THE EXACT COMPUTATION PARADIGM [1]

CHEE YAP
Courant Institute of Mathematical Sciences
New York University, New York, NY 10012

and

THOMAS DUBÉ
The College of the Holy Cross
Worcester, MA 10610

ABSTRACT

We describe a paradigm for numerical computing, based on exact computation. This emerging paradigm has many advantages compared to the standard paradigm which is based on fixed-precision. We first survey the literature on multiprecision number packages, a prerequisite for exact computation. Next we survey some recent applications of this paradigm. Finally, we outline some basic theory and techniques in this paradigm.

[1]This paper will appear as a chapter in the 2nd edition of **Computing in Euclidean Geometry**, edited by D.-Z. Du and F.K. Hwang, published by *World Scientific Press*, 1994.

1. Two Numerical Computing Paradigms

Computation has always been intimately associated with numbers: computability theory was early on formulated as a theory of computable numbers, the first computers have been number crunchers and the original mass-produced computers were pocket calculators. Although one's first exposure to computers today is likely to be some non-numerical application, numerical computation remains a major application of computers. In particular, the emerging inter-disciplinary area of *scientific computation* is predominantly numerical. A newly inaugurated magazine from IEEE, *Computational Science and Engineering*, covers the spectrum of activities defining the subject (see the lead article by Rice [67] and others). Some observers of science are beginning to call "computation" as represented in scientific computing the "third scientific method", adding to the traditional methods of theory and experimentation [31]. The bulk of scientific computation is based on the *fixed precision paradigm* (f.p. paradigm) of computation. In the simplest form, this paradigm assumes that some fixed precision (usually machine-dependent) is given for the representation of numbers. *Subject to this requirement*, the user may use whatever devices suitable for reducing the inevitable rounding errors. This leads to two basic activities: analysis of the errors in f.p. computations and study of techniques to restructure f.p. computations to minimize the errors. A *mild form* of fixed-precision allows the user to specify a precision in advance of a computation; subsequent operations are carried out to that precision and the algorithm does not adapt to the precision used. Many multiprecision packages (for instance, Brent's `MP` [16] or Bailey's `MPFUN` [5], reviewed in the next section) are designed to be used under this mild form of f.p. paradigm.

Today, the de facto standard of the fixed precision paradigm is some form of *floating-point representation*. As such, the reader would not go far astray by reading the abbreviation "f.p." as "floating-point". It turns out that the best way to implement floating-point arithmetic is far from obvious. Consequently, with the proliferation of computer architectures this gives rise to severe problems of portability of numerical code. The widely accepted IEEE standard for f.p. arithmetic (see [65, Appendix A]) is a big step towards containing this issue of portability. In effect, it makes the errors in f.p. computation *machine-independent*. But the fundamental source of error in f.p. computation remains intact.

There are other ways of using any given fixed amount of precision. In fact, floating-point representation was originally considered and rejected by von Neumann in favor of the fixed-point representation. It is generally attributed to the extensive backward error analyses of Wilkinson that floating-point computation became understood and widely accepted [43]. Specialized floating-point hardware has become a standard in modern workstations. The "idea" of floating-point representation is to decouple significance from magnitude: with a given precision, floating-point numbers can represent both numbers of large magnitude with few bits of significance or conversely, small magnitude and many bits of significance. We can take its idea to a

logical limit – one version of this idea is the *level index arithmetic* [23] (or, a variant called *symmetric level-index arithmetic* [24]). Usually, these systems are developed within the fixed-precision framework; in this case, it must eventually face the same problems of round-off error. Implementation of these arithmetics are expected to be more complicated, but see [78].

Nonrobustness Issues. Even with the best designed standards, rather intractable problems of rounding errors seems to haunt the f.p. paradigm. Notice that rounding errors in individual arithmetic operations is generally not regarded as problematic per se; what concerns us is the build-up of errors and their complicated interaction within the logic of a computation that ultimately leading to *catastrophic errors*. We can classify the approaches to "robust f.p. algorithms" under two broad categories.

(I) *Arithmetic Level:* A large variety of techniques such as the use of double extended precision, guard-bits, gradual underflow, etc, can be introduced to more effectively use the available precision. These can often be implemented in hardware. See the survey [35] for a discussion of these techniques and the IEEE standard. An interesting idea from Bohlender, et al [11]) is the *exact floating point semantics* in which an arithmetic operation returns the lower order bits as well as the usual higher order bits. Even here, rounding errors occur for the division and square-root operation. The use of level index arithmetic noted above seems to go a long way towards delaying the inevitable catastrophic build-up of errors. Interval arithmetic is another well-known approach.

(II) *Geometric Level:* Hoffmann [49] surveys several of the approaches here. For instance, a fairly general paradigm is to classify the data (input as well as computed) into combinatorial and numerical, and to give primacy to the former when making decisions. One interpretation of this idea is that we are willing to perturb the numerical data in order to preserve the combinatorial data, thereby avoiding "topological inconsistency". This is possible in simple cases, but in general, reasoning about the combinatorial structures seems intractable. Another approach is to invent some kind of "approximate geometry" and map the original geometry into this approximation. Unfortunately, most proposed approximate geometries (even for a concept such as a "line" [85]) have limited applicability and are invariably difficult to reason with.

In [85], we concluded that "robustness" for geometric algorithms (except in limited contexts) is generally infeasible under the f.p. paradigm. This seems justifiable in the face of the wide-spread failure to give a satisfactory solution to non-robustness issues. In fact, even without asking for a general solution, robust solutions to specific key problems are not forthcoming (e.g., Yu [87, abstract and conclusion] suggests that the prospects for robust Boolean algorithms for polyhedral solids are still in the distant future). Despite this negative conclusion, it is evident that f.p. computation is extremely useful. This is especially true for many areas of scientific computation where the data itself is inexact, and the phenomenon to be modeled is too complex for exact answers. The role of computation in such applications is, to paraphrase a

dictum of the numerical analyst Hamming, "to produce insight, not numbers". In such situations, even the widespread but ad hoc practice[2] of "epsilon tweaking" has some value.

Alternative to f.p. paradigm. But what shall we do in the applications and occasions when we really would need to know exact answers (bearing in mind that practically no answers derived from a reasonably complex f.p. computation are guaranteed)? This brings us to what we believe to be the antidote to non-robustness: the *exact computation paradigm*. In the simplest form, this paradigm guarantees that (a) all numerical values are represented exactly and (b) all branching decisions are error-free. It is immediate from (a) and (b) that *multiprecision arithmetic* is a prerequisite for exact computation. But notice that the use of multiprecision arithmetic per se is no guarantee that a computational method is exact.

There is naturally a price to be paid for using exact computation, notably a slow-down in performance. This is unavoidable for some applications. Many problems in computer algebra and computational number theory falls under this category because only exact answers make sense (e.g., testing irreducibility of a polynomial). There are also applications for which we cannot afford any significant slow-down – we say such applications are *cycle-critical*. Clearly, calling an application "cycle-critical" is a matter of degree and depends on the computational resources available. With increasing computing power, more and more applications will no longer be cycle-critical. Other factors such as robustness, user-friendliness, reusability of code become more important for non cycle-critical applications. Of course, our interest is in applications where robustness is at issue. The simple form of the f.p. paradigm was developed in the early days of computers for the good reason that most applications *then* appears to be cycle-critical. To see how far we have advanced, there was a time when each single-precision multiplication is considered a valuable resource! One early advocate of exact computation is G.E. Collins who said in a slightly different context [25] that using integers of arbitrary size is "natural since computers are now fast enough that any restrictions imposed by the word length of the computer are artificial and unnecessary". Still, we must acknowledge that cycle-critical applications will always be with us. Therefore it is important to target exact computation for those application areas for which continued reliance on the f.p. paradigm no longer makes sense.

Weak form of exact computation. One could do exact computation in a *weak form*: before the actual computation begins, one precomputes some precision t to represent numbers and then one proceeds almost as in a f.p. computation. This weak form suffices in many applications. For instance, to compute the determinant of an $n \times n$ matrix, setting the precision t to $nL + n \lg n$ bits (assuming the input numbers

[2]We refer to the existence of small constants (epsilons) in f.p. codes that practitioner use to decide when a number ought to be regarded as zero. These epsilons are determined empirically, and clearly the situation cries out for some principled approach.

are L bit integers) is sufficient to give the exact answer, assuming the straightforward method of evaluating a determinant as a polynomial in $n!$ terms. This method is useful when n is a small constant ($n \leq 5$ for many problems in computational geometry). But this begs the question: how is the previous "mild form" of f.p. paradigm different from this "weak form" of exact paradigm? Sometimes there is no real difference. For example, in the case of evaluating determinants, if n were fixed and we happen to pick a precision large enough so that the result is exact, then it is pedantic to distinguish the 2 forms. On the other hand, if n is variable, then the mild form can never choose a precision that will give exact answers for all inputs. Ultimately, what makes a computation fall under the exact paradigm is in the intent of the user. The intent of the mild f.p. paradigm is best summed up as follows. Brent [16, p. 65], commenting on what it means to compute to precision t using his package `MP`, says: "our aim was not to provide routines which always gives t correctly rounded digits; there is no need for this because t may easily be increased if necessary". Presumably, most users pick t by trial and error. In contrast, the user of the weak exact computation paradigm must precede his algorithm design with an analysis of the value of t sufficient to ensure exact results. Although it is not always obvious what an optimal t should be, the theory of root bounds (section 6) can give safe choices.

We believe that the exact computation paradigm will grow[3] in importance, beyond its traditional domains such as computer algebra. In order to make the exact paradigm take[4] its "rightful place" next to the f.p. paradigm, many new computational techniques must be developed. Perhaps the first step is to disspell the naive view that exact computation is "only" about building better multiprecision packages, although that is a prerequisite. Instead, we have to examine the support of higher-level constructs. Indeed, the higher level constructs may lead to modified organization of traditional multiprecision packages. One may say that the fundamental goal of exact computation research is that of controlling numerical precision: the term *precision-driven computation* may be used to describe this goal. In section 6, we see one manifestation of this idea.

In this article, we shall survey and explore the fundamental ideas of exact computation. In section 2, we review the literature on arbitrary precision number packages. Next we survey previous work that applies exact computation in section 3. This is mostly work in computational geometry although we noted some other interesting areas. We outline a theoretical framework for exact computation in section 4. In sections 5 and 6, we describe some fundamental techniques of exact computation. We conclude in section 7.

[3]Given that the present dominant position of the f.p. paradigm seems unassailable [65, Goldberg, p. A-12], perhaps this remark cannot be proved wrong.

[4]Roughly speaking, the "rightful place" amounts to having both paradigms equally developed so that given any application, and at any specified computational power/resource, one can evaluate whether to use one or the other paradigm.

2. Arbitrary Precision Numbers

Since arbitrary precision numerical computation is the underpinning for exact computation, it behooves us to survey the literature on their implementation. This is mostly implemented in software, perhaps embedded into a computer language. We will not treat hardware implementations, which usually amount to specialized hardware for very large (rather than arbitrary size) integers, often used in applications such as cryptography. For instance, [20] describes a 256-bit ALU that can be dynamically reconfigured by a program. Furthermore, we will only discuss the 4 basic arithmetic operations, even if the packages have more advanced features such as algebraic number manipulations (e.g., [1]).

2.1. Number Representations

Depending on the application domain, numerical computing requires the manipulation of integer, rational, real and complex numbers. Since the earliest days of `FORTRAN`, computer programmers have been offered an option of two basic numeric data types: the fixed-size integers and fixed-size floating-point values. These facilities correspond roughly to the mathematical concepts of integers and reals (indeed, `FORTRAN`, `Pascal` and others call these fixed-size representations `Integer` and `Real`). These representations are well-known, but we will mention a few points about them to illustrate some issues which will arise later.

Fixed-Size Integers. The fixed size representation using m bits can only represent 2^m different values. Since the integers are well ordered, it seems natural to represent a set of contiguous values, typically the signed values $-(2^{m-1}) \ldots (2^{m-1} - 1)$ or the unsigned values $0 \ldots (2^m - 1)$. There are several different ways to represent signed values using the available bits:

1. sign and magnitude
2. one's complement
3. two's complement
4. implicit subtraction

As long as the set of values are the same, and the results of the operations are the same, these differences in representation are unimportant to the user of the fixed-size integer type. But no finite subset of numbers (except the trivial set $\{0\}$) is closed under addition and multiplication. In case of fixed-size integers, when an operation results in a value that is not representable, we say that an *integer overflow* has occurred. The system can

1. silently provide an incorrect value (perhaps the best possible),

2. provide an incorrect value, but warn the user by raising an exception or setting a condition flag, or

3. warn the user, and make the correct answer available in an alternate representation (often as a pair of m-bit integers representing the high-order and low-order bits).

As a minor concession to the need for different precisions in computation, the user is often presented with a limited choice for the number m of bits. Usually $m \in \{16, 32, 64\}$. If integer overflow occurs using a smaller size, the user can repeat the calculation using larger fixed-size integers.

Fixed-Size Floating-Point Numbers. Floating-point numbers represent an attempt to provide a larger range of numbers. Each system F of *fixed-size* floating-point numbers is characterized by four integer parameters,

$$F = F(b, p, emin, emax)$$

where b is the base, p the precision or length of the significand, $emin$ and $emax$ are the minimum and maximum exponent. A number $x \in F(b, p, emin, emax)$ is represented by a pair (e, f) of integers where $emin \leq e \leq emax$ and $f = \pm(d_{-1}d_{-2}\cdots d_{-p})_b$ in base b notation. The real number represented by (e, f) is $\pm(0.d_{-1}d_{-2}\cdots d_{-p})_b \cdot b^e$. Typically, b is 10 (for ease of human interface) or some power of 2 (for efficient implementation). In a fixed-size representation using m bits, there is once again at most 2^m different values that can be represented. After choosing b, the choice of the parameters $p, emin, emax$ is principally one of trading off higher precision with a larger range of numbers that can be approximated. Again, operations can result in non-representable numbers. There are now two possibilities: as in the integer case, we have *overflow* if the magnitude of the result is too large. We can also have *underflow* if the value falls in between two consecutive representable numbers. For instance, $(0.12)_{10}$ and $(0.0034)_{10}$ are representable numbers in $F(10, 2, -2, 2)$, but their sum $(0.1234)_{10}$ is not representable. (Our examples normally assumes base 10, so the subscript 10 may be omitted.) We must then systematically pick one of the two neighboring representable numbers, 0.12 or 0.13, as the *rounded result*. The computation is generally not halted for underflows, except possibly when the underflow occurs in the neighborhood of zero. In the excellent survey [42], Goldberg describes this rounding as the "characteristic feature of floating-point computation".

Rump ([69], cited in [2]) provides a good example of an expression which is evaluated incorrectly using ordinary floating-point arithmetic. Let

$$f \;=\; 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b}$$

with

$$a = 77617.0 \quad \text{and} \quad b = 33096.0.$$

Rump evaluated this expression using `FORTRAN` on an IBM System 370 mainframe using single, double, and extended precision. In all three cases, the computed value began with the digits:

$$f = 1.172603.$$

Unfortunately, this *consensus* valuation is incorrect. The correct value (to 40 decimal digits) is

$$f = -0.8273960599468213681411650954798162919999.$$

We duplicated this test using double-precision IEEE floating-point on a SPARC workstation. It evaluated f as $-1.18059e+21$. The sign is right, but the magnitude is a little bit too large.

We conclude this section by noting that there are alternative representations of real numbers. Usually, they amount to giving the set of representable numbers an even greater dynamic range. The most notable of these is *level-index arithmetic* [23] (or, a variant called *symmetric level-index arithmetic* [24]). Usually, these systems are developed within the fixed-precision framework; in this case, it must eventually face same problems of round-off error. Implementation of such arithmetic is expected to be more complicated, but see [78].

2.2. Multiprecision integer and rational numbers

Multiprecision integers is the computer version of the mathematical concept: barring storage limitations, every integer can be represented. In practice, the storage limitation of a computer is often not the most critical in computations involving such integers. Hence, implementors often find it convenient to put some upper limit on the maximum size of these integers.

The representation of multiprecision integers naturally requires a sequence of fixed-sized machine integers. There are two obvious ways to implement this in conventional programming languages: as a linked list or as an array. The former is more flexible but uses more storage and is slower because of the additional indirection of pointers. It is possible to combine these two approaches at the expense of the complexity of the algorithms. Traditionally, the library routines implementing multiprecision integers are called "BigNumber" or "BigInteger" packages. The extension from multiprecision integers to rational numbers is usually straightforward, so we will normally only focus on the integer case.

BigNumber packages can further be distinguished between those that fixes a maximum length for the integers (such systems are likely to use arrays) versus those that dynamically allocate storage for the integers as needed (linked lists is ideal for this). The former seems to be useful when we want efficiency and exploit the special nature of the computation. For instance, this is useful for bounded-depth geometric algorithms [85, 40]

One of the earliest papers on BigNumber packages is the 1960 paper by Pope and Stein [66]. Most computer systems provides at least one such package in their

library. Unix systems distributed from AT&T comes with a package called `mp`. A carefully designed and exceptionally well-documented system [73] called `BigNum` was jointly developed by INRIA and Digital Paris Research Laboratory. This system, which is available via anonymous ftp, tries to balance the demands of efficiency and portability: it is written in `C` with a small kernel in assembly.

There is another direction that is worth mentioning. Most BigNum packages assumes the base b representation of integers, where b is a parameter usually chosen to be a power of 10 or of 2. A number n is represented by a sequence of digits, $(d_\ell, d_{\ell-1}, \ldots, d_1, d_0)$ where $d_i \in \{0, 1, \ldots, b-1\}$. For theoretical reasons (e.g., [81]) as well as for efficiency (e.g., [40]), it is often desirable to use a redundant representation.

2.3. Multiple-Precision Floating-Point Arithmetic

Since the exponent and significand of floating-point numbers are integers, the idea of multiple-precision floating-point numbers follows naturally from multiple-precision integers. In fact, Pope and Stein [66] mentions the possibility of extending multiple-precision numbers to floating-point.

Using more bits of precision allows one to compute results with greater accuracy, but it does not by itself offer any guarantee about how many digits of the result are indeed accurate. One method of bounding the accumulated error in an expression is to use interval analysis, in which the actual value of an expression is bounded between two values that form the end-points of the interval. Moore suggested in 1966 [61] that multiple precision floating-point values might be appropriate for use as the interval end-points.

2.4. `FORTRAN` *Based Multi-Precision Floating-Point Packages*

2.4.1. Brent's `MP`

For many years, the most widely used multi-precision floating-point package was Brent's `MP` [16, 15]. `MP` allows users to explicitly specify the base and the number of words used in representing the significand of a floating-point number. Using more bits allows one to represent more values. There is still the problem that operations on t-bit numbers produce results which are not representable using t bits. However, it is hoped that by starting with sufficiently many bits the final results will have some validity. This mimics the improvement in accuracy which one gains by moving from single to double precision, but it does not eliminate the fundamental problems of floating-point numbers. Brent suggests that if one wants exact results then one should use multi-precision rational numbers. The `MP` package provides a complete alternative for machine floating-point numbers, including algorithms for trigonometric functions, exponentials, logarithms, the gamma function, Bessel functions, etc. Although this package is now 16 years old, it continues to be used today.

2.4.2. Smith's FM

New FORTRAN-based big-float packages are still being written. One recent package FM [75] is tuned for significands of moderate size. Smith acknowledges the fact that MP uses asymptotically faster algorithms, but claim to outperform MP in some benchmark tests. This system uses guard digits to assure that each binary operation results in the nearest representable number. This helps to reduce the severity of round-off errors, though the errors ars still allowed to accumulate unchecked. The FM system provides special symbols for +OVERFLOW, -OVERFLOW and UNKNOWN to help aid recovery from exceptions such as overflow and division by zero. The set of operations available in this package once again covers all of the standard functions that one would expect from a fixed precision floating-point replacement.

2.5. Bailey's MPFUN

While FM is fine-tuned for numbers of moderate precision, MPFUN by Bailey [4, 5] is a system designed for high-precision computations. One motivation for MPFUN was to study the digits of some mathematical constants including Euler's gamma, $\log \pi$, $e + \pi$, and the Feigenbaum delta. MPFUN represents floating-point numbers with a base of either 2^{22} or 2^{24} depending on the system. Fast multiplication is implemented using discrete Fourier transforms. Bailey reports that this gives the best performance among the various multiplication algorithms which was tried, including Karatsuba-Winograd. Division is implemented using Newton-Raphson iteration to find $\frac{1}{b}$. The package includes functions for square root, cube root, exponentiation, polynomial root finding, trig functions and logarithms. The MPFUN package also includes an implementation of complex numbers.

Benchmark testing shows that MPFUN can significantly out-perform Mathematica and MP. Bailey also compares his timing against machine double-precision, and reports being 135 times slower for finding FFT to 40 digits. Since MPFUN is currently being distributed with the popular unix implementation, Linux, MPFUN is probably now the most widely distributed (if not most widely-used) multi-precision floating-point library. To accompany MPFUN is a translator for converting FORTRAN-77 programs into programs that use the MPFUN library. With this translator, users need not code all arithmetic operations as calls to library functions. Instead, they can write more typical FORTRAN expressions and allow the translator to insert the necessary library calls. To control the translation, the input programs may be interspersed with directives which specify:

- the number of bits of precision in a variable,
- the number of bits to be used for output,
- whether expressions are evaluated using MPFUN or double precision (FAST option),

- the epsilon to be used when comparing two values for equality.

Bailey reports running several benchmark programs through the translator and finding that the performance is not significantly different from "hand-coded" `MPFUN` solutions to the same problems.

2.6. Multi-Precision Floating-Point in Programming Languages

`MPFUN` with the automatic translator represents one recent system which incorporates multi-precision floating-point directly into a programming language. Making the arithmetic package part of the language eases the use of multi-precision floating-point numbers. It frees the users from the needed library calls, and allows them to concentrate on the algorithms for solving the problems. This has long been the case with multiple precision integers, which have been incorporated into languages such as Lisp and Smalltalk. One of the first languages to include multi-precision floating-point was IBM's `ACRITH` scientific computation package. This included a high-level language similar to `FORTRAN` for gaining access to the computation engine. The `ACRITH` system was used as the basis for `FORTRAN-SC` [59, 83] which incorporated the package directly into a dialect of `FORTRAN`.

Other programming languages were also modified to include multiple-precision floating-point as a built-in data type. These include the dialects `Pascal-SC` [80, 12] and `Numerical Turing` [51]. `Numerical Turing` makes use of `CADAC` arithmetic [52]. This arithmetic was designed to give the user a clean interface to control the multi-precision arithmetic. `CADAC` uses an internal decimal representation so that constants and input will be represented exactly. It also allow users to specify the precision of variables, and provides functions to check the precision of a variable.

2.7. Multi-Precision Floating-Point in Symbolic Algebra Systems

There are quite a number of computer algebra systems today, all having their own multi-precision number packages. Here we will review a few of the well-known systems.

2.7.1. `Lisp`-based systems

Big-Float packages are available in the popular computer algebra systems `MACSYMA` and `Reduce` which are Lisp-based. Sasaski, the author of the `Reduce` package, reports [70, 54] that these systems are only slightly slower than Brent's `FORTRAN` implementation, and have the advantage of being much easier to use because Lisp handles the memory management rather than leaving it up to the user. Sasaski's system has the additional property that the number of bits used to represent the result of an operation need not be set a priori, but is chosen automatically by the system.

2.7.2. Maple

The computer algebra system `Maple` also includes a full implementation of a multi-precision floating-point package. `Maple` will sometimes truncate the number of digits printed in evaluating an expression, if it knows that they are not accurate, but this is not always the case.

In the example from Rump, the expression is evaluated using 20, 30, and 40 digits of accuracy. Using 40 digits of accuracy, `Maple` prints 40 digits which are all correct. Using 30 digits of accuracy, 30 digits are printed which are all wrong.

```
>f := (a,b) -> 1335/4*b^6+a^2*(11*a^2*b^2-b^6-121*b^4-2)+11/2*b^8+a/(2*b);
>a := 77617.0;
                        a := 77617.0
>b := 33096.0;
                        b := 33096.0
>Digits := 20;
                        Digits := 20
>f(a,b);
                                        17
              -.99999999999999998827*10

>Digits := 30;
                        Digits := 30
>f(a,b);

                                                  8
            .10000000117260394005317863185 88*10

>Digits := 40;
                        Digits := 40
>f(a,b);

          -0.8273960599468213681411650954798162919 99
```

2.7.3. Mathematica

The multi-precision floating-point package in `Mathematica` maintains the number of valid digits in an expression. Using Rump's example again, `Mathematica` correctly evaluates the number of valid digits in evaluating the expression.

```
In[1]:= f[a_,b_] := 1335/4b^6+a^2(11a^2b^2-b^6-121b^4-2)+11/2b^8+a/(2b)

In[2]:= g[d_] := f[SetPrecision[77617.0,d], SetPrecision[33096.0,d]]
```

```
In[3]:= g[20]

                7
Out[3]= 0. 10

In[4]:= Precision[%]

Out[4]= 0

In[5]:= g[40]

Out[5]= -0.83

In[6]:= Precision[%]

Out[6]= 2

In[7]:= g[60]

Out[7]= -0.8273960599468213681412

In[8]:= Precision[%]

Out[8]= 22
```

2.8. Adaptable-Precision Computation

In extensible object-oriented languages such as Smalltalk or C++, it is possible to add object classes to the language which then provide functionality within the framework of the language, without the need for the user to make explicit function calls and worry about details such as memory management. Because of this, C++ has become a popular language for implementing some of the newer `Real` arithmetic classes which go beyond multi-precision floating-point. A feature of these packages is the ability to adapt the precision of the intermediate computations to produce the required accuracy in the final result.

Aberth and Schaefer. One `C++` package is the multi-precision range arithmetic package of Aberth and Schaefer [2]. They represent ranges as a multi-precision floating-point number with an explicit range width:

$$\{(\text{sign}), M_1 M_2 \cdots M_n \pm R\} \cdot B^E$$

The base B is a power of 10. Their package includes means for specifying the precision of a computation and determining the precision of a result. This makes it possible to write programs that can repeat parts of a computation if the desired precision is not obtained. They demonstrate this capability with an example program which successfully computes the value for Rump's expression.

Schwartz. Schwarz describes his `C++ Real` class from the user's point of view in [71], and his implementation in [72]. The user can declare variables of type `Real`, and then make use of "infinite precision". The package uses lazy evaluation in which the leading bits are calculated and the remaining bits are represented by an expression tree. Every time the number is re-evaluated to greater precision, the remaining expression changes. A redundant balanced binary representation is used to avoid the problem of changing the earlier bits. The `precision` class associated with this package allows the user to specify the desired accuracy in relative and (simultaneously) in absolute terms. The package (like many of the `FORTRAN` implementations) provides a full replacement for standard floating-point including rounding directions, trig functions, exponentials and logs, and the real constants π and e. Some of this is possible because Schwarz also provides a class `Series` for representing evaluable exact representations for the infinite series used for defining these values.

Programming with potential infinite objects. There are several papers on real numbers viewed as (potential) infinite objects. These papers take the viewpoint of programming methodology, or the application of lazy evaluation, or as case studies in the semantics of computing with infinite objects. The papers [10, 84] are representative of this literature. The work of Schwarz above is also in this tradition. At present these implementations seems to have been tested only on individual operations rather than in the context of some non-trivial algorithm. A very interesting paper of Vuillemin [81] (somewhat anticipated by Gosper [6]) describes the use of continued fractions for exact real arithmetic. He also points out the essential role of redundant representations for real numbers.

3. Applications of Exact Computation

We limit our scope to applications for which the concept of approximate result is meaningful. Basically, these are problems that could have been meaningfully treated under the f.p. paradigm. Most problems in the computational sciences and engineering, and also in computational geometry, would fall under this scope; traditional applications of computer algebra would not. Surprisingly, the documented instances of exact computation that falls within our scope is rather sparse and mostly of recent vintage. We can only conclude that exact computation is just as rarely used in practice.

3.1. Computational Geometry

It is well-known that computing the sign of a determinant is a fundamental "primitive operation" in geometric algorithms. Thus algorithms for convex hulls, Voronoi diagrams, Delaunay triangulation can be reduced to this operation (e.g., [44]). More generally, problems related to computing the arrangement (or subparts thereof) of a set of hyperplanes can all be reduced such primitives. We note that Clarkson [22] has shown that the sign of a determinant can be computed more efficiently than evaluating the determinant: using ideas from lattice reduction, he transforms the determinant into an "almost" orthogonal shape and then employs any standard evaluating method. The practicality of this approach is unclear at present.

We should mention a word about experimental methodology. It seems to be a standard practice to compare any exact algorithm against a "comparable" floating-point implementation. Strictly speaking, this is comparing apples and oranges since floating-point algorithms usually have no guaranteed correctness. Yet realistically, this comparison makes sense since the default implementation by practitioners uses floating-point arithmetic. The simplest notion of "comparable" is where the exact algorithm is identical with the floating-point algorithm, except that each arithmetic operation is done exactly. This is easy to achieve using operator overloading, a feature that is available in modern languages such as `C++`. Note that in exact arithmetic, we have the opportunity to use input numbers of arbitrary length but this has no counterpart in f.p. computation, so the effects of growing input number lengths cannot be compared.

Karasick, Lieber and Nackman. These authors described in [55] their experience in implementing the Guibas-Stolfi algorithm for Delaunay triangulation for a set of planar points [44]. The input points have coordinates that are rational numbers, and in the course of the algorithm they must determine the signs of $n \times n$ determinants where $n \leq 4$. Assuming the rational numbers have 3 digits in the numerator and denominator (a digit is 16-bits here), their initial implementation took 1200 seconds to triangulate 10 random points near the unit circle. A comparable double-precision run took only 0.1 seconds; this is 3 orders of magnitude faster. They proceeded to describe a range of tactics to reduce the running time until eventually, the exact method is able to triangulate a fairly large input (500 points randomly generated around the unit circle) and achieve a speed comparable to the floating-point implementation (59 seconds versus 14 seconds). Their sign of determinant algorithm ("Sgauss") replaces the original matrix entries by bounding intervals of low precision and then computing the determinant by a variant of Gaussian elimination using interval arithmetic. If the determinant is contained in an interval that does not contain 0, we already have the correct sign; otherwise, repeat the process with higher precision bounds for the matrix entries. Among their conclusions are (1) both interval arithmetic and fixed-precision arithmetic can be effective as filters to avoid expensive rational exact arithmetic; (2) memory management can be a huge overhead in number packages; (3) object-

oriented programming tools were crucial for their experimentation. We remark that although this paper is concerned with speeding-up one *specific* algorithm, many of their techniques have wider applicability.

Fortune and van Wyk. A more general approach to exact geometric computation is described by Fortune and van Wyk in [40]. They again tested Delaunay triangulation algorithms (in 2 and 3 dimensions). But in contrast to Karasick et al, they focused on exact *integer* arithmetic. They evaluated several determinant evaluation methods, and two widely available multiprecision number packages ("mp" which is available with many Unix installations and "BigNum" from INRIA). The authors pointed out the useful distinction between the comparative speeds of the primitives (i.e., determinant computation) versus the comparative speeds of the overall algorithm. In the particular algorithms tested, the primitive operations use between 20% and 50% of the overall runtime. Thus, if the primitives were re-implemented using exact arithmetic and these now run x times slower, then we may estimate the total runtime to be between $(x+4)/5$ and $(x+1)/2$ times slower. They found that x is between 40 and 140, depending on the input bit-length and choice of arithmetic.

The basic ideas of Fortune and van Wyk's approach are embodied in their system `LN`, described in more detail in [41]. `LN` is an *expression compiler* (really, a preprocessor to produce `C++` code). Roughly speaking, given a polynomial expression the compiler produces a straightline code that computes the value of the expression exactly. It turns out that the (integer) variables in an expression are typed according to its length, so that the code produced depends on this length, determined at compile time of `LN`. The idea is to (i) avoid run-time memory allocation, overhead for subroutine calls or loops, and (ii) to exploit a feature of geometric algorithms, viz., all numerical computations in the algorithm can be reduced to a small number of such expressions. The `LN` interpretation of these ideas actually has an intricate interface to `C++`, and we refer to the original papers for details. We remark that there is a tradeoff between (i) and (ii). More precisely, the efficiency demands of (i) forces that the typing of variables by length, but this has led to a proliferation of compiled expressions, in violation of (ii). In fact, this so severely limited the applicability of `LN` (see below) that this strict typing is expected to be relaxed in future versions.

There are several other interesting ideas in `LN`. (a) Each multiprecision integer is represented by a sequence $A = (a_0, \ldots, a_m)$ of double-precision floating point variables, although the a_i's really represent integers up to 53 bits long. The integer represented by A is $\sum_{i=0}^{m} a_i 2^{ri}$ where r is the *radix*. The authors optimized the value to be $r = 23$. Let us call A *normalized* if each a_i is at most r bits long. Under the IEEE floating-point standard, hardware arithmetic on the a_i's is guaranteed to give exact answers if the result is an integer at most 53 bits long. Thus, in many cases, we could add two or more such representations componentwise; a similar remark holds for multiplication. Otherwise, we must "normalize" A before the operations. (b) Floating-point filters are built into the compiled expressions. They are useful in

determining the signs of expressions; such knowledge is built into LN.

It is interesting to note that both (a) and (b) exploit the heavily optimized hardware for floating-point computation in modern workstations. It is a slight paradox (and a testimony to the domination of the floating-point standard) that floating-point multiplication is significantly faster than integer multiplication on many RISC machines.

Milenkovic and Chang. These authors [19] reported on their experience with using the LN package for performing Boolean operations on 3-dimensional polyhedra. Initially, the implementation uses 23 different types of points and 37 types of expressions. This proliferation of types is a consequence of the strong typing by size in LN: for instance, input points must be differently typed than the intersection of an edge with a face. Only about 2/3 of the LN code could be compiled before compiler resources were exceeded. Eventually, the number of types of points and expressions were reduced to 13 and 21 respectively, and this could be compiled. The authors concluded that dynamic error evaluation instead of the current static error bounds would sacrifice some runtime efficiency for a more acceptable overall performance in LN.

Benouamer, Jaillon, Michelucci and Moreau. These authors [8] constructed a C/C++ package for lazy evaluation of expressions. Basically, each numerical value ("LazyNumber") is represented by its defining expression and an interval containing the value. The current implementation uses rational expressions. The intervals are narrowed as needed, but this is done from bottom up (i.e., by narrowing the intervals at the leaves, this is propagated up in the natural way). This is in contrast to the top-down approach of [32] for increasing precision. Applied to the Bentley-Ottmann algorithm for reporting all pairwise intersections among a set of line segments [9], they reported that the machine floating-point arithmetic is 4–10 times faster than the lazy version. In turn the lazy version is (for example) 75 times faster than the use of exact arithmetic arithmetic to relative precision 10^{-9}.

Burnikel, Mehlhorn and Schirra. The authors reported in [18] their experience in implementing an incremental randomized algorithm for the Voronoi diagram of a set of line segments. Each closed line segments is decomposed into 3 objects: its two endpoints and the open line segments. Initially, all the endpoints are randomly inserted and their Voronoi diagram incrementally updated with each insertion. Then the open line segments are randomly inserted and again the diagram is updated. The fundamental test amounts to determining if a given feature of the current Voronoi diagram "conflicts" with the object: for instance, if the feature is a Voronoi vertex v and the object is an open line segment ℓ, we must compare the "clearance radius" r of v (the distance from v to one of its defining objects) with the distance d from v to ℓ. The $r : d$ comparison is called an "incircle test". The authors showed that

the distances r, d need only be computed to $48L$-bits of precision (assuming input coordinates are rational numbers with L bits in the numerator and denominator). They tested two methods of performing the incircle test: by using exact integer arithmetic with repeated squaring (this is possible by the special nature of the $r : d$ test) and by using bigFloats to evaluate the expressions r, d. Their use of bigFloats amounts to evaluating the expressions to larger precision until a decision is definite. They tested three classes of input data ("easy", "realistic" and "difficulty"), with $L = 10, 20, 30$. Their conclusions are that (1) the actual precision needed is much less than the $48L$ upper bound, and (2) using exact integer seems to be slightly faster than using bigFloats.

Dubé and Yap. In [32], we explored two ideas for exact computation: the use of bigFloats and an expression evaluator. While bigFloats have been around almost as long as bigIntegers (see section 2), their usefulness for exact computation seems novel. The theoretical basis for their use in exact computation comes from the existence of root separation bounds (see [60, 86] and next section). A `bigFloat` package based on Gnu's `Integer` was implemented in `C++`. Each bigFloat here has a dynamically maintained error bound; whenever the error bound exceeds one digit, significance of the bigFloat is reduced to improve efficiency. We converted the `C` implementation[5] of Fortune's Voronoi diagram algorithm [39] into a `C++` program, and overloaded the arithmetic operators with bigFloat operators. The basic comparisons in the algorithm involve numbers of the form

$$\frac{a + \sqrt{b}}{d}$$

where a, b, d are essentially $3L, 6L, 2L$ bit numbers (inputs are L-bit integers). We showed that comparing two such numbers by repeated squaring needs $20L$ bit integer operations and involves a fairly large Boolean predicate, while computing the numbers approximately to $25L + O(1)$ bits (using bigFloats) suffices to make a determination. An advantage that accrues with computing the numbers to $25L + O(1)$ bits is that subsequent comparisons involving the same numbers do not require arithmetic operations.

We also implemented an expression package `bigExpression` that is based on `bigFloat`. The principles behind our design will be described in the next section. We performed two sets of tests: (1) Using randomly generated point sets of size from 50 to 400, our bigFloat version of Fortune's algorithm is about 10 times slower than the f.p. version. Our bigExpression version is about 70 times slower. (2) In computing the sign of 4×4 determinants, the relative speeds depended on the input bit-sizes. With input numbers of about 200 bits, `bigFloat` was about 300 times slower than f.p., but 8 times faster than `bigInteger`. The *bigExpression* version is about 12 times

[5]Fortune's code is available from netlib@research.att.com using the command: send sweep2 from voronoi.

slower than *bigFloat*. We have not optimized the packages, and profiling shows that memory management is a significant part of the running times.

3.2. Miscellaneous

Not much has been written about exact computation in other domains. Moreover, when exact computation is considered, authors often claim that it is infeasible. These conclusions are usually prompted by the worse-case bounds necessary to compare two numbers exactly. But there are ways to circumvent this (see the next section). We prefer to conclude from this that it is unreasonable to do exact computation using general-purpose algebraic computing systems (such as MATHEMATICA, MAPLE); rather we must exploit special properties in each domains. We briefly look at some domains that are natural targets for exact computation.

Geometric editing and modeling. Exact computation is relevant in geometric editing and modeling since we normally want a design or model to have certain verifiable properties ([48] is a general reference on geometric modeling). The properties of interest are usually topological ones (e.g., [63]). Traditionally, a geometric modeling/editing system is seen as a *drafting tool* for building geometric scenes by explicit, deterministic commands. A more powerful view is to see it as a *constraint solver*, where geometric scenes are defined by constraints which must be solved. As a drafting tool, we face two basic classes of algorithmic problems: Boolean operations on geometric elements and "classification problems" (predicates about containment or intersection of geometric elements). The thesis of Yu [87] studies the precision necessary for the classification problems in polyhedral modeling, but is not optimistic about exact algorithms. There is a large literature about constraint solving geometric editing/modeling systems ([77, 13] are two early papers). Many of them are heuristic and partial in their solution capabilities. Recent attempts at such systems aim at more complete constraint solving abilities, by exploiting the appropriate algebraic tools such as Gröbner bases, Wu-Ritt theory or elimination theory. For instance, Ericson and Yap [36] described `linetool`, a planar geometric editing system based on points, line segments and circular arcs. Their system was intended to be complete in its constraint solving ability, but the computational bottlenecks severely limited its applicability. More recently, a system proposed by [14] has many of the same features as `linetool`. Their algorithmic approach is based on graph-reduction techniques and have similarities to ideas of John Owen. This system seems more practical since it is aimed at less than the full range of polynomial constraints.

Constraint Logic Programming Languages. Logic programming languages [30] takes an extremely interesting twist when we endow them with knowledge of some mathematical domain (typically real algebra). These *Constraint Logic Programming* (CLP) languages were described in [53]. Clearly, this is closely related to the geomet-

ric editing/modeling applications above. The linguistic and logical properties of CLP languages (e.g., [46, 58]) and their (potential) applications are well-treated in the literature (e.g., [47]). Of special relevance to us is the fact that any implementation of such languages must understand much about the underlying mathematical domain, especially its computational properties; see some treatment of this in [29, 28, 74, 57, 33]. In practice, it seems that implementations of CLP languages use f.p. computation although, evidently, any *correct* CLP compiler must use exact computation! Ultimately, the CLP compiler must try to balance the opposing demands of correctness and efficiency: some mixed-mode computational ability (f.p. and exact) is probably appropriate and this constitutes a wide-open research area.

Computational metrology. There is considerable recent ferment in the dimensional tolerancing and metrology (see [62, 79] and articles in that volume). In particular, the unreliable computational results from coordinate measuring machines (CMMs) [82] precipitated a minor crisis. CMMs proliferated in the 80s and represent the state of the art in dimensional metrology [76]. In response to the crisis, both the EC community [38, 3] and the US [37] have propose to construct a "reference software" to test vendor software. The systematic study of these issues has spawned the subject of computational (dimensional) metrology [50]. There are many sources of errors in the output of CMMs, but from our viewpoint, it is evident that computational errors arising from the use of (1) inexact algorithms (such as least-squares fit algorithms) and (2) floating-point computation is avoidable. The replacement of inexact algorithms by algorithms developed in computational geometry is not new (e.g. [68]); but the deployment of exact methods brings a new level of sophistication to the reference software. Since the basic computational problems in this area [3] have "bounded depth" [85], there is also hope that exact solutions may even be used directly in vendor software.

4. Theory of Exact Computation

A large class of problems for which exact computation is theoretically possible are the *semi-algebraic problems*. This class will provide a suitable backdrop for most problems of interest in exact computation. We define this class and state the fundamental result here. Some basic knowledge of complexity theory and first-order logic is assumed in this description.

Algebraic numbers. Ultimately, the claim that we can solve many geometric problems exactly is based on the fact that we can compute with algebraic numbers exactly. An *algebraic number* is any complex root of a univariate polynomial with integer coefficients. Thus $\sqrt{2}$ is algebraic since it is a root of $x^2 - 2$. Unless otherwise stated, we implicitly assume the algebraic numbers here are real. Although algebraic numbers generally do not have finite representation in positional notations, they have

finite description. For instance, if α is a real root of the integer polynomial $A(x)$, we can represent α by the pair $(A(x), I)$ where I is a interval with rational endpoints containing α but no other real roots of $A(x)$. This is called the *isolating interval representation* of algebraic numbers. For instance, $\sqrt{2}$ can be represented as the pair $(x^2 - 2, [1, 2])$. Note that $A(x)$ and I are both non-unique. We can make $A(x)$ unique by insisting it to be the *minimal polynomial* of α (i.e., smallest degree primitive polynomial with positive leading coefficient), but that is too expensive to enforce. We can effective perform the 4 arithmetic operations on such representations, compare two such numbers, and so on. See [17, 86].

A *semi-algebraic predicate* $\Phi(x_1, \ldots, x_n)$ is a Boolean combination of "atomic predicates" of the form:

$$P(x_1, \ldots, x_n)\, \rho\, 0, \qquad \rho \in \{=, >, <, \geq, \leq\},$$

where $P(x_1, \ldots, x_n)$ is an rational polynomial in real variables $x_1, \ldots, x_n$. Such a predicate $\Phi(x_1, \ldots, x_n)$ defines a *semi-algebraic set* given by

$$\{(a_1, \ldots, a_n) \in \mathbb{E}^n : \Phi(a_1, \ldots, a_n) \text{ holds}\}.$$

For instance,

$$\Phi_0(x, y) : (x^2 + y^2 - 1 < 0) \vee (x + y - 5 = 0)$$

is a semi-algebraic predicate in variables x, y and it defines a semi-algebraic that is the union of an open disc and a straight line. A *Tarski formula* is a first-order formula based on semi-algebraic predicates. With Φ_0 as defined above, it is not hard to see that

$$(\forall x)(\exists y)\Phi_0(x, y)$$

is a true Tarski sentence (a *sentence* is a formula without free variables). A fundamental result of Tarski is that sets definable by Tarski formulas are semi-algebraic (the converse is trivial).

Informally, a computational problem Π is semi-algebraic if it can be "reduced" to Tarski formulas. Let us define a *computational problem* Π as a map

$$\Pi : X \mapsto \Pi(X),$$

where X ranges over all input instances and $\Pi(X) \subseteq \mathbb{E}^n$ (here $n = n(X)$ depends on the input instance). We assume that input instances X are suitably encoded for Turing machine computation. An *algorithm* for Π is a Turing machine that on input X produces an element of $\Pi(X)$ (again, suitably encoded). Finally, we say a problem Π is *semi-algebraic* if there is a Turing machine M that, on input X, computes a Tarski formula $M(X)$ in polynomial-time such that the semi-algebraic set $S_M(X)$ defined by $M(X)$ is contained in $\Pi(X)$. Moreover, $S_M(X)$ is empty iff $\Pi(X)$ is empty. We call M a *reduction machine* for Π in this case.

Example: Minimum width cylindrical shell problem. This is a typical problem of dimensional metrology. Consider the problem of computing a minimum width *cylindrical shell* C that contains a given finite set $X \subseteq \mathbb{E}^3$ of points. The shell C is the region that lies between two concentric cylinders of radii r and R (with $0 \leq r \leq R$) and its width $R - r$ is minimum. Containment of X by C means this: if L is the axis of the cylindrical shell C, then $x \in X$ implies $r \leq d(x, L) \leq R$, where $d(x, L)$ is the distance between x and a closest point on L. We assume that points in X have rational coordinates. Although it is not immediately obvious how to solve this problem, let us show that this is a semi-algebraic problem. We represent a shell C by the parameters (L, r, R), where L is in turn represented (non-uniquely) by two distinct points $p, q \in L$, Then a minimum width shell is precisely captured by the conjunction of the following two formulas:

1. $(\forall x \in X)[r \leq d(x, L) \leq R]$
2. $(\forall L', r', R')(\exists x \in X)[\neg(R' - r' < R - r) \vee \neg(r' \leq d(x, L') \leq R')]$.

It is easy to turn this conjunction into a Tarski formula, $\Phi_X(L, r, R)$, using the fact that X is a finite set of rational points. It is also clear that we can construct a Turing machine M that, on input X, computes $M(X) := \Phi_X(L, r, R)$ in polynomial-time. Since $\Phi_X(L, r, R)$ defines precisely the set $\Pi(X)$ of minimum width cylindrical shells containing X, we conclude that the problem is semi-algebraic. ■

A fundamental result of Collins [26, 17] says that, given a Tarski-formula Φ, we can decide if it defines a non-empty semi-algebraic set, and in case of non-emptiness, to compute a sample point in the semi-algebraic set. Moreover, all this can be accomplished in double-exponential time by a Turing machine. Using this result, we can solve any semi-algebraic problem Π as follows: on input X, compute the Tarski formula $M(X)$ using a reduction machine M for Π. Then invoke Collins' algorithm to compute a sample point Y in the set defined by $M(X)$. This proves:

Theorem 1 *A semi-algebraic problem can be solved in double-exponential time on a Turing-machine.*

Discussion. This result is fundamental in exact computation because, as noted in [85], practically all computational problems in contemporary Computational Geometry are semi-algebraic. There has been considerable theoretical advances since the original work of Tarski and Collins. For this literature, see for instance [45] and the references contained therein. Collins' original result retains much usefulness for us despite these new results. There are several reasons: one is that it is relatively simple and it has been implemented [17, p. 79]. In some sense, algorithms of exponential complexity are mainly of theoretical interest. However, such algorithms may have practical use if the exponential complexity (being a worst case bound) is not attained by every input instance: we may be lucky enough to be interested in some instances that can be computed quickly. Collin's algorithm turns out to

have this useful property, while some of the newer algorithms (being based on the BKR algorithm [7]) do not. Buchberger's algorithm [17] is another exponential algorithm of this nature. In terms of complexity, the newer algorithms derive their improvements by replacing double-exponential time by single-exponential space (or, equivalently, single-exponential *parallel* time). But at the present state of knowledge, this must be classified as a "theoretical" improvement since it is not known if single-exponential space is properly contained in double-exponential time. Nevertheless, these new bounds can exploit implicit parameters (like the dimension of the solution space), in which case the improvements have practical significance. Finally, Collins represents the sample points using isolating intervals. Algorithms based on the BKR algorithm often do not compute sample points; but when they do (e.g., [56]), the representation of the real algebraic numbers are such that the *order properties* of real numbers are mostly implicit in the representation. Another popular encoding of real algebraic numbers in which order properties are implicit is the so-called *Thom encoding* [27, 86]. In practice, such representations are inconvenient when we have to compare two real algebraic numbers.

5. Some Techniques in Exact Computation

We describe some basic techniques that are useful in exact computation. Some of these have been mentioned in the applications section. Also, many of these details are elaborations on [32].

5.1. Expression Packages

Geometric algorithms do not call basic arithmetic operations on individual numbers in a "random" fashion; rather one can usually group the individual operations into evaluations of a small number of expressions. For instance, we had already noted that many subproblems of hyperplane arrangement [34] can be reduced to evaluating one determinantal expression. The opportunity for optimization is much greater at the level of expressions than at the level of bigNumber packages. In this sense, the most important package for exact geometric computation is an expression package that supports "exact computation".

Such packages are based on some underlying class of "numerical expressions". A *numerical expression* E is recursively built from the (numerical) variables and constants using a fixed set of (numerical) operator symbols, in the usual way. The most important class of expressions are the *polynomials*, defined over the operator symbols are $+, -, *$. In [32], we extended the polynomials to include the $\div$ and $\sqrt{\cdot}$ operations – let us refer to these as *radical rational expressions*. This family captures most of the expressions in practice. Structurally, we can view E as a labeled directed acyclic graph (dag) with one source node (called the *root*) such that each sink node is labeled by a numerical variable (called the *parameter*), and each internal node (non-sink) is labeled by an operator. The successors of an internal node u

are ordered so that there is a bijection between the successors and the arguments of the operator labeling u. E.g., if u is labeled by a binary operator, it must have two successors. When the parameters in E are instantiated with explicit numerical values, we can *evaluate* E in the obvious "bottom-up" fashion, by propagating values to all the internal nodes. In particular, the value of the root is called the *value of the expression*. Depending on the application and assumptions, any software realization of "expression" will accrue additional constructs and properties over and above the underlying numerical expressions. Let us explore such assumptions.

(I) The *fundamental premise* of expression packages is that each constructed expression will be repeatedly evaluated in the course of the algorithm, each time for different values of its parameters. Therefore it may pay off to preprocess such expressions to improve their evaluation process. Fortune and Vanwyk [40] explored the compilation of expressions prior to their evaluation – their approach introduces a strong typing of expressions by the "bit-sizes" of parameters. Yap [85] suggests the restructuring and identification of common subexpressions, and more generally, compiler optimization techniques. For instance, taking advantage of associativity of "+", we may restructure a subtree of binary additions into a "summation" operator that which takes any number of arguments. Furthermore, if the signs of summands can be estimated at compile-time, we should divide the summands into two groups, comprising respectively the positive and the negative arguments, so that they are added among themselves first. (This avoids unnecessary precision in our computation, see next.)

(II) The *precision premises* say that the precision of intermediate values can often be limited without compromising the goal of exact computation. Perhaps the simplest precision premise is this: *that it suffices to know the sign of the expression.* This premise is actually quite common. The papers [55, 40] exploited this premise by introducing *floating-point filters* that try to get a quick decision on the sign of determinants. In [32], we attach a *precision-bound* and an *approximate value* to each node of an expression. We guarantee that the approximate value satisfies the precision-bound. This means that changing the precision-bounds may force us to recompute approximate values at each node. Users can use this mechanism to exploit any known precision premises in their application.

5.2. Error-bounds versus precision-bounds

In some sense, these two terms reflect the difference between a pessimist and an optimist, whether a cup is half-empty or half-full. But we can distinguish these two concepts in the context of expressions: once we have computed approximate values for the parameters of an expression, the approximate values of the internal nodes can be automatically evaluated from bottom-up. Any *error-bounds* in the parameters propagate upwards in a completely deterministic manner. For instance, if the express is $c = a+b$ and both a and b have an absolute error of at most ± 0.1 then we can place an error-bound of ± 0.2 on c. On the other hand, a *precision-bound* is a user-specified

quantity, usually imposed only at the root of an expression. E.g., the user can specify that the expression $c = a + b$ must be computed to within a precision-bound of ± 0.2 in absolute terms. The system then propagates this precision *top-down* all the way to the parameters. Note that this propagation is not deterministic: we could specify that a, b must each be computed to within absolute precision ± 0.1, but clearly there are many other choices. For instance, if we know that $|a| < 0.01$, then we can ignore a and simply require b to be computed to absolute precision ± 0.19, and output the approximated value of b as the approximation to c. We have used only absolute bounds in this illustration, but in general, these concepts extend to relative bounds as well.

5.3. Error bounds in bigFloat

In our design [32], we use bigFloats as approximate values of nodes in expressions. Note that bigFloats are useful even for approximating rational numbers. Thus, it may be sufficient to approximate the rational value

$$1212/343434 \tag{1}$$

by the bigFloat value 0.35×10^{-2} (as usual, our examples use base 10). Since we expect our bigFloats to be used as approximations, we decided that it would be useful to build into our bigFloat numbers the notion of an error bound. This error bound is automatically propagated in arithmetic operations; it is said to be a *dynamic error bound* since it is maintained at run-time.

Back to example (1), we would actually use the *bigFloat number with error-bound* (or, *bigFloat range*)

$$\widetilde{b} = (0.35 \pm 0.01) \times 10^{-2}$$

as the approximate value. Explicitly, this bigFloat range would be represented as the triple $\widetilde{b} \cong (35, -2, 1)$. In general, the triple (f, e, d) of integers represents the range

$$\langle f \pm d \rangle \times B^e$$

where B is the base of the radix system and $\langle n \rangle$ applied to any integer n means that we place the radix point just before the base B representation of n. We call the bigFloat number (f, e, d) *exact* when $d = 0$. To illustrate the issue of propagating error bounds, suppose we square the range $\widetilde{b}$. How should we represent the result, $(\widetilde{b})^2$? One solution is to square the number 0.35×10^{-2} yielding 0.1225×10^{-4} and recompute the error bound. Thus

$$(\widetilde{b})^2 = (0.1225 \pm 0.0071) \times 10^{-4}$$

or $(\widetilde{b})^2 \cong (1225, -4, 71)$. A slightly more accurate procedure is to square the upper and lower bounds of the bigFloat range, and take their midpoint as the representative

un-normalized	value range	normalized
(1000,0,10)	0.0990-0.1010	(100,0,1)
(1000,0,11)	0.0989-0.1011	(100,0,2)
(1001,0,10)	0.0991-0.1011	(100,0,2)
(1004,0,19)	0.0985-0.1023	(100,0,3)

Figure 1: Normalising values to one error digit

value, with appropriate error bound. [This procedure would yields $(\widetilde{b})^2 = (0.1226 \pm 0.0070) \times 10^{-4}$.] This propagation of error bounds is seen to be cumbersome: as d grows arbitrarily large in (f, e, d), more and more lower order digits in f would be unreliable. It seems a waste of time and space to compute and store f to this many *in*significant digits. We decided (like [2]) to impose the following inequality:

$$0 \leq d < B \tag{2}$$

Since $B = 10$ in our example, this means that $0 \leq d < 10$. Back to our example, we must therefore truncate the triple $(1225, -4, 71)$ to $(123, -4, 8)$, representing the range $(0.123 \pm 0.008) \times 10^{-4}$.

When a bigFloat number is represented as above with the radix before the first non-zero digit, and a single digit of error, then we say that it is in *normalized* form. During the course of a computation, we may produce numbers which are not normalized. (To improve accuracy, we can delay normalization until the end of an expression evaluation.) Returning these values to normalized form is quite easy. The radix can be shifted to the proper location by altering the exponent, and the error can be safely truncated to a single digit as illustrated in figure 1.

Clearly this is a compromise between the accuracy of our bounds and avoidance of unnecessary precision. Simple experiments show that keeping this limited range information (2) is already quite effective.

5.4. Composite precision bounds.

It seems that a flexible general-purpose expression package ought to support both relative as well as absolute precision bounds. Let a, r be real numbers. Let us say that a number $\widehat{x}$ approximates another number x to *absolute precision* a if $|\widehat{x} - x| \leq 2^{-a}$; and the approximation is to *relative precision* r if $|\widehat{x} - x| \leq |x|2^{-r}$. Combining the two concepts, let $[a, r]$ denote a *(composite) precision bound*. But what shall an approximation to *precision* $[a, r]$ mean? Schwarz [71] takes this to mean the conjunction of the absolute precision a and relative precision r. We prefer to take the disjunction of the two separate bounds. We say the number $\widehat{x}$ is *an approximation of* x *to precision* $[a, r]$ if

$$|\widehat{x} - x| \leq \max\{2^{-a}, 2^{-r}|x|\}, \tag{3}$$

written,

$$\widehat{x} \cong x[a, r]. \tag{4}$$

The triple $x[a,r]$ occurs so often in this combination that it is convenient to write X for $x[a,r]$ and call it an *approximate number*. Thus equation (4) becomes $\widehat{x} \cong X$. Note that we can have absolute (resp., relative) bounds by choosing $r = \infty$ (resp., $a = \infty$).

Motivation: Our original idea is that $\widehat{x} \cong x[a,r]$ should mean that

$$\widehat{x} = x(1 + \epsilon \cdot 2^{-r}) + \delta \cdot 2^{-a}$$

for some $0 \leq |\epsilon|, |\delta| < 1$. But this seems difficult to handle directly. So we simplify it to mean

$$\widehat{x} = x + \overline{\max}\{x \cdot \epsilon \cdot 2^{-r}, \delta \cdot 2^{-a}\},$$

that is, by replacing the sum of two error terms by the "maximum" of the error terms. The "maximum" here (denoted $\overline{\max}$) of two real values a, b is just the value whose absolute value is largest. E.g., $\overline{\max}\{-3, 2\} = -3$. Clearly replacing a sum by such a maximum at most increases the possible error by a factor of 2, so it is an acceptable replacement. But notice that the modified concept amounts to equation (3).

5.5. Algorithms for Maintaining bigFloat Ranges

The maintenance of valid intervals using bigFloat range arithmetic is illustrated in this section using the examples of addition, multiplication and the square root operation.

The system will usually attempt to provide the most accurate result possible in performing the operation in the given inputs. There may be some situations in which this behavior is not desirable. Consider adding the integer 1 to the bigFloat value $(100, -10000, 1)$. It is possible to evaluate this sum with 10,000 digits of accuracy, but this may be far more than is actually needed. To aid in these situations, a *global composite precision bound* is specified in our bigFloat package. (The user would normally accept the default bound provided by the package but can modify this default if desired.) When performing an operation $\widehat{c} = \widehat{a} \oplus \widehat{b}$, the system produces (roughly speaking) the most accurate range $\widehat{c}$ possible *without exceeding the global precision bound*. Notice that this use of precision bounds is different than their use in the expression package.

Addition The algorithm for addition is simple, but points out some of the issues involved. A *safe* algorithm would be to:

1. align the least significant digits by shifting
2. add the digits and the error values
3. normalize.

For example, to add (123,0,1) and $(246, -2, 3)$, we may first align the least significant digit to produce

$$(12300, 0, 100) + (246, -2, 3) \;=\; (12546, 0, 103).$$

The normalized answer is then (125, 0, 2).

While this is a *safe* algorithm, it may involve computing far more digits then will appear in the normalized form. Instead, we wish to first estimate the magnitude of the least significant digit of the sum. To do this we must consider the global precision bounds and the magnitude of the least significant digits of the summands.

To determine if the relative precision bound allows a coarser approximation of the sum, we must locate the leading digit of the sum. If the summands have similar signs, this is easy; otherwise, this involves comparing the numbers from the leading digits down toward the less significant digits. Of course, we may terminate this comparison as soon as we determine that meeting the relative precision exceeds the global precision bound. Once the magnitude of the least significant digit of the sum is determined, the digits below this may be truncated (being careful to keep the ranges safe), and then the values can be added. In the above example, we could truncate the representation of the second summand to $(2, -2, 1)$ and add

$$(123, 0, 1) + (2, -2, 1) \;=\; (125, 0, 2).$$

Multiplication A *safe* algorithm for multiplication is to form the product of (f_1, e_1, d_1) and (f_2, e_2, d_2) as $(f_1 f_2, e_1 + e_2, f_1 d_2 + f_2 d_1 + e_1 e_2)$, assuming that f_1, f_2 are positive. If $\langle f_1 \rangle \cdot \langle f_2 \rangle < 0.1$, then the exponent must be adjusted to the proper value.

In most situations, multiplying two n digit numbers results in a $2n$ digit product. But, if both of the bigFloat numbers are not exact, then we expect at most n valid digits in the product. To avoid the computation of digits which do not appear in the normalized product, we must once again estimate the magnitude of the least significant digit of the normalized product. To do this we must consider the composite precision bounds for the operation and the magnitude of the terms in the error value $f_1 d_2 + f_2 d_1 + e_1 e_2$. Once the magnitude of the least significant digit of the product is determined, then we can carry out only the needed portion of the multiplication.

Square Root To compute the square root β of a number α, we use the Newton iteration $\beta_{i+1} = \frac{1}{2}(\beta_i + \frac{\alpha}{\beta_i})$. The sequence of values $\beta_1, \beta_2, \ldots$ converges toward the desired root. At the start of an iteration, we regard β_i as an exact value (i.e. the error $d = 0$). This is because we can regard β_i as a true element of the converging sequence. If α were exact, and we could perform all of the operation exactly, then we would produce a next approximation β_{i+1} such that the root β lies between β_i and β_{i+1}. Instead, due to inaccuracies we can only find an approximation $\widehat{\beta_{i+1}}$, But, we are still assured that the root β lies between β_i and a value in the range described by $\widehat{\beta_{i+1}}$. We may continue the iteration until the accuracies of the β_i fail to increase, or until the default composite precision bounds are met.

6. Precision-Driven Computation

The papers [32] and [8] may appear to be similar in their use of run-time techniques. But on closer examination, there is a crucial difference. In [8], the authors repeatedly increase the precision of values in the leaves of an expression until it reaches an acceptable precision at the root. In contrast to their "lazy approach", we like to view our approach in a more active way. The user specifies a precision-bound at the root; we propagate this bound to all the other nodes in a top-down fashion. At the leaves, we assume that there are primitives to compute approximations to the any numerical constant to any specified precision. We then propagate these values back up the tree. We call this approach the *precision-driven approach*.

6.1. Algorithms for precision-driven arithmetic

We need the concept of the *most significant bit* (MSB) of a real number. For each real number x, we would like determine the value

$$\mu_x := \lfloor \lg |x| \rfloor .$$

Note that $\lg(\cdot)$ is logarithm to base 2. Informally, μ_x tells us the position of the MSB in the binary representation of x, assuming that the binary places are indexed by the integers as follows

$$\cdots, 3, 2, 1, 0, \bullet, -1, -2, -3, \cdots$$

In particular, the zero-th bit is just to the left of the binary point (indicated by $\bullet$). Thus $|x| \geq 1$ iff $\mu_x \geq 0$. In practice, it is difficult to determine μ_x exactly by looking at a finite number of bits of x. This is because of the phenomenon of carry propagation. So we introduce the "approximate version" of μ_x. For any approximate number $X = x[a, r]$, define

$$\mu_{x[a,r]} = \mu_X$$

to be equal to the *least integer* e that satisfies

$$2^e \leq |x| < 2^{1+e} + \max\{2^{e-r}, 2^{-a}\}. \tag{5}$$

For instance, let $x = (1000.001)_2$ (in binary notation). Depending on the precision $[a, r]$, we can see that $\mu_{x[a,r]}$ is either 2 or 3. If $a = 2, r = 5$ then we have $\mu_{x[2,5]} = 2$ since $(100.00)_2 \leq x < (1000.01)_2$. Notice that without the stipulation that the e must be the "least integer" in our definition, we could have defined $\mu_{x[2,5]}$ to be 3 since $e = 3$ would satisfy equation (5) as well. The reader can also check that $\mu_{x[3,6]} = 3$. In general, it is not hard to see that for all $[a, r]$,

$$\mu_{x[a,r]} \in \{\mu_x, \mu_x - 1\}.$$

Hence, an integer of the form μ_x and $\mu_x - 1$ will be called an *approximate MSB* of x. Usually, we write "M_x" for an approximate MSB of x.

In the following, it will be convenient if μ_X is known for each approximate number $X = x[a, r]$. This means that we compute at least the most significant bit of an approximation $\widehat{x}$ to X. This may be undesirable for 2 reasons:
(a) This requirement on $\widehat{x}$ may sometimes exceed the requirements of the given composite precision bound $[a, r]$.
(b) This may force us to determine the sign of the exact value, which may be very expensive. Hence, instead of obtaining μ_X, we may sometimes only have a lower bound on μ_X (which is in turn a lower bound on μ_x). We shall see that this is sufficient, provided we henceforth assume that the relative precision r in a precision-bound $[a, r]$ is always non-negative:

$$r \geq 0.$$

Addition Algorithm. Suppose we are given expressions x, y, z where $z = x + y$, and we are given $[a_z, r_z]$. The goal is to compute $\widehat{z} \cong z[a_z, r_z]$. We divide this task into two steps: first we want to compute $[a_x, r_x]$ and $[a_y, r_y]$ such that for any $\widehat{x}$ and $\widehat{y}$, if

$$\left.\begin{array}{rcl} \widehat{x} & \cong & X := x[a_x, r_x], \\ \widehat{y} & \cong & Y := y[a_y, r_y], \end{array}\right\} \tag{6}$$

then

$$\widehat{x} + \widehat{y} \cong Z := z[a_z, r_z]. \tag{7}$$

The second step is to compute such an approximation $\widehat{x}, \widehat{y}$ and choosing $\widehat{z} = \widehat{x} + \widehat{y}$. Since this second step is simply a bigFloat addition, we only focus on the first step. Consider the following two inequalities:

$$\begin{aligned} \max\{\mu_z - r_z, -a_z\} - 1 &\geq \max\{\mu_x - r_x, -a_x\}, \\ \max\{\mu_z - r_z, -a_z\} - 1 &\geq \max\{\mu_y - r_y, -a_y\}. \end{aligned}$$

It is not hard to verify that these two inequalities are sufficient to ensure equation (7). In general, the value of μ_z is unavailable and so we may substitute a lower bound μ_z^* on μ_z and use the inequalities

$$\max\{\mu_z^* - r_z, -a_z\} - 1 \geq \max\{\mu_x - r_x, -a_x\}, \tag{8}$$
$$\max\{\mu_z^* - r_z, -a_z\} - 1 \geq \max\{\mu_y - r_y, -a_y\}. \tag{9}$$

By symmetry, it is enough to see how to achieve one of the two inequalities, say, equation (8). Once we know μ_z^*, we can choose a_x, r_x as small as possible so as to make equation (8) into an equality:

$$r_x = \mu_x - \max\{\mu_z^* - r_z, -a_z\} + 1, \qquad a_x = -\max\{\mu_z^* - r_z, -a_z\} + 1. \tag{10}$$

(Note: actually we must set $r_x := \max\{0, \mu_x - \max\{\mu_z^* - r_z, -a_z\} + 1\}$ since we assume all relative precisions are non-negative.)

We now turn to the problem of computing μ_z^*. While computing μ_z it is useful to determine the sign of z simultaneously. Suppose that recursively, we can compute M_x, M_y where M_x and M_y are approximate MSB's of x and y, respectively. Moreover, assume we also computed the signs of x and y. There are two cases for deriving μ_z. (CASE A) Suppose x, y have the same sign. This is the easy case. The MSB of Z can be only one greater than the MSB's of X and Y, so

$$\mu_z = \max\{M_x, M_y\} + \delta$$

where $\delta = 0, 1, 2$ or 3.

We could perform the addition using coarse approximations of x and y to determine δ more precisely, but usually it suffices to use the lower bound

$$\mu_z^* := \max\{M_x, M_y\}.$$

(CASE B) Suppose x, y have opposite signs. We give an iterative procedure to find a lower bound on μ_z. We will iteratively assign

$$\mu_z^* \leftarrow \max\{M_x, M_y\} - h,$$

for various values of h. Initially, $h = 0$, and then $h = 1$. If h is "too small", we double h for the next iteration. What does it mean to be "too small"? Well, it means that when we use the guessed value of μ_z to compute $[a_x, r_x]$, etc, as before, the resulting value of $\widehat{x} + \widehat{y}$ is 0. Otherwise, we have a fairly good approximation for μ_z. One more round of computation can yield a suitable lower bound μ_z^*.

Multiplication Algorithm Suppose that we are given the expression $z = x \cdot y$ and precision bound $[a_z, r_z]$, and we want to compute $\widehat{z} \cong z[a_z, r_z]$.

We need to determine the precision bounds $[a_x, r_x]$ and $[a_y, r_y]$ for evaluating x and y. Once we have these precision bounds, we can compute $\widehat{x}$ and $\widehat{y}$ where

$$\epsilon_x = |x - \widehat{x}|, \qquad \epsilon_y = |y - \widehat{y}|$$

and

$$\begin{aligned} \epsilon_x &\leq \max\{|x|2^{-r_x}, 2^{-a_x}\}, \\ \epsilon_y &\leq \max\{|y|2^{-r_y}, 2^{-a_y}\}. \end{aligned}$$

Multiplying the values $\widehat{x}$ and $\widehat{y}$ yields

$$\begin{aligned} \widehat{x} \cdot \widehat{y} &= (x - \epsilon_x)(y - \epsilon_y) \\ &= xy - (x\epsilon_y + y\epsilon_x - \epsilon_x\epsilon y). \end{aligned}$$

This will therefore be a valid approximation of z if

$$|x\epsilon_y + y\epsilon_x + \epsilon_x\epsilon y| \leq \max\{|z|2^{-r_z}, 2^{-a_z}\}.$$

This inequality is satisfied if we have:

$$\begin{aligned} |x\epsilon_y| &\leq \frac{1}{3}\max\{|z|2^{-r_z}, 2^{-a_z}\}, \\ |y\epsilon_x| &\leq \frac{1}{3}\max\{|z|2^{-r_z}, 2^{-a_z}\}, \\ |\epsilon_x\epsilon_y| &\leq \frac{1}{3}\max\{|z|2^{-r_z}, 2^{-a_z}\}. \end{aligned}$$

This in turn can be replaced by

$$\begin{aligned} 1 + \mu_x + \log_2(\epsilon_y) &\leq \max\{\mu_z - r_z, -a_z\} - 2, \\ 1 + \mu_y + \log_2(\epsilon_x) &\leq \max\{\mu_z - r_z, -a_z\} - 2, \\ \log_2(\epsilon_x) + \log_2(\epsilon_y) &\leq \max\{\mu_z - r_z, -a_z\} - 2. \end{aligned}$$

The last inequality in this set is redundant because of our assumption that $r_x \geq 0$ (since this implies $\epsilon_x \leq |x|$). The ϵ's will be sufficiently small to satisfy these inequalities if we choose

$$\begin{aligned} a_x &= \mu_y - \max\{\mu_z - r_z, -a_z\} + 3, \\ a_y &= \mu_x - \max\{\mu_z - r_z, -a_z\} + 3, \\ r_x = r_y &= \mu_x + \mu_y - \max\{\mu_z - r_z, -a_z\} + 3. \end{aligned}$$

With multiplication, we do not have any difficulties estimating μ_z since $\mu_z = \mu_x + \mu_y + \delta$ where δ is either 0 or 1. Therefore, we have the following algorithm.

1. perform coarse evaluations of x and y to determine μ_x and μ_y.
2. use these values to determine a_x, a_y, r_x and r_y.
3. compute $\widehat{x} = x[a_x, r_x]$ and $\widehat{y} = y[a_y, r_y]$.
4. use bigFloat multiplication to compute $z[a_z, r_z] = \widehat{x} \cdot \widehat{y}$.

Remarks: We can apply this "precision-driven paradigm" much more widely. For instance, take any optimization problem and convert it to an "approximation" algorithm just by specifying an addition parameter $1 > \varepsilon \geq 0$. This means that the answer is guaranteed to give an answer that is (say we are minimizing) at most $V^*(1 + \varepsilon)$; here V^* is the actual minimum value attained by the problem instance. A classic example is the problem of Euclidean shortest paths [64, 21]. We consider this approach as falling within the exact computation paradigm: thus an "exact approximate algorithm" is not an oxymoron.

6.2. Incremental Computation

Another important technique arising from the use of variable precision-bounds is the concept of "incremental computation". If we have computed the sum $a + b$ to some precision, and we want to increase the precision, how much additional work is required? We illustrate the ideas by explaining only the incremental addition algorithm.

We may now assume the situation of equation (6), and moreover, some lower bounds on μ_x, μ_y, μ_z are known. Now let α_z, ρ_z be given, meaning that we want to compute the incremental improvement,

$$\widehat{\widehat{z}} = z[a_z + \alpha_z, r_z + \rho_z]$$

for some $\alpha_z, \rho_z \geq 0$. We can determine the necessary values for $\alpha_x, \rho_x, \alpha_y, \rho_y$ as above. So recursively, we assume that $\widehat{\widehat{x}}, \widehat{\widehat{y}}$ are been computed:

$$\widehat{\widehat{x}} = x[a_x + \alpha_x, r_x + \rho_x], \qquad \widehat{\widehat{y}} = y[a_y + \alpha_y, r_y + \rho_y].$$

Now let $\delta_x = \widehat{\widehat{x}} - \widehat{x}$ and $\delta_y = \widehat{\widehat{y}} - \widehat{y}$. Then we can compute

$$\widehat{\widehat{z}} = \widehat{z} + \delta_x + \delta_y.$$

There is a savings here since we do not need to really modify the higher order bits of $\widehat{\widehat{z}}$ (except through bit propagation). There are other quantifiable gains: if $\delta\ell$ is the incremental number of bits in $\widehat{\widehat{z}}$ then $\delta_x + \delta_y$ takes only $O(\delta\ell)$ time to compute. So

$$\widehat{z} + \delta_x + \delta_y$$

takes $O(\delta\ell)$ plus possibly the propagation of a carry bit of length ℓ. Note that we can avoid the unpredictable cost of carry propagation if we resort to redundant representation.

At present, incremental computation is not implemented in our system (our bigFloat package would have to be modified to take advantage of this).

6.3. Theory of Root Bounds

A basic operation on expressions is to compare their values. Assuming that expressions are closed under differences (if α, β are expressions, then so is $\alpha - \beta$), it suffices to compare an expression to zero, i.e., determine its sign. To determine the sign of expression α, we can increase the precision-bound of α until we have a definite answer. The catch is to know how to conclude that α is really 0 without looking at infinitely many bits! The theory behind this comes from root separation bounds (e.g., [86]). Notice that without such a theory, we have no basis for doing "exact" computation; traditionally, implementors would just rely on some heuristic cut-off *epsilon* value.

We illustrate this for the family of radical rational expressions (defined above), assuming that the constants and parameters in expressions are rational numbers.

Thus all values are real algebraic numbers. A classic bound from Cauchy [86] says that if α is non-zero then

$$|\alpha| > \frac{1}{1+h}$$

where h is a height bound on α. (The *height* of an algebraic number is the maximum absolute value of the coefficients in its minimal polynomial.) Thus, we only need to evaluate α to an absolute precision of $1 + \log_2(1+h)$ to determine its sign.

It turns out that to maintain bounds on heights of algebraic numbers, we must also maintain bounds on their degrees. We now show the recursive rule for maintaining these bounds. Suppose the algebraic value β is derived from two expressions whose values are α_1 and α_2. Inductively, assume that we have upper bounds on their degrees and heights: (d_1, h_1) and (d_2, h_2). Let $A_i(X)$ be the polynomial with degree $\leq d_i$ and height $\leq h_i$ such that $A_i(\alpha_i) = 0$, $i = 1, 2$. We will describe a polynomial $B(X)$ such that $B(\beta) = 0$ of height at most h and $\deg B \leq d$. We have the following cases:

- (BASIS CASE) $\beta = p/q$ is a rational number, where $p, q \in \mathbb{Z}$. Let $B(X) = qX - p$, $d = 1$ and $h = \max\{|p|, |q|\}$.
- (RECIPROCAL) $\beta = 1/\alpha_1$, $\alpha_1 \neq 0$: let $B(X) = X^{d_1} A_1(1/X)$, $d = d_1$ and $h = h_1$.
- (SQUARE ROOT) $\beta = \sqrt{\alpha_1}$: let $B(X) = A_1(X^2)$, $d = 2d_1$ and $h = h_1$.
- (PRODUCT) $\beta = \alpha_1\alpha_2$: let $B(X) = \mathtt{res}_Y(A_1(Y), Y^{d_2} A_2(X/Y))$, $d = d_1 d_2$ and

$$h = (h_1\sqrt{1+d_1})^{d_2}(h_2\sqrt{1+d_2})^{d_1}.$$

- (SUM/DIFFERENCE) $\beta = \alpha_2 \pm \alpha_1$: let $B(X) = \mathtt{res}_Y(A_1(Y), A_2(X \mp Y))$, $d = d_1 d_2$ and

$$h = (h_1 2^{1+d_1})^{d_2}(h_2\sqrt{1+d_2})^{d_1}.$$

Here $\mathtt{res}_Y(A, B)$ refers to the Sylvester resultant of A and B, viewed as polynomials in the variable Y. The above choices of d, h are justified in [86], using a generalized Hadamard bound. Note that in practice, we will take the ceiling of the indicated height bounds, to avoid non-integer bounds.

7. Conclusion

We see the rise of exact computation as an emerging numerical computation paradigm. One of its main uses is as antidote to nonrobustness problems. It is not a panacea here because of computational bottlenecks. Still, we believe it can very effective for a large class of problems. While f.p. computation will continue to be very important, the development of exact computation is still crucial for several other reasons:

1. First of all, it is probably the computational method of choice whenever it is practicably feasible. This is because an exact algorithm will be simpler than any corresponding "robust" f.p. algorithm.
2. The intellectual challenge of doing exact computation is much deeper than appears at first sight. We expect that many more theoretical and practical tools will be developed. Until recently, attempts to use exact computation often amounts to computing each numerical value exactly. This is a very limited view of the paradigm.
3. Practically all the algorithms developed in algorithmics (computational geometry, in particular) assume exact computation. It is unthinkable to re-develop "robust substitutes" for all these algorithms under the f.p. paradigm. (Current, the number of such substitutes are embarassingly small.)
4. There are many uses of exact computation that complements the f.p. computations: for instance, to validate a f.p. algorithm, it is probably best to compare its results against the exact answer. David Bailey has noted that exact algorithms are very useful in debugging software and hardware.
5. Finally, we believe that this paradigm will shed light on f.p. computations. For instance, we mentioned in the introduction the widespread practice of "epsilon-tweaking" in f.p. algorithms. It is evident that the exact computation paradigm forces us to study precisely these epsilons! In fact, if we allow ourselves enough precision, the correct choice of an epsilon may lead to algorithms that are "exact" in a suitable sense. Even for applications where f.p. computation is the method of choice, we cannot really appreciate the deeper reasons for this choice until we understand the true cost of the exact solution alternative.

References

[1] J.A. Abbott, R.J. Bradford, and J.H. Davenport. The Bath algebraic number package. In *Proc. 1986 ACM Symp. on Symbolic and Algebraic Computation*, 1986.

[2] Oliver Aberth and Mark J. Schaefer. Precise computation using range arithmetic, via C++. *ACM Transaction on Mathematical Software*, 18(4):481–491, 1992.

[3] G.T. Anthony, H.M. Anthony, B. Bittner, B.P. Butler, M.G. Cox, R. Drieschner, R. Elligsen, A. B. Forbes, H. Groß, S.A. Hannaby, P.M. Harris, and J. Kok. Chebyshev best-fit geometric elements. NPL Technical Report DITC 221/93, National Physical Laboratory, Division of Information Technology and Computing, NPL, Teddington, Middlesex, U.K. TW11 0LW, October, 1992.

[4] David Bailey. Algorithm 719 - multiprecision translation and execution of FORTRAN programs. *ACM Transaction on Mathematical Software*, 19(3):288–319, 1993.

[5] David H. Bailey. MPFUN: a portable high performance multiprecision package. Technical Report RNR-90-022, NASA Ames Research Center, 1990. Email: dbailey@nas.nasa.gov.

[6] M Beeler, R.W. Gosper, and R. Schroepppel. HAKMEM. A.I. Memo 239, M.I.T., February 1972.

[7] M. Ben-Or, D. Kozen, and J. Reif. The complexity of elementary algebra and geometry. *Journal of Computer and System Sciences*, 32:251–264, 1986.

[8] M. Benouamer, P. Jaillon, D. Michelucci, and J.-M. Moreau. A lazy solution to imprecision in computational geometry. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 73–78, Waterloo, Canada, 1993.

[9] J. Bentley and T. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Computers*, C-28(9):643–647, 1979.

[10] H.-J. Boehm, R. Cartwright, M. Riggle, and M. J. O'Donnell. Exact real arithmetic: a case study in higher order programming. *Proc. ACM conf. on List and Functional Programming*, pages 162–173, 1986.

[11] G. Bohlender, W. Walter, P. Kornerup, and D.W. Matula. Semantics for exact floating point operations. *IEEE Symp. on Computer Arithmetic*, 10:22–26, 1991. Grenoble, France.

[12] Gerd Bohlender, Christian Ullrich, Jürgen Wolff von Gudenberg, and Louis B. Rall. *Pascal-SC*, volume 17 of *Perspectives in Computing*. Academic Press, Boston-San Diego-New York, 1990.

[13] A. Borning. The programming language aspects of thinglab, a constraint-based simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3:353–387, 1981.

[14] W. Bouma, I. Fudos, C. Hoffmann, J. Cai, and R. Paige. A geometric constraint solver. Technical Report CSD-TR-93-054, Computer Science Department, Purdue University, August, 1993.

[15] Richard P. Brent. ALGORITHM 524: MP, a Fortran multiple-precision arithmetic. *ACM Trans. on Math. Software*, 4:71–81, 1978.

[16] Richard P. Brent. A Fortran multiple-precision arithmetic package. *ACM Trans. on Math. Software*, 4:57–70, 1978.

[17] B. Buchberger, G. E. Collins, and R. Loos (eds.). *Computer Algebra*. Springer-Verlag, 1983.

[18] Christoph Burnikel, Kurt Mehlhorn, and Stefan Schirra. How to compute the Voronoi diagram of line segments: theoretical and experimental results. In *Proc. 2nd European Symposium on Algorithms (ESA'94)*, 1994. Utrecht, the Netherlands, September 26-28, 1994 (to appear).

[19] Jacqueline D. Chang and Victor Milenkovic. An experiment using LN for exact geometric computations. *Proceed. 5th Canadian Conference on Computational Geometry*, pages 67–72, 1993. University of Waterloo.

[20] D.M. Chiarulli, W.G. Rudd, and D.A. Buell. DRAFT: a dynamically reconfigurable processor for integer arithmetic. In *Proc. 7th Symp. on Computer Arithmetic*, pages 309–321. IEEE Computer Society Press, 1989.

[21] Joonsoo Choi, Jürgen Sellen, and Chee Yap. Approximate Euclidean shortest path in 3-space. *10th ACM Symposium on Computational Geometry*, 1993. (to appear).

[22] Kenneth L. Clarkson. Safe and effective determinant evaluation. *IEEE Foundations of Computer Science*, 33:387–395, 1992.

[23] C.W. Clenshaw, F.W.J. Olver, and P.R. Turner. Level-index arithmetic: an introductory survey. In P.R. Turner, editor, *Numerical Analysis and Parallel Processing*, pages 95–168. Springer-Verlag, 1987. Lecture Notes in Mathematics, No.1397.

[24] C.W. Clenshaw and P.R. Turner. The symmetric level-index system. *IMA J. Num. Anal.*, 8:517–526, 1988.

[25] G. E. Collins. Computer algebra of polynomials and rational functions. *American Mathematical Monthly*, 80:725–755, 1975.

[26] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *2nd GI Conf. on Automata Theory and Formal Languages, Lecture Notes in Computer Science*, volume 33, pages 134–183. Springer-Verlag, 1975.

[27] M. Coste and M. F. Roy. Thom's lemma, the coding of real algebraic numbers and the computation of the topology of semi-algebraic sets. *Journal of Symbolic Computation*, 5:121–130, 1988.

[28] J. Cox and K. McAloon. *Decision Procedures for Constraint Based Extensions of Datalog*, chapter 2. MIT-Press, 1993.

[29] J. Cox, K. McAloon, and C. Tretkoff. Computational complexity and constraint logic programming languages. *Annals of Math. and Artificial Intellegence*, 5:163–190, 1992.

[30] D. DeGroot and G. Lindstrom. *Logic Programming, Functions, Relations, and Equations*. Prentice-Hall, 1986.

[31] Richard N. Draper. Supercomputing '93. *Computational Science & Engineering*, 1(1):85–86, 1994.

[32] T. Dubé and Chee Yap. A basis for implementing exact geometric algorithms, September, 1993. Extended Abstract.

[33] Thomas Dubé and Chee Yap. The geometry in constraint logic programs. *Proceed., First Workshop on Principles and Practice of Constraint Programming*, 1993.

[34] Herbert Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, 1987.

[35] David erg. What every computer scientist should know about floating-point , arithmetic. *ACM Computing Surveys*, 23:5–48, 1991.

[36] Lars Warren Ericson and Chee K. Yap. The design of LINETOOL: a geometric editor. *ACM Symp. on Computational Geometry*, 4:83–92, 1988.

[37] Shaw C. Feng and Theodore H. Hopp. A review of current geometric tolerancing theories and inspection data analysis algorithms. Technical Report NISTIR-4509, National Institute of Standards and Technology, U.S. Department of Commerce. Factory Automation Systems Division, Gaithersburg, MD 20899, February 1991.

[38] A. B. Forbes. Geometric tolerance assessment. NPL Technical Report DITC 210/92, National Physical Laboratory, Division of Information Technology and Computing, NPL, Teddington, Middlesex, U.K. TW11 0LW, October, 1992.

[39] S. J. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[40] Steven Fortune and Christopher van Wyk. Efficient exact arithmetic for computational geometry. *ACM Symp. on Computational Geometry*, 9:163–172, 1993.

[41] Steven Fortune and Christopher van Wyk. LN User Manual, 1993. AT&T Bell Laboratories.

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

[43] G. Golub and B. Parlett. Dedication to James Wallace Givens, Jr. *SIAM J. Matrix Analysis and Applications*, 12(1):(forward), 1991.

[44] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.*, 4:74–123, 1985.

[45] Joos Heintz and Jacques Morgenstern. On the intrinsic complexity of elimination theory. Research Report 1923, INRIA, Sophia, Antipolis, I3S, 06560 Valbonne, France, May, 1993.

[46] N. Heintze, J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. *The CLP($\mathcal{R}$) Programmer's Manual*, version 1.2 edition, September 1992.

[47] N. Heintze, S. Michaylov, and P. Stuckey. CLP($\mathcal{R}$) and some electrial engineering problems. In J.-L. Lassez, editor, *Logic Programming, Proceedings of the Fourth International Conference*, pages 675–703, 1987.

[48] Christoph M. Hoffman. *Geometric and Solid Modeling: an introduction.* Morgan Kaufmann Publishers, Inc, San Mateo, California 94403, 1989.

[49] Christoff M. Hoffmann. The problems of accuracy and robustness in geometric computation. *IEEE Computer*, 22(3), March 1989.

[50] Theodore H. Hopp. Computational metrology. In *Proc. 1993 International Forum on Dimensional Tolerancing and Metrology*, pages 207–217, New York, NY, 1993. The American Society of Mechanical Engineers. CRTD-Vol.27.

[51] T.E. Hull, A. Abraham, M.S. Cohen, A.F.X. Curley, C.B. Hall, D.A. Penny, and J.T.M. Sawchuk. Numerical Turing. *ACM SIGNUM newsletter*, 20(3):26–34, July, 1985.

[52] T.E. Hull and M.S. Cohen. Toward an ideal computer arithmetic. In *Proceedings of the 8th Symposium on Computer Arithmetic*, pages 5–48. IEEE, 1987.

[53] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987.

[54] Yasumasa Kanada and Tateaki Sasaki. LISP-based "big-float" system is not slow. *ACM SIGSAM Bulletin*, 15(2):13–19, May, 1981.

[55] M. Karasick, D. Lieber, and L. R. Nackman. Efficient Delaunay triangulation using rational arithmetic. *ACM Trans. on Graphics*, 10:71–91, 1991.

[56] Dexter Kozen and Chee Yap. Algebraic cell decomposition in *nc. IEEE Foundations of Computer Science*, 26:515–521, 1985.

[57] J.L. Lassez, T. Huynh, and K. McAloon. Simplification and elimination of redundant linaer arithmetic constraints. In *Logic Programming, Proceedings of the North American Conference*, pages 37–51, 1989.

[58] William Leler. *Constraint Programming Languages: their specification and generation*. Addison-Wesley, Reading, Massachusetts, 1988.

[59] M. Metzger. FORTRAN-SC, a FORTRAN extension for engineering/scientific computation with access to ACRITH: Demonstration of the compiler and sample programs. In R. E. Moore, editor, *Reliability in Computing*, pages 63–79. Academic Press, Boston-San Diego-New York, 1988.

[60] Maurice Mignotte. *Mathematics for Computer Algebra*. Springer-Verlag, 1992.

[61] Ramon E. Moore. *Interval Analysis*. Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, N.J., 1966.

[62] Alvin G. Neumann. The new ASME Y14.5M standard on dimensioning and tolerancing. In *Proc. 1993 International Forum on Dimensional Tolerancing and Metrology*, pages 7–18, New York, NY, 1993. The American Society of Mechanical Engineers. CRTD-Vol.27 (Also: Manuf.Review, March 1994).

[63] S. Ocken, J. T. Schwartz, and M. Sharir. Precise implementation of CAD primitives using rational parameterization of standard surfaces. In J. Hopcroft, J. Schwartz, and M. Sharir, editors, *Planning, Geometry, and Complexity of Robot Motion*, pages 245–266. Ablex Pub. Corp., Norwood, NJ, 1987.

[64] C. H. Papadimitriou. An algorithm for shortest-path motion in three dimensions. *Inform. Process. Lett.*, 20:259–263, 1985.

[65] David A. Patterson and John L. Hennessy. *Computer Architecture: a quantitative approach*. Morgan Kaufmann Publishers, Inc, San Mateo, California, 1990. (with an appendix on Computer Arithmetic by David Goldberg).

[66] D. A. Pope and M. L. Stein. Multiple precision arithmetic. *Communications of the ACM*, 3:652–654, 1960.

[67] John R. Rice. Academic programs in computational science and engineering. *Computational Science & Engineering*, 1(1):13–21, 1994.

[68] U. Roy and X. Zhang. Establishment of a pair of concentric circles with the minimum radial separation for assessing rounding error. *Computer Aided Design*, 24(3):161–168, 1992.

[69] S. M. Rump. Algorithms for verified inclusions – theory and practice. In R. E. Moore, editor, *Reliability in Computing*, pages 109–126. Academic Press, Boston-San Diego-New York, 1988.

[70] Tateaki Sasaki. An arbitrary precision real arithmetic package in REDUCE. In *Lecture Notes in Computer Science*, volume 72, pages 358–368. Springer-Verlag, 1979. Proc. ACM EUROSAM'79.

[71] Jerry Schwarz. A C++ library for infinite precision floating point. *Proc. USENIX C++ Conference*, pages 271–281, 1988.

[72] Jerry Schwarz. Implementing infinite precision arithmetic. In *Proc. 9th Symp. on Computer Arithmetic*, pages 10–17. IEEE Computer Society Press, 1989.

[73] B. Serpette, J. Vuillemin, and J.C. Hervé. BigNum: a portable and efficient package for arbitrary-precision arithmetic. Research Report 2, Digital Paris Research Laboratory, May, 1989.

[74] Ehud Shapiro. Alternation and the computational complexity of logic programs. *J. Logic programming*, 1:19–33, 1094.

[75] David Smith. Algorithm 693 - a FORTRAN package for floating-point multiple-precision arithmetic. *ACM Transaction on Mathematical Software*, 17(2):273–283, 1991.

[76] David Stieren, Ralph Veale, Howard Harary, and Shaw Feng. U.s. Navy coordinate measuring machines: a study of needs. Technical Report NISTIR-5378, National Institute of Standards and Technology, U.S. Department of Commerce. Factory Automation Systems Division, Gaithersburg, MD 20899, December 1993.

[77] G. Sussman and G. Steele. Constraints – a language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14:1–39, 1980.

[78] Peter R. Turner. A software implementation of SLI arithmetic. In *Proc. 9th Symp. on Computer Arithmetic*, pages 18–24. IEEE Computer Society Press, 1989.

[79] Herbert B. Voelcker. A current perspective on tolerancing and metrology. In *Proc. 1993 International Forum on Dimensional Tolerancing and Metrology*, pages 49–60, New York, NY, 1993. The American Society of Mechanical Engineers. CRTD-Vol.27.

[80] J. Wolff von Gudenberg. Reliable expression evaluation in PASCAL-SC. In R. E. Moore, editor, *Reliability in Computing*, pages 81–97. Academic Press, Boston-San Diego-New York, 1988.

[81] Jean Vuillemin. Exact real computer arithmetic with continued fractions. *IEEE Transaction on Computers*, 39(5):605–614, 1990. Also, 1988 ACM Conf. on LISP & Functional Programming, Salt Lake City.

[82] R. K. Walker. CMM form tolerance algorithm testing. GIDEP Alert X1-A1-88-01 and X1-A1-88-01a, Government-Industry Data Exchange Program, Pomona, CA, 1988.

[83] W. Walter. FORTRAN-SC, a FORTRAN extension for engineering/scientific computation with access to ACRITH: Language descrioption with examples. In R. E. Moore, editor, *Reliability in Computing*, pages 43–62. Academic Press, Boston-San Diego-New York, 1988.

[84] E. Wiedmer. Computing with infinite objects. *Theoretical Computer Science*, 10:133–155, 1980.

[85] Chee Yap. Towards exact geometric computation. In *Fifth Canadian Conference on Computational Geometry*, pages 405–419, Waterloo, Canada, August 5–9 1993. Invited Lecture.

[86] Chee Yap. *Fundamental Problems in Algorithmic Algebra*. Princeton University Press, to appear. Available on request from author (and via anonymous ftp).

[87] Jiaxun Yu. *Exact arithmetic solid modeling*. Ph.D. dissertation, Department of Computer Science, Purdue University, West Lafayette, IN 47907, June 1992.