{i=1}^{n} e_i = \sum_{j=1}^{k} f_j$. Then $n = k$ and $e_i = f_i$ for $1 \le i \le n$.*

PROOF. Clearly, e_n and f_k have the same sign. We assume they are positive, the other case is symmetric. Let $s = \sum_{i=1}^{n} e_i$. We have $s \ge e_n$ and hence $\operatorname{msb}(s) \ge \operatorname{msb}(e_n)$. Furthermore

$$\begin{aligned} s &= \sum_{i=1}^{n} e_i \\ &\le \sum_{i=1}^{n} 2\operatorname{msb}(e_i)(1 - \varepsilon_{\mathrm{m}}) \\ &\le \sum_{i=1}^{n} 2\operatorname{msb}(e_n)\varepsilon_{\mathrm{m}}^{n-i}(1 - \varepsilon_{\mathrm{m}}) \\ &= 2\operatorname{msb}(e_n)\left[\sum_{i=0}^{n-1} \varepsilon_{\mathrm{m}}^{i} - \sum_{i=1}^{n} \varepsilon_{\mathrm{m}}^{i}\right] \\ &= 2\operatorname{msb}(e_n)(1 - \varepsilon_{\mathrm{m}}^{n}), \end{aligned}$$

and therefore $\operatorname{msb}(s) \le \operatorname{msb}(e_n)$. The same reasoning applies to f_k and therefore $\operatorname{msb}(e_n) = \operatorname{msb}(f_k)$. Let $\sigma = 2\varepsilon_{\mathrm{m}} \operatorname{msb}(f_k)$, then $e_n - f_k \in \sigma\mathbb{Z}$. Assume $e_n > f_k$, then

$$\sigma \;\le\; e_n - f_k \;=\; \sum_{j=1}^{k-1} f_j - \sum_{i=1}^{n-1} e_i \;\le\; \sum_{j=1}^{k-1} f_j \;<\; 2\operatorname{msb}(f_{k-1}) \;\le\; 2\varepsilon_{\mathrm{m}} \operatorname{msb}(f_k),$$

which is a contradiction. The case $e_n < f_k$ follows analogously. Therefore $e_n = f_k$. The claim follows by induction. □

We call our algorithm to convert expansions into monotone expansions MONOTONIZE. MONOTONIZE is similar to the first stage of Shewchuk's compression algorithm. In Shewchuk's compression, starting with the most significant summand, the summands are added to a running total x, using FASTTWOSUM. Once the error term y is

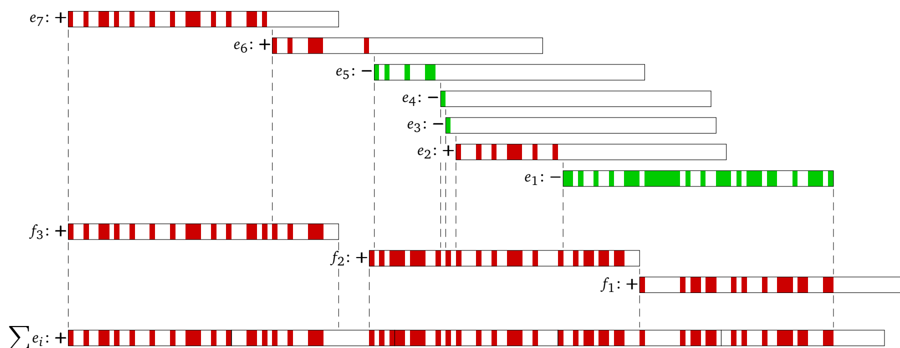


FIGURE 4.5. Conversion of an expansion into a maximally non-overlapping expansion by MONOTONIZE.

non-zero, x is stored away and y becomes the new running total x'. At this point, x and x' are maximal non-overlapping. Since more summands are added to x', it may however later overlap x. The first stage continues until all summands have been processed. Then the second stage of compression removes any overlap. However, after both stages, stored summands may have different signs.

The difference in MONOTONIZE is, that it handles the case of an error term with wrong sign differently and needs no second stage. Let $e = e_1, e_2, \ldots, e_n$ be an expansion with $e_n > 0$. Since the summands are non-overlapping, we have $E = \sum_{i=1}^{n} e_i > 0$ and we need to produce positive output summands. Using FAST-TWOSUM, we add the summands to a running total x until the error term y becomes non-zero. Adding a less significant, non-overlapping summand e_i to x is much like incrementing or decrementing x. No bit of x more significant that $\mathrm{lsb}(x)$ will be altered. Hence, once $y \neq 0$, the remaining summands $e_i, e_{i-1}, \ldots$ can not alter the bits in x but only those in y.

If y is positive, this means x now stores the leading bits of E. We store x into an output summand f_j and continue with y as new running total. If however y is negative, then x is too large. We can however store $\mathrm{pred}(x)$ as output summand, since the error term $w = x - \mathrm{pred}(x)$ is positive and protects the bits in $\mathrm{pred}(x)$ from being changed. Note that w is a floating-point number. With FASTTWOSUM(w, y) we compute a new running total and error term and continue as before. The complete algorithm is given below. Examples for input and output of MONOTONIZE are shown in Figure 4.5, a simplified flowchart for the case $e_n > 0$ is given in Figure 4.6.

Algorithm 4.9 (MONOTONIZE).
Let $e = e_1, e_2, \ldots, e_n$ be an expansion. Assume rounding to nearest and that no overflow occurs. Then MONOTONIZE$(e_1, e_2, \ldots, e_n)$ *computes the unique monotone, maximal non-overlapping expansion $f = f_1, f_2, \ldots, f_k$ with*

$$\sum_{i=1}^{n} e_i = \sum_{j=1}^{k} f_j.$$

The number of output summands is limited by $k \leq \lceil \log_2(2\tau/\eta)/p \rceil$, and MONOTONIZE *takes $O(n+k)$ steps. If $\sum_{i=1}^{n} |e_i| < \tau(2-\varepsilon_{\mathrm{m}})$ or when e is strongly non-overlapping, then no overflow occurs.*

```
 1: procedure MONOTONIZE(e_1, e_2, ..., e_n)
 2:     i ← n
 3:     j ← l
 4:     while i > 1 and e_i = 0 do i--
 5:     a ← e_{i--}
 6:     s ← sign(a)
 7:     while i > 0 do
 8:         (x, y) ← FASTTWOSUM(a, e_{i--})
 9:         while sy < 0 do
10:             v ← NEXTTOWARDZERO(x)
11:             w ← x ⊖ v
12:             f_{j--} ← v
13:             (x, y) ← FASTTWOSUM(w, y)
14:         if sy > 0 then
15:             f_{j--} ← x
16:             a ← y
17:         else
18:             a ← x
19:     f_j ← a
20:     return f_j, f_{j+1}, ..., f_l as f_1, f_2, ..., f_k
```

NEXTTOWARDZERO computes the next floating-point number in direction towards zero. It is a modification of Algorithm 2.12 for computing pred and succ by Rump et al. [**92**] and given below. Unlike other algorithms on expansions, MONOTONIZE creates new summands in order of decreasing significance and may increase the number of summands.

The central property of MONOTONIZE is that the f_j are maximal non-overlapping. We need to show that when $y \neq 0$ occurs, the new output summand does not overlap already stored summands. We will see that when x overlaps the last summand, then x is a power of two, the overlap is exactly one bit, and y is negative. Hence, by passing on to the next floating-point number in direction towards zero, the potential overlap is removed. We will now demonstrate the properties of MONOTONIZE, concentrating

on the case $e_n > 0$. We start by analyzing single operations in MONOTONIZE more closely. The first lemma considers the FASTTWOSUM operation on the left in Figure 4.6.

Lemma 4.10. *Let $a, e_i \in \mathbb{F}$ be non-overlapping with $a > |e_i|$. Furthermore, let $x, y \in \mathbb{F}$ with $x = a \oplus e_i$ and $x + y = a + e_i$. Then $x > 0$ and either*

$$\mathrm{msb}(x) \leq \mathrm{msb}(a) \qquad \textit{or} \qquad x = 2\,\mathrm{msb}(a),\ y < 0.$$

PROOF. Since $a + e_i > 0$ also $x > 0$ by Equation (2.10). Since a and e_i are non-overlapping, we have $2\,\mathrm{msb}(a) > a + e_i$ and with monotonicity of rounding follows $2\,\mathrm{msb}(a) \geq x$. (In case $2\,\mathrm{msb}(a) \notin \mathbb{F}$, actually $2\,\mathrm{msb}(a) > x$, since $x \in \mathbb{F}$ by assumption). If $2\,\mathrm{msb}(a) > x$, then $\mathrm{msb}(a) \geq \mathrm{msb}(x)$. Otherwise $2\,\mathrm{msb}(a) = x$, in this case $x > a + e_i$ and hence $y < 0$. □

Assuming additionally that a is smaller than, and does not overlap the last output summand f_j, Lemma 4.10 tells us that we are exactly in the desired situation: x overlaps f_j in at most one bit and if it does, it is a power of two and $y < 0$. The next two lemmata consider the other path of computation leading to new values of x and y, through the FASTTWOSUM operation on the bottom of Figure 4.6. With x' and y' we denote the values of x and y at the start of this path.

Lemma 4.11. *Let $x', y' \in \mathbb{F}$ with $x' = x' \oplus y'$ and $x' > 0, y' < 0$. Then $x' > \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta$. Let furthermore $v = \mathrm{pred}(x')$ and $w = x' \ominus v$. Then $v > 0$, $\mathrm{msb}(w) = w = x' - v$, and $w > -y'$.*

PROOF. Since y' is the rounding error arising in $x' \oplus y'$, x' and y' are maximal non-overlapping. Assume $x' \leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta$. Then $\mathrm{msb}(x' + y') \leq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta$ and hence $x' \oplus y' = x' + y'$ by Lemma 2.4. This is a contradiction to $y' \neq 0$.

From $x' > \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta$ follows $v \geq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta > 0$. Since x' is the successor of v, we have $w = x' - v = 2\varepsilon_{\mathtt{m}}\,\mathrm{msb}(v) \in \mathbb{F}$. Finally, $w \geq -2y' > -y'$ since $v < x' + y' < x'$ and $x' + y'$ is closer to x' than v due to rounding to nearest. □

Note that w and v are not maximal non-overlapping but overlap in one bit. The next FASTTWOSUM operation, computing a new running total x and error term y from w and y' fixes this.

Lemma 4.12. *Let $w, y' \in \mathbb{F}$ with $\mathrm{msb}(w) = w > -y' > 0$. Furthermore let $x, y \in \mathbb{F}$ with $x = w \oplus y'$ and $x + y = w + y'$. Then $x > 0$ and either*

$$\mathrm{msb}(x) < \mathrm{msb}(w) \qquad \textit{or} \qquad x = w,\ y = y'.$$

PROOF. Since $w + y' > 0$ also $x > 0$ by Equation (2.10). We have $\mathrm{msb}(w) > w + y'$ and with $\mathrm{msb}(w) \in \mathbb{F}$ follows $\mathrm{msb}(w) \geq x$ by monotonicity of rounding. If $\mathrm{msb}(w) > x$, then $\mathrm{msb}(w) > \mathrm{msb}(x)$. Otherwise $w = x$ and $y = y'$. □

Thus, either x and the last output summand v are maximally non-overlapping, or we are again in the situation where x is a power of two, the overlap is exactly one bit and $y < 0$. We are now ready to combine the previous results to prove the properties of MONOTONIZE.

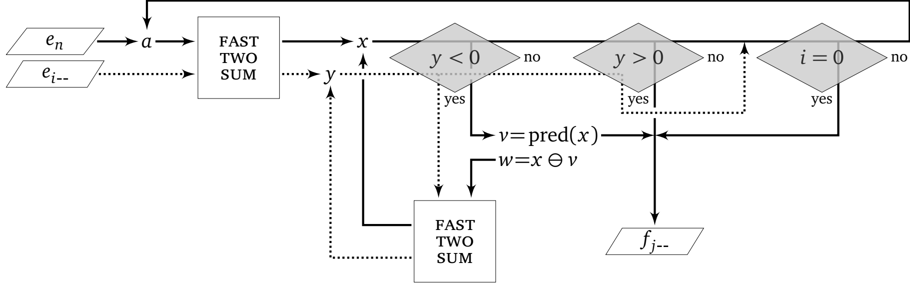


FIGURE 4.6. Flowchart for MONOTONIZE, applied to an expansion with $e_n > 0$.

Lemma 4.13. *Run* MONOTONIZE(e) *with an expansion* $e = e_1, e_2, \dots, e_n$ *with* $e_n > 0$. *Assume that no overflow occurs and denote by* $f_1, f_2, \dots, f_k$ *the sequence of output summands. After any of the two* FASTTWOSUM *operations, we have a sequence of summands*

$$s = e_1,\ e_2,\ \dots,\ e_i,\ y,\ x,\ f_j,\ f_{j+1},\ \dots,\ f_k.$$

Then

$$\text{(A)} \qquad y + x + \sum_{l=j}^{k} f_l = \sum_{l=i+1}^{n} e_l$$

and

$$\text{(B)} \qquad x, f_l > 0 \qquad \textit{for } j \le l \le k$$

$$\text{(C)} \qquad \mathrm{msb}(f_{l-1}) \le \varepsilon_{\mathrm{m}} \,\mathrm{msb}(f_l) \qquad \textit{for } j < l \le k$$

$$\text{(D)} \qquad \mathrm{msb}(y) \le \varepsilon_{\mathrm{m}} \,\mathrm{msb}(x).$$

Furthermore, either

$$\text{(E)} \qquad \mathrm{msb}(x) \le \varepsilon_{\mathrm{m}} \,\mathrm{msb}(f_j) \qquad \textit{or} \qquad x = 2\varepsilon_{\mathrm{m}} \,\mathrm{msb}(f_j),\ y < 0.$$

Thus, MONOTONIZE iteratively transforms the sequence of summands into a new sequence with the same sum. Along the way, the subsequence $e_1, \dots, e_i, y, x$ stays non-overlapping, while the subsequence $y, x, f_j, \dots, f_k$ is maximal non-overlapping, with the already known exception: x may overlap f_j in a single bit. Furthermore the subsequence $x, f_j, \dots, f_k$ is monotone. The following proof is purely technical, it simply combines the previous three lemma with induction, making sure the assumptions are always fulfilled and everything works together nicely.

PROOF. First we note that (**D**) follows from error estimate Equation (2.15) and $\mathrm{msb}(y) \le |y|$, since x and y are always the results of a FASTTWOSUM operation. It remains however to show that FASTTWOSUM can be applied, i.e., that the input numbers are ordered by absolute value.

For (**A**), (**B**), (**C**) and (**E**) we proceed by induction. The base case occurs at the start of MONOTONIZE. The sequence

$$s' = e_1, \ldots, e_n \qquad \text{is turned into} \qquad s = e_1, \ldots, e_{n-2}, y, x$$

by

$$(x, y) \leftarrow \text{FASTTWOSUM}(e_n, e_{n-1}).$$

The summands $e_n > 0$ and e_{n-1} are nonoverlapping and hence we can apply FASTTWOSUM and Lemma 4.10. Equations (**A**) and (**B**) follow directly and for (**C**) and (**E**) there is nothing to show.

For the induction step, let y' be the error term in the sequence, before the FASTTWOSUM operation to be considered. We distinguish three cases, $y' < 0$, $y' > 0$, and $y' = 0$, corresponding to different computation paths in MONOTONIZE. In case $y' < 0$, the sequence

$$s' = \ldots, e_i, y', x', f_{j+1}, \ldots \qquad \text{is turned into} \qquad s = \ldots, e_i, y, x, f_j, f_{j+1}, \ldots$$

by

$$\begin{aligned} f_j &\leftarrow \text{pred}(x') \\ w &\leftarrow x' \ominus \text{pred}(x') \\ (x, y) &\leftarrow \text{FASTTWOSUM}(w, y'). \end{aligned}$$

By induction hypothesis, for s' we have,

(**A′**) $$y' + x' + \sum_{l=j+1}^{k} f_l = \sum_{l=i+1}^{n} e_l$$

(**B′**) $x', f_l, > 0$ for $j + 1 \le l \le k$

(**C′**) $\text{msb}(f_{l-1}) \le \varepsilon_{\text{m}} \text{msb}(f_l)$ for $j + 1 < l \le k$

(**D′**) $\text{msb}(y') \le \varepsilon_{\text{m}} \text{msb}(x')$.

Furthermore, either

(**E′**) $\text{msb}(x') \le \varepsilon_{\text{m}} \text{msb}(f_{j+1})$ or $x' = 2\varepsilon_{\text{m}} \text{msb}(f_{j+1}),\ y' < 0$.

We can apply Lemma 4.11 to x' and y'. Hence $f_j > 0$, which together with (**B′**) shows (**B**). Furthermore $w = x' - \text{pred}(x')$ and $\text{msb}(w) = w > -y'$. Thus we can apply FASTTWOSUM and Lemma 4.12 to w and y'. It follows that

$$y' + x' = y' + w + f_j = y + x + f_j$$

and together with (**A′**) follows (**A**). Since $f_j < x'$, it follows with (**E′**) that $\text{msb}(f_j) \le \varepsilon_{\text{m}} \text{msb}(f_{j+1})$, which together with (**C′**) yields (**C**). Since w is the distance from f_j to the next larger floating-point number, we have $w = 2\varepsilon_{\text{m}} \text{msb}(f_j)$. Following Lemma 4.12, we either have $\text{msb}(x) < \text{msb}(w) = w = 2\varepsilon_{\text{m}} \text{msb}(f_j)$, i.e., $\text{msb}(x) \le \varepsilon_{\text{m}} \text{msb}(f_j)$ or $x = w = 2\varepsilon_{\text{m}} \text{msb}(f_j),\ y = y' < 0$. This shows (**E**) and concludes this case.

In case $y' \geq 0$, the FASTTWOSUM operation on the left in Figure 4.6 is used. We next show that e_i does not overlap a and hence FASTTWOSUM and Lemma 4.10 can be applied. Let $\sigma = 2^k, k \in \mathbb{Z}$ be maximal such that $e_{i+1}, \ldots, e_n \in \sigma\mathbb{Z}$. We need to show that no non-zero bit smaller than σ is created before e_i is used. MONOTONIZE manipulates summands with addition and subtraction, which are safe by Equation (2.9). The only other operation is computing the predecessor $v = \mathrm{pred}(x)$. While v may have smaller non-zero bits than x, at this point the error term y contains already smaller bits, i.e., $\mathrm{lsb}(v) \geq \mathrm{lsb}(y)$ and no non-zero bit smaller than already present is created. Hence $a \in \sigma\mathbb{Z}$ and a does not overlap e_i.
We return to the induction. In case $y' > 0$, the sequence

$$s' = \ldots, e_i, e_{i+1}, y', x', f_{j+1}, \ldots \qquad \text{is turned into} \qquad s = \ldots, e_i, y, x, f_j, f_{j+1}, \ldots$$

by

$$f_j \leftarrow x'$$
$$(x, y) \leftarrow \textsc{FastTwoSum}(y', e_{i+1}).$$

By induction hypothesis, for s' we have,

$$(\mathbf{A}') \qquad y' + x' + \sum_{l=j+1}^{k} f_l = \sum_{l=i+2}^{n} e_l$$

$$(\mathbf{B}') \qquad x', f_l > 0 \qquad \text{for } j+1 \leq l \leq k$$

$$(\mathbf{C}') \qquad \mathrm{msb}(f_{l-1}) \leq \varepsilon_{\mathtt{m}} \, \mathrm{msb}(f_l) \qquad \text{for } j+1 < l \leq k$$

$$(\mathbf{E}') \qquad \mathrm{msb}(x') \leq \varepsilon_{\mathtt{m}} \, \mathrm{msb}(f_{j+1}) \qquad \text{since } y' > 0$$

$$(\mathbf{D}') \qquad \mathrm{msb}(y') \leq \varepsilon_{\mathtt{m}} \, \mathrm{msb}(x').$$

From FASTTWOSUM follows

$$e_{i+1} + y' + x' = y + x + f_j$$

and together with ($\mathbf{A}'$) follows ($\mathbf{A}$). By Lemma 4.10, with $a = y'$, we have $x > 0$ and together with ($\mathbf{B}'$) follows ($\mathbf{B}$). Furthermore ($\mathbf{C}$) follows from ($\mathbf{C}'$) and ($\mathbf{E}'$). With Lemma 4.10 and ($\mathbf{D}'$) we have either $x = 2\,\mathrm{msb}(y') = 2\varepsilon_{\mathtt{m}}\,\mathrm{msb}(f_j)$ and $y < 0$, or $\mathrm{msb}(x) \leq \mathrm{msb}(y') \leq \varepsilon_{\mathtt{m}}\,\mathrm{msb}(f_j)$ i.e., ($\mathbf{E}$) holds.
In case $y' = 0$, the sequence

$$s' = \ldots, e_i, e_{i+1}, y' = 0, x', f_j, \ldots \qquad \text{is turned into} \qquad s = \ldots, e_i, y, x, f_j, \ldots$$

by

$$(x, y) \leftarrow \textsc{FastTwoSum}(x', e_{i+1}).$$

By induction hypothesis, for s' we have,

$$(\mathbf{A}') \qquad x' + \sum_{l=j}^{k} f_l = \sum_{l=i+2}^{n} e_l$$

(**B**′)	$x', f_l > 0$	for $j \le l \le k$
(**C**′)	$\mathrm{msb}(f_{l-1}) \le \varepsilon_{\mathtt{m}} \, \mathrm{msb}(f_l)$	for $j < l \le k$
(**E**′)	$\mathrm{msb}(x') \le \varepsilon_{\mathtt{m}} \, \mathrm{msb}(f_j)$	since $y' = 0$.

For (**C**) there nothing to show, it is identical to (**C**′). From FastTwoSum follows

$$e_{i+1} + x' = y + x$$

and together with (**A**′) we have (**A**). By Lemma 4.10, with $a = x'$, he have $x > 0$, and together with (**B**′) follows (**B**). With Lemma 4.10 and (**E**′) we have either $x = 2\,\mathrm{msb}(x') = 2\varepsilon_{\mathtt{m}} \, \mathrm{msb}(f_j)$ and $y < 0$ or $\mathrm{msb}(x) \le \mathrm{msb}(x') \le \varepsilon_{\mathtt{m}} \, \mathrm{msb}(f_j)$, i.e., (**E**) holds. □

We can now subsume the properties of Monotonize, including the case of negative or zero leading summand.

Lemma 4.14. *Let $e = e_1, e_2, \ldots, e_n$ be an expansion and assume no overflow occurs. Then* Monotonize(e) *computes a monotone, maximal non-overlapping expansion $f_1, f_2, \ldots, f_k$ with $\sum_{i=1}^{n} e_i = \sum_{i=1}^{k} f_i$ and $k \le \lceil \log_2(2\tau/\eta)/p \rceil$ summands.* Monotonize *takes $O(n+k)$ steps.*

The bound on k is tight, running Monotonize with $e = -\eta, \tau$ as input expansion creates an output expansion with $\lceil \log_2(2\tau/\eta)/p \rceil$ summands. This example furthermore shows that $k > n$ is possible.

Proof. Monotonize first skips zero summands. If there are only zero summands, a single zero is returned which is a monotone, maximal non-overlapping expansion. Otherwise it continues computing with a leading non-zero summand. Let s be the sign of this summand. By symmetry of $\mathbb{F}$ and **fl**, a result analogous to Lemma 4.13 also holds for a negative leading summand. After the last FastTwoSum operation we have $sy \ge 0$ since otherwise Monotonize would perform further FastTwoSum operations. At this point we either have a sequence

$$y = 0,\ x,\ f_2,\ f_3,\ \ldots,\ f_k \qquad \text{with} \qquad \mathrm{msb}(x) \le \varepsilon_{\mathtt{m}} \, \mathrm{msb}(f_2)$$

and the computation ends with $f_1 \leftarrow x$, or we have a sequence

$$y \ne 0,\ x,\ f_3,\ f_4,\ \ldots,\ f_k \qquad \text{with} \qquad \mathrm{msb}(x) \le \varepsilon_{\mathtt{m}} \, \mathrm{msb}(f_3)$$

and the computation ends with $f_2 \leftarrow x$, $f_1 \leftarrow y$. Therefore, and by Lemma 4.13

$$\sum_{i=1}^{k} f_i = \sum_{i=1}^{n} e_i$$

and

$$\mathrm{msb}(f_{l-1}) \le \varepsilon_{\mathtt{m}} \, \mathrm{msb}(f_l) \qquad \text{for } 1 < l \le k$$

and all summands have the same sign. Output summands are optimally spaced, i.e., their most significant bits are at least p bits apart. Hence there can be at most

$\lceil \log_2(2\tau/\eta)/p \rceil$ summands. Each iteration consumes an input summand or creates an output summand, therefore the number of iterations is bounded by $O(n+k)$. □

Finally we give two criteria, ensuring that no overflow occurs in MONOTONIZE and all computations are therefore correct. Note that the number $\tau(2-\varepsilon_{\mathtt{m}})$ is the smallest number that may be rounded to $+\infty$, it is halfway between $2\tau(1-\varepsilon_{\mathtt{m}})$ and $\operatorname{succ}(2\tau(1-\varepsilon_{\mathtt{m}})) = 2\tau$, where the successor is taken in $\tilde{\mathbb{F}}$.

Lemma 4.15. *Let $e = e_1, e_2, \ldots, e_n$ be an expansion with $\sum_{i=1}^{n} |e_i| < \tau(2-\varepsilon_{\mathtt{m}})$. Then no overflow occurs in the call* MONOTONIZE(e).

PROOF. We only have to show $a \oplus b \in \mathbb{F}$ for all a, b that we call FASTTWOSUM(a, b) for, cf. Algorithm 2.7. Assume $e_n > 0$. At the start of MONOTONIZE we add $e_n, e_{n-1}, e_{n-2}, \ldots$ to our running total x until the exact sum is not a floating-point number for first time. Let l be the index where this happens, then

$$x = \mathbf{fl}\left(\sum_{i=l}^{n} e_i\right).$$

Since

$$\left|\sum_{i=l}^{n} e_i\right| \leq \sum_{i=1}^{n} |e_i| < \tau(2-\varepsilon_{\mathtt{m}})$$

we round to a real number, i.e., $x \in \mathbb{F}$. We store either x or $\operatorname{pred}(x)$ as first output summand f_k and therefore $f_k \in \mathbb{F}$. All further computations involve numbers smaller than f_k by Lemma 4.13 and hence no overflow can occur. □

The previous criterion allows us to show that for strongly non-overlapping expansions, MONOTONIZE will always work correctly.

Lemma 4.16. *Let $e = e_1, e_2, \ldots, e_n$, be a strongly non-overlapping expansion, then $\sum_{i=1}^{n} |e_i| < \tau(2-\varepsilon_{\mathtt{m}})$ and no overflow occurs in the call* MONOTONIZE(e).

PROOF. The sequence $|e_1|, |e_2|, \ldots |e_n|$ is a strongly non-overlapping expansion too. Consider the binary representation of $E = \sum_{i=1}^{n} |e_i|$. We have $\operatorname{msb}(E) \leq \tau$ and E contains a zero bit at least every $p+1$ bits. Hence $E < \tau(2-\varepsilon_{\mathtt{m}})$. □

The last result is particularly fortunate. Strongly non-overlapping expansions are the most relevant type in practice and impeding overflow may be the reason, that conversion to a bigfloat number is necessary in the first place. If we want to convert strongly non-overlapping expansions with the help of MONOTONIZE, we do not need a secondary or backup strategy, covering the case of overflow.

We are now finished demonstrating the properties of MONOTONIZE, but still need to present the NEXTTOWARDZERO subroutine. Let $x \in \mathbb{F}$, then NEXTTOWARDZERO(x) shall compute $\operatorname{pred}(x)$ if $x > 0$ and $\operatorname{succ}(x)$ if $x < 0$. Following Lemma 4.11, $|x| > \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta$ in this case. NEXTTOWARDZERO is a slight modification of Algorithm 2.12 by Rump et al. [**92**] to simultaneously compute predecessor and successor of a

floating-point number. We replace $|f|$ by f in the computation of e, to always compute the next number in direction towards zero instead of towards $-\infty$. This gives the desired result by symmetry of $\mathbb{F}$ and **fl**. The part handling denormalized numbers is not necessary, but numbers near $\frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta$ still need special care.

Algorithm 4.17 (NextTowardZero).
Let $f \in \mathbb{F}$ with $|f| > \frac{1}{2}\varepsilon_{\mathtt{m}}^{-1}\eta$. If $f > 0$, then NextTowardZero(f) *returns* pred(f)*, otherwise* succ(f) *is returned.*

1: ***procedure*** NextTowardZero(f)
2: $\quad \psi \leftarrow \varepsilon_{\mathtt{m}}(1 + 2\varepsilon_{\mathtt{m}}) = \mathrm{succ}(\varepsilon_{\mathtt{m}})$
3: $\quad$ ***if*** $|f| \geq \frac{1}{2}\varepsilon_{\mathtt{m}}^{-2}\eta$ ***then***
4: $\quad\quad e \leftarrow \psi \otimes f$
5: $\quad\quad p \leftarrow f \ominus e$
6: $\quad$ ***else***
7: $\quad\quad F \leftarrow \varepsilon_{\mathtt{m}}^{-1} f$
8: $\quad\quad e \leftarrow \psi \otimes F$
9: $\quad\quad p \leftarrow \varepsilon_{\mathtt{m}}(F \ominus e)$
10: $\quad$ ***return*** p

4.2.2. Joining the Mantissae of Monotone Expansions. In this section we present C++ code to convert `binary64` expansions into MPFR numbers. We concentrate on this special case, since the algorithms involve bit manipulation and many of the important details are actually harder to describe in pseudocode. The principles do however carry over to other IEEE 754-2008 formats, programming languages, and bigfloat numbers.

Hardware Representation of `binary64`. On many contemporary architectures, the C++ type `double` is an IEEE 754-2008 number in `binary64` format. For conversion to bigfloat numbers, we need direct access to the sign, the exponent and the mantissa. The floating-point standard fixes the storage format, which enables platform independent access. Code similar to the one below is present in other projects, for example the MPFR library. Using the `union` keyword, we interpret a `double` d as 64 bit unsigned integer l.

⟨*direct access to double representation*⟩≡
```
union ieee_binary64{
  double d;
  unsigned long long l;

  ⟨raw bit access⟩
  ⟨demangled access⟩
};
```
The placement of the individual data of d in l is shown in Figure 4.7. A `double` stores one bit for the sign, 11 bits for the exponent and 52 bits for a part of the mantissa, called fraction. The following methods directly return this data as integers, which we denote by s_b, e_b and f respectively.

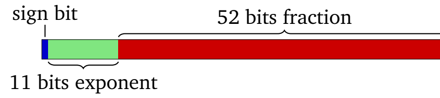


FIGURE 4.7. The IEEE 754-2008 `binary64` format.

⟨*raw bit access*⟩≡
```
  inline long sign_b(){
    return (l & 0x8000000000000000ULL) >> 63;
  }

  inline long exponent_b(){
    return (l & 0x7FF0000000000000ULL) >> 52;
  }

  inline unsigned long long fraction(){
    return (l & 0x000FFFFFFFFFFFFFULL);
  }
```
A triple $(s_b, e_b, f) \in \{0,1\} \times \{0,\dots,2^{11}-1\} \times \{0,\dots,2^{52}-1\}$ represents the following `double` d. If $e_b = 0$ then d is zero or denormalized non-zero:

$$d = (-1)^{s_b} \times f \times 2^{-1074}$$

If $0 < e_b < 2^{11} - 1$ then d is a normalized number:

$$d = (-1)^{s_b} \times (2^{52} + f) \times 2^{e_b - 1075}$$

If $e_b = 2^{11} - 1$ then $d \notin \mathbb{F}$:

$$f = 0 \quad \Rightarrow \quad d = (-1)^{s_b} \times \infty$$
$$f \neq 0 \quad \Rightarrow \quad d = \mathtt{nan}$$

These raw numbers are however not sufficient for our needs. Assuming $d \notin \{0, \mathtt{nan}, \pm\infty\}$, we need integral sign s, mantissa m and exponent e such that

$$d = s \times m \times 2^e. \tag{4.7}$$

The difficulty is, to handle normalized and denormalized numbers uniformly. By casting e_b to `bool` and back, we map non-zero e_b to one and zero e_b to zero. This allows us to compute the other quantities with simple integer arithmetic and avoid branching.

⟨*demangled access*⟩≡
```
  inline bool normalized(){
    return static_cast<bool>(l & 0x7FF0000000000000ULL);
  }
```

If $d \notin \{0, \mathtt{nan}, \pm\infty\}$, the following functions return s, e, m satisfying Equation (4.7).

⟨*demangled access*⟩+≡
```
inline long sign(){
  return 1-2*sign_b();
}

inline long exponent(){
  const long norm = static_cast<long>(normalized());
  return eb - 1075 + norm;
}

inline unsigned long long mantissa(){
  const unsigned long long norm =
    static_cast<unsigned long long>(normalized());
  return ( (norm << 52) + fraction() );
}
```

Representation of mpfr Bigfloat Numbers. An MPFR number consists of the following four fields (excerpt from `mpfr.h`).

⟨*mpfr.h*⟩≡
```
typedef struct {
  mpfr_prec_t  _mpfr_prec;
  mpfr_sign_t  _mpfr_sign;
  mpfr_exp_t   _mpfr_exp;
  mp_limb_t   *_mpfr_d;
} __mpfr_struct;
```

For $0 \neq x \in \mathbb{R}$ the fields have the following semantics.

⟨*mpfr.h*⟩+≡
```
/*
 The represented number is
    _sign*(_d[k-1]/B+_d[k-2]/B^2+...+_d[0]/B^k)*2^_exp
 where k=ceil(_mp_prec/GMP_NUMB_BITS) and B=2^GMP_NUMB_BITS.

 For the msb (most significant bit) normalized representation,
 we must have _d[k-1]>=B/2, unless the number is singular.

 We must also have the last k*GMP_NUMB_BITS-_prec bits set to zero.
*/
```

Thus, the mantissa is stored in `_mpfr_d`, in limbs of `GMP_NUMB_BITS` bits each. Limbs at a higher index store more significant bits and the most significant bit of the most significant limb must be non-zero and has value 0.5. Hence, the mantissa is interpreted as a number in $[0.5, 1)$. Sign, exponent, and precision are stored in `_mpfr_sign`, `_mpfr_exp`, and `_mpfr_prec` respectively. The cases $x \in \{0, \mathtt{nan}, \pm\infty\}$ correspond to `_mpfr_exp` being near the smallest number representable by `mpfr_exp_t`. To set a number to zero, we use an MPFR function call and

we never set a number to nan or $\pm\infty$. Exponents arising in expansions are always in the safe range of mpfr_exp_t.

To set an MPFR number to some $0 \neq x \in \mathbb{R}$ we first set the precision by calling mpfr_set_prec(). This allocates the array _mpfr_d with sufficient entries. Then we write _mpfr_sign, _mpfr_exp, and the entries of _mpfr_d. Thus, we only rely on our knowledge of the representation of non-zero, real numbers in MPFR.

EXPANSIONS WITH ONE SUMMAND. For warmup we discuss converting a double d into an MPFR number. Similar, but more general code is available in MPFR. We restrict ourselves to the case $d \in \mathbb{F}$ and always convert d exactly, i.e., omit the possibility to reduce the precision by rounding. We use one 64 bit integer to access the bits of d, instead of two 32 bit integers and our code can be inlined. Quite surprisingly, all these small differences enable our code to be slightly faster than the corresponding MPFR code, as we observe in Section 4.2.3.

Algorithm 4.18.
Let $d \in \mathbb{F}$*, then* mpfr_set_double(rop,d) *computes an* MPFR *number* rop *with* rop $= d$.

⟨*convert a double to mpfr*⟩≡
```
inline void mpfr_set_double(mpfr_t rop,const double d){
  assert(ra_isfinite(d));
  ⟨handle zero and set precision⟩
  ⟨set sign and exponent⟩
  ⟨set mantissa⟩
}
```

First we handle the case $d = 0$ using an MPFR function call. If $d \neq 0$ we set the precision of rop to 53 bits, allocating the mantissa.

⟨*handle zero and set precision*⟩≡
```
if(d == 0.0){
  mpfr_set_ui(rop,0,GMP_RNDN);
  return;
}
mpfr_set_prec(rop,53);
```

If d is normalized, the most significant non-zero bit in the mantissa, as returned by ieee_binary64, has value 2^{52}. The leading bit of an MPFR mantissa must be non-zero and has value 0.5. Therefore, we have to adjust the exponent by 53. In case d is denormalized, we normalize it by multiplying with 2^{53}, to ensure that the most significant bit is non-zero. In this case, the necessary exponent corrections cancel.

⟨*set sign and exponent*⟩≡
```
ieee_binary64 X;
X.d = d;
rop->_mpfr_sign = X.sign();
if(X.normalized()){
  rop->_mpfr_exp  = X.exponent()+53;
```

```
}else{
  double p = 9007199254740992.0;
  assert(p == ldexp(1.0,53));
  X.d *= p;
  assert(X.normalized());
  rop->_mpfr_exp  = X.exponent(); //-53+53;
}
```

Since d in X is now normalized, the most significant non-zero bit in the mantissa is the 53rd bit. Hence, in case of 64 bit limbs we have to shift 11 bits to the left. For 32 bit limbs, the mantissa overlaps two limbs and different shifting is needed, cf. m' in Figure 4.8.

⟨*set mantissa*⟩≡

```
#if GMP_NUMB_BITS==32
  rop->_mpfr_d[1] = (X.mantissa() >> 21);
  rop->_mpfr_d[0] = (X.mantissa() << 11);
#else //GMP_NUMB_BITS==64
  rop->_mpfr_d[0] = (X.mantissa() << 11);
#endif
```

Expansions with more Summands. We are now ready to give an algorithm for the conversion of monotone expansions with two or more summands. The main loop of our algorithm, handling all but the first summand, is free from branches.

Algorithm 4.19.
Let $e = e_0, e_1, \ldots, e_{n-1}$ *be a monotone expansion, then* `mpfr_set_monotone_expansion(rop,n,e)` *computes an* MPFR *number* `rop` *with* `rop` $= \sum_{i=0}^{n-1} e_i$.

⟨*convert a monotone expansion to mpfr*⟩≡

```
inline void
mpfr_set_monotone_expansion(mpfr_t rop,
                            const int n ,const double *const e){
  ⟨handle less than two summands⟩
  ⟨set up precision, sign and exponent⟩
  ⟨clear mantissa and write first summand⟩
  ⟨write remaining summands⟩
}
```

First we handle the cases $n \in \{0, 1\}$. If the expansion has at least two summands, all summands are non-zero and we can use `ieee_binary64` safely.

⟨*handle less than two summands*⟩≡

```
if(n == 0){
  mpfr_set_ui(rop,0,GMP_RNDN);
  return;
}else if(n == 1){
  mpfr_set_double(rop,e[0]);
  return;
}
```

The sign and exponent of rop are determined by the leading summand e_{n-1} and are computed as above in Algorithm 4.18. The exponent maxexp of rop can be interpreted as pointing directly in front of the most significant non-zero bit of e. The exponent minexp of e_0 points somewhere behind the least significant non-zero bit, so it is sufficient to set the precision to maxexp − minexp. In case e_{n-1} is not normalized, this value may be smaller than 53 and even as low as 1. Therefore, we adjust minexp to fulfill the minimal precision requirements of MPFR. Setting the precision allocates a mantissa with $k = \lceil(\texttt{maxexp} - \texttt{minexp})/\texttt{GMP_NUMB_BITS}\rceil$ limbs. We reuse maxexp to align the remaining summands to the mantissa of rop. Figure 4.8 shows the alignment of the first and later summands on the mantissa of rop for the case of 32 bit limbs.

⟨*set up precision, sign and exponent*⟩≡

```
ieee_binary64 X;
X.d = e[0];
mp_exp_t minexp = X.exponent();

X.d = e[n-1];
mp_exp_t maxexp = X.exponent()+53;

if(!X.normalized()){
  const double d = 9007199254740992.0;
  assert(d == ldexp(1.0,53));
  X.d *= d;
  assert(X.normalized());
  maxexp = X.exponent(); //+53-53;
  minexp = std::min(minexp,maxexp-MPFR_PREC_MIN);
}

const mpfr_prec_t prec = maxexp-minexp;
assert(prec >= MPFR_PREC_MIN);
mpfr_set_prec(rop,prec);
assert(rop->_mpfr_prec == prec);

const int k = (prec+GMP_NUMB_BITS-1)/GMP_NUMB_BITS;
assert( (k-1)*GMP_NUMB_BITS < prec && prec <= k*GMP_NUMB_BITS );

rop->_mpfr_sign = X.sign();
rop->_mpfr_exp  = maxexp;
mp_limb_t *const mant = rop->_mpfr_d;
```

Writing the mantissa of e_{n-1} is again done as in Algorithm 4.18. The lower limbs of rop may be overlapped by multiple mantissae, however in all but one mantissa the overlapping bits are zero, cf. again Figure 4.4. We zero the lower limbs of rop so we can write the remaining summands by bitwise or.

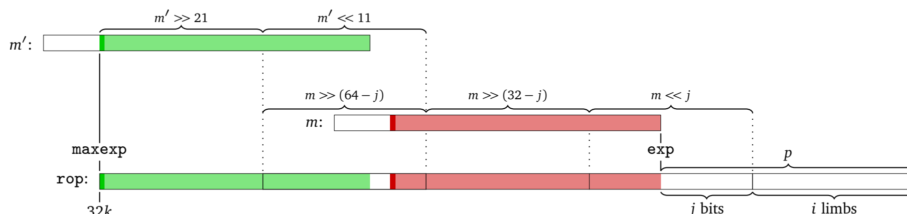


FIGURE 4.8. Aligning the mantissa of the leading summand m' and any other mantissa m to a 32 bit limb arbitrary precision mantissa.

⟨*clear mantissa and write first summand*⟩≡

```
#if GMP_NUMB_BITS==32
  mant[k-1] = (X.mantissa() >> 21);
  mant[k-2] = (X.mantissa() << 11);
  for(int i=k-3;i>=0;--i) mant[i] = 0;
#else //GMP_NUMB_BITS==64
  mant[k-1] = (X.mantissa() << 11);
  for(int i=k-2;i>=0;--i) mant[i] = 0;
#endif
```

We iterate over the remaining summands and copy each mantissa. This loop is completely free of branches, including computations done by `ieee_binary64`.

⟨*write remaining summands*⟩≡

```
int i = n-2;
while(i>=0){
  const double wrt = e[i--];
  ⟨write summand⟩
}
```

For each summand we compute the position p of the last bit of its mantissa m in the mantissa of `rop`, cf. Figure 4.8. Then we compute the index i of the limb this bit belongs to and the number of bits j it has to be shifted to the left.

⟨*write summand*⟩≡

```
{
  X.d = wrt;
  mpfr_exp_t exp = X.exponent();
  unsigned long long m = X.mantissa();

  const int p = GMP_NUMB_BITS*k - maxexp + exp;
  assert(p >= 0);

  const int i = p/GMP_NUMB_BITS;
  const int j = p%GMP_NUMB_BITS;
  assert(0 <= i && i < k);
```

```
#if GMP_NUMB_BITS==32
   ⟨write mantissa to 32 bit limbs⟩
#else //GMP_NUMB_BITS==64
   ⟨write mantissa to 64 bit limbs⟩
#endif
}
```

For 32 bit limbs, the mantissa m may overlap up to three limbs. While index i always points to an existing limb, the limbs at position $i+1$ and $i+2$ may not exist, i.e., we are out of array bounds. In this case, however, the part m_1 of m overlapping this non-existing limb is zero. Instead of branching whether we are outside bounds, we use m_1 to recalculate the index. If $m_1 = 0$ then $i_1 = 0$ and otherwise $i_1 = i+1$. Since we write the mantissa using bitwise `or`, no harm is done in the first case and the mantissa is written correctly in the other case.

⟨write mantissa to 32 bit limbs⟩≡

```
const unsigned long long m1 = (m  >> (32 - j));
const unsigned long long m2 = (m1 >> 32);

const int i1 = (i+1) * static_cast<bool>(m1);
const int i2 = (i+2) * static_cast<bool>(m2);

assert(i+1 < k || m1 == 0);
assert(i+2 < k || m2 == 0);

mant[i]  |= (m << j);
mant[i1] |= m1;
mant[i2] |= m2;
```

For 64 bit limbs, the mantissa may overlap up to two limbs. We use the same trick as above to avoid branching when out of array bounds. But there is another issue. Shifting m by 64 bits will quite unintuitively leave m unchanged, instead of resulting in $m = 0$. Since j may be zero, we perform the right shift in two steps.

⟨write mantissa to 64 bit limbs⟩≡

```
const int j1 = (64 - j) >> 1;
const int j2 = (64 - j) - j1;

const unsigned long long m1 = ((m >> j1) >> j2);
const int i1 = static_cast<bool>(m1)*(i+1);

assert(i+1 < k || m1 == 0);

mant[i]  |= (m << j);
mant[i1] |= m1;
```

We implement code similar to the one presented here for the conversion of monotone expansions to `leda::bigfloat` numbers [**70, 69**]. The internal storage format and normalization conditions of `leda::bigfloat` are quite different from those of

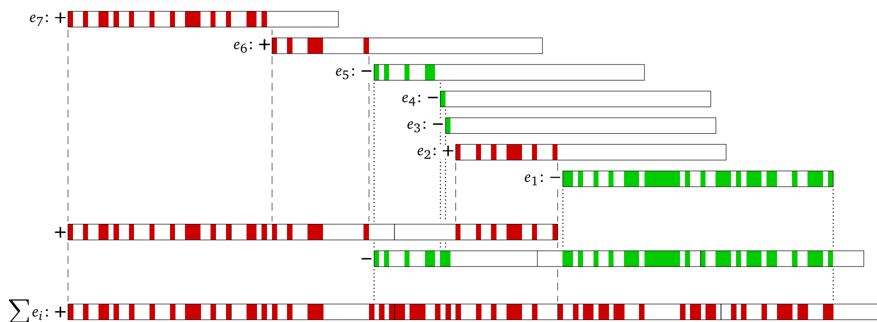


FIGURE 4.9. Conversion of an expansion into a bigfloat number by splitting.

MPFR. The mantissa is however stored in essentially the same fashion as an array of limbs and it is reasonable to believe that other bigfloat number types follow that convention as well [**110**]. Once exponent and sign are set and the first mantissa is placed, the remaining summands can be handled in the way described above.

4.2.3. Comparison to Straightforward Conversion. Using the algorithms from above, we can convert a general expansion into a bigfloat number by first transforming it into an equivalent maximally non-overlapping monotone expansion and then joining the mantissae of the new summands into a bigfloat number. This option is illustrated in Figure 4.5. Alternatively, we may split the expansion into two monotone expansions, convert them separately and then perform a single exact addition of two bigfloat numbers. This option is illustrated in Figure 4.9. While overflow may occur in the first conversion strategy for very large expansions, the second option is free from overflow on principle, since there are no floating-point operations involved. It may hence serve as a backup strategy.

For both MPFR and LEDA we tried several direct conversion approaches, using functionality provided by the library. Among the things we tried are summing by increasing or decreasing magnitude, increasing the output precision with each summand or setting sufficient output precision before starting to sum up, as well as some library specific approaches. MPFR has a function `mpfr_sum()` for computing the sum of several MPFR numbers in one step, while LEDA supports exact addition, i.e., may internally compute and set the precision sufficient for an addition to be exact. For each library we selected the two fastest direct approach for comparison.

As input data, we use randomly generated expansions, which we create by evaluating a polynomial expression using Shewchuk's arithmetic operations. This way, test expansions are more likely to have a structure that actually occurs in applications. As expression D we compute a 4×4 determinant of 4×4 determinants of randomly generated numbers. D has a polynomial degree of $d = 16$. Using the `rand48` family of random number generators, we select a floating-point number $f \in [0,1]$, a sign $s \in \{-1,+1\}$ and an exponent $e \in \{-17,\dots,17\}$ and use $s \times f \times 2^e$

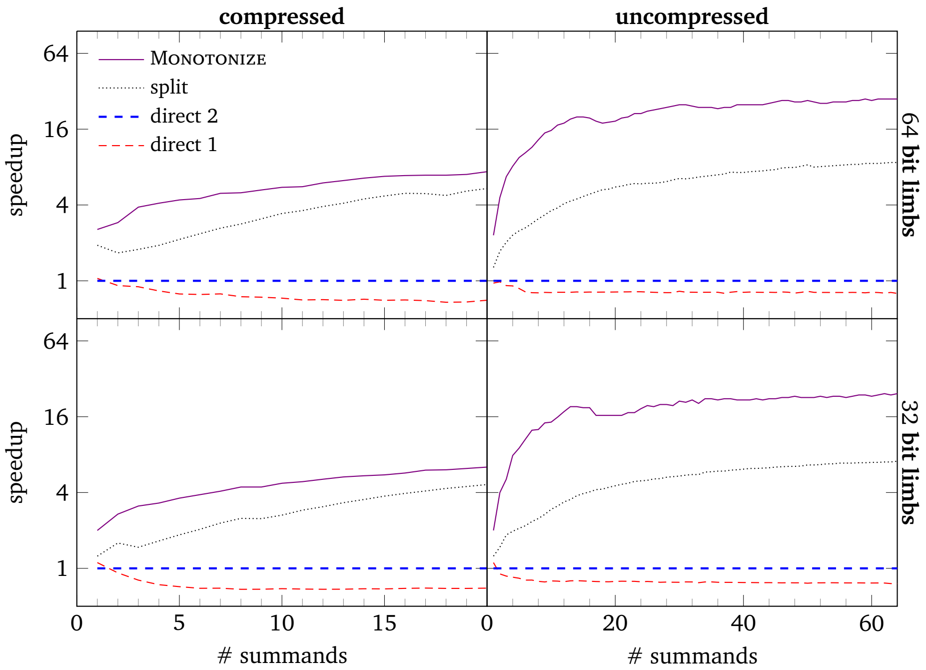


FIGURE 4.10. Converting expansions to MPFR. Figures show the speedup for each conversion method, relative to direct 2 - - -, on a logarithmic scale.

as input number. All our input numbers can be uniformly scaled to integers with $p = 53+35$ bit precision. Hence, D can be represented with approximately $dp \approx 1400$ bits. We observed however that D was in fact representable with approximately 1000 bits only on average.

The evaluation of D gives us strongly non-overlapping expansions with about 220 summands on average. If we additionally compress the sums, we get non-adjacent expansions with about 20 summands on average. In a compressed expansion, each summand carries about 52 bits of information. Therefore, before compressing, each summand carries less than 5 bits of information on average. Thus, uncompressed and compressed expansions are input sets with quite different characteristics. This is particularly relevant, since MONOTONIZE compresses as a side effect and hence may reduce the number of summands significantly before the actual conversion step. We consider expansions with $n = 1, 2, \ldots, 64$ summands in the uncompressed case and $n = 1, 2, \ldots, 20$ in the compressed case. To generate expansion with fewer summands than originally created, we simply ignore leading summands.

We run experiments with a limb size of both 64 bit and 32 bit on the `descartes` platform. For 32 bit limbs we use a slightly different setup. Code is compiled with

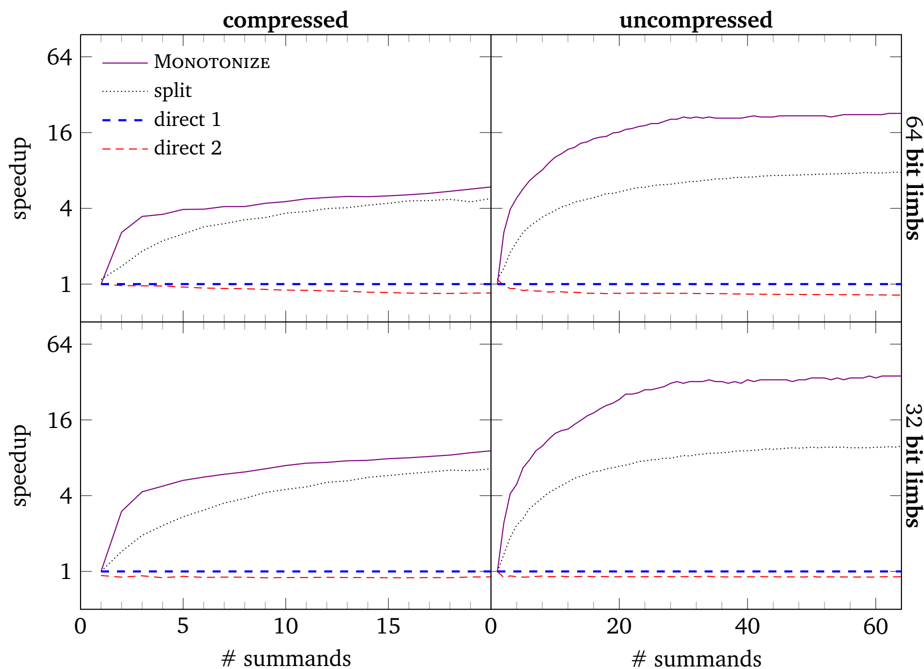


Figure 4.11. Converting expansions to `leda::bigfloat`. Figures show the speedup for each conversion method, relative to direct 1 - - -, on a logarithmic scale.

the `-m32` flag of g++, since neither MPFR nor LEDA support 32 bit limbs in a 64 bit environment and we use libraries GMP 5.0.2, MPFR 3.0.1, and LEDA 6.3. To get measurable running times, we generate 2000 expansions as described above, and measure the total time for converting all expansions a 1000 times. The results are shown in Figure 4.10 and Figure 4.11. The graphs do not show running time but the speedup for each method with respect to the faster direct approach. This improves the display of differences for the important range with very few summands, where the actual running times are very small.

Both, conversion by Monotonize and conversion by splitting clearly outperform the direct approaches, and conversion by Monotonize is uniformly the fastest method. The speedup is about the same for 32 and 64 bit limbs. As expected, Monotonize achieves greater speedup for uncompressed than compressed expansions. For MPFR, there is a small local minimum in the speedup achieved by Monotonize for uncompressed expansions, near the mark of 20 summands. This roughly corresponds to our observation that summands carry less than 5 bits of information. Near this range the number of output summands of Monotonize jumps from one to two. For

MPFR, our new approaches are strictly faster, even in the case of one summand only. Both direct approaches simply call `mpfr_set_d()`, while our new approaches use Algorithm 4.18. Hence we use Algorithm 4.18 in `Mpfr_approximation_policy`.

Our new conversion methods are so much faster than computing the sum straightforwardly, that they may help to speed up the exact computation of arbitrary sums, not only expansions, too. As an intermediate step, one has to convert the sum into an expansion first. This may be done using the divide and conquer distillation procedure by Shewchuk [**104**], described in Section 2.2.3, or an extension of the AccSum algorithm by Rump et al. [**95**].

CHAPTER 5

Exact Floating-Point Algorithms in *RealAlgebraic*

In this chapter we discuss, implement and evaluate strategies to integrate error-free transformations into an expression dag based number type on the example of *RealAlgebraic*. We have seen in Section 4.1.3, that exact floating-point algorithms based on error-free transformations can lead to very fast and exact implementations of arithmetic. But error-free transformations also have quite a few limitations. Basically, only ring operations can be implemented exactly and even these may fail in case overflow or underflow occurs.

Utilizing error-free transformations directly is far from simple or straightforward. To allow a non-expert to benefit from their efficiency, it is necessary to wrap them into a more user-friendly solution, for example a number type. The main goal of expression dag based number types is to integrate fast evaluation strategies, which may occasionally fail to compute a sign, with more expensive but also more conservative strategies, into a user-friendly solution that eventually computes all signs correctly. Error-free transformations, with their high speed but also their apparent limitations fit well into this scheme.

In the first part of this chapter, we discuss our *LocalPolicy* model `Local_double_sum`, which places exact arithmetic based on error-free transformations at the very first evaluation stage. This way, dag creation is avoided and deferred to the point where error-free transformations can not provide an exact result for an operation. Since there are many ways to implement exact arithmetic based on error-free transformations, `Local_double_sum` is again configurable by several policies. We present our concepts and models, and the rationale behind them.

In the second part, we compare different variants of `Local_double_sum` by means of experiments, with the goal of finding an optimal `Local_double_sum` variant. Our findings are, that in general, the choice of implementation parameters depends on the geometric problem solved. We therefore give guidelines, how one may choose these parameters a priori without resorting to experiments and give a default variant, which should give good performance in most circumstances.

We then compare the default `Local_double_sum` variant to other *RealAlgebraic* variants and other expression dag based number type. It turns out that placing exact arithmetic based on error-free transformations at the very first evaluation stage, improves performance for geometric problems and input data with many near degenerate configurations but is less advisable for non-degenerate data. Therefore

we close with a short discussion how error-free transformations may be integrated into the later stages of expression dag evaluation.

5.1. Deferring Dag Construction

In this section we catch up on the discussion of *LocalPolicy* model `Local_double_sum`, which we omitted previously in Chapter 3. The goal behind `Local_double_sum` is to let *RealAlgebraic* benefit from the speed of error-free transformations and exact floating-point algorithms. To this end, `Local_double_sum` represents a number as the sum of a sequence of floating-point numbers and provides basic arithmetic operations on this representation, as required by the *LocalPolicy* concept. Just like *RealAlgebraic* itself, `Local_double_sum` is configurable by means of policies, where each policy reflects a single implementation alternative. The design and implementation are based on the following considerations.

When we first started work on `Local_double_sum`, our *RealAlgebraic* implementation was much slower that exact predicate implementations based on error-free transformations for all types of input data. Therefore we decided to bring error-free transformations to the very first evaluation stage, i.e., on the *LocalPolicy* level and before dag creation. One source of overhead in the first evaluation stages is dynamic memory management for dag node creation. But if the floating-point filter can determine the sign, dag creation was pointless. By deferring dag creation we postpone and potentially eliminate this source of overhead. To avoid memory management within `Local_double_sum`, too, we limit the number of summands to some small constant. The decision to represent a number as a sum of floating-point numbers does not permit operations such as division or radicals. For the remaining operations addition, subtraction, and multiplication it is far from clear how to implement them optimally. Finally, a framework like *RealAlgebraic* promises its user nearly unconditional correctness of computations, i.e., the only limit should be memory or time constraints. Inexactness arising from overflow or underflow are not acceptable and therefore must be taken care of without the user noticing. With this in mind, we identify four orthogonal design decisions which are reflected in four concepts that govern the behavior of our implementation.

***DoubleSumOperations*:** Models for this concept provide a set of raw operations on sums of floating-point numbers, namely ring operations, sign computation and compression. A compression transforms a sum of floating-point numbers into an equivalent sum with potentially fewer summands.

***DoubleSumMaxLength*:** Models provide a limit on the number of summands allowed.

***DoubleSumCompression*:** Models decide when and how to apply compression to reduce the number of summands.

***DoubleSumProtection*:** Models provide a systematic way to handle overflow or underflow.

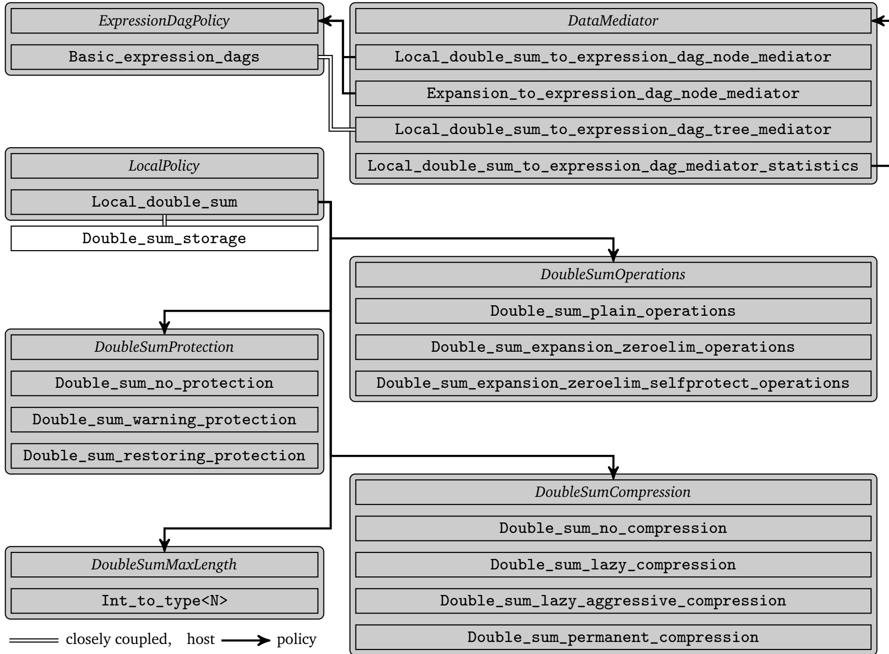


FIGURE 5.1. Concepts and models in `Local_double_sum`.

These four policies are complemented by the *DataMediator* policy of *RealAlgebraic*. Contrary to the other *LocalPolicy* models, for `Local_double_sum` there are several interesting ways to convert a sum of floating-point numbers into an expression dag. Here we benefit from our earlier decision to separate conversion to an expression dag from the *LocalPolicy* concept. An overview of the five relevant concepts and their models is given in Figure 5.1, which complements Figure 3.1.

5.1.1. Basic Arithmetic Operations. Our decision to implement a *LocalPolicy* based on error-free transformations, as well as the decision of how to implement arithmetic operations is based on the experiments from Section 4.1.3. We have the following three models for *DoubleSumOperations*.

`Double_sum_plain_operations` (`plaiO`): This model is build around the SIGNK algorithm, which leads to the fastest predicates in Section 4.1.3. Arithmetic operations are implemented as plainly as possible. Addition and subtraction simply copy summands, the multiplication performs TWOPRODUCT for all pairs of summands. To compress a sum, we perform VECSUM, but additionally eliminate zero summands on the fly. Sign computation is performed using SIGNK.

In Figure 4.3, sign computation based on ACCSUM shows a performance similar to SIGNK, however SIGNK is better suited in our case for practical reasons.

SIGNK improves the representation of a sum with each VECSUM step and never increases the number of summands. We can thus let it work on the original sum. ACCSUM on the other hand produces an additional summand in the extraction step, which complicates its use in a framework with a limited number of summands.

`Double_sum_expansion_zeroelim_operations` **(**`expa0`**):** This model is based on arithmetic with floating-point expansions as described by Shewchuk [**104, 105**]. We maintain zero-free, strongly non-overlapping expansions. The basic algorithms for arithmetic on this type of expansions are FASTEXPANSIONSUM, the corresponding subtraction routine, and SCALEEXPANSION. We use an implementation of these algorithms that additionally eliminates zero summands on the fly. We implement missing functionality, i.e., operations involving expansions with only one summand and a general multiplication routine following suggestions by Shewchuk [**104**, section 2.8]. For compression, we use COMPRESS. The sign of a zero-free expansion is always the sign of the most significant summand and can simply be read of.

`Double_sum_expansion_zeroelim_selfprotect_operations` **(**`prot0`**):** This model derives from `expa0` and performs all operations in exactly the same manner, but is free from overflow and underflow. We discuss the way this is achieved in Section 5.1.3 below.

Note that the models employ two opposing strategies. With plain sums, arithmetic operations are lazy, at the cost of unstructured sums and potentially many summands. Compression and sign computation do all the work. With expansions, a normal form is maintained by arithmetic operations. This makes them more expensive, but also reduces the number of summands. The sign of an expansion can be determined at no extra cost!

5.1.2. Number of Summands and Compression. Arithmetic operations on sums of floating-point numbers rapidly increase the number of summands. For polynomial expressions c over floating-point numbers, define $\#\mathrm{s}(c)$ by $\#\mathrm{s}(f) = 1$ for $f \in \tilde{\mathbb{F}}$, and

$$\#\mathrm{s}(a \pm b) = \#\mathrm{s}(a) + \#\mathrm{s}(b), \qquad \#\mathrm{s}(a \times b) = 2\#\mathrm{s}(a)\#\mathrm{s}(b).$$

Then $\#\mathrm{s}(c)$ is an upper bound on the number of floating-point summands required to represent c. Note that $\#\mathrm{s}(c)$ grows exponentially in the degree of the expression c. The two individual rules are sharp in general. To see this in case of the addition, let

$$e = \left(\varepsilon_{\mathtt{m}}^{-2i}\right)_{i=1}^{n}, \qquad f = \left(\varepsilon_{\mathtt{m}}^{-2n-2j}\right)_{j=1}^{m}.$$

Then the sequence $e_1, e_2, \ldots, e_n, f_1, f_2, \ldots, f_m$ is a non-adjacent and maximally non-overlapping expansion, representing $e + f$ and no sequence representing $e + f$ with fewer summands exists. To show the claim for the multiplication, we have to choose the e_i and f_j such that the product $e_i \times f_j$ requires two summands for storage.

Furthermore, we have to space the e_i and f_j sufficiently, such that the mn individual products do not interfere with each other. This is for example achieved by setting

$$e = \left(\varepsilon_{\mathtt{m}}^{-3i}(1+2\varepsilon_{\mathtt{m}})\right)_{i=1}^{n}, \qquad f = \left(\varepsilon_{\mathtt{m}}^{-4nj}(1+2\varepsilon_{\mathtt{m}})\right)_{j=1}^{m}.$$

The arithmetic operations implemented in `plaiO` attain the given bounds. We know on the other hand, that polynomial expressions of degree d over b bit integers can be evaluated exactly with $d(b+O(1))$ bit precision [**56**]. This translates to roughly $(d \times b)/p$ summands, which is linear in the degree d. The arithmetic operations of `expaO` may show better behavior than predicted by #s in many cases, but we already observed in Section 4.2.3, that without compression, expansions often carry only a few bits of information per summand.

For these reasons, it is likely that often a more compact representation of a number with fewer summands can be computed. Fewer summands make further operations on a number cheaper and, since we limit the maximum number of summands, enable us to defer dag creation for larger expressions. Since it is unclear when to attempt compression in an optimal way, we implement the following schemes.

`Double_sum_no_compression` (`noC`): No compression is triggered.

`Double_sum_lazy_compression` (`lazyC`): Triggers a single compression step on the operands to an arithmetic operation, if the number of summands of the result, as predicted by #s, is larger than the maximum number of summands.

`Double_sum_lazy_aggressive_compression` (`laagC`): Initially behaves like `lazyC`, but triggers additional compression steps as long as the number of summands was decreased in the previous step.

`Double_sum_permanent_compression` (`permC`): Triggers a single compression step on each result of an arithmetic operation.

From top to bottom, these policies provide an increasing amount of additional compression. Due to the different approaches in our *DoubleSumOperations* models, we can expect, that `plaiO` will benefit more from additional compression than `expaO`.

5.1.3. Handling Floating-Point Exceptions. Error-free transformations are, despite their name, not completely free from errors or inexactness. Any such error or inexactness is however linked to a floating-point exception. Since we consider polynomial expressions only, and make sure that input numbers are always elements from $\mathbb{F}$, underflow and overflow are the only relevant exceptions to us.

The IEEE 754 standard requires the availability of a set of flags that are raised when an exception occurs and may be checked and reset by the user. This provides us with a way to be notified of floating-point exceptions after the fact. Based on this mechanism we provide the following three policies for handling floating-point exceptions. For these to be effective, a basic condition is that operations terminate in all cases, especially if floating-point exceptions occur. Hence our rigorous discussion that SIGNK always terminates.

`Double_sum_restoring_protection` **(**`restP`**):** This model stores a backup of any sum about to be overwritten by an operation. After the operation, it checks the exception flags. If an exceptions occurred, it resets the exception flags and restores the sum. Eventually, the operation will be forwarded to the expression dag. In this way, floating-point exceptions do not affect the correctness of a computation and are invisible to the user, as intended.

`Double_sum_warning_protection` **(**`warnP`**):** This model resets the exception flags prior to an operation and checks them afterwards. If an exception occurs, an error handler is called. Unlike the previous model, this one is free from false positives, but requires user interaction to handle the situation.

`Double_sum_no_protection` **(**`noP`**):** This model does not protect from floating-point exceptions in any way. It allows us to determine the cost of the former alternatives.

Since accessing exception flags and making backup copies can be quite expensive, we do not employ the protection mechanisms indiscriminately. For example, addition and subtraction in `plaiO` do not perform floating-point operations and are therefore free from floating-point exceptions. The same holds for the sign computation in `expaO`. Therefore, all operations in *DoubleSumOperations* are accompanied with a tag indicating whether floating-point exceptions may actually occur, allowing to enable or disable protection from floating-point exceptions at compile time.

No overflow in Compress. Another operation that is free from floating-point exceptions in our case is the Compress algorithm in `expaO`. Compress uses only floating-point additions and subtractions, it is thus unaffected by underflow. Less obviously, it is free from overflow when run on a strongly non-overlapping expansion. Intuitively, compression does not change the value of the sum and due to the non-overlapping property all bits and especially the largest bit are already present in the input expansion. Overflow can only occur when Compress generates larger non-zero bits than present in the input expansion.

Lemma 5.1. *Assume floating-point arithmetic over $\mathbb{F}$, with rounding to nearest and tie-breaking to even. Let $e = e_1, e_2, \ldots, e_m$ be a strongly non-overlapping expansion. Then, no overflow occurs in the call* Compress(e).

This property does not hold for non-overlapping expansions. If we consider the sequence of bits of a strongly non-overlapping expansion, there must be a zero bit at least every $p + 1$ bits [**104**]. Running Compress with the largest strongly non-overlapping expansion over $\mathbb{F}$, no overflow occurs. But if the number is made just a bit larger, i.e., one of the zero bits is set to non-zero and all less significant bits are set to zero, overflow does occur. See Figure 5.2 for an illustration. Our proof makes explicit use of tie-breaking to even. Intuitively, the zero bit every $p + 1$ bits catches any carry coming from less significant summands, thus avoiding a carry on the largest summand. The tie-breaking to even rule places this zero bit conveniently at the transition between two summands, which we exploit in our proof. We conjecture that a proof is possible without appealing to the tie breaking rule. But, since tie breaking

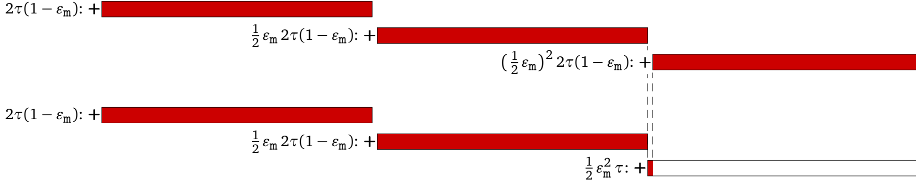


FIGURE 5.2. In COMPRESS, no overflow occurs on the upper expansion, but on the lower one.

to even is needed for strongly non-overlapping expansions and FASTEXPANSIONSUM anyway, this is not relevant in our context.

PROOF. To prove Lemma 5.1, we consider a recursive formulation of COMPRESS that allows an inductive argument. The base case occurs, when $\sum_{i=1}^{m} e_i$ is a floating-point number. Otherwise, COMPRESS sums $e_m, e_{m-1}, \dots$ iteratively until the result is not a floating-point number for the first time. Let this occur when adding e_j, and let $Q = \sum_{i=j+1}^{m} e_i$, then we have g_m and q with

$$g_m = Q \oplus e_j = \mathbf{fl}\left(\sum_{i=j}^{m} e_i\right), \qquad g_m + q = \sum_{i=j}^{m} e_i.$$

COMPRESS keeps g_m as intermediate summand and recursively compresses the sequence $e_1, e_2, \dots, e_{j-1}, q$, obtaining as output the sequence $f_1, f_2, \dots, f_{l-2}, f_{l-1}$. Then it computes f_l and q' with

$$f_l = g_m \oplus f_{l-1}, \qquad f_l + q' = g_m + f_{l-1}.$$

COMPRESS returns one of the expansions $f_1, f_2, \dots, f_{l-2}, q', f_l$, or $f_1, f_2, \dots, f_{l-2}, f_l$, depending on whether q' is non-zero. The computation of both g_m and f_l is at risk from overflow. We show by induction on the recursive calls, that $\operatorname{msb}(f_l) \le \operatorname{msb}(e_m)$ and infer as a side effect that no overflow occurs. We can assume that no input summand is zero, since a zero summand simply results in an iteration of the first loop not changing any values.

The sequence $|e_j|, |e_{j+1}|, \dots, |e_m|$ is a strongly non-overlapping expansion, too. Hence, the binary representation of $E = \sum_{i=j}^{m} |e_i|$ contains a zero bit at least every $p+1$ bits. Together with $\operatorname{msb}(E) = \operatorname{msb}(e_m)$ this yields $E < \operatorname{msb}(e_m)(2 - \varepsilon_{\mathrm{m}})$. In total we have

$$\left|\sum_{i=j}^{m} e_i\right| < \operatorname{msb}(e_m)(2 - \varepsilon_{\mathrm{m}}).$$

The right hand side $\operatorname{msb}(e_m)(2 - \varepsilon_{\mathrm{m}})$ is the smallest number which might be rounded to $2\operatorname{msb}(e_m)$ (or $+\infty$, if $2\operatorname{msb}(e_m) \notin \mathbb{F}$). Therefore, the sum is rounded towards zero, no overflow occurs in the computation of g_m, and $\operatorname{msb}(g_m) \le \operatorname{msb}(e_m)$. Furthermore, g_m and q are non-adjacent by Lemma 2.13 because tie-breaking to even is in effect.

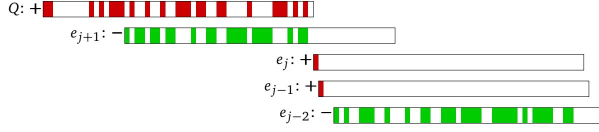


FIGURE 5.3. A critical case in proof of Lemma 5.1.

This discussion also settles the base case of the induction, when $\sum_{i=1}^{m} e_i$ itself is a floating-point number and g_m is returned as compression result.

Now we turn to the induction step. We claim the sequence $e_1, e_2, \ldots, e_{j-1}, q$ is a strongly non-overlapping expansion. If e_j is not adjacent to e_{j-1}, then q is neither, because $q \in \operatorname{lsb}(e_j)\mathbb{Z}$ by Equation (2.9). The claim holds for this case. If e_j is adjacent to e_{j-1}, then both are a power of two. Now consider the computation of $Q \oplus e_j$. The situation is visualized in Figure 5.3. First, Q is not adjacent to e_j, because e_j and e_{j+1} are non-adjacent and $Q \in \operatorname{lsb}(e_{j+1})\mathbb{Z}$. We have $\operatorname{msb}(Q + e_j) \geq \varepsilon_{\mathtt{m}}^{-1} e_j$, because otherwise

$$\operatorname{msb}(Q + e_j) \;\leq\; \tfrac{1}{2}\varepsilon_{\mathtt{m}}^{-1} e_j \;=\; \tfrac{1}{2}\varepsilon_{\mathtt{m}}^{-1} \operatorname{lsb}(Q + e_j)$$

and $Q + e_j \in \mathbb{F}$. With tie-breaking to even, we have $g_m = Q$ and $q = e_j$. This shows, that $e_1, e_2, \ldots, e_{j-1}, q$ is a strongly non-overlapping expansion in this case, too.

By induction, the recursive application of COMPRESS gives us a non-adjacent expansion $f_1, f_2, \ldots, f_{l-1}$ with $\operatorname{msb}(f_{l-1}) \leq \operatorname{msb}(q)$. We compute $f_l = g_m \oplus f_{l-1}$. Because g_m and q are non-adjacent, g_m and f_{l-1} are non-adjacent, too. But then, by the same reasoning applied to the computation of g_m above,

$$|g_m + f_{l-1}| < \operatorname{msb}(g_m)(2 - \varepsilon_{\mathtt{m}}),$$

no overflow occurs in the computation of f_l and $\operatorname{msb}(f_l) \leq \operatorname{msb}(g_m) \leq \operatorname{msb}(e_m)$. □

With this result, we need no protection from floating-point exceptions for the compression step in `expa0`, or `prot0`.

AVOIDING FLOATING-POINT EXCEPTIONS. A common strategy in practice is to avoid overflow and underflow in the first place, by detecting that input consists of very small or very large numbers and take appropriate action, e.g., rescale input numbers to a safe range or at least warn the user.

We implement this approach in `Double_sum_expansion_zeroelim_self-protect_operations`, which derives from `expa0` and provides exactly the same set of operations, but prior to any operation checks the size of input numbers and computes bounds on the output. If the operation is at risk from overflow or underflow, it is not performed but eventually forwarded to the expression dag. No operation in `prot0` requires protection from a *DoubleSumProtection* model.

We already discussed, that sign computation and compression are free from floating-point exception, which leaves us with the ring operations addition, subtraction, and multiplication. Bounding the magnitude of output summands can be done efficiently for expansions because summands are ordered by magnitude.

Let $e = e_1, e_2, \dots, e_m$ be a zero-free, strongly non-overlapping expansion with $E = \sum_{i=1}^{m} e_i$. We want to bound E from above and below in terms of the leading summand e_m. Note that e_m may consist of a single non-zero bit only. We have

$$|E| \le \sum_{i=1}^{m} |e_i| < 2\,\mathrm{msb}(e_m) \le 2|e_m|, \tag{5.1}$$

since $|e_1|, |e_2|, \dots, |e_m|$ is non-overlapping. Finding a lower bound is slightly more involved. We distinguish two cases. First, let $|e_m|$ and $|e_{m-1}|$ be non-adjacent. Applying Equation (5.1) to the expansion $e_1, e_2, \dots, e_{m-1}$, we have

$$|E| \ge |e_m| - \left|\sum_{i=1}^{m-1} e_i\right| > |e_m| - 2\,\mathrm{msb}(e_{m-1}) \ge \tfrac{1}{2}|e_m|. \tag{5.2}$$

The other case is, that $|e_m|$ and $|e_{m-1}|$ are adjacent. Then both are a power of two, and $|e_{m-1}|$ and $|e_{m-2}|$ are non-adjacent. Using Equation (5.2) we have

$$|E| \ge |e_m| - |e_{m-1}| - \left|\sum_{i=1}^{m-2} e_i\right| = |e_{m-1}| - \left|\sum_{i=1}^{m-2} e_i\right| \ge \tfrac{1}{2}|e_{m-1}| = \tfrac{1}{4}|e_m|.$$

Combining these results, we have

$$\tfrac{1}{4}|e_m| \le |E| \le 2|e_m|$$

in any case. Let $f = f_1, f_2, \dots, f_n$, be another zero-free, strongly non-overlapping expansion with $F = \sum_{j=1}^{n} f_j$. We discuss addition and subtraction first, which are free from underflow on principle. Let $H = E + F$. For any summand h in a strongly non-overlapping expansion representing H, we have

$$|h| \le 4|H| \le 4\big(|E| + |F|\big) \le 8\big(|e_m| + |f_n|\big) \le 8(1 + \varepsilon_{\mathtt{m}})\big(|e_m| \oplus |f_n|\big)$$

Therefore, if

$$\big(|e_m| \oplus |f_n|\big) \le \tfrac{1}{8}\tau, \tag{5.3}$$

we have $|h| \le 2\tau(1 - \varepsilon_{\mathtt{m}})$ and $h \in \mathbb{F}$. FastExpansionSum does not generate intermediate summands larger than output summands and is therefore safe from overflow if Equation (5.3) holds. Computing the difference of two expansions is safe from overflow under the same condition.

Multiplication may suffer from both overflow or underflow. We already gave criteria for TwoProduct to be safe from overflow and underflow in Section 2.2.2 and the reasoning for the general multiplication is not much different. Computing the

product of e and f using any combination of ScaleExpansion and FastExpansionSum is free from underflow, if

$$e_1 f_1 = 0 \qquad \text{or} \qquad |e_1||f_1| > \tfrac{1}{2}\varepsilon_{\mathtt{m}}^{-2}\eta, \tag{5.4}$$

since we never leave the ring $\sigma\mathbb{Z}$, where σ is the product of the smallest non-zero bits in e and f. Remember that $e_1 = 0$ implies $m = 1$ and $E = 0$, i.e., there is no need to check other summands in this case. Now let $H = E \times F$ and let h be a summand in a strongly non-overlapping expansion representing H. Then

$$|h| \le 16(1 + \varepsilon_{\mathtt{m}})\big(|e_m| \otimes |f_n|\big),$$

and $h \in \mathbb{F}$, if

$$\big(|e_m| \otimes |f_n|\big) \le \tfrac{1}{16}\tau. \tag{5.5}$$

The multiplication routine performs ScaleExpansion on f for each summand of e and then adds the intermediate expansions using FastExpansionSum. It does not create intermediate summands larger than output summands, and therefore the bounds above apply to all intermediate summands as well. If TwoProduct is based on a fused-multiply-add instruction, then no overflow occurs in a multiplication of two expansions if Equation (5.5) holds. If however TwoProduct is based on split, the splitting step may generate larger numbers. In this case, multiplication of expansions is safe from overflow, if

$$\max\{|e_m|, 2^{\lceil p/2 \rceil} + 1\} \otimes \max\{|f_n|, 2^{\lceil p/2 \rceil} + 1\} \le \tfrac{1}{16}\tau. \tag{5.6}$$

The criteria (5.3), (5.4), (5.5), and (5.6) are used by `protO` to ensure operations are free from floating-point exceptions. We did not implement a similar approach for `plaiO`, since bounding the output summands would first require to locate the most significant and least significant input summand, which seemed too time consuming in an unstructured sum.

5.1.4. Conversion to Expression Dags. Arithmetic operations in `Local_double_sum` may fail for one of three reasons. An operation which is not addition, subtraction, multiplication, or sign computation is requested, the number of summands for the result of an operation, as predicted by #s, exceeds the maximum number of summands, or a floating point exception is impeding or has already occurred. In all of these cases, the input to the failed operation has to be transformed into an expression dag representation. We provide the following *DataMediator* models which may be used with `Local_double_sum`.

`Local_double_sum_to_expression_dag_node_mediator` **(**`nodeM`**):** To convert a sum into an expression dag, this model computes the value of the sum exactly using bigfloat numbers and operations from the *ApproximationPolicy*. Then it creates a single dag node storing the result.

`Expansion_to_expression_dag_node_mediator` **(**`expaM`**):** This model converts the sum into a single dag node, too. In combination with `plaiO`, it behaves exactly like `nodeM`, but for expansions the improved conversion method based

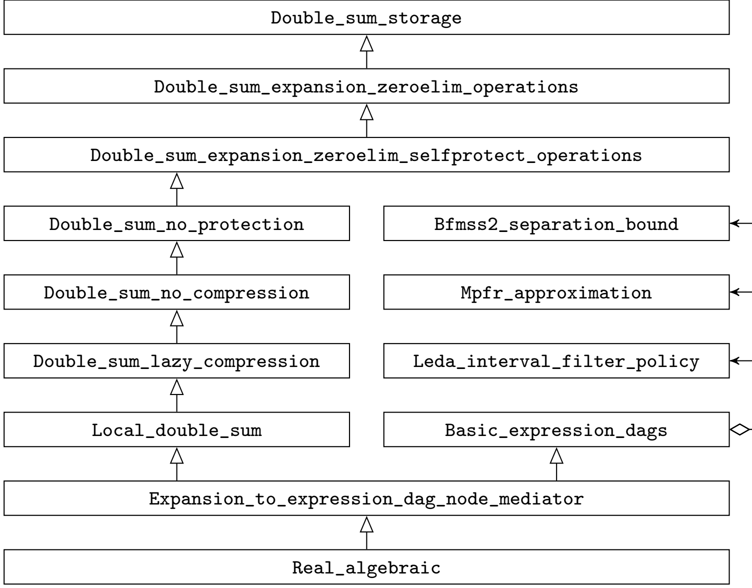


FIGURE 5.4. Collaboration of classes in a *RealAlgebraic* variant with `Local_double_sum` as *LocalPolicy*.

on MONOTONIZE from Section 4.2 is used. There is another small improvement. Let E be some number and e_m and e_{m-1} the two leading summands in the unique monotone, maximally non-overlapping expansion representing E. Then $|E - e_m| \leq \text{succ}(|e_{m-1}|)$ and e_m and $\text{succ}(|e_{m-1}|)$ are nearly optimal midpoint and radius of a floating-point interval containing E. Computing $\text{succ}(|e_{m-1}|)$ is free from overflow, since $|e_{m-1}| < |e_m|$. We initialize the floating-point interval in the dag node directly from these numbers, instead of computing it from the exact bigfloat representation.

`Local_double_sum_to_expression_dag_tree_mediator` (`treeM`): Directly converts the sum into an expression tree, more precisely a binary tree of minimal height, whose leaves store the summands and whose intermediate nodes are addition nodes. With this *DataMediator*, `Local_double_sum` may be seen as an expression rewriting engine, rewriting polynomial expressions over floating-point numbers into equivalent sums of floating-point numbers.

`Local_double_sum_to_expression_dag_mediator_statistics` (`statM`): This model collects a histogram for the number of summands in sums converted to an expression dag representation. It must be instantiated with another *DataMediator* model, which performs the actual conversion. These statistics allow us to study the effect different parameters have on the ability of `Local_double_sum` to defer dag creation.

Another conversion alternative would be to introduce addition nodes with arity greater than two, like they are available in `CORE::Expr 2`, to *RealAlgebraic*. That would allow a conversion strategy which is similar to `treeM`, but saves the creation of all but one intermediate node.

5.1.5. Collaboration of Policies. We can obtain a `Local_double_sum` variant by collecting a set of models for the policies and pass them to the host class `Local_double_sum`. The code below creates a *RealAlgebraic* variant by replacing the *LocalPolicy* in `Default_real_algebraic` with a `Local_double_sum` variant, but otherwise keeps the default policies. The number type `Default_dsumL_real_algebraic` created here, is the one labeled `dsumL` in the experiments in Section 3.4 and the baseline variant for the experiments in the second part of this chapter.

```
template <class Derived>
struct dsumL_default_policies_base :
 public Default_real_algebraic_policies_base<Derived>{
  typedef Int_to_type<8>                           DoubleSumMaxLength;
  typedef Double_sum_expansion_zeroelim_selfprotect_operations<Derived>
                                                   DoubleSumOperations;
  typedef Double_sum_no_protection<Derived>        DoubleSumProtection;
  typedef Double_sum_lazy_compression<Derived>     DoubleSumCompression;
  typedef Local_double_sum<Derived>                LocalPolicy;
  typedef Expansion_to_expression_dag_mediator<Derived>
                                                   DataMediator;
};

struct dsumL_default_policies :
 public dsumL_default_policies_base<dsumL_default_policies> {};

typedef Real_algebraic<dsumL_default_policies>
                                         Default_dsumL_real_algebraic;
```

`Local_double_sum` gains access to the functionality of its policies by means of inheritance. We use a linear inheritance scheme, with policies ordered by dependencies. *DoubleSumCompression* models for example may modify sums through compression and therefore need access to a *DoubleSumOperations* model as well as a *DoubleSumProtection* model. The complete collaboration diagram for `Default_dsumL_real_algebraic` is given in Figure 5.4.

Figure 5.5 shows the implementation of multiplication in `Local_double_sum` and illustrates how policies interact. Note that the implementation must conform to the *LocalPolicy* concept discussed in Section 3.1. The base class `Double_sum_storage` provides an array to store summands and a variable `length` for the number of summands. A `length` of zero means that no sum is stored or more generally that no local representation is available. Conforming to the *LocalPolicy* concept, we use the `length` of the result as return value.

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

    multiplication_length_predictor predictor;
    Compression::compress_operands(a,b,predictor);

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

      typedef typename
      Select< Compression::Tag_compress_result_needs_protection::value
             || Operations::Tag_multiplication_needs_protection::value,
             typename Protection::Protector,
             typename Protection::NoProtector >::Result Protector;

      Protector p(*this);
      Operations::multiplication(a,b);
      Compression::compress_result();
      Protection::restore(p);

    }
  }
  return static_cast<bool>(Storage::length);
}
```

FIGURE 5.5. Implementation of `local_multiplication()` in `Local_double_sum`.

As a first step, we ask the *DoubleSumCompression* model to compress the operands a and b. Protection from floating-point exceptions in this step must be provided within the *DoubleSumCompression* model itself but of course relies on the given *DoubleSumProtection* model. The functor `multiplication_length_predictor` computes $\#s(a \times b)$. We use it first to guide the compression of operands and then to decide if the result may exceed the maximum number of summands. In case the result will fit, we select a `Protector` type from the *DoubleSumProtection* model, based on tags indicating the need for protection in the raw multiplication and subsequent compression step. It should be noted that this step has no runtime cost, selecting the `Protector` is performed at compile time. We create an instance of the `Protector` type guarding the multiplication result, then compute the result and ask the *DoubleSumCompression* model to compress it. Finally, the *DoubleSumProtection* model is asked to react on any floating-point exceptions that occurred in the two previous steps. Here, the `restP` model for example simply resets the `length` of the result to zero in case an exception occurred.

Many of the functions provided by policies and called from inside the multiplication step consist of a few lines of code only. Like the actual `Protector` type, these functions are known at compile time. We rely on the compiler to eliminate

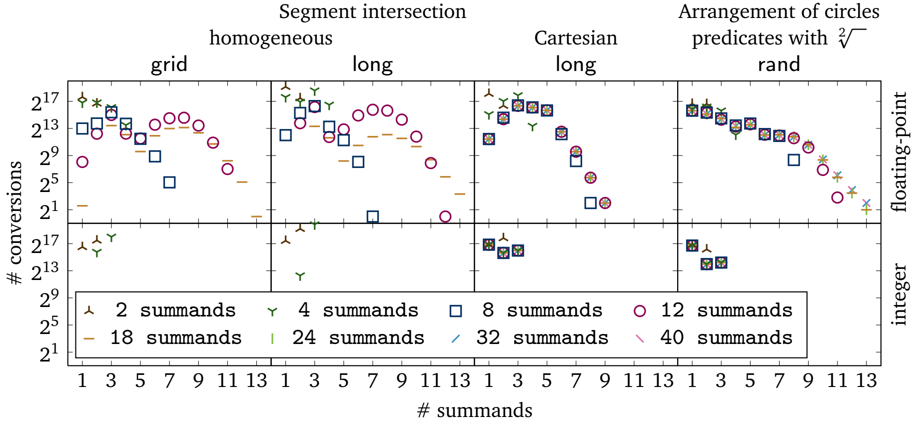


FIGURE 5.6. Effect of the maximum number of summands on the size and frequency of sums converted to an expression dag representation. The x-axis indicates the actual number of summands in converted sums, while the label indicates the maximum number of summands allowed.

the overhead of function calls by inlining and optimize the complete multiplication step as one unit. This is the advantage of template based design: flexibility and fine grained control without runtime overhead.

5.2. Experiments

Using the setup from Section 3.4, we investigate the effects of using `Local_double_sum` as *LocalPolicy* on the efficiency of *RealAlgebraic*. We compare the models for all our policies, with the goal of finding an overall optimal `Local_double_sum` variant. The default variant `Default_dsumL_real_algebraic` from the previous section is the result of these experiments. Then we compare it to other *LocalPolicy* models within *RealAlgebraic* and to other number types as well.

5.2.1. Evaluation of different `Local_double_sum` variants. The number of available policies and models allows to create a very large number of `Local_double_sum` variants. To reduce this amount, we resort again to the strategy of adopting a baseline variant and examine models for one policy at a time only. As baseline variant we use `Default_dsumL_real_algebraic`, which is a modification of `Default_real_algebraic`, the baseline variant from Section 3.4, but using `Local_double_sum` as *LocalPolicy*. Selected results are shown in figures 5.6 – 5.12, complete results are given in Appendix A. As before, number types are labeled with the models in which they differ from the baseline variant.

While examining one policy at a time helps to reduce the number of variants, the effects from models for one policy are not independent of the models selected for other

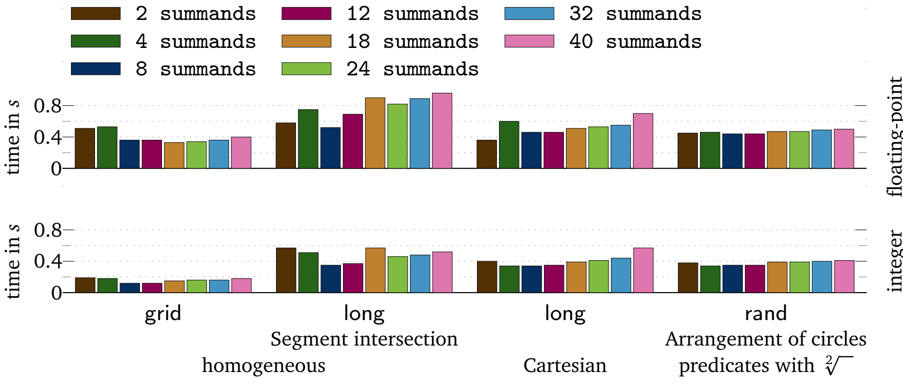


FIGURE 5.7. Effect of the maximum number of summands on the running time on the `descartes` platform. Selected problems correspond to Figure 5.6.

policies. For example, sums maintained by `plaiO` and `expaO` have very different characteristics and are therefore likely to need a different amount of compression for optimal performance. Similarly, the basic addition and subtraction provided by `plaiO` are inherently free from floating-point exceptions. Addition based on `plaiO` only needs protection when combined with `permC`. The results presented here for each policy are therefore only valid in the context of the other models fixed for the experiment. The baseline variant used for the experiments presented here, is the result from repeated experiments of the same kind, in which the baseline variant was iteratively modified with the goal of finding an optimal `Local_double_sum` variant.

NUMBER OF SUMMANDS. For a fixed number of input summands, multiplication needs significantly more output space than addition or subtraction. Therefore we perform experiments with a maximum number of summands $n \in \{2k^2, 2k(k+1) \mid k = 1,2,3,4\}$, which aligns with the space requirements for multiplication given by

$$\#s(a \times b) = 2\#s(a)\#s(b).$$

Nevertheless, multiplication is severely restricted. For example, for a maximum of eight summands, multiplication can only be performed when the operands have at most one and four, or two and two summands.

Results are consistent between different platforms. For integer input data, the optimal maximum number of summands is somewhere in the range of four to eight summands, surprisingly consistent for different problems and types of input data. For floating-point input data, the global optimum is slightly higher at eight summands, however for the problem of computing segment intersections the distribution is rather bimodal, preferring either fewer or more than eight summands, depending on the

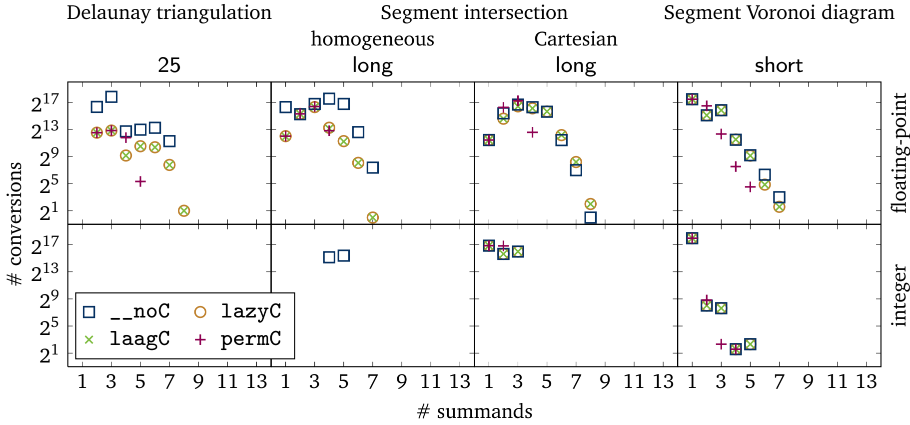


FIGURE 5.8. Effect of different *DoubleSumCompression* models on the size and frequency of sums converted to an expression dag representation.

input data. Following these observations, we select a maximum of eight summands for the baseline variant.

The maximum number of summands mediates a tradeoff between the ability to evaluate larger polynomial expressions and the storage space for a single *RealAlgebraic* number. Due to the static buffer for sums, increasing the maximum number of summands increases the storage space for each *RealAlgebraic*, regardless of whether the space is actually used. Wasting space may in turn have a negative influence on the performance, since it implies that fewer useful data fits into the various hardware caches. Assuming exact evaluation with `Local_double_sum` is faster than evaluation using expression dags, the optimal run time should be achieved for the smallest number of maximum summands that allows all polynomial sub-expressions to be evaluated completely without expression dags.

Figure 5.6 shows the effect of the maximum number of summands on the frequency and actual number of summands of sums converted to an expression dag representation for a few selected problems, Figure 5.7 shows the corresponding running times. We discuss integer data first. Homogeneous segment intersection employs polynomial predicates only. No sums with more than three summands are ever converted, and starting with a maximum number of eight summands, no conversions occur at all. And indeed, optimal performance is achieved for eight summands for these problems. The other two problems shown involve expressions with division and square root. Starting with a maximum number of four summands, always the same number of conversions occur, this is also the level where optimal performance is achieved. For these problems, conversions are triggered by the structure of the expression, i.e., a division or square root operation, not the lack of space for summands.

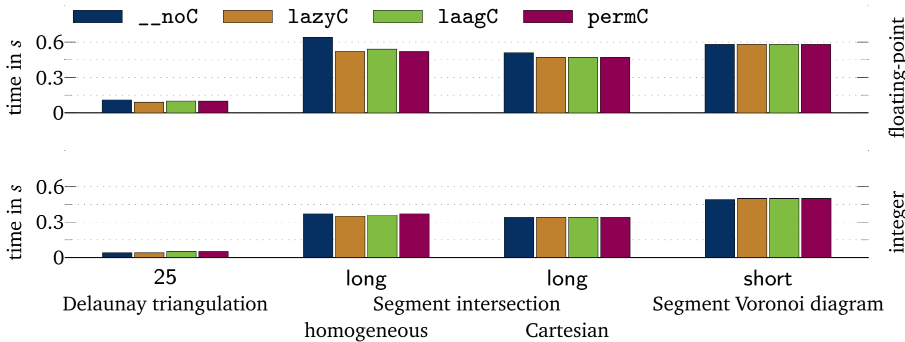


FIGURE 5.9. Effect of different *DoubleSumCompression* models on the running time on the descartes platform. Selected problems correspond to Figure 5.8.

For floating-point input data the situation is different. For homogeneous segment intersection and both input data sets, the amount of sums converted to an expression dag is of similar quality in Figure 5.6. With an increasing maximum number of summands, fewer sums with more summands are converted, and starting with a maximum number of 24 summands, predicates are evaluated completely without expression dags. But while for grid input data, optimal performance is achieved when allowing 18 to 24 summands, this is by no means the case for long input data, which has minima at two and at eight summands. The grid input data triggers more degenerate predicate evaluations, which require bigfloat evaluations of the expression dag, while the long input data involves only non-degenerate predicate calls, which are solved by the floating-point filter. Therefore the strategy to postpone dag creation longer by using more summands pays of for grid input data, but not for long input data. Cartesian segment intersection on long input data again involves only non-degenerate predicate calls and optimal performance is achieved for two summands. There is however another minimum at eight and 12 summands, which according to Figure 5.6 corresponds to the point where deferring dag construction is limited by the structure of the expression. The results for arrangements of circles on rand input data may be interpreted similarly, though actual differences are very small.

For non-degenerate input, allowing more summands seems to pay off for integer input, but not for floating-point input. The reason is the fixed cost of the floating-point filter, which for non-degenerate input is the only relevant stage of dag evaluation. The cost for this stage is the same for both integer and floating-point input. Exact evaluation using expansions needs fewer summands for integer input and hence is faster than dag creation and the floating-point filter in this case.

Finally, it is interesting to estimate the number of summands necessary to evaluate predicates completely with Local_double_sum. For segment intersection, the most

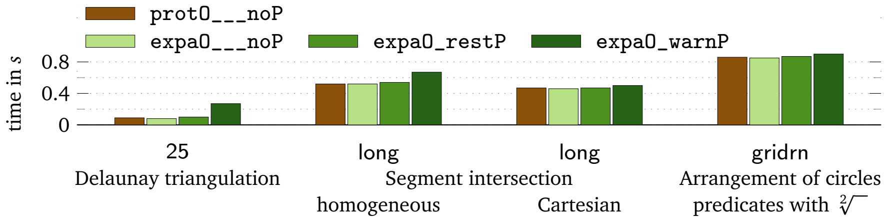


FIGURE 5.10. Effect of different *DoubleSumProtection* models on the running time. Floating-point data on `descartes`.

involved predicate is the comparison of coordinates of intersection points. In the homogeneous case, this involves computing the sign of

$$E = D_x D'_w - D'_x D_w,$$

where the matrices D_x and D_w are given in Equation (1.4). We have $\#\mathrm{s}(E) = 2048$, so at first it seems surprising this expression can be evaluated exactly using a few summands only. We can however get better estimates. For each intermediate result, we compute the number of bits and the minimum number of summands required to represent it exactly. Applying #s to these figures gives the number of summands necessary to perform the next operation exactly. For example, D_x and D_w have polynomial degree three and two, respectively. For 25 bit integer input they are representable with 78 and 53 bits, and after compression fit into an expansion with two and one summand. Hence, the product of these two sums is computable with four summands. The result however is exactly representable with 131 bits, which fits into an expansion of three summands. The final difference E needs six summands initially, but fits into three summands.

To perform the same computation for floating-point input, we need to estimate a precision that allows to simultaneously scale all input numbers to a predicate call to integers. For our input sets to segment intersection, a precision of 70 bits always suffices. Most input numbers do however have the same exponent, so on average a precision much closer to 53 bits should suffice. For an input precision of 53 bits, D_x and D_w are both representable with three summands, therefore 18 summands suffice to compute E. For an input precision of 70 bits, this goes up to 30 summands. For Cartesian segment intersection, computing the sign of

$$E = D_x / D_w - D'_x / D'_w,$$

is the hardest predicate. The largest polynomial sub-expression is D_x, which is exactly computable with four summands for 25 bit input, and 12 summands for 53 bit to 70 bit input.

Figure 5.6 and Figure 5.7 show that good performance is achieved when the maximum number of summands corresponds to the estimates given here, for the reasons previously discussed. Thus, estimating the necessary number of summands in

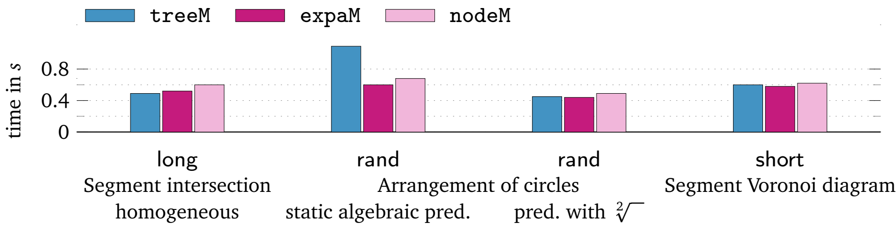


FIGURE 5.11. Effect of different *DataMediator* models that may be used with `Local_double_sum` on the running time. Floating-point data on `descartes`.

the described way allows the user to make a good choice for the maximum number of summands for her problem without resorting to experiment.

Protection from Floating-Point Exceptions. Results for different strategies to protect from floating-point exceptions are consistent over all platforms, types of input data, and algorithms, though more significant for algorithms where a larger fraction of the total running times is spend within `Local_double_sum`. Selected results are shown in Figure 5.10. No exceptions actually occur in our experiments. `warnP` is clearly the slowest approach, implying that the cost for resetting exception flags is very high. Surprisingly and despite making backup copies, `restP` is not much slower than having no protection at all. The fastest approach is however that of `protO`, which, with the exception of computing Delaunay triangulations, is nearly indistinguishable from using `expaO` without protection.

Compression. Results are consistent between different platforms and algorithms, but there are differences with respect to the input precision. Computing the Delaunay triangulation is the geometric algorithm where differences between *DoubleSumCompression* models are most significant. For floating-point input, `lazyC` is clearly the best variant and `noC` clearly the worst. `laagC` and `permC` are in between and show about equal performance. On the remaining algorithms, differences vanish with increasing complexity of the involved predicates. The situation is slightly different for integer input. Here `noC` shows a much better performance, especially on the `thales` platform, and `permC` is the slowest variant for homogeneous segment intersection.

Figure 5.8 shows the effect of different *DoubleSumCompression* models on the size and frequency of sums converted to an expression dag representation. For integer input, fewer conversions with fewer summands take place than for floating-point input. This explains the differences between integer and floating-point input. For lower precision input, additional compression does not pay off, since the results fit into the available space without it. For polynomial expressions, both `lazyC` and `laagC` significantly reduce the number of conversions in comparison to `noC`. This is not the case for more general expressions, here the main cause for conversion is the structure of the expression, not the limit on the number of summands. `permC` is

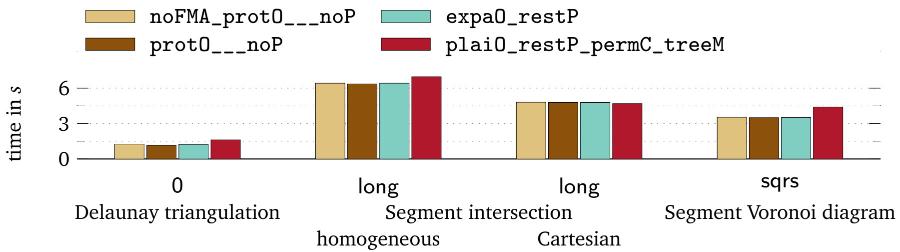


FIGURE 5.12. Running times for different choices to implement basic arithmetic operations. Floating-point data on `thales`.

the most effective compression strategy, it further reduces the number of conversions with many summands, though at the cost of increasing the number of conversions with few summands.

Overall, `lazyC` shows the best performance, for a variety of reasons. It applies compression only when there is a direct advantage to it and a single application of COMPRESS generally suffices to bring an expansion into optimal form. `permC` does not lead to a better running time, since it does not further reduce the number of conversions, i.e., the work that must be done on the expression dag, in particular when `expaM` is used. Thus we choose `lazyC` for the baseline variant.

CONVERSION TO EXPRESSION DAGS. Results are consistent over platforms and algorithms, but there are differences with respect to the amount of degeneracy in the input data. Selected results are shown in Figure 5.11. `expaM` is always better than `nodeM`, the few examples where this is not the case occur, when no conversions take place at all and can be attributed to noise in the running time measurements. This is not surprising, since both compute the same result, but `expaM` is faster. In comparison to the results from Section 4.2.3 the gain is relatively small. The reason is, that the majority of converted sums has one to four summands only, and by Amdahl's law the cost for conversion has only limited influence on the total cost.

`treeM` follows the strategy to create expression dags and is in some cases faster than `expaM`. This occurs in particular for non-degenerate input sets, where we already observed that increasing the number of summands does not increase the performance, i.e., those cases where creating an expression dag right away is superior to deferring dag creation using `Local_double_sum`. Overall however, `expaM` is favored by the majority of input sets and algorithms. Furthermore, choosing `treeM` over `expaM` can lead to greater decline in running time than choosing `expaM` over `treeM`. Therefore, we select `expaM` for the baseline variant.

BASIC ARITHMETIC OPERATIONS. The three models for *DoubleSumOperations* prefer different models for the remaining policies. Hence, for each *DoubleSumOperations* model we select a combination which is optimal for that model. `expaO` requires `restP` to handle floating-point exceptions, but is otherwise the same as `protO`.

plaiO needs restP, too, and achieves optimal performance with permC and treeM. On the thales platform, a fused-multiply-add instruction **fma** is available and will be used for error-free transformations by default. On this platform we add a variant not using **fma** to the set of competitors, to investigate the differences. Selected results from the thales platform are shown in Figure 5.12.

Results are consistent between different platforms. For nearly all algorithms and input sets, protO is clearly the fastest variant. Thus, the lazy strategy of plaiO, to postpone the actual work to the point where the sign is requested or compression is necessary does not pay off. There are a few exceptions though. For example, for Cartesian segment intersection on integer data, plaiO is the fastest variant on the non-degenerate input sets short and long. On the thales platform this also extends to floating-point input. Generally we recommend to use protO and hence chose it for the baseline variant.

Employing **fma** improves the performance in all cases, but substantial differences occur only for computing the Delaunay triangulation. Clearly, the multiplication takes up to small a portion of the total running time for more significant effects. The new floating-point standard IEEE 754-2008 [**46**] mandates the availability of a fused-multiply-add instruction, and new hardware providing it is under way. Since a single **fma** call replaces 16 other floating-point operations in the implementation of TwoProduct, we can expect similar improvements on those future platforms and thus recommend to implement TwoProduct based on **fma** when available.

5.2.2. Comparison to other Exact Geometric Computation Approaches. On the whole, the differences between several models for policies of Local_double_-sum seem rather small. This is in part owning to the fact that for many problems, only a small part of the total running time is spend within Local_double_sum and an even smaller fraction is influenced by a single policy. The majority of the running time is still spend in expression dag evaluation. Hence, the significance of differences is largest for problems with simpler predicates. The exception to this rule is the choice of the maximum number of summands. This policy evidently has the largest influence on how successful Local_double_sum is in avoiding dag creation and evaluation. The other policies merely influence the running time spend within Local_double_sum itself.

In Section 3.4.3 we already discussed experiments comparing *RealAlgebraic* variants nodaL, doubL, and dsumL to other exact number types and to approaches to Exact Geometric Computation on the level of geometric primitives. In these experiments, dsumL is the baseline Local_double_sum variant determined through the experiments described in this section. When applied straightforwardly, all three *RealAlgebraic* variants are competitive to and often much faster than other exact number types. Thus, the baseline *RealAlgebraic* variant nodaL is already very efficient to begin with. Our experiments show, that deferring dag construction can still improve performance in many cases.

For floating-point input data, `dsumL` is an improvement over `nodaL`, and then usually the fastest *RealAlgebraic* variant, if there is a certain amount of degeneracies in the input data.

For integer input, `dsumL` is an improvement over `nodaL` for the same input sets, but `doubL` is often even faster. The reason is that much lower precision is sufficient to evaluate predicates exactly. Hence, `doubL` can defer dag creation as effectively as `dsumL`, but faster. Only for homogeneous segment intersection, `dsumL` is faster than `doubL` on integer input. Homogeneous coordinate representation allows to avoid division, but requires predicates with higher polynomial degree and thus more precision to compute the final result exactly. Whether `doubL` or `dsumL` is the better approach to defer dag creation thus does not depend on input precision but on the precision necessary to evaluate polynomial sub-expression in all predicates exactly.

From these observations a few guidelines, which *LocalPolicy* to use, follow. If one knows that input data is free from degeneracies, no attempt to defer dag construction should be made, i.e., `nodaL` should be used. If one expects degenerate predicate evaluations, `doubL` oder `dsumL` should be used. Which of these, depends on the precision of input numbers and on the degree of the largest polynomial sub-expressions occurring in predicates. From this data, one can estimate the number of summands necessary to evaluate these expressions exactly. If one or two summands suffice, `doubL` should be used, otherwise `dsumL` with an appropriate number of summands should be used.

Approaches to Exact Geometric Computation on the level of geometric primitives are faster than the three examined *RealAlgebraic* variants in many cases, but there are a few exceptions, for example when computing the Delaunay triangulation of point sets in the plane. The major difference between our input data sets for this problem is the amount of nearly degenerate predicate calls they enforce. Figure 3.18 shows that `dsumL` behaves non-adaptively, i.e., the running time does not vary with the amount of degeneracies in the input data. Note that these problems are completely solved by `dsumL` and no expression dags are actually constructed. Although some strategies of `dsumL`, i.e., elimination of zero summands, or the lazy compression scheme, may lead to adaptive behavior, the maximum limit of eight summands is to small for these strategies to have a visible effect. Despite being non-adaptive at low precision, `dsumL` is a very efficient strategy compared to other approaches. For data sets without degeneracies, it can almost compete with the best adaptive number types, `nodaL` and `CORE::Expr 1`. With an increasing amount of degeneracies, however, it quickly becomes the fastest overall strategy, even outperforming the `Epick` kernel.

Similar behavior can be observed for other geometric problems in our experiments. The larger the amount of degeneracies in the input data, the more likely some *RealAlgebraic* variant is faster than Exact Geometric Computation approaches on the level of geometric primitives. In case any *RealAlgebraic* variant is faster, then usually `dsumL` is the fastest overall approach, showing that exact floating-point algorithms allow for very fast exact evaluation.

5.3. Exact Floating-Point Algorithms for Dag Evaluation

In this section we draw some general conclusion from the experiments of the previous section and discuss potentially better ways to employ exact floating-point algorithms based on error-free transformations in expression dag based number types.

Evidently, exact floating-point algorithms based on error-free transformations allow for very fast exact evaluation of polynomial expressions. Using them to defer dag creation does however not lead to an improvement for non-degenerate data. This shows, that they are still slower than dag creation and the first stage of expression evaluation with a dynamic floating-point filter. On the other hand, exact floating-point algorithms appear to be much faster than evaluation with bigfloat arithmetic, since deferring dag creation improves the overall running time in cases where bigfloat evaluation becomes necessary. An improvement is made even in the case of general expressions involving division and square roots, where only relatively small polynomial sub-expressions are evaluated exactly and an expression dag is created for the remaining expression.

This shows, that to achieve a performance improvement for both non-degenerate and degenerate data, exact floating-point algorithms should be placed between the floating-point filter stage and evaluation with bigfloat arithmetic. Such a strategy would first create the complete expression dag and evaluate it with a floating-point filter. Afterwards, exact floating-point algorithms will be used to collapse polynomial sub-expressions over the leaves of the dag to a single node. Precision driven arithmetic with bigfloat arithmetic would then be used on the remaining dag only.

For predicate calls that are decided by the floating-point filter, i.e., non-degenerate input data, such a strategy can achieve the same running time as the *RealAlgebraic* variant `nodaL`, which is faster than `dsumL` in these cases. For all other predicate calls, the running time would be similar to the running time achieved by `dsumL`, plus the additional time for dag creation and floating-point filter evaluation. This will be a bit slower than `dsumL` but is still likely to be faster than `nodaL`.

The different running times would be achieved on a per predicate basis, while currently one can select between `nodaL` or `dsumL` per data set only. Since rarely any input set forces purely non-degenerate or degenerate predicate calls only, it is likely that such a strategy would perform better or at least as good as the `nodaL` and `dsumL` number types in most cases.

INTEGRATION INTO *RealAlgebraic*. A relatively straightforward way to integrate exact floating-point algorithms into the dag evaluation stages of the *RealAlgebraic* framework is, to provide an *ApproximationPolicy* model based on error-free transformations. We now propose how to implement such a model. Based on results from the previous section we discard the option to represent a number as a plain sum of floating-point numbers but choose floating-point expansions. Thus, our model may be called `Expansion_approximation_policy` (`expaA`).

Since not all functionality that `expaA` must provide, can be implemented based on expansions alone, it must incorporate bigfloat arithmetic, too. A number shall

be represented either as an expansion, or as a bigfloat. Arithmetic operations on expansions shall be performed exactly, operations on bigfloat numbers approximately. Note that the *ApproximationPolicy* interface permits exact operations and our current *ExpressionDagPolicy* models already detects when an arithmetic operation is exact. Once a dag node becomes known exactly in that way, its children will not be accessed any more and may be pruned. When an operation may not be performed with expansions, a conversion from expansion to bigfloat representation takes place. The concrete type of bigfloat arithmetic should be of no concern to `expaA`, hence it shall take another *ApproximationPolicy* model as template parameter, which provides the bigfloat arithmetic.

For the implementation of arithmetic operations based on expansions within `expaA`, the same questions arise as for the implementation of `Local_double_sum` and most answers can be taken from the previous sections. For example, floating-point exceptions should be handled by checking the magnitude of input numbers and avoiding them in the first place, and a single application of COMPRESS suffices to bring an expansion into nearly optimal form. Conversion to a bigfloat number should be done using the efficient conversion method based on MONOTONIZE from Section 4.2.

Next to arithmetic operations, `expaA` has to provide some utility functions, e.g., rounding to a floating-point number, or read and write access to the exponent of a bigfloat number. Efficient implementations of these operations for expansions still have to be devised. One option not explored for `Local_double_sum`, is allowing interspersed zeros in expansions. This would simplify arithmetic operations, but complicate sign computation. Furthermore it requires more regular application of compression to keep the number of summands small.

Another option for `expaA` not explored with `Local_double_sum` is, to not limit the number of summands, as this would require dynamic memory management. The advantage would be, that only an impending floating-point exception, or a division or radical operation will force conversion to a bigfloat number. Arithmetic operations on expansions may well be faster than bigfloat arithmetic on the complete set of representable numbers. Recall that only numbers x with $\eta \leq \operatorname{lsb}(x)$, $\operatorname{msb}(x) \leq \tau$ are representable as expansion, which implies that only about 2000 bits are available in the `binary64` format. If the number of summands is not limited, the question when to apply compression takes a different form. The trigger in form of the maximum number of summands is not available in that case, but the representation as expansion should still be kept compact to keep arithmetic operations efficient.

We think that using expansions for exact evaluation of polynomial expressions in the first stage of expression dag evaluation in the described way may lead to further improvements of expression dag based number types. While many implementation parameters are already determined through our experiments with `Local_double_sum`, some parameters are still open and new parameters arise, so there is still room for further research and experimentation.

CHAPTER 6

Conclusion

In this thesis we discussed the implementation of expression dag based number types for geometric applications. We intended to improve the efficiency of such number types by means of Algorithm Engineering following the cycle of (re-)design, analysis, implementation and experimental evaluation. Our stated goal was to narrow or close the performance gap between expression dag based number types and other approaches to Exact Geometric Computation. One major tool that we considered were error-free transformations. These are small and fast algorithms based on imprecise hardware floating-point operations, which nevertheless allow for exact numerical computations.

Implementation of Expression Dag Based Number Types. In Chapter 3 we presented our new expression dag based number type *RealAlgebraic*. Our design allows to easily create a variety of number types, which all provide the same functionality, but differ in the implementation of major components for expression dag based number types. We conducted experiments to find an optimal *RealAlgebraic* variant and compared several *RealAlgebraic* variants to other number types, and to some problem specific approaches to Exact Geometric Computation.

Concerning the implementation of expression dag based number types, we observed that which component of an implementation accounts for most of the running time is strongly dependent on the type of geometric problem and on properties of the input data. To achieve a reasonable efficiency over a wide range of problems, each component must be implemented efficiently. The default *RealAlgebraic* variant `nodaL` is more efficient than other expression dag based number types, since there are in each case one or two components, where *RealAlgebraic* features a better implementation.

In comparison to problem specific approaches to Exact Geometric Computation, *RealAlgebraic* performs the better, the more involved the geometric problem at hand is. For Delaunay triangulation, `nodaL` is about a factor of six slower than Shewchuk's adaptive predicates on the `descartes` platform, cf. Figure 3.18 and Figure 4.3, and a factor of five to ten slower than the `Epick` kernel on any platform. For Cartesian segment intersection, `nodaL` is within 50% of the `Epeck` kernel, except for the axis dataset where the slowdown reaches a factor of four. For arrangements of circles, `nodaL` is faster than static algebraic predicates in a few cases, and for the problem of computing the Voronoi diagram of segments, *RealAlgebraic* is clearly the fastest

approach. This is a nice improvement over results reported for other expression dag based number types reported in the literature, see again Section 1.4.

We further identified a problem in current dag evaluation strategies, that may negate the main advantage of sharing common sub-expressions. In certain cases, common sub-expression may be evaluated as often as they are referenced within the dag. We propose a new algorithm TopPrecDrivArith which does not exhibit this behavior. The implementation and practical evaluation of that algorithm remains future work.

Exact Evaluation based on Error-Free Transformations. In Chapter 4 we presented improved algorithms based on error-free transformations for computing the sign of a sum of floating-point numbers exactly and used them for the implementation of geometric predicates. Many authors have suggested to implement geometric predicates based on error-free transformations and Shewchuk's adaptive predicates are considered state of the art implementations for the 2D and 3D orientation and incircle test. But despite their advantages over traditional software number types in terms of speed, exact computations based on error-free transformations do not seem to have found widespread usage in geometric applications.

The reason may be, that so far no-one has cast these techniques into a form that is efficient, yet simple to use. A major obstacle to this are the limitations of error-free transformations. Error-free transformations allow for approximately 2000 bits of precision only, before operations inevitably underflow or overflow. An implementation therefore has to either live with these limitations, or provide a fallback strategy, e.g., in form of a more general software number type.

We have presented new algorithms that allow for switching from floating-point expansions to such a fallback strategy efficiently. It is essential that our methods work correctly when the input expansions are close to overflow or underflow, as this may trigger switching to the fallback strategy in the first place. If the evaluation of an expression using expansions fails, our new conversion methods allow to resume the evaluation with more general software arithmetic at exactly the point where it failed. This basically means that it becomes unnecessary to store the expression for later re-evaluation.

Potential applications include static algebraic predicates, which reduce the sign computation of an algebraic number with an a-priori known structure to the evaluation of several polynomial expressions, or more generally the implementation of number types providing exact ring operations over integers or floating-point numbers. The integration of expansions into such number types should follow the guidelines given in Section 5.3. Since exact number types are the backbone of Exact Geometric Computation, many geometric application might benefit from the potential speed improvements.

Deferring Dag Construction. In Chapter 5 we discussed options to integrate exact computations based on error-free transformations into expression dag based number types. We presented a new approach to defer dag creation, which represents

a number exactly as a sum of floating-point numbers and avoids dag creation as long as such a representation can easily be obtained. This goes beyond the abilities of previous expression dag based number types, which allow to store a single floating-point number at most. Our experiments confirm that our extended approach is more effective than previous approaches for floating-point input and allows to avoid dag creation for larger sub-expressions. Furthermore it can lead to significant performance improvements, but only if there is a certain amount of degeneracies in the input data. The experimental results suggest, that exact computation with sums of floating-point numbers and error-free transformations may be better placed at a later stage of dag evaluation. The implementation and evaluation of such a strategy remain future work.

Nevertheless, we have achieved performance improvements over the default *RealAlgebraic* variant for many types of input data, which cover all types of geometric problems considered, while rigorously treating all limitations of error-free transformations and without sacrificing user-friendliness or limiting the scope or generality of our number-type. We achieved performance improvements even in cases where conversion to a more general software number type is always necessary due to the presence of division and square root operations. We hope that our results with *RealAlgebraic* serve as an incentive to try integrating exact computations based on error-free transformations into other approaches to Exact Geometric Computation as well.

Bibliography

1. Andrei Alexandrescu, *Modern* C++ *design: Generic programming and design patterns applied*, Addison-Wesley, 2001.
2. Gene Myron Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, Proceedings of the spring joint computer conference (AFIPS '67, spring), ACM, 1967, pp. 483–485.
3. Mohand O. Benouamer, Philippe Jaillon, Dominique Michelucci, and Jean-Michel Moreau, *A lazy exact arithmetic*, Proceedings of the 11th Symposium on Computer Arithmetic (ARITH'93), IEEE, 1993, pp. 242–249.
4. *boost* C++ *Libraries*, `http://www.boost.org/`.
5. Hervé Brönnimann, Christoph Burnikel, and Sylvain Pion, *Interval arithmetic yields efficient dynamic filters for computational geometry*, Discrete Applied Mathematics **109** (2001), no. 1–2, 25–47.
6. Hervé Brönnimann, Ioannis Z. Emiris, Victor Y. Pan, and Sylvain Pion, *Computing exact geometric predicates using modular arithmetic with single precision*, Proceedings of the 13th Symposium on Computational Geometry (SoCG'97), ACM, 1997, pp. 174–182.
7. Maxey Brooke, *Limerick-gimerick*, Word Ways **13** (1980), no. 1, article 10.
8. Christoph Burnikel, *Exact computation of voronoi diagrams and line segment intersection*, Ph.D. thesis, Universität des Saarlandes, 1996.
9. Christoph Burnikel, Rudolf Fleischer, Stefan Funke, Kurt Mehlhorn, Stefan Schirra, and Susanne Schmitt, *The LEDA class* ***real*** *number – extended version*, Tech. Report ECG-TR-363110-01, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2005.
10. Christoph Burnikel, Rudolf Fleischer, Kurt Mehlhorn, and Stefan Schirra, *Efficient exact geometric computation made easy*, Proceedings of the 15th Symposium on Computational Geometry (SoCG'99), ACM, 1999, pp. 341–350.
11. ———, *A strong and easily computable separation bound for arithmetic expressions involving radicals*, Algorithmica **27** (2000), 87–99.
12. Christoph Burnikel, Stefan Funke, Kurt Mehlhorn, Stefan Schirra, and Susanne Schmitt, *A Separation Bound for Real Algebraic Expressions*, Algorithmica **55** (2009), no. 1, 14–28.
13. Christoph Burnikel, Stefan Funke, and Michael Seel, *Exact geometric computation using cascading*, International Journal of Computational Geometry and Applications **11** (2001), no. 3, 245–266.
14. Christoph Burnikel, Kurt Mehlhorn, and Stefan Schirra, *How to compute the voronoi diagram of line segments: Theoretical and experimental results*, Proceedings of the 2nd European Symposium on Algorithms (ESA'94), LNCS, vol. 855, 1994, pp. 227–239.
15. ———, *The LEDA class* ***real*** *number*, Research Report MPI-I-96-1-001, Max-Planck-Institut für Informatik, Saarbrücken, Germany, January 1996.
16. *CGAL: Computational Geometry Algorithms Library*, `http://www.cgal.org/`.
17. Ee-Chien Chang, Sung W. Choi, DoYong Kwon, Hyungja Park, and Chee-Keng Yap, *Shortest path amidst disc obstacles is computable*, Proceedings of the 21st Symposium on Computational Geometry (SoCG'05), ACM, 2005, pp. 116–125.
18. *CORE: A core library for robust numeric and geometric computation*, `http://cs.nyu.edu/exact/`.

19. Mark de Berg, Otfried Cheong, Marc van Krefeld, and Mark Overmars, *Computational Geometry: Algorithms and Applications*, 3rd rev. ed., Springer, 2008.
20. Theodorus Jozef Dekker, *A floating-point technique for extending the available precision*, Numerische Mathematik **18** (1971), no. 3, 224–242.
21. James Demmel and Yozo Hida, *Fast and accurate floating point summation with application to computational geometry*, Numerical Algorithms **37** (2005), 101–112.
22. Olivier Devillers, Alexandra Fronville, Bernard Mourrain, and Monique Teillaud, *Algebraic methods and arithmetic filtering for exact predicates on circle arcs*, Computational Geometry: Theory and Applications **22** (2002), 119–142.
23. Olivier Devillers and Sylvain Pion, *Efficient exact geometric predicates for delauny triangulations*, Proceedings of the 5th Workshop on Algorithm Engineerung and Experiments (ALENEX'03), SIAM, 2003, pp. 37–44.
24. Zilin Du, *Guaranteed precision for transcendental and algebraic computation made easy*, Ph.D. thesis, Courant Institute of Mathematical Sciences, New York University, May 2006.
25. Thomas Dubé and Chee-Keng Yap, *A basis for implementing exact geometric algorithms*, extended abstract, Courant Institute of Mathematical Sciences, New York University, October 1993.
26. Ioannis Z. Emiris, Athanasios Kakargias, Sylvain Pion, Monique Teillaud, and Elias P. Tsigaridas, *Towards an open curved kernel*, Proceedings of the 20th Symposium on Computational Geometry (SoCG'04), ACM, 2004, pp. 438–446.
27. Ioannis Z. Emiris and Elias P. Tsigaridas, *Comparing real algebraic numbers of small degree*, Proceedings of the 12th European Symposium on Algorithms (ESA'04), LNCS, vol. 3221, 2004, pp. 652–663.
28. A. Robin Forrest, *Computational geometry and software engineering: Towards a geometric computing environment*, Techniques for Computer Graphics (David F. Rogers and Rae A. Earnshaw, eds.), Springer, 1987, pp. 23–37.
29. Steven Fortune and Christopher J. van Wyk, *Efficient exact arithmetic for computational geometry*, Proceedings of the 9th Symposium on Computational Geometry (SoCG'93), ACM, 1993, pp. 163–172.
30. ———, *Static analysis yields efficient exact integer arithmetic for computational geometry*, ACM Transactions on Graphics **15** (1996), no. 3, 223–248.
31. Stefan Funke, Christian Klein, Kurt Mehlhorn, and Susanne Schmitt, *Controlled perturbation for delaunay triangulations*, Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms (SODA'05), SIAM, 2005, pp. 1047–1056.
32. Stefan Funke and Kurt Mehlhorn, *LOOK: A lazy object-oriented kernel design for geometric computation*, Computational Geometry: Theory and Applications **22** (2002), no. 1–3, 99–118.
33. Stefan Funke, Kurt Mehlhorn, and Stefan Näher, *Structural filtering: A paradigm for efficient and exact geometric programs*, Computational Geometry: Theory and Applications **31** (2005), no. 3, 179–194.
34. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley, 1995.
35. Bernd Gärtner, *Pitfalls in computing with pseudorandom determinants*, Symposium on Computational Geometry, 2000, pp. 148–155.
36. Marina Gavrilova, Dmitri Gavrilov, and Jon G. Rokne, *New algorithms for the exact computation of the sign of algebraic expressions*, Canadian Conference on Electrical and Computer Engineering, May 1996, pp. 314–317.
37. *GMP: The GNU multiple precision arithmetic library*, `http://www.gmplib.org/`.
38. David Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Computing Surveys **23** (1991), no. 1, 5–48.
39. Stef Graillat, *Applications of fast and accurate summation in computational geometry*, Research Report RR2005–03, Laboratoire LP2A, Université de Perpignan, 2005.
40. Dan Halperin and Eran Leiserowitz, *Controlled perturbation for arrangements of circles*, Proceedings of the 19th Symposium on Computational Geometry (SoCG'03), ACM, January 2003, pp. 264–273.

41. Dan Halperin and Christian R. Shelton, *A perturbation scheme for spherical arrangements with application to molecular modeling*, Computational Geometry: Theory and Applications **10** (1998), 273–287.
42. Martin Held, *VRONI: An engineering approach to the reliable and efficient computation of Voronoi diagrams of points and line segments*, Computational Geometry: Theory and Applications **18** (2001), no. 2, 95–123.
43. Martin Held and Willi Mann, *An experimental analysis of floating-point versus exact arithmetic*, 23rd Canadian Conference on Computational Geometry (CCCG'11), August 2011, pp. 489–494.
44. Nicolas J. Higham, *Accuracy and stability of numerical algorithms*, 2. ed., SIAM, 2002.
45. *ANSI/IEEE Standard 754-1985: IEEE standard for binary floating-point arithmetic*, 1985, Reprinted in SIGPLAN Notices, 22(2):9-25, 1987.
46. *IEEE Standard 754-2008: IEEE standard for floating-point arithmetic*, 2008, Revision of [**45**].
47. William Kahan, *Further remarks on reducing truncation errors*, Communications of the ACM **8** (1965), no. 1, 40.
48. Vijay Karamcheti, Chen Li, Igor Pechtchanski, and Chee-Keng Yap, *A core library for robust numeric and geometric computation*, Proceedings of the 15th Symposium on Computational Geometry (SoCG'99), ACM, 1999, pp. 351–359.
49. Michael S. Karasick, Derek Lieber, and Lee R. Nackman, *Efficient Delaunay triangulation using rational arithmetic*, ACM Transactions on Graphics **10** (1991), no. 1, 71–91.
50. Menelaos I. Karavelas, *A robust and efficient implementation for the segment Voronoi diagram*, Proceedings of the 1st International Symposium on Voronoi Diagrams in Science and Engineering, 2004, pp. 51–62.
51. Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee-Keng Yap, *Classroom examples of robustness problems in geometric computation*, Computational Geometry: Theory and Applications **40** (2008), no. 1, 61–78.
52. Donald Ervin Knuth, *Seminumerical algorithms*, The Art Of Computer Programming, vol. 2, Addison-Wesley, 1969.
53. *LEDA: Library of Efficient Data Structures and Algorithms*, `http://www.algorithmic-solutions.com/`.
54. Chen Li, Sylvain Pion, and Chee-Keng Yap, *Recent progress in exact geometric computation*, Journal of Logic and Algebraic Programming **64** (2005), no. 1, 85–111.
55. Chen Li and Chee-Keng Yap, *A new constructive root bound for algebraic expressions*, Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms (SODA'01), SIAM, 2001, pp. 496–505.
56. Giuseppe Liotta, Franco P. Preparata, and Roberto Tamassia, *Robust proximity queries: An illustration of degree-driven algorithm design*, SIAM Journal on Computing **28** (1998), no. 3, 864–889.
57. Georg Mackenbrock, Helmut Ratschek, and Jon G. Rokne, *Experimental reliable code for 2D convex hull construction*, 1998, `http://pages.cpsc.ucalgary.ca/~rokne/convex/`.
58. Kurt Mehlhorn and Stefan Näher, *LEDA: A platform for combinatorial and geometric computing*, Cambridge University Press, Cambridge, November 1999.
59. Kurt Mehlhorn, Ralf Osbild, and Michael Sagraloff, *Reliable and efficient computational geometry via controlled perturbation*, Proceedings of the 33rd International Colloquium on Automata, Languages and Programming (ICALP'06), Part 1, LNCS, vol. 4051, 2006, pp. 299–310.
60. Maurice Mignotte, *Identification of algebraic numbers*, Journal of Algorithms **3** (1982), 197–204.
61. ______, *Mathematics for computer algebra*, Springer, 1992.
62. Bhubaneswar Mishra, *Algorithmic algebra*, Texts and Monographs in Computer Science, Springer, 1993.
63. Marc Mörig, *Deferring dag construction by storing sums of floats speeds-up exact decision computations based on expression dags*, 3rd International Congress on Mathematical Software (ICMS 2010), LNCS, vol. 6327, September 2010, pp. 109–120.

64. Marc Mörig, Ivo Rössling, and Stefan Schirra, *On the design and implementation of a generic number type for real algebraic number computations based on expression dags*, Mathematics in Computer Science **4** (2010), no. 4, 539–556.
65. ______, *On the design and implementation of a generic number type for real algebraic number computations based on expression dags*, Tech. Report FIN-001-2010, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, February 2010.
66. Marc Mörig and Stefan Schirra, *On the design and performance of reliable geometric predicates using error-free transformations and exact sign of sum algorithms*, 19th Canadian Conference on Computational Geometry (CCCG'07), August 2007, pp. 45–48.
67. ______, *Engineering an exact sign of sum algorithm*, Tech. Report FIN-002-2010, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, February 2010.
68. Marc Mörig, Sven Scholz, Tobias Tscheuschner, and Eric Berberich, *Chapter 6. Implementation Aspects*, in Müller-Hannemann and Schirra [**73**], pp. 237–289.
69. Marc Mörig and Silvio Weging, *Companion web page to* Summing Expansions Exactly and Efficiently, 2011, `http://wwwisg.cs.uni-magdeburg.de/ag/RealAlgebraic/ex2bf.html`.
70. ______, *Summing expansions exactly and efficiently*, Tech. Report FIN-09-2011, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, November 2011.
71. *MPFR: A multiple precision floating-point library*, `http://www.mpfr.org/`.
72. Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres, *Handbook of floating-point arithmetic*, Birkhäuser Boston, 2010.
73. Matthias Müller-Hannemann and Stefan Schirra (eds.), *Algorithm Engineering*, LNCS, vol. 5971, Springer, 2010.
74. David R. Musser, *Introspective sorting and selection algorithms*, Softw., Pract. Exper. **27** (1997), no. 8, 983–993.
75. Stefan Näher and Martin Taphorn, *Experimental evaluation of structural filtering as a tool for exact and efficient geometric computing*, 19th Canadian Conference on Computational Geometry (CCCG'07), 2007, pp. 41–44.
76. Aleksandar Nanevski, Guy Blelloch, and Robert Harper, *Automatic generation of staged geometric predicates*, Higher-Order and Symbolic Computation **16** (2003), no. 4, 379–400.
77. Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi, *Accurate sum and dot product*, SIAM Journal on Scientific Computing **26** (2005), no. 6, 1955–1988.
78. Thomas Ottmann, Gerald Thiemt, and Christian P. Ullrich, *Numerical stability of geometric algorithms*, Proceedings of the 3rd Symposium on Computational Geometry (SoCG'03), ACM, 1987, pp. 119–125.
79. Kouji Ouchi, *Real/Expr: Implementation of an exact computation package*, Master's thesis, Courant Institute of Mathematical Sciences, New York University, January 1997.
80. Michael L. Overton, *Numerical computing with IEEE floating-point arithmetic*, SIAM, 2001.
81. Katsuhisa Ozaki, Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi, *Adaptive and efficient algorithm for 2D orientation problem*, Japan Journal of Industrial and Applied Mathematics **26** (2009), no. 2–3, 215–231.
82. Victor Y. Pan, Yanqiang Yu, and C. Stewart, *Algebraic and numerical techniques for the computation of matrix determinants*, Computers and Mathematics with Applications **34** (1997), no. 1, 43–70.
83. Sylvain Pion and Andreas Fabri, *A generic lazy evaluation scheme for exact geometric computations*, Science of Computer Programming **76** (2011), no. 4, 307–323.
84. Sylvain Pion and Chee-Keng Yap, *Constructive root bound for k-ary rational input numbers*, Theoretical Computer Science **369** (2006), no. 1–3, 361–376.
85. Franco P. Preparata and Michael Ian Shamos, *Computational Geometry: An Introduction*, 1st ed., Springer, 1985.
86. Douglas M. Priest, *Algorithms for arbitrary precision floating point arithmetic*, Proceedings of the 10th Symposium on Computer Arithmetic (ARITH'91), IEEE, June 1991, pp. 132–143.

87. Helmut Ratschek and Jon Rokne, *Geometric computations with interval and new robust methods: Applications in computer graphics, GIS and computational geometry*, Horwood Publishing, Chichester, USA, 2003.
88. Helmut Ratschek and Jon G. Rokne, *Exact computation of the sign of a finite sum*, Applied Mathematics and Computation **99** (1999), no. 2–3, 99–127.
89. *RealAlgebraic: A number type for exact geometric computation*, `http://wwwisg.cs.uni-magdeburg.de/ag/RealAlgebraic/`.
90. Daniel Richardson, *How to recognize zero*, Journal of Symbolic Computation **24** (1997), no. 6, 627–645.
91. Siegfried Rump, *Error estimation of floating-point summation and dot product*, BIT Numerical Mathematics **52** (2012), no. 1, 201–220.
92. Siegfried Rump, Paul Zimmermann, Sylvie Boldo, and Guillaume Melquiond, *Computing predecessor and successor in rounding to nearest*, BIT Numerical Mathematics **49** (2009), no. 2, 419–431.
93. Siegfried M. Rump, *Ultimately fast accurate summation*, SIAM Journal on Scientific Computing **31** (2009), no. 5, 3466–3502.
94. Siegfried M. Rump, Takeshi Ogita, and Shin'ichi Oishi, *Accurate floating-point summation part I: Faithful rounding*, SIAM Journal on Scientific Computing **31** (2008), no. 1, 189–224.
95. ———, *Accurate floating-point summation part II: Sign, k-fold faithful and rounding to nearest*, SIAM Journal on Scientific Computing **31** (2008), no. 2, 1269–1302.
96. Edward R. Scheinerman, *When close enough is close enough*, American Mathematical Monthly **107** (2000), 489–499.
97. Stefan Schirra, *Robustness and precision issues in geometric computation*, Handbook of Computational Geometry (Jörg Rüdiger Sack and Jorge Urrutia, eds.), Elsevier, Amsterdam, The Netherlands, January 2000, pp. 597–632.
98. ———, *Invited lecture: Real numbers and robustness in computational geometry*, Real Numbers and Computers'6, November 2004, Tech. Report 08-2004, Computer Science, University of Trier, pp. 7–21.
99. ———, *Much Ado about Zero*, Efficient Algorithms, LNCS, vol. 5760, September 2009, pp. 408–421.
100. ———, *A note on sekigawa's zero separation bound*, Computer Algebra in Scientific Computing - 15th International Workshop (CASC 2013), LNCS, vol. 8136, September 2013, pp. 331–339.
101. Susanne Schmitt, *Web page for* `leda::real` *extendend* [**9**], `http://www.mpi-inf.mpg.de/projects/exacus/leda_extension/`.
102. ———, *Improved separation bounds for the diamond operator*, Tech. Report ECG-TR-363108-01, Effective Computational Geometry for Curves and Surfaces, Sophia Antipolis, France, 2004.
103. Hiroshi Sekigawa, *Zero determination of algebraic numbers using approximate computation and its application to algorithms in computer algebra*, Ph.D. thesis, University of Tokio, 2004.
104. Jonathan Richard Shewchuk, *Adaptive precision floating-point arithmetic and fast robust geometric predicates*, Discrete and Computational Geometry **18** (1997), no. 3, 305–363.
105. ———, *Companion web page to* [**104**], 1997, `http://www.cs.cmu.edu/~quake/robust.html`.
106. Pat H. Sterbenz, *Floating-point computation*, Prentice-Hall, 1974.
107. Jorge Stolfi, *Oriented projective geometry*, Academic Press, 1991.
108. Kokichi Sugihara and Masao Iri, *A robust topology-oriented incremental algorithm for Voronoi diagrams*, International Journal of Computational Geometry and Applications **4** (1994), no. 2, 179–228.
109. Kokichi Sugihara, Masao Iri, Hiroshi Inagaki, and Toshiyuki Imai, *Topology-oriented implementation – An approach to robust geometric algorithms*, Algorithmica **27** (2000), no. 1, 5–20.
110. Joachim von zur Gathen and Jürgen Gerhard, *Modern computer algebra*, 2nd ed., Cambridge University Press, 2003.
111. Ron Wein, *High-level filtering for arrangements of conic arcs*, Proceedings of the 10th European Symposium on Algorithms (ESA'02), LNCS, vol. 2461, 2002, pp. 884–895.
112. Fujio Yamaguchi, *A shift of playground for geometric processing from euclidean to homogeneous*, The Visual Computer **14** (1998), no. 7, 315–327.

113. Chee-Keng Yap, *Towards exact geometric computation*, Computational Geometry: Theory and Applications **7** (1997), no. 1–2, 3–23.
114. ______, *Fundamental problems of algorithmic algebra*, Oxford University Press, 2000.
115. ______, *On guaranteed accuracy computation*, ch. 12, pp. 322–373, World Scientific, 2004.
116. ______, *On guaranteed accuracy computation*, Geometric Computation, World Scientific, 2004, pp. 322–373.
117. ______, *Robust geometric computation*, Handbook of Discrete and Computational Geometry, CRC, 2nd ed., 2004, pp. 927–952.
118. Chee-Keng Yap and Thomas Dubé, *The exact computation paradigm*, Computing in Euclidean Geometry, World Scientific, 2nd ed., 1995, pp. 452–486.
119. Jihun Yu, Chee-Keng Yap, Zilin Du, Sylvain Pion, and Hervé Brönnimann, *The design of Core 2: A library for exact numeric computation in geometry and algebra*, 3rd International Congress on Mathematical Software (ICMS 2010), LNCS, vol. 6327, September 2010.

APPENDIX A

Complete Results from Experiments

The following tables contain results from experiments in Chapter 3 and Chapter 5. For each number type or geometric kernel and each geometric problem and input type, they contain the average running time for solving the geometric problem 25 times. A dash "–" indicates that a number type or geometric kernel is not applicable to the geometric problem, while a star "⋆" means the experiments crashed on at least one of the input sets.

The results are split into groups, reflecting their discussion in the text. The best running time in each group is typeset in bold and table cells are colored according to the relative deviation from the best time within the group. The color coding scheme is given in Figure A.1.

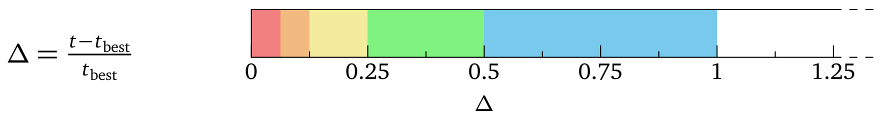


FIGURE A.1. Color coding of running times.

	Delaunay triangulation					Segment intersect-				
						homogeneous				
	0	25	50	75	100	short	long	grid	axis	short
nodaL	**0.06**	**0.08**	0.11	0.13	0.16	0.31	0.34	0.48	0.82	**0.20**
doubL	**0.06**	**0.08**	0.11	0.13	0.15	**0.29**	0.42	0.47	0.75	0.22
dsumL	0.09	0.09	**0.09**	**0.09**	**0.09**	0.34	0.53	0.36	0.50	0.32
leda::real	0.33	0.46	0.59	0.69	0.79	0.74	1.06	2.10	2.76	0.45
EXT::real	0.26	0.39	0.51	0.63	0.73	0.71	1.04	2.09	2.90	0.45
CORE::Expr 1.8	0.07	0.31	0.55	0.80	1.04	**0.29**	**0.33**	0.78	0.90	**0.20**
CORE::Expr 2.1.1	0.20	0.34	0.47	0.62	0.77	0.52	0.59	1.08	1.24	0.29
Lazy_exact_nt<Gmpzf>	0.16	0.19	0.21	0.23	0.26	0.33	0.45	0.26	0.43	–
Gmpzf	0.35	0.35	0.37	0.35	0.37	0.49	0.64	**0.23**	**0.42**	–
Lazy_exact_nt<Gmpq>	0.16	0.25	0.32	0.37	0.44	0.34	0.46	0.53	0.78	0.29
Gmpq	1.20	1.22	1.21	1.23	1.27	1.08	1.44	0.53	0.87	0.51
Epick	**0.01**	**0.06**	**0.09**	**0.13**	**0.16**	–	–	–	–	–
Epeck	0.02	**0.06**	0.10	0.14	0.17	–	–	–	–	**0.14**
boost_pool	**0.06**	**0.09**	0.12	**0.14**	**0.16**	**0.30**	**0.34**	**0.48**	**0.79**	**0.19**
leda_pool	**0.06**	**0.09**	**0.11**	**0.14**	**0.16**	0.31	**0.34**	**0.48**	0.80	0.20
no_pool	0.13	0.17	0.19	0.21	0.24	0.35	0.39	**0.48**	0.80	0.21
ledaF__nodaL	**0.06**	**0.08**	**0.11**	**0.13**	**0.15**	**0.31**	**0.35**	0.50	0.80	**0.20**
boostF_nodaL	0.09	0.12	0.13	0.15	0.17	0.34	0.39	**0.43**	**0.66**	0.21
ledaF__dintL	**0.08**	**0.10**	**0.13**	**0.15**	**0.17**	**0.29**	**0.38**	**0.52**	0.83	**0.21**
boostF_dintL	0.23	0.25	0.27	0.28	0.33	0.46	0.67	0.55	**0.80**	0.28
mpfrA	0.06	**0.08**	**0.11**	**0.13**	**0.15**	**0.31**	**0.34**	**0.48**	**0.81**	**0.20**
pure_mpfrA	0.06	0.09	0.13	0.16	0.19	**0.31**	0.35	0.52	0.86	**0.20**
ledaA	**0.05**	0.18	0.29	0.40	0.48	**0.31**	0.35	2.12	4.17	**0.20**
basicD	**0.05**	**0.08**	**0.11**	**0.13**	**0.15**	**0.30**	**0.34**	**0.48**	**0.82**	**0.20**
oledaD	0.06	0.12	0.17	0.23	0.27	0.33	0.39	1.03	1.55	0.22
nodaL	**0.06**	**0.08**	0.11	0.13	0.16	0.31	**0.34**	0.48	0.82	**0.20**
doubL	**0.06**	**0.08**	0.11	0.13	0.15	**0.29**	0.42	0.47	0.75	0.22
dwicL	0.08	0.10	0.12	0.15	0.17	**0.29**	0.37	0.49	0.82	0.21
dintL	0.08	0.10	0.13	0.15	0.17	0.30	0.38	0.50	0.85	0.21
dsumL	0.09	0.09	**0.09**	**0.09**	**0.09**	0.34	0.53	**0.36**	**0.50**	0.32
2 summands	0.17	0.18	0.20	0.20	0.21	0.36	0.58	0.51	0.79	**0.27**
4 summands	0.11	0.11	0.12	0.12	0.12	0.57	0.75	0.53	0.72	0.38
8 summands	**0.09**	**0.09**	**0.09**	**0.09**	**0.09**	**0.34**	**0.52**	0.36	0.49	0.33
12 summands	**0.09**	**0.09**	**0.09**	**0.09**	**0.09**	0.37	0.69	0.36	**0.37**	0.32
18 summands	0.10	0.10	0.10	0.10	0.10	0.43	0.90	**0.33**	0.41	0.36
24 summands	0.10	0.10	0.10	0.10	0.10	0.51	0.82	0.34	0.38	0.37
32 summands	0.10	0.11	0.10	0.11	0.11	0.50	0.89	0.36	0.38	0.39
40 summands	0.11	0.11	0.11	0.11	0.11	0.53	0.96	0.40	0.41	0.81
protO___noP	0.09	0.09	0.09	0.09	0.09	**0.34**	**0.52**	**0.36**	**0.50**	**0.32**
expaO___noP	**0.08**	**0.08**	**0.08**	**0.08**	**0.08**	**0.34**	**0.52**	**0.36**	**0.50**	**0.32**
expaO_warnP	0.27	0.27	0.27	0.28	0.28	0.47	0.67	0.41	0.62	0.35
expaO_restP	0.10	0.10	0.11	0.10	0.11	0.36	0.54	0.37	0.51	**0.32**
__noC	0.11	0.11	0.11	0.12	0.12	0.40	0.64	0.40	0.53	0.35
lazyC	**0.09**	**0.09**	**0.09**	**0.09**	**0.09**	**0.34**	**0.52**	0.37	0.50	**0.32**
laagC	0.10	0.10	0.10	0.10	0.10	**0.34**	0.54	**0.36**	**0.49**	0.33
permC	0.10	0.10	0.10	0.10	0.10	0.35	**0.52**	**0.36**	0.52	**0.32**
treeM	**0.09**	**0.08**	**0.09**	**0.09**	**0.09**	**0.31**	**0.49**	0.44	0.61	**0.30**
expaM	**0.09**	0.09	**0.09**	**0.09**	**0.09**	0.36	0.52	**0.37**	**0.49**	0.33
nodeM	**0.09**	0.09	0.10	0.10	0.10	0.39	0.60	0.43	0.58	0.42
expaO_restP	**0.10**	**0.11**	**0.11**	**0.11**	**0.11**	**0.35**	**0.53**	**0.37**	**0.51**	**0.33**
plaiO_restP_permC_treeM	**0.10**	**0.11**	0.13	0.14	0.15	0.39	0.65	0.56	0.66	**0.33**

TABLE A.1. Results for floating-point data on descartes.

ion			Arrangement of circles								Segment Voronoi diagram			
Cartesian			static algebraic predicates				predicates with $\sqrt[2]{\ }$							
long	grid	axis	rand	gridrn	pack	gridnn	rand	gridrn	pack	gridnn	mst	sqrs	short	shoax
0.26	0.30	0.35	0.99	1.54	0.30	1.13	0.38	0.79	0.32	0.89	0.68	0.35	0.62	0.38
0.29	0.30	**0.08**	0.86	1.72	0.29	0.98	0.38	0.78	0.31	0.86	0.68	0.30	0.61	0.34
0.47	0.27	0.21	0.61	1.35	**0.24**	0.91	0.44	0.87	**0.27**	0.82	0.68	**0.22**	**0.59**	**0.30**
0.63	1.10	0.36	1.20	2.26	0.81	3.06	1.04	1.75	1.97	3.50	0.69	0.55	1.98	0.73
0.64	1.13	0.51	1.12	2.13	0.81	3.17	1.02	1.75	1.44	3.64	0.69	0.53	2.03	0.71
0.26	⋆	0.62	3.08	4.42	1.25	2.39	0.75	1.47	4.28	4.74	0.71	1.39	1.50	1.31
0.33	0.49	0.50	1.24	1.64	0.76	1.40	0.51	1.15	0.80	1.29	0.69	1.28	1.40	1.72
–	–	–	–	–	–	–	–	–	–	–	0.59	0.54	–	–
–	–	–	–	–	–	–	–	–	–	–	**0.58**	0.51	–	–
0.27	0.31	0.16	**0.33**	**0.68**	0.29	0.68	**0.36**	**0.71**	0.29	**0.60**	0.62	0.79	0.84	0.97
0.66	**0.24**	0.27	0.78	0.98	0.31	**0.52**	0.84	1.38	0.34	0.65	0.60	0.84	1.03	1.40
–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
0.17	**0.52**	**0.09**	**0.28**	**0.55**	**0.32**	**0.69**	**0.42**	**0.80**	**0.39**	**0.75**	**0.62**	**0.99**	**1.07**	**1.21**
0.24	0.29	**0.33**	0.95	1.51	**0.30**	1.12	0.37	0.76	**0.31**	0.89	**0.67**	0.35	0.62	**0.38**
0.25	0.30	0.35	1.00	1.53	0.31	1.15	0.37	0.79	**0.31**	0.91	0.68	**0.34**	**0.61**	**0.38**
0.24	**0.28**	**0.33**	**0.93**	**1.20**	0.32	**1.02**	**0.34**	**0.73**	0.33	**0.84**	**0.67**	0.38	0.67	0.44
0.25	0.30	0.34	0.99	1.56	0.30	1.12	**0.37**	**0.78**	**0.31**	**0.89**	0.68	0.35	**0.62**	0.38
0.27	**0.29**	**0.14**	**0.98**	**1.35**	**0.29**	**0.81**	0.39	0.80	0.32	1.03	**0.67**	**0.31**	0.63	**0.36**
0.27	**0.32**	0.45	**0.88**	1.76	**0.31**	1.13	**0.39**	**0.81**	**0.32**	0.91	**0.68**	**0.36**	**0.63**	**0.40**
0.40	0.34	**0.15**	0.90	**1.45**	**0.31**	**0.77**	0.45	0.85	0.35	**0.81**	**0.68**	0.38	0.71	0.44
0.26	**0.30**	**0.36**	**0.99**	**1.54**	**0.29**	**1.15**	**0.37**	**0.80**	**0.31**	**0.88**	**0.68**	**0.35**	**0.62**	**0.38**
0.25	0.32	0.37	1.03	1.61	0.33	1.17	0.38	**0.80**	0.32	0.92	**0.68**	0.37	0.65	0.41
0.25	0.95	1.25	1.95	3.39	0.99	5.10	0.68	1.17	1.27	3.94	0.69	0.90	1.76	1.18
0.26	**0.30**	**0.34**	**1.00**	**1.54**	**0.30**	**1.14**	**0.37**	**0.79**	**0.31**	**0.90**	**0.67**	**0.35**	**0.61**	**0.37**
0.29	0.57	0.47	1.66	2.49	0.54	2.19	0.44	0.92	0.52	1.66	0.68	0.48	0.95	0.51
0.26	0.30	0.35	0.99	1.54	0.30	1.13	**0.38**	0.79	0.32	0.89	**0.68**	0.35	0.62	0.38
0.29	0.30	**0.08**	0.86	1.72	0.29	0.98	**0.38**	**0.78**	0.31	0.86	**0.68**	0.30	0.61	0.34
0.29	0.31	0.46	0.86	1.73	0.30	1.10	**0.38**	0.81	0.32	0.91	**0.68**	0.34	0.62	0.37
0.28	0.31	0.45	0.87	1.75	0.31	1.11	0.39	0.81	0.32	0.90	**0.68**	0.37	0.63	0.40
0.47	**0.27**	0.21	**0.61**	**1.35**	**0.24**	**0.91**	0.44	0.87	**0.27**	**0.82**	**0.68**	**0.22**	**0.59**	**0.30**
0.36	0.33	**0.19**	0.83	1.68	0.29	1.01	0.45	0.90	0.31	0.85	0.68	0.29	0.61	0.33
0.60	0.38	0.20	0.85	1.51	0.28	1.02	0.46	**0.85**	0.29	0.83	**0.67**	0.24	0.59	**0.30**
0.46	0.27	0.21	**0.60**	**1.36**	**0.24**	**0.91**	**0.44**	0.87	**0.27**	0.82	**0.67**	**0.22**	**0.58**	**0.30**
0.46	**0.25**	0.22	0.61	1.40	**0.24**	0.94	**0.44**	0.93	**0.27**	**0.80**	**0.67**	0.23	0.59	0.31
0.51	0.27	0.24	0.63	1.43	0.27	0.95	0.47	0.95	**0.27**	0.81	0.68	0.24	0.60	0.33
0.53	0.28	0.25	0.79	1.43	0.28	1.03	0.47	0.95	**0.27**	0.82	0.68	0.25	0.61	0.33
0.55	0.29	0.27	0.66	1.48	0.27	0.99	0.49	0.98	0.28	0.81	0.68	0.25	0.62	0.35
0.70	0.35	0.40	0.68	1.66	0.28	0.99	0.50	1.00	0.28	0.81	0.69	0.26	0.62	0.36
0.47	0.28	**0.21**	**0.60**	**1.36**	**0.24**	**0.90**	**0.44**	0.86	**0.27**	0.83	**0.67**	**0.22**	**0.58**	**0.30**
0.46	**0.27**	**0.21**	**0.60**	1.37	**0.24**	0.92	**0.44**	**0.85**	**0.27**	**0.81**	**0.67**	**0.22**	**0.58**	**0.30**
0.50	0.28	0.24	0.62	1.40	0.26	0.93	0.47	0.90	0.28	0.84	0.68	0.26	0.62	0.38
0.47	**0.27**	**0.21**	0.61	1.40	**0.24**	0.97	**0.44**	0.87	**0.27**	0.84	**0.67**	0.23	0.59	0.31
0.51	0.30	0.22	0.64	1.44	0.25	0.92	0.46	0.87	0.28	0.83	**0.67**	0.23	**0.58**	0.31
0.47	**0.27**	0.22	**0.60**	1.36	**0.24**	**0.91**	**0.44**	**0.84**	**0.27**	**0.81**	0.68	**0.22**	**0.58**	**0.30**
0.47	**0.27**	0.22	**0.60**	**1.35**	**0.24**	0.92	0.45	0.85	**0.27**	0.83	0.68	**0.22**	**0.58**	0.31
0.47	**0.27**	**0.21**	**0.60**	1.36	**0.24**	**0.91**	**0.44**	0.85	**0.27**	**0.81**	**0.67**	0.23	**0.58**	**0.30**
0.46	0.36	**0.21**	1.09	2.27	0.33	1.18	0.45	0.89	0.30	0.88	**0.68**	0.25	0.60	0.33
0.47	**0.27**	**0.21**	**0.60**	**1.35**	**0.24**	**0.90**	**0.44**	**0.85**	**0.27**	**0.81**	**0.68**	**0.22**	**0.58**	**0.30**
0.60	0.33	**0.21**	0.68	1.53	0.27	0.95	0.49	0.96	0.29	0.84	**0.68**	0.24	0.62	0.34
0.47	**0.27**	**0.21**	**0.61**	**1.40**	**0.24**	**0.98**	**0.44**	**0.88**	**0.27**	**0.84**	**0.67**	**0.23**	**0.59**	**0.31**
0.53	0.45	0.22	1.09	1.95	0.33	1.12	0.48	0.92	0.35	0.90	0.68	0.27	0.60	0.34

	Delaunay triangulation					Segment intersect-				
						homogeneous				
	0	25	50	75	100	short	long	grid	axis	short
nodaL	0.05	0.05	0.05	0.05	0.05	0.31	0.35	0.23	0.73	**0.20**
doubL	**0.04**	**0.04**	**0.04**	**0.04**	**0.04**	0.26	0.39	0.15	0.42	0.21
dsumL	**0.04**	**0.04**	**0.04**	**0.04**	**0.04**	**0.23**	0.35	**0.12**	**0.19**	0.25
leda::real	0.10	0.10	0.10	0.10	0.11	⋆	⋆	0.53	1.45	0.51
EXT::real	0.09	0.09	0.09	0.09	0.09	⋆	⋆	0.52	1.43	0.49
CORE::Expr 1.8	0.07	0.09	0.14	0.20	0.30	0.30	**0.33**	0.30	0.84	**0.20**
CORE::Expr 2.1.1	0.19	0.22	0.26	0.32	0.39	0.52	0.60	0.36	1.02	0.28
Lazy_exact_nt<Gmpzf>	0.16	0.17	0.16	0.17	0.18	0.33	0.44	0.17	0.42	–
Gmpzf	0.34	0.35	0.35	0.36	0.37	0.43	0.62	0.21	0.37	–
Lazy_exact_nt<Gmpq>	0.16	0.17	0.18	0.17	0.18	0.33	0.46	0.22	0.70	0.29
Gmpq	0.76	0.77	0.78	0.78	0.82	0.97	1.28	0.46	0.81	0.42
Epick	**0.01**	**0.01**	**0.01**	**0.01**	**0.01**	–	–	–	–	–
Epeck	0.02	0.02	0.02	0.02	0.02	–	–	–	–	**0.13**
boost_pool	0.06	0.06	0.06	0.06	0.06	**0.30**	**0.34**	**0.23**	**0.72**	**0.19**
leda_pool	**0.05**	**0.05**	**0.05**	**0.05**	**0.05**	0.31	0.35	**0.23**	0.75	0.20
no_pool	0.12	0.13	0.12	0.12	0.13	0.36	0.39	**0.23**	0.73	0.21
ledaF__nodaL	**0.05**	**0.05**	**0.05**	**0.05**	**0.05**	**0.31**	**0.35**	0.23	0.74	**0.20**
boostF_nodaL	0.09	0.09	0.09	0.09	0.10	0.33	0.39	**0.19**	**0.58**	0.21
ledaF__dintL	**0.05**	**0.05**	**0.05**	**0.05**	**0.05**	**0.30**	**0.42**	0.25	0.77	**0.21**
boostF_dintL	0.19	0.19	0.19	0.19	0.20	0.43	0.63	**0.24**	**0.60**	0.27
mpfrA	**0.05**	**0.05**	**0.05**	**0.05**	**0.05**	**0.31**	**0.35**	**0.23**	**0.75**	**0.20**
pure_mpfrA	**0.05**	**0.05**	**0.05**	**0.05**	**0.05**	**0.31**	0.36	0.24	0.78	**0.20**
ledaA	**0.05**	**0.05**	**0.05**	**0.05**	**0.05**	**0.31**	0.36	0.55	3.07	**0.20**
basicD	**0.05**	**0.05**	**0.05**	**0.05**	**0.05**	**0.31**	**0.35**	**0.23**	**0.74**	**0.20**
oledaD	**0.05**	0.06	**0.05**	0.06	**0.05**	0.33	0.39	0.31	1.22	0.22
nodaL	0.05	0.05	0.05	0.05	0.05	0.31	**0.35**	0.23	0.73	**0.20**
doubL	**0.04**	**0.04**	**0.04**	**0.04**	**0.04**	0.26	0.39	0.15	0.42	0.21
dwicL	0.05	0.05	0.05	0.06	0.06	0.30	0.42	0.25	0.73	0.22
dintL	0.05	0.05	0.05	0.05	0.05	0.30	0.43	0.25	0.75	0.21
dsumL	**0.04**	**0.04**	**0.04**	**0.04**	**0.04**	**0.23**	**0.35**	**0.12**	**0.19**	0.25
2 summands	0.06	0.07	0.06	0.07	0.07	0.37	0.57	0.19	0.52	0.28
4 summands	**0.04**	**0.04**	**0.04**	**0.04**	**0.04**	0.33	0.51	0.18	0.31	**0.23**
8 summands	**0.04**	**0.04**	**0.04**	**0.04**	**0.04**	**0.23**	**0.35**	**0.12**	**0.19**	0.24
12 summands	0.05	**0.04**	**0.04**	**0.04**	0.05	0.25	0.37	**0.12**	0.20	0.26
18 summands	0.05	0.05	0.05	0.05	0.05	0.28	0.57	0.15	0.30	0.29
24 summands	0.05	0.05	0.05	0.05	0.05	0.36	0.46	0.16	0.26	0.30
32 summands	0.05	0.05	0.05	0.06	0.06	0.32	0.48	0.16	0.27	0.32
40 summands	0.06	0.06	0.06	0.06	0.06	0.36	0.52	0.18	0.30	0.70
protO___noP	0.04	**0.04**	**0.04**	**0.04**	0.05	0.23	0.35	0.12	0.19	0.25
expaO___noP	**0.03**	**0.04**	**0.04**	**0.04**	**0.04**	**0.22**	**0.33**	**0.11**	**0.18**	**0.24**
expaO_warnP	0.22	0.22	0.22	0.23	0.23	0.40	0.61	0.21	0.35	0.27
expaO_restP	0.06	0.06	0.06	0.06	0.06	0.25	0.37	0.13	0.20	0.25
__noC	**0.04**	**0.04**	0.05	0.05	**0.04**	0.24	0.37	**0.12**	**0.19**	**0.25**
lazyC	0.05	**0.04**	**0.04**	**0.04**	0.05	0.24	**0.35**	**0.12**	**0.19**	**0.25**
laagC	0.05	0.05	0.05	0.05	0.05	**0.23**	0.36	**0.12**	**0.19**	**0.25**
permC	**0.04**	0.05	0.05	0.05	0.05	0.26	0.37	0.13	0.21	**0.25**
treeM	**0.04**	**0.04**	**0.04**	**0.04**	**0.04**	**0.23**	**0.35**	**0.12**	**0.18**	**0.22**
expaM	**0.04**	**0.04**	**0.04**	**0.04**	0.05	**0.23**	**0.35**	**0.12**	0.19	0.25
nodeM	**0.04**	**0.04**	**0.04**	**0.04**	0.05	**0.23**	0.37	**0.12**	0.19	0.29
expaO_restP	**0.06**	**0.06**	**0.06**	**0.06**	**0.06**	**0.25**	**0.37**	**0.13**	**0.21**	0.25
plaiO_restP_permC_treeM	0.07	**0.06**	**0.06**	**0.06**	0.07	0.27	0.40	0.14	**0.21**	**0.23**

TABLE A.2. Results for integer data on descartes.

ion			Arrangement of circles								Segment Voronoi diagram			
Cartesian			static algebraic predicates				predicates with $\sqrt[2]{}$							
long	grid	axis	rand	gridrn	pack	gridnn	rand	gridrn	pack	gridnn	mst	sqrs	short	shoax
0.25	0.15	0.33	0.85	1.26	0.24	0.97	0.36	0.74	0.21	0.85	0.66	0.18	0.56	0.31
0.32	**0.12**	**0.08**	0.67	1.04	0.22	0.57	**0.33**	**0.61**	**0.17**	0.58	0.67	**0.11**	**0.50**	**0.25**
0.34	0.14	0.14	0.54	1.21	0.21	0.77	0.35	0.71	0.19	0.71	0.67	**0.11**	**0.50**	**0.25**
0.74	0.38	0.13	1.01	1.70	0.41	2.52	0.82	1.39	0.64	2.77	0.67	0.16	1.26	0.46
0.73	0.40	0.16	0.95	1.62	0.40	2.59	0.83	1.43	0.65	2.91	0.67	0.16	1.27	0.44
0.26	0.24	0.61	2.97	4.18	1.18	2.12	0.51	1.03	0.93	2.23	0.68	0.39	1.23	1.02
0.33	0.21	0.50	1.16	1.46	0.52	1.10	0.47	1.05	0.48	0.99	0.66	0.41	1.26	1.47
–	–	–	–	–	–	–	–	–	–	–	**0.56**	0.25	–	–
–	–	–	–	–	–	–	–	–	–	–	**0.56**	0.19	–	–
0.28	0.14	0.17	**0.32**	**0.62**	**0.19**	0.52	0.34	0.64	0.20	**0.46**	0.57	0.34	0.68	0.84
0.52	0.19	0.26	0.69	0.86	0.28	**0.41**	0.74	1.24	0.30	0.56	0.57	0.28	0.77	1.14
–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
0.17	**0.14**	**0.09**	**0.28**	**0.49**	**0.23**	**0.52**	**0.41**	**0.74**	**0.26**	**0.59**	**0.56**	**0.40**	**0.90**	**1.08**
0.24	0.14	**0.31**	**0.81**	1.26	**0.24**	0.97	0.35	0.74	**0.21**	0.85	**0.67**	**0.18**	**0.56**	0.32
0.25	0.14	0.33	0.86	1.27	0.25	0.99	0.36	0.75	**0.21**	0.84	**0.67**	**0.18**	**0.56**	**0.31**
0.25	**0.13**	**0.31**	**0.81**	**1.00**	0.26	**0.88**	**0.33**	**0.68**	0.23	**0.78**	**0.67**	0.19	0.60	0.37
0.26	**0.14**	0.34	**0.85**	1.26	**0.24**	0.98	**0.36**	**0.74**	**0.21**	**0.83**	0.67	0.18	**0.56**	0.31
0.27	**0.14**	**0.13**	0.86	**1.23**	0.25	**0.75**	0.37	0.75	0.23	1.01	**0.66**	**0.12**	0.57	**0.30**
0.30	**0.16**	0.41	**0.68**	1.44	**0.23**	0.94	**0.35**	0.74	**0.18**	0.86	**0.67**	0.18	**0.52**	**0.32**
0.41	**0.16**	**0.15**	**0.68**	**1.07**	0.24	**0.60**	0.40	**0.72**	0.20	**0.63**	**0.67**	**0.13**	0.58	0.36
0.26	**0.14**	**0.33**	**0.85**	**1.26**	**0.24**	**0.97**	**0.36**	**0.75**	**0.21**	**0.83**	0.67	**0.17**	**0.56**	**0.31**
0.25	0.15	0.35	0.88	1.33	0.26	1.03	**0.36**	**0.75**	0.22	0.86	**0.66**	0.19	0.58	0.34
0.25	0.30	1.10	1.39	2.33	0.56	4.39	0.53	1.01	0.73	4.11	**0.66**	0.41	1.42	0.74
0.25	**0.15**	**0.33**	**0.86**	**1.26**	**0.24**	**1.00**	**0.36**	**0.75**	**0.21**	**0.83**	**0.67**	**0.18**	**0.56**	**0.31**
0.29	0.19	0.46	1.16	1.74	0.38	1.85	0.41	0.82	0.32	1.38	**0.67**	0.26	0.76	0.38
0.25	0.15	0.33	0.85	1.26	0.24	0.97	0.36	0.74	0.21	0.85	0.66	0.18	0.56	0.31
0.32	**0.12**	**0.08**	0.67	**1.04**	0.22	**0.57**	**0.33**	**0.61**	**0.17**	**0.58**	0.67	**0.11**	**0.50**	**0.25**
0.31	0.16	0.42	0.68	1.40	0.22	0.92	0.34	0.75	0.18	0.85	0.67	0.17	0.51	0.30
0.31	0.16	0.41	0.69	1.43	0.23	0.94	0.34	0.73	0.18	0.86	0.67	0.18	0.53	0.32
0.34	0.14	0.14	**0.54**	1.21	**0.21**	0.77	0.35	0.71	0.19	0.71	0.67	**0.11**	**0.50**	**0.25**
0.40	0.14	**0.12**	0.61	1.29	0.23	0.83	0.38	0.73	0.19	0.73	0.67	0.12	**0.49**	**0.26**
0.34	**0.13**	0.14	**0.68**	**1.20**	0.22	0.79	**0.34**	**0.69**	**0.18**	0.73	0.67	**0.11**	0.50	**0.26**
0.34	0.14	0.15	**0.54**	1.22	**0.21**	**0.77**	0.35	0.71	0.19	**0.72**	**0.66**	**0.11**	**0.49**	**0.26**
0.35	0.14	0.15	0.56	1.26	0.22	0.81	0.35	0.72	0.19	**0.72**	0.67	**0.11**	0.50	0.27
0.39	0.16	0.18	0.59	1.31	0.25	0.83	0.39	0.76	0.20	0.75	0.67	0.12	0.52	0.28
0.41	0.16	0.19	0.75	1.31	0.26	0.90	0.39	0.76	0.20	0.73	0.68	0.12	0.52	0.29
0.44	0.17	0.20	0.62	1.36	0.25	0.85	0.40	0.78	0.20	0.75	0.68	0.12	0.53	0.31
0.57	0.22	0.30	0.64	1.50	0.26	0.87	0.41	0.80	0.20	0.76	0.68	0.12	0.53	0.31
0.34	0.14	0.15	**0.55**	1.23	**0.21**	**0.78**	0.35	0.71	**0.19**	0.72	0.67	**0.11**	**0.50**	**0.26**
0.33	**0.13**	**0.14**	**0.55**	**1.22**	**0.21**	0.79	**0.34**	**0.70**	**0.19**	**0.71**	0.67	**0.11**	**0.50**	**0.26**
0.37	0.15	0.17	0.57	1.27	0.23	0.80	0.37	0.76	0.20	0.73	**0.66**	0.12	0.54	0.33
0.34	0.14	0.15	0.60	1.28	0.23	0.83	0.37	0.74	**0.19**	0.75	0.67	**0.11**	0.51	0.27
0.34	**0.14**	0.15	**0.55**	**1.22**	**0.21**	0.78	0.36	0.71	**0.19**	0.73	0.67	**0.11**	**0.49**	**0.26**
0.34	**0.14**	0.15	**0.55**	**1.22**	0.22	**0.77**	**0.34**	**0.70**	**0.19**	0.72	0.68	**0.11**	0.50	**0.26**
0.34	**0.14**	0.15	**0.55**	1.24	0.22	0.79	0.35	0.72	**0.19**	**0.71**	**0.66**	**0.11**	0.50	**0.26**
0.34	**0.14**	**0.14**	**0.55**	**1.22**	0.22	0.78	0.35	0.71	**0.19**	**0.71**	0.68	**0.11**	0.50	**0.26**
0.31	**0.13**	**0.14**	0.68	1.45	0.24	0.84	**0.35**	**0.71**	**0.19**	0.73	0.68	0.12	**0.50**	**0.26**
0.34	0.14	0.15	**0.56**	**1.22**	**0.21**	**0.77**	**0.35**	**0.71**	**0.19**	**0.71**	**0.67**	**0.11**	**0.50**	**0.26**
0.40	0.15	**0.14**	0.58	1.27	0.22	0.80	0.37	0.73	**0.19**	0.73	0.68	**0.11**	**0.50**	0.27
0.34	0.15	**0.15**	**0.60**	**1.26**	**0.23**	**0.83**	0.37	**0.73**	**0.20**	**0.74**	**0.67**	**0.11**	**0.51**	**0.27**
0.31	**0.14**	**0.15**	0.75	1.52	0.26	0.91	**0.36**	**0.73**	**0.20**	0.76	0.68	**0.11**	**0.51**	**0.27**

	Delaunay triangulation					Segment intersect-				
						homogeneous				
	0	25	50	75	100	short	long	grid	axis	short
nodaL	0.15	**0.21**	0.28	0.33	0.38	0.74	0.76	1.09	1.86	0.43
doubL	0.17	0.24	0.30	0.35	0.39	**0.69**	0.94	1.04	1.81	0.47
dsumL	0.24	0.24	**0.24**	**0.25**	**0.25**	0.87	1.31	0.90	1.31	0.70
leda::real	0.83	1.18	1.48	1.74	2.01	1.50	2.13	5.17	6.98	0.88
EXT::real	0.68	1.00	1.31	1.56	1.81	1.54	2.26	5.11	7.13	0.95
CORE::Expr 1.8	**0.14**	0.70	1.27	1.89	2.47	0.70	**0.66**	1.62	1.92	**0.42**
CORE::Expr 2.1.1	0.44	0.77	1.11	1.46	1.83	1.29	1.34	2.46	2.87	0.60
Lazy_exact_nt<Gmpzf>	0.64	0.71	0.77	0.82	0.87	0.92	1.22	0.69	1.18	–
Gmpzf	0.85	0.86	0.87	0.88	0.90	1.18	1.53	**0.56**	**0.99**	–
Lazy_exact_nt<Gmpq>	0.64	0.83	1.00	1.14	1.26	0.93	1.23	1.27	1.94	0.75
Gmpq	2.74	2.73	2.77	2.78	2.89	2.72	3.43	1.25	2.11	1.21
Epick	**0.02**	**0.12**	**0.21**	**0.29**	**0.35**	–	–	–	–	–
Epeck	0.08	0.18	0.28	0.35	0.42	–	–	–	–	**0.37**
boost_pool	**0.14**	**0.22**	**0.27**	**0.33**	**0.37**	**0.70**	**0.70**	**1.08**	1.89	**0.42**
leda_pool	**0.14**	**0.22**	**0.27**	**0.33**	0.39	0.74	0.73	1.09	**1.85**	0.43
no_pool	0.33	0.41	0.48	0.55	0.60	0.92	0.87	1.12	1.99	0.47
ledaF__nodaL	**0.14**	**0.21**	**0.27**	**0.33**	**0.38**	**0.74**	**0.75**	**1.09**	**1.87**	**0.43**
boostF_nodaL	0.50	0.61	0.69	0.77	0.84	1.07	1.22	1.43	2.17	0.55
ledaF__dintL	**0.22**	**0.29**	**0.36**	**0.40**	**0.45**	**0.68**	**0.87**	**1.15**	**1.99**	**0.46**
boostF_dintL	1.74	1.83	1.91	1.99	2.08	2.33	3.38	2.26	3.52	1.14
mpfrA	**0.14**	**0.21**	**0.27**	**0.33**	**0.38**	**0.74**	**0.75**	**1.09**	**1.86**	**0.43**
pure_mpfrA	**0.14**	0.25	0.32	0.40	0.47	**0.74**	**0.75**	1.19	1.99	**0.43**
ledaA	0.15	0.46	0.75	1.00	1.23	**0.74**	**0.75**	5.39	10.69	**0.43**
basicD	**0.14**	**0.21**	**0.27**	**0.33**	**0.39**	**0.74**	**0.75**	**1.09**	**1.87**	**0.43**
oledaD	0.18	0.30	0.42	0.52	0.60	0.81	0.84	2.13	3.35	0.46
nodaL	**0.15**	**0.21**	0.28	0.33	0.38	0.74	**0.76**	1.09	1.86	**0.43**
doubL	0.17	0.24	0.30	0.35	0.39	0.69	0.94	1.04	1.81	0.47
dwicL	0.21	0.28	0.34	0.39	0.44	**0.65**	0.84	1.14	1.95	0.48
dintL	0.22	0.29	0.35	0.40	0.45	0.67	0.87	1.16	1.96	0.46
dsumL	0.24	0.24	**0.24**	**0.25**	**0.25**	0.87	1.31	**0.90**	**1.31**	0.70
2 summands	0.43	0.46	0.49	0.51	0.54	**0.80**	**1.22**	1.12	1.85	**0.60**
4 summands	0.28	0.29	0.30	0.31	0.32	1.39	1.63	1.17	1.70	0.78
8 summands	**0.24**	0.25	0.25	0.25	0.26	0.87	1.30	0.91	1.30	0.72
12 summands	**0.24**	**0.24**	**0.24**	**0.24**	**0.25**	0.98	1.78	0.90	0.97	0.74
18 summands	0.26	0.26	0.26	0.26	0.27	1.07	2.37	0.85	1.05	0.79
24 summands	0.27	0.26	0.26	0.27	0.27	1.30	2.05	**0.84**	**0.94**	0.81
32 summands	0.28	0.28	0.28	0.28	0.29	1.26	2.25	0.91	0.97	0.87
40 summands	0.29	0.29	0.29	0.29	0.30	1.36	2.46	1.00	1.07	2.14
protO___noP	0.24	0.24	0.25	0.25	0.25	0.86	1.30	0.91	1.30	0.72
expaO___noP	**0.21**	**0.21**	**0.22**	**0.22**	**0.22**	**0.83**	**1.27**	**0.89**	**1.29**	**0.70**
expaO_warnP	0.79	0.79	0.80	0.81	0.83	1.27	1.75	1.01	1.68	0.81
expaO_restP	0.30	0.30	0.30	0.31	0.31	0.91	1.37	0.94	1.36	0.73
__noC	0.28	0.29	0.29	0.30	0.31	1.00	1.53	0.97	1.40	0.73
lazyC	**0.24**	**0.24**	**0.24**	**0.25**	**0.25**	**0.87**	1.31	0.92	**1.32**	**0.71**
laagC	0.26	0.26	0.26	0.26	0.27	**0.87**	1.31	0.92	1.33	**0.71**
permC	0.27	0.27	0.27	0.28	0.28	0.88	**1.28**	**0.88**	1.34	**0.71**
treeM	**0.24**	**0.24**	**0.24**	**0.25**	0.26	**0.81**	**1.27**	1.09	1.56	**0.70**
expaM	**0.24**	**0.24**	**0.24**	**0.25**	**0.25**	0.87	1.31	**0.90**	**1.31**	0.71
nodeM	0.25	0.26	0.26	0.26	0.27	0.99	1.50	1.05	1.49	0.94
expaO_restP	0.30	**0.30**	**0.30**	**0.31**	**0.32**	**0.91**	**1.36**	**0.94**	**1.35**	**0.73**
plaiO_restP_permC_treeM	**0.28**	0.32	0.36	0.39	0.42	1.01	1.65	1.35	1.72	0.75

TABLE A.3. Results for floating-point data on minkowski.

ion			Arrangement of circles								Segment Voronoi diagram			
Cartesian			static algebraic predicates				predicates with $\sqrt[2]{\ }$							
long	grid	axis	rand	gridrn	pack	gridnn	rand	gridrn	pack	gridnn	mst	sqrs	short	shoax
0.53	0.65	0.73	2.21	3.10	0.70	2.48	**0.77**	**1.57**	0.65	1.98	2.30	0.89	1.78	0.96
0.61	0.67	**0.18**	1.88	3.44	0.67	2.25	0.80	1.59	0.65	1.88	2.28	0.77	1.76	0.88
1.00	0.60	0.53	1.36	2.78	**0.54**	2.01	0.93	1.73	**0.57**	1.77	2.26	**0.58**	**1.69**	**0.79**
1.20	2.63	0.84	2.29	4.34	1.94	7.45	2.15	3.44	4.61	8.40	2.29	1.36	5.26	1.82
1.35	2.68	1.19	2.17	4.13	1.91	7.59	2.16	3.54	3.31	8.65	2.30	1.32	5.28	1.79
0.50	⋆	1.34	7.05	9.88	2.96	5.46	1.63	3.12	4.53	10.41	2.31	3.22	3.75	2.82
0.69	1.10	1.14	2.79	3.58	1.44	3.20	1.08	2.37	1.83	2.78	2.33	3.12	3.71	4.23
–	–	–	–	–	–	–	–	–	–	–	2.09	1.63	–	–
–	–	–	–	–	–	–	–	–	–	–	**2.06**	1.32	–	–
0.62	0.71	0.38	**0.77**	**1.50**	0.65	1.59	0.83	1.61	0.65	**1.40**	2.18	2.17	2.38	2.66
1.51	**0.56**	0.66	1.82	2.26	0.73	**1.18**	1.94	3.27	0.77	1.51	2.14	2.16	2.76	3.40
–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
0.46	**1.28**	**0.26**	**0.69**	**1.26**	**0.74**	**1.61**	**0.99**	**1.91**	**0.90**	**1.69**	**2.21**	**2.70**	**2.99**	**3.35**
0.51	0.65	**0.72**	**2.13**	3.06	**0.70**	2.48	**0.77**	1.52	**0.65**	1.97	**2.28**	**0.89**	**1.77**	**0.95**
0.53	0.65	**0.72**	2.20	3.14	**0.70**	2.48	**0.77**	1.55	**0.65**	1.98	**2.28**	**0.89**	**1.77**	0.96
0.52	**0.64**	0.77	2.17	**2.80**	0.76	**2.42**	**0.77**	**1.50**	0.69	**1.94**	2.29	0.98	1.90	1.15
0.53	**0.65**	0.74	**2.21**	**3.12**	**0.69**	2.50	**0.77**	**1.58**	**0.65**	**1.97**	2.28	**0.89**	**1.77**	**0.96**
0.70	0.80	**0.37**	2.43	3.30	0.83	**2.22**	0.92	1.86	0.78	2.60	**2.27**	1.00	2.08	1.20
0.58	**0.69**	0.96	**1.91**	**3.52**	**0.71**	2.50	**0.80**	**1.63**	**0.67**	**2.00**	**2.29**	**0.91**	**1.80**	**1.01**
1.60	1.17	**0.85**	2.38	3.76	0.96	**2.39**	1.48	2.78	1.02	2.32	2.31	1.91	2.92	2.41
0.53	**0.65**	**0.73**	**2.21**	**3.13**	**0.70**	**2.47**	0.78	**1.57**	**0.66**	**2.02**	2.31	**0.89**	**1.77**	**0.96**
0.53	0.71	0.78	2.31	3.33	0.76	2.64	**0.77**	1.58	0.68	2.05	**2.30**	0.96	1.85	1.05
0.53	2.41	3.09	4.63	7.64	2.49	13.22	1.46	2.54	3.05	9.57	2.32	2.27	4.83	3.12
0.53	**0.65**	**0.73**	**2.20**	**3.11**	**0.69**	**2.47**	**0.77**	**1.58**	**0.65**	**2.00**	**2.29**	**0.88**	**1.77**	**0.96**
0.59	1.20	0.98	3.51	4.92	1.19	4.60	0.91	1.83	1.09	3.53	**2.29**	1.16	2.47	1.24
0.53	0.65	0.73	2.21	3.10	0.70	2.48	**0.77**	**1.57**	0.65	1.98	2.30	0.89	1.78	0.96
0.61	0.67	**0.18**	1.88	3.44	0.67	2.25	0.80	1.59	0.65	1.88	2.28	0.77	1.76	0.88
0.62	0.70	1.00	1.91	3.54	0.71	2.47	0.80	1.63	0.66	1.99	**2.26**	0.87	1.79	0.97
0.59	0.69	0.97	1.90	3.52	0.71	2.52	0.80	1.60	0.67	1.99	2.27	0.91	1.80	1.00
1.00	**0.60**	0.53	**1.36**	**2.78**	**0.54**	**2.01**	0.93	1.73	**0.57**	**1.77**	**2.26**	**0.58**	**1.69**	**0.79**
0.76	0.72	**0.44**	1.78	3.35	0.68	2.19	**0.91**	1.78	0.65	1.88	2.26	0.74	1.76	0.85
1.17	0.78	0.50	1.89	3.04	0.64	2.19	0.93	**1.69**	0.61	1.84	2.26	0.62	**1.69**	**0.78**
1.01	0.61	0.53	**1.35**	**2.76**	**0.54**	**2.03**	0.92	1.73	**0.57**	1.78	2.24	**0.58**	**1.69**	0.79
1.03	**0.58**	0.55	1.39	2.88	0.56	2.06	0.95	1.89	**0.57**	**1.75**	**2.22**	0.59	1.70	0.80
1.09	0.60	0.58	1.43	2.94	0.61	2.10	0.99	1.96	0.58	1.77	2.27	0.61	1.72	0.83
1.14	0.62	0.62	1.84	2.94	0.63	2.19	1.01	1.98	0.58	1.81	2.29	0.62	1.73	0.85
1.23	0.66	0.67	1.49	3.03	0.60	2.14	1.02	2.04	0.59	1.79	2.29	0.64	1.75	0.88
1.45	0.74	0.85	1.53	3.44	0.62	2.18	1.06	2.10	0.59	1.79	2.28	0.66	1.75	0.89
1.02	0.61	0.54	**1.36**	**2.76**	**0.54**	**2.00**	**0.92**	1.74	0.57	**1.78**	**2.24**	**0.58**	1.71	0.80
0.98	**0.60**	**0.51**	**1.36**	2.81	**0.54**	2.01	**0.92**	**1.72**	**0.56**	1.80	2.25	**0.58**	**1.68**	**0.78**
1.11	0.65	0.60	1.45	2.92	0.60	2.05	0.99	1.84	0.60	1.84	2.26	0.68	1.83	1.01
1.02	0.62	0.54	1.39	2.91	0.55	2.19	0.93	1.78	0.57	1.91	2.25	0.60	1.71	0.82
1.05	0.67	**0.51**	1.43	2.94	0.59	2.04	0.94	1.75	0.61	1.82	**2.24**	0.60	**1.69**	**0.79**
1.00	**0.60**	0.52	**1.36**	**2.77**	**0.54**	**2.00**	**0.92**	**1.72**	**0.57**	**1.78**	2.25	0.59	**1.69**	**0.79**
1.00	0.61	0.54	**1.36**	**2.77**	**0.54**	2.05	**0.92**	1.73	**0.57**	1.79	**2.24**	**0.58**	**1.69**	**0.79**
1.00	**0.60**	0.52	1.37	2.80	**0.54**	2.01	0.93	1.74	**0.57**	**1.78**	**2.24**	0.59	**1.69**	0.80
1.04	0.82	**0.52**	2.23	4.23	0.73	2.61	0.95	1.83	0.63	1.96	**2.25**	0.65	1.74	0.85
1.00	**0.60**	0.54	**1.36**	**2.76**	**0.54**	**2.02**	**0.92**	**1.73**	**0.56**	**1.79**	**2.25**	**0.59**	**1.69**	**0.79**
1.37	0.74	**0.52**	1.56	3.10	0.61	2.13	1.06	1.92	0.61	1.84	2.26	0.62	1.77	0.90
1.02	**0.62**	**0.54**	**1.37**	**2.87**	**0.54**	**2.20**	**0.93**	**1.78**	**0.58**	**1.83**	2.26	**0.60**	**1.71**	**0.83**
1.17	0.98	0.55	2.22	3.82	0.78	2.52	1.00	1.83	0.73	1.98	**2.24**	0.70	1.75	0.89

	Delaunay triangulation					Segment intersect-				
						homogeneous				
	0	25	50	75	100	short	long	grid	axis	short
nodaL	0.13	0.13	0.13	0.13	0.13	0.74	0.74	0.51	1.77	0.43
doubL	**0.10**	**0.10**	**0.10**	**0.10**	**0.11**	0.63	0.89	0.37	1.01	0.47
dsumL	0.11	0.11	0.12	0.12	0.12	**0.59**	0.88	**0.31**	**0.48**	0.56
leda::real	0.26	0.26	0.26	0.27	0.27	⋆	⋆	1.14	3.53	0.97
EXT::real	0.23	0.23	0.23	0.23	0.24	⋆	⋆	1.17	3.62	1.02
CORE::Expr 1.8	0.15	0.20	0.31	0.46	0.71	0.70	**0.67**	0.61	1.79	**0.42**
CORE::Expr 2.1.1	0.45	0.51	0.63	0.75	0.93	1.28	1.32	0.81	2.40	0.60
Lazy_exact_nt<Gmpzf>	0.64	0.65	0.65	0.66	0.68	0.92	1.21	0.48	1.13	–
Gmpzf	0.81	0.83	0.83	0.85	0.96	1.06	1.39	0.50	0.91	–
Lazy_exact_nt<Gmpq>	0.65	0.65	0.66	0.66	0.68	0.93	1.23	0.57	1.73	0.74
Gmpq	1.74	1.74	1.77	1.81	1.86	2.37	3.01	1.05	1.91	0.98
Epick	**0.02**	**0.02**	**0.02**	**0.02**	**0.02**	–	–	–	–	–
Epeck	0.07	0.07	0.07	0.07	0.08	–	–	–	–	**0.38**
boost_pool	**0.12**	**0.13**	0.13	**0.13**	**0.13**	**0.69**	**0.70**	**0.50**	**1.76**	**0.41**
leda_pool	**0.12**	**0.13**	**0.12**	**0.13**	**0.13**	0.73	0.74	**0.50**	**1.76**	0.43
no_pool	0.30	0.31	0.31	0.31	0.33	0.92	0.87	0.55	1.86	0.47
ledaF__nodaL	**0.12**	**0.14**	**0.12**	**0.13**	**0.13**	**0.74**	**0.74**	**0.51**	**1.75**	**0.43**
boostF_nodaL	0.50	0.51	0.50	0.51	0.53	1.08	1.23	0.60	1.88	0.55
ledaF__dintL	**0.11**	**0.11**	**0.12**	**0.12**	**0.12**	**0.70**	**0.98**	**0.56**	**1.73**	**0.47**
boostF_dintL	1.51	1.52	1.53	1.55	1.60	2.23	3.27	1.26	2.93	1.12
mpfrA	**0.12**	**0.12**	0.13	**0.13**	**0.13**	**0.74**	**0.74**	**0.51**	**1.77**	0.43
pure_mpfrA	0.13	**0.12**	0.13	**0.13**	**0.13**	**0.74**	0.75	0.53	1.88	**0.42**
ledaA	**0.12**	**0.12**	**0.12**	0.14	**0.13**	**0.74**	0.75	1.36	7.98	0.43
basicD	**0.14**	**0.12**	**0.12**	**0.13**	**0.13**	**0.74**	**0.75**	**0.51**	**1.77**	**0.43**
oledaD	0.15	0.14	0.15	0.15	0.15	0.80	0.83	0.67	2.70	0.46
nodaL	0.13	0.13	0.13	0.13	0.13	0.74	**0.74**	0.51	1.77	**0.43**
doubL	**0.10**	**0.10**	**0.10**	**0.10**	**0.11**	0.63	0.89	0.37	1.01	0.47
dwicL	0.13	0.14	0.14	0.14	0.14	0.69	0.95	0.55	1.68	0.50
dintL	0.11	0.11	0.11	0.12	0.12	0.70	0.98	0.55	1.72	0.47
dsumL	0.11	0.11	0.12	0.12	0.12	**0.59**	0.88	**0.31**	**0.48**	0.56
2 summands	0.17	0.17	0.17	0.17	0.18	0.85	1.23	0.46	1.26	0.58
4 summands	**0.11**	**0.11**	**0.11**	**0.11**	**0.12**	0.82	1.21	0.43	0.72	**0.51**
8 summands	**0.11**	**0.11**	0.12	0.12	**0.12**	**0.60**	**0.88**	**0.31**	**0.48**	0.57
12 summands	0.12	0.12	0.12	0.12	**0.12**	0.63	0.91	**0.31**	0.49	0.60
18 summands	0.13	0.13	0.13	0.13	0.13	0.69	1.44	0.35	0.72	0.65
24 summands	0.13	0.14	0.14	0.14	0.14	0.89	1.10	0.38	0.62	0.69
32 summands	0.15	0.15	0.15	0.15	0.15	0.81	1.17	0.42	0.67	0.75
40 summands	0.16	0.16	0.16	0.16	0.16	0.89	1.31	0.46	0.76	1.88
protO___noP	0.11	0.11	0.11	0.11	0.12	0.60	0.87	0.30	0.46	0.58
expaO___noP	**0.09**	**0.09**	**0.09**	**0.09**	**0.09**	**0.57**	**0.83**	**0.29**	**0.45**	**0.56**
expaO_warnP	0.65	0.66	0.67	0.67	0.69	1.12	1.65	0.56	0.98	0.66
expaO_restP	0.18	0.19	0.18	0.18	0.19	0.65	0.96	0.33	0.55	0.58
__noC	0.12	**0.11**	**0.12**	**0.12**	**0.12**	0.61	0.93	**0.31**	**0.48**	**0.56**
lazyC	**0.11**	**0.11**	**0.12**	**0.12**	**0.12**	**0.60**	0.88	**0.31**	**0.48**	**0.56**
laagC	0.12	0.12	**0.12**	**0.12**	**0.12**	**0.60**	**0.87**	**0.31**	**0.48**	0.57
permC	0.12	0.13	**0.12**	0.13	0.13	0.63	0.94	0.33	0.53	0.57
treeM	0.12	**0.11**	**0.11**	**0.11**	**0.12**	**0.59**	**0.88**	0.31	**0.48**	**0.53**
expaM	**0.11**	**0.11**	**0.11**	**0.11**	**0.12**	0.60	**0.88**	0.31	0.49	0.57
nodeM	0.12	**0.11**	**0.11**	**0.11**	**0.12**	**0.59**	0.89	**0.30**	**0.48**	0.68
expaO_restP	**0.18**	**0.18**	**0.18**	**0.18**	**0.19**	**0.66**	**0.96**	**0.33**	**0.55**	0.59
plaiO_restP_permC_treeM	0.19	0.19	0.19	0.19	0.20	0.71	1.06	0.37	0.57	**0.54**

TABLE A.4. Results for integer data on `minkowski`.

ion			Arrangement of circles								Segment Voronoi diagram			
Cartesian			static algebraic predicates				predicates with $\sqrt[2]{}$							
long	grid	axis	rand	gridrn	pack	gridnn	rand	gridrn	pack	gridnn	mst	sqrs	short	shoax
0.53	0.31	0.68	1.84	2.56	0.55	2.16	0.73	1.46	0.43	1.88	2.27	0.47	1.64	0.80
0.70	**0.26**	**0.18**	1.41	2.09	0.48	1.27	**0.70**	**1.27**	**0.35**	1.30	2.25	0.32	**1.49**	**0.65**
0.74	0.30	0.34	1.24	2.47	0.48	1.71	0.75	1.47	0.38	1.57	2.25	**0.31**	1.50	0.68
1.39	0.81	0.28	1.94	3.19	0.96	6.08	1.64	2.68	1.54	6.65	2.24	0.43	3.41	1.14
1.51	0.89	0.34	1.86	3.00	0.94	6.16	1.69	2.85	1.54	7.00	2.24	0.42	3.44	1.11
0.50	0.49	1.32	6.67	9.13	2.65	4.67	1.11	2.19	2.00	4.77	2.24	0.96	3.16	2.39
0.68	0.46	1.17	2.58	3.18	1.17	2.49	1.03	2.17	1.01	2.24	2.26	1.06	3.43	3.69
–	–	–	–	–	–	–	–	–	–	–	**2.00**	0.74	–	–
–	–	–	–	–	–	–	–	–	–	–	2.01	0.52	–	–
0.62	0.31	0.38	**0.74**	**1.35**	**0.45**	1.17	0.80	1.46	0.45	**1.03**	2.04	0.93	2.00	2.38
1.21	0.44	0.64	1.60	1.91	0.61	**0.92**	1.73	2.82	0.67	1.26	2.05	0.74	2.16	2.86
–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
0.46	**0.35**	**0.26**	**0.67**	**1.13**	**0.50**	**1.18**	**0.98**	**1.76**	**0.62**	**1.30**	**2.05**	**1.10**	**2.57**	**3.00**
0.51	**0.30**	0.69	**1.80**	2.53	**0.54**	2.15	0.72	**1.46**	**0.43**	1.90	**2.26**	**0.47**	**1.64**	**0.79**
0.53	**0.30**	**0.68**	1.85	2.55	**0.54**	2.18	0.73	1.49	**0.43**	1.88	**2.26**	**0.47**	**1.64**	0.80
0.52	**0.30**	0.73	1.85	**2.32**	0.61	**2.07**	**0.71**	**1.46**	0.47	**1.82**	**2.26**	0.51	1.75	0.98
0.53	**0.31**	0.68	**1.85**	**2.56**	**0.54**	2.15	**0.73**	**1.47**	**0.43**	**1.87**	2.27	0.47	**1.64**	**0.81**
0.69	0.36	**0.37**	2.08	2.97	0.70	**2.01**	0.87	1.73	0.58	2.52	**2.25**	**0.42**	1.92	1.06
0.65	**0.34**	0.89	**1.47**	2.83	**0.51**	2.08	**0.70**	**1.45**	**0.37**	**1.83**	2.27	**0.47**	**1.52**	**0.81**
1.63	0.64	**0.84**	1.80	**2.76**	0.71	**1.88**	1.35	2.48	0.67	1.87	**2.26**	0.62	2.47	2.17
0.54	**0.31**	**0.68**	**1.85**	**2.56**	**0.55**	**2.16**	**0.73**	**1.47**	**0.43**	**1.87**	2.28	**0.47**	**1.64**	**0.80**
0.52	0.32	0.74	1.93	2.75	0.60	2.28	0.74	**1.47**	0.45	1.92	2.27	0.51	1.71	0.89
0.53	0.75	2.82	3.25	5.17	1.36	10.89	1.15	2.12	1.66	10.28	**2.26**	1.11	3.95	1.98
0.53	**0.31**	**0.68**	**1.85**	**2.55**	**0.55**	**2.16**	**0.74**	**1.47**	**0.43**	**1.88**	2.28	**0.47**	**1.64**	**0.80**
0.58	0.39	0.94	2.47	3.36	0.81	4.06	0.84	1.67	0.63	3.05	**2.25**	0.62	2.04	0.95
0.53	0.31	0.68	1.84	2.56	0.55	2.16	0.73	1.46	0.43	1.88	2.27	0.47	1.64	0.80
0.70	**0.26**	**0.18**	1.41	**2.09**	**0.48**	**1.27**	**0.70**	**1.27**	**0.35**	**1.30**	**2.25**	0.32	**1.49**	**0.65**
0.69	0.35	0.92	1.48	2.82	0.51	2.06	0.72	1.47	0.38	1.86	**2.25**	0.46	1.52	0.78
0.65	0.34	0.89	1.47	2.84	0.50	2.08	**0.70**	1.44	0.37	1.83	2.26	0.47	1.52	0.81
0.74	0.30	0.34	**1.24**	2.47	**0.48**	1.71	0.75	1.47	0.38	1.57	**2.25**	**0.31**	1.50	0.68
0.78	0.30	**0.29**	1.32	2.55	0.51	1.76	0.77	1.48	0.38	1.58	2.24	0.33	1.50	**0.66**
0.72	**0.28**	0.32	1.52	**2.40**	0.49	1.74	**0.73**	**1.42**	**0.37**	**1.54**	2.26	**0.31**	1.49	**0.66**
0.76	0.31	0.34	**1.23**	2.43	**0.47**	**1.71**	0.74	1.45	0.38	1.57	2.24	**0.31**	1.50	0.67
0.81	0.32	0.38	1.27	2.51	0.49	1.76	0.77	1.52	0.38	1.60	**2.20**	**0.31**	1.51	0.69
0.88	0.35	0.42	1.33	2.58	0.54	1.82	0.82	1.60	0.39	1.62	2.26	0.32	1.52	0.73
0.94	0.37	0.45	1.73	2.63	0.56	1.91	0.84	1.62	0.40	1.63	2.27	0.32	1.54	0.74
1.00	0.39	0.49	1.39	2.75	0.54	1.85	0.85	1.67	0.40	1.65	2.28	0.33	1.55	0.76
1.23	0.47	0.66	1.43	3.09	0.56	1.90	0.88	1.71	0.40	1.65	2.27	0.33	1.56	0.78
0.77	0.31	0.36	**1.23**	2.45	**0.47**	**1.71**	0.74	1.48	0.38	**1.56**	**2.23**	**0.31**	1.50	0.68
0.74	**0.30**	**0.34**	**1.23**	**2.44**	**0.47**	**1.71**	**0.73**	**1.45**	**0.37**	**1.56**	2.25	**0.31**	**1.49**	**0.66**
0.86	0.35	0.42	1.32	2.61	0.53	1.77	0.81	1.60	0.42	1.64	2.25	0.34	1.62	0.89
0.77	0.33	0.36	1.35	2.57	0.50	1.83	0.79	1.54	0.39	1.65	2.24	0.32	1.54	0.73
0.74	**0.30**	**0.34**	**1.23**	2.47	0.48	**1.71**	**0.74**	1.47	**0.38**	1.56	**2.22**	**0.31**	**1.49**	**0.67**
0.74	**0.30**	0.35	**1.23**	**2.44**	**0.47**	**1.71**	**0.74**	**1.46**	**0.38**	1.57	2.23	**0.31**	1.50	0.68
0.75	0.31	**0.34**	**1.23**	2.45	**0.47**	1.73	0.75	1.48	**0.38**	1.58	2.23	**0.31**	**1.49**	**0.67**
0.76	**0.30**	**0.34**	1.24	2.45	**0.47**	1.72	0.75	1.47	**0.38**	**1.55**	2.24	**0.31**	**1.49**	0.68
0.72	0.31	0.35	1.50	2.85	0.52	1.85	**0.74**	**1.45**	0.39	1.59	**2.23**	0.32	**1.49**	0.68
0.75	**0.30**	0.36	**1.25**	**2.44**	**0.47**	**1.71**	**0.74**	1.47	**0.38**	**1.56**	2.24	**0.31**	**1.49**	**0.67**
0.92	0.35	**0.34**	1.30	2.55	0.49	1.76	0.80	1.54	0.39	1.58	2.24	0.32	1.50	0.68
0.77	0.34	**0.36**	**1.35**	**2.58**	**0.51**	**1.84**	0.78	1.54	**0.40**	**1.67**	2.24	**0.31**	**1.53**	**0.72**
0.71	**0.33**	**0.36**	1.57	2.94	0.57	1.98	**0.77**	**1.51**	**0.40**	1.68	**2.23**	0.33	**1.53**	0.74

	Delaunay triangulation					Segment intersect-				
						homogeneous				
	0	25	50	75	100	short	long	grid	axis	short
nodaL	**0.99**	1.53	2.02	2.46	2.86	**2.56**	**3.50**	6.09	11.26	**1.65**
doubL	1.06	1.61	2.10	2.54	2.93	2.97	4.42	5.72	10.81	1.98
dsumL	1.16	**1.17**	**1.19**	**1.20**	**1.24**	4.01	6.34	5.13	7.48	3.15
CORE::Expr 1.8	1.14	5.43	9.77	14.23	18.74	2.96	4.07	10.94	13.75	1.93
CORE::Expr 2.1.1	4.50	7.00	9.68	12.42	15.31	8.66	12.22	16.41	20.72	3.75
Lazy_exact_nt<Gmpzf>	3.57	4.25	4.72	4.96	5.29	4.44	6.41	**4.14**	**6.83**	–
Gmpzf	6.89	6.96	6.98	7.08	7.30	8.01	11.65	4.39	7.78	–
Lazy_exact_nt<Gmpq>	3.52	4.81	5.91	6.69	7.39	4.40	6.35	8.12	11.64	2.64
Gmpq	14.30	14.45	14.56	14.75	15.25	16.74	25.28	9.50	15.63	7.30
Epick	**0.06**	**0.73**	1.30	1.77	2.17	–	–	–	–	–
Epeck	0.14	0.75	**1.27**	**1.72**	**2.08**	–	–	–	–	**1.37**
boost_pool	**0.99**	**1.55**	**2.04**	**2.49**	**2.89**	**2.57**	**3.50**	**6.12**	**11.27**	**1.65**
no_pool	2.63	3.39	4.08	4.70	5.26	3.83	5.31	7.40	13.34	2.05
ledaF__nodaL	**0.99**	**1.54**	**2.03**	**2.47**	2.87	**2.56**	**3.49**	6.12	11.38	**1.66**
boostF_nodaL	1.28	1.75	2.15	2.50	**2.80**	2.82	3.87	**5.59**	**8.95**	1.77
ledaF__dintL	**1.34**	**1.90**	**2.39**	**2.83**	**3.23**	**3.03**	**4.37**	**6.55**	11.97	**1.84**
boostF_dintL	3.39	3.84	4.23	4.57	4.93	5.24	7.70	7.22	**11.71**	2.80
mpfrA	**0.99**	**1.54**	**2.02**	**2.47**	**2.86**	**2.56**	**3.49**	**6.07**	**11.23**	1.65
pure_mpfrA	1.00	1.61	2.16	2.65	3.09	**2.56**	3.50	6.42	11.72	**1.64**
basicD	**0.99**	**1.54**	**2.03**	**2.47**	**2.87**	**2.55**	**3.50**	**6.09**	**11.22**	**1.65**
oledaD	1.13	2.10	2.96	3.77	4.43	2.76	3.79	12.63	20.88	1.74
nodaL	**0.99**	1.53	2.02	2.46	2.86	**2.56**	**3.50**	6.09	11.26	**1.65**
doubL	1.06	1.61	2.10	2.54	2.93	2.97	4.42	5.72	10.81	1.98
dwicL	1.44	1.99	2.47	2.92	3.32	3.01	4.36	6.44	11.84	2.01
dintL	1.35	1.89	2.38	2.82	3.22	3.03	4.35	6.50	11.90	1.84
dsumL	1.16	**1.17**	**1.19**	**1.20**	**1.24**	4.01	6.34	**5.13**	**7.48**	3.15
2 summands	3.64	3.90	4.12	4.33	4.58	**3.57**	**5.43**	6.12	10.75	**2.52**
4 summands	2.07	2.15	2.22	2.28	2.38	5.20	7.29	6.16	9.64	3.24
8 summands	1.16	1.18	1.19	1.20	1.25	4.01	6.32	5.14	7.52	3.15
12 summands	**1.09**	**1.10**	**1.11**	**1.12**	**1.15**	4.25	7.84	4.87	4.91	3.31
18 summands	1.17	1.17	1.18	1.19	1.22	4.64	8.14	3.64	**4.71**	3.54
24 summands	1.20	1.21	1.21	1.22	1.25	5.18	8.54	**3.45**	4.96	3.80
32 summands	1.27	1.27	1.28	1.29	1.32	5.76	9.67	3.87	5.51	4.12
40 summands	1.31	1.32	1.32	1.34	1.37	6.38	10.83	4.36	6.18	5.24
protO___noP	1.16	1.18	1.19	1.20	1.24	4.02	6.36	5.15	7.50	3.16
expaO___noP	**0.98**	**0.99**	**1.00**	**1.01**	**1.05**	**3.86**	**6.18**	**5.12**	**7.36**	**3.08**
expaO_warnP	1.77	1.79	1.81	1.83	1.89	4.50	6.89	5.28	7.98	3.27
expaO_restP	1.23	1.25	1.26	1.28	1.32	4.07	6.41	5.16	7.58	3.16
__noC	1.93	2.00	2.05	2.10	2.17	4.63	7.31	5.49	7.73	3.25
lazyC	**1.17**	**1.18**	**1.19**	**1.21**	**1.25**	**4.02**	6.36	5.16	**7.49**	**3.16**
laagC	1.24	1.25	1.26	1.28	1.32	4.08	6.42	5.17	7.54	3.20
permC	1.25	1.26	1.27	1.29	1.33	4.14	**6.26**	**5.02**	7.53	3.20
treeM	**1.13**	**1.17**	**1.19**	1.22	1.26	**3.76**	**5.95**	5.88	9.03	**2.88**
expaM	1.17	1.18	**1.19**	**1.21**	**1.24**	4.02	6.36	**5.16**	**7.49**	3.17
nodeM	1.28	1.29	1.31	1.32	1.38	4.53	7.30	5.85	8.40	4.27
protO___noP	**1.16**	**1.18**	**1.19**	**1.20**	**1.24**	**4.02**	**6.36**	**5.15**	**7.50**	3.16
plaiO_restP_permC_treeM	1.62	1.96	2.24	2.48	2.70	4.51	6.97	7.13	9.76	**3.01**
noFMA_protO___noP	1.26	1.28	1.29	1.31	1.35	4.09	6.42	5.16	7.56	3.17
protO___noP	**1.16**	**1.18**	**1.19**	**1.20**	**1.24**	**4.02**	**6.36**	**5.15**	**7.50**	**3.16**

TABLE A.5. Results for floating-point data on thales.

ion			Arrangement of circles								Segment Voronoi diagram			
Cartesian			static algebraic predicates				predicates with $\sqrt[2]{}$							
long	grid	axis	rand	gridrn	pack	gridnn	rand	gridrn	pack	gridnn	mst	sqrs	short	shoax
2.34	3.37	4.42	7.20	13.52	3.48	15.75	**3.72**	**6.83**	4.19	13.16	11.91	5.62	11.28	7.06
2.88	3.52	**1.31**	7.04	13.46	3.35	13.90	3.84	6.98	4.26	12.73	11.26	4.55	10.96	6.18
4.79	**3.06**	3.44	4.87	10.81	**2.56**	12.50	4.71	8.41	3.74	12.44	11.73	**3.50**	**10.93**	**5.80**
2.76	6.85	9.21	41.12	62.89	20.54	39.00	13.15	22.27	42.08	107.55	11.69	26.21	27.92	21.97
5.34	7.39	9.01	14.33	27.79	10.28	24.42	7.37	16.00	11.85	21.68	11.96	18.08	24.06	29.24
–	–	–	–	–	–	–	–	–	–	–	9.38	10.57	–	–
–	–	–	–	–	–	–	–	–	–	–	**8.83**	9.91	–	–
3.61	4.17	2.40	**3.18**	**6.83**	3.68	10.01	4.24	8.61	**3.55**	**8.59**	9.46	13.80	13.84	16.66
10.54	3.82	4.28	8.51	13.15	4.40	**7.73**	13.29	21.96	4.89	10.98	8.93	15.92	17.18	25.83
–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
1.91	**8.42**	**1.24**	**3.26**	**6.93**	**3.99**	**10.01**	**4.86**	**9.81**	**4.78**	**10.04**	**9.37**	**16.53**	**17.29**	**20.55**
2.34	**3.39**	**4.42**	**7.24**	**13.60**	**3.50**	**15.81**	**3.74**	**6.88**	**4.19**	**13.19**	11.97	**5.63**	**11.33**	**7.07**
2.87	3.88	5.39	8.63	16.84	4.37	17.64	4.30	8.12	4.68	14.06	**11.30**	6.57	12.21	8.88
2.34	3.38	4.46	**7.21**	13.58	3.50	15.83	**3.73**	**6.85**	**4.18**	**13.21**	11.94	5.64	**11.29**	7.08
2.50	**3.32**	**1.69**	7.28	**12.34**	**3.46**	**11.62**	3.88	7.05	4.26	18.26	**11.93**	**4.90**	11.61	**6.74**
2.60	**3.57**	5.75	**7.14**	13.82	**3.58**	15.90	**3.88**	**7.14**	**4.32**	13.46	**11.29**	**5.47**	**11.12**	**7.09**
4.03	3.98	**2.31**	7.56	**12.25**	3.69	**11.80**	4.90	8.61	4.73	**12.63**	11.72	5.77	12.84	7.13
2.34	**3.36**	**4.42**	**7.20**	**13.52**	**3.48**	**15.79**	**3.71**	**6.83**	**4.18**	**13.17**	**11.91**	**5.63**	**11.27**	**7.06**
2.31	3.52	4.58	7.53	14.15	3.68	16.25	3.76	6.91	4.28	13.42	11.96	5.77	11.50	7.26
2.33	**3.37**	**4.43**	**7.19**	**13.53**	**3.48**	**15.75**	**3.72**	**6.84**	**4.17**	**13.15**	**11.96**	**5.63**	**11.27**	**7.06**
2.48	6.71	5.98	15.24	24.79	6.60	29.61	4.45	8.16	6.43	20.91	12.07	7.32	15.41	8.85
2.34	3.37	4.42	7.20	13.52	3.48	15.75	**3.72**	**6.83**	4.19	13.16	11.91	5.62	11.28	7.06
2.88	3.52	**1.31**	7.04	13.46	3.35	13.90	3.84	6.98	4.26	12.73	**11.26**	4.55	10.96	6.18
2.90	3.62	5.96	7.13	13.85	3.59	15.81	3.87	7.13	4.26	13.39	11.64	5.53	11.35	7.25
2.61	3.55	5.77	7.13	13.86	3.58	15.88	3.88	7.13	4.33	13.44	11.68	5.53	11.34	7.24
4.79	**3.06**	3.44	**4.87**	**10.81**	**2.56**	**12.50**	4.71	8.41	**3.74**	**12.44**	11.73	**3.50**	**10.93**	**5.80**
3.39	3.63	**2.90**	6.81	13.29	3.35	13.73	**4.41**	**7.67**	4.26	12.69	11.65	4.51	11.29	6.19
5.00	3.80	3.02	5.85	11.99	3.00	13.16	4.55	7.77	4.00	12.48	**11.27**	3.53	**10.54**	**5.40**
4.78	3.07	3.44	**4.90**	**10.84**	**2.58**	**12.57**	4.71	8.44	3.74	12.45	11.28	3.46	10.70	5.66
4.99	**2.97**	3.65	4.93	11.04	2.62	12.61	4.75	8.55	**3.72**	12.52	11.44	**3.43**	10.77	5.74
5.35	3.10	3.90	4.99	10.89	2.67	12.60	4.82	8.60	3.73	**12.41**	11.46	3.55	10.91	5.95
5.76	3.27	4.17	5.31	11.06	2.75	12.74	4.87	8.68	3.74	12.42	11.48	3.64	11.05	6.14
6.26	3.48	4.53	5.28	11.47	2.84	12.83	4.91	8.76	3.77	12.49	11.38	3.78	11.16	6.37
7.14	3.85	5.28	5.44	11.93	2.94	12.95	4.97	8.80	3.79	12.48	11.42	3.92	11.35	6.63
4.79	3.07	3.45	4.88	10.82	2.56	**12.49**	4.72	8.44	3.73	12.44	11.67	3.50	10.94	5.82
4.69	**3.02**	**3.38**	**4.85**	**10.75**	**2.55**	12.54	**4.67**	8.37	**3.71**	**12.42**	**11.33**	**3.36**	**10.65**	**5.56**
4.92	3.11	3.53	4.96	10.95	2.63	12.53	4.80	8.58	3.76	12.45	11.71	3.62	11.07	6.04
4.79	3.07	3.44	4.95	11.26	2.58	14.09	4.72	**8.29**	3.73	13.20	11.34	3.51	10.80	5.71
4.98	3.35	3.44	5.18	11.49	2.86	12.75	4.79	8.47	3.96	12.64	11.71	3.66	10.89	5.74
4.81	3.08	3.45	**4.89**	**10.85**	**2.58**	**12.50**	**4.72**	8.47	**3.73**	**12.45**	11.72	3.50	10.92	5.82
4.85	3.09	3.47	4.94	10.90	2.59	12.52	4.75	8.49	3.75	12.47	11.73	3.52	10.95	5.82
4.82	**3.07**	**3.40**	4.92	10.86	**2.58**	12.52	4.74	**8.46**	**3.73**	12.49	**11.29**	**3.44**	**10.70**	**5.66**
4.41	4.01	3.47	8.42	16.50	3.70	15.23	**4.60**	**8.23**	4.18	13.28	**11.72**	3.97	11.12	6.31
4.81	**3.08**	**3.45**	**4.89**	**10.85**	**2.58**	**12.50**	4.73	8.46	**3.73**	**12.45**	**11.72**	**3.50**	**10.92**	**5.82**
6.59	3.83	3.46	5.98	12.62	2.99	13.14	5.42	9.52	3.99	12.76	11.73	3.69	11.45	6.56
4.79	**3.07**	**3.45**	**4.88**	**10.82**	**2.56**	**12.49**	**4.72**	8.44	**3.73**	**12.44**	**11.67**	**3.50**	**10.94**	**5.82**
4.69	4.77	3.50	8.27	15.11	4.00	14.85	4.73	**8.33**	4.86	13.41	11.72	4.40	11.17	6.42
4.81	**3.07**	3.48	4.89	10.83	2.58	12.51	4.73	8.46	**3.73**	**12.42**	11.77	3.54	10.96	5.84
4.79	**3.07**	**3.45**	**4.88**	**10.82**	**2.56**	**12.49**	**4.72**	**8.44**	**3.73**	12.44	**11.67**	**3.50**	**10.94**	**5.82**

	Delaunay triangulation					Segment intersect-				
						homogeneous				
	0	25	50	75	100	short	long	grid	axis	short
nodaL	0.89	0.90	0.91	0.92	0.94	**2.54**	**3.49**	2.46	9.89	**1.64**
doubL	**0.55**	**0.55**	**0.56**	**0.56**	**0.58**	2.77	4.21	1.92	6.42	1.97
dsumL	0.76	0.76	0.77	0.78	0.80	3.12	4.82	**1.74**	**3.06**	2.72
CORE::Expr 1.8	1.14	1.61	2.38	3.56	5.30	2.95	4.08	3.82	12.65	1.93
CORE::Expr 2.1.1	4.51	4.98	6.10	7.13	8.33	8.66	12.22	6.68	17.92	3.75
Lazy_exact_nt<Gmpzf>	3.56	3.62	3.61	3.66	3.78	4.44	6.41	2.76	6.52	–
Gmpzf	6.56	6.61	6.65	6.71	6.95	7.40	10.62	3.88	7.17	–
Lazy_exact_nt<Gmpq>	3.52	3.57	3.57	3.60	3.73	4.41	6.35	3.23	9.90	2.64
Gmpq	11.75	11.84	11.98	12.14	12.58	12.07	17.12	6.31	11.44	5.29
Epick	**0.07**	**0.06**	**0.07**	**0.07**	**0.07**	–	–	–	–	–
Epeck	0.14	0.14	0.14	0.14	0.15	–	–	–	–	**1.37**
boost_pool	**0.90**	**0.91**	**0.91**	**0.92**	**0.95**	**2.55**	**3.50**	**2.47**	**9.90**	**1.64**
no_pool	2.55	2.56	2.59	2.61	2.67	3.82	5.31	3.30	12.05	2.04
ledaF__nodaL	**0.90**	**0.90**	**0.92**	**0.92**	**0.95**	**2.55**	**3.49**	2.47	9.96	**1.65**
boostF_nodaL	1.28	1.29	1.30	1.32	1.36	2.82	3.88	**2.06**	**7.56**	1.77
ledaF__dintL	**0.71**	**0.71**	**0.72**	**0.73**	**0.75**	**2.99**	**4.43**	**2.79**	9.99	**1.81**
boostF_dintL	2.73	2.74	2.77	2.80	2.88	5.03	7.50	3.17	**9.10**	2.79
mpfrA	**0.90**	**0.90**	**0.91**	**0.92**	**0.95**	**2.54**	**3.49**	**2.46**	**9.87**	1.64
pure_mpfrA	0.91	0.91	0.92	0.93	0.96	**2.54**	**3.49**	2.54	10.33	**1.63**
basicD	**0.89**	**0.90**	**0.91**	**0.92**	**0.94**	**2.55**	**3.48**	**2.46**	**9.87**	**1.64**
oledaD	1.04	1.05	1.05	1.06	1.09	2.75	3.79	3.43	16.13	1.73
nodaL	0.89	0.90	0.91	0.92	0.94	**2.54**	**3.49**	2.46	9.89	**1.64**
doubL	**0.55**	**0.55**	**0.56**	**0.56**	**0.58**	2.77	4.21	1.92	6.42	1.97
dwicL	0.92	0.91	0.94	0.93	0.97	3.03	4.48	2.79	9.89	2.02
dintL	0.71	0.71	0.72	0.73	0.75	2.98	4.42	2.78	9.96	1.81
dsumL	0.76	0.76	0.77	0.78	0.80	3.12	4.82	**1.74**	**3.06**	2.72
2 summands	1.31	1.32	1.32	1.33	1.37	4.49	6.52	2.51	7.82	2.60
4 summands	**0.71**	**0.71**	**0.72**	**0.73**	**0.75**	4.07	7.47	2.71	4.79	**2.33**
8 summands	0.76	0.77	0.77	0.78	0.80	**3.10**	**4.81**	**1.74**	**3.06**	2.72
12 summands	0.78	0.78	0.79	0.80	0.82	3.32	5.11	1.86	3.31	2.88
18 summands	0.82	0.82	0.83	0.84	0.86	3.70	5.98	2.09	3.88	3.11
24 summands	0.85	0.86	0.86	0.87	0.89	4.19	6.32	2.33	4.17	3.37
32 summands	0.91	0.91	0.92	0.93	0.95	4.62	7.15	2.64	4.72	3.70
40 summands	0.95	0.96	0.97	0.97	1.00	5.22	8.11	3.01	5.39	4.81
protO___noP	0.76	0.77	0.77	0.78	0.80	3.11	4.81	1.74	3.07	2.72
expaO___noP	**0.58**	**0.58**	**0.59**	**0.59**	**0.61**	**2.92**	**4.53**	**1.64**	**2.89**	**2.65**
expaO_warnP	1.37	1.38	1.39	1.40	1.45	3.75	5.77	2.07	3.69	2.81
expaO_restP	0.83	0.84	0.84	0.85	0.87	3.16	4.92	1.77	3.13	2.71
__noC	**0.67**	**0.68**	**0.68**	**0.69**	**0.71**	3.14	5.11	1.77	**3.01**	**2.70**
lazyC	0.76	0.77	0.77	0.78	0.80	**3.12**	**4.82**	**1.74**	3.08	2.73
laagC	0.79	0.79	0.80	0.81	0.83	3.16	4.89	1.77	3.12	2.75
permC	0.75	0.76	0.76	0.77	0.79	3.30	5.15	1.85	3.31	2.72
treeM	**0.76**	**0.77**	**0.77**	**0.78**	**0.80**	3.11	4.80	**1.73**	3.06	**2.46**
expaM	**0.76**	**0.77**	**0.77**	**0.78**	**0.80**	3.12	4.82	1.74	3.08	2.73
nodeM	**0.76**	**0.77**	**0.77**	**0.78**	**0.80**	**3.10**	**4.79**	1.74	**3.05**	3.21
protO___noP	**0.76**	**0.77**	**0.77**	**0.78**	**0.80**	**3.11**	**4.81**	1.74	**3.07**	2.72
plaiO_restP_permC_treeM	1.01	1.01	1.02	1.03	1.06	3.68	5.90	2.09	3.52	**2.48**
noFMA_protO___noP	0.88	0.89	0.89	0.90	0.93	3.23	4.96	1.81	3.19	2.75
protO___noP	**0.76**	**0.77**	**0.77**	**0.78**	**0.80**	**3.11**	**4.81**	**1.74**	**3.07**	**2.72**

TABLE A.6. Results for integer data on thales.

ion			Arrangement of circles								Segment Voronoi diagram			
Cartesian			static algebraic predicates				predicates with $\sqrt[2]{\ }$							
long	grid	axis	rand	gridrn	pack	gridnn	rand	gridrn	pack	gridnn	mst	sqrs	short	shoax
2.34	1.48	4.23	5.78	11.03	2.59	12.35	3.30	6.43	2.38	11.50	11.79	2.71	10.47	5.82
2.95	**1.36**	**1.31**	5.03	8.00	2.28	7.69	**3.20**	**5.84**	**2.01**	8.73	11.15	1.61	**8.95**	**4.47**
4.04	1.71	2.29	4.47	9.33	**2.26**	9.55	3.78	7.24	2.16	10.38	11.65	**1.58**	9.41	4.99
2.75	2.89	9.18	40.38	59.77	19.04	33.05	6.37	11.88	15.22	35.49	11.17	8.27	22.92	18.90
5.35	3.45	9.03	13.44	25.65	8.58	18.84	6.77	14.54	6.51	15.45	11.56	6.29	21.63	25.16
–	–	–	–	–	–	–	–	–	–	–	8.79	4.40	–	–
–	–	–	–	–	–	–	–	–	–	–	8.46	3.46	–	–
3.61	1.86	2.40	**3.08**	**6.14**	2.41	7.04	4.09	7.92	2.38	**6.21**	8.54	5.61	11.77	15.07
7.37	2.74	4.35	7.39	11.74	3.88	**5.98**	12.17	20.55	4.45	9.36	**8.28**	5.06	14.76	23.46
–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
1.90	**1.79**	**1.25**	**3.16**	**6.27**	**2.65**	**7.08**	**4.72**	**9.01**	**3.12**	**7.53**	**8.45**	**6.49**	**15.12**	**18.85**
2.33	**1.48**	**4.24**	**5.80**	**11.09**	**2.60**	**12.37**	**3.32**	**6.45**	**2.39**	**11.53**	11.88	**2.72**	**10.49**	**5.83**
2.87	1.79	5.20	7.14	13.95	3.41	14.04	3.89	7.63	2.81	12.40	**11.12**	3.10	11.33	7.62
2.34	1.49	4.27	**5.81**	11.07	**2.61**	**12.40**	**3.31**	**6.43**	**2.38**	**11.54**	11.87	2.72	**10.46**	5.83
2.51	1.37	**1.66**	6.00	**10.78**	2.82	**9.41**	3.48	6.60	2.73	16.40	**11.80**	**1.85**	10.78	**5.65**
2.65	**1.64**	5.27	**5.31**	10.87	**2.42**	12.11	**3.25**	**6.52**	**2.14**	11.74	**11.22**	2.65	**9.10**	**5.59**
4.12	1.83	**2.31**	5.62	**8.96**	2.64	**8.41**	4.22	7.77	2.52	**9.66**	11.61	**1.84**	10.64	6.15
2.34	**1.48**	**4.23**	**5.78**	**11.04**	**2.59**	**12.34**	**3.30**	**6.42**	**2.37**	**11.47**	11.83	**2.71**	**10.46**	**5.82**
2.31	1.52	4.38	6.04	11.58	2.73	12.78	3.34	6.49	2.43	11.69	**11.82**	2.78	10.61	5.99
2.34	**1.48**	**4.23**	**5.78**	**11.03**	**2.59**	**12.35**	**3.30**	**6.41**	**2.38**	**11.50**	**11.79**	**2.71**	**10.45**	**5.82**
2.47	1.92	5.71	9.50	16.15	4.26	23.40	3.84	7.35	3.55	17.93	11.86	3.72	13.11	6.72
2.34	1.48	4.23	5.78	11.03	2.59	12.35	3.30	6.43	2.38	11.50	11.79	2.71	10.47	5.82
2.95	**1.36**	**1.31**	5.03	**8.00**	2.28	**7.69**	**3.20**	**5.84**	**2.01**	**8.73**	**11.15**	1.61	**8.95**	**4.47**
2.98	1.75	5.51	5.38	10.99	2.45	12.05	3.30	6.58	2.16	11.72	11.56	2.68	9.37	5.76
2.66	1.64	5.32	5.32	10.93	2.42	12.09	3.25	6.52	2.14	11.73	11.52	2.69	9.32	5.74
4.04	1.71	2.29	**4.47**	9.33	**2.26**	9.55	3.78	7.24	2.16	10.38	11.65	**1.58**	9.41	4.99
3.87	1.56	**1.76**	4.81	9.65	2.33	9.74	3.62	6.81	2.15	10.25	11.54	1.71	9.23	4.71
3.48	**1.48**	1.89	4.53	**9.00**	**2.20**	**9.41**	**3.43**	**6.57**	**2.05**	**10.09**	11.28	**1.51**	**9.05**	**4.59**
4.03	1.70	2.29	**4.50**	9.39	2.27	9.63	3.79	7.27	2.15	10.38	**11.27**	1.54	9.18	4.86
4.30	1.81	2.48	4.58	9.54	2.32	9.65	3.84	7.37	2.16	10.38	11.32	1.57	9.23	4.94
4.68	1.96	2.74	4.71	9.79	2.41	9.77	3.92	7.45	2.19	10.46	11.33	1.60	9.36	5.14
5.08	2.11	3.02	5.02	9.96	2.49	9.97	3.97	7.54	2.20	10.48	11.37	1.62	9.49	5.31
5.58	2.31	3.37	5.00	10.33	2.58	9.99	4.02	7.62	2.23	10.51	11.37	1.65	9.61	5.54
6.47	2.67	4.11	5.15	10.81	2.69	10.12	4.05	7.65	2.25	10.52	11.40	1.69	9.77	5.78
4.06	1.72	2.29	4.48	9.36	2.26	**9.56**	3.80	7.27	2.16	10.38	11.60	1.58	9.41	5.00
3.94	**1.67**	**2.22**	**4.45**	**9.30**	**2.25**	**9.56**	**3.75**	**7.20**	**2.14**	**10.36**	**11.23**	**1.53**	**9.13**	**4.77**
4.17	1.75	2.37	4.56	9.48	2.32	9.58	3.87	7.40	2.20	10.42	11.66	1.62	9.54	5.25
4.05	1.76	2.29	5.08	11.48	2.52	11.11	3.90	7.45	2.41	11.29	11.33	1.55	9.36	5.04
4.04	**1.70**	2.27	**4.47**	**9.33**	**2.25**	**9.50**	**3.78**	**7.25**	**2.14**	**10.34**	11.64	1.57	9.36	4.94
4.06	1.71	2.29	4.49	9.38	2.27	9.53	3.80	7.27	2.16	10.39	11.67	1.58	9.39	5.00
4.09	1.72	2.31	4.52	9.40	2.28	9.56	3.81	7.30	2.16	10.36	11.67	1.59	9.40	5.00
4.05	**1.70**	**2.25**	4.50	9.36	2.27	9.53	3.79	7.28	2.16	10.35	**11.28**	**1.54**	**9.13**	**4.80**
3.67	**1.67**	2.30	5.48	10.89	2.50	10.22	**3.69**	**7.16**	2.18	10.46	11.67	1.65	9.41	5.01
4.06	1.71	**2.29**	**4.49**	**9.38**	**2.27**	**9.53**	3.80	7.28	**2.16**	**10.39**	11.67	**1.58**	**9.40**	**5.00**
4.82	1.92	2.30	4.85	9.89	2.36	9.79	4.02	7.48	2.20	10.46	**11.66**	1.62	**9.40**	5.04
4.06	**1.72**	**2.29**	**4.48**	**9.36**	**2.26**	**9.56**	3.80	**7.27**	**2.16**	**10.38**	**11.60**	**1.58**	**9.41**	**5.00**
3.69	1.73	2.35	5.79	11.12	2.68	10.62	**3.75**	7.30	2.23	10.65	11.64	1.66	9.55	5.26
4.09	**1.72**	2.33	4.50	9.37	2.28	9.58	3.82	7.31	2.17	**10.38**	11.68	1.59	9.42	5.04
4.06	**1.72**	**2.29**	**4.48**	**9.36**	**2.26**	**9.56**	**3.80**	**7.27**	**2.16**	**10.38**	**11.60**	**1.58**	**9.41**	**5.00**