A Basis for Implementing Exact Geometric Algorithms

(Extended Abstract)

Thomas Dubé*
The College of the Holy Cross
Worcester, Massachusetts

Chee-Keng Yap†
Courant Institute, NYU
New York, New York

October 15, 1993

Abstract

Our ultimate goal is to develop *exact geometric computation* as an viable alternative to the usual computing paradigm based on fixed-precision arithmetic. Use of exact computation has numerous advantages; in particular, it will abolish the non-robustness issues that has so far defied satisfactory solution. In this paper we describe two computational tools which can be a basis for exact geometric computing:

- `bigFloat`: a multi-precision floating-point number system with automatic error-handling.
- `bigExpression`: an expressions package based on a precision-driven mechanism. This package is built on top of `bigFloat`.

We discuss the rationale for the design of these packages. Experimental results are reported. The major contributions of our work are:

- We demonstrated for the first time that, because of the existence of root bounds, approximate values (say, represented in `bigFloat`) are useful in exact computation. In fact, we use the example to Fortune's algorithm to show that this is more attractive than the usual approach of reduction to exact integer computation.
- We introduced the concept of *precision-driven* computation through our design of `bigExpression`. This is a general paradigm and should be contrasted with the so-called *lazy-evaluation* appraoch.

1 Introduction

There is a growing interest among computational geometers in implementation issues. This ranges from general computing environments (e.g., [16, 17, 13, 14]) to the study of individual algorithms (e.g., [15, 9, 20]). Not surprisingly, this interest is accompanied by concerns about non-robustness in geometric algorithms. These issues are essentially a byproduct of computing in fixed-precision arithmetic, which is invariably some floating-point package. This is a well-known and widespread concern, touching practically every community of scientific computing. To underscore this, the 1989 Turing award is a recognition of Kahan's contribution towards

*Part of this work is done while this author was visiting the Courant Institute.
†This author is supported by NSF grant #CCR-9002819.

a rational design for floating-point architecture. Yet we should keep in perspective that this contribution, important as it is, can (i) at best delay the onset of non-robustness problems and (ii) only makes the inevitable failures more predictable and machine-independent! In stark contrast, non-robustness issues are non-existent if we compute "exactly". At least in the context of geometric algorithms, this concept of *exact computation* is clarified and expanded in our companion paper [21]. Our fundamental thesis there is that

- Exact geometric computation appears feasible for a large class of problems.
- If robustness is a serious issue for an application, then exact computation (in one of its forms) is perhaps the only reliable general solution.
- Exact computation involves a rich body of *computational tactics* – it certainly goes beyond the naive notion that each numerical operation must be computed exactly.
- These tactics must be embodied in software packages built on top of suitable number packages (which itself must go beyond the traditional large number packages).

We hasten to add, non-robustness *is* tolerable in many applications. But there is a growing number of applications for which it no longer makes sense to compute in the fixed-precision mode. Furthermore, it is no surprise that exact computation comes with a computational cost. Ultimately, it is the user who must consider the tradeoffs between the cost of non-robustness against the the cost of exact computation. One of our goals is a practical one: *to reduce the cost of exact computation as much as possible* (to reach the theoretical limit, as it were). We believe that at present, the cost of exact computation is nowhere near what it should be (cf. the "anecdotes" in [21]).

In conclusion, we believe that exact computation represents an *emerging* new computational paradigm [23]. The present paper is a contribution towards this paradigm. In particular, we describe the design of two software packages that could be the "basis" of efficient and convenient exact geometric computation. Our implementation is based on `C++`.

2 Related Work

One of the most well-known multiprecision number packages is from Brent [4]. More recent work includes Serpette, Vuillemin and Herve [19], and Bailey [2, 1]. The latter is written in Fortran and is notable in that it is written with vector supercomputers and RISC floating point architecture in mind. Both Brent's and Bailey's system uses floating-point numbers and are Fortran-based. Serpette et al's system is C-based with a special assembly coded kernel for some machines. For a general survey of multi-precision packages, see [23].

The expression compiler of Fortune and Van Wyk [10, 11] has much of the same motivation as our `bigExpression` package. Some fundamental differences are their stronger use of compiler techniques and their use of static error bounds. Chang and Milenkovic described their experience with the system of Fortune and Van Wyk in [6].

Benouamer, Jaillon, Michelucci and Moreau [3] described a `C/C++` package for lazy evaluation of expression. Their approach is similar to ours in the use of run-time techniques, but with a fundamental differences: while theirs is a "lazy approach", we call our more active approach "precision-driven". They approximate values using intervals with rational endpoints, in contrast to our use of big floats. With the Bentley-Ottmann algorithm, 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}.

Recently, Burnikel, Mehlhorn and Schirra [5] reported on the implementation of an exact Voronoi diagram algorithm for line segments. As in our paper, they analyzed the precision necessary for exact comparisons using big floats, and contrast that with the usual approach of repeated-squaring. Their experimental results seems to suggest a tie between these two approaches. This is in contrast to our analysis for a related problem (see below).

3 Background Considerations

Floating-Point Numbers and `bigFloat`. We said that exact computation does not always require exact intermediate results. This remark may not obvious because it is often implied in the past that exact computation is synonymous with exact numerical computation. Below we will show the theoretical basis for using approximate values. In any case, if we are to use approximate values, what form shall they take? As the last 40 years of numerical computing have testified, they seem best embodied in the concept of floating-point numbers. To describe such numbers, fix some integer $B > 1$ as the *base*. Usually B equals 10 or a power of 2. For any non-zero integer f, define its *base-B normalization* to be

$$\langle f \rangle_B := f \cdot B^{-\lceil \log_B |f| \rceil}.$$

We omit the subscript B in $\langle f \rangle_B$ when it is not ambiguous. Thus $1 > |\langle f \rangle| \geq 1/B$, and it is tantamount to viewing f as a B-ary number and placing a B-ary point just before the most significant digit of f. Thus, $\langle 123 \rangle_{10} = 0.123$. Then *floating-point (f.p.) numbers, to base B* are real numbers of the form $B^e \cdot \langle f \rangle_B$, where e, f are arbitrary integers. Briefly, floating-point numbers are useful because they decouple the precision of the approximation (the number of digits in f) from the magnitude represented by e. Actually, this idea can

be carried to an extreme using *level-index arithmetic* (see [7]). In analogy to the familiar `bigInteger` packages, we call our computer realization of such numbers `bigFloat`.

Error bounds and `bigFloat`. The system of base-B floating-point numbers are intended to be approximations for real numbers. If we implement a floating-point package without automatic tracking of error, then in the context of exact computation, we expect the user would explicitly keep track of some error bounds. To improve the usefulness of our package, we automatically carry with each f.p. number an *error bound*. Thus the triple (f, e, d) represents the "floating-point number with error" or *floating-point range*,

$$\langle f \pm d \rangle \cdot B^e.$$

floating-point range. When we perform arithmetic operations on two such ranges, it is easy to automatically propagate the error bounds. But for efficiency reasons, we will usually *normalize* the error so that $0 \leq d < B$. Of course, we can trade-off efficiency against accuracy of the error bound by using a larger range of values for d. But simple examples show that our choice (with $B = 2^{15}$) is quite effective in giving useful error bounds.

Expressions and Precision-bounds. There is another related notion of error bound: there are situations where we want to specify an "error bound" a variable e, and even vary these bounds in the course of a computation. This seems to contradict the previous view of error bounds; the apparent contradiction comes from our tendency to confuse the concept of an *expression* with its *value* (which is a number). For instance, a variable e may really represent the expression $ad - bc$. But e also has a value, provided the variables a, b, c, d all have values. Once we make this distinction, we can ask for the value of e to within any "error bound". To distinguish this user-specified notion of error bound from the one in `bigFloat`, we call this the *precision-bound*. A precision-bounds is given by a pair of integers, $[a, r]$, where a bounds the absolute error and r the relative error. The precise semantics will be explained below. In our treatment, an `Expr` variable e is associated with three data items:

- A numerical expression E. We implement only the operations $+, -, \times, \div, \sqrt{}$, and so E may be called a *rational radical expression*.
- A precision-bound $[a, r]$.
- An approximate value α. Usually α is a `bigFloat` or a rational.

We guarantee that α approximates the value of E to the precision-bound $[a, r]$. Note that the user chooses E and $[a, r]$ but the system generates α. Furthermore, when the user change $[a, r]$ or E, this is an implicit request for the system to update α.

Semantics of precision-bounds. Let $[a, r]$ be a pair of integers, called a *(composite) precision-bound*. We say that a real number $\widehat{x}$ *approximates* another real x to *precision* $[a, r]$ if *either* the absolute error $|x - \widehat{x}|$ is at most 2^{-a} *or* the relative error $|(x - \widehat{x})/x|$ is at most 2^{-r}. We call this the "or" semantics, since it is clear that could just have easily defined the "and" semantics. Depending on the application, it is common to specify the precision in terms of either absolute or relative terms. We choose to use the composite notion because it flexibly encompasses both. If for example, a user wishes to specify only relative precision r they may do so by specifying a bound of $[\infty, r]$.

Since our "or" semantics for composite precision-bounds is not the obvious choice, we try to motivate it. In fact, Schwarz [18] had already used the "and" semantics. 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. But our "or" semantics is easily seen to be a close approximation.

Let us illustrate one use of composite precision bounds. Suppose that a certain computation involves only rational numbers with at most N and D bits in the numerator and denominator, respectively. Under certain assumptions, it seems reasonable to approximate these values with big floats that does not exceed absolute precision D or relative precision $N + D$. That is, we can set a global precision bound of $[a, r] = [D, N + D]$ for all expressions in this computation.

Another reason why we chose to consider both relative and absolute precision is their relationship to the basic arithmetic operations. Roughly speaking, addition and subtraction preserve absolute precision, while multiplication and division preserve relative precision. The desire to specify relative precision was partially responsible for the decision to include dynamic error bounds with bigFloat values. If static error bounds are used, then it is only possible to work with absolute precision.

Basis for Geometric Computation. In some sense, multiprecision number packages serve as the ultimate basis for exact computation. The problem is that, from the viewpoint of a number package, exact computation amounts to computing each arithmetic operation exactly (or to user specified precision). This is a limited view which misses the bigger picture of geometric computing: *an important aspect of any geometric algorithm is that calls to number packages can invariably be structured into larger units called* expressions. At the expression level, exact computation takes on a whole new meaning – and opportunities for optimization seem wide open. If arithmetic operations are atoms, then expressions are the molecules or

even polymer, if we may use a physical analogy. The distinguishing mark of a geometric algorithm is the intertwining of combinatorial with numerical computation [21]. It seems that an unstructured interaction between these two aspects of geometric algorithms is generally undesirable. We suggest that for geometric algorithms, the level of expressions is the appropriate one for interaction. It is in this sense that we think of `bigExpression` as a "basis" for exact geometric computation: ideally, we want to design geometric algorithms so that they never access number packages except through `bigExpression`.

4 The Exact Basis for Using Approximations

Although big float packages have been around almost as long as big integer packages, their role has always been in support of the fixed-precision computation paradigm (see [23]). For instance, Brent's `MP` is designed for this mode of computation, even though the precision is no longer dictated by the hardware. Since the use big floats in exact computation seems novel, it is important (1) to establish that the basis for their use in exact computation, and (2) to demonstrate their usefulness. We now treat (1), leaving (2) to the next section.

The fundamental issue here is how to determine the sign of an expression. Since we can specify precision bounds, we can keep increasing the precision until we obtain a positive or negative sign. But what if the sign is really zero? We must have some á priori bound on when to stop increasing the precision *and* conclude that the sign is zero. Basically, we can determine such á priori bounds because of the existence of root bounds. We now make this precise.

If α is an algebraic number that is a root of the integer polynomial $A(X)$ then we use the following bound from Landau (see [22]): $|\alpha| \geq (\|A(X)\|_2)^{-1}$ where $\|A(X)\|_2$ refers to the Euclidean length of the coefficient vector of $A(X)$. Assume that α is the value of a "rational radical expression" that is recursively built-up from the rational constants using the operations of

$$+, -, \times, \div, \sqrt{}. \tag{1}$$

Our fundamental goal can be reduced to obtaining lower bound on $|\alpha|$ when $\alpha \neq 0$. Towards this end, we maintain with each node of E an upper bound on the degree and length of the algebraic number represented at that node. If α is an algebraic number, we call the pair (d, ℓ) a *degree-length* bound on α if there exists a polynomial $A(X) \in \mathbb{Z}[X]$ such that $A(\alpha) = 0$, $\deg(A) \leq d$ and $\|A\|_2 \leq \ell$. Note that this implies that $|\alpha| \geq 1/\ell$ (Landau's bound) and so we only need to a big float approximation of α with absolute precision $(\lg \ell) + O(1)$ in order to determine its sign. We now derive the recursive rules for maintaining this bound.

Suppose the algebraic number β is obtained from α_1 and α_2 by one of the 5 operations

in (1). Inductively, assume a degree-length bound of (d_i, ℓ_i) on α_i, $(i = 1, 2)$, and let $A_i(X)$ be a polynomial that achieves this bound. We now describe a polynomial $B(X)$ such that $B(\beta) = 0$, and a corresponding degree-length bound (d, ℓ) on β.

Theorem 1 *The pair (d, ℓ) is a degree-length bound for β in each of the cases, as listed below.*

- (BASIS) $\beta = p/q$ is a rational number, where $p, q \in \mathbb{Z}$. Choose $B(X) = qX - p$, $d = 1$ and $\ell = \sqrt{p^2 + q^2}$.
- (INVERSE) $\beta = 1/\alpha_1$: choose $B(X) = X^{d_1} A_1(1/X)$, $d = d_1$ and $\ell = \ell_1$.
- (SQUARE-ROOT) $\beta = \sqrt{\alpha_1}$: choose $B(X) = A_1(X^2)$, $d = 2d_1$ and $\ell = h_1$.
- (PRODUCT) $\beta = \alpha_1 \alpha_2$: choose $B(X) = \mathtt{res}_Y(A_1(Y), Y^{d_2} A_2(X/Y))$, $d = d_1 d_2$ and
$$\ell = \ell_1^{d_2} \ell_2^{d_1}.$$
- (SUM/DIFFERENCE) $\beta = \alpha_2 \pm \alpha_1$: choose $B(X) = \mathtt{res}_Y(A_1(Y), A_2(X \mp Y))$, $d = d_1 d_2$ and
$$\ell = \ell_1^{d_2} \ell_2^{d_1} 2^{d_1 d_2 + \min\{d_1, d_2\}}.$$

Remarks.
1. Here $\mathtt{res}_Y(A, B)$ is the resultant of polynomials A and B in Y. Only the SUM/DIFFERENCE case in this theorem is non-trivial, and it is based a bound of Graham and Goldstein [12, 22].
2. The use of big floats as approximate values is by no means essential. We could use any dense number system (e.g., rational numbers) or intervals, as in [3].

5 The Advantage of Using `bigFloat`

The example arises in the exact implementation of Fortune's plane sweep algorithm [8] for Voronoi diagrams of a point set.

Comparison of Priorities. In Fortune's algorithm, we need to order a sequence of "events" according to their "priorities". It turns out that we need to make comparisons of the form

$$\frac{a + \sqrt{b}}{d} : \frac{a' + \sqrt{b'}}{d'}. \tag{2}$$

where a, b, d essentially have $3L, 6L$ and $2L$ bits, respectively, assuming input points has L-bit integer coordinates.

The Method of Repeated-Squaring It is clear that this comparison can be made by repeated squaring. In fact, Fortune[1] has already noted that this may involve $20L$-bit integers. But it turns out that the details are somewhat involved, a fact which previous authors seems to have missed. To see this, we may assume that we want to verify whether $d'(a+\sqrt{b}) \geq d(a'+\sqrt{b'})$ or,

$$(I): \quad d'\sqrt{b} \geq d\sqrt{b'} + e \tag{3}$$

where $e = da' - da'$. Note that we assume $d, d' \neq 0$, not necessarily positive. Then equation (3) is equivalent to the disjunction of the following three conjuncts:

$$(II): \quad (d' \leq 0), (d\sqrt{b'} + e \leq 0), (d'^2 b \leq (d\sqrt{b'} + e)^2). \tag{4}$$

$$(III): \quad (d' \geq 0), (d'^2 b \leq (d\sqrt{b'} + e)^2). \tag{5}$$

$$(IV): \quad (d' \geq 0), (d\sqrt{b'} + e \leq 0). \tag{6}$$

These can ultimately be expanded into a Boolean function of the sign of the following 6 expressions:

$$d, \quad d', \quad e, \quad d^2 b' - e^2, \quad d'^2 b - d^2 b' - e^2, \quad 4d^2 e^2 b' - (d'^2 b - d^2 b' - e^2)^2.$$

Alternatively, we can expand $(II) \vee (III) \vee (IV)$ into a disjunction of 18 conjuncts involving the signs of these expressions. In any case, having to evaluate such a large number of expressions leaves something to be desired. Similar observations clearly apply to the repeated-squaring technique.

The Method of Approximate Numbers. To use approximate numbers, we need the following lemma:

Lemma 2 *It suffices to compute the values in (2) to $25L + O(1)$ bits in order to make an error-free comparison.*

Classical root separation bounds gives us $60L$ bits. Our proof uses a bound of Goldstein-Graham [12, 22].

The advantage of repeated squaring over the approximate square-root approach is that only integer operations are used. But approximate square-roots are essentially as fast as multiplication (in theory and not much slower in practice). The biggest advantage of approximate square-roots is that we do not need to perform up to $20L$-bit arithmetic for *each* comparison – instead, we compute each α to $25L$-bits of accuracy once. Subsequent comparisons using the pre-computed approximate square-roots is simple and fast, as compared to the evaluation of a large Boolean function in the repeated squaring approach.

[1] Private communication.

Remark: Mehlhorn points out that we can improve the root separation bound to $20L$-bits. But recently, Sellen and Yap (to appear) further reduced this to $15L$-bits, and they show that this $15L$-bit bound is the best possible. So the comparative advantage of using big floats proves to be even greater.

6 `bigExpression`: precision-driven computation

The most important feature of our package is the use of precision-driven computation. This should be contrasted to the system of Benouamer, Jaillon, Michelucci and Moreau [3]. They used a "lazy approach" that increases the precision of leaves in an expression, and automatically propagate this increased precision up the tree until the root. It is not clear how one predicts the necessary precision at the leaves to ensure a desired precision at the root, so presumably there is a loop to repeat this process. Their approach may be regarded as a "bottom-up" propagation of error-bounds; our evaluation evaluation algorithm has an additional "top-down" propagation of precision-bounds. Recall that our expression has an associated precision bound. *We first propagate the precision bound from the root to all the nodes of the expression.* At each leaf, we may assume that we have a procedure to extract an approximate value to satisfy the precision bound at the leaf. Then these values are propagated back to the root – but the precision achieved at the root is now guaranteed to be the desired one.

Our distinction between error-bounds and precision-bounds, in some sense, just reflects the difference between a half-empty cup and a half-full cup, whether you look at the same phenomenon as a pessimist or as an optimist. But we can distinguish them within the context of our evaluation process:
(i) Error-bounds appear in the leaves of an expression when we substitute parameter values with approximations. These error-bounds 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.
(ii) 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.

The hard part of our precision approach is the top-down propagation of precision bounds! Once these bounds are computed, the rest is presumably automatic. But the algorithms for propagating precision-bounds is not at all obvious. The interaction of the relative and absolute bounds adds to the complications. In the full paper, we will describe these algorithms (see also [23]). Here, we illustration the case of square-roots.

Propagating Precision-bounds for Square-root. Suppose we require the value of a tree node E up to some precision $\epsilon = (a_0, r_0)$. We determine *a priori* the required precision of the sub-expression values:

Lemma 3 *To evaluate the square root of an expression E_1 to a precision (a_0, r_0), it suffices to evaluate E_1 with precision $(2a_0 + 2, r_0 + 1)$.*

7 Experimental Results

Preliminary testing indicates that the packages are not unacceptably slow, and do provide a reasonable alternative to fixed-precision computation. Among the algorithms which we have used to test our packages is Fortune's netlib distributed code for Voronoi diagram. We modified it for `C++`. Using his distributed data set containing 100 sites, we find that using `bigFloat` the execution is approximately 10 times slower than using machine floating point. When we use the `bigExpression` package, the execution time is approximately 66 times slower than floating point. Similar results have been produced with various point sets ranging in size from 50 to 400 points. For this problem, a profiler indicates that 55% of the execution time is spent for memory allocation, and the bulk of the remaining time is used in performing the `bigFloat`/`bigExpression` computations. Other time spent in the algorithm is negligible.

Another test involved computing the determinant of a Hilbert matrix. The initial entries in the matrix were rational numbers. We computed the determinant two ways:

- converting the numbers to `bigExpression` and performing the determinant algorithm allowing the precision of the sub-expressions to be determined automatically by the system. The final result is guaranteed to have at least 40 (accurate) bits of precision.
- converting the numbers to `bigFloat` with k bits of precision, and then computing the determinant using only the available precision in the system. The number of bits of precision p in the computed determinant is of course less than k.

The results of these experiments are summarized in Fig. 1. In these tests it was found that approximately 69% of the time was spent in memory management and the rest was used in

matrix dimension	time (sec) `bigExpression`	time (sec) `bigFloat`	starting precision k	ending precision p
8	2.34	0.16	40	3
		0.18	60	19
		0.19	80	51
12	24.88	0.55	80	12
		0.59	100	28
		0.63	120	44
16	110.90	1.37	120	1
		1.43	140	17
		1.54	160	49
20	393.37	3.13	160	1
		3.24	180	17
		3.44	200	33
		3.65	220	49

Figure 1: Timing results for Hilbert matrix determinant

the computation.

At this point, we have not yet optimized our packages. In particular we do not yet handle our own memory management and profiling shows this to be the most costly part. We believe that future realizations of these packages will be considerably faster.

8 Conclusion

The exact computation paradigm is important to develop for a variety of reasons. At least, users should be given a viable alternative when the standard computing paradigm based on fixed-precision computation becomes inadequate.

1. In this paper, we propose two basic pieces of software that are essential for the above basic goal. We described considerations that went into our design of two packages. In our system, the automatic tracking of error appear in two places: in `bigFloat`s and in `bigExpression`s. The latter embodies a novel evaluation mechanism based on the *precision-driven paradigm* which we expect to have other applications.

2. `bigExpression`s is relatively easy to use. It does change the programming style somewhat, forcing the user to construct expressions ahead of any computation. But this discipline is probably ultimately for the good.

3. The present implementation can be improved in several ways. This includes (a) devel-

oping an incremental big number package, and (b) incorporating a determinant operator as primitive, (c) incorporating more compiler technology into its design. The reason for (b) is that currently, determinants can only be implemented as the evaluation of a polynomial in $n!$ terms and we cannot take advantage of polynomial algorithms such as Bareiss's algorithm. This is only feasible for small n. Note that (c) is an enormous topic with many possibilities: global optimization, expression restructuring, common expressions, run-time tactics, etc.

4. Currently, we are implementing a *heterogeneous number package* based on a `Real` class. A `Real` object can equally represent a big integer, big rational, big float, machine types or other user constructed number packages. Eventually we will incorporate general algebraic numbers as well. Although this is in general expensive, it should be available when the user has a genuine need for it. With the user-specified precision, the user only needs to pay for the amount of precision desired.

5. The availability of such packages would in any case be a boon to the experimental side of the field. Even when an application could not ultimately afford the cost overhead of exact computation, it is still useful in debugging and testing fixed-precision algorithms. By substituting our package for the fixed-precision arithmetic, the user can see if a mysterious failure in their fixed-precision computation is really due to non-robustness problems. As one is never sure about the results of most fixed-precision computations, it is nice to be able to check results using our package.

References

[1] 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`.

[2] David H. Bailey. Automatic translation of Fortran programs to multiprecision. Technical Report RNR-91-025, NASA Ames Research Center, May 6, 1993. Email: `dbailey@nas.nasa.gov`.

[3] 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.

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

[5] 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).

[6] 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.

[7] 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.

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

[9] Steven Fortune. Numerical stability of algorithms for 2-d Delaunay triangulations and Voronoi diagrams. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 83–92, 1992.

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

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

[12] A. J. Goldstein and R. L. Graham. A Hadamard-type bound on the coefficients of a determinant of polynomials. *SIAM Review*, 16:394–395, 1974.

[13] A. Knight, J. May, M. McAffer, T. Nguyen, and J.-R. Sack. A computational geometry workbench. *ACM Symp. on Computational Geometry*, 6:370, 1990.

[14] Kurt Mehlhorn and Stefan Näher. Algorithm design and software libraries: Recent developments in the leda project. *Algorithms, Software, Architectures, Information Processing 92*, 1:493–505, 1992.

[15] Victor Milenkovic. Robust polygon modeling. *Computer-Aided Design*, to appear, fall 1993. (special issue on "Uncertainties in Geometric Computations).

[16] P.J.de Rezende and W.R. Jacometti. **GeoLab**: an environment for development of algorithms in Computational Geometry. *Canadian Conference on Computational Geometry*, 5:175–180, 1993.

[17] P. Schorn. An object-oriented workbench for experimental geometric computation. *Canadian Conference on Computational Geometry*, 2:172–175, 1990.

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

[19] 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.

[20] K. Sugihara and M. Iri. An approach to the problem of numerical errors in geometric algorithms. *Proceedings, 37th Annual Convention of the Information Processing Society of Japan, Kyoto*, pages 1665–1666, September 12–14, 1988.

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

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

[23] Chee Yap and Thomas Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*. World Scientific Press, 1994. (To appear, 2nd edition).