On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations

by

Douglas M. Priest

Draft date: 9 November 1992.

On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations

by

Douglas M. Priest

Abstract

Floating point arithmetics generally possess many regularity properties in addition to those that are typically used in roundoff error analyses; these properties can be exploited to produce computations that are more accurate and cost effective than many programmers might think possible. Furthermore, many of these properties are quite simple to state and to comprehend, but few programmers seem to be aware of them (or at least willing to rely on them).

This dissertation presents some of these properties and explores their consequences for computability, accuracy, cost, and portability. For example, we consider several algorithms for summing a sequence of numbers and show that under very general hypotheses, we can compute a sum to full working precision at only somewhat greater cost than a simple accumulation, which can often produce a sum with no significant figures at all. This example, as well as others we present, can be generalized further by substituting still more complex algorithms; consequently, examples such as these oblige us to consider more carefully the tradeoffs between cost and accuracy.

At one end of the accuracy spectrum we find one of the least obvious consequences of the properties of floating point arithmetic: the accuracy of a computation consisting of rational arithmetic operations and comparisons need not be limited by the precision of the floating point arithmetic in which it is carried out. Of course, the more accuracy desired, the greater the cost of the computation, and the cost of computing a very accurate result may be quite high; we illustrate this possibility in the

case of polynomial evaluation. At the other end of the spectrum, however, we give an example of a problem for which simply computing a result to a modest guaranteed accuracy costs far less than the contortions required to accommodate inaccurate results. As a consequence of examples such as these, we conclude that programmers and theorists alike must be willing to adopt a more sophisticated view of floating point arithmetic, even if only to consider that more accurate and reliable computations than those presently in common use might be possible based on stronger hypotheses than are customarily assumed.

Contents

Acknowledgements

I am grateful to Prof. Steve Smale for posing the problem of developing a more rigorous foundation for a theory of numerical computation and complexity and in particular providing some of the ideas and examples which helped me to formulate my definition of numerical stability.

I am also deeply indebted to Prof. W. Kahan for helping me to interpret the significance of this work, suggesting many of the problems which led to the results herein, and providing valuable insight, advice, and encouragement.

I also wish to thank Prof. John Canny for reading this dissertation and providing helpful comments and a stimulating environment during my graduate career; similar thanks are due to James Renegar and Michael Shub.

Finally, I wish to thank Lenore Blum for making available the resources of the International Computer Science Institute and for organizing informal student seminars. Thanks also to the following fellow students and former students: Edward Bishop, Richard Gibson, Michael Hirsch, Gregorio Malajovich, Rose St. John.

Chapter 1

Introduction

The purpose of accuracy in a numerical computation is the preservation of relevant mathematical properties of the result. Obviously the properties preserved by a computation depend on the properties of the floating point arithmetic in which the computation is carried out, so if we wish to preserve specific properties of the solution, we may need to rely on specific properties of the arithmetic. Fortunately, we can ensure the accuracy of many computations using properties which, though specific, are not particularly arcane. Many of the properties we require can be stated in simple terms and applied to a wide variety of problems. Of course, since the vast majority of roundoff error analyses assume only a particularly simple, indeed rather weak property, we must wonder whether stronger hypotheses are really necessary, or, to put it bluntly, who cares?

Consider the following problem which arises as a preprocessing step in many ray tracing[1] programs: in order to be able to find as efficiently as possible the intersection of any line in three dimensions with a given triangle, we want to precompute the coefficients of the equation of the plane containing that triangle, scaled so that the nonconstant coefficients form a unit vector normal to the plane; at the same time, we want to discard degenerate triangles (those whose vertices are collinear). We can easily write down equations in terms of the coordinates of the vertices which give a normal vector whose length is proportional to the area of the triangle, so in particular, the vector vanishes if and only if the triangle is degenerate. Hence, we can solve our problem by first computing this unscaled normal vector; then, if the computed vector

[1]Ray tracing is a technique for producing high quality rendered images of three dimensional scenes composed of geometric primitives such as triangles. The basic step in ray tracing is finding the point of intersection of a ray with one of these primitives; for a typical image, this step must be performed many thousands of times.

vanishes, we discard the triangle, and otherwise we can safely divide the vector by its length and subsequently compute the constant coefficient of the plane equation.

The careful numerical programmer will at once recognize the difficulty with our proposed solution: the unscaled normal vector computed by finite precision floating point arithmetic may not vanish when the triangle is truly degenerate, or worse, the computed normal may vanish for a nondegenerate triangle. Of course, the consequence of the former case is that we will simply attempt to compute intersections of lines with a degenerate triangle, and either no intersections will be found or those intersections which occur will likely not produce any unusual artifacts; we will simply have wasted some computing time. The latter case, in contrast, can be disastrous. Of course, since our computation is stable, this case will occur only for long, skinny triangles which we could regard as being so nearly degenerate that they might as well be discarded anyway, but the naïve user who has just spent hours of computer time rendering a scene will be disappointed, to say the least, when he discovers a long, skinny, triangular gap in the middle of his image. Can we convince that user that a stable algorithm is the best we can achieve?

Suppose instead that we assume that the floating point numbers which comprise the vertex coordinates are exact, and we compute the components of the normal vector so accurately that we can guarantee that each computed component is one of the two floating point numbers nearest the exact value that would be given by our formula. Then we can conclude in particular that a computed component is zero if and only if the exact component is also zero (so long as the computation does not incur underflow). Hence we have recovered the crucial property of the computation, that the computed normal vector vanishes if and only if the triangle is truly degenerate, and we need not disappoint the user who happens to have long, skinny triangles in his data.

(The reader may object to our assumption of exact coordinates and argue that if the coordinates are not exact then our computed normal vector need not agree with the true normal vector determined by the exact data, so there is no reason to compute the normal vector so accurately. Alas, that conclusion is *non sequitur* since the remainder of our ray tracing computations will continue to use the same vertex data whether it is exact or not, and we are obliged to treat the data consistently even if it is not exact. Furthermore, in many applications vertex coordinates do turn out to be exact. In any case, the reader is invited to estimate the cost of testing for nearly degenerate triangles, keeping in mind that the program must also "sew up" any holes created by discarding triangles that are not exactly degenerate and be sure that the seams thus produced will be invisible in the final image. The latter constraint requires that vertex coordinates be perturbed so that certain vertices lie exactly on line segments determined by other vertices, and this task demands precisely the kind of careful computation the avoidance of which is the only reason to prefer the more pessimistic degeneracy test in the first place.)

Although our proposed solution is quite simple, many programmers might not write a program which solves the problem in this way simply because they are unaware of or unwilling to rely on the properties of floating point arithmetic which are necessary to guarantee the desired accuracy. Their programs must either be far more expensive than ours or occasionally produce images with holes. Although a hole would be noticed only rarely, and the programmer might well convince the user that the data "deserved" it, such excuses hardly seem justified considering the ease with which a provably correct program can be written instead.

In this example, as in most numerical computations, the only justifiable reason for sacrificing accuracy is to improve speed; yet, the examples in subsequent chapters show that the extra time required to perform a computation carefully is often quite modest considering the benefit that guaranteed accuracy confers. Since we usually need only assume several simple, reasonable properties in order to achieve that accuracy, we are now obliged to consider more carefully the tradeoffs between cost and accuracy, so that programmers can better predict when the extra accuracy will be worth the cost of obtaining it.

In chapter 2 we present some of the properties upon which provably accurate floating point computations may be based and consider their applicability to different floating point arithmetics and their consequences for computability and cost. Chapter 3 defines a formal model of floating point computation based on the properties described in chapter 2 and shows that any problem which can be solved exactly in a certain model of computation in exact arithmetic can be solved to within any desired accuracy in our model of computation in floating point arithmetic—in other words, floating point roundoff error need not limit what we can compute but only affects the cost of computing it. We also define a notion of numerical stability and reinterpret our result as saying that for a large class of problems, we can mechanically construct numerically stable algorithms. Chapter 4 presents further examples of applications of the properties of chapter 2. Finally, chapter 5 draws some conclusions concerning the question raised above: "Who cares?"

Chapter 2

Properties of Floating Point Arithmetic and their Consequences for Computability and Accuracy

Although the theoretical capabilities and limitations of numerical computation obviously depend on the set of properties that floating point arithmetic is assumed to possess, the exact nature of that dependence is far from obvious. Nearly all well known theoretical results on numerical computation are based solely on a simple property we refer to as the "$(1 + \epsilon)$" property, which first appeared in the earliest floating point roundoff error analyses of Givens, Wilkinson, and their colleagues (see the comments in [64]). Considering the weakness of the $(1 + \epsilon)$ property, these error analyses achieve remarkably strong results by relying on the elegant technique of backward error analysis, and the error estimates thus obtained are still widely accepted as the best possible for floating point arithmetic. Indeed, the fact that so much theory can be based on so few hypotheses has led many numerical analysts to suppose that nothing more than the $(1+\epsilon)$ property need be assumed about floating point arithmetic in order to deduce the behavior of every worthwhile computation. Although Wilkinson himself never suggested that every stable computation should be provable from that property alone, a reader might be tempted to infer that idea from the observation that "it seems highly improbable that any method of solving any problem which is restricted to the use of t-digit arithmetic will, in general, do better than give the solution to a t-digit approximation to the original problem." [64, p. 32]

Nevertheless, there are useful computations which in fact do much better than give the solution to an approximation to the original problem. We will give a number of examples of such computations below and in subsequent chapters, and these examples will demonstrate that solving problems to greater accuracy is not only possible but often efficient enough to be desirable. Of course, the correctness of these computations cannot be proved using only the $(1+\epsilon)$ property, so we are led to question what other properties we must assume, how widely applicable those properties are, and when simpler properties will not suffice.

In this chapter, we consider a number of different properties possessed by various floating point arithmetics and show that subtle changes in our hypotheses can lead to striking differences in the accuracy we can achieve, the cost of that accuracy, and even the class of problems we can hope to solve at all. In section 1, we define floating point arithmetics and the $(1+\epsilon)$ property and give an example to demonstrate the artificial limits we impose by restricting our assumptions to this property alone. In section 2, we list several properties of floating point subtraction which are shared by nearly all arithmetics and show that, despite their generality, these properties can be used to prove the correctness of a simple computation. Section 3 introduces the notion of "faithfulness" and presents a remarkably useful algorithm which depends crucially upon the strength of this property; we also consider several simplifications of the algorithm which demonstrate that cost as well as computability can depend on the hypotheses assumed. Section 4 describes properties possessed by certain arithmetics which are not faithful and considers generalizations of the results of section 3. Finally, in section 5, we survey a number of proposed axiom systems for floating point arithmetic and discuss their applicability in theory as well as in practice.

2.1 Floating Point Arithmetic and the "$(1+\epsilon)$" Property

We begin by recalling the simplest hypothesis one can require of a floating point arithmetic. In order to motivate this hypothesis and establish notation which will help to clarify the other properties discussed below, we first give a general definition of floating point arithmetic.

DEFINITION: For positive integers t and $\beta > 1$, the *t-digit, radix β floating point numbers* are those numbers of the form $x = m\beta^k$, where m and k are integers and

$|m| < \beta^t$. Note that for nonzero numbers, we may standardize the choice of m and k by requiring $\beta^{t-1} \le |m|$; with this choice, we call β^k a *unit in the last place* of x, denoted $\mathrm{ulp}(x)$. (For simplicity, we define $\mathrm{ulp}(0) = 0$.) A *floating point arithmetic* is a mapping which assigns to each pair of floating point numbers a and b and each $\circ \in \{+, -, \times, /\}$ another floating point number, denoted $\mathrm{fl}(a \circ b)$, provided $b \neq 0$ when $\circ = /$. (As a rule, we ignore the limitations imposed by underflow and overflow, which effectively bound the range of exponents k; hence, unless otherwise stated, the phrase "provided no underflow or overflow occurs" should be understood to apply to all algorithms and results.)

Given that the result of any computation must be a floating point number as defined above, we might ask how accurate we may reasonably expect a single floating point operation to be. Suppose ξ is the exact result of an arithmetic operation and x is the result of the same operation computed in floating point arithmetic; then clearly ξ can differ from x by as much as half a unit in the last place of x in the worst case even if the arithmetic always produces the floating point number nearest the exact result. Moreover, if the arithmetic is somewhat less careful and is only guaranteed to return one of the two numbers nearest the exact result, then we may find that ξ differs from x by nearly a unit in the last place of x in the worst case. Consequently, the relative error $|\xi - x|/|\xi|$ might be nearly as great as $\mathrm{ulp}(x)/|x|$, which in turn can be as large as β^{1-t}. This observation prompts us to formulate the following definition.

DEFINITION: A t-digit, radix β floating point arithmetic possesses the $(1+\epsilon)$ *property* if, for floating point numbers a and b and $\circ \in \{+, -, \times, /\}$ with $b \neq 0$ if $\circ = /$, there exists ϵ such that $\mathrm{fl}(a \circ b) = (a \circ b)(1+\epsilon)$ and $|\epsilon| < \beta^{1-t}$.

Despite its weakness, the $(1+\epsilon)$ property has proved attractive to error analysts thanks to its simple form and to the surprising power of the method of backward error analysis as illustrated in the famous work of Wilkinson [64]. As a result, the $(1+\epsilon)$ property has formed the basis for nearly every numerical error analysis known to date. So compelling are the arguments and results obtained from such analyses, in fact, that many numerical programmers have concluded that nothing more than this property need be known about floating point arithmetic in order to establish the correctness and accuracy of numerical computations.

Nevertheless, a number of indispensable programs exist whose accuracy cannot be proved from the $(1+\epsilon)$ property alone. One simple example, first analyzed by Kahan and Parlett [39], is a program which evaluates the function

$$f_N(x) := \begin{cases} \frac{x^N-1}{x-1} & \text{if } x \neq 1 \\ N & \text{otherwise} \end{cases}$$

for a positive integer N and real x. This function arises in financial computations in which N represents a number of interest compounding periods and $x = 1 + i$ where i is the interest rate. Suppose we compute $f_N(x)$ in the obvious manner suggested by the formula above, forming the N-th power by multiplication. Using only the $(1+\epsilon)$ property, the most we can say about the computed value of x^N is that it can be written $x^N(1 + N\epsilon)$ for some $|\epsilon| < \beta^{1-t}$. Consequently, the computed value of $f_N(x)$ must differ by at most a few units in the last place from the quantity $f_N(x)+Nx^N\epsilon/(x-1)$. When x is very close to one, the latter term can become nearly as large as the former in absolute value, so the computed value can apparently lose all significance. On the other hand, by using the definition of floating point numbers above together with some of the properties defined in subsequent sections, we can prove that the computation of $f_N(x)$ actually produces a computed value correct to within a relative error of approximately $\sqrt{2N\beta^{2-t}}$, so that on most computers the result will be correct in at least roughly half of the figures carried. (See appendix B for the proof.)

If we are only allowed to assume the $(1 + \epsilon)$ property, the best way to evaluate f_N and retain useful accuracy is to write the function as $f_N(x) = \sum_{j=0}^{N-1} x^j$ and use synthetic division. For positive x, this method can be proved to produce a result with a relative error proportional to $N^2\beta^{1-t}$, which will generally be at least as good as the error bound for the method described above. On the other hand, the latter method requires $2N-2$ arithmetic operations compared to $N+2$ operations for the former computation. Hence, by restricting ourselves to the $(1+\epsilon)$ hypothesis, we must at least double the cost of computing $f_N(x)$ accurately. This extra cost is not necessitated by the computation itself, but is merely an artifact of the limitations of the model of roundoff. In order to fully exploit both the accuracy and efficiency possible in numerical computation, then, we must be willing to adopt stronger hypotheses than the $(1+\epsilon)$ property on the accuracy of floating point arithmetic.

2.2 Examples of Stronger Properties: Conditions for Exact Subtraction

As examples of useful hypotheses stronger than the $(1+\epsilon)$ property, we state three properties which give conditions under which the difference of two floating point numbers is computed exactly, and we illustrate the application of these properties by proving a conjecture due to Kahan. Since the hypotheses and conjecture claim that certain results of floating point computations may be exact, these statements clearly could not be deduced using only the $(1+\epsilon)$ property.

The properties we shall use are both strong and widely applicable. To wit, the following two properties hold for the arithmetics of every commercially significant machine in use in North America today. (Throughout this section, lower case letters denote floating point numbers.)

Property S1: If $0 \le a \le b$ and $\mathrm{fl}(b-a) = b-a$ exactly, then $\mathrm{fl}(c-a) = c-a$ exactly for all $c \in [a,b]$ (and similarly if $0 \ge a \ge b$).

Property S2: If $0 \le a \le b$ and $c = \mathrm{fl}(b-a)$, then $\mathrm{fl}(b-c) = b-c$ exactly (and similarly if $0 \ge a \ge b$).

In contrast, the next property is satisfied by all arithmetics known to the author except that of the Cray 2 (see section 4).

Property S3: If $0 \le \mathrm{ulp}(b)/2 \le a \le b$, and $c = \mathrm{fl}(b - \mathrm{fl}(b-a))$ satisfies $c > a$, then $\mathrm{fl}(c-d) = c-d$ exactly for all $d \in [a,c]$ (and similarly if $0 \ge -\mathrm{ulp}(b)/2 \ge a \ge b$).

In addition to the preceding properties, we require two simple hypotheses which are embodied in the following definitions.

Definition: Floating point subtraction is *anti-symmetric* if $\mathrm{fl}(a-b) = -\mathrm{fl}(b-a)$ for all a and b; it is *monotonic* if $a \le b$ implies $\mathrm{fl}(a-c) \le \mathrm{fl}(b-c)$ for all c.

Note that all arithmetics known to the author have monotonic subtraction. Furthermore, subtraction can fail to be anti-symmetric in IEEE standard arithmetic only when a rounding mode other than the default is used (see [30, 31]) or in arithmetics such as those of the GE/Honeywell 600 series or the HP 2000 series which round a result halfway between two representable numbers to the next larger number, regardless of its sign. (Strictly speaking, in arithmetics with signed zeroes, the *computed* value $\mathrm{fl}(a-a)$ can differ from the *computed* value $\mathrm{fl}(-\mathrm{fl}(a-a))$, but the numeric value x of the result of $\mathrm{fl}(a-a)$ still satisfies $x = -x$, and this latter property suffices to satisfy anti-symmetry.)

To illustrate a practical application of the preceding properties, we consider the problem of shifting an interval: given floating point numbers $0 \le x \le y$, we wish to find a number s, preferably positive, so that $\mathrm{fl}(w-s) = w-s$ exactly for all $w \in [x, y]$. This problem arises, for example, in the reduction of arguments for the evaluation of certain elementary functions. Kahan [38] conjectured that the following algorithm provides a portable way to compute s, and we will show that the preceding properties may be used to prove his conjecture for all commercially significant machines except the Cray 2; as we will see in section 4, the failure of property S3 for Cray 2 arithmetic leads to a counterexample for that machine.

Proposition: Given positive floating point numbers x, y, and z such that $x > z$, $y > z$, $x \ge \mathrm{ulp}(y)$, and $y \ge \mathrm{ulp}(x)$. If the floating point subtraction is anti-symmetric and monotonic and satisfies properties S1–S3, then the following algorithm computes a number s such that $\mathrm{fl}(w-s) = w-s$ exactly for all w between x and y, inclusive.

Algorithm: (Compute shift for the interval $[x, y]$)

```
procedure shift( x, y, z )
begin
t := fl(y − fl(y − z))
s := fl(x − fl(x − t))
```

return s
end

PROOF: We consider two cases. Suppose first that $y \leq x$. By monotonicity and anti-symmetry, we have $0 \leq t \leq y$ and hence $0 \leq s \leq x$. By property S2, then, $\mathrm{fl}(x - s) = x - s$ exactly, and hence by property S1, $\mathrm{fl}(w - s) = w - s$ exactly for all $w \in [s, x]$. If $s \leq y$, then the proposition is proved for this case, so assume instead that $s > y$. By monotonicity, we have $\mathrm{fl}(x - y) \leq \mathrm{fl}(x - t)$ and hence $x - \mathrm{fl}(x - y) \geq x - \mathrm{fl}(x - t)$. But by property S2, $x - \mathrm{fl}(x - t) = \mathrm{fl}(x - \mathrm{fl}(x - t)) = s$, so since $s > y$ we have $x - \mathrm{fl}(x - y) > y$. By hypothesis, $y \geq \mathrm{ulp}(x)$, so by properties S2 and S3, $\mathrm{fl}((x - \mathrm{fl}(x - y)) - w) = (x - \mathrm{fl}(x - y)) - w$ exactly for all $w \in [y, x - \mathrm{fl}(x - y)]$, and since $s \leq x - \mathrm{fl}(x - y)$, we may certainly restrict the last conclusion to $w \in [y, s]$. Then we may apply property S1 for each such w to conclude that $\mathrm{fl}(c - w) = c - w$ exactly for all $c \in [w, x - \mathrm{fl}(x - y)]$, and in particular, $\mathrm{fl}(s - w) = s - w$ exactly. By anti-symmetry, then, the proposition holds.

Now consider the case $x < y$. Again we have $0 \leq t \leq y$, and by property S2, $\mathrm{fl}(y - t) = y - t$ exactly. Suppose $t \leq x$. Then $0 \leq s \leq x < y$, and hence by property S1, $\mathrm{fl}(x - t) = x - t$ exactly, so again using property S2, we have $s = \mathrm{fl}(x - \mathrm{fl}(x - t)) = x - \mathrm{fl}(x - t) = x - (x - t) = t$, and thus by property S1, $\mathrm{fl}(w - s) = \mathrm{fl}(w - t) = w - t$ exactly for all $w \in [t, y]$; restricting this conclusion to $w \in [x, y]$ proves the proposition. Now suppose instead that $t > x$. By monotonicity and property S2, $y - \mathrm{fl}(y - x) \geq y - \mathrm{fl}(y - z) = t > x$, and by hypothesis, $x \geq \mathrm{ulp}(y)$, so by property S3, $\mathrm{fl}((y - \mathrm{fl}(y - x)) - x) = (y - \mathrm{fl}(y - x)) - x$ exactly; hence by property S1, $\mathrm{fl}(t - x) = t - x$ exactly, so by property S2 and anti-symmetry, we again have $s = \mathrm{fl}(x - \mathrm{fl}(x - t)) = x - (x - t) = t$. Now property S1 guarantees that $\mathrm{fl}(w - s) = \mathrm{fl}(w - t) = w - t$ exactly for all $w \in [t, y]$, while property S3 implies that $\mathrm{fl}((y - \mathrm{fl}(y - x)) - w) = (y - \mathrm{fl}(y - x)) - w$ exactly for all $w \in [x, y - \mathrm{fl}(y - x)]$, which we can restrict to $w \in [x, t]$. Applying property S1 for each such w, we find that $\mathrm{fl}(c - w) = c - w$ exactly for all $c \in [w, y - \mathrm{fl}(y - x)]$; in particular, taking $c = t$ proves the proposition. ∎

As the preceding proposition shows, properties S1–S3 determine a stronger model of floating point computation than the $(1+\epsilon)$ property alone, but since they are concerned only with addition and subtraction, they are obviously not sufficient to deduce all that we might wish to know about the accuracy and cost of floating point computations. Consequently, we must seek further properties upon which to base accurate and efficient numerical computations.

2.3 Faithfulness and Further Properties

We now consider a general property known as "faithfulness" which at first sight appears only slightly stronger than the $(1+\epsilon)$ property but is in fact much more powerful. This property is very nearly all that we require to prove the correctness of many useful computations; in particular, we will derive a result which forms the basis of a technique for simulating computation to very high precision using only fixed precision floating point arithmetic operations. Of course, since we are concerned with cost as well as computability, we will also propose additional properties which allow us to simplify our algorithms considerably. We begin with the following definition.

DEFINITION: For floating point numbers a and b and $\circ \in \{+, -, \times, /\}$, let $c = a \circ b$ exactly (assuming $b \neq 0$ if $\circ = /$). Let x and y be consecutive floating point numbers with the same sign as c such that $|x| \leq |c| < |y|$. Then the floating point arithmetic is called *faithful* if $\mathrm{fl}(a \circ b) = x$ whenever $c = x$ and $\mathrm{fl}(a \circ b)$ is either x or y whenever $c \neq x$.

The preceding definition simply asserts that the computed result of an arithmetic operation is one of the two floating point numbers nearest the exact result, which is the assumption that motivated the $(1+\epsilon)$ property in the first place. Nevertheless, faithfulness is a much stronger property, as we will demonstrate in the following results. We begin with two simple lemmas.

LEMMA 1: (Sterbenz [61]) If a and b are t-digit, radix β floating point numbers such that $1/2 \leq a/b \leq 2$, then $a - b$ is also a t-digit, radix β floating point number.

In particular, in a faithful arithmetic, $\mathrm{fl}(a-b) = a-b$ exactly.

PROOF: For the first statement, assume without loss that a and b are floating point numbers with $a > b > 0$. Then $\mathrm{ulp}(a) \geq \mathrm{ulp}(b)$, so a and b are both multiples of $\mathrm{ulp}(b)$, as is $a-b$. By hypothesis, $a \leq 2b$ so $a-b \leq b$; hence $a-b$ is a multiple of $\mathrm{ulp}(b)$ which is no larger than b, and thus $a-b$ is a t-digit, radix β floating point number. The second statement follows trivially by the definition of faithfulness. ∎

LEMMA 2: Every faithful arithmetic with $t > 1$ satisfies properties S1–S3 above.

PROOF: Throughout the proof, a and b denote floating point numbers such that $0 \leq a \leq b$. For property S1, assume $\mathrm{fl}(b-a) = b-a$ exactly. By lemma 1, $\mathrm{fl}(c-a) = c-a$ exactly for all floating point numbers $c \in [a, 2a]$, so we may assume without loss that $b > 2a$, and we need only show that $\mathrm{fl}(c-a) = c-a$ exactly for $c \in (2a, b]$. To this end, let b' denote the largest t-digit floating point number strictly less than b; we claim that $\mathrm{fl}(b'-a) = b'-a$ exactly. To see this, first note that $b \neq 0$, so we may write $b = m\beta^k$ with $\beta^{t-1} \leq m < \beta^t$. By hypothesis, $\mathrm{fl}(b-a) = b-a$ exactly, hence $b-a$ is a t-digit number, evidently positive, and we may write $b-a = n\beta^j$ with $\beta^{t-1} \leq n < \beta^t$. As $a < b/2$, we have $b-a > b/2$, and since $\beta \geq 2$, we must have $j = k$ or $j = k-1$. Now note that $b = b' + \mathrm{ulp}(b')$; hence $b'-a = b-a-\mathrm{ulp}(b') = n\beta^j - \mathrm{ulp}(b')$. If $\mathrm{ulp}(b') = \mathrm{ulp}(b) = \beta^k$, then clearly $b'-a$ is a t-digit number. The only other possibility is that $\mathrm{ulp}(b') = \beta^{k-1}$, but this can happen only if $b = \beta^{k+t-1}$, in which case we must have $j = k-1$, and again $b'-a$ is a t-digit number. In either case, then, by faithfulness we must have $\mathrm{fl}(b'-a) = b'-a$ exactly, which establishes the claim. Now repeating the preceding argument with b replaced by b', we conclude inductively that $\mathrm{fl}(c-a) = c-a$ exactly for all $c \in (2a, b]$, and property S1 holds.

For property S2, let $c = \mathrm{fl}(b-a)$. If $a \geq b/2$ then by lemma 1, $c = b-a$ exactly; therefore $b-c = a$ is a t-digit number, so by faithfulness, $\mathrm{fl}(b-c) = b-c$ exactly. Suppose instead $a < b/2$. Let a' denote the largest t-digit number not greater than $b/2$ and note that $b-a'$ is the smallest t-digit number not less than $b/2$. Evidently

$a \leq a'$, so by faithfulness, $c \geq b - a' \geq b/2$; hence by lemma 1, $\mathrm{fl}(b-c) = b-c$ exactly, and property S2 holds.

For property S3, assume $a \geq \mathrm{ulp}(b)/2$ and let $e = \mathrm{fl}(b-a)$ and $c = \mathrm{fl}(b-e)$; note that $c = b - e$ exactly by property S2. Suppose $c > a$; then evidently $e < b - a$. By faithfulness, then, e must be the greatest t-digit number less than $b - a$. Now if $\mathrm{ulp}(b)/2 \leq a < \mathrm{ulp}(b)$, then since $b - \mathrm{ulp}(b)$ is also a t-digit number, we must have $e \geq b - \mathrm{ulp}(b)$, so clearly $c \leq \mathrm{ulp}(b) \leq 2a$. If instead $a \geq \mathrm{ulp}(b)$, then note that $e + \mathrm{ulp}(e)$ is the smallest t-digit number greater than e, so we must have $b - a \leq e + \mathrm{ulp}(e) \leq e + \mathrm{ulp}(b)$ and hence $e \geq b - a - \mathrm{ulp}(b)$ and $c \leq a + \mathrm{ulp}(b) \leq 2a$. In either case, we now have $a < c \leq 2a$ and thus $d \leq c \leq 2d$ for all $d \in [a, c]$; hence by lemma 1, property S3 holds. ∎

The following proposition is perhaps the most striking example of the strength of faithfulness as compared with the $(1+\epsilon)$ property: it states that we can compute exactly the roundoff error incurred in the floating point addition of two numbers. As observed by Møller [51], Dekker [16], Linnainmaa [44], and others, this technique can be used to effectively extend the precision of a floating point computation, in some cases to the point of computing an exact representation of the result (see appendix A). We first prove the most general form of the proposition; subsequently, we show that with a few additional hypotheses, we can reduce the cost of the algorithm by nearly half, thereby obtaining the forms proved correct for the various arithmetics considered in [16, 33, 40, 43, 51, 54].

Proposition: Let a and b be floating point numbers. If the floating point arithmetic is faithful, then the numbers c and d computed by the following algorithm satisfy $c + d = a + b$ and either $c = d = 0$ or $|d| < \mathrm{ulp}(c)$.

Algorithm: (Sum and roundoff error)

```
procedure sum_and_error( a, b )
begin
if |a| < |b|
    swap( a, b )
c := fl(a + b), e := fl(c − a)
```

$g := \mathrm{fl}(c-e)$, $h := \mathrm{fl}(g-a)$, $f := \mathrm{fl}(b-h)$
$d := \mathrm{fl}(f-e)$
if $\mathrm{fl}(d+e) \neq f$
    $c := a$, $d := b$
return c, d
end

PROOF: Without loss, assume $|a| \geq |b|$. If $b = 0$ then clearly the algorithm produces $c = a$, $d = 0$, so the proposition holds. For the general case, assume $a > 0$; the proof when $a < 0$ is essentially identical. Let $c = \mathrm{fl}(a+b)$ as above and define $r = a+b-c$, so r is the error in computing c. We prove the proposition by establishing three claims.

Claim 1: $d = r$ if and only if r is a t-digit floating point number. Of course, if r is not a t-digit number, then we cannot have $d = r$. For the converse, suppose first that $-a \leq b \leq -a/2$. Then $1 \leq a/(-b) \leq 2$ so $a + b = a - (-b)$ is a t-digit number; hence, by faithfulness, $c = \mathrm{fl}(a+b) = a+b$ exactly and $r = 0$. Also by faithfulness, we must have $e = \mathrm{fl}(c-a) = b$, $g = a$, $h = 0$, $f = b$, and $d = 0 = r$, a t-digit number.

Now suppose $-a/2 < b < 0$. Let a' denote the largest t-digit number not greater than $a/2$, and note that $a - a'$ is the smallest t-digit number not less than $a/2$. Since $|b| \leq a'$, we must have $a + b \geq a - a'$, so by faithfulness, $c = \mathrm{fl}(a+b) \geq a - a' \geq a/2$. Then $1/2 \leq c/a \leq 1$, so $e = \mathrm{fl}(c-a) = c - a = b - r$ exactly. Again by faithfulness we have $g = a$, $h = 0$, and $f = b$, so if r is a t-digit number then $d = \mathrm{fl}(f-e) = r$.

Finally, suppose $b > 0$. If $c \leq 2a$ then $1 \leq c/a \leq 2$ so $e = \mathrm{fl}(c-a) = c - a = b - r$ exactly. As before, $g = a$, $h = 0$, $f = b$, and thus $d = r$ if r is a t-digit number. If instead $c > 2a$, then $e = \mathrm{fl}(c-a)$ may not be computed exactly; let $s = c - a - e$, so s is the roundoff error incurred in computing e. Since the arithmetic is faithful, by lemma 2 it also satisfies the properties of the previous section; using property S2, we see that $g = \mathrm{fl}(c-e) = \mathrm{fl}(c - \mathrm{fl}(c-a))$ is computed exactly, so $g = c - e = a + s$.

Now as $b \leq a$ we have $\mathrm{ulp}(b) \leq \mathrm{ulp}(a)$, but we can only have $c > 2a$ if $\mathrm{ulp}(b) = \mathrm{ulp}(a)$, so all computed quantities must be integer multiples of $\mathrm{ulp}(a)$. Since a sum can carry by at most one digit, we have $\mathrm{ulp}(c) \leq \beta\mathrm{ulp}(a)$. By faithfulness, then, $|r| <$

$\mathrm{ulp}(c)$, so r must be a multiple of $\mathrm{ulp}(a)$ satisfying $|r| < \beta\mathrm{ulp}(a)$; in particular, r is a t-digit number. (In fact, r is a one-digit multiple of $\mathrm{ulp}(a)$.) Evidently $|s| < \beta\mathrm{ulp}(a)$ as well, hence $h = \mathrm{fl}(g - a) = s$ exactly, and clearly $f = \mathrm{fl}(b - h) = b - s$ is also exact. Thus $d = \mathrm{fl}(f - e) = r$, and the claim holds.

Claim 2: $\mathrm{fl}(d + e)$ is always computed exactly. Recall that $\mathrm{fl}(d + e) = f$ exactly whenever r is a t-digit number. Suppose r is not a t-digit number. Then we must have $e = \mathrm{fl}(c - a) = c - a = b - r$, $g = a$, $h = 0$ and $f = b$, so $d = \mathrm{fl}(b - e)$. Now a, b, and c are all integer multiples of $\mathrm{ulp}(b)$, so r is also an integer multiple of $\mathrm{ulp}(b)$. If $br \geq 0$ then clearly $|r| \leq |b|$, so r would be a t-digit number; therefore suppose $br < 0$ and also $|r| > |b|$. Then $de < 0$ and $1/2 \leq d/(-e) \leq 1$ so $d + e = d - (-e)$ is a t-digit number, and consequently $\mathrm{fl}(d + e) = d + e$ exactly (although we need not have $d + e = f$).

Claim 3: If r is not a t-digit number then $|b| < \mathrm{ulp}(a)$. To see this, assume r is not a t-digit number, and suppose $|b| \geq \mathrm{ulp}(a)$. Recall that $d \neq r$ only if $|b| < |r|$. By faithfulness, $|r| < \mathrm{ulp}(c)$, so we must have $\mathrm{ulp}(a) < \mathrm{ulp}(c)$, but since $|b| \leq |a|$ this can happen only if $\mathrm{ulp}(c) = \beta\mathrm{ulp}(a)$. Now a, b, and c are all integer multiples of $\mathrm{ulp}(b) \geq \beta^{1-t}\mathrm{ulp}(a) = \beta^{-t}\mathrm{ulp}(c)$, so r is an integer multiple of $\beta^{-t}\mathrm{ulp}(c)$ and $|r| < \mathrm{ulp}(c)$. Thus r is a t-digit number, a contradiction.

From the preceding claims, the proof of the proposition proceeds as follows. If r is a t-digit number then $d = r$ and the test in line 8 must fail because $\mathrm{fl}(d + e) = f$ exactly; in this case, the algorithm outputs the computed sum c and the roundoff error d which, by faithfulness, must satisfy $|d| < \mathrm{ulp}(c)$. Otherwise, the test in line 8 must succeed because $\mathrm{fl}(d + e)$ is computed exactly and $d + e \neq f$, so in this case the algorithm simply returns a and b, which already satisfy $|b| < \mathrm{ulp}(a)$. ∎

Although the preceding algorithm clearly demonstrates the power of faithfulness, it can be improved upon significantly due to the fact that faithful floating point arithmetics generally possess still further properties. We state two of these properties, then introduce several definitions which will help to classify those arithmetics which satisfy the properties.

PROPERTY A1: The roundoff error of a floating point sum is itself a floating point number; that is, for all a and b, $a+b-\mathrm{fl}(a+b)$ is a t-digit number.

PROPERTY A2: If $|b| \le |a|$ then $|\mathrm{fl}(a+b)| \le 2|a|$.

DEFINITION: For floating point numbers a and b and $\circ \in \{+,-,\times,/\}$, let $c = a \circ b$ exactly (assuming $b \ne 0$ if $\circ = /$). Let x and y be consecutive floating point numbers with the same sign as c such that $|x| \le |c| < |y|$. Then the floating point arithmetic is *correctly rounding* if it is faithful and also $\mathrm{fl}(a \circ b) = x$ whenever $|c-x| < |c-y|$ and $\mathrm{fl}(a \circ b) = y$ whenever $|c-y| < |c-x|$. In the case $\circ = +$, the arithmetic's addition is called *properly truncating* if $\mathrm{fl}(a+b) = x$ whenever $c = x$ or $ab > 0$, but $\mathrm{fl}(a+b) = y$ whenever $c \ne x$ and $ab < 0$; the addition is *correctly chopping* if $\mathrm{fl}(a+b) = x$ always.

Note that all correctly rounding arithmetics satisfy property A1, as do those with properly truncating addition. All faithful binary arithmetics and all arithmetics with either properly truncating or correctly chopping addition satisfy property A2. In particular, binary correctly rounding arithmetics, such as those which conform to the IEEE 754 standard [30] as well as DEC VAX arithmetics, satisfy both properties. For other reasons, IBM 370 arithmetic also satisfies both properties. (We note that property A1 can fail in IEEE 754 arithmetics if the rounding mode is not the default round-to-nearest: for example, if the rounding mode calls for results to be rounded down or towards zero, then subtracting a very small positive number from a very large positive number yields a roundoff which is not representable. Property A1 also fails if the results of arithmetic operations are first rounded to a higher precision than that used to store numbers, and subsequently rounded to storage precision; see section 5.2 for an example.)

We can use the preceding properties to simplify the sum-and-roundoff algorithm above, thereby obtaining as special cases several of the algorithms described in the references. For example, the following simplification was established by Knuth [40] for correctly rounding arithmetic; here we prove a slightly more general result which includes properly truncating arithmetics. (Møller [51] obtained a similar, weaker result using still different hypotheses.)

COROLLARY 1: If the arithmetic satisfies property A1, then lines 8 and 9 may be eliminated from the sum-and-error algorithm.

PROOF: Returning to the proof of the proposition, we are assuming that r is always a t-digit number; hence the test in line 8 never succeeds. ∎

Dekker [16] and Linnainmaa [43] consider a different variation, which also appears in Pichat [54]. Again, we state a slightly more general result which follows easily from the preceding proposition.

COROLLARY 2: If the arithmetic satisfies property A2, then line 6 may be eliminated and b substituted for f in lines 7 and 8 of the above algorithm. In particular, if the arithmetic satisfies both properties, then lines 6, 8, and 9 may all be eliminated and b substituted for f in line 7.

PROOF: For the first statement, note that the hypothesis implies $|c| \leq 2|a|$, so we always have $g = a$, $h = 0$, and $f = b$. The second statement follows by combining the first with the previous corollary. ∎

The simplest algorithm described in corollary 2 requires a comparison of absolute values followed by three floating point additions to compute a sum and its roundoff error. Knuth [40] proved that an alternative form of the algorithm, requiring six additions and no branching, computes a sum and its roundoff in correctly rounding binary arithmetics; using the preceding properties, we can show that Knuth's algorithm also works for the other arithmetics to which the simplifications of corollary 2 apply.

PROPOSITION: Let a and b be floating point numbers. If the floating point arithmetic is faithful and satisfies properties A1 and A2, then the numbers c and d computed by the following algorithm satisfy $c + d = a + b$ and either $c = d = 0$ or $|d| < \mathrm{ulp}(c)$.

ALGORITHM: (Sum and roundoff error, branchless)

```
procedure sum_and_error( a, b )
begin
```

$c := \mathrm{fl}(a+b)$, $e := \mathrm{fl}(c-a)$
$f := \mathrm{fl}(b-e)$
$g := \mathrm{fl}(c-e)$, $h := \mathrm{fl}(g-a)$
$d := \mathrm{fl}(f-h)$
return c, d
end

PROOF: It suffices to prove that f, g, h, and d are all computed exactly, since then $d = a+b-\mathrm{fl}(a+b)$ is precisely the roundoff error in the sum $c = \mathrm{fl}(a+b)$, from which the proposition follows. Moreover, since we are assuming that the roundoff of a sum is itself a floating point number (property A1), it suffices to show g, h, and f are computed exactly, since then d must be computed exactly by faithfulness. By the same reasoning, if g is computed exactly, then $g-a = c-a-\mathrm{fl}(c-a)$ is again the roundoff of a sum, so h must be computed exactly. Consequently, we need only show that g and f are computed exactly. We consider several cases.

In the case $|a| \geq |b|$, recall that the proof of the original proposition showed that $e = c-a$ exactly (because property A2 precludes the possibility that $c > 2a$). In this case, then, $g = a$ exactly and f must be computed exactly, since $b-e = a+b-\mathrm{fl}(a+b)$ is itself the roundoff error.

Now assume $|a| < |b|$. Assume also $b > 0$; the proof when $b < 0$ is essentially identical. If $b/2 \leq -a < b$, then by lemma 1, $a+b = b-(-a)$ is a floating point number, so by faithfulness, $c = \mathrm{fl}(a+b) = a+b$ exactly and thus $e = b$ which implies $f = 0$ and $g = a$ exactly. If instead $0 < -a < b/2$, we have the inequalities

$$b/2 \leq c \leq e \leq 2c \leq 2b$$

of which $b/2 \leq c$ follows from arguments in the proof of the original proposition, $e \leq 2c$ follows from property A2 and the fact that $-a < b/2 \leq c$, and the rest are obvious. Now using lemma 1, we have $g = \mathrm{fl}(c-e) = c-e$ exactly from the innermost inequalities and $f = \mathrm{fl}(b-e) = b-e$ exactly from the outermost ones.

Finally, suppose $a > 0$. Recall that by lemma 2, we may assume properties S1–S3, and note that as $c \geq a$ we have $g = \text{fl}(c - e) = \text{fl}(c - \text{fl}(c - a))$ computed exactly by property S2. Moreover, note that $e \leq c$ and by property A2, $c \leq 2b$, so if $e \geq b$ then $f = \text{fl}(b - e) = b - e$ exactly by lemma 1, whereas if $e < b$ then since $b \leq c$ and $g = \text{fl}(c - e)$ is computed exactly, we have $f = \text{fl}(b - e) = b - e$ exactly by property S1. So in either case, f is computed exactly, and the proof is finished. ∎

The preceding results together with the algorithms derived from them in appendix A provide overwhelming evidence of the vast difference in computability, accuracy, and cost that we can achieve using faithfulness and a few other properties above and beyond the $(1 + \epsilon)$ property. Since these properties are satisfied by many of the machines commonly used for extensive numerical computations, we suggest that a model of floating point computation which does not assume them must be unnecessarily weak.

2.4 Properties of Unfaithful Arithmetics

Despite the obvious advantages of faithfulness, a number of floating point arithmetics still lack that property, most notably Cray supercomputers. We consider the extent to which faithfulness is necessary to achieve the results obtained above by examining addition and subtraction as performed by the Cray X-MP and Y-MP; in particular, we show that these machines satisfy properties S1–S3 of section 2, and that with a more complicated algorithm, we can generalize the sum-and-roundoff computation of the previous section. We note, however, that the Cray 2 performs subtraction in a slightly different way which, though it often yields somewhat more accurate answers than the X/Y-MP, nevertheless leads to counterexamples to our results for that machine.

The following definition describes the way in which addition and subtraction are performed on the Cray X/Y-MP.

DEFINITION: A floating point arithmetic adds and subtracts by *truncation without guard digit* if it possesses the following three properties:

(1) for all $0 \leq a \leq b$, $\mathrm{fl}(b-a) = b - a + (a \bmod \mathrm{ulp}(b))$,

(2) for all $a, b \geq 0$, if $c = \mathrm{fl}(a+b)$ then $c = a + b - ((a+b) \bmod \mathrm{ulp}(c))$, and

(3) for all a and b, $\mathrm{fl}(a+b) = \mathrm{fl}(b+a) = \mathrm{fl}(a-(-b)) = -\mathrm{fl}((-b)-a)$.

To see that subtraction by truncation without guard digit is not faithful, consider the following example in four-digit decimal arithmetic: let $a = 999.9$ and $b = 1000$; then by property (1), $\mathrm{fl}(a-b) = 0.1 + (999.9 \bmod 1) = 0.1 + 0.9 = 1$, whereas the exact difference is 0.1. This example further shows that the computed answer can differ from the correct answer by a *factor* of the radix of the arithmetic, so the $(1+\epsilon)$ property does not apply either. (A more general form of that property does hold, however, namely there exist ϵ and δ such that $\mathrm{fl}(a-b) = a(1+\epsilon) - b(1+\delta)$ and $|\epsilon|, |\delta| < \beta^{1-t}$.)

On the other hand, an arithmetic which adds and subtracts by truncation without guard digit does so monotonically, its subtraction is obviously anti-symmetric by property (3), addition of magnitudes is performed faithfully, and properties A1 and A2 are satisfied. Furthermore the interval shift computation of section 2 works as advertised, thanks to the following result.

Lemma: If an arithmetic adds and subtracts by truncation without guard digit, then it satisfies properties S1–S3.

Proof: Let $0 \leq a \leq b$ and note that $\mathrm{fl}(b-a) = b-a$ exactly if $a \bmod \mathrm{ulp}(b) = 0$, in which case $a \bmod \mathrm{ulp}(c) = 0$ for any $c \in [a, b]$, whence $\mathrm{fl}(c-a) = c-a$ exactly, and property S1 holds. Also, $\mathrm{fl}(b-a) \bmod \mathrm{ulp}(b) = (b - a + (a \bmod \mathrm{ulp}(b)) \bmod \mathrm{ulp}(b))$, but both b and $a - (a \bmod \mathrm{ulp}(b))$ are divisible by $\mathrm{ulp}(b)$, so we must have $\mathrm{fl}(b - a) \bmod \mathrm{ulp}(b) = 0$, and thus $\mathrm{fl}(b - \mathrm{fl}(b-a)) = b - \mathrm{fl}(b-a)$ exactly, and property S2 holds. Finally, since $b - \mathrm{fl}(b-a) \leq a$ for all $0 \leq a \leq b$, property S3 holds vacuously.∎

Since subtraction without guard digit so grossly violates faithfulness, we cannot expect the algorithms of the preceding section to work correctly as written, so instead we seek to generalize them. The following result shows that we can compute the roundoff error of a sum and subsequently use it to correct the computed sum so that the values almost always match what would be computed in a faithful arithmetic. Moreover, this generalization remains correct for the hypotheses assumed above, so this result gives us an almost totally portable, if expensive way to compute a sum and roundoff.

PROPOSITION: If the floating point arithmetic satisfies either of the hypotheses (i) the arithmetic is binary and adds and subtracts by truncation without guard bit, or (ii) the arithmetic is faithful and satisfies properties A1 and A2, then given floating point numbers a and b, the following algorithm computes c and d such that $c = \mathrm{fl}(a+b)$ and $c + d = a + b$ exactly.

ALGORITHM: (Compute sum and roundoff error)

```
procedure sum_and_error( a, b )
begin
c := fl(a + b), e := fl(c − a)
f := fl(b − e)
g := fl(c − e), h := fl(g − a)
j := fl(e + f), k := fl(j − b), l := fl(f − k)
d := fl(l − h)
return c, d
end
```

PROOF: Under hypothesis (ii), the result follows immediately from the proposition of the preceding section: in that case, we showed that $f = b - e$ exactly, so we have $j = b$, $k = 0$, $l = f$, and the rest of the computation is identical to the algorithm considered above.) Therefore assume hypothesis (i), and note that it suffices to show that g, h, j, k, l, and d are computed exactly, since then $d = a + b - c$ and the proposition follows.

Consider first the case $|a| \geq |b|$. Assume $a \geq 0$; the case $a < 0$ is essentially identical. Suppose first $b \leq 0$. Then since $c = \mathrm{fl}(a+b) = \mathrm{fl}(a-(-b))$, we have $e = \mathrm{fl}(c-a) = -\mathrm{fl}(a-c) = -(a-c) = c-a$ exactly using property (3) and property S2. As addition of magnitudes is faithful, we then have $g = \mathrm{fl}(c-e) = \mathrm{fl}(c+(-e)) = a$ exactly and $h = 0$. Now by property (1), $-e = a-c = -b+(b \bmod \mathrm{ulp}(a)) \leq -b \leq a$, and since $\mathrm{fl}(a-(-e)) = \mathrm{fl}(a - \mathrm{fl}(a-c)) = a-(-e)$ exactly by property S2, we have $f = \mathrm{fl}(b-e) = -\mathrm{fl}((-b)-(-e)) = b-e$ exactly by property S1. Hence by faithfulness of magnitude addition, $j = \mathrm{fl}(e+f) = b$ exactly, so $k = 0$ and $l = f$. Since $h = 0$, $d = l$ exactly, and the proposition holds.

Now suppose $b > 0$. Then $c \geq a$, so $g = \mathrm{fl}(c-e) = \mathrm{fl}(c - \mathrm{fl}(c-a)) = c-e$ exactly by property S2. Also, by property (1), $e \geq c-a$ so $c-e \leq a$ and $\mathrm{fl}(c - \mathrm{fl}(c-e)) = c - \mathrm{fl}(c-e)$ exactly by property S2, hence $h = \mathrm{fl}(g-a) = -\mathrm{fl}(a - \mathrm{fl}(c-e)) = g-a$ exactly by property S1. Note that $h = (c-a) - \mathrm{fl}(c-a) = -a \bmod \mathrm{ulp}(c)$. Now if $e \leq b$, then since all computed quantities are multiples of $\mathrm{ulp}(b)$, we have $f = \mathrm{fl}(b-e) = b-e$ exactly, so by faithfulness of magnitude addition, $j = \mathrm{fl}(e+f) = b$ exactly, $k = 0$, $l = f = b - \mathrm{fl}(c-a)$, so by faithfulness of addition and property A1, $d = \mathrm{fl}(l-h) = b-(c-a)$ exactly. If instead $e > b$ then by properties (3) and S2, $j = \mathrm{fl}(e+f) = \mathrm{fl}(e-(-f)) = \mathrm{fl}(e - \mathrm{fl}(e-b)) = e+f$ exactly. By property (1), $e+f = b-(b \bmod \mathrm{ulp}(e)) \leq b$, and by property S2, $\mathrm{fl}(e - \mathrm{fl}(e+f)) = e - \mathrm{fl}(e+f)$ exactly, so by property S1, $\mathrm{fl}(b - \mathrm{fl}(e+f)) = b - \mathrm{fl}(e+f)$ exactly, and hence $k = \mathrm{fl}(j-b) = j-b$ exactly. Note that $e = b + (a \bmod \mathrm{ulp}(c)) - ((a+b) \bmod \mathrm{ulp}(c)) \leq b + (a \bmod \mathrm{ulp}(c))$, but since the arithmetic is binary, $\mathrm{ulp}(c) \leq 2\mathrm{ulp}(a)$ and thus $a \bmod \mathrm{ulp}(c) \leq \mathrm{ulp}(a)$; now if $b < \mathrm{ulp}(a)$ then we would have $e = 0$, contradicting $e > b$, so evidently $b \geq \mathrm{ulp}(a)$ and $e \leq 2b$, hence by Sterbenz' property and faithfulness of magnitude addition, $l = \mathrm{fl}(f-k) = b-e$ exactly. Finally, we have $-b \leq l, h \leq 0$ and all computed quantities are multiplies of $\mathrm{ulp}(b)$, so $d = \mathrm{fl}(l-h) = l-h$ exactly, and the proposition holds.

Next consider the case $|a| < |b|$, and assume $b > 0$. Suppose $a \leq 0$. Then by properties (1), (2) and (3), $e = b + (-a \bmod \mathrm{ulp}(b)) - ((c-a) \bmod \mathrm{ulp}(e)) \leq b + (-a \bmod \mathrm{ulp}(b)) < b + \mathrm{ulp}(b)$, hence $e \leq b$ and thus $\mathrm{ulp}(e) \leq \mathrm{ulp}(b)$, but then since $c - a = b + (-a \bmod \mathrm{ulp}(b))$, we have $-a \bmod \mathrm{ulp}(b) = (c-a) \bmod \mathrm{ulp}(e)$, and consequently $e = b$ exactly. Then $f = 0$, $j = b$, and $k = l = 0$. Moreover, by property (3) and property S2, $g = \mathrm{fl}(c-e) = -\mathrm{fl}(e-c) = -\mathrm{fl}(b-c) = -\mathrm{fl}(b - \mathrm{fl}(b-(-a)))$ is computed exactly, as is $\mathrm{fl}(b-(e-c)) = \mathrm{fl}(b-\mathrm{fl}(b-c))$. Since $c \geq a+b$ by property (1), we have $e-c \leq -a \leq b$, so by property S1, $h = \mathrm{fl}(g-a) = \mathrm{fl}((-a)-(e-c)) = (c-e)-a$ exactly. Hence $d = -h = a + b - c$, and the proposition holds.

Finally assume $a > 0$. Then $g = \mathrm{fl}(c - e) = \mathrm{fl}(c - \mathrm{fl}(c - a)) = c - e$ exactly by property S2. By property (1), $e \geq c - a$ so $c - e \leq a < c$ and since $\mathrm{fl}(c - g) = c - g$ exactly by property S2, we have $h = \mathrm{fl}(g-a) = -\mathrm{fl}(a-g) = g-a$ exactly by properties S1 and (3). If $e \leq b$ then since $\mathrm{fl}(c-e) = c-e$ exactly, $f = \mathrm{fl}(b-e) = b-e$ exactly by property S1. Then $j = \mathrm{fl}(e+f) = b$ exactly by faithfulness of magnitude addition, so $k = 0$ and $l = f$; by property A1 and faithfulness of addition, then, $d = \mathrm{fl}(l-h) = b-(c-a)$ exactly. If instead $e > b$ then $j = \mathrm{fl}(e+f) = \mathrm{fl}(e-(-f)) = \mathrm{fl}(e-\mathrm{fl}(e-b)) = e+f$ exactly by property S2. By property (1), $-f \geq e - b$ so $e + f \leq b < e$ and since $\mathrm{fl}(e - j) = \mathrm{fl}(e - \mathrm{fl}(e + f))$ exactly by property S2, we have $\mathrm{fl}(b - j) = b - j$ exactly by property S1, and hence $k = \mathrm{fl}(j - b) = j - b$ exactly. Now as $e > b$, f, j, and k are all multiples of $\mathrm{ulp}(b)$, and $-b \leq f, k \leq 0$, so $l = \mathrm{fl}(f - k) = f - k = b - e$ exactly. Note that by property (2), $c \leq a+b$, so $h = (c-e)-a \leq l = b-e < 0$; also, since l is a multiple of $\mathrm{ulp}(b)$, we have $\mathrm{fl}(b + l) = \mathrm{fl}(b - (-l)) = b + l$ exactly, hence $-l \leq -h \leq a < b$, so $d = \mathrm{fl}(l - h) = \mathrm{fl}((-h) - (-l)) = l - h$ exactly by property S1, and the proof is complete. ∎

Although the preceding algorithm recovers the roundoff of a sum computed in Cray X/Y-MP arithmetic, many computations which recover the roundoff, such as the algorithms in appendix A, also rely on the assumption that the roundoff error is much smaller than the computed sum, as it is in faithful arithmetics. For other arithmetics, then, simply computing the roundoff error may not be enough; we may need to correct the computed sum and roundoff to produce a new sum and roundoff so that the corrected sum is one of the two floating point numbers nearest the exact sum. For example, replacing the line "output c, d" with the lines

$$
\begin{aligned}
&s := \mathrm{fl}(c + d)\\
&t := \mathrm{fl}(s - c)\\
&r := \mathrm{fl}(d - t)
\end{aligned}
$$

output s, r

in the preceding algorithm would nearly always effect such a correction in a binary arithmetic which adds and subtracts by truncation without guard bit. For an example where the sum thus corrected would still fail to be one of the two floating point numbers nearest the exact sum, take $a = 2^t$, $b = -(1+f)$ with $0 < f < 1$; then the output is simply $s = a$, $r = b$, but the two numbers nearest the exact sum are $2^t - 1$ and $2^t - 2$. Whether algorithms such as those in appendix A would work despite this flaw remains to be seen.

More importantly, the preceding algorithm is still not sufficient to compute the roundoff of a sum in Cray 2 arithmetic. To see why, we first describe the subtraction of the Cray 2 in the following definition.

DEFINITION: A floating point arithmetic subtracts by *prerounding without guard digit* if, for all $0 \leq a \leq b$,

$$\mathrm{fl}(b-a) = b - a + (a \bmod \mathrm{ulp}(b)) - \begin{cases} \mathrm{ulp}(b) & \text{if } a \bmod \mathrm{ulp}(b) \geq \mathrm{ulp}(b)/2 \\ 0 & \text{otherwise} \end{cases}$$

Note that an arithmetic which subtracts by prerounding without guard digit satisfies properties S1 and S2 but not property S3: for a counterexample in six bit binary arithmetic, take $a = 111101_2 2^0$ and $b = 111000_2 2^3$; then $\mathrm{fl}(b-a) = 110000_2 2^3$ and $c = \mathrm{fl}(b - \mathrm{fl}(b-a)) = 100000_2 2^1$ satisfies $c > a$, but $\mathrm{fl}(c-a) = 100000_2 2^{-4} \neq 110000_2 2^{-4} = c - a$. From such examples, we can find counterexamples to the interval-shift and sum-and-roundoff algorithms. For the interval-shift program, take $x = 101100_2 2^4$, $y = 111111_2 2^0$, and $z = 111001_2 2^0$; then $t = z$, $\mathrm{fl}(x-z) = 101000_2 2^4$, and $s = 100000_2 2^1$, but $\mathrm{fl}(y-s) = 0 \neq y - s$. For the sum-and-roundoff algorithm, take $a = 100000_2 2^1$ and $b = -111111_2 2^0$; then $c = 0$, $e = -a$, $f = 0$, $g = a$, $h = 0$, $j = -a$, $k = 0$, $l = 0$, and $d = 0$, but $c + d = 0 \neq a + b$.

By these and other counterexamples, we do not intend to suggest that the Cray 2 is less accurate than the Cray X/Y-MP; in fact, Carter [12] presents an analysis that suggests the opposite: because the Cray 2 sometimes errs in its subtraction by subtracting too little and other times errs by subtracting too much, the errors tend to cancel one another more frequently than those of the Cray X/Y-MP, which always errs by subtracting too little. Neither do we suggest that programs such as those above could not be written for the Cray 2, since Kahan [35] gives an example of a program which computes the roundoff of a sum in a still more devious way which works on all commercially significant North American machines. Instead, our intent is merely to show that seemingly insignificant differences in the behavior of floating point arithmetics can drastically increase both the cost of performing accurate computations and the intellectual overhead required to prove those computations correct.

2.5 Axiom Systems for Floating Point Computation

In the preceding sections, we have noted a number of properties possessed by various floating point arithmetics; a number of authors have proposed adopting combinations of these and other properties as formal systems of axioms for floating point from which the correctness of any numerical computation must be deducible if that computation is to be deemed reliable. We briefly survey some of these proposals to highlight some of the difficulties they pose due both to their strengths, which often exclude certain floating point arithmetics, and their weaknesses, which preclude the derivation of results such as those above. We conclude that the ideal model of floating point computation must incorporate the strongest properties possible, not the weakest, in order to provide the best estimates of cost and accuracy; in practice, we can then regard a given arithmetic as an appropriate weakening of the model which may necessitate a more expensive program or a more complicated analysis.

The first attempt to formulate a complete set of axioms for numerical analysis was made by van Wijngaarden [62]. His axioms describe a faithful, monotonic arithmetic (with a few additional axioms specifically dealing with the interaction between real and integer quantities), but one cannot deduce the results of the previous sections from his axioms alone because he does not explicitly specify the set of representable numbers used to approximate real arithmetic. Consequently, while his axioms do not apply to programs running under unfaithful floating point arithmetics, they do apply to programs running under other types of arithmetics which could not possibly possess properties analogous to properties S1–S3, for example. The same deficiency afflicts the axioms suggested by Holm [29] and Wirth [66], although both specify an otherwise faithful, monotonic, commutative, and anti-symmetric arithmetic. Holm uses only fifteen axioms, compared with van Wijngaarden's 32, by explicitly referring to two rounding functions which relate the computed result of an arithmetic operation to its exact result. Despite the fact that he does not require the arithmetic to be floating point, Holm demonstrates the usefulness of his axioms by proving forty simple theorems, many of which we have used implicitly above. Wirth states many of the same properties as Holm, further condensed into just nine axioms. Wirth includes one axiom, however, which asserts that for $x \geq y \geq 0$, $\mathrm{fl}(\mathrm{fl}(x - y) + y) = x$ exactly; this axiom is somewhat unfortunate, since although it holds for Cray X/Y-MP arithmetics, it fails for most others, including IEEE 754 (in any rounding mode), DEC VAX, IBM 370, and Cray 2.

Dekker's axioms [17] appear to be the first which specifically require that the arithmetic operate on floating point numbers. He proposes four groups of axioms of which the weakest asserts nothing more than the $(1 + \epsilon)$ property while the strongest demands correct rounding. Dekker's axioms are flawed in their assumptions concerning underflow: he explicitly prohibits both gradual underflow and flush-to-zero, preferring instead that the arithmetic return the smallest nonzero number of the appropriate sign when a result would otherwise underflow. More importantly, the goal of an axiomatic approach is the unification of hypotheses, and Dekker's distinction between different families of properties contributes little to that goal.

The Brown model [11], subsequently expressed as a formal specification of floating point arithmetic by Wichmann [63], also specifically includes a definition of floating point arithmetic. Brown defines “model numbers” to be floating point numbers within a given exponent range, although he allows an arithmetic to possess other representable numbers between consecutive model numbers. His axioms then require that the arithmetic be at least as accurate as a faithful arithmetic on the model numbers; that is, any operation on model numbers must produce a number between the two model numbers nearest the exact result (and operations on non-model numbers must be no less accurate than the corresponding operations on neighboring model numbers). Brown's axioms are sufficiently general to capture the behavior of nearly every computer with floating point hardware and thus are necessarily pessimistic. For example, the axioms do not actually imply faithfulness, even on model numbers: the result of an operation on model numbers could very well be a nonmodel number strictly between the two model numbers nearest the exact result. Furthermore, Brown concedes the possibility that the same arithmetic expression or variable may evaluate to different values at different times, provided the values are all bounded by the appropriate model numbers. (This unfortunate weakness is intended to accommodate those machines which compute internally to a higher precision than a program has requested, so that a given value may be taken to refer at one point in the program to the higher precision result and at another point to the result rounded to the precision actually specified.) Consequently, proving the accuracy of a computation from Brown's axioms must involve a certain amount of nondeterminism; one could not possibly predict the behavior described in the previous sections.

Another axiom system appears in the proposed Language Independent Arithmetic Standard, or LIAS [32]. The LIAS axioms assert that only floating point numbers are representable, and they rely on a rounding function to describe a faithful arithmetic in which multiplication and division are monotonic, while addition and subtraction are weakly monotonic in so far as $u \le v \Rightarrow \mathrm{fl}(u+x) \le \mathrm{fl}(v+x)$, although the more general form $u+x \le v+y \Rightarrow \mathrm{fl}(u+x) \le \mathrm{fl}(v+y)$ may be violated. The LIAS axioms, because they require faithful addition, do not apply to Crays and hence cannot be used to prove the results of the previous section; moreover, they do not reflect all that can be said about faithful arithmetics either, so they cannot be used to deduce consequences such as the simplified sum-and-roundoff computations considered above.

Evidently no one has yet succeeded in encapsulating all that we wish to know about floating point arithmetic in a single set of axioms. We conjecture that such a task is in fact impossible. One motivation for this conjecture is the following example: compute $\arccos(x/\sqrt{x^2+y^2})$, given suitable subroutines for the arccosine and square root. In order to avoid an error in the arccosine function, we need to be certain that the argument does not exceed one in absolute value. Fortunately, as Kahan has observed, on almost all machines in widespread use today, the computed value of the argument provably cannot exceed one, but the proof almost certainly cannot be derived from any one set of axioms because it depends upon different properties in different arithmetics. Consequently, we doubt that an axiomatic approach can ever suffice to enable us to prove every statement worth proving about the correctness of floating point computations.

The best we can hope to do is to describe the behavior of each different arithmetic in sufficient detail that properties such as those in the preceding sections can be verified readily. For the programmer of portable software, however, discerning those properties upon which his software depends can be a difficult task. In this regard, the IEEE 754 standard provides a good starting point, since the arithmetic it describes seems to possess properties at least as strong as those of any other arithmetic known to date. Any program which works on at least one existing computer will likely work at least as well on a computer whose arithmetic conforms to that standard as on any other; thus, we may regard the standard as defining an ideal model of floating point arithmetic with which other arithmetics are more or less comparable. To the extent that software can be proved correct under weaker properties than IEEE 754 requires, it may be deemed more portable.

Often we may find that in order to generalize the hypotheses upon which a computation depends, we need to substitute more complicated algorithms. For example, returning to the problem of computing $\arccos(x/\sqrt{x^2+y^2})$, we can extend this example to those arithmetics for which the argument can exceed one by simply testing it prior to computing the arccosine. As another example, the algorithms in appendix A could likely be made to work correctly in Cray arithmetics by using the modified sum-and-roundoff computation of the previous section or Kahan's sum-and-roundoff computation in [35]. Nevertheless, there seem to be very few programs, and no useful ones, which work perfectly in IEEE 754 arithmetic but which cannot be modified to work at all on any other machines. For this reason, we believe programmers should not be denied the benefit of the properties the IEEE 754 standard guarantees or the weaker properties mentioned above; any axiom system which precludes the derivation of those properties must be considered inadequate.

Chapter 3

Computability, the Cost of Accurate Computations, and Numerical Stability

Although the majority of algebraic complexity theory ignores the effect of roundoff error in numerical computation, assuming instead that computation is carried out in some mathematical domain in which arithmetic operations can be computed exactly, most complexity theorists acknowledge that the effects of floating point roundoff errors must be accounted for if their work is to find practical application. In order to incorporate the effects of finite precision arithmetic in a formal model of computation, then, we must consider several key questions, of which the most important is: What level of detail regarding the properties of floating point arithmetic is sufficient to obtain realistic estimates of the costs and limitations of computation? As we have seen in chapter 2, the common practice of assuming no more than the $(1+\epsilon)$ property leads to overly pessimistic bounds on the range of problems we can solve accurately and the cost required to do so. Of course, the opposite extreme of assuming exact arithmetic is overly optimistic, since clearly there exist problems which we cannot solve exactly in floating point. Consequently, we require more detail than either of these extremes assumes if we wish to avoid unnecessary limitations and to obtain realistic estimates of cost. Our goal is to reveal the necessary level of detail by example and show that of the few models which achieve this detail, each suggests a different balance of tradeoffs between cost and accuracy.

In section 1, we propose a model of floating point computation which assumes at least a faithful arithmetic. We then use the algorithms of appendix A to show that, compared with a certain exact arithmetic model, our model imposes no unnecessary limitation on the class of problems which can be solved. Specifically, we prove that any problem which can be solved exactly in a model of computation based on exact rational arithmetic operations and comparisons can also be solved (at least approximately, to any desired accuracy) by a floating point computation. We further argue that we cannot essentially simplify the assumptions of our model without imposing artificial limits, so our hypotheses are not only sufficient but also expose the minimum level of detail we must assume to obtain the most realistic theory of floating point computation. Section 2 considers the extension of the theorem of section 1 to more general exact arithmetic models of computation; we sketch an idea which suggests that even exact models which incorporate algebraic operations (i.e., n-th roots) admit the same set of approximately solvable problems as our model of floating point computation. On the other hand, some exact models which include elementary transcendental functions can solve problems which cannot possibly be solved in floating point arithmetic.

Since finite precision arithmetic need not appreciably limit the class of problems we can solve accurately, we are naturally led to consider the costs of accurate computations. In section 3, we compare our model with several others which exhibit similar computational power ([41, 2]). As a result, we show that the differences between these models can lead to significant variations in the cost we can predict for certain types of computations given the accuracy we wish to achieve. Consequently, in order to determine the most efficient methods for solving a variety of problems, theorists and programmers alike are obliged to consider carefully the tradeoffs between cost and accuracy in each model.

Two other important questions raised by the incorporation of floating point roundoff error into a theory of computation concern the notion of numerical stability: first, given a problem, can we find a numerically stable algorithm which solves it, and second, can we decide whether a given algorithm is numerically stable? In order to discuss these questions, we first formulate a rigorous definition of numerical stability in section 4. We then reinterpret the theorem of section 1 as providing a partial answer to the first question; that is, any problem defined by an exact, piecewise rational algorithm can be solved by a numerically stable floating point algorithm. We conclude with an example which illustrates the potential difficulties posed by the second question.

3.1 A Model of Floating Point Computation

In order to discuss the limits and costs of computation, we must specify a formal model of computation in which we can describe algorithms. Our formal model of floating point computation parallels the Blum-Shub-Smale (BSS) model of computation over a ring; see [6]. One could instead define a model which resembles the real RAM model of Preparata and Shamos [55] or the algebraic computation tree of Ben-Or [5], but for our purposes these models are roughly equivalent to the BSS model. One could similarly define floating point algorithms literally in terms of Fortran programs, albeit with the understanding that such programs are not allowed to employ tricks such as accessing floating point numbers as integers. One consequence of this restriction is that with the exception of the input and output functions, the form of a floating point machine is independent of t, the precision of the arithmetic. (Such a machine can *compute* the precision t and radix β, however; see [14, 22, 44, 47], for example.)

As in appendix A, we consider the extension of floating point arithmetic to arbitrary precision by representing each number as an unevaluated sum of fixed precision floating point numbers. Specifically, we fix the radix β and let F_t denote the set of all t-digit, radix β floating point numbers and F the ring generated by F_t. (Note that F is independent of t.) Each $x \in F$ may be written as a finite sum $x = \sum_{i=1}^{n} x_i$ with $x_i \in F_t$, and for nonzero x, we may choose x_i satisfying a *non-overlapping condition*, namely that $\mathrm{ulp}(x_i) \geq \beta^t \mathrm{ulp}(x_j)$ for $i < j$. Such an expression is called a *t-digit expansion* for x; each x_i is a *component* of the expansion. (A t-digit expansion for zero consists of a single zero component.) We can encode an expansion of n components as the $(n+1)$-tuple $(n, x_1, \ldots, x_n)$; let E denote the set of all expansions thus encoded.

With the preceding terminology and notation, we are ready to define our model of floating point computation. We state the definition somewhat loosely for brevity.

DEFINITION: A *floating point machine* (or *floating point algorithm*) consists of an input space F^j, an output space F^k, a state space $\mathbf{Z}_+^m \times E^n$ (where $1 \leq j, k, n \leq \infty$ and $0 \leq m < \infty$), and a finite connected directed graph having essentially the same structure as a BSS machine with the following differences: the input function takes numbers in F to corresponding expansions; the output function takes expansions to corresponding numbers in F; computations consist of sequences of faithful floating point arithmetic operations on components of expansions, possibly including some integer arithmetic overhead; and branching predicates have the form $x > y$ (or $x \geq y$ or $x = y$) for two floating point numbers x and y. Two floating point machines are *equivalent* if they compute the same input-output map.

To show that the model thus defined imposes no unnecessary limits on computability, we wish to compare floating point algorithms with exact computations. We begin by defining a model of real number computation and considering the problems which can be solved in such a model. Here we follow some of the development in Smale [58].

The most intuitive notion of a problem is a mapping which assigns to each instance of the problem, or input, the corresponding solution, or output. The inputs and outputs usually lie in distinguished spaces (which we take to be $\mathbf{R}^n$ for appropriate n), so we may equivalently define the problem as the graph of that mapping, regarded as a subset of the product of the input and output spaces. We will use the subset definition, which has advantages in dealing with problems whose solutions are not always well-behaved functions of the input.

For now, we are concerned solely with the effect of roundoff error on otherwise exact computations. Consequently, although error in a computation can also arise from uncertainty or error in the input data and from finite or discrete approximations of infinite or continuous mathematical processes, we will nevertheless think of a problem as being the graph of the input-output mapping of some algorithm carried out with exact, real number arithmetic. From this point of view, errors arising from approximations are incorporated into the problem, reflecting the fact that such errors are quite distinct from roundoff errors and must be handled much differently in practice. Likewise, we will always regard the data actually input to the algorithm as exact, since uncertainty in the input is not related to roundoff and must also be dealt with differently.

Another consequence of viewing a problem as the output of an exact computation is the fact that each instance of the problem, or input data, has a unique corresponding solution, namely the solution computed by the exact algorithm, even though the original problem which the algorithm is intended to solve may have non-unique solutions. Again, we impose this point of view since we are concerned solely with relating the solution computed in the presence of roundoff to the solution which would be computed in exact arithmetic.

Finally, our notion of an exact, real number algorithm is a variant of the real number BSS machine with two restrictions. (Again, we could use a real RAM or algebraic computation tree model instead.) First, we restrict machines to have only zero and one as machine constants; all other constants are regarded as inputs. We justify this restriction by observing that constants of a computation must be approximated by numbers representable in the computer arithmetic, and the accuracy of this approximation clearly can affect the accuracy of the result independently of the roundoff errors which occur during the computation. By regarding these constants as inputs, then, we are merely specifying that the error in their approximation be regarded as an input error rather than a roundoff error. (In other words, the constants which are part of a computer program are just as much an input to that program as are the data encountered during the program's execution, and inaccuracies in those constants constitute input errors since such inaccuracies cannot be known to or detected by the program itself. We also note that with some slight modifications to our arguments, we could apply a much more modest restriction, say, that the machine have integer or rational constants, for example.) Second, we assume that each computation and branch node of a real number machine has an associated ordered list of arithmetic operations which carry out the evaluation of the rational functions associated with that node. We impose this restriction because the results of a computation in rounded arithmetic may depend on the order of evaluation.

Based on the above remarks, we make the following definitions.

Definition: Let $\mathcal{I} = \mathbf{R}^j$ and $\mathcal{O} = \mathbf{R}^k$ for some $1 \leq j, k \leq \infty$. (Here $\mathbf{R}^\infty$ denotes the set of all eventually vanishing sequences of real numbers.) A *problem* is a subset $\mathcal{X} \subset \mathcal{I} \times \mathcal{O}$. A *real number algorithm* is a BSS machine over the real numbers with input space $\mathcal{I}$, output space $\mathcal{O}$, and constants in the set $\{0, 1\}$, such that each computation and branch node has an ordered list of arithmetic operations which compute the rational functions associated with that node. Let $\pi : \mathcal{I} \times \mathcal{O} \to \mathcal{I}$ denote the projection $\pi(x, y) = x$ and $\phi : \mathcal{I} \to \mathcal{O}$ denote the input-output map of an algorithm M; that is, M outputs $\phi(x)$ on input x (or $\phi(x)$ is undefined if M does not halt on input x.) Then the algorithm M *solves* the problem $\mathcal{X}$ if for each $x \in \pi(\mathcal{X})$, M halts on input x and $(x, \phi(x)) \in \mathcal{X}$. Moreover, if $\mathcal{X}$ is precisely the graph of ϕ, we call $\mathcal{X}$ the *canonical problem* solved by M. Two real number algorithms are *equivalent* if they solve the same canonical problem, that is, they compute the same input-output map.

Since we have assumed that the computations in a real number algorithm are implicitly ordered, we can associate a floating point algorithm with a real number algorithm in the obvious way, as expressed in the following definition.

Definition: The *floating point implementation* of a real number algorithm is the floating point machine which has the same structure as the given algorithm with sequences of real arithmetic operations replaced by corresponding sequences of floating point arithmetic operations.

Given a floating point algorithm, we can likewise obtain a corresponding real number algorithm by replacing floating point arithmetic operations with real arithmetic operations. Notice, though, that floating point implementations of equivalent real number algorithms need not be equivalent, and real number algorithms corresponding to equivalent floating point algorithms need not be equivalent. We can, however, show that any real number algorithm which can be described in the BSS model is equivalent to another real number algorithm whose floating point implementation computes a good approximation to the input-output map of the original algorithm. In order to clarify what "good approximation" means, we introduce the following definitions.

DEFINITION: An *error criterion* for a problem $\mathcal{X} \subset \mathcal{I} \times \mathcal{O}$ is a non-negative function $e : \mathcal{I} \times \mathcal{O} \to \mathbf{R}$ such that $e(x, y) = 0$ if and only if $(x, y) \in \mathcal{X}$. An error criterion e is *effective* if there exist positive constants c, q, and ϵ_0 such that for each $(x, y') \in \mathcal{X}$ and each $y \in \mathcal{O}$, if $\max(\|y - y'\|, \|y - y'\|/\|y'\|) < \epsilon_0$ then $e(x, y) \leq c \max(\|y - y'\|, \|y - y'\|/\|y'\|)^q$.

To consider some examples of error criteria, suppose $\mathcal{X}$ is the canonical problem solved by some real number algorithm, so that each problem instance has a unique solution. Denoting the solution corresponding to the instance x by y', we can classify certain types of error criteria as follows. The *forward* error criteria have the form $e(x, y) = f(x, \|y - y'\|)$ for some function f; that is, they depend only on the distance from y to the true solution y'. (Common examples are the forward absolute criterion $e(x, y) = \|y - y'\|$ and the forward relative criterion $e(x, y) = \|y - y'\|/\|y'\|$.) In contrast, *backward* error criteria depend on the distance from x to another problem instance x' whose solution is y (assuming one exists); that is, they have the form $e(x, y) = f(x, \|x - x'\|)$. In some cases, one may define a third type of error criterion called a *residual* criterion, though this can always be rewritten (or at least bounded above and below) in terms of forward and backward error criteria.

We have chosen to use the term *effective* to describe error criteria which are essentially bounded above by forward error criteria for sufficiently small error. The requirement of effectiveness is merely a technical condition we will need below, and should not severely limit the practical applicability of our results. For example, forward relative and forward absolute error criteria are obviously effective; many backward error criteria are also effective. Generally, an error criterion can only fail to be effective if its modulus of continuity with respect to y is unbounded at some point of $\mathcal{X}$ with $y \neq 0$ or grows without bound along some curve of $\mathcal{X}$ which tends to infinity with $y \not\to 0$. (For example, take $\mathcal{X} = \{(x, y) \in \mathbf{R}^2 : \; x(y-1) = 1\}$ and $e(x, y) = \min(|x - 1/(y-1)|, 1)$; then along the line $y = 1$, $\max(\|y - y'\|, \|y - y'\|/\|y'\|) \to 0$ as $|x| \to \infty$, but $e(x, y) = 1$. Hence this error criterion is not effective.)

Note that backward error criteria and forward error criteria can often be related to one another via the geometry of the set $\mathcal{X}$; in the case of a canonical problem, the relationship is quantified specifically by the modulus of continuity of ϕ, the input-output map of the associated machine, and if ϕ is differentiable at x, the modulus of continuity can be estimated by the *condition number*, $\|D\phi(x)\|$. Consequently, a program which computes an approximate solution with a small backward error need not guarantee a small forward error in a neighborhood of an ill-conditioned problem instance, but may be quite accurate with respect to forward error in areas bounded away from such instances.

We now prove that any problem solvable by a real number algorithm as defined above can be solved approximately to within any given error by a floating point algorithm.

THEOREM: Let $\mathcal{X}$ be the canonical problem solved by some real number algorithm with input space $\mathbf{R}^j$. Define a problem $\mathcal{X}'$ over the input space $\mathbf{R}^j \times \mathbf{R}$ by $\mathcal{X}' = \{((x, \epsilon), y) : \; (x, y) \in \mathcal{X}\}$. Then

(i) there is a real number algorithm which solves $\mathcal{X}'$, and

(ii) this latter algorithm is equivalent to one whose floating point implementation on input $(x, \epsilon) \in \pi(\mathcal{X}') \cap F^{j+1}$ with $\epsilon > 0$ outputs y such that for any effective error criterion e for $\mathcal{X}'$ with constants c, q, and ϵ_0 we have $e((x, \epsilon), y) < c\epsilon^q$ whenever $\epsilon < \epsilon_0$.

PROOF: (i) Let M be a real number algorithm solving $\mathcal{X}$. Construct M' from M by extending the input space to $\mathbf{R}^j \times \mathbf{R}$ and ignoring the last input coordinate. Then M' solves $\mathcal{X}'$.

(ii) From the machine M' constructed in (i), construct another machine M'' which also solves $\mathcal{X}'$ by clearing fractions throughout M', instead simulating rational arithmetic and, if need be, performing a final division immediately preceding each output node. Note that since M' has only 0 and 1 machine constants, so does M''. Form M''' from M'' by replacing each addition and multiplication operation by the corresponding algorithm from appendix A and replacing each division prior to an output node by a variant of the division algorithm in appendix A which approximates the quotient by an expansion such that the remainder is not larger than ϵ in absolute value, nor is it larger than $\epsilon/(1+\epsilon)$ times the approximate quotient. (Prepend a branch which simply outputs zero whenever $\epsilon \leq 0$, so that we may assume $\epsilon > 0$.) Now consider the floating point implementation of M''': since its constants are in $\{0,1\}$, they can be represented exactly by single floating point numbers, regardless of the radix or precision; hence all computations other than a final division are carried out exactly, with no error.

Let e be any effective error criterion for $\mathcal{X}'$, with constants c, q, and ϵ_0 as in the definition of effectivity. Take any $(x,\epsilon) \in \pi(\mathcal{X}') \cap F^{j+1}$ with $\epsilon > 0$ and let y denote the output of the floating point implementation of M''' on input (x,ϵ), and y' the output of the original machine M on input x. If $y' = 0$ then clearly $y = 0$, so the answer is exact; hence we may assume $y' \neq 0$. If the output y was obtained without a final division in the machine M''', then it must also be exact, so assume a final division occurred. Since the remainder is smaller than ϵ in absolute value, $\|y - y'\| < \epsilon$. Since the remainder is also smaller than $\epsilon\|y\|/(1+\epsilon)$, we have $\|y-y'\|/\|y'\| < \epsilon$. Thus if $\epsilon < \epsilon_0$ we have $\max(\|y-y'\|, \|y-y'\|/\|y'\|) < \epsilon < \epsilon_0$, and hence $e((x,\epsilon),y) \leq c\max(\|y-y'\|, \|y-y'\|/\|y'\|)^q < c\epsilon^q$. ∎

The preceding theorem shows that the level of detail we have assumed in our model of floating point computation is sufficient to allow us to conclude that roundoff error need not limit what we can compute, relative to a certain exact arithmetic model of computation. We claim that while the same conclusion could be based on somewhat weaker hypotheses than we have used, we cannot rely on a lower level of detail without sacrificing a certain degree of computability or cost. To see this, consider the ray tracing problem of chapter 1: this problem amounts to the determination of the sign of a polynomial function of several variables. Using the construction of the preceding proof, we see that this problem can be solved exactly in our model of floating point computation, since we can simply evaluate the polynomial with no error and test its sign. On the other hand, in any model which, like the $(1+\epsilon)$ property, assumes that each arithmetic operation incurs an error which can at best only be estimated, we cannot evaluate the polynomial exactly, and thus we cannot always solve the preceding problem reliably. In order not to exclude such a problem from those which we can solve, then, we must use a model which enables us to know that a substantial number of arithmetic operations produce exact results (and this in turn requires the knowledge of which real numbers are exactly representable). We conclude that any model of floating point computation which purports to allow us to obtain correct and reasonable estimates of the range of problems we can solve accurately and the cost of doing so must incorporate essentially as much detail as ours.

3.2 Generalizations to Other Exact Arithmetic Models

In order to discuss the limitations of a model of computation with roundoff error, one must specify a model of computation in exact arithmetic with which to compare it. We have chosen to use the BSS model of computation over the real numbers; one could equivalently look at any of a number of other models of real computation such as algebraic computation trees or real RAMs, but the results would be essentially the same for any model in which the primitive operations include only rational arithmetic operations and comparisons. On the other hand, computer programs often make use of operations such as square roots, logarithms, exponentials, and trigonometric functions which are not exactly computable in the BSS model. Of course, these functions are generally computed by subroutines which essentially evaluate carefully chosen piecewise rational approximations to the actual functions, so in a practical sense the restriction to rational arithmetic operations is primarily a matter of cost (and to a lesser extent, of accuracy). Nevertheless, one might ask how our results would differ if we include some of these nonrational functions among the primitive operations allowed in our model of exact computation.

First consider a real number model in which algebraic operations (i.e., n-th roots) are exactly computable. Note that certain problems which are solvable in such a model, such as computing the characteristic function of the interval $[\sqrt{2}, \sqrt{3}]$, are in fact solvable in a purely rational model, since the computations involving algebraic functions can be rewritten to use only rational operations instead. In particular, all branching predicates which involve algebraic functions can be rewritten in the form of deciding whether a certain polynomial has a root in a given interval, and the latter can be decided using only rational arithmetic operations. Thus the only problems which are exactly solvable in an algebraic model but not in a purely rational model are those for which the output itself involves the evaluation of an algebraic function; a simple example is the problem of computing the real cube root of a real number. On the other hand, for these problems the output function can be rewritten in the form of finding a root of a polynomial, and we can also solve this problem as accurately as desired using only rational arithmetic operations. Hence, the theorem of the last section remains true even if our model of exact arithmetic includes algebraic operations as well as rational ones. The primary difference is one of cost: since we must not only clear fractions but also transform algebraic functions to manipulations of polynomials, the bounds on the cost of the transformed will depend not only on the cost of the original computation, the input size, and the accuracy desired, but also on the degrees of the polynomials we must construct.

The situation is quite different if we consider a real number model in which certain elementary transcendental functions are exactly computable. Certainly any problem which can be solved accurately enough using only an approximation of fixed accuracy to such a transcendental function belongs to the domain of problems to which the theorem of the last section applies, since one can simply substitute a real number machine that computes a sufficiently accurate piecewise rational approximation for the appropriate function. There are, however, problems which seem to require the exact evaluation of transcendental functions; for example, consider a computation which branches on the condition $\cos^2 x - \sin^2 x - \cos 2x > 0$. In any model with a reasonably useful set of transcendental functions, there seems to be no mechanical way to transform a branching predicate into a form that requires only rational operations, as is true with algebraic functions. Hence such a model must be able to solve problems which can't be solved in floating point simply by computing an approximation to arbitrarily high precision. In order to attempt to prove a theorem like that of the previous section, then, we would need to solve some hard number theory problems of the form "How close can a transcendental function of rational inputs come to a rational number (with appropriate restrictions on size)?" For the purposes of this dissertation as well as many practical applications, such questions require more effort than the answers would be worth. Therefore we have elected to avoid them by restricting to simpler models of real number and floating point computations.

3.3 Cost-Accuracy Tradeoffs in Models for Accurate Computation

Having shown that a reasonable model of floating point computation need not unduly limit the class of problems we can solve, we now consider the question of the cost of solving a problem accurately. In particular, we claim that the subtle differences between our model and other, equally powerful models which have been proposed can lead to striking differences in our ability to predict and express the cost as a function of the accuracy desired. In addition to our model, we consider a model which is partly implicit in the MPFUN multiple precision software package of Bailey [2] as well as one suggested by Kulisch and Miranker [41].

As part of his MPFUN multiple precision arithmetic package [2], Bailey provides a program translator, or precompiler, which accepts a standard Fortran program augmented with special comments and produces another Fortran program which calls MPFUN subroutines to perform computations to a variable level of precision. (Although the maximum precision which can be used is fixed when the original program is translated, the actual precision for each operation can be changed as the program runs.) Thus, his package presents the user with a model of floating point computation in which a number can have an arbitrary (finite) number of digits and the precision of each arithmetic operation in a computation can either be fixed in advance or chosen according to previously computed values. Of course, the cost of each operation depends upon the precision requested; for example, Bailey implements an FFT-based multiplication method which computes the product of two n-digit numbers at a cost proportional to $n \log n \log \log n$.

Another model for accurate floating point computation has been suggested by Kulisch and Miranker [41]. Their model assumes a fixed precision floating point arithmetic with one extra primitive operation, namely an inner product of two vectors. They assess unit cost for each scalar operation and a cost depending on the length of the vectors involved for an inner product. (For example, the inner product of two n-vectors could be implemented using algorithms such as those in section 4.1 at a cost proportional to $n \log n$ or with special purpose hardware at a cost proportional to n.) They also assume that each operation can either be rounded correctly or rounded in a given direction; they use the directed roundings primarily to obtain error bounds via interval arithmetic.

Formally, both of these models are essentially equivalent to ours. Bailey's package uses arrays of working precision floating point numbers to represent multiple precision numbers, and his subroutines are implemented using only working precision floating point arithmetic. Hence, his package essentially simulates a multiple precision model from within a fixed precision one such as ours. (He actually requires slightly weaker hypotheses on the underlying arithmetic, namely that operations on floating point numbers that are actually integers of moderate magnitude are exact. Thus, his package is portable to Cray computers that violate our model.) Likewise, the inner product assumed by Kulisch and Miranker can be implemented as a subroutine using only scalar floating point operations. Even directed roundings can be simulated in any faithful arithmetic that does not provide them, albeit at a cost that would not be tolerable for practical purposes (see Clemmesen [13]).

The primary difference between these three models lies in the paradigms for numerical programming that each suggests. Specifically, Bailey's package suggests a paradigm in which we simply implement an algorithm in the most straightforward way, choosing the number of digits to carry either in advance or in terms of previously computed results. In particular, we attempt, where possible, to avoid transformations such as the conversion to polynomial form suggested in the proof of the theorem of section 1. (In section 4 we will see that such transformations cannot always be avoided.) On the other hand, Kulisch and Miranker suggest a paradigm in which we transform a problem to the solution of a linear system which can then be solved to working precision by a direct elimination method followed by iterative refinement with *a posteriori* error bounds. Finally, our model, in particular the construction in the proof of the theorem of section 1, suggests a paradigm in which we determine the accuracy requirements for each intermediate result and insure that those accuracy requirements are met by converting certain computations into a polynomial form which can then be evaluated exactly as an expansion. Although we may in the worst case convert an entire computation to the evaluation of a single polynomial (or a ratio of two polynomials), as in several of the examples in chapter 4, we certainly do not preclude the possibility of reducing the size of intermediate results and hence the cost of subsequent computations either by performing exact divisions at an earlier stage, as is done in integer-preserving Gaussian elimination, for example (see Bareiss [3]), or by rounding off intermediate results; we simply suggest that these steps be done only when they can be proved not to contaminate the result.

Consider how these different paradigms impact the relationship between the cost of solving a problem and the accuracy desired of the solution. Certainly the paradigm suggested by Bailey's package requires the least effort in posing the problem as a computation, since no transformations are involved. Thus, the primary concern is choosing the precision of each operation. For many computations, we can simply perform all operations to the same precision and if that precision is high enough, we will obtain sufficient accuracy in the solution. Assuming we have an error analysis of the algorithm and perhaps an estimate of the condition of the problem instance, we can determine how high the precision should be. On the other hand, without such an analysis, determining how much precision must be carried might be impossible: there are some computations, such as the example in section 4 below, for which rounding each intermediate result to the same precision can produce a final result which is entirely wrong no matter how high a precision is used. For these computations, some transformation must be required so that certain steps can be done without error. Of course, if we transform into polynomial form, we can eventually carry enough precision that we can evaluate the appropriate polynomials exactly, but that precision must depend on the relative magnitudes of the input numbers and intermediate results, and not merely their sizes.

In contrast, the paradigm suggested by our model has the property that by using the cost estimates for the algorithms in appendix A, we can bound the cost of the floating point computation performed by the transformed algorithm in terms of the cost of the original real number algorithm, the number of components of input expansions (i.e., the input size), and $|\log \epsilon|$, where ϵ is the error, as in the theorem of section 1. Unfortunately, as one might expect, this bound can be intractably large. Consider, for example, the simple problem of evaluating a polynomial of degree d

given its coefficients and an argument. By using synthetic division and evaluating each intermediate result exactly, we obtain a program which computes the exact value of a polynomial at a given argument, but our program must accept inputs in the form of expansions and perform exact addition and multiplication on them. From the operation counts given with the algorithms in appendix A, we find that if the argument is represented as an expansion with n components, the cost of the evaluation can grow as fast as $\binom{n+d}{d}$ (see section 4.3). Hence the bound cannot be dominated by a polynomial function of the cost of the original computation and the input size. (For this example, the cost is independent of $|\log \epsilon|$; in general, the dependence on $|\log \epsilon|$ is at most linear.)

The dependence of cost on accuracy is somewhat less clear in the paradigm suggested by Kulisch and Miranker than in either of the other two, partly because the algorithms they use are intended to compute only to working precision (although intermediate results often need to be computed to higher precision, as in Lohner's algorithm for evaluating multivariate polynomials [45]), but primarily because they require iterative refinement to ensure the accuracy of the result. Thus, their cost estimates must always be based on estimates of the rate of convergence of the refinement process, and often these estimates will be too pessimistic. Many computations may turn out to require only one refinement step even though we cannot know this in advance. On the other hand, in those situations for which many steps are required, the refinement process can require more work than simply carrying higher precision or simulating it with expansions. Thus, while Kulisch and Miranker and their colleagues have shown that an inner product primitive and directed roundings are in principal sufficient to compute to any desired accuracy, it is not at all clear that their methods for doing so are the most economical.

These differences present the numerical programmer with an important decision: whether to implement a program in a straightforward way, carrying higher precision as need be using a package such as Bailey's and relying on a mathematical analysis to determine the precision necessary, or to transform the computation to a less obvious form whose accuracy is easier to guarantee but which may, depending on the nature of the problem, cost far more. We argue that we cannot draw a sharp distinction between these alternatives; the choice must be left to the programmer's wisdom and experience to decide which paradigm best suits his need. Although we can make generalizations regarding which method of implementation is more efficient in certain asymptotic limits (for example, Bailey's package is far more efficient than our algorithms for lengthy computations involving large, high precision quantities), the cost and accuracy tradeoffs between these limits are not sufficiently clear cut that we can discount any of these models.

3.4 The Existence and Classification of Numerically Stable Algorithms

One of the fundamental problems one may pose in a formal theory of numerical computation is the classification of numerically stable algorithms. In order to pose this problem in a rigorous way, we must first give a formal definition of numerical stability. To this end, we consider the class of real number algorithms defined above, and we regard the canonical problems solved by such algorithms as the exact results, so that we are focusing only on the effect of roundoff error on a computation. Under these restrictions, we propose a definition of numerical stability which generalizes the traditional concept, although our definition may look foreign to the reader accustomed to the traditional notion of numerical stability. We then reinterpret the theorem of section 1 as saying that every real number algorithm is equivalent to one whose floating point implementation is numerically stable.

Loosely speaking, numerical analysts say that an algorithm is numerically stable if it computes an answer whose error is no worse than that warranted by the input data and the precision of the computation, or to be somewhat more precise, if the computed solution is within a few units in the last place of the exact solution of a problem determined by data which are within a few units in the last place of the given data.

Of course, this last statement cannot be accepted as a definition until the meaning of "a few" is specified, and in any case, this notion is inappropriate for algorithms which are intended to approximate a solution for exact input data. Likewise, unstable algorithms often exhibit their instability only on a subset of problems; they need not be rejected out of hand if the problems of interest are known to avoid that subset. Finally, we remind the reader of the obvious fact that equivalent exact, real number algorithms, when implemented as computer programs, need not be equivalent in the presence of roundoff. Hence, we wish to generalize the customary notion of numerical stability to allow for different error measures, restricted problem domains, and numerically distinct algorithms. We therefore propose the following definition.

DEFINITION: Let $\mathcal{X}$ be the canonical problem solved by a real number algorithm with input space $\mathcal{I} = \mathbf{R}^j$, $1 \leq j \leq \infty$. The floating point implementation of that algorithm is *numerically stable* on a subset $X \subset \pi(\mathcal{X})$ with respect to an error criterion e if for each $\epsilon > 0$ there exists T such that for all $t \geq T$ and all $x \in X \cap F^j$, the solution y computed by the floating point machine using precision t satisfies $e(x, y) < \epsilon$.

Since we regard the output of the real number algorithm as exact, when we speak of numerical stability, we refer to the floating point implementation and not the real number algorithm. Numerical stability of a floating point algorithm implies numerical stability of all equivalent floating point algorithms that correspond to equivalent real number algorithms but need not imply numerical stability of nonequivalent floating point algorithms, even those corresponding to equivalent real number algorithms. Of course, in general, a real number algorithm may possess several equivalent variations whose floating point implementations are not equivalent but compute nearly the same input-output map, so that numerical stability of one implies numerical stability of all; in this case, one might loosely speak of the numerical stability of the method embodied in the algorithm without specifying a particular variation or implementation. We caution, however, that such vagueness may lead to some confusion, since subtle variations can produce very different algorithms. Consider the following example, evidently first discovered by Rice [57]. The Gram-Schmidt algorithm is a well-known way to compute a factorization $A = QR$ of a matrix A so that Q is orthogonal and R is upper triangular with non-negative diagonal; the algorithm is often presented in an order that can be characterized by saying that the R factor is computed a column at a time from left to right. Numerical programming lore maintains that if Gram-Schmidt is implemented in floating point arithmetic then the resulting Q factor may be far from orthogonal due to roundoff errors and that the so-called "modified" Gram-Schmidt algorithm, which is simply a reordering of Gram-Schmidt which computes the R factor a row at a time from top to bottom, produces a Q which is more nearly orthogonal. (Some people express this difference by claiming that modified Gram-Schmidt is "more stable" than Gram-Schmidt with respect to orthogonality of Q.) On the other hand, one can easily write a program which looks almost identical to the unmodified Gram-Schmidt algorithm, computes R a column at a time, but in floating point arithmetic computes exactly the same factors Q and R which are computed by the modified Gram-Schmidt process. Even Wilkinson admitted that for many years he used this latter algorithm without realizing that it differs from ordinary Gram-Schmidt [65].

For the class of algorithms and problems we have considered, the traditional notion of numerical stability (namely that an algorithm is stable if it computes the exact solution of a problem instance close to the given instance) is simply a special case of the preceding definition. Specifically, if on input x a floating point algorithm outputs a solution which is the true solution of a problem instance x' within k units in the last place of x, then evidently the backward relative error of the computed solution satisfies $e(x, y) = \|x - x'\|/\|x\| < k\beta^{1-t}$, so the error will be less than ϵ provided $t \geq 1 - \log_\beta(\epsilon/k)$; hence we say the algorithm is numerically stable on all of $\pi(\mathcal{X})$ with respect to a backward relative error criterion. In more intuitive terms, the traditional concept of stability expresses the idea that, compared to the solution in exact arithmetic, the computed solution is as accurate as we can reasonably expect given the precision of the arithmetic, which is tantamount to saying that increasing the precision decreases the cumulative effect of the roundoff error, though perhaps not by the same amount. Our definition likewise suggests that increasing the precision decreases the roundoff error, though not necessarily by the same amount; we have simply generalized the relationship between precision and accuracy. (In this respect, our definition and the theorem which follows have a similar spirit to the work of Malajovich [46] on dynamical systems.)

The preceding definition differs from the traditional one in that we have specifically defined numerical stability for floating point implementations of real number algorithms, and consequently we may call algorithms such as those of chapter 2 numerically stable despite the fact that their stability depends on specific properties of floating point arithmetic. Clearly such algorithms deserve to be called stable, since they compute very accurate results on the majority of computers; the fact that their stability can be lost when they are implemented in floating point arithmetics that lack certain reasonable properties is unfortunate but is no reason to deny their potential accuracy. As Henrici notes [27, p. 11], "the reader should not form the impression that stability is an invariant property which an algorithm either has or does not have. In the last analysis, the stability of an algorithm depends on the computing machine with which it is performed." Although this point is sometimes taken for granted, it deserves emphasis, since numerical stability is concerned with the effect of roundoff error on an otherwise exact computation, and the roundoff error is quite dependent on the properties of the arithmetic. (Recall the results of chapter 2 which illustrate the difference between Cray X/Y-MP and Cray 2 arithmetic, for example.)

We briefly note some possible modifications to the preceding definition of numerical stability. First, many programs can take advantage of the fact that more than one precision of floating point arithmetic may be available in a system: certain intermediate results may be computed to higher precision than either the input data or the output result. Consequently, one could generalize the definition of floating point machine to include annotations which specify relationships between the precisions of different intermediate computations; one would then maintain these relationships while considering different precisions for the overall computation. Note that, even with floating point algorithms annotated in this way, stability could still depend on specific properties not present in all arithmetics.

One other obvious modification would be to require a bound on the minimum precision T needed to reduce the error to below a given tolerance ϵ; specifically, one might demand that there exist positive constants c and q such that we can take $T \leq c|\log \epsilon|^q$. Such a requirement would be in the spirit of stipulating a polynomial bound on the cost of a computation, including the cost of multiple precision arithmetic; Smale [58] has expressed a similar idea. We have not explicitly required such a bound since we will prove that taking $T \leq |\log_\beta \epsilon|$, possibly with a constant factor, is essentially sufficient for all piecewise rational computations which would be exact in the absence of roundoff.

Modifications aside, the preceding definition raises two fundamental questions for a formal theory of numerical computation: first, which problems can be solved (at least approximately) by some numerically stable floating point algorithm, and second, which real number algorithms have numerically stable floating point implementations? We do not know a complete answer to either question, but we can give a partial answer to the first question by recasting the theorem of section 1 in the following form.

Theorem: Let $\mathcal{X}$ be the canonical problem solved by some real number algorithm with input space $\mathcal{I} = \mathbf{R}^j$. Then there is an equivalent real number algorithm whose floating point implementation is numerically stable for all inputs $x \in \pi(\mathcal{X}) \cap F^j$ with respect to any effective error criterion for $\mathcal{X}$.

Proof: Let M be a real number algorithm solving $\mathcal{X}$. Construct M' from M as in the proof of the theorem of section 1, clearing fractions and simulating rational arithmetic. This time, replace any final division step by a variant of the division algorithm in appendix A which approximates the quotient by an expansion with at least one component and whose last component has absolute value less than one. Now consider the floating point implementation of M'. Again, since all constants are in $\{0,1\}$, they can be represented exactly by single floating point numbers, regardless of the radix or precision; hence all computations other than a final division are carried out exactly, with no error.

Let e be an effective error criterion, with constants c, q, and ϵ_0 as in the definition of effectivity, and fix $\epsilon > 0$. Take any $x \in \pi(\mathcal{X}) \cap F^j$ and let y denote the output of the floating point implementation of M' on input x, and y' the output of M on input x. If $y' = 0$ then clearly $y = 0$, so the answer is exact; hence we may assume $y' \neq 0$. If the output y was obtained without a final division in the machine M', then it must also be exact, so assume a final division occurred. Since the output expansion has at least one component, evidently $\|y - y'\|/\|y'\| < \beta^{1-t}$, and since the last component is smaller than one, $\|y - y'\| < \beta^{1-t}$. Hence $\max(\|y - y'\|, \|y - y'\|/\|y'\|) < \beta^{1-t}$. Now set $T = \lceil 1 + \max(\log_\beta 1/\epsilon_0, \frac{1}{q}\log_\beta c/\epsilon)\rceil$. Then for $t \geq T$ we have $\max(\|y - y'\|, \|y - y'\|/\|y'\|) < \beta^{1-t} \leq \epsilon_0$, and hence $e(x, y) \leq c \max(\|y-y'\|, \|y-y'\|/\|y'\|)^q < c\beta^{q(1-t)} \leq \epsilon$. Therefore, the floating point implementation of M' is stable for all $x \in \pi(\mathcal{X}) \cap F^j$ with respect to the error criterion e. Since e was arbitrary, the theorem is proved. ∎

In short, the preceding theorem says that any problem which can be solved exactly by a real number machine (with exact rational arithmetic operations) can be solved approximately by a numerically stable floating point algorithm. The theorem comes close to answering a question raised by Bauer [4]: "A constructive derivation of a benign process [for an arbitrary problem] would be a great success—but it is not even clear whether to every problem that is defined by some process, a benign process exists that solves the problem." The theorem doesn't completely answer Bauer's question because his term "benign" refers to a stronger property than numerical stability, his model of roundoff error is simply the $(1+\epsilon)$ property, and his notion of "process", or algorithm, is not limited to rational arithmetic operations. Nevertheless, the answer provided by the theorem seems to be nearly as good an answer as we can hope to find, simply because any problem which cannot be solved exactly by a real number algorithm must either involve an approximation, the choice of which determines an error that cannot be reduced regardless of the precision of arithmetic, or else simply not be computable at all. The problem of numerically integrating a differential equation falls into the first category: in order to pass from the infinite, continuous notion of derivative to a finite, discrete computation we must make some approximation which limits the accuracy we can achieve independent of the precision of the arithmetic. An example from the second category is the problem of finding the extrema of a continuous function on a compact set without using the modulus of continuity. In order to give a more complete answer to our first question, then, we need to know which problems can be solved approximately in exact arithmetic and which are indeed so hard as not to be computable under any circumstances. Evidently this question is tantamount to asking what problems can be solved in constructive analysis, and that question lies well beyond the scope of this work.

Returning to the second question, namely how to decide if a given algorithm is numerically stable, we make the following observations. Certainly there are ways to identify some numerically stable and numerically unstable algorithms. For example, if an algorithm admits a backward roundoff error analysis, it is numerically stable with respect to a backward error criterion; in particular, any algorithm whose computation graph is a tree (i.e., each input and intermediate result is used only once; see [4]) admits a backward error analysis and is thus numerically stable. Likewise, since numerical instability arises as a result of close approach to a singularity, any algorithm which avoids such a singularity must be numerically stable. Unfortunately, singularities can occur in non-obvious ways, so the latter remark does not provide a practical means of identifying numerically stable algorithms. Thus, with few exceptions, there seem to be no simple ways to completely characterize numerically stable algorithms.

Finally, one might ask whether there exist problems for which the only way to achieve acceptable accuracy is via the construction of the above theorem. The following example due to J.-M. Muller provides an affirmative answer. The task is to approximate the limit as $n \to \infty$ of the sequence defined by $x_0 = 11/2$, $x_1 = 61/11$, and $x_{n+1} = 111 - (1130 - 3000/x_{n-1})/x_n$. We can regard this task as an instance of a more general problem, such as computing the limit as a function of a, b, c, d of the sequence $x_0 = a/b$, $x_1 = c/d$, $x_{n+1} = 111 - (1130 - 3000/x_{n-1})/x_n$. Hence, the input consists of a vector $(a, b, c, d) \in \mathbf{R}^4$, and we can visualize the problem as a subset of $\mathbf{R}^5$. (We could of course generalize still further.)

The exact sequence $\{x_n\}$ defined by the above formula increases slowly up to six. If we implement the computation in the obvious way using floating point arithmetic, we find that the computed sequence increases slowly for a few iterations, then jumps suddenly, eventually converging rapidly to 100. The same limit is reached regardless of the precision of the arithmetic (or the radix, or the way in which the operations are rounded), although the particular sequence we obtain varies considerably with the properties of the arithmetic.

Despite the apparently unstable behavior of the computation, we maintain that the floating point algorithm is numerically stable with respect to a backward error criterion (but not with respect to a forward error criterion). That is, due to roundoff, the computed limit turns out to be the true limit of the same recurrence formula with slightly different initial values, although it is not close to the true limit for the given initial values. We have backward stability but not forward stability because the problem has a gap at the given input data, or in other words, the input-output function is discontinuous there. We call the recurrence formula "unstable" when this happens, but that instability is to be distinguished from *numerical* instability, which refers solely to the effect of roundoff on an otherwise exact computation. (That is, the term numerical instability refers to instability of the computation, not the problem.)

The fact that the algorithm produces different intermediate values on different machines is no reason not to call it numerically stable; recall that numerical stability can depend on the properties of the arithmetic. Whereas most of the algorithms we have considered are forward stable in faithful floating point arithmetics but not in other arithmetics, Muller's example exhibits an algorithm which is backward stable in almost any floating point arithmetic, although that numerical stability occurs in ways which depend on the precision, radix, and rounding procedures. In fact, even if we were to regard the problem as one of computing the sequence itself rather than the limit, we could still say that the algorithm is numerically stable with respect to a combination of forward and backward error criteria because the computed sequence is close, point for point, to some exact sequence corresponding to initial data close to the given initial data. The particular initial data we are given happen to determine an infinitely ill-conditioned (even ill-posed) instance of the problem, so the computed sequence is not likely to be close to the true sequence for the given data, but the algorithm itself still deserves to be called numerically stable.

Note that we have assumed the inputs are the numerators and denominators of the fractions for x_0 and x_1 (which for this example can be input exactly) and the indicated divisions are part of the computation. If instead we were to take x_0 and x_1 themselves as the inputs, then we could not input the exact value 61/11 for x_1, so we would have to replace the expression 61/11 with the nearest floating point number; in this case the computed limit would be the correct limit for the problem thus specified, and we could then say that the algorithm is numerically stable with respect to a forward error criterion.

Muller's example is interesting because although the computation is not numerically stable with respect to a forward error criterion, one cannot deduce this merely by looking at the algorithm. That is, since numerical instability arises as a result of a close approach to a singularity, we can sometimes guess that a certain computation will be numerically unstable just by looking at it. But Muller's example involves numbers and operations which look perfectly innocuous, so there is no way to tell that the problem lies on a singularity, namely the discontinuity mentioned above, without carrying out a nontrivial mathematical analysis. (By using interval arithmetic we can tell that the computation is potentially numerically unstable, since the successive intervals eventually include the entire real line.)

In many ways, Muller's example resembles the ray tracing example in chapter 1: the input is exact, and the computation involves only rational arithmetic operations, so the only source of error is roundoff, but because the task amounts to the evaluation of a discontinuous function, it cannot be made forward stable simply by carrying more precision, no matter how high that precision is chosen. (Strictly speaking, there is an implicit termination criterion required in the computation of the limit, and if that criterion is fixed independent of the precision, we can eventually carry enough precision that the termination criterion is satisfied before the roundoff accumulates enough to significantly contaminate the result. Of course, in order to determine reliably how much precision we need as a function of the termination criterion used, we have to know enough about the right answer to obviate the need to do the computation anyway.) Consequently, the only reliable way to obtain a forward stable algorithm is to apply the construction of the above theorem: convert to polynomial form by clearing fractions and simulating rational arithmetic. The resulting algorithm will of course be very costly, but there seems to be no reasonable way to obtain a less expensive computation simply by rounding off intermediate results.

Chapter 4

Examples of the Cost of Accurate Computations

In the preceding chapters, we have suggested that the accuracy of a numerical computation as well as the cost of achieving that accuracy depend intimately on the properties one assumes are satisfied by a floating point arithmetic, and we have concluded that rather than merely seeking algorithms which are stable with respect to one particular measure of error, we should instead attempt to determine the relationship between accuracy and cost. In this chapter, we further illustrate these points by considering several examples of accurate computations. (By the term "accurate" we mean "more accurate than can be predicted by the $(1+\epsilon)$ property alone".) We show that a number of useful computations can be proved correct with relatively simple proofs based on the properties and algorithms presented in chapter 2. Our examples also show that in some cases we can improve significantly on the accuracy of more obvious algorithms with only a modest increase in cost, while for other computations, the cost of extra accuracy may be intolerable.

Section 1 considers a number of variations on the problem of summing a set of numbers, augmenting the repertoire of methods studied by Higham [28] with an algorithm which produces a sum correct to working precision and another which produces the exact sum in the form of an expansion. In section 2, we apply the techniques in appendix A to several simple geometric problems. These problems are good candidates for accurate computation, since the cost of computing a result carefully is often less than the cost of systematically accounting for roundoff. Section 3 considers the problem of evaluating a polynomial to within a small relative error: here we find less hopeful results, including examples which suggest that evaluating to a small relative error may be as expensive in the worst case as evaluating exactly. Finally, section 4 presents some general observations regarding the cost of accurate computations.

4.1 The Cost of Accurate Summation

Many numerical computations involve protracted sums of floating point numbers, and such sums are a common source of severe loss of significance. (That is, while Sterbenz's lemma guarantees that the faithful subtraction of two nearly equal numbers is computed exactly, a preexisting error which is small compared with either number alone can appear much larger in relation to the difference.) Consequently, the problem of computing a sum of several numbers accurately has received much attention; see Higham [28] for a survey of a number of methods and a general error analysis for the class of algorithms based on rearranging the order of terms. We consider two other classes of summation algorithms, known as *compensated summation* and *distillation*, respectively, which rely on the method of computing the roundoff error of a sum presented in chapter 2 to achieve greater accuracy than the simpler methods, albeit at somewhat greater cost.

Simply Compensated Summation

Perhaps the most obvious way to apply the sum-and-roundoff technique to compute a more accurate sum is simply to accumulate the terms as usual, recovering the roundoff each time a new term is added into the running sum and recombining that error into the total. If the error is recombined by immediately adding it to the next term in the sum, we obtain *simply compensated summation.* Kahan [36] and Knuth [40] both give error bounds for this method which show that in a faithful arithmetic, the result of a sum $\sum_{i=1}^{n} x_i$ computed by simply compensated summation can be written as $\sum_{i=1}^{n} x_i'$ where $|x_i' - x_i|/|x_i| \leq 2\epsilon + O((n-i)\epsilon^2)$; in other words, this method produces a sum with a small backward relative error. (Here ϵ denotes a bound on the roundoff of a single operation; take $\epsilon = \beta^{1-t}/2$ for correctly rounding arithmetics, $\epsilon = \beta^{1-t}$ otherwise.)

Higham [28, p. 18] considers using simply compensated summation with the summands sorted in order of decreasing magnitude and asks whether this method can then produce a sum with a small forward relative error, at least in the case $n = 3$. We prove that this is so and further show that while a similar result cannot hold in general, since the computed sum can be nonzero when the exact sum is zero, we can at least improve on the bounds above when the summands are sorted by decreasing magnitude.

In addition to the hypotheses of chapter 2, we require one further property which is satisfied by all faithful binary arithmetics as well as arithmetics which chop an addition of magnitudes. Note that, unlike properties S1–S3, this property cannot be inferred solely from faithfulness. From this property, we also derive a simple lemma.

PROPERTY S4: If $|a| \leq |b|$, a is a multiple of $\mathrm{ulp}(b)$, and $c = \mathrm{fl}(a + b)$ then $\mathrm{fl}(c - a) = c - a$ exactly.

LEMMA 1: For floating point numbers a and b, if a is a multiple of $\mathrm{ulp}(b)$, $c = \mathrm{fl}(a + b)$, and the arithmetic is faithful and satisfies properties A1, A2, and S4, then $\mathrm{fl}(c - a) = c - a$ exactly, and $d = \mathrm{fl}(b - \mathrm{fl}(c - a))$ satisfies $c + d = a + b$.

PROOF: If $|a| \geq |b|$, we can appeal to the proof of the simplified sum-and-roundoff computation in chapter 2. If $|a| < |b|$, then $\mathrm{fl}(c - a) = c - a$ exactly by property S4, and since by property A1 the roundoff error $b - (c - a)$ must be a floating point number, by faithfulness we must have $d = \mathrm{fl}(b - \mathrm{fl}(c - a)) = \mathrm{fl}(b - (c - a)) = b - (c - a)$ exactly. ∎

PROPOSITION: Given floating point numbers $x_1, \ldots, x_n$ with $|x_1| \geq \cdots \geq |x_n| > 0$ and $n \leq 1/\epsilon$. If the floating point arithmetic is faithful and satisfies properties A1, A2, and S4, then the following algorithm computes s_n such that $s_n = \sum x_i'$ where $|x_i' - x_i|/|x_i| \leq \epsilon + O(\epsilon^2)$.

ALGORITHM: (Simply compensated summation)

```
procedure comp_sum( n, x_1, ..., x_n )
begin
s_1 := x_1, c_1 := 0
for k = 2, ..., n
```

$y_k := \mathrm{fl}(c_{k-1} + x_k)$
$s_k := \mathrm{fl}(y_k + s_{k-1})$
$u_k := \mathrm{fl}(s_k - s_{k-1})$
$c_k := \mathrm{fl}(y_k - u_k)$
return s_n
end

PROOF: Referring to the proof given by Kahan [36], it suffices to show that c_k recovers the roundoff error in the computation of s_k. By lemma 1, we need only show that for all k, either $y_k = 0$ or s_{k-1} is a multiple of $\mathrm{ulp}(y_k)$. We proceed by induction: for $k = 2$, we have $s_1 = x_1$ and $y_2 = x_2 \neq 0$, and since $|x_1| \geq |x_2|$, clearly s_1 is a multiple of $\mathrm{ulp}(y_2)$.

Now assume the induction hypothesis for $k-1$. Without loss suppose that $y_k \neq 0$ and also that $s_{k-1} \neq 0$ (since if $s_{k-1} = 0$ then clearly it is a multiple of $\mathrm{ulp}(y_k)$). If in the k-th iteration of the loop we have $|s_{k-1}| \geq |y_k|$, then obviously s_{k-1} is a multiple of $\mathrm{ulp}(y_k)$, so assume $|s_{k-1}| < |y_k|$. By the induction hypothesis, and lemma 1, c_{k-1} is the roundoff error in s_{k-1}, so by faithfulness, $|c_{k-1}| < \mathrm{ulp}(s_{k-1})$; also note that since $|x_j| \leq |x_k|$ for all $j < k$, all quantities computed prior to the k-th iteration are multiples of $\mathrm{ulp}(x_k)$. Now if $|x_k| \geq |s_{k-1}|$ then clearly $|c_{k-1}| < \mathrm{ulp}(s_{k-1}) \leq \mathrm{ulp}(x_k)$, and since c_{k-1} is also a multiple of $\mathrm{ulp}(x_k)$, we must have $c_{k-1} = 0$, so $y_k = x_k$ and s_{k-1} is a multiple of $\mathrm{ulp}(x_k) = \mathrm{ulp}(y_k)$. If instead $|x_k| < |s_{k-1}|$, then since $|c_{k-1}| < \mathrm{ulp}(s_{k-1})$, by faithfulness we can have $|y_k| = |\mathrm{fl}(x_k + c_{k-1})| > |s_{k-1}|$ only if $|x_k| > |s_{k-1}| - \mathrm{ulp}(s_{k-1})$. Note that there can only be floating point numbers between $|s_{k-1}|$ and $|s_{k-1}| - \mathrm{ulp}(s_{k-1})$ if $|s_{k-1}|$ is a power of β, say $|s_{k-1}| = \beta^j$ for some j. Hence in this case, we must have $|y_k| = \beta^j + \beta^{j+1-t}$, so again s_{k-1} is a multiple of $\mathrm{ulp}(y_k)$ and the induction is complete. ∎

From the preceding proof we see that we may often have $y_k = x_k + c_{k-1}$ exactly as well, so the final sum s_n is the exact sum of quantities x_i', of which only a few differ from the given x_i, and then by scarcely more than a rounding error. Although such accuracy is nearly perfect, we can prove still more with further hypotheses: in particular, if the arithmetic is binary, we can obtain a small forward relative error in the case $n = 3$. (Note that the following result also holds regardless of the radix in arithmetics which add and subtract by truncation with guard digit.)

COROLLARY: If $\beta = 2$, $t \geq 3$, and the arithmetic is faithful and satisfies property A1, then in the case $n = 3$, the preceding algorithm computes s_3 satisfying $|s_3 - S| \leq (2\epsilon + O(\epsilon^2))|S|$, where $S = x_1 + x_2 + x_3$.

PROOF: Recall that a faithful binary arithmetic necessarily satisfies properties A2 and S4; hence, from the proof of the preceding proposition we see that c_2 is the roundoff error in s_2; in other words, $s_2 + c_2 = x_1 + x_2$ exactly. Let a and b denote the roundoff errors in the computations of y_3 and s_3 respectively, so that $y_3 + a = x_3 + c_2$ and $s_3 + b = y_3 + s_2$. We then have $s_3 = s_2 + y_3 - b = s_2 + c_2 + x_3 - a - b = x_1 + x_2 + x_3 - a - b$, so writing $S = x_1 + x_2 + x_3$, we see that $|s_3 - S| = |a + b| \leq (|a| + |b|)$. By the definition of ϵ, we also have $|b| \leq \epsilon|s_2 + y_3| = \epsilon|s_2 + c_2 + x_3 - a| = \epsilon|S - a| \leq \epsilon|S| + \epsilon|a|$, so $|s_3 - S| \leq \epsilon|S| + (1 + \epsilon)|a|$. It remains to show that $|a| \leq \epsilon(1 + O(\epsilon))|S|$.

Now assume without loss that $x_1 > 0$, and consider first the case $x_2 > 0$. In this case, we must have $S \geq x_1$. Again by definition of ϵ we have $|a| \leq \epsilon|x_3 + c_2| \leq \epsilon|x_3| + \epsilon|c_2|$; moreover, since the $|x_i|$ are decreasing, we find $|x_3| \leq |x_1| \leq |S|$ and $|c_2| \leq \epsilon|x_1 + x_2| \leq 2\epsilon|x_1| \leq 2\epsilon|S|$, so $|a| \leq \epsilon(1 + 2\epsilon)|S|$.

Now take the case $x_2 < 0$. If $x_1/2 \leq |x_2| \leq x_1$ we have $s_2 = x_1 + x_2$ exactly by Sterbenz's lemma, and thus $c_2 = a = 0$. Therefore assume $|x_2| < x_1/2$ and $c_2 \neq 0$. Since $|x_3| \leq |x_2|$, evidently $S > 0$ in this case. Now $S = s_2 + c_2 + x_3$, so if $c_2 + x_3 \geq -s_2/2$ then $S \geq |c_2 + x_3|$ and by definition of ϵ we have $|a| \leq \epsilon|c_2 + x_3| \leq \epsilon|S|$. If instead $c_2 + x_3 < -s_2/2$ then $-x_3 > c_2 + s_2/2 = (x_1 + x_2 + c_2)/2 > |c_2|$ since $|c_2| \leq \epsilon|x_1 + x_2|$ and $\epsilon < 1/4$. In particular, $x_3 < 0$ and $|x_3| \geq |c_2|$, so as c_2 is a multiple of $\mathrm{ulp}(x_3)$, if $c_2 > 0$ then $a = 0$; therefore assume $c_2 < 0$. Suppose $|S| < \beta^t \mathrm{ulp}(x_3)$; since $|x_3| < \beta^t \mathrm{ulp}(x_3)$ by definition, we have $x_1 + x_2 = S - x_3 = S + |x_3| \leq \beta^{t+1} \mathrm{ulp}(x_3)$, but since all computed quantities are multiples of $\mathrm{ulp}(x_3)$, by faithfulness $c_2 = x_1 + x_2 - \mathrm{fl}(x_1 + x_2)$ must be a one-digit multiple of $\mathrm{ulp}(x_3)$, and since $\beta = 2$, this implies $c_2 = -\mathrm{ulp}(x_3)$, $y_3 = \mathrm{fl}(x_3 + c_2)$ is computed exactly, and again $a = 0$. In the final case $|S| \geq \beta^t \mathrm{ulp}(x_3)$, note that since $\beta = 2$, the error in the sum $y_3 = \mathrm{fl}(c_2 + x_3)$ is at most $\mathrm{ulp}(x_3)$ in magnitude; that is, $|a| \le \mathrm{ulp}(x_3) \le \beta^{-t}|S| \le \epsilon|S|$. ∎

Based on the preceding arguments, one may wonder what is the smallest n for which simply compensated summation is not guaranteed to yield a result with a small forward relative error in an arithmetic which conforms to the IEEE 754 standard; the smallest known to the author is $n = 6$: set $x_1 = 2^{t+1}$, $x_2 = 2^{t+1} - 2$, and $x_3 = x_4 = x_5 = x_6 = -(2^t - 1)$; then the exact sum is 2, but the sum computed by simply compensated summation is 3.

Double Precision and Doubly Compensated Summation

Although a sum with a small backward relative error may be sufficiently accurate for most purposes, some computations, including the examples in subsequent sections, require the sum of a large number of summands computed to full working precision. We now consider two methods for computing a sum to that accuracy: the first method assumes that the computer we are using provides a floating point arithmetic with at least twice the precision of the summands themselves; the second method dispenses with that assumption by simulating doubled-precision in software, using a variant of an algorithm due to Kahan [36]. Like the algorithm presented above, both methods assume the summands are ordered by decreasing magnitude.

PROPOSITION: Let $t_1 > 2$ and $x_1, \ldots, x_n$ be t_1-digit floating point numbers with $|x_1| \ge \cdots \ge |x_n| > 0$. Assume the floating point arithmetic is faithful and carries t_2 digits, with $t_2 \ge 2t_1$. Then if $n \le \beta^{t_2 - t_1 - 3}$, the following algorithm computes s_n such that $|s_n - \sum x_i| \le \beta^{1-t_1}|\sum x_i|$.

ALGORITHM: (Double precision summation)

procedure dp_sum($n, x_1, \ldots, x_n$)
begin
$s_1 := x_1$
for $k = 2, \ldots, n$
    $s_k := \mathrm{fl}(s_{k-1} + x_k)$
return s

end

Proof: Since the $|x_i|$ are decreasing, after the k-th pass through the loop, all computed quantities must be multiples of $\mathrm{ulp}_1(x_k)$. (Here $\mathrm{ulp}_i(x)$ denotes a unit in the last place of x regarded as a t_i-digit number.) By faithfulness, then, we clearly have $s_k = \sum_{i=1}^{k} x_i$ exactly at least up to the first k for which $\mathrm{ulp}_1(x_k) \le \mathrm{ulp}_2(s_k)$. If no such k exists then $s_n = \sum x_i$ exactly, so assume such a k exists and without loss suppose that $\mathrm{ulp}_2(s_k) = 1$ for this k. Then $|s_k| \ge \beta^{t_2-1}$ while $|x_k| < \beta^{t_1}$ and thus $|x_j| < \beta^{t_1}$ for all $j > k$.

We now prove by induction that $\beta^{t_2-2} \le |s_j| < \beta^{t_2+1}$ for all $j \ge k$. Clearly the inequalities hold for $j = k$, so assume they hold for some $j - 1 \ge k$. For the upper bound, note that by faithfulness, the roundoff error in the sum $s_j = \mathrm{fl}(s_{j-1} + x_j)$ is no larger than $\mathrm{ulp}_2(s_j)$, but since $|x_j| \le |x_{j-1}| \le |s_{j-1}|$ by the induction hypothesis, evidently $\mathrm{ulp}_2(s_j) \le \beta \mathrm{ulp}_2(s_{j-1})$. Consequently, $|s_j| = |\mathrm{fl}(s_{j-1} + x_j)| \le |s_{j-1}| + |x_j| + \beta \mathrm{ulp}_2(s_{j-1}) \le |s_{j-1}| + \beta^{t_1} + \beta^2$, using the upper bound on $|x_j|$ and the fact that $\mathrm{ulp}_2(s_{j-1}) \le \beta$, the latter following from the induction hypothesis. From similar bounds for $|s_{j-1}|$ and previous terms, we obtain $|s_j| \le |s_k| + (j-k)(\beta^{t_1} + \beta^2)$; since $|s_k| \le \beta^{t_2}$ and $j - k \le n \le \beta^{t_2-t_1-3}$, we have $|s_j| \le \beta^{t_2} + \beta^{t_2-2} < \beta^{t_2+1}$ as claimed. For the lower bound on $|s_j|$, we likewise obtain $|s_j| \ge |s_{j-1}| - |x_j| - \beta \mathrm{ulp}_2(s_{j-1}) \ge |s_{j-1}| - \beta^{t_1} - \beta^2$, and continuing we find $|s_j| \ge |s_k| - (j-k)(\beta^{t_1} + \beta^2) \ge \beta^{t_2-1} - n(\beta^{t_1} + \beta^2) \ge \beta^{t_2-2}$.

We now have $|s_n| \ge \beta^{t_2-2}$, and since $\mathrm{ulp}_2(s_j) \le \beta$ for all $j > k$, the roundoff error in each sum $s_j = \mathrm{fl}(s_{j-1} + x_j)$ is at most β, so the total error $e = s_n - \sum x_i$ satisfies $|e| \le n\beta$. Since $n\beta \le \beta^{t_2-t_1-2}$ and $\beta \ge 2$, we have $|e|(1 + \beta^{1-t_1}) \le 2|e| \le 2\beta^{t_2-t_1-2} \le \beta^{t_2-t_1-1} \le \beta^{1-t_1}|s_n|$. From the outermost of these inequalities, we find $|e| \le \beta^{1-t_1}(|s_n| - |e|) \le \beta^{1-t_1}|s_n - e|$ (since $|s_n| > |e|$); in other words, $|s_n - \sum x_i| \le \beta^{1-t_1}|\sum x_i|$. ∎

If the summands are already represented in the widest precision available, we cannot simply rely on a double precision accumulator as above. Of course, using the algorithms in appendix A we could readily write a program which would effectively compute with at least twice working precision, but since those algorithms are generalized for arbitrary precision, our program would be more expensive than need be. Instead, we merely need to add a working precision quantity to a doubled-precision number (i.e., an expansion of two components) with an error no larger than would be obtained in a true double precision sum. Kahan [36] gave a simple program to do just that which first sorts the working precision number and the two components of the doubled-precision number by magnitude, then performs eight working precision additions; for our purpose, since the summands are assumed to be ordered by decreasing magnitude already, we omit the sort from Kahan's algorithm and use ten working precision additions instead. Since the resulting algorithm resembles simply compensated summation with a double application of the basic sum-and-error computation, we refer to this method as *doubly compensated summation.*

PROPOSITION: Given t-digit floating point numbers $x_1, \ldots, x_n$ with $|x_1| \geq \cdots \geq |x_n| > 0$ and $n \leq \beta^{t-3}$. If the floating point arithmetic is faithful and satisfies properties A1, A2, and S4, then the following algorithm computes s_n such that $|s_n - \sum x_i| \leq \beta^{1-t} |\sum x_i|$.

ALGORITHM: (Doubly compensated summation)

procedure dcomp_sum($n, x_1, \ldots, x_n$)
begin
$s_1 := x_1$, $c_1 := 0$
for $k = 2, \ldots, n$
$\quad y_k := \mathrm{fl}(c_{k-1} + x_k)$
$\quad u_k := \mathrm{fl}(x_k - \mathrm{fl}(y_k - c_{k-1}))$
$\quad t_k := \mathrm{fl}(y_k + s_{k-1})$
$\quad v_k := \mathrm{fl}(y_k - \mathrm{fl}(t_k - sk - 1))$
$\quad z_k := \mathrm{fl}(u_k + v_k)$
$\quad s_k := \mathrm{fl}(t_k + z_k)$
$\quad c_k := \mathrm{fl}(z_k - \mathrm{fl}(s_k - t_k))$
return s_n
end

PROOF: We show that for each k the expression $s_k + c_k$ constitutes an expansion which approximates the sum $s_{k-1} + c_{k-1} + x_k$ to two components; the result then follows by analogy with the proof of the double precision summation algorithm. To this end, it suffices to show that $|s_k + c_k - (s_{k-1} + c_{k-1} + x_k)| < \beta^{1-2t}|s_k + c_k|$ and either $s_k = c_k = 0$ or $|c_k| < \mathrm{ulp}(s_k)$.

Fix $k > 1$ and consider the quantities computed in the k-th pass through the loop. Since the $|x_i|$ are decreasing, all quantities computed in the k-th pass are multiples of $\mathrm{ulp}(x_k)$, so in particular, c_{k-1} is a multiple of $\mathrm{ulp}(x_k)$. Likewise, from the proof of the simply compensated summation algorithm above, we know that either $y_k = 0$ or s_{k-1} is a multiple of $\mathrm{ulp}(y_k)$. Hence by lemma 1, $u_k = \mathrm{fl}(x_k - \mathrm{fl}(y_k - c_{k-1})) = x_k - (y_k - c_{k-1})$ exactly and $v_k = \mathrm{fl}(y_k - \mathrm{fl}(t_k - s_{k-1})) = y_k - (t_k - s_{k-1})$ exactly; that is, u_k is the roundoff in the computation of y_k and v_k is the roundoff in the computation of t_k.

Note that if $u_k = 0$ then $z_k = v_k$ and by faithfulness obviously $|t_k| \geq |z_k|$, so by lemma 1, $c_k = \mathrm{fl}(z_k - \mathrm{fl}(s_k - t_k)) = z_k - (s_k - t_k)$ exactly and either $s_k = c_k = 0$ or $|c_k| < \mathrm{ulp}(s_k)$; in this case we have $s_k + c_k = t_k + z_k = t_k + v_k = y_k + s_{k-1} = c_{k-1} + x_k + s_{k-1}$ exactly. Therefore assume $u_k \neq 0$, which can happen only if $y_k \neq 0$.

Similarly, if $v_k = 0$ then $z_k = u_k$ so $|z_k| < \mathrm{ulp}(y_k)$, but since s_{k-1} is a multiple of $\mathrm{ulp}(y_k)$, evidently t_k is also a multiple of $\mathrm{ulp}(y_k)$, so either $t_k = 0$ or again $|t_k| \geq |z_k|$. If $t_k = 0$ we have $c_k = 0$ and $s_k = z_k = u_k = u_k + t_k + v_k = u_k + y_k + s_{k-1} = c_{k-1} + x_k + s_{k-1}$ exactly. If $|t_k| \geq |z_k|$ then we can again apply lemma 1 to see that $s_k + c_k = t_k + z_k = t_k + v_k + u_k = c_{k-1} + x_k + s_{k-1}$ exactly and either $s_k = c_k = 0$ or $|c_k| < \mathrm{ulp}(s_k)$. So assume $v_k \neq 0$ also, which can happen only if $t_k \neq 0$.

Note that by faithfulness, $|v_k| < \mathrm{ulp}(t_k)$ and $|u_k| < \mathrm{ulp}(y_k)$, so $|v_k + u_k| < |v_k| + \mathrm{ulp}(y_k)$, but as v_k is a multiple of $\mathrm{ulp}(y_k)$, this latter quantity can be no larger than the smallest power of β larger than $|v_k|$, namely $\beta^t \mathrm{ulp}(v_k)$; on the other hand, $\mathrm{ulp}(t_k)$ is a power of β larger than $|v_k|$, so $|v_k + u_k| < \beta^t \mathrm{ulp}(v_k) \leq \mathrm{ulp}(t_k)$ and by faithfulness, $|z_k| = |\mathrm{fl}(v_k + u_k)| \leq \beta^t \mathrm{ulp}(v_k) \leq \mathrm{ulp}(t_k)$ as well. Consequently, we must have $|t_k| \geq |z_k|$ so by lemma 1, $s_k + c_k = t_k + z_k$ exactly and either $s_k = c_k = 0$ or

$|c_k| < \mathrm{ulp}(s_k)$. It remains only to estimate $|t_k + z_k - (s_{k-1} + c_{k-1} + x_k)|$.

Note that $t_k + z_k - s_{k-1} - c_{k-1} - x_k = z_k - u_k - v_k$, i.e., the roundoff error in computing z_k; write $e = z_k - u_k - v_k$. Since $|z_k| \le \beta^t \mathrm{ulp}(v_k)$, by faithfulness, $|e| < \mathrm{ulp}(v_k) \le \beta^{-t}\mathrm{ulp}(t_k) \le \beta^{1-2t}|t_k|$. If also $|e| < \beta^{1-2t}|t_k + z_k| = \beta^{1-2t}|s_k + c_k|$ then the proposition holds. Suppose instead $|e| \ge \beta^{1-2t}|t_k + z_k| \ge \beta^{1-2t}(|t_k| - \mathrm{ulp}(t_k))$ since $|z_k| \le \mathrm{ulp}(t_k)$. Then obviously t_k and z_k have opposite signs, say $t_k > 0$ and $z_k < 0$; this can happen only if $v_k < 0$ as well. Also, we have $\beta^{t-1}\mathrm{ulp}(t_k) \ge \beta^{2t-1}|e| \ge |t_k| - \mathrm{ulp}(t_k)$, which can happen only if t_k is a power of β, say $t_k = \beta^t$ so $\mathrm{ulp}(t_k) = \beta$. Then $\mathrm{ulp}(v_k) > |e| \ge \beta^{1-2t}(\beta^t - \beta) = \beta^{1-t} - \beta^{2-2t}$, so $\mathrm{ulp}(v_k) \ge \beta^{1-t}$ and hence $|v_k| \ge 1$; then $y_k + s_{k-1} = t_k + v_k \le \beta^t - 1$ but $t_k = \mathrm{fl}(y_k + s_{k-1}) = \beta^t$ contradicting faithfulness. Therefore $|e| < \beta^{1-2t}|s_k + c_k|$ as claimed, and the proposition follows. ∎

Distillation

We now consider a class of algorithms which offer still a further refinement of the accuracy of a sum of n terms, namely they produce an expansion for the exact sum (or as many leading terms of that expansion as desired), at a cost of requiring more than $O(n)$ operations in the worst case. Again, we rely on the basic sum-and-roundoff computation of chapter 2, only now we use that algorithm repeatedly to reduce and condense the original n summands to another set of m numbers having the same sum but preferably with $m \ll n$; Kahan has coined the term *distillation* to describe this reduction process.

Pichat [54] presented the first complete distillation algorithm: his method consists of repeatedly passing over a partially distilled array (which is initialized with the original summands) applying the basic sum-and-roundoff computation to adjacent pairs of terms. Pichat proved that this procedure converges, i.e., after a sufficient number of passes one obtains the first component of the final expansion, which is all he wished to find. (Of course, if the summands are presorted, the doubly compensated summation method given above produces the first component at a cost comparable to two or three passes of Pichat's algorithm.) Bohlender [7] later showed that Pichat's algorithm produces the entire expansion for the sum in at most $n-1$ passes, and he added a stopping criterion for the case where only a given number of leading components are desired.

Many other algorithms which evaluate a sum or an inner product to working precision (i.e., yielding the first component of an expansion) have been developed; see Bohlender [8] for a survey of several approaches. Unfortunately, many of these techniques rely on special features, such as an extra wide fixed point accumulator, directed roundings, or interval arithmetic, which must be implemented in a combination of hardware and low-level software to be efficient. Other techniques, such as one proposed by Kulisch and Miranker [41], require direct access to the exponent and significand fields of floating point numbers, and obtaining these quantities, even when the programming environment provides a convenient way to do so, is much too time-consuming compared to the highly optimized and streamlined floating point additions and comparisons of either Pichat's algorithm or doubly compensated summation, at least for the modest values of n one typically encounters.

On the other hand, Pichat's algorithm is not always the most efficient method for distillation using only faithful floating point addition and comparison. Kahan [36] gives a distillation algorithm which first sorts the summands by decreasing magnitude, then uses successive passes like Pichat's algorithm but alternating in order: first smallest to largest, then largest to smallest. Whereas Pichat's algorithm requires $O(n^2)$ arithmetic operations in the worst case (i.e., $n-1$ passes through n numbers), Kahan claims his algorithm requires at most $O(n \log n)$ operations. We now present yet another algorithm which is easily seen to have the same worst case bound. Our method, a straightforward divide-and-conquer approach using the addition algorithm in appendix A, recursively adds successive pairs of partially distilled sums in a binary tree-like reduction. A similar approach was proposed for some parallel computer architectures by Leuprecht and Oberaigner [42].

PROPOSITION: Given floating point numbers $x_1, x_2, \ldots, x_n$. If the floating point arithmetic is faithful, the following algorithm computes an expansion $y = \sum_{i=1}^{m} y_i$ with $m \leq n$ such that $y = \sum x_i$. The algorithm requires at most $49n\lceil\log_2 n\rceil$ floating point operations; if the arithmetic also satisfies properties A1 and A2, then the simplest sum-and-roundoff computation may be substituted, reducing the cost to at most $39n\lceil\log_2 n\rceil$ operations. (See appendix A for definitions of the add and renorm algorithms.)

ALGORITHM: (Distillation)

```
procedure distill( n, x_1, ..., x_n )
begin
if n = 1
        l := 1, w_1 := x_1
else
        m := ⌊n/2⌋
        ( j, y_1, ..., y_j ) := distill( m, x_1, ..., x_m )
        ( k, z_1, ..., z_k ) := distill( n − m, x_{m+1}, ..., x_n )
        ( l, w_1, ..., w_l ) := renorm( add( j, y_1, ..., y_j, k, z_1, ..., z_k ) )
return l, w_1, ..., w_l
end
```

PROOF: That the output is an expansion for $\sum x_i$ is obvious. As for the cost, note that the number of components of a sum as computed by the renorm and add algorithms does not exceed the sum of the numbers of components of the summands; hence for each level of the binary tree, the total number of components of the expansions added at that level is at most n. Therefore the cost of each level is at most $49n$, and since the number of such levels is clearly $\lceil\log_2 n\rceil$, the total cost is at most $49n\lceil\log_2 n\rceil$. The cost estimate with the simplified sum-and-roundoff computation proceeds similarly. ∎

Kahan's algorithm in [36], though very different from the algorithm above, also incurs a cost at most proportional to $n \log n$, presumably with a modest constant, as opposed to a worst case cost of $6n(n-1)$ for Pichat's algorithm (assuming the simplest sum-and-roundoff procedure is used). Although algorithms are known which have asymptotic cost at most proportional to n, they are impractical for most purposes since they explicitly refer to the exponent and significand fields of the numbers to be added; as noted above, these quantities are very expensive to obtain compared with the cost of a few additions and comparisons. Moreover, Kahan notes that the actual cost of most distillation algorithms in practice is usually far less than the worst case bound, so we are not likely to gain anything by continuing to look for asymptotically cheaper distillation methods.

To summarize the results of this section, then, we have considered several methods for accurately computing the sum of n floating point numbers. If $S(n)$ denotes the cost of sorting the summands by decreasing magnitude, we find that we can compute a sum with a small backward relative error at a cost of $S(n) + 4(n-1)$ operations, though the forward relative error of that computed sum may be unbounded unless $n = 3$. We can compute a sum with a small forward relative error for general n at a cost of $S(n)+10(n-1)$ operations or, if a format of at least twice the precision as the summands is supported, $S(n)$ plus $n-1$ double precision additions. Finally, we can compute an expression for the exact sum at a cost of at most $49n\lceil \log_2 n \rceil$ operations without the initial sort. Although these costs are rather expensive compared with the simpler summation methods considered by Higham, he finds that the simpler methods can generally not be proven to give better than a modest backward relative error (usually depending on n and the ordering of the summands); consequently, in computations which depend on accuracy, such as those in the following sections, the accuracy guaranteed by the methods considered here may well be expected to repay their cost.

4.2 The Cost of Simple Geometric Computations

Often the simplest numerical problems present the greatest computational difficulties when roundoff must be accounted for; in these situations, we are likely to find that even though accuracy itself is not a principal goal, simply computing a result to a guaranteed accuracy solves the problem at far less cost than the tangle of defensive programming, degeneracy checking, and special cases we would otherwise require. We illustrate this point with several examples.

Finding the Area of a Triangle

Recall the ray tracing problem introduced in chapter 1. In this problem, we wish to decide whether a triangle in $\mathbf{R}^3$ given by its vertex coordinates is degenerate (i.e., all three vertices are collinear), and if not, to compute a unit normal vector to the plane containing the triangle. Ideally, we could solve both problems simultaneously: we first obtain a normal vector whose length is the area of the triangle by computing the areas of the projections of the triangle onto the three coordinate planes; if this vector is zero then the triangle is degenerate, and otherwise we can divide the vector by its length to obtain a unit normal. In the presence of roundoff, however, the computed normal vector can be nonzero even though the triangle is degenerate, or worse, the computed vector can vanish for a nondegenerate triangle. Although we might contemplate using another method to decide whether the triangle is degenerate before attempting to compute the normal, we would still need to guarantee that no nondegenerate triangles are discarded, and we would still be obliged to test our computed normal vector against zero before dividing by its length (perhaps substituting some default normal vector if the computed vector vanishes). Fortunately we can find a much simpler solution: by computing the areas of the projected triangles accurate to working precision, we can guarantee that the computed normal vector vanishes if and only if the triangle is exactly degenerate.

Other geometric computations also involve finding the areas of triangles, and these computations too can benefit from an accurate result. For example, some algorithms for constructing the Delaunay triangulation of a given set of points in the plane rely on a test which determines whether a point P lies inside the circumcircle of three other points; the obvious way to implement such a test is to compute the coordinates of the circumcenter of the three points and then test whether P is farther from the circumcenter than any of the other points. Now notice that the formulas which give the circumcenter of three points are rational functions of the coordinates of those points, and the denominator of those rational functions is the area of the triangle determined by the points. Obviously we need to compute that area accurately, then, if we are to have any hope of making the correct decision for P. (In particular, we again want to guarantee that the computed area is zero if and only if the three points are collinear.)

Consider the cost of computing the area to working precision. The area of the triangle whose vertices are (x_1, y_1), (x_2, y_2), and (x_3, y_3) is given by the formula $\sum(y_{i+1}x_i - y_i x_{i+1})/2$ where the sum is taken over cyclic permutations of the indices. (In the preceding examples, we only require a quantity proportional to the area, so we dispense with the factor of 1/2.) Consequently, in the worst case we might require a computation as expensive as the following: split each x_i to obtain x_i', x_i'' (12 operations; see appendix A for the definition of the splitting algorithm), split each y_i to obtain y_i', y_i'', y_i''' (24 operations), compute the 36 partial products of $y_{i+1}x_i$ and $-y_i x_{i+1}$ (48 operations, counting the sign changes), sort the partial products by decreasing magnitude (648 operations, assuming an $n \log_2 n$ sorting algorithm with three floating point operations per magnitude comparison), sum the partial products by doubly compensated summation (350 operations); thus, this computation requires a total of 1082 operations. For a more realistic estimate, suppose the coordinates are given as single precision numbers and we can compute the products in double precision: then we require nine operations to compute the six products and three sign changes, 45 operations for a simple n^2 sort, and 50 operations for the summation, for a total of 104 operations. Of course, a naïve implementation can compute the area (inaccurately) with only 13 operations, but how much additional work must such a program do to avoid the potential problems that inaccuracy might cause? Considering the typical complexity of maintaining topological information on geometric data, we suggest that the cost of recovering from an inaccurately computed area is likely to exceed the cost of computing the area accurately in the first place by a wide margin.

Finding the Intersection of Line Segments

Another example of a simple computation which entails much more overhead to account for roundoff than to simply avoid it in the first place is the problem of finding the intersection point of two line segments in the plane (or more generally, a line segment and a line given by two points). This problem arises in a number of computer graphics applications; in particular, we encounter it in the following clipping problem: given the vertices of a convex polygon P and one or more line segments, construct the set of line segments obtained by intersecting the given line segments with the interior of P. As P is convex, we can solve this problem by intersecting a given line segment one at a time with the lines determined by the edges of P, each time retaining only that part of the line segment lying on the "inside" of the edge. Often we wish not only to compute the clipped line segments, but also to determine some topological information about the clipped data in relation to the boundary of P; if we attempt to compute this information despite roundoff effects, we may find that the resulting data structure has a geometry which violates the topological properties we have determined. In order to determine the topology correctly, then, we must introduce artificial perturbations into the geometry and manipulate certain derived objects symbolically without actually computing them, as suggested by the data normalization and hidden variable methods of Milenkovic [49]. Needless to say, in order to maintain consistency, this approach requires great care and expense. Again, we propose a simpler solution: by computing the vertices of the clipped line segments correct to working precision, we can guarantee that we do not lose key topological relationships when computing the geometry.

For the purpose of this section, we assume an arithmetic conforming to the IEEE 754 standard [30]. In particular, the arithmetic is binary and correctly rounding, so the simplest sum-and-roundoff computation of chapter 2 may be used throughout; in addition, we assume that two different precisions are available, corresponding to two different values of t: *single precision*, for which the precision is t_1 ($t_1 = 24$ in the IEEE standard), and *double precision*, with a precision t_2 (the standard specifies $t_2 = 53$). The arithmetic includes operations between single precision operands yielding a single precision result, operations between double precision operands yielding a double precision result, and two conversion operations: one which converts a single precision number to double precision (exactly), and one which correctly rounds a double precision number to single precision provided no overflow or underflow occurs. Most importantly, we suppose that double precision, as the name implies, carries at least twice as many digits as single precision, i.e., $t_2 \geq 2t_1$; this assumption guarantees that we can multiply single precision numbers exactly by converting to and multiplying with double precision.

We formally state our problem as follows: given single precision numbers (x_1, y_1), (x_2, y_2), (x_3, y_3), and (x_4, y_4), decide whether the line segment $(x_1, y_1)(x_2, y_2)$ intersects the line determined by (x_3, y_3) and (x_4, y_4) at a unique point, and if so, compute the coordinates (x, y) of the intersection point accurate to single precision. (That is, if the exact intersection point is (x', y'), then return a point (x, y) such that x is the nearest single precision number to x' and y is the nearest single precision number to y'.) We first devise a straightforward but somewhat costly algorithm which is easily seen to solve the problem. Writing the intersection problem as a system of two linear equations, we see that the line segment intersects the line at a unique point if and only if $d \neq 0$, where $d = (y_4 - y_3)(x_1 - x_2) + (x_3 - x_4)(y_1 - y_2)$. In this case, the coordinates of the intersection point are given by

$$x = \frac{1}{d}\left[(x_1 - x_2)t + (x_3 - x_4)s\right]$$

$$y = \frac{1}{d}\left[(y_1 - y_2)t + (y_3 - y_4)s\right]$$

where $s = (x_2 y_1 - x_1 y_2)$ and $t = (x_3 y_4 - x_4 y_3)$. We can compute d by rewriting the previous expression as a sum of products, multiplying each term using double precision, then summing the terms to obtain a value of d which is accurate to double precision. (Ottman, *et al*, use the accurate inner product evaluation to compute d in their algorithm.) Then the exact value of d is zero if and only if the double precision value is zero. Similarly, we can express the numerators of the expressions for x and y as sums of products, then multiply these terms (although since each term has degree three, we need to use the splitting method from appendix A) and sum to obtain values which are again accurate to double precision. Taking the quotient of these double precision values with the computed value of d gives coordinates which are accurate to almost double precision, and at any rate sufficiently accurate to round to the correct single precision quantities, provided no underflow occurs. We formalize these ideas in the following proposition; we omit the straightforward proof.

PROPOSITION: The following algorithm solves the intersection problem with at most 1259 double precision floating point arithmetic operations including precision conversions. Moreover, no computation overflows, and the only computation which can underflow is the final conversion to single precision. (Here fl denotes double precision computation.)

ALGORITHM: (Intersect line and segment, version 1)

procedure intersect1($x_1, y_1, \ldots, x_4, y_4$)
begin
convert $x_1, y_1, \ldots, x_4, y_4$ to double precision numbers $x_1', y_1', \ldots, x_4', y_4'$
$e_1 := \mathrm{fl}(x_1' \times y_4')$, $e_2 := \mathrm{fl}(-x_1' \times y_3')$
$e_3 := \mathrm{fl}(-x_2' \times y_4')$, $e_4 := \mathrm{fl}(x_2' \times y_3')$
$e_5 := \mathrm{fl}(x_3' \times y_1')$, $e_6 := \mathrm{fl}(-x_3' \times y_2')$
$e_7 := \mathrm{fl}(-x_4' \times y_1')$, $e_8 := \mathrm{fl}(x_4' \times y_2')$
$d :=$ dcomp_sum($8, e_1, \ldots, e_8$)
if $d = 0$
    return "No unique intersection"
else
    for $i = 1, \ldots, 8$
        split e_i to obtain e_i', e_i'' with at most $t_2 - t_1$ nonzero digits

$b_1 := \mathrm{fl}(e'_1 \times x'_3),\ \ldots\ ,b_{16} := \mathrm{fl}(e''_8 \times x'_1)$
$b :=$ dcomp_sum($16, b_1, \ldots, b_{16}$)
$c_1 := \mathrm{fl}(e'_5 \times y'_4),\ \ldots\ ,c_{16} := \mathrm{fl}(e''_1 \times y'_2)$
$c :=$ dcomp_sum($16, c_1, \ldots, c_{16}$)
$x' := \mathrm{fl}(b/d)$, $y' := \mathrm{fl}(c/d)$
convert x', y' to single precision numbers x, y
return x, y
end

The preceding algorithm is easy to deduce but seems rather costly. Can we do better? For a global algorithm, the answer seems to be "no", since we must avoid destructive cancellation in computing sums. Nevertheless, we can rearrange the computation slightly to guarantee that only two sums can incur destructive cancellation, and then only rarely, by proceeding as follows. Note that we may write the coordinates of the intersection point as $x = x_1 + a(x_2 - x_1)$ and $y = y_1 + a(y_2 - y_1)$ where $a = u/(u - v)$, $u = (y_4 - y_3)(x_1 - x_3) + (x_3 - x_4)(y_1 - y_3)$, and $v = (y_4 - y_3)(x_2 - x_3) + (x_3 - x_4)(y_2 - y_3)$. In terms of u and v, we have the following possibilities: if $uv > 0$, the line segment does not intersect the line; if $u = 0$ and $v = 0$, the line segment coincides with the line; if $u = 0$ but $v \neq 0$, the line segment intersects the line at P_1, and similarly if $u \neq 0$ but $v = 0$; finally, if $uv < 0$, the line segment intersects the line at some point between P_1 and P_2.

Suppose we compute u and v accurate to double precision, by means described above. We can test for each of the aforementioned cases, and we need only perform additional computations if $uv < 0$, in which case no cancellation will occur when we compute $u - v$, and thus a is computed to near double precision. Likewise, we can compute $x_2 - x_1$ and $y_2 - y_1$ to near double precision, and therefore the products $a(x_2 - x_1)$ and $a(y_2 - y_1)$ may be computed to near double precision. Specifically, each of these quantities will be accurate to all but the last two significant bits. Thus if neither sum $x_1 + a(x_2 - x_1)$ and $y_1 + a(y_2 - y_1)$ cancels in more than $t_2 - 2 - t_1$ leading bits, then both will be sufficiently accurate to round to the correct single precision values. Moreover, we can test whether this much cancellation has occurred, and if so, we can always resort to the lengthy algorithm presented above. To summarize, we state our results (again without proof).

PROPOSITION: Let $c = 2^{t_1 - t_2 + 2}$. The following algorithm solves the above problem with at most 234 double precision floating point operations including conversions, *provided the condition in the last if statement fails.* Moreover, no computation overflows, and the only computation which can underflow is the final conversion to single precision. (Again, fl denotes computation in double precision.)

ALGORITHM: (Intersect line and segment, version 2)

procedure intersect2($x_1, y_1, \ldots, x_4, y_4$)
begin
convert $x_1, y_1, \ldots, x_4, y_4$ to double precision numbers $x'_1, y'_1, \ldots, x'_4, y'_4$
$u_1 := \mathrm{fl}(y'_4 \times x'_1), \ldots, u_6 = \mathrm{fl}(x'_4 \times y'_3)$
$u :=$ dcomp_sum($6, u_1, \ldots, u_6$)
$v_1 := \mathrm{fl}(y'_4 \times x'_1), \ldots, v_6 = \mathrm{fl}(x'_4 \times y'_3)$
$v :=$ dcomp_sum($6, v_1, \ldots, v_6$)
if $u = 0$
    if $v = 0$
        return "Intersection not unique"
    else
        return x_1, y_1
else if $v = 0$
    return x_2, y_2
else if ($u < 0$ and $v < 0$) or ($u > 0$ and $v > 0$)
    return "No intersection"
$a := \mathrm{fl}(u/\mathrm{fl}(u - v))$
$x' := \mathrm{fl}(x'_1 + \mathrm{fl}(a \times \mathrm{fl}(x'_2 - x'_1)))$, $y' := \mathrm{fl}(y'_1 + \mathrm{fl}(a \times \mathrm{fl}(y'_2 - y'_1)))$
if $|x'| \leq \mathrm{fl}(c \times |x'_1|)$ or $|y'| \leq \mathrm{fl}(c \times |y'_1|)$
    (x, y) := intersect1($x_1, y_1, \ldots, x_4, y_4$)
else
    convert x', y' to single precision numbers x, y
return x, y
end

The last if test fails only when either the x or y coordinate of the intersection point is much closer to zero than x_1, respectively y_1. Although we cannot expect to give a meaningful estimate of the probability with which the test fails, we can say loosely that it will happen "only rarely". Even then, we simply resort to the longer computation given as version 1, and here we could exercise greater care and reuse some of the quantities already computed in version 2, so that the overall cost will be slightly less than the sum of the costs of each version alone. (In this respect, our approach is similar to that suggested by Dobkin and Silver [19].) Moreover, the most pessimistic quantities in the cost estimates arise from the summation procedure, which we have taken to be doubly compensated summation with an initial $O(n^2)$ sort; hence, the costs could be further improved by using a faster sort, a quadruple precision accumulator (if quadruple precision is supported in hardware), or perhaps a different summation algorithm. Therefore, we suggest that the preceding algorithm is not only provably accurate and robust, but could easily be made very efficient as well.

We remark that Ottman, *et al*, [52, 53] show how to compute intersection points to full accuracy using single precision arithmetic, a means of computing inner products to full precision, and interval arithmetic methods based on directed roundings. In contrast, although we have assumed IEEE standard arithmetic for simplicity, our method does not require directed roundings and can be generalized to any faithful arithmetic by using the more complicated sum-and-roundoff algorithm of chapter 2 together with other algorithms in appendix A.

Along similar lines, Milenkovic [50] showed that if line equation coefficients are represented as single precision numbers with t_1 bits and computations are performed in double precision with t_2 bits, where $t_2 \geq 2t_1 + 1$, then the intersection of a line and line segment may be computed to full accuracy; unfortunately, his analysis requires that line segments defined by vertex coordinates be perturbed and replaced by lines defined by line equation coefficients despite the fact that perturbing two nearly parallel lines may move their intersection point a large distance. Moreover, Milenkovic's method also assumes that the overall computation is non-redundant; that is, once an intersection point is found, the line segments which define it are never again tested for intersection. This requirement seems reasonable, but for simple problems it is not desirable since the cost of maintaining full topological knowledge can dominate the computation. On the other hand, our algorithm may easily be modified to allow approximate endpoint intersections to be recognized, so that for simple problems, we need not assume non-redundancy.

4.3 The Cost of Accurate Polynomial Evaluation

Based on the preceding examples, one might surmise that computing a result to a modest accuracy entails only a modest cost; unfortunately, the situation is not so simple. True, for short computations such as those above, one can easily identify the important intermediate quantities which, when computed to full precision (or more), guarantee an accurate final result. For more complex computations, however, we often cannot know in advance which values must be computed to high precision, let alone how high that precision should be.

To illustrate the potential difficulties in attempting to characterize the relationship between cost and accuracy, we consider the problem of evaluating a real polynomial at a real point. Of course, any numerical programmer knows that the synthetic division method has minimal cost over all branchless real number algorithms, and Wilkinson [64] showed that a straightforward implementation of synthetic division in a faithful floating point arithmetic is stable for polynomials of bounded degree with respect to a backward relative error criterion; i.e., for the polynomial $\sum_{i=0}^{d} a_i x^i$, the computed value y can be written $y = \sum_{i=0}^{d} b_i x^i$ where $|b_i - a_i| < (2i+1)\log(1+\beta^{1-t})|a_i|$. Another estimate due to Kahan (see Adams [1]) shows that the same algorithm is stable for bounded coefficients and arguments with respect to a forward absolute error criterion: specifically, $|y - \sum a_i x^i| < 2\beta^{2-t} \sum |s_i x^i|$ where $s_i = \sum_{j=i}^{n} a_j x^{j-i}$. (The s_i are simply the intermediate quantities in the synthetic division process.)

For a forward relative error criterion, a simple implementation of synthetic division is unstable when the argument is close to a root of the polynomial: there the computed value may have no significant figures at all. In this section, we consider other variations of synthetic division and present estimates which suggest that there may be no easy way to characterize the cost of obtaining a result with a specified relative accuracy.

Cost and Precision Estimates

Of course, we may contemplate evaluating the polynomial exactly, since the additions and multiplications involved can all be computed without error using the algorithms in appendix A. Indeed, if the coefficients a_i and the argument x are given as single floating point numbers, then exact evaluation by the obvious extension of synthetic division requires at most $O(d^2)$ floating point arithmetic operations. Even if the coefficients are given as expansions of at most m components each, the cost will be at most $O(md^3)$ operations. If x is given as an expansion with n components, however, then the cost of exact evaluation by synthetic division cannot be bounded by a polynomial function of n and d; instead, the best bound that can be given grows approximately like $\binom{n+d}{d}$. On the other hand, this latter bound is obtained by considering the worst case growth in the number of components of intermediate quantities of the computation, and in this case, the exact value will also require many components for its representation. Consequently, if we do not need the exact value of the polynomial but only an approximation accurate to within a modest relative error, we should consider attempting to use less than full precision in the computation.

How much precision must we use? In general we cannot precisely ascertain the necessary precision, but we may consider a number of methods for estimating an upper bound; we could, for example, use computable lower bounds for the distances between distinct roots of an integer polynomial (of which the given polynomial must be a rational multiple, since the coefficients are floating point numbers and hence rational), then infer from those bounds a lower bound on the magnitude of the value of the polynomial at the given argument, from which we can then determine an absolute error which will be small enough to yield the desired relative error and simply evaluate the polynomial with enough precision to guarantee at most that much absolute error. Unfortunately, the bounds upon which this approach and others like it are based tend to be so large (such as Güting's [26] bounds, which are exponential in d, for example) that the resulting precision estimates will almost always yield a computation which is more expensive than simply evaluating the polynomial exactly anyway.

Whether better bounds can be found (and computed inexpensively) remains to be seen. Even with the best bounds, we conjecture that there are polynomials and arguments for which evaluation to any moderate relative accuracy is not significantly less expensive than exact evaluation. Our conjecture is based on the fact that many real polynomials have roots which are irrational, and hence cannot be found exactly by algebraic means but can be approximated by very sparse expansions, i.e., expansions $\sum m_i \beta^{k_i}$ for which $k_i \gg k_j$ for $i < j$. A program which evaluates a polynomial to modest relative accuracy at an argument close to a root must commit very small errors, if any, since the actual value of the polynomial at that argument is near zero. If the argument is a sparse expansion, the multiplications involved will cause fill-in at an exponential rate, and while most of the resulting components will eventually cancel off, the initial stages of the computation may be quite expensive. A simple example will illustrate.

Proposition: For any n there are infinitely many d such that there exist a polynomial f of degree d and a point x given as an expansion of n components with the property that the evaluation of $f(x)$ to any relative accuracy less than one by synthetic division with truncation of intermediate expansions is at least as expensive as the exact evaluation of $f(x)$ by synthetic division.

Proof: Let $s(n)$ be some fast-increasing function of n, say $s(n) = n^n$. Define $z = \sum_{i=1}^{n-1} \beta^{s(i)}$; then z is a sparse expansion. Take d to be any odd integer greater than $\beta^{s(n)}$ and let $\delta = \beta^{-ds(n)}[z^d]$, where $[x]$ denotes one of the floating point numbers nearest x. Consider the polynomial $f(x) = (x-1)^d - \delta$. This polynomial has one real root at $1 + \delta^{1/d} = 1 + \beta^{-s(n)}z'$, where $|z' - z| < |z|(1 + \beta^{1-t}/d + O(1/d^2))$; hence the expansion $x = 1 + \beta^{-s(n)}z$ is a sparse expansion which approximates the root to within $\beta^{1-t-s(n)}$. The magnitude of $f(x)$ is then at most $\beta^{1-t}\delta$, but all except the last value obtained during the evaluation of $f(x)$ by straightforward synthetic division are at least $\beta^{s(n)}\delta$ in magnitude, so no trailing components can be rounded off in the process. ∎

The above argument nearly constitutes a proof that evaluating $f(x)$ to any relative accuracy less than one is as hard as exactly evaluating *any* polynomial of degree $\Omega(d)$ at *any* n-component argument. To wit, since the initial coefficients of f grow very fast, the first several additions cannot cancel significantly, so we do indeed obtain fill-in at the maximum rate. The argument breaks down because we require that $s(n)$ grow rapidly with n to insure that z be sufficiently sparse as to achieve the greatest possible fill-in, but we must choose $d \geq \beta^{s(n)}$ in order to guarantee that the tail z' of the root agree with z. Since we need such a large d, we may not be able to bound the number of stages at which fill-in is maximal by any linear function of d; still, we can estimate that fill-in is maximal until roughly the point where the total number of components of an intermediate value (less than $\beta^{js(n)}$ at the j-th step) exceeds a significant fraction of the total fixed point "width" of that value (at least $js(n)$ at the j-th step), so if $s(n)$ grows fast enough, we can expect maximal fill-in for a large number of initial stages of the computation.

A "Nearly Optimal" Algorithm

The preceding arguments show that we cannot always know in advance (or compute by any means less expensive than the polynomial evaluation itself) a sharp *a priori* estimate of the precision we must use as a function of the relative accuracy desired of the result. On the other hand, we expect that for most polynomials and arguments, evaluating to a limited relative accuracy should be much easier than evaluating exactly. Can we devise a program, then, which is guaranteed to evaluate a polynomial to within a specified relative error without performing significantly more work than necessary? The following argument due to Kahan [34] suggests an affirmative answer.

(Before we present the analysis, we digress to make a distinction between the cost of a program which produces a result with a specified accuracy and that of one which is "guaranteed" to produce such a result. Specifically, a program that is guaranteed to produce a value within a certain error can only make such a guarantee on the basis of either an *a priori* or an *a posteriori* estimate of the error, so in the case that no non-trivial *a priori* information is available, the program must perform enough work to satisfy an *a posteriori* bound. If that bound is not sharp, there may be a large discrepancy between the theoretical minimum amount of work required to produce a solution which is sufficiently accurate and the minimum amount of work required to guarantee sufficient accuracy. Nevertheless, to the extent that the bound is the best available, we must consider the latter quantity rather than the former to be the minimum work necessary.)

We formulate our computation as follows. Since we cannot efficiently determine a reasonable *a priori* bound on the precision we should use, and we do not wish to use an unreasonable precision, we must start by simply evaluating the polynomial with a nominal precision. We can then determine whether the computed value happens to be accurate enough by using error bounds obtained during the computation such as the absolute error bound given above. If we are very fortunate, the computed value will turn out to be accurate enough and we may stop. If we are less fortunate, the computed value will not be sufficiently accurate, but from that value and the error bounds, we may be able to determine how much more precision we must carry to obtain an accurate value. In the worst case, we will have computed an inaccurate value and obtained no additional information which would allow us to determine the necessary precision to use. In this case, we must simply choose a higher precision and reevaluate; of course, if that precision is still not sufficient, then we must continue the process, choosing ever higher precisions to use. One might think that such trial-and-error would quickly become much more expensive than simply evaluating the polynomial exactly, which we were hoping to avoid. While such misfortune is indeed possible, we can nevertheless choose our sequence of trial precisions in such a way that the resulting floating point algorithm is "nearly optimal" in the following sense: the total amount of work performed, including all unsuccessful trials, is at most a small multiple of the amount of work that would have been performed had we used the minimum necessary precision (in the sense of the preceding paragraph). More precisely, we claim the following.

NOTATION: Let $TSD(f, x, p)$ denote the evaluation of a polynomial f at an argument x by synthetic division with intermediate expansions truncated to $\lceil p \rceil$ components. Let $C(d, n, p)$ denote an upper bound on the cost of $TSD(f, x, p)$ over all f of degree d and all x given as expansions with n components. (Note that we can take $C(d, n, p) = cdp \min(n, p)$ for some constant c, reflecting the fact that the most expensive steps are the multiplications of the intermediate s_i by x.)

PROPOSITION: Given the coefficients of a polynomial f of degree d, an argument x with n components, and $\epsilon > 0$, let r be any positive number such that $TSD(f, x, p)$ yields a value y satisfying $|y - f(x)| \leq r\beta^{1-pt}$. Let q be the smallest positive integer such that $TSD(f, x, q)$ produces a value y satisfying $r\beta^{1-qt} < \epsilon(|y| - r\beta^{1-qt})$. Then the algorithm outlined below computes a value y satisfying $|y - f(x)| \leq \epsilon|f(x)|$ with a total cost not larger than $4C(d, n, q) + C(d, n, 1) + O(\lceil \log_2 q \rceil)$.

ALGORITHM: (Polynomial evaluation)

procedure evalpoly(d, $f_0, \ldots, f_d$, x, ϵ)
begin
compute y by $TSD(f, x, 1)$; simultaneously compute a floating point number $r \geq 2\beta \sum |s_i x^i|$ where s_i is the i-th intermediate quantity in the computation of y
$p \leftarrow 1$
while $r\beta^{1-pt} \geq \epsilon(|y| - r\beta^{1-pt})$ do
  if $p < n$
    $\bar{p} \leftarrow \sqrt{2}p$
  else
    $\bar{p} \leftarrow 2p$
  if $|y| > r\beta^{1-pt}$

$p \leftarrow \max(2\bar{p}, \lceil \frac{1}{t}(1 - \log_\beta(\epsilon(|y| - r\beta^{1-pt})/r))\rceil)$

else

$p \leftarrow \bar{p}$

compute y by $TSD(f, x, p)$, terminating if no components were truncated

return y

end

PROOF: Let p denote the number of components carried during the last evaluation of $f(x)$. Then evidently $|y - f(x)| \leq r\beta^{1-pt} < \epsilon(|y| - r\beta^{1-pt}) \leq \epsilon|f(x)|$ so $|y - f(x)| \leq \epsilon|f(x)|$ as claimed. For the cost estimate, note first that the precision is increased by a factor of $\sqrt{2}$ at each iteration until it exceeds n, after which it is doubled at each iteration. (We may assume without loss that this remark holds even for the last iteration.) Consequently, the cost of the synthetic division step doubles with each iteration, so the total cost of all synthetic divisions is at most $2C(d, n, p)$. Let q be as defined in the proposition, and note that since q lies between the number of components carried in the last iteration and the number carried in the second-to-last (otherwise, iteration would have terminated previously), we must have $2C(d, n, p) \leq 4C(d, n, q)$. The additional terms in the cost estimate arise from the computation of r and the overhead for each iteration. ∎

In the preceding theorem, the factor of four in the cost estimate results from the fact that the cost is essentially doubled at each iteration; hence, one may ask whether that factor can be reduced by a different choice of trial precisions. In fact, four is the best possible constant and any sequence which achieves it must be essentially a repeated doubling, as we now show: let $\{w_1, w_2, \ldots\}$ be the sequence of costs associated with the trial precisions. We may assume the sequence increases, so that the algorithm terminates after n iterations where n is the smallest index satisfying $w_{n-1} < w_q \leq w_n$, with w_q denoting the cost associated with the precision q defined in the proposition. Now the total cost of all evaluations can be as large as $\sum_{j=1}^n w_j$, so the ratio of this cost to the optimum w_q can be as large as $(\sum_{j=1}^n w_j)/w_q$ which in turn can approach $(\sum_{j=1}^n w_j)/w_{n-1}$. We now appeal to a result first obtained by

Kahan [34]; here we use a special case of his theorem and give a somewhat simpler proof.

LEMMA:

$$\min_{\{w_j\}} \sup_{k\geq 1} \frac{\sum_{j=1}^{k+1} w_j}{w_k} = 4$$

where the minimum is taken over the set of all sequences $\{w_j\}_{j=1}^{\infty}$ of positive real numbers. Furthermore, a sequence $\{w_j\}$ achieves this minimum if and only if $\sum_{j=1}^{k} w_j \leq w_{k+1} \leq 3w_k - \sum_{j=1}^{k-1} w_j$ for all k.

PROOF: Note first that for the sequence $\{w_j = 2^{j-1}\}$ we have $\sup_k(\sum_{j=1}^{k+1} w_j)/w_k = \sup_k(2^{k+1}-1)/2^{k-1} = 4$. To show that no smaller supremum can be achieved, suppose now that $\{w_j\}$ is any sequence of positive numbers such that $(\sum_{j=1}^{k+1} w_j)/w_k \leq 4$ for all k. Define $s_k = \sum_{j=1}^{k} w_j$ and $r_k = s_k/s_{k+1}$ for each k. Since all w_j are positive, each r_k satisfies $0 < r_k < 1$. We have $(\sum_{j=1}^{k+1} w_j)/w_k = 1/r_{k+1}(1-r_k)$, and hence by hypothesis, $1/r_{k+1}(1-r_k) \leq 4$ for all k. Combining this with the fact that $1/x(1-x) \geq 4$ for all $x \in (0,1)$, we conclude that $r_{k+1} \geq 1/4(1-r_k) \geq r_k$, so the sequence $\{r_k\}$ is increasing. Since it is bounded above, it must converge; say $r_k \to L$. Since $1/r_{k+1}(1-r_k) \leq 4$ for all k, we have $1/r_{k+1}(1-r_k) \to 1/L(1-L)$ and also $1/L(1-L) \leq 4$. The latter can happen only when $L = 1/2$, for which equality holds. Hence $\sup_k(\sum_{j=1}^{k+1} w_j)/w_k = \sup_k 1/r_{k+1}(1-r_k) = 4$.

For the second part of the lemma, suppose $\{w_j\}$ satisfies $\sup_k(\sum_{j=1}^{k+1} w_j)/w_k = 4$ and define r_k as above. By previous arguments, the sequence $\{r_k\}$ increases and converges to $1/2$, hence each $r_k \leq 1/2$ and thus $2\sum_{j=1}^{k} w_j \leq \sum_{j=1}^{k+1} w_j$, which gives $w_{k+1} \geq \sum_{j=1}^{k} w_j$. On the other hand, we have $\sum_{j=1}^{k+1} w_j \leq 4w_k$, whence $w_{k+1} \leq 3w_k - \sum_{j=1}^{k-1} w_j$. For the converse, if $\{w_j\}$ satisfies $w_{k+1} \leq 3w_k - \sum_{j=1}^{k-1} w_j$ for each k, then clearly $\sup_k(\sum_{j=1}^{k+1} w_j)/w_k \leq 4$; from the preceding paragraph, we know that equality must hold. ∎

Although we cannot give a good estimate of the total cost of the preceding algorithm solely as a function of ϵ and the degree d of f, we can express the cost in relation to the true value $f(x)$. Specifically, let r be the coefficient computed in the first step of the algorithm. Assume the final precision required is larger than n, the size of x; then the cost of the algorithm is at most roughly $4C(d,n,q) = O(dq)$, where q is the smallest integer satisfying $r\beta^{1-qt} < \epsilon(|y| - r\beta^{1-qt})$ for some $|y| \geq |f(x)| - r\beta^{1-qt}$; hence $q \approx \frac{1}{t}(1 - \log_\beta \frac{\epsilon|f(x)|}{(1+2\epsilon)r})$.

We note that a similar bound is claimed by Böhm [9] for another polynomial evaluation algorithm. His method first computes an approximation to $f(x)$ using working precision, then repeatedly refines this approximation by first computing interval bounds on the backward error implicit in the coefficients of f from the values obtained during the previous computation, then computing interval bounds on the corresponding forward error in $f(x)$; the midpoint of the forward error interval is added as a correction to the approximate value of $f(x)$, while the interval itself determines whether that approximation is sufficiently accurate. Böhm proves a convergence estimate for this process which can be used to estimate the total number of refinements needed. He also claims that in practice, that estimate is too pessimistic; instead, the forward error interval appears to shrink roughly by a factor like β^{1-t} at each step. Consequently, we can estimate the cost of his algorithm as follows: suppose the error of the initial approximation to $f(x)$ satisfies our bound, namely $|y - f(x)| \leq r\beta^{1-t}$. Then after $k-1$ refinements, the error will be reduced to $|y - f(x)| \leq r\beta^{k(1-t)}$, so the number of refinements required to reduce the error to $\epsilon|f(x)|$ will be given by $k \approx \frac{1}{1-t}\log_\beta \frac{\epsilon|f(x)|}{r}$. In the case that the argument x and the coefficients of f are all single floating point numbers, each refinement pass costs $O(d)$, so his method will have the same asymptotic cost as ours. (Of course, this case is one for which exact evaluation is relatively inexpensive as well.)

Lohner [45] extends Böhm's algorithm to include the case where the coefficients of f are given as expansions. In Lohner's version, the initial approximation to $f(x)$ is again computed using working precision, but each subsequent refinement step computes backward error bounds using the full expansions of the coefficients. If the coefficients have as many as m components, then the cost of each refinement pass is $O(md)$; consequently, for $k < m$, the cost of the k-th refinement step in Lohner's algorithm could well be reduced by using only the first k components of each coefficient, in which case the cost of that refinement pass would compare with $C(d, 1, k)$, the cost of synthetic division with truncation to k components. Moreover, Lohner's method could also be extended to allow the argument x to be a multiple component expansion. In this case, the backward error estimation in the k-th iteration would need to include k leading components of x, and the cost would be comparable to $C(d, n, k)$.

The preceding cost estimates reflect the fact that in Böhm's and Lohner's algorithms, each refinement step actually computes interval bounds for each intermediate result in the synthetic division, and thus can be viewed as the computation of one more component in an expansion for each result. Whereas our algorithm as stated simply repeats the entire computation each time the precision is increased, their methods avoid recomputing leading components which have already been computed. Instead they require recomputation of the backward error bounds using the full k-component expansions at each pass, and this latter computation is essentially as expensive as evaluating by ordinary synthetic division with truncation to k components. Unlike our algorithm, however, their method can *only* increase the precision level one component at a time, and thus may require more work to reach an accurate value, particularly if the true value is quite small and the precisions of x and the coefficients of f are large. We also note that because the backward error bounds are generally somewhat pessimistic while the forward error bounds for the intermediate results are computed simply using synthetic division carried out with interval arithmetic, Böhm's and Lohner's algorithms can occasionally incur overflow during the iterative refinement on polynomials which our algorithm can handle without difficulty.

Finally we note another approach to accurate polynomial evaluation considered by Mesztenyi and Witzgall [48]. They prove that if the polynomial is given in an appropriate Newton form (i.e., in the form $f(x) = \sum_{i=0}^{d} c_i \prod_{j=1}^{i}(x - x_j)$) then evaluation by the obvious modification of synthetic division using only working precision produces a value which is accurate to within a small forward relative error. They also give an algorithm which converts a polynomial given in the usual power form to a suitable Newton form. Unfortunately that conversion requires finding approximations to the roots and critical points of the polynomial in a given interval, which in general depends on the accurate evaluation in turn. Hence, unless we are given the polynomial in the appropriate Newton form to begin with, we cannot use their method to reduce the cost of accurate evaluation.

4.4 Sparsity and The Cost of Accurate Computations

The preceding examples illustrate that the cost of guaranteed accuracy can be quite difficult to characterize reliably; for some computations, the cost may be well within reason, while for others we simply cannot find in advance any modest bound on the cost required. Nevertheless, we can make several general observations and draw some tentative conclusions about the conditions under which we may expect to compute accurate results at a reasonable cost.

As noted above, a t-digit expansion as defined in appendix A can be viewed as a sparse floating point representation of a number, by analogy with a sparse representation of a polynomial: specifically, a sparse polynomial may be represented by a finite set of pairs (a_j, k_j), $k_j \in \mathbf{N}$, which together encode the polynomial $\sum a_j x^{k_j}$, while an expansion is represented by a finite set of floating point numbers, i.e., pairs (a_j, k_j), $k_j \in \mathbf{Z}$, which together encode the number $\sum a_j \beta^{k_j}$, where β is the radix of the arithmetic. We generally express the cost of a computation as a function of the *size* of the input. The size of a sparse polynomial is not its degree, which is the natural notion of size for a densely represented polynomial, but the number of non-vanishing coefficients; likewise, the size of an expansion is not its precision, which is the natural measure for multiple precision floating point numbers, but the number of components. Consequently, we expect that the cost of a computation as a function of input size may be large when the inputs are sparse but the intermediate results fill in significantly. One way to avoid that possibility is simply to limit the possible inputs: if the input size is bounded or the input is guaranteed to be dense, then computed results cannot fill in too fast; for example, accurate polynomial evaluation is relatively inexpensive when the argument consists of either a single component or a dense expansion. Similarly, we can bound the fill-in for computations which involve only a few operations, such as the geometric computations described above. (This fact is more useful than it might seem at first sight, since in practice a long computation may involve only a few intermediate results which require high accuracy.)

Finally, in chapter 5 we will examine more closely the relationship between arbitrary precision arithmetic with expansions and multiple precision floating point arithmetic, such as that provided by a number of software packages [2, 10, 59]. For now we simply remark that, unlike multiple precision floating point, the sparse representation of expansions allows us to bound the cost of a computation solely in terms of the sizes of the inputs and the sequence of operations, independent of the relative magnitudes of the numbers involved. Hence, while the sparsity may lead to a cost estimate that appears unduly large, that cost may still be much smaller than the cost of computing a result with comparable accuracy using multiple precision floating point.

Chapter 5

Implications of Accurate Computations

In this chapter, we attempt to put the foregoing results into perspective and answer the question with which we began: "Who cares?" Certainly the vast majority of numerical computation performed to date has been analyzed and carried out without benefit of any stronger assumption than the $(1 + \epsilon)$ property; consequently, many numerical analysts and programmers and many of the computer scientists who design the machines on which numerical computations are performed see no reason to delve into the esoteric properties and proofs of the preceding chapters, much less overwhelm the average user with them. Alas, the average user is overwhelmed nevertheless, often with overly pessimistic cost and error estimates, sometimes with ill-formed ideas or ill-considered advice, and occasionally with gross misinformation. Of course, many experts claim that results such as ours are flukes and that reasonable computations should not rely on "tricks", but a trick deceives only those who do not know how it is done; to the magician, a trick is simply an artful combination of knowledge and skill.

Theorists and practitioners of numerical computation alike stand to benefit from the knowledge and skill the results of the preceding chapters confer: the numerical analyst may reconsider algorithms once deemed unstable to see if they can be rescued by a judicious use of careful computation or extra precision, the complexity theorist may notice that even an ill-conditioned problem admits an inexpensive and accurate solution, the programmer may find that computing an accurate solution to a problem is easier than systematically accounting for inaccuracy, and the computer manufacturer may be forced to consider more carefully the impact of a modification in the design of a computer's arithmetic. Of course, the knowledge upon which our results are based has been available for some time; we have merely shown that the cost of assimilating that knowledge is not as great as is commonly thought. In the following sections, we consider some of the ways in which the fruits of that knowledge can repay its cost.

5.1 Implications for Numerical Analysis and Complexity Theory

The examples of the preceding chapters raise a number of questions for the theory of numerical computation. Perhaps the most practical question is whether a simple modification to an otherwise inaccurate computation can restore sufficient accuracy. Often certain algorithms which appear unstable are nevertheless desirable because they are particularly inexpensive, amenable to parallel computation, or uniquely capable of taking advantage of some special property of the problem. Consequently, rather than simply discarding such algorithms and seeking other, more stable methods, we may have good reason to consider whether these algorithms can be implemented in a form which does not lose accuracy due to roundoff. The results of chapter 3 show that any piecewise rational algorithm which computes the correct result in exact arithmetic can indeed be implemented in such a form, although the cost of the resulting computation may render it infeasible. Nevertheless, for many algorithms the instability may not be characteristic of the method as a whole but simply an artifact of careless computation of one or two intermediate results. In these cases, we may be able to find simple rearrangements of operations or minor additions to the algorithm which, by changing our ability to predict the accuracy of certain computations, enable us to recover the desired accuracy.

As a trivial example, consider the following computation analyzed by Kahan [36]. Define

$$q(y) := |y - \sqrt{y^2+1}| - \frac{1}{y+\sqrt{y^2+1}}, \quad f(z) := \begin{cases} \frac{e^z-1}{z} & \text{if } z \neq 0 \\ 1 & \text{otherwise} \end{cases}, \quad g(x) := f(q(x)^2)$$

In the absence of roundoff, $g(x) = 1$ for all x. In a faithful floating point arithmetic, the most obvious implementation of this computation will compute $g(n) = 0$ instead for almost all of $n = 15, 16, 17, \ldots, 9999$, no matter how great the precision. Consider how this error becomes evident: due to roundoff, $q(n)$ is almost always not zero but instead comparable to a unit in the last place of n; then $z = q(n)^2$ is a positive number much smaller than a unit in the last place of 1, so $e^z \approx 1 + z$ almost always rounds to 1, and the computation finds $g(n) = 0$. Suppose instead that e^z were approximated by an expansion accurate to two components; the first component would be 1 and the second z, so the computed value of $e^z - 1$ would be z and hence we would recover $g(n) = 1$, restoring stability. Note that in this case simply carrying a greater *fixed* precision would not achieve the same effect. (Of course, an unimaginative student might admonish the original program for testing $z = 0$ and instead insist that the function $f(z)$ be computed by something like

$$f(z) := \begin{cases} \frac{e^z - 1}{z} & \text{if } |z| > \tau \\ 1 + z/2 + z^2/6 & \text{otherwise} \end{cases}$$

for some small positive number τ, but unless some care is taken, this computation will likely be less accurate than our proposed solution. The experienced numerical programmer, on the other hand, will recognize that the best solution is to compute f using a function like $\mathrm{expm1}(z) := e^z - 1$ which is available in many programming environments and computes its value to full relative accuracy even when $|z|$ is small. If expm1 is not available, a more subtle program may be needed; see [36].)

A more relevant example is provided by Sorensen and Tang [60]. They show that by evaluating a certain function to at least twice working precision, a well-known parallel algorithm for computing eigensystems of real symmetric matrices can be guaranteed to give eigenvectors which are orthogonal to working accuracy. In order to evaluate the function that accurately, they can either assume single (respectively double) precision as a basis and use double (respectively quadruple) precision for the function evaluation or else simulate doubled-precision arithmetic by techniques similar to the algorithms in appendix A. If such careful computation cannot be performed, perhaps because the properties upon which it depends are not possessed by the computer we wish to use, then we must resort to a clever approach suggested by Gu and Eisenstat [25]. They avoid the use of multiple precision computation by evaluating the relevant function to working precision, yielding a possibly inaccurate result for the given data, then perturbing the original problem to a new problem whose solution is precisely the computed solution. Gu and Eisenstat's method, while it does not depend on the specific properties which Sorensen and Tang require, incurs a small extra cost in the form of computing the perturbed problem. Perhaps the more important cost, though, is the intellectual expense of finding such a method in the first place; certainly the question of whether accuracy can be restored simply by computing one or two quantities to higher precision seems easier to answer than that of how to compute new data which correspond to an inaccurate solution.

Another question raised by the preceding results concerns the relationship between accuracy and cost. Traditionally, numerical analysts have tended to suggest that a result computed by a numerically stable algorithm is as accurate as one can reasonably expect, so users of numerical software have become accustomed to expecting the accuracy of a result to be determined by the precision of the arithmetic, the properties of the algorithm, and the condition of the problem. This point of view is needlessly pessimistic; the accuracy we can achieve is in fact limited only by the cost we are willing to tolerate. The accuracy we require, on the other hand, depends on how the result will be used: the result must be sufficiently accurate to guarantee that the decisions made on the basis of the computed solution agree with the decisions that would be made on the basis of the exact solution. We therefore must characterize the cost as a function of accuracy so that a user can predict when the benefit of that accuracy will be worth the work needed to achieve it.

Intuitively, we expect the cost of a computation to increase as we demand more accuracy of the result; indeed, a straightforward backward error analysis based on the $(1+\epsilon)$ property suggests that the more accuracy we require, the more precision we must carry, which of course will incur greater cost, so the accuracy and cost are related by a condition number. As the algorithms in appendix A show, however, we can often compute very accurate results without carrying higher precision (although we may be effectively simulating it). Moreover, in some cases, including some of the examples in chapter 4, we can bound the cost of a computation by a constant independent of a condition number, regardless of how much accuracy we seek. Here again, our results can only be obtained by relying on stronger hypotheses than the $(1+\epsilon)$ property; consequently, we see that the cost of computing a result depends not only on the accuracy required but also on the set of properties we wish to assume.

The problem of analyzing the cost of a computation belongs to the domain of complexity theory, but complexity theory has largely ignored the question of roundoff altogether and assumed exact computation. Consequently, much of that theory has focused on algorithms which have questionable numerical behavior. In order to derive more practical results from this field, then, we must introduce an appropriate model of the roundoff error in floating point computations. As we have seen, we need to consider that model carefully: problems such as deciding whether a given matrix is singular cannot be solved at all if we are only allowed to assume $(1+\epsilon)$, whereas they can be solved at moderate cost if we assume faithfulness and at even lower cost if we assume a binary correctly rounding arithmetic. Since the theoretical limitations we predict depend crucially on the hypotheses we make, we must be willing to consider a more categorical model of floating point than the $(1+\epsilon)$ property; although we can always derive that property as a consequence if need be, we cannot deduce stronger properties from it.

5.2 Implications for Programming Environments

The ideas presented in the preceding chapters raise many issues for the numerical programmer; some of these issues inevitably provoke heated debate in the scientific computation community which we cannot resolve here. We will merely attempt to defend our view that both programmers and those who create programming environments must adopt a more sophisticated view of floating point arithmetic by considering two of the more controversial topics: multiple precision computation and standards.

Multiple Precision Computation

As the examples in chapter 4 show, certain computational problems have the property that simply computing a result to a guaranteed accuracy costs less than the manipulations required to act sensibly on an inaccurate result; for this reason, a number of programmers have proposed methods for incorporating multiple precision arithmetic facilities into common programming environments. We briefly note some of these methods in order to highlight the difference between our approach to accurate computation and more conventional multiple precision schemes.

The most common method of performing multiple precision arithmetic is to use a software package which implements floating point arithmetic of a user-definable precision; the most well known example is Brent's MP package [10]. Smith [59] recently published a similar package which features more careful rounding, improved exception handling, and simpler routines for elementary functions (which are faster than Brent's for moderate precision computations). Both Brent's and Smith's packages represent multiple precision floating point numbers as arrays of integers and operate on them with integer arithmetic; consequently, they require conversion routines to be called whenever data in the machine's native floating point format is to be mixed with multiple precision floating point. Moreover, on many modern systems designed for high-performance scientific computation, integer arithmetic operations are not as greatly optimized as floating point arithmetic, so intensive computation with integers is not as efficient as a similar floating point computation.

Bailey [2] addresses this latter problem in his MPFUN package: he represents multiple precision floating point numbers as arrays of floating point numbers in the machine's normal format and uses floating point operations to manipulate multiple precision numbers. He also provides two sets of basic operations, one based on straightforward algorithms and intended for moderate precision computation, and another which takes advantage of asymptotically faster algorithms such as an FFT-based multiplication scheme to achieve better performance when extremely high precision (above about 1000 digits) is used. Bailey quotes computing times which show that his package is indeed much faster than Brent's, a fact which he attributes primarily to the use of floating point rather than integer arithmetic. Like Brent's package, however, MPFUN requires a subroutine call to perform each multiple precision arithmetic operation, and it also requires format conversions when multiple precision floating point and ordinary floating point operations are intermixed; for computations involving only a modest amount of computation with precision slightly higher than usual, the overhead of these operations can be significant.

While Bailey's package certainly provides an efficient way to compute with high precision, it differs from the methods we propose in several respects. These differences stem from an underlying difference between our view of roundoff error and the view implicit in traditional approaches to multiple precision computation. Traditionally, roundoff error is regarded as an inevitable error whose cumulative effect can be bounded in terms of the precision of the arithmetic (among other quantities) and hence made smaller by simply carrying more digits; consequently, Bailey's package is designed to enable a program to carry a large but predetermined number of digits efficiently. In contrast, we regard roundoff as an optional perturbation, committed at the program's discretion and presumably in a manner which is known not to sacrifice accuracy, for the purpose of reducing the size of intermediate quantities and thus reducing the cost of computing what would otherwise be an exact result. The algorithms in appendix A are thus designed to compute exact sums and products, represented in a form which lends itself to subsequent rounding as determined appropriate by the program. In particular, the way in which a result is rounded can be decided *a posteriori* on the basis of the exact result, rather than by choosing a precision in advance. For this reason, we refer to the algorithms in appendix A as providing "arbitrary" precision arithmetic capabilities as opposed to conventional multiple precision.

One can of course compute exact sums and products using Bailey's package by simply carrying enough digits, but the number of digits needed to compute an exact sum can be quite large if the summands have very different magnitudes. For example, if we wish to compute a line segment intersection by the obvious algorithm using Bailey's MPFUN package and IEEE 754 arithmetic, we may need to carry up to 11 words of precision in order to guarantee that we obtain the same result as the algorithm presented in chapter 4; if the single and double precision operations in that algorithm were replaced with double and quadruple precision, respectively, then to compute to comparable accuracy with Bailey's package we would need up to 86 words. We need this high precision to avoid the loss of information when quantities near the underflow limit are added to ones close to the overflow limit; due to its sparsity, an expansion can achieve comparable accuracy with only a few components. (Note, though, that by using an analog of the double precision summation algorithm of chapter 4 with higher precision implemented using Bailey's package, we could reduce the precision requirements quoted above to 6 or 14 words of precision, respectively.)

On the basis of these differences regarding the treatment of roundoff, we can understand some of the practical differences between Bailey's package and the algorithms in appendix A. For example, in the examples we have presented in previous chapters, a typical computation involves performing a short series of exact products and sums to produce an intermediate result which can then safely be rounded to working precision or perhaps two or three times working precision. Correspondingly, our algorithms are quite efficient for working with numbers of small size (i.e., two or three components), which may consist either of numbers with only a modest precision or sparse numbers with high precision which result from a short computation. Since Bailey's package represents multiple precision numbers in a form which cannot easily be rounded or manipulated directly by the program except through the MPFUN routines themselves, a typical application of his package would likely carry many intermediate quantities to high precision throughout the computation. His algorithms are thus designed to be very efficient for working with numbers of large size (i.e., requiring many words of storage).

The difference between these two views of roundoff error can also be expressed in terms of paradigms for numerical programming. In a traditional paradigm, one performs a roundoff error analysis for a given algorithm, determines the precision necessary to obtain a sufficiently small error bound, then implements the algorithm by transliterating each formula into a programming language such as Fortran and declaring each variable to have the appropriate precision. Bailey's package fits naturally into this approach: he provides a translator which enables a programmer to write ordinary Fortran code with a few special "comments" which declare the precisions associated with certain variables. His translator then substitutes the appropriate calls to an MPFUN subroutine for each arithmetic operation involving multiple precision variables. (Crary [15] describes a similar translator which can be used with other multiple precision packages such as Brent's MP package.) Note that in this paradigm, each different precision of arithmetic functions as a different data type with implicit conversions introduced when numbers of different precisions are mixed. In fact, different precisions can correspond to completely different arithmetics: in Bailey's package, for example, single and double precision correspond to the native machine's arithmetic, but higher precisions, though implemented using the native arithmetic as a basis, correspond to a radix 2^{24} floating point format. (In computer science terminology, the arithmetics corresponding to different precisions are *orthogonal*.)

In contrast, our approach is based on a paradigm in which one first analyzes the algorithm to determine accuracy requirements for each intermediate result, determines the simplest way to insure those requirements, then writes a program which may not resemble the original algorithm textually, but rather attempts to simulate the same computations as the exact algorithm, at least up to the necessary accuracy. In order to compute to a specified accuracy, we need to compute intermediate quantities in a way that is essentially different from merely carrying high precision; instead, our algorithms compute what amount to working precision approximations augmented by smaller terms which recover all or part of the roundoff that would otherwise be lost. As we have noted above, this form can be regarded as a sparse representation of a number, as compared with the dense form most multiple precision arithmetics use. Consequently, our algorithms cannot be characterized as computing to a fixed, multiple precision. Moreover, although the number of components in an expansion can be viewed as an approximate measure of precision, our programming paradigm is fundamentally based on only one precision of arithmetic, so that multiple component numbers and the algorithms which act upon them are indistinguishable from ordinary working precision numbers and computations. When greater accuracy is desired, we do not supplant the machine's arithmetic with new data types and operations but merely supplement the program with additional steps to compute the same results more carefully.

Of course, one could implement a conventional multiple precision software package based on the algorithms in appendix A. Such a package would have the advantage that multiple precision numbers are represented as obvious extensions of working precision numbers, and hence both can be intermixed with no conversion. Another advantage of working with expansions is that simple operations such as the sum-and-roundoff computation and summation algorithms of chapter 4 can be implemented inline, avoiding the overhead of subroutine calls and allowing the compiler somewhat more freedom to optimize computations. (We admit that a clever compiler could optimize across subroutine calls or substitute inline code automatically. On the other hand, as we will see below, our algorithms depend crucially on certain compilers not to optimize too much anyway, so for simple computations, inline coding appears to be advantageous.) Nevertheless, for extended computations with more than a slight increase in precision, Bailey's MPFUN routines are far more efficient.

Bailey's package also has a significant advantage in portability. While our algorithms rely on very specific properties of floating point arithmetic which are not shared by every machine encountered in practice, the MPFUN package relies primarily on the fact that most floating point operations are exact when the operands are sufficiently small integers, a property satisfied by nearly all arithmetics. Of course, this fact cannot be deduced solely from the $(1+\epsilon)$ property either, so evidently the MPFUN routines provide still more examples of algorithms which rely on stronger properties of floating point arithmetic. (Faithfulness would be a sufficient property to prove their correctness, for example.)

Along a somewhat different line, we note the methods for accurate computation proposed by Kulisch and Miranker and their colleagues (see [41] and references therein), who show that with directed roundings and a means of evaluating an inner product to working precision, one can solve a wide variety of numerical problems to guaranteed accuracy. The primary technique of their approach is iterative refinement, which, as we noted in connection with Böhm's polynomial evaluation algorithm, can be regarded in some ways as comparable to our arbitrary precision methods. Since iterative refinement amounts to solving a system of linear equations whose right hand side is the residual error, the process can break down in the presence of degeneracies which render the system inconsistent. By recognizing that inconsistency, however, they may be able to substitute a different problem which does not exhibit the degeneracy. In either case, their approach seems to be subject to more numerical difficulty (such as the unnecessary overflow in Böhm's algorithm) than a straightforward application of multiple precision. Hence, while they show that a few simple primitive operations suffice to solve many problems, they do not attempt to discount the possibility that more efficient ways to solve the same problems could be formulated using the same primitives.

Portability and Standards

For the numerical programmer, the results of the preceding chapters once again raise the painful issue of portability. As we noted in chapter 2, not all floating point arithmetics possess the properties upon which our results are based, so our programs will not work as advertised on every machine (although they might be modified to do so at significantly greater cost). Since the programs have obvious practical value, the programmer wishing to use them faces a dilemma: should he sacrifice portability to achieve guaranteed accuracy, modify the programs to be portable but unnecessarily expensive, or eschew the programs altogether? Ideally, the resolution of such dilemmas is the goal of standardization. In practice, the diversity of experience, understanding, and opinion concerning floating point computation renders standards difficult to agree upon, much less rely upon.

The IEEE 754-1985 Standard for Binary Floating Point Arithmetic [30] provides an excellent basis for the standardization of floating point computations: it specifies binary, correctly rounding arithmetic (at least in the default rounding mode), which therefore satisfies all of the properties we have mentioned, and it is being widely adopted by the manufacturers of many of the computers most commonly used in scientific computation. (Apparently even Cray Research, Inc., has announced that it intends to begin producing machines which "more or less" conform to the standard.)

Of course, not all machines conform to the IEEE 754 standard, and that fact has prompted the recent proposal of another standard, the Language Independent Arithmetic Standard (LIAS) [32], which purports to specify all that a programmer need assume about floating point arithmetic to write portable numerical software. Of course, since the LIAS specifies a faithful arithmetic, it must exclude current Crays, and hence cannot be entirely as helpful as its authors intend. In fact, the LIAS appears to be potentially harmful, since it demands faithfulness yet does not require any of the further properties we have listed in chapter 2 that all existing faithful arithmetics possess; consequently, programs which must be written under that standard's assumptions alone cannot be as economical as ours. (For example, none of the simplifications to the basic sum-and-roundoff computation of chapter 2 would be permissible.)

Still another surprise awaits the author of portable numerical software: even IEEE 754 standard-conforming hardware does not necessarily guarantee correctly rounded arithmetic in a high-level programming environment. Many processors which provide floating point operations use an internal stack to record the results of intermediate computations, and values on that stack are kept in a floating point format with greater precision than the format customarily used to store floating point numbers in main memory. As a result, when a program contains a statement of the form "$a := b + c$", the compiler produces machine instructions which perform the following steps: (i) fetch the value of b and push it onto the top of the stack, at the same time extending it to the internal precision; (ii) fetch the value of c and add it to the value at the top of the stack, rounding the result to the internal precision; and (iii) store the new value at the top of the stack to the memory location of a, rounding it to the precision specified for a. Hence, the value actually assigned to a is obtained by rounding twice rather than once, and if the first rounding happens to produce a number in internal precision which is exactly halfway between numbers representable in the memory precision, the round-to-even rule may well dictate that the value assigned to a differ from what would have been computed had that value been rounded only once.

Of course, a reasonable programmer might expect, in accordance with the IEEE standard, that a should be rounded to the nearest number representable with the specified precision; yet although it is not, we can accuse neither the machine nor the compiler of failing to conform to the standard. The compiler may be excused on the grounds that "Some languages place the results of intermediate calculations in destinations beyond the user's control. Nonetheless, this standard defines the result of an operation in terms of that destination's format and the operands' values." [30, p. 7] Since results are initially left on the floating point stack in the internal format, the standard specifies that they be rounded to that format. Moreover, although some processors can be set to round intermediate results to double precision instead, this setting would no longer comply with the requirement that "An implementation should not provide operations that combine [higher precision] operands to produce a [lower precision] result ..., with only one rounding." [30, footnote, p. 10] That is, since the first fetch operation converts its operand to internal precision, any subsequent arithmetic operation must also round its result to internal precision.

Unfortunately, although twice-rounded arithmetic is faithful, it can violate property A1 depending on the precisions involved.[1] Consequently, the simplest versions of the sum-and-roundoff computation cannot be used in many environments which round twice. Worse yet, many of the compilers for machines with internal stacks attempt, unless explicitly told otherwise, to optimize computations by continuing to substitute a value on the stack for a named variable in a program, even though the value actually assigned to that variable differs (due to rounding) from the value on the stack: such compilers can wreak even more havoc on computations by, for example, translating a statement of the form "if $y \neq 0$ then $z := x/y$" into code which results in division by zero. Programs such as ours could not possibly be proved correct in a programming environment that unfathomable, since there is no way to know whether the value substituted for a variable is the same as that last assigned to it. (Farnum [20] and Goldberg [24] note this and other potential problems which compiler writers ideally should avoid. Their advice appears to be too often ignored, however, judging from Cody's account [14] of the prestidigitation required to make his MACHAR program produce correct results on a variety of machines.)

The only remedy for problems such as these seems to be to define the notion of the

[1] For example, in IEEE 754 arithmetic rounded first to 64 and then to 53 significant bits, the sum of $2^{52}+1$ and $.5-2^{-54}$ rounds first to $2^{52}+1.5$, which then rounds to $2^{52}+2$ by the round-to-even rule for halfway cases. The roundoff is then $.5+2^{-54}$, which is not representable in 53 bits. One can prove, though, that a single rational operation or square root with n-bit operands rounded first to p bits and subsequently to n bits will yield the same value as if the operation were rounded to n bits directly *provided* $p \geq 2n+1$ (see Goldberg [23]).

"destination" of an arithmetic operation within each programming language standard so as to close the loophole mentioned above. Then the IEEE standard would reflect the true behavior of the programming environment as seen by the programmer, thus making it possible to write a program which exactly specifies the computations to be performed. Since many hardware implementations of floating point can only provide that behavior when they are used in a rather inefficient way, programmers will likely have to work much harder to guarantee the correctness of their programs for some time to come. They could be aided in that task by the publication of complete documentation, much like the LIAS proposes though in greater detail, which would at least theoretically enable a person to predict precisely how a given program would perform in a given programming environment; at present, no one seems to be sufficiently motivated to undertake the formidable task of compiling that documentation.

Conclusions

The people who create programming language standards rely on programmers to define those aspects of a programming environment that deserve standardization; similarly, the people who document computer systems rely on programmers to define those aspects that require documentation. So long as numerical programmers do not expect more from floating point arithmetic than the $(1 + \epsilon)$ property can guarantee, they will not demand standards or documentation which can provide more, and manufacturers will continue to cite that lack of demand as a sufficient reason to build systems which cannot guarantee more. In order to insure even the least degree of portability for numerical software, then, programmers must take responsibility for insisting on more reliable and predictable behavior from floating point arithmetic.

Although the success of scientific computation depends crucially upon the reliability and predictability of floating point arithmetic, misconceptions about floating point computation continue to propagate. Only by cultivating a more mature understanding of floating point arithmetic can we accommodate the diverse backgrounds, needs, and interests of numerical analysts, computation theorists, computer scientists, programmers, and computer system manufacturers without provoking a crisis in which the computations which will play the most important role in our future do not yield to an analysis of their reliability and cost effectiveness. This dissertation demonstrates that such an understanding need not be expensive to acquire: one need only be cognizant of a few simple but important properties. The price we might pay for lack of such an understanding is more difficult to estimate. As a start, we might consider the recent report [21] on a Scud missile attack on Dhahran, Saudi Arabia, during the 1991 Persian Gulf war, in which a simple floating point roundoff error led to the failure of a Patriot missile defense system to correctly track a Scud missile (which subsequently hit an Army barracks, killing 28 people). One can only speculate how that bit of history might have differed had the programmers of the Patriot's software been endowed with a more sophisticated view of floating point arithmetic.

Bibliography

[1] Adams, D., A Stopping Criterion for Polynomial Root Finding, *Comm. ACM* **10** (1967), 655–658.

[2] Bailey, D., A Portable High Performance Multiprecision Package, RNR Technical Report RNR-90-022, NASA Ames Research Center, 1992.

[3] Bareiss, E., Sylvester's Identity and Multistep Integer-Preserving Gaussian Elimination, *Math. Comp.* **22** (1968), 565–578.

[4] Bauer, F., Computational Graphs and Rounding Error, *SIAM J. Num. Anal.* **11** (1974), 87–96.

[5] Ben-Or, M., Lower Bounds for Algebraic Computation Trees, in *Proc. 15th ACM Symposium on the Theory of Computing*, ACM Press, New York, 1983.

[6] Blum, L., M. Shub, and S. Smale, On a Theory of Computation and Complexity over the Real Numbers: NP-completeness, Recursive Functions, and Universal Machines, *Bull. Amer. Math. Soc.* **21** (1989), 1–46.

[7] Bohlender, G., Floating-Point Computation of Functions with Maximum Accuracy, *IEEE Trans. Comput.* **C-26** (1977), 621–632.

[8] ——, What Do We Need Beyond IEEE Arithmetic?, in C. Ullrich, Ed., *Computer Arithmetic and Self-Validating Numerical Methods*, Academic Press, Boston, 1990.

[9] Böhm, H., Evaluation of Arithmetic Expressions with Maximum Accuracy, in U. Kulisch and W. Miranker, Eds., *A New Approach to Scientific Computation*, Academic Press, New York, New York, 1983.

[10] Brent, R., A Fortran Multiple Precision Arithmetic Package, *ACM Trans. Math. Soft.* **4** (1978), 57–70.

[11] Brown, W. S., A Simple but Realistic Model of Floating-Point Computation, *ACM Trans. Math. Soft.* **7** (1981), 445–480.

[12] Carter, R., Y-MP Floating Point and Cholesky Factorization, *Intl. J. High Speed Comput.* **3** (1991), 215–222.

[13] Clemmesen, M., Interval Arithmetic Implementations Using Floating Point Arithmetic, *SIGNUM Newsletter* **19** (1984), 2–8.

[14] Cody, W., Algorithm 665. MACHAR: A Subroutine to Dynamically Determine Machine Parameters, *ACM Trans. Math. Soft.* **14** (1988), 303–311.

[15] Crary, F. D., A Versatile Precompiler for Nonstandard Arithmetics, *ACM Trans. Math. Soft.* **5** (1979), 204–217.

[16] Dekker, T., A Floating-Point Technique for Extending the Available Precision, *Numer. Math.* **18** (1971), 224–242.

[17] ——, Correctness Proof and Machine Arithmetic, in Fosdick, L., ed., *Performance Evaluation of Numerical Software*, North-Holland, Amsterdam, 1979.

[18] Demmel, J., Underflow and the Reliability of Numerical Software, *SIAM J. Sci. Stat. Comput.* **5** (1984), 887–919.

[19] Dobkin, D., and D. Silver, Recipes for Geometry and Numerical Analysis, in *Proc. Fourth Annual Symposium on Computational Geometry*, Association for Computing Machinery, New York, New York, 1988.

[20] Farnum, C., Compiler Support for Floating-Point Computation, *Software: Practice and Experience* **18** (1988), 701–709.

[21] GAO/IMTEC-92-26 Report, Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia.

[22] Gentleman, W., and S. Marovich, More on Algorithms That Reveal Properties of Floating Point Arithmetic Units, *Comm. ACM* **17** (1974), 276–277.

[23] Goldberg, D., Computer Arithmetic, Appendix A in D. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, San Mateo, Calif., 1990.

[24] ——, What Every Computer Scientist Should Know About Floating-Point Arithmetic, *ACM Computing Surveys* **23** (1991), 5–48.

[25] Gu, M., and S. Eisenstat, A Stable and Efficient Algorithm for the Rank-one Modification of the Symmetric Eigenproblem, Yale University Research Report YALEU/DCS/RR-916, 1992.

[26] Güting, R., Polynomials with Multiple Zeroes, *Mathematika* **14** (1967), 181–196.

[27] Henrici, P., *Elements of Numerical Analysis*, Wiley and Sons, New York, 1964.

[28] Higham, N., The Accuracy of Floating Point Summation, Numerical Analysis Report, University of Manchester, England, 1991.

[29] Holm, J., Floating-Point Arithmetic and Program Correctness Proofs, Ph.D. thesis, Cornell University, 1980.

[30] IEEE 754-1985 Standard for Binary Floating-Point Arithmetic, Institute of Electrical and Electronics Engineers, New York, 1985.

[31] IEEE 854-1987 Standard for Radix-Independent Floating-Point Arithmetic, Institute of Electrical and Electronics Engineers, New York, 1987.

[32] ISO/IEC CD 10967-1:1992 Information Technology – Language Independent Arithmetic – Part 1: Integer and Floating Point Arithmetic, Committee Draft Version 4.0, ISO/IEC JTC1/SC22/WG11, 1992.

[33] Kahan, W., Further Remarks on Reducing Truncation Errors, *Comm. ACM* **8** (1965), 40.

[34] ——, If At First You Don't Succeed, Try, Try, Try Again ... How Hard?, unpublished notes.

[35] ——, A Survey of Error Analysis, in C. V. Freeman, Ed., *Proc. IFIP Cong. 1971*, vol. 2, North-Holland, Amsterdam, 1972.

[36] ——, Paradoxes in Concepts of Accuracy, lecture notes from Joint Seminar on Issues and Directions in Scientific Computation, U. C. Berkeley, 1989.

[37] ——, Analysis and Refutation of the LCAS, *ACM SIGNUM Newsletter* **26** (1991), 2–15.

[38] ——, oral communication.

[39] Kahan, W., and B. N. Parlett, Can You Count on Your Calculator?, Electronics Research Laboratory Technical Report UCB/ERL M77/21, 1977.

[40] Knuth, D., *The Art of Computer Programming*, vol. 2 (2nd ed.), Addison-Wesley, Reading, Mass., 1981.

[41] Kulisch, U., and W. Miranker, *Computer Arithmetic in Theory and Practice*, Academic Press, New York, New York, 1981.

[42] Leuprecht, H., and W. Oberaigner, Parallel Algorithms for the Rounding-Exact Summation of Floating-Point Numbers, *Computing* **28** (1982), 89–104.

[43] Linnainmaa, S., Analysis of Some Known Methods of Improving the Accuracy of Floating-Point Sums, *BIT* **14** (1974), 167–202.

[44] ——, Software for Doubled-Precision Floating-Point Computations, *ACM Trans. Math. Soft.* **7** (1981), 272–283.

[45] Lohner, R., Precise Evaluation of Polynomials in Several Variables, in U. Kulisch and H. Stetter, Eds., *Scientific Computation with Automatic Result Verification*, Springer-Verlag, Wien, 1988.

[46] Malajovich, G., Computability in Dynamical Systems, preprint.

[47] Malcolm, M., Algorithms to Reveal Properties of Floating Point Arithmetic, *Comm. ACM* **15** (1972), 949–951.

[48] Mesztenyi, C., and C. Witzgall, Stable Evaluation of Polynomials, *J. Res. Nat. Bur. Stds.* **71B** (1967), 11–17.

[49] Milenkovic, V., Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic, *Artificial Intelligence* **37** (1988), 377–401.

[50] ——, Double Precision Geometry: A General Technique for Calculating Line and Segment Intersections Using Rounded Arithmetic, in *Proc. Thirtieth Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Los Alamitos, Calif., 1989.

[51] Møller, O., Quasi Double Precision in Floating-Point Addition, *BIT* **5** (1965), 37–50.

[52] Ottman, T., G. Thiemt, and C. Ullrich, Numerical Stability of Simple Geometric Algorithms in the Plane, in E. Børger, Ed., *Computation Theory and Logic*, Springer-Verlag, Berlin, 1987.

[53] ——, On Arithmetical Problems of Geometric Computations in the Plane, in U. Kulisch and H. Stetter, Eds., *Scientific Computation with Automatic Result Verification*, Springer-Verlag, Wien, 1988.

[54] Pichat, M., Correction d'une Somme en Arithmetique à Virgule Flottante, *Numer. Math.* **19** (1972), 400–406.

[55] Preparata, F., and M. Shamos, *Computational Geometry: an Introduction*, Springer-Verlag, New York, 1985.

[56] Priest, D., Algorithms for Arbitrary Precision Floating Point Arithmetic, in P. Kornerup and D. Matula, Eds., *Proc. 10th Symposium on Computer Arithmetic*, IEEE Computer Society Press, Los Alamitos, Calif., 1991.

[57] Rice, J., Experiment on Gram-Schmidt Orthogonalization, *Math. Comp.* **20** (1966), 325–328.

[58] Smale, S., Some Remarks on the Foundations of Numerical Analysis, *SIAM Review* **32** (1990), 211-220.

[59] Smith, D., Algorithm 693: A FORTRAN Package for Floating-Point Multiple-Precision Arithmetic, *ACM Trans. Math. Soft.* **17** (1991), 273–283.

[60] Sorensen, D., and P. Tang, On the Orthogonality of Eigenvectors Computed by Divide-and-Conquer Techniques, *SIAM J. Num. Anal.* **28** (1991), 1752–1775.

[61] Sterbenz, P., *Floating-Point Computation*, Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

[62] van Wijngaarden, A., Numerical Analysis as an Independent Science, *BIT* **6** (1966), 68–81.

[63] Wichmann, B., Towards a Formal Specification of Floating Point, *Computer J.* **32** (1989), 432–436.

[64] Wilkinson, J., *Rounding Errors in Algebraic Processes*, Prentice-Hall, Englewood Cliffs, New Jersey, 1964.

[65] ——, Modern Error Analysis, *SIAM Review* **13** (1971), 548–568.

[66] Wirth, N., *Systematic Programming: An Introduction*, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

Appendix A

Algorithms for Arbitrary Precision Floating Point Arithmetic

In this appendix, we present algorithms which use the basic sum-and-roundoff computation of chapter 2 to perform arbitrary precision arithmetic using only faithful fixed precision floating point arithmetic operations. These algorithms were first published in [56]; we include them here for completeness.

A.1 Arbitrary Precision Arithmetic

We view arbitrary precision arithmetic in the following context. Let $F_{\beta,t}$ denote the set of t-digit, radix β floating point numbers and F_β the subring of $\mathbf{Q}$ generated by $F_{\beta,t}$. Note that each $x \in F_\beta$ may be written as a finite sum $x = \sum x_i$ with $x_i \in F_{\beta,t}$. In particular, if $x \neq 0$ we can choose the x_i satisfying a *non-overlapping condition*, namely that $\mathrm{ulp}(x_i) \geq \beta^t \mathrm{ulp}(x_j)$ for $i < j$. (In other words, if $i < j$ then the most significant digit of x_j is at least one order of magnitude smaller than the least significant digit of x_i.) Such an expression is called a *t-digit expansion* for x; each term of the sum is a *component* of the expansion. A t-digit expansion for zero consists of a single zero component. Our goal is to show that all arithmetic operations over the ring F_β may be computed using only t-digit floating point arithmetic on components of t-digit expansions.

We use the term "arbitrary precision" rather than "multiple precision" to describe the arithmetic performed by the algorithms below, since "multiple precision" connotes a floating point arithmetic of fixed precision wider than the usual working precision, whereas the arithmetic derived from our algorithms has no fixed precision. (Strictly speaking, it does not even qualify as a floating point arithmetic under the definition of section 2.1; the use of the term "floating point" in the title of this appendix and the paper [56], though perhaps misleading, is intended to reflect the fact that our algorithms could be used to simulate conventional multiple precision floating point.) Instead, the precision of our arithmetic is "arbitrary" in the sense that the algorithms below compute *exact* sums and products in the ring F_β which are therefore obviously accurate to any precision; moreover, while our division algorithm computes an approximate quotient accurate to a specified number of components, the algorithm can be modified to compute a quotient to any accuracy needed to satisfy a computable error criterion.

In the results which follow, we state the cost of our algorithms as the maximum number of t-digit floating point arithmetic operations performed; note that the number of operations required in the sum-and-roundoff algorithm is at most eleven and may be reduced to as few as six if a simplified version may be used. (Here and below we count a comparison of two numbers as one floating point operation, since comparison implies subtraction. We also count the absolute value operation as one t-digit floating point operation, namely a comparison against zero. One may regard this as pessimistic, since the sign of a floating point number is represented in only one bit. On the other hand, separate counts for each type of operation may easily be obtained from the algorithms presented; we have chosen to combine the counts of different operations only for simplicity.)

A.2 Addition

The addition algorithm we present is a simple variation of the classical algorithm in which the radix points are aligned and digits are summed pairwise from right to left. The difference is that we are summing components rather than digits, and we regard the error of the computed sum as the "sum" of the components and the computed sum itself as a "carry". Since we can only add two components at once, we always add the carry of the last addition to the next smaller component not yet added. (In effect, we are merge-sorting the components of the two expansions by increasing magnitude and adding in this order.) The exact sum is then the sum of the final carry and the error terms from each preceding addition. Since the error terms need not satisfy the non-overlapping condition for expansions, we must renormalize their sum to obtain the result as a t-digit expansion.

We present the addition and renormalization algorithms separately since we will reuse the renormalization process later. To facilitate the proofs we need one more definition: for $0 \leq d < t$, a finite sequence $x_1, x_2, \ldots, x_n$ of t-digit floating point numbers is said to *overlap by at most d digits* if for each $j = 1, 2, \ldots, n-1$ there exist $i \leq j$ and k such that $x_1, \ldots, x_{i-1}$ are integer multiples of β^k, x_i is an integer multiple of β^{k-d}, $x_{i+1}, \ldots, x_j$ are all zero, and $|x_{j+1}| < \beta^k$. (Loosely speaking, this condition says that if the x_i were written in positional notation, the significant digits of any two successive non-zero terms would coincide in at most d digit positions, and moreover, no three terms would mutually coincide in any position. For example, in four digit decimal floating point, the sequence $12340, 5678, 9.123$ overlaps by at most three digits, but the sequence $12340, 5678, 91.23$ does not because all three numbers coincide in the tens place.) Note in particular that if $\beta^{k-1} \leq |x_1| < \beta^k$ and the sequence $x_1, \ldots, x_n$ overlaps by at most d digits then $|\sum x_j| < \beta^{k+1}$; that is, the sum can carry over to at most one larger place than the largest term. Note also that the components of a t-digit expansion do not overlap at all (i.e., they overlap by at most zero digits).

Proposition: Let $x = \sum_{i=1}^{n} x_i$ be a t-digit expansion with n components and $y = \sum_{i=1}^{m} y_i$ a t-digit expansion with m components. If the arithmetic is faithful then the following algorithm computes a sequence $e_1, \ldots, e_{n+m}$ which overlaps by at most one digit and satisfies $x + y = \sum e_j$.

Algorithm: (Addition)

procedure add($n, x_1, \ldots, x_n, m, y_1, \ldots, y_m$)
begin
$i := n$, $j := m$
if $|x_i| < |y_j|$

while $i > 1$ and $|x_{i-1}| \leq |y_j|$
$e_{i+j} := x_i$, $i := i - 1$
else if $|x_i| > |y_j|$
while $j > 1$ and $|y_{j-1}| \leq |x_i|$
$e_{i+j} := y_j$, $j := j - 1$
$a := x_i$, $b := y_j$
while $i > 1$ or $j > 1$
(c, e_{i+j}) := sum_and_error(a, b)
$a := c$
if $i = 1$ or ($j > 1$ and $|y_{j-1}| < |x_{i-1}|$)
$b := y_{j-1}$, $j := j - 1$
else
$b := x_{i-1}$, $i := i - 1$
(c, e_2) := sum_and_error(a, b)
$e_1 := c$
return $n + m, e_1, \ldots, e_{n+m}$
end

PROOF: Clearly $x + y = \sum_{j=1}^{n+m} e_j$. To prove that the e_j overlap by at most one digit, we proceed by induction on the number of times the algorithm calls sum_and_error. In particular, we show that after each execution of sum_and_error there exists k such that $|e_{i+j}| < \beta^k$ and $c, x_1, \ldots, x_{i-1}, y_1, \ldots, y_{j-1}$ are all integer multiples of β^{k-1} with all but one being multiples of β^k. The proposition will follow easily.

First observe that prior to the first call to sum_and_error, we have $i = n$ or $j = m$ or both. If $i = n$ but $j < m$ then the final loop starts with $e_{n+j+1}, \ldots, e_{n+m}$ equal to the last $m - j$ components of y, which do not overlap by assumption. Since this case can occur only if $y \neq 0$, we may write $y_j = p\beta^u$ with $\beta^{t-1} \leq |p| < \beta^t$. Then $|x_n| \geq |y_j|$, so $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_j$ are all integer multiples of β^u, while $|e_{n+j+1}| = |y_{j+1}| < \beta^u$. A similar argument applies if $i < n$ and $j = m$. Hence in any case we may begin the induction assuming that the algorithm has entered the final loop; in particular, if $i > 1$ then $|x_{i-1}| > |y_j|$, whereas if $j > 1$ then $|y_{j-1}| > |x_i|$.

To start the induction, let $a = x_i$, $b = y_j$, and (c, e_{i+j}) denote the result returned from the first call to sum_and_error. If $c = 0$ then $e_{i+j} = 0$ and the initial case holds.

Otherwise write $c = q\beta^v$ with $\beta^{t-1} \leq |q| < \beta^t$ and note that $|e_{i+j}| < \beta^v$. Suppose $|a| \geq |b|$; the case $|a| < |b|$ is similar. Write $a = x_i = p\beta^u$ with $\beta^{t-1} \leq |p| < \beta^t$. Then $x_1, \ldots, x_{i-1}$ are multiples of β^{u+t}, and since $|y_{j-1}| > |x_i|$ then y_{j-1} is a multiple of β^u so $y_1, \ldots, y_{j-2}$ are also multiples of β^{u+t}. Since $u \geq v-1$, we find that $c, x_1, \ldots, x_{i-1}, y_1, \ldots, y_{j-1}$ are all multiples of β^{v-1} and all but y_{j-1} are multiples of $\beta^{v-1+t} \geq \beta^v$ so again the initial case holds. Thus the proposition is established for the first call to sum_and_error.

Now suppose we have returned from the end of the final loop. Then the current value of a is the previous value of c and the current value of b is one of $\{x_i, y_j\}$. Let c and e_{i+j} be the quantities calculated in this call to sum_and_error and note that $c = \sum_{k=i}^{n} x_k + \sum_{k=j}^{m} y_k - \sum_{k=i+j}^{n+m} e_k$. If $c = 0$ then $e_{i+j} = 0$ and we need only appeal to the induction hypothesis. Otherwise write $c = q\beta^v$ with $\beta^{t-1} \leq |q| < \beta^t$ and again note $|e_{i+j}| < \beta^v$. Assume $b = x_i$; the case $b = y_j$ is similar. Note that $|b| \leq |y_{j-1}|$ (otherwise we would have chosen $b = y_{j-1}$ in the previous iteration of the loop) and that $|b| \geq |y_j|$ (otherwise we would have chosen $b = x_i$ prior to the previous iteration). Write $b = p\beta^u$ with $\beta^{t-1} \leq |p| < \beta^t$. Then

$$\beta^{v+t-1} \leq |c| \leq \left|\sum_{k=i}^{n} x_k\right| + \left|\sum_{k=j}^{m} y_k\right| + \left|\sum_{k=i+j}^{n+m} e_k\right|$$

$$< \beta^{u+t} + \beta^{u+t} + \beta^{v+1}$$

where the bounds on the first two terms follow from the non-overlapping conditions on x and y and the inequality $|y_j| \leq |b| = |x_i|$, and the bound for the last term follows from the induction hypothesis. Consequently, $\beta^{v+t-1} < \beta^{u+t+1} + \beta^{v+1}$, so $v + t - 1 < \max(u + t + 2, v + 2)$. Since $t \geq 3$ we must have $v + t - 1 < u + t + 2$ so $v < u + 3$ and hence $v \leq u + 2$. Now $x_1, \ldots, x_{i-1}$ are multiples of β^{u+t} and thus of β^v. To complete the induction it suffices to show that y_{j-1} is a multiple of β^{v-1}; then $y_1, \ldots, y_{j-2}$ will be multiples of $\beta^{v-1+t} \geq \beta^v$. Write $y_{j-1} = r\beta^s$ with $\beta^{t-1} \leq |r| < \beta^t$ so that $|\sum_{k=j}^{m} y_k| < \beta^s$. Then the previous inequality becomes $\beta^{v+t-1} < \beta^{u+t} + \beta^s + \beta^{v+1}$.

Since $v \leq u+2$, if $s < u+t$ then we have $v+t-1 < u+t+1$ so $v < u+2$ and hence $v \leq u+1$, and since $|y_{j-1}| \geq |b|$ this implies y_{j-1} is a multiple of $\beta^u = \beta^{v-1}$ and the induction step follows. If instead $s \geq u+t$ then clearly $s \geq v-1$, so again y_{j-1} is a multiple of β^{v-1}.

From the above argument, we clearly must have either $c, x_1, \ldots, x_{i-1}, y_1, \ldots, y_{j-1}$ all multiples of β^v or else exactly one of $\{x_{i-1}, y_{j-1}\}$ is a multiple of β^{v-1} but not of β^v. Suppose as above that y_{j-1} is not a multiple of β^v. Then clearly $|y_{j-1}| < |x_{i-1}|$ so the next iteration of the loop will choose $b = y_{j-1}$. Consequently, either all of the quantities $e_1, \ldots, e_{i+j-1}$ are multiples of β^v or there exists $l \leq i+j-1$ such that $e_1, \ldots, e_{l-1}$ are multiples of β^v, e_l is a multiple of β^{v-1}, and $e_{l+1}, \ldots, e_{i+j-1}$ are zero. This completes the proof. ∎

PROPOSITION: Let $e_1, \ldots, e_n$ overlap by at most $t-2$ digits. If the arithmetic is faithful then the following algorithm computes an expansion $s = \sum_{j=1}^{m} s_j$ with $m \leq n$ such that $s = \sum e_j$.

ALGORITHM: (Renormalization)

```
procedure renorm( n, e_1, ..., e_n )
begin
c := e_n
for i := n − 1, n − 2, ..., 1
      ( c, f_{i+1} ) := sum_and_error( c, e_i )
f_1 := c
s_1 := f_1, k := 1
for j := 2, 3, ..., n
      ( c, d ) := sum_and_error( s_k, f_j )
      s_k := c
      if d ≠ 0
            l := k − 1, k := k + 1
            while l ≥ 1
                  ( c′, d′ ) := sum_and_error( s_l, s_{l+1} )
                  if d′ = 0
                        s_l := c′, l := l − 1, k := k − 1
                  else
                        l := 1
            s_k := d
```

return $k, s_1, \ldots, s_k$
end

PROOF: After the first for loop terminates, we clearly have $\sum f_j = \sum e_j$. Furthermore, we claim that the f_j do not overlap at all. We again argue by induction. Specifically, let c and f_{i+1} be the return values from a call to sum_and_error in the first loop. If $c = 0$ then $f_{i+1} = 0$ so the induction holds; otherwise write $c = q\beta^v$ with $\beta^{t-1} \leq |q| < \beta^t$ so that $|f_{i+1}| < \beta^v$. Also write $e_i = p\beta^u$ with $\beta^{t-1} \leq |p| < \beta^t$. Then since the e_j overlap by at most $t-2$ digits, we have $v \leq u+1$. Consequently, $e_1, \ldots, e_{i-1}$ are all multiples of $\beta^{u+2} \geq \beta^v$, so $f_1, \ldots, f_i$ must be multiples of β^v while $|f_{i+1}| < \beta^v$. This establishes the claim.

Proceeding to the rest of the algorithm, we note that the output $s_1, \ldots, s_k$ clearly satisfies $\sum_{j=1}^k s_j = \sum_{j=1}^n f_j$. We need only show that the output qualifies as an expansion. Induct on k: let c and d be the return values from a call to sum_and_error in the outer for loop. If $d = 0$ there is nothing to prove, so assume $d \neq 0$. Then also $s_k = c \neq 0$ so we may write $s_k = q\beta^v$ with $\beta^{t-1} \leq |q| < \beta^t$ and we have $|d| < \beta^v$. Since the f_i do not overlap, we also have $|d + \sum_{i>j} f_i| < \beta^v$. Therefore each subsequent call to sum_and_error in the outer for loop must yield $|c| \leq \beta^v$ with equality only if c is rounded (so that again $d \neq 0$). If we always have $|c| < \beta^v$ then clearly $|s_i| < \beta^v$ for $i > k$ so the induction holds. If instead we obtain $|c| = \beta^v$ at some later step, then by the induction hypothesis, upon completion of the predicate of the outer if statement, we still find $s_1, \ldots, s_k$ (for the value of k then in effect) is an expansion. In addition, we then have $|d| < \beta^{v-t}$, so again the induction holds, and the proof is complete. ∎

It follows from the proof above that sum_and_err is never called from the innermost while loop more than $n-1$ times. An easy operation count yields the following.

COROLLARY: Let x and y be t-digit expansions having n and m components, respectively. Then the composition of algorithms add and renorm computes a t-digit expansion $s = x + y = \sum_{i=1}^k s_i$ with $k \leq n + m$ using at most $49(n+m-1)+9$ t-digit floating point operations.

A.3 Multiplication

To compute the exact product of two t-digit expansions, we must guarantee that no significant digits are lost when the product of two components is computed in t-digit arithmetic. We accomplish this by splitting each component into a sum of either two or three numbers, each with fewer nonzero digits than the original components. We may then multiply these split components with no error, finally adding all the partial products and renormalizing to obtain a t-digit expansion. The method of splitting components was first proposed by Dekker [16], and we shall rely on the proof given by Linnainmaa [44]. To clarify the statements and proofs, we define the number of *leading nonzero digits* of a t-digit number x to be the smallest integer d such that $x = m\beta^k$ with $\beta^{d-1} \leq |m| < \beta^d$, or zero if $x = 0$.

Proposition: Let x be a t-digit floating point number. For $1 < k < t$, define $a_k = \beta^{t-k} + 1$. Provided the floating point arithmetic is faithful, the following algorithm computes t-digit numbers x' and x'' such that $x = x' + x''$, x' has at most k leading nonzero digits, and x'' has at most $t - k + 1$ leading nonzero digits.

Algorithm: (Splitting of a t-digit number)

procedure split(x, k)
begin
$y := \mathrm{fl}(a_k \times x)$, $z := \mathrm{fl}(y - x)$
$x' := \mathrm{fl}(y - z)$, $x'' := \mathrm{fl}(x - x')$
return x', x''
end

Proof: This is Theorem 1 in [44]. Linnainmaa actually states the theorem with an additional hypothesis which guarantees that in fact x'' has at most $t - k$ leading nonzero digits; in a remark following his proof, however, he shows that even if the additional hypothesis is not satisfied, x' will be computed with $k - 1$ leading nonzero digits and x'' will have $t - k + 1$ leading nonzero digits. ∎

We are now ready to give the multiplication algorithm. Note that we choose to split each component of the first expansion into two parts and each component of the second expansion into three parts. Dekker [16] shows that with binary correctly rounding arithmetic it suffices to split each factor into just two parts; Linnainmaa [44] gives other criteria under which splitting into two parts is sufficient. Consequently, the following algorithm can be improved in many cases. Note that we also do not sum the partial products into one net accumulator but instead separate them into groups (denoted $a^{(1)}, \ldots, a^{(6)}$) which are guaranteed to overlap by at most two digits so that they may be renormalized without first being processed through the add algorithm. We then accumulate these renormalized expansions into a smaller accumulator (b) which in turn is added to the overall product accumulator. This nesting of additions significantly reduces the coefficient of the largest order term in the cost estimate.

PROPOSITION: Let $x = \sum_{i=1}^{n} x_i$ be a t-digit expansion with n components and $y = \sum_{i=1}^{m} y_i$ a t-digit expansion with m components. Assume the arithmetic is faithful and that $t \geq 7$, and set $k_2 = \lfloor t/2 \rfloor + 1$ and $k_3 = \lfloor t/3 \rfloor + 1$. Then the following algorithm computes the product xy expressed as a t-digit expansion with at most $2nm$ components. The algorithm requires at most $98n^2m + 1049nm + 8m - 446n$ t-digit floating point operations.

ALGORITHM: (Multiplication)

procedure multiply($n, x_1, \ldots, x_n,\ m, y_1, \ldots, y_m$)
begin
for $i := 1, 2, \ldots, n$
    (x_i', x_i'') := split(x_i, k_2)
for $i := 1, 2, \ldots, m$
    (y_i', z) := split(y_i, k_3)
    (y_i'', y_i''') := split(z, k_3)
$p_1 := 0,\ k := 1$
for $i := 1, 2, \ldots, n$
    for $j := 1, 2, \ldots, m$
        $a_j^{(1)} := \mathrm{fl}(x_i' \times y_j')$
        $a_j^{(2)} := \mathrm{fl}(x_i' \times y_j'')$
        $a_j^{(3)} := \mathrm{fl}(x_i' \times y_j''')$
        $a_j^{(4)} := \mathrm{fl}(x_i'' \times y_j')$
        $a_j^{(5)} := \mathrm{fl}(x_i'' \times y_j'')$

$a_j^{(6)} := \mathrm{fl}(x_i'' \times y_j''')$
$(j, b_1, \ldots, b_j) := \mathrm{renorm}(m, a_1^{(1)}, \ldots, a_m^{(1)})$
$(l, c_1, \ldots, c_l) := \mathrm{renorm}(m, a_1^{(2)}, \ldots, a_m^{(2)})$
$(j, b_1, \ldots, b_j) := \mathrm{renorm}(\mathrm{add}(l, c_1, \ldots, c_l, j, b_1, \ldots, b_j))$
$(l, c_1, \ldots, c_l) := \mathrm{renorm}(m, a_1^{(3)}, \ldots, a_m^{(3)})$
$(j, b_1, \ldots, b_j) := \mathrm{renorm}(\mathrm{add}(l, c_1, \ldots, c_l, j, b_1, \ldots, b_j))$
$(l, c_1, \ldots, c_l) := \mathrm{renorm}(m, a_1^{(4)}, \ldots, a_m^{(4)})$
$(j, b_1, \ldots, b_j) := \mathrm{renorm}(\mathrm{add}(l, c_1, \ldots, c_l, j, b_1, \ldots, b_j))$
$(l, c_1, \ldots, c_l) := \mathrm{renorm}(m, a_1^{(5)}, \ldots, a_m^{(5)})$
$(j, b_1, \ldots, b_j) := \mathrm{renorm}(\mathrm{add}(l, c_1, \ldots, c_l, j, b_1, \ldots, b_j))$
$(l, c_1, \ldots, c_l) := \mathrm{renorm}(m, a_1^{(6)}, \ldots, a_m^{(6)})$
$(j, b_1, \ldots, b_j) := \mathrm{renorm}(\mathrm{add}(l, c_1, \ldots, c_l, j, b_1, \ldots, b_j))$
$(k, p_1, \ldots, p_k) := \mathrm{renorm}(\mathrm{add}(j, b_1, \ldots, b_j, k, p_1, \ldots, p_k))$
return $k, p_1, \ldots, p_k$
end

PROOF: Note that for each i both x_i', x_i'' have at most $\lfloor t/2 \rfloor + 1$ leading nonzero digits, and for each j, y_j', y_j'', y_j''' have at most $\lfloor t/3 \rfloor + 1$ leading nonzero digits. Since $t \geq 7$, each of the six products in the innermost for loop is computed exactly. By the non-overlapping condition for expansions together with the bounds on the number of leading nonzero digits of split components, we see that the six sequences $a_j^{(1)}, \ldots, a_j^{(6)}$, $j = 1, \ldots, m$ computed in the inner loop overlap by at most two digits. Hence the renormalization and addition procedures in the outer loop succeed without error, so the output satisfies $p = xy$. Note that for each i the sums accumulated in b form expansions with at most $2m$ components. Therefore p never exceeds $2nm$ components. A straightforward operation count completes the proof. ∎

The above multiplication algorithm is based on the classical algorithm of multiplying digits pairwise; hence, the number of multiplications is proportional to the product of the numbers of components in each expansion. In particular, if both expansions have n components, the number of basic multiply-and-add steps is $O(n^2)$. Algorithms which multiply n-digit numbers in fewer than $O(n \log^2 n)$ steps are known (see Knuth [40]), but we have elected not to use them for several reasons. Specifically, most of these algorithms improve upon the classical method only for very large n, while in practice we would expect to apply the above algorithm only to expansions with relatively few components (e.g., at most 39 for a typical implementation of binary floating point arithmetic using 64 bit storage; see the remarks on underflow in section 5 below). In addition, some of these algorithms are based on integer arithmetic, so that the input is assumed to consist of a contiguous string of n digits representing each multiplicand. Since we are given input in the form of t-digit expansions, the digits of which need not be contiguous, we cannot always benefit from the application of the techniques suggested.

Notice that although the total number of multiplications is only proportional to nm, the total cost is proportional to n^2m. This extra factor of n in the total cost arises from the cost of adding the partial products. Using slightly more intermediate storage, we can rearrange the order of the additions in the form of a binary tree and reduce this term to $nm \log n$, but we believe that reducing the cost to $O(nm)$ is not practical.

A.4 Division

Our division algorithm is analogous to the usual method of long division. In that algorithm, one takes the quotient of the most significant digits of the remaining dividend and divisor as a guess for the next digit of the overall quotient, and if the product of this guess with the divisor would be larger than the dividend, one chooses a smaller digit. For our algorithm, since we do not require the components of an expansion to have the same sign, we do not need to adjust the guess in this way. We simply take the quotient of the most significant components (as computed by t-digit arithmetic) as the next term of the quotient, and we show that once a sufficient number of terms have been computed, we can renormalize to express the quotient as an expansion.

PROPOSITION: Let $x = \sum_{i=1}^{n} x_i$ be a t-digit expansion with n components and $y = \sum_{i=1}^{m} y_i$ a t-digit expansion with m components, and suppose $y \neq 0$. Assume the arithmetic is faithful and $t \geq 9$; for $d > 0$ let $k = \lfloor (t-4)d/t \rfloor$. Then the following algorithm computes a t-digit expansion $q = \sum q_i$ with at most d components such that $|q - x/y| < \beta^{1-k}|x/y|$. The algorithm requires at most $98(d-1)^2 m + (d-1)(1255m + 49n - 479) + 7$ t-digit floating point operations.

ALGORITHM: (Division)

```
procedure divide( d, n, x_1, ..., x_n, m, y_1, ..., y_m )
begin
j^(1) := n
for i := 1, 2, ..., n
      e_i^(1) := x_i
for i := 1, 2, ..., d
      q_i := fl(e_1^(i) / y_1)
      if i < d
            ( k, f_1, ..., f_k ) := multiply( 1, q_i, m, y_1, ..., y_m )
            for l := 1, 2, ..., k
                  f_l := -f_l
            ( j^(i+1), e_1^(i+1), ... ) := renorm( add( k, f_1, ..., f_k, j^(i), e_1^(i), ... ) )
( k, s_1, ..., s_k ) := renorm( i, q_1, ..., q_i )
return k, s_1, ..., s_k
end
```

PROOF: The result is trivial if $x = 0$; for $x \neq 0$ it suffices to show that the sequence $q_1, \ldots, q_i$ computed in the main loop overlaps by at most four digits. Proceeding by induction on i, let q_i be the quantity computed in the i-th iteration of the loop for some i and write $q_i = p\beta^u$ with $\beta^{t-1} \leq |p| < \beta^t$. Note that $q_i = (e_1^{(i)}/y_1)(1+\epsilon)$, $f = q_i y = q_i y_1 (1+\delta)$ and $e_1^{(i)} = e^{(i)}(1+\eta)$ where $|\epsilon|, |\delta|, |\eta| < \beta^{1-t}$. Hence writing $\alpha = 1 - (1+\epsilon)(1+\delta)(1+\eta)$ we have $e^{(i+1)} = e^{(i)} - f = e^{(i)}\alpha$ and $|\alpha| < \beta^{3-t}$. Similarly, $e_1^{(i+1)} = e^{(i+1)}(1+\mu) = e^{(i)}\alpha(1+\mu) = e_1^{(i)}\alpha(1+\mu)(1+\eta)^{-1}$ and hence $q_{i+1} = q_i\alpha(1+\mu)(1+\nu)(1+\eta)^{-1}(1+\epsilon)^{-1}$ with $|\mu|, |\nu| < \beta^{1-t}$. Consequently, $|q_{i+1}| < |q_i|\beta^{4-t} < \beta^{u+4}$. Since $t \geq 9$, the induction hypothesis implies that $q_1, \ldots, q_{i-1}$ are multiples of $\beta^{u+t-4} > \beta^{u+4}$ and the proof is complete, save for an easy operation count. ∎

A.5 Further Remarks

We have expressed the cost of our algorithms as the number of floating point arithmetic operations required as a function of the number of components of the input expansions. Note that a typical number in F_β does not have a unique t-digit expansion; in fact, even the number of components is not uniquely determined. Since the cost of each operation depends on the number of components, we could in principle demand that we always compute *minimal* t-digit expansions. We choose not to do so, however, since this would require the development of an algorithm which finds a minimal expansion given an arbitrary one, and such algorithms are difficult to formulate in a machine-independent way. Instead we only require that an expansion satisfy the non-overlapping condition, i.e., that significant digits of successive components do not overlap. This requirement incurs additional cost in the form of the renormalization steps following each operation, but it provides a convenient form for subsequent rounding and usually leads to "nearly minimal" expansions. We conjecture that for many applications, the cost of producing minimal expansions outweighs the cost of working with nearly minimal ones. (In fact, for some applications we may save time by renormalizing even less frequently; our algorithms could easily be made more efficient in this respect, although some of the proofs would need to be refined.)

In the preceding algorithms, we have tacitly assumed that the exponent range of the arithmetic is unlimited, so that no overflow or underflow occurs. In fact, although it is possible for our distillation and multiplication algorithms to overflow when the exact results lie within representable range, such overflow can occur only when the inputs are already very close to the overflow threshold and is therefore not likely. On the other hand, since the trailing components of expansions quickly become very small, underflow imposes a certain limitation on the accuracy of these algorithms. (For example, IEEE standard double precision with a 64 bit storage format contains eleven bits of exponent and 53 bits of fraction. Hence the net exponent range is $2^{11} = 2048$, so the maximum number of representable components in an expansion is

$\lceil 2048/53 \rceil = 39$. Exact calculations can easily exceed this limit after several operations.) Nevertheless, we believe that in most applications, intermediate results may be rounded or rescaled so that underflow does not pose a serious problem. Even when underflow cannot be avoided, Demmel [18] points out that the cumulative effect of the resulting errors can often be estimated by extending the error analysis to include an underflow term whose magnitude is bounded by the underflow threshold, if underflow is gradual, or by the ratio of this threshold to the unit roundoff otherwise.

Finally we note that the techniques we have proposed for calculating to arbitrary precision can be used equally well to simulate multiple precision floating point or fixed point arithmetic. Specifically, if after each calculation we truncate the resulting expansion to a fixed number of components, then the relative error of each intermediate result will be bounded by a corresponding constant, as is true of ordinary floating point arithmetic. Alternatively, if we drop only trailing components whose magnitude is below a given threshold (as with underflow), then we can instead bound the absolute error of each operation, thereby emulating fixed point arithmetic. More importantly, in either case we may adjust the precision dynamically, calculating certain intermediate results to very high precision and others to lower precision, to obtain the desired accuracy in the solution.

Appendix B

An Example of the Inadequacy of $(1+\epsilon)$

In this appendix, we prove that the evaluation of the function

$$f_N(x) := \begin{cases} \frac{x^N-1}{x-1} & \text{if } x \neq 1 \\ N & \text{otherwise} \end{cases}$$

for a positive integer N and real x gives a result that is correct to at least half of the figures carried. As noted in chapter 2, this fact cannot be proved using only the $(1+\epsilon)$ property.

Proposition: Fix a positive integer $N \ll \beta^{t/2}$. If floating point multiplication is either correctly rounded or chopped and subtraction and division satisfy the $(1+\epsilon)$ property, then the following algorithm computes a value z satisfying $|z - f_N(x)|/|f_N(x)| \leq \sqrt{2N\beta^{2-t}} + O(N\beta^{1-t})$.

Algorithm:

```
procedure fsubN( x )
begin
if x = 1
      z := N
else
      compute y := x^N by recursive multiplication
      z := fl(fl(y − 1)/fl(x − 1))
return z
end
```

Proof: If $x = 1$ then $z = N = f_N(x)$, so assume $x \neq 1$. Note that $|z - f_N(x)|/|f_N(x)| = |z(x-1) - (x^N - 1)|/|x^N - 1|$. Since subtraction and division both satisfy the $1+\epsilon$ property, we can write $z = (y-1)(1+3\epsilon)/(x-1)$ for some $|\epsilon| < \beta^{1-t}$, where y is the computed value of x^N as in the algorithm. Hence writing $\gamma = |y - x^N|/|x^N - 1|$, we have $|z - f_N(x)|/|f_N(x)| \leq \gamma + 3\beta^{1-t}(1+\gamma)$. Therefore it suffices to show $\gamma < \sqrt{2N\beta^{2-t}} + O(N\beta^{1-t})$.

Suppose first that either $|x| \geq 1 + \sqrt{\beta^{-t}/2N}$ or $|x| \leq 1 - \sqrt{\beta^{-t}/2N}$. Since multiplication satisfies the $(1+\epsilon)$ property, we can write $y = x^N(1+N\delta)$ for some $|\delta| < \beta^{1-t}$, and thus $\gamma = |N\delta x^N|/|x^N - 1|$. But note that $|x^N|/|x^N-1| \leq 1/\sqrt{N\beta^{-t}/2} + O(1)$ so we have $\gamma < \sqrt{2N\beta^{2-t}} + O(N\beta^{1-t})$ and the proposition holds.

Now suppose $1 - \sqrt{\beta^{-t}/2N} < |x| < 1 + \sqrt{\beta^{-t}/2N}$. Write $w = |x| - 1$ and note that w must be a multiple of β^{-t} (and of β^{1-t} if $|x| > 1$) with $|w| < \sqrt{\beta^{-t}/2N}$. In particular, the numbers $1 + w, 1 + 2w, 1 + 3w, \ldots, 1 + Nw$ are all representable floating point numbers. Note further that for any positive integer $i < N$ we have $(1+iw)(1+w) = 1+(i+1)w+iw^2$ which exceeds the representable number $1+(i+1)w$ by $iw^2 < \beta^{-t}/2$. Thus the product $\mathrm{fl}((1+iw) \times (1+w))$ must be either chopped or correctly rounded to $1 + (i+1)w$, and therefore we must have $|y| = 1 + Nw$. Thus $\gamma = |y - x^N|/|x^N - 1| \leq N|w|/2 + O(N^2w^2) < \sqrt{N\beta^{-t}/8} + O(N\beta^{1-t})$. This completes the proof. ∎

Kahan and Parlett note that the same result holds when x^N is computed by several other means, including the formula $\exp(N \log x)$, provided both functions are reasonably accurate. They also exhibit a slight modification to the algorithm which gives a result correct in all but the last digit.