How to Read Floating Point Numbers Accurately

William D Clinger
University of Oregon

ABSTRACT

Consider the problem of converting decimal scientific notation for a number into the best binary floating point approximation to that number, for some fixed precision. This problem cannot be solved using arithmetic of any fixed precision. Hence the *IEEE Standard for Binary Floating-Point Arithmetic* does not require the result of such a conversion to be the best approximation.

This paper presents an efficient algorithm that always finds the best approximation. The algorithm uses a few extra bits of precision to compute an IEEE-conforming approximation while testing an intermediate result to determine whether the approximation could be other than the best. If the approximation might not be the best, then the best approximation is determined by a few simple operations on multiple-precision integers, where the precision is determined by the input. When using 64 bits of precision to compute IEEE double precision results, the algorithm avoids higher-precision arithmetic over 99% of the time.

The input problem considered by this paper is the inverse of an output problem considered by Steele and White: Given a binary floating point number, print a correctly rounded decimal representation of it using the smallest number of digits that will allow the number to be read without loss of accuracy. The Steele and White algorithm assumes that the input problem is solved; an imperfect solution to the input problem, as allowed by the IEEE standard and ubiquitous in current practice, defeats the purpose of their algorithm.

1. INTRODUCTION

It seems reasonable to assume that a floating point constant appearing in code or data, when read by a compiler or a standard input routine, will be converted into the floating point number that best approximates the constant. Most programming languages do not require this, however, nor does the *IEEE Standard for Binary Floating-Point Arithmetic* [IEEE85].

Standard practice is to settle for an easily computed approximation that is close but not necessarily closest [Coonen80]. For example, the IEEE standard specifies that the error introduced when converting from a decimal external representation to either the single or double precision internal representation, using round to nearest, shall be no more than .97 units in the least significant bit of the result. For the best approximation, the error is at most .5 units.

Steele and White have considered the problem of printing a floating point number using the fewest digits that will allow the number to be read back without loss of accuracy [Steele90]. Their algorithm assumes the input routine will always find the best approximation, and does not work without this assumption. In particular, their algorithm does not work under the weaker assumption of an IEEE-conforming input routine.

The problem considered here is the input problem: Given decimal scientific notation for a number, compute the best binary floating point approximation to the given number. This differs from the output problem considered by Steele and White because the internal representation of floating point numbers has fixed precision and is quantized, but the external representation has variable precision and is dense in the space of real numbers. This asymmetry makes the input and output problems very different.

As shown in Section 4, the input problem cannot be solved using fixed precision arithmetic. Subsequent sections develop a practical algorithm for this problem. The algorithm must occasionally work with very large precisions, but usually finds the best approximation through a floating point computation whose precision is a few bits greater than the precision required of the result.

The key idea is this. Consider the problem of computing the value of some function g, rounded to the nearest 10000, given an oracle that delivers the value of g rounded to the nearest 10. Given an input x, the obvious approach is to ask the oracle for the value of $g(x)$ to the nearest 10, and then to round that result to the nearest 10000. Unfortunately, this does not always work. If $g(x)$ is 11074996 and $g(y)$ is 11075004, for example, then the answers should be 11070000 and 11080000, respectively, but the oracle will deliver 11075000 for both x and y. This approach usually works, though. It fails only when the value delivered by the oracle ends in 5000.

To find the best approximation, the algorithm of this paper uses an extended precision to compute an approximation whose accuracy is significantly better than the accuracy required in the result. It then examines the low order bits of that intermediate result to bound the additional error that will be introduced when this particular value is rounded to the precision required of the final result. If the sum of the bounds for the intermediate approximation error and for the rounding error is less than the error allowed in the final result, then the rounded value must be correct. Otherwise the rounded value must be checked using arithmetic of greater precision.

By choosing a large enough extended precision, the probability that an even greater precision will be needed can be made as small as desired. For IEEE double precision numbers, an experiment reported in Section 9 found that IEEE extended precision is enough to find the best approximation over 99% of the time.

2. EXTERNAL AND INTERNAL RADIXES

For concreteness it is often appropriate to assume that the input is given in decimal scientific notation and that the output is a binary floating point number, but most results presented here hold for more general radixes. Let $\Delta \geq 2$ be the radix used to express the input, and let $\beta \geq 2$ be the radix of the floating point output. Δ is the *external* radix, and β is the *internal* radix. In practice Δ is usually 10 and β is usually 2 or 16.

Definition 1. [Matula70] Radixes Δ and β are *commensurable* if and only if both are integral powers of a common integral root. Equivalently, Δ and β are commensurable iff $\log_\beta \Delta$ is rational.

For example, 2 and 16 are commensurable radixes, as are 8 and 16. On the other hand 10 and 2 are incommensurable, as are 10 and 16. As shown in Section 4, the problem of computing the best floating point approximation is trivial if Δ and β are commensurable radixes. Most of this paper therefore assumes that Δ and β are incommensurable, and for simplicity the internal radix β is always assumed to be even.

For convenience "bit" will be used to refer to digits in the internal radix β, and "digit" will be used for digits in the external radix Δ.

3. FLOATING POINT NUMBERS

This paper deals exclusively with positive floating point numbers. Overflow and underflow are considered only in Section 8. Because the following definition refers to precision and not to a data structure, this paper considers an IEEE single precision number to be a 24-bit floating point number and an IEEE double to be a 53-bit floating point number.

Let n, the precision in bits of a floating point number, be a positive integer. For the purposes of this paper, an n-bit floating point number consists of an integer significand m and an integer exponent q with $0 < m < \beta^n$, representing the value $m \times \beta^q$. A floating point number is normalized iff $\beta^{n-1} \le m < \beta^n$. For a fixed precision n, the significand m and exponent q of a normalized floating point number are uniquely determined by its value $m \times \beta^q$.

For any fixed precision n, a closest floating point approximation to a real number a is a normalized floating point number $m \times \beta^q$ such that $a = (m + \epsilon) \times \beta^q$ where $|\epsilon| \le 1/2$, and where $m = \beta^{n-1}$ only if $-1/(2\beta) \le \epsilon$. The closest approximation is uniquely determined unless $|\epsilon| = 1/2$, or $m = \beta^{n-1}$ and $\epsilon = -1/(2\beta)$, in which cases there are two closest approximations. The best approximation is a closest approximation where, in a case of two closest approximations, the tie is broken by a fixed rounding rule such as round to even.

For any real number a, the fractional part of a is defined to be $\{a\} = a - \lfloor a \rfloor$. The value of a rounded to the nearest integer, assuming ties round to even, is defined to be

$$[a] = \begin{cases} \lfloor a \rfloor & \text{if } \{a\} < 1/2 \\ \lceil a \rceil & \text{if } \{a\} > 1/2 \\ \lfloor a \rfloor & \text{if } \{a\} = 1/2 \text{ and } \lfloor a \rfloor \text{ is even} \\ \lceil a \rceil & \text{if } \{a\} = 1/2 \text{ and } \lfloor a \rfloor \text{ is odd} \end{cases}$$

This definition can easily be changed to accomodate other tie-breaking rules, but the algorithms presented later will assume that ties round to even. The significand of the best n-bit approximation to a is

$$\left[\beta^{n-1}\beta^{\{\log_\beta a\}}\right]$$

unless this value is β^n, in which case the significand is β^{n-1}.

The floating point product of normalized floating point numbers $x \times \beta^q$ and $y \times \beta^r$ is a best approximation to $xy \times \beta^{q+r}$. That is, the floating point multiplication operator is assumed to be reliably accurate, to round to nearest, and to resolve ties that result when the mathematical product is exactly halfway between two adjacent floating point numbers by the same rounding rule used to define the best approximation. These assumptions hold for the default rounding mode in IEEE arithmetic, but do not hold for many other implementations of floating point numbers.

The loss of accuracy that results from approximating a real number by a floating point number is likely to be amplified by subsequent multiplications. The following lemma tightly bounds this loss of accuracy.

Lemma 2. If

$$\begin{array}{rlll} a = (x + \epsilon_1) \times \beta^q & |\epsilon_1| \le \delta_1 & \beta^{p-1} \le x < \beta^p \\ b = (y + \epsilon_2) \times \beta^r & |\epsilon_2| \le \delta_2 & \beta^{p-1} \le y < \beta^p \\ xy \times \beta^{q+r} = (z + \epsilon_3) \times \beta^s & |\epsilon_3| \le \frac{1}{2} & \beta^{p-1} \le z < \beta^p \end{array}$$

where x, y, z, q, r, s are integers, then

$$ab = (z + \epsilon) \times \beta^s$$

where

$$|\epsilon| \le \frac{1}{2} + \beta\,(\delta_1 + \delta_2) + (\delta_1\delta_2 - \delta_1 - \delta_2) \times \beta^{-p+1}.$$

Proof:

$$\begin{aligned} ab &= (x + \epsilon_1) \times \beta^q \times (y + \epsilon_2) \times \beta^r \\ &= xy \times \beta^{q+r} + (\epsilon_1 y + \epsilon_2 x + \epsilon_1\epsilon_2) \times \beta^{q+r} \\ &= (z + \epsilon_3) \times \beta^s + (\epsilon_1 y + \epsilon_2 x + \epsilon_1\epsilon_2) \times \beta^{q+r} \end{aligned}$$

Case 1: $s = p + q + r$. Then

$$\begin{aligned} ab &= (z + \epsilon_3) \times \beta^{p+q+r} + (\epsilon_1 y + \epsilon_2 x + \epsilon_1\epsilon_2) \times \beta^{q+r} \\ &= (z + \epsilon_3 + \epsilon_1 y \times \beta^{-p} + \epsilon_2 x \times \beta^{-p} + \epsilon_1\epsilon_2 \times \beta^{-p}) \times \beta^s \end{aligned}$$

so

$$\begin{aligned} |\epsilon| &= |\epsilon_3 + \epsilon_1 y \times \beta^{-p} + \epsilon_2 x \times \beta^{-p} + \epsilon_1\epsilon_2 \times \beta^{-p}| \\ &\le \frac{1}{2} + \delta_1 \times (\beta^p - 1) \times \beta^{-p} \\ &\quad + \delta_2 \times (\beta^p - 1) \times \beta^{-p} + \delta_1\delta_2 \times \beta^{-p} \\ &= \frac{1}{2} + \delta_1 + \delta_2 + (\delta_1\delta_2 - \delta_1 - \delta_2) \times \beta^{-p} \end{aligned}$$

Case 2: $s = p + q + r - 1$. Then

$$\begin{aligned} ab &= (z + \epsilon_3) \times \beta^{p+q+r-1} + (\epsilon_1 y + \epsilon_2 x + \epsilon_1\epsilon_2) \times \beta^{q+r} \\ &= (z + \epsilon_3 + \epsilon_1 y \times \beta^{-p+1} \\ &\quad + \epsilon_2 x \times \beta^{-p+1} + \epsilon_1\epsilon_2 \times \beta^{-p+1}) \times \beta^s \end{aligned}$$

so

$$\begin{aligned} |\epsilon| &= |\epsilon_3 + \epsilon_1 y \times \beta^{-p+1} + \epsilon_2 x \times \beta^{-p+1} + \epsilon_1\epsilon_2 \times \beta^{-p+1}| \\ &\le \frac{1}{2} + \delta_1 \times (\beta^p - 1) \times \beta^{-p+1} \\ &\quad + \delta_2 \times (\beta^p - 1) \times \beta^{-p+1} + \delta_1\delta_2 \times \beta^{-p+1} \\ &= \frac{1}{2} + \beta \times (\delta_1 + \delta_2) + (\delta_1\delta_2 - \delta_1 - \delta_2) \times \beta^{-p+1} \quad \blacksquare \end{aligned}$$

Corollary 3. If, in addition, $0 < \delta_1 + \delta_2 < 4$ or $\delta_1 < 1$ or $\delta_2 < 1$, then

$$|\epsilon| < \frac{1}{2} + \beta\,(\delta_1 + \delta_2).$$

Proof: If $\delta_1 < 1$ then $\delta_1\delta_2 - \delta_1 - \delta_2 = (\delta_1 - 1)\,\delta_2 - \delta_1 < 0$.

If $\delta_1 + \delta_2 < 4$ then consider $f(x, y) = xy - x - y$ restricted to $A = \{\langle x, y\rangle \mid 0 \le x \,\&\, 0 \le y \,\&\, x + y \le 4\}$:

$$\frac{\partial f}{\partial x} = y - 1 \qquad \frac{\partial f}{\partial y} = x - 1$$

but $f(1,1) = -1$ is a saddle point, not a maximum. The maximum of f on A therefore occurs on the boundary of A, and $f(x,y)$ is strictly less than the maximum if $\langle x,y\rangle$ is in the interior of A. Let $g(x) = f(x, 4-x)$.

$$\begin{array}{rcccl} f(0,y) & = & -y & \leq & 0 \\ f(x,0) & = & -x & \leq & 0 \\ g(x) & = & f(x,4-x) & = & -x^2+4x-4 \end{array}$$

$dg/dx = -2x+4$ so the maximum of g occurs at $x=0$, $x=2$, or $x=4$. $g(0) = g(4) = -4$ and $g(2) = 0$ so f is negative on the interior of A. ∎

4. UNLIMITED PRECISION IS NEEDED

The problem of finding the best n-bit binary floating point approximation to a number written in decimal scientific notation is equivalent to the problem of finding the best n-bit approximation to $f \times 10^e$, where f and e are integers and f is positive.

What would it mean to say that this problem can or cannot be solved using arithmetic of finite precision? Algorithms that use finite precision floating point arithmetic do not simply correspond to finite automata, because the definition of a floating point number limits the precision of the significand but does not limit the range of the exponent. Limiting the range of the floating point exponent would reduce the problem to table lookup, as shown in Section 8, but the table is so large that this is of greater theoretical than practical interest. The approach taken here is to identify finite precision with finite automata, but to allow the automata to compute only the significand of the best approximation. The intuitive justification for this is that the significand of a floating point product is entirely determined by the significands of the factors, so the exponents of the factors should not matter.

Theorem 4. [Matula68] If Δ and β are commensurable, then there exists a finite automaton that computes the significand of the best n-bit floating point approximation to $f \times \Delta^e$.

Proof: Let α, u, and v be integers with $\Delta = \alpha^u$ and $\beta = \alpha^v$. The automaton constructs the (normalized) leading $nv+1$ α-ary digits of f, together with a sticky bit that tells whether any of the remaining α-ary digits of f are nonzero. The automaton also counts the number of digits of f modulo v, and adds ue to this count modulo v. It then shifts right by as many digits as are called for by (the additive inverse modulo v of) this count, rounds to nv digits using the guard digit and sticky bit, and performs the trivial conversion from nv α-ary digits to n β-ary digits. ∎

Theorem 5. For $n \geq 4$, no finite automaton computes the significand of the best n-bit binary floating point approximation to $f \times 10^e$, where f and e are presented in base 10.

Proof: This is a special case of the lemmas below. ∎

Lemma 6. If x and y are positive real numbers with

$$\frac{1}{2\beta^{n-1}\log\beta} < \left\{\log_\beta x - \log_\beta y\right\} < 1 - \frac{1}{2\beta^{n-1}\log\beta}$$

then the best n-bit floating point approximations to x and y have distinct significands.

Proof: By symmetry suppose $\{\log_\beta x\} < \{\log_\beta y\}$. Let $w = \{\log_\beta x\}$ and $w+\delta = \{\log_\beta y\}$. Then

$$\begin{array}{rcl} \beta^{w+\delta} - \beta^w & = & \beta^w(\beta^\delta - 1) \\ & \geq & \beta^\delta - 1 \\ & > & \delta\log\beta \\ & > & \dfrac{1}{2\beta^{n-1}} \end{array}$$

so

$$\left|[\beta^{n-1}\beta^{w+\delta}] - [\beta^{n-1}\beta^w]\right| > \frac{1}{2}. \quad ∎$$

Lemma 7. (Kronecker's Theorem in one dimension) If θ is irrational, then $\{\,\{n\theta\} \mid n \in \omega\}$ is dense in the interval $(0,1)$.

Proof: See [HW60]. ∎

Lemma 8. If Δ and β are incommensurable, $\beta \geq 3$ or $n \geq 2$, and e is presented with its least significant digit first, then no finite automaton computes the significand of the best n-bit approximation to Δ^e.

Proof: Suppose e is presented in radix γ, and let D be a DFA. Let i and j be integers such that D is in the same state after reading $i+j$ zeroes as after reading i zeroes. By Kronecker's theorem there exists an integer k such that

$$\frac{1}{2\beta^{n-1}\log\beta} < \left\{k\left(\gamma^{i+j} - \gamma^i\right)\log_\beta\Delta\right\} < 1 - \frac{1}{2\beta^{n-1}\log\beta}.$$

Take $e_1 = k\gamma^{i+j}$ and $e_2 = k\gamma^i$. D computes the same result for Δ^{e_1} as for Δ^{e_2}, but their significands are distinct by Lemma 6. ∎

Lemma 9. If θ is irrational and $\gamma > 1$ is an integer, then there exist infinitely many nonnegative integers k such that

$$\frac{\gamma-1}{\gamma^2} < \left\{\gamma^k\theta\right\} < \frac{\gamma^2-\gamma+1}{\gamma^2}.$$

Proof: If all digits to the right of the radix point in the γ-ary representation of θ are zero or $\gamma-1$, then let k be such that the kth digit to the right of the radix point is $\gamma-1$ and the following digit is zero. Otherwise let k be such that the kth digit is neither zero nor $\gamma-1$.

Since $\gamma^k\theta$ is also irrational, a larger such k always exists. ∎

Lemma 10. If Δ and β are incommensurable, e is presented in base γ with its most significant digit first, and $\gamma^2/(\gamma-1) < 2\beta^{n-1}\log\beta$, then no finite automaton computes the significand of the best n-bit approximation to Δ^e.

Proof: Let D be a DFA, and let i and j be integers such that D is in the same state after reading γ^{i+j} as after reading γ^i. By Lemma 9 there exists an integer k such that

$$\frac{\gamma-1}{\gamma^2} < \left\{\gamma^k\left(\gamma^{i+j} - \gamma^i\right)\log_\beta\Delta\right\} < \frac{\gamma^2-\gamma+1}{\gamma^2}$$

so take $e_1 = \gamma^{i+j+k}$ and $e_2 = \gamma^{i+k}$. ∎

The lemma above is unpleasantly technical, for there is no apparent reason why the base in which the inputs are presented should affect the difficulty of the problem. The lemma probably holds without such assumptions, but a more sophisticated proof will be required.

Figure 1 shows a straightforward algorithm, `AlgorithmM`, that uses integer arithmetic of unlimited precision to compute the best n-bit floating point approximation to $f \times \Delta^e$. As written, the algorithm assumes ties are broken by rounding to even.

Like the other algorithms in this paper, `AlgorithmM` is expressed as a purely functional Scheme program, with the

```
; Given exact integers f and e, with f nonnegative,
; returns the floating point number closest to
; f * delta^e.

(define (AlgorithmM f e)

  ; f * delta^e = u/v * beta^k

  (define (loop u v k)
    (let ((x (quotient u v)))
      (cond ((and (<= beta^n-1 x) (< x beta^n))
             (ratio->float u v k))
            ((< x beta^n-1)
             (loop (* beta u) v (- k 1)))
            ((<= beta^n x)
             (loop u (* beta v) (+ k 1))))))

  (if (negative? e)
      (loop f (expt 10 (- e)) 0)
      (loop (* f (expt 10 e)) 1 0)))

; Given exact positive integers u and v with
; beta^(n-1) <= u/v < beta^n, and exact integer k,
; returns the float closest to u/v * beta^k.

(define (ratio->float u v k)
  (let* ((q   (quotient u v))
         (r   (- u (* q v)))
         (v-r (- v r))
         (z   (make-float q k)))
    (cond ((< r v-r) z)
          ((> r v-r) (nextfloat z))
          ((even? q) z)
          (else (nextfloat z)))))

(define delta 10)
(define beta 2)
(define n 53)
(define beta^n (expt beta n))
(define beta^n-1 (expt beta (- n 1)))
```

Figure 1. `AlgorithmM`.

assumption that all integer arithmetic is exact, *i.e.* of unlimited precision [Rees86]. For integers m and k with $\beta^{n-1} \leq m < \beta^n$, `(make-float` m k`)` is assumed to return the n-bit floating point number $m \times \beta^k$. The `nextfloat` procedure, shown in Figure 2, returns the least normalized floating point number greater than its argument.

For most applications `AlgorithmM` is impractical because it uses too much high-precision arithmetic. The next section obtains a better algorithm by starting from a close but not necessarily closest approximation.

5. AN ITERATIVE ALGORITHM

It is quite easy, using a few extra bits of precision, to find an approximation that differs from the best approximation by only a few units in the last place of the significand. In fact, it is fairly easy to find an n-bit approximation that differs from the best approximation by no more than one unit.

`AlgorithmR`, in Figure 3, takes a good approximation

```
; Given a normalized floating point number
; z = m * beta^k, returns the normalized floating
; point number whose value is (m+1) * beta^k.

(define (nextfloat z)
  (let ((m (float-significand z))
        (k (float-exponent z)))
    (if (= m (- beta^n 1))
        (make-float beta^n-1 (+ k 1))
        (make-float (+ m 1) k))))

; Given a normalized floating point number
; z = m * beta^k, returns the greatest normalized
; floating point number less than z.  Note that the
; value returned may be greater than (m-1) * beta^k.

(define (prevfloat z)
  (let ((m (float-significand z))
        (k (float-exponent z)))
    (if (= m beta^n-1)
        (make-float (- beta^n 1) (- k 1))
        (make-float (- m 1) k))))
```

Figure 2. `Nextfloat` and `prevfloat`.

$m \times \beta^k$ and checks it using integer arithmetic of unlimited precision. If the given approximation is too small or too large, it then repeats the process with the next larger or smaller floating point number.

The algorithm begins by finding positive integers x and y such that

$$\frac{x}{y} = \frac{f \times \Delta^e}{m \times \beta^k}.$$

The purpose of this is to eliminate any further dispatching on the signs of e and k, but the choice of x and y may also take advantage of any common factors possessed by Δ and β so as to reduce the size of the integers that will be manipulated.

Let ϵ be the error such that $f \times \Delta^e = (m+\epsilon) \times \beta^k$. Then $x/y = (m+\epsilon)/m$ so

$$\epsilon = \frac{m\,(x-y)}{y}.$$

The algorithm proceeds by comparing $|\epsilon|$ to $1/2$, taking care to avoid division.

`AlgorithmR` can be criticized for its unimaginably slow convergence when given a poor starting approximation, and for the fact that it performs several expensive but loop-invariant computations on each iteration. When `AlgorithmR` is incorporated into an efficient algorithm, however, the starting approximation will always be either the best approximation or one of the two floating point numbers adjacent to the best approximation.

For such a starting approximation, the tail-recursive calls to `loop` from within the `compare` procedure can be replaced by their arguments, `(prevfloat z)` and `(nextfloat z)`. So modified, the algorithm always executes the body of the loop exactly once.

6. FIXED PRECISION COMES CLOSE

```
; Given exact integers f and e, with f positive,
; and a floating point number z0 close to f * delta^e,
; returns the best floating point approximation to
; f * delta^e.

(define (AlgorithmR f e z0)

  (define (loop z)

    (define m (float-significand z))
    (define k (float-exponent z))

    ; Given exact positive integers x and y with
    ; x/y = (f*delta^e)/(m*beta^k), returns the best
    ; approximation to f*delta^e.

    (define (compare x y)
      (let* ((D (- x y))
             (D2 (* 2 m (abs D))))
        (cond ((< D2 y)
               (if (and (= m beta^n-1)
                        (negative? D)
                        (> (* beta D2) y))
                   (loop (prevfloat z))
                   z))
              ((= D2 y)
               (cond ((even? m)
                      (if (and (= m beta^n-1)
                               (negative? D))
                          (loop (prevfloat z))
                          z))
                     ((negative? D)
                      (prevfloat z))
                     ((positive? D)
                      (nextfloat z))))
              ((negative? D)
               (loop (prevfloat z)))
              ((positive? D)
               (loop (nextfloat z))))))

    (cond ((and (>= e 0) (>= k 0))
           (compare (* f (expt delta e))
                    (* m (expt beta k))))
          ((and (>= e 0) (< k 0))
           (compare (* f (expt delta e)
                          (expt beta (- k)))
                    m))
          ((and (< e 0) (>= k 0))
           (compare f (* m (expt beta k)
                           (expt delta (- e)))))
          ((and (< e 0) (< k 0))
           (compare (* f (expt beta (- k)))
                    (* m (expt delta (- e)))))))

  (loop z0))

(define beta^n+1 (expt beta (+ n 1)))
```

Figure 3. AlgorithmR.

To prevent the numerical analysis from becoming too abstract, this section and the next assume that $\beta = 2$. The results of these two sections can be applied to other even internal radixes by repeating the numerical analysis.

Let $p \geq n + 4$ be a convenient extended precision. An excellent starting approximation for `AlgorithmR` can be obtained by finding reasonably close p-bit floating point approximations to f and to Δ^e and multiplying them. This is hardly an efficient solution, because `AlgorithmR` involves integer arithmetic of unlimited precision, but the results of Section 4 say there will be times when such arithmetic cannot be avoided.

A closest p-bit approximation to f can be computed quite easily. While f itself may be a large integer requiring multiple precision, f is likely to be representable in $n + 7$ bits because the number of decimal digits needed to specify any n-bit binary floating point number is the least d such that $10^{d-1} > 2^n$ [Goldberg67] whence

$$\lceil log_2 10^d \rceil \leq n + 7.$$

A close approximation to Δ^e is expensive to compute when the absolute value of e is large. The most practical solution seems to be a pre-computed table of powers of Δ, containing the range of powers that is apt to occur in practice. With practical floating point formats the range of floating point exponents is usually limited, so very large exponents will overflow and very small exponents will underflow unless the number of digits in the input is unreasonably large. `AlgorithmM` can be used when the input exponent is out of the table's range.

Even when limited to the range needed for reasonable inputs, the table of powers may be fairly large. The size of the table can be reduced, at the expense of accuracy, by factoring it into two smaller tables. One table contains values for small powers of Δ^e, with $0 \leq e < h$, while another table contains approximations to 10^{hj} for integral j. It is convenient to assume that h is small enough that the small powers are represented exactly as p-bit floating point numbers. If this is so, and all other table entries are best approximations, and $\beta = 2$, then Corollary 3 says that the error in the value calculated for Δ^e is strictly less than $\frac{3}{2}$ units in the least significant bit.

If the floating point approximation to f is the best possible, and the approximation to Δ^e is within $\frac{3}{2}$ units, and $\beta = 2$, then the error in the product is less than $\frac{9}{2}$ units. If this calculation is performed using $p \geq n + 4$ bits of precision, then rounding the product to the nearest n bits yields either the best n-bit binary floating point approximation to $f \times \Delta^e$ or a next-best approximation.

7. AN EFFICIENT, NON-ITERATIVE ALGORITHM

Algorithm `Bellerophon`, shown in Figure 4 for the special case of $\Delta = 10$ and $\beta = 2$, is a practical algorithm based on the idea explained in the introduction. The terms used in Figure 4 differ from those used to describe previous algorithms, in that "float" refers to p-bit floating point numbers and "shortfloat" refers to n-bit numbers.

Given integers f and e, `Bellerophon` dispatches on the error introduced when f and Δ^e are approximated by floating point numbers with p bits of precision, where p is large enough to ensure that the product of the approximations, rounded to n bits, is either the best or a next-best approximation to $f \times \Delta^e$.

```
; Given exact integers f and e with f > 0,
; return the float with n bits of precision that best approximates it.
; Tries to do the calculation using floats with p bits of precision.
; The error bounds used here assume perfect floating point arithmetic,
; as in the IEEE standard.  They are independent of p and n.

(define (Bellerophon f e)
  (cond ((and (< f two^n) (>= e 0) (< e h) (< e log5-of-two^n))
         (shortfloat-multiply (int->shortfloat f)
                              (float->shortfloat (ten-to-e e))))
        ((and (< f two^n) (< e 0) (< (- e) h) (< (- e) log5-of-two^n))
         (shortfloat-divide (int->shortfloat f)
                            (float->shortfloat (ten-to-e (- e)))))
        ((and (< f two^p) (>= e 0) (< e h))
         (multiply-and-test f e 0))
        ((and (< f two^p) (or (< e 0) (>= e h)))
         (multiply-and-test f e 3))
        ((and (>= f two^p) (>= e 0) (< e h))
         (multiply-and-test f e 1))
        ((and (>= f two^p) (or (< e 0) (>= e h)))
         (multiply-and-test f e 4))))

; Slop, expressed in units of the least significant bit, is an
; inclusive bound for the error accumulated during the floating
; point calculation of an approximation to f * 10^e.  (Slop is
; not a bound for the true error, but bounds the difference
; between the approximation z and the best possible approximation
; that uses p bits of significand.)
;
; Fail is a slow but perfect backup algorithm.
; Z is passed so fail can use it as a starting approximation.

(define (multiply-and-test f e slop)
  (let ((x (int->float f))
        (y (ten-to-e e)))
    (let ((z (float-multiply x y)))
      (let ((lowbits (remainder (float-significand z) two^p-n)))

        ; is the slop large enough to make a difference when
        ; rounding to n bits?

        (if (<= (abs (- lowbits two^p-n-1)) slop)
            (fail f e z)
            (float->shortfloat z))))))

(define (fail f e z)
  (AlgorithmR f e (float->shortfloat z)))

(define n 53)                        ; IEEE double
(define p 64)                        ; an extended precision
(define two^p (expt 2 p))
(define two^p-1 (expt 2 (- p 1)))
(define two^p-n (expt 2 (- p n)))
(define two^p-n-1 (expt 2 (- p n 1)))
(define two^n (expt 2 n))
(define two^n-1 (expt 2 (- n 1)))

(define log5-of-two^n
  (inexact->exact (ceiling (/ (log two^n) (log 5)))))
```

Figure 4. Algorithm Bellerophon.

If f and Δ^e can both be represented exactly using n bits, then an n-bit floating point multiplication yields the best approximation to their product. If f and Δ^{-e} can both be represented exactly using n bits, then an n-bit division yields the best approximation. (This assumes that floating point division, like multiplication, is reliably accurate.)

Otherwise `Bellerophon` approximates f and Δ^e by p-bit floating point numbers x and y, and computes their floating point product $z = m \times \beta^q$, where $\beta^{p-1} \leq m < \beta^p$. Hence

$$f \times \Delta^e = (z + \epsilon) \times \beta^q$$

where for $\beta = 2$ the error ϵ is bounded by the values shown in Figure 5.

Unless z lies about halfway between two adjacent n-bit floating point numbers, the error ϵ will be absorbed when z is rounded to n bits. The `multiply-and-test` procedure therefore tests to see if z could be within ϵ of the midpoint. If not, then the correct answer is obtained by rounding z to n bits. Otherwise the efficient part of the algorithm fails, and the rounded value of z is passed to `AlgorithmR` as a starting approximation. Since the rounded value of z is always either the best or a next-best approximation, `AlgorithmR` always converges in one loop.

Theorem 11. Algorithm `Bellerophon` computes the best n-bit approximation to $f \times \Delta^e$.

Proof: This proof deals with the generalizations of Figures 4 and 5 to any even internal radix β. In general, Algorithm `Bellerophon` computes a p-bit floating point number $z = m \times \beta^k$ such that $f \times \Delta^e = (z + \epsilon) \times \beta^k$ and

$$|\epsilon| \leq \texttt{slop} + \frac{1}{2}$$

where the value of `slop` is determined by numerical analysis.

Let z_1 and z_0 be integers such that

$$z = z_1 \times \beta^{p-n} + z_0$$
$$\beta^{n-1} \leq z_1 < \beta^n$$
$$0 \leq z_0 < \beta^{p-n}$$

There are three cases, depending on whether z_0 is well below, well above, or near $\frac{1}{2}\beta^{p-n}$.

Case 1: $z_0 + \texttt{slop} < \frac{1}{2}\beta^{p-n}$. Rounding z to n bits yields $z_1 \times \beta^{k+p-n}$, and $f \times \Delta^e = (z_1 + \epsilon') \times \beta^{k+p-n}$ where the error ϵ' is

$$\begin{aligned}
|\epsilon'| &= \left|\frac{f \times \Delta^e}{\beta^{k+p-n}} - z_1\right| \\
&= \left|\frac{(z+\epsilon) \times \beta^k}{\beta^{k+p-n}} - z_1\right| \\
&= \left|\frac{z_0 + \epsilon}{\beta^{p-n}}\right| \\
&\leq \beta^{n-p}(z_0 + |\epsilon|) \\
&\leq \beta^{n-p}(z_0 + \texttt{slop} + \frac{1}{2}) \\
&< \beta^{n-p} \times \frac{1}{2}\beta^{p-n} \\
&= \frac{1}{2}
\end{aligned}$$

Case 2: $\frac{1}{2}\beta^{p-n} < z_0 - \texttt{slop}$. Rounding z to n bits yields $(z_1+1) \times \beta^{k+p-n}$, and $f \times \Delta^e = (z_1+1+\epsilon') \times \beta^{k+p-n}$ where

$$\begin{aligned}
|\epsilon'| &= \left|\frac{(z+\epsilon) \times \beta^k}{\beta^{k+p-n}} - z_1 - 1\right| \\
&= \left|\frac{z_0 + \epsilon}{\beta^{p-n}} - 1\right| \\
&\leq \beta^{n-p}(\beta^{p-n} - z_0 + |\epsilon|) \\
&\leq \beta^{n-p}(\beta^{p-n} - z_0 + \texttt{slop} + \frac{1}{2}) \\
&< \beta^{n-p} \times (\beta^{p-n} - \frac{1}{2}\beta^{p-n}) \\
&= \frac{1}{2}
\end{aligned}$$

Case 3: $\frac{1}{2}\beta^{p-n} - \texttt{slop} \leq z_0 \leq \frac{1}{2}\beta^{p-n} + \texttt{slop}$. This is the failure case in which another algorithm is used. ■

8. OVERFLOW AND UNDERFLOW

Overflow and underflow become possible when the range of floating point exponents is restricted. Algorithm `Bellerophon` can be modified to deal with overflow and underflow by testing the n-bit result to see if it is an infinity, the largest representable floating point number, the smallest normalized floating point number, denormalized, or zero. In such cases the computation may need to be repeated using some other algorithm, depending on the policies that have been established for handling overflow and underflow within the particular floating point number system in question.

With IEEE arithmetic, for example, a denormalized result may be required. Denormalized results can be generated by a modified form of `AlgorithmM` that terminates immediately when the minimum exponent is reached.

When exponents are bounded, the input problem can be solved by table lookup:

Theorem 12. If floating point exponents are bounded, then there exists a finite automaton that takes f and e as inputs and computes the significand of $f \times \Delta^{e-d(f)}$, where $d(f) = \lfloor \log_\Delta f \rfloor$.

Proof: There are only a finite number of floating point numbers and only a finite number of inputs e such that, for some f, $f \times \Delta^{e-d(f)}$ does not overflow or underflow.

The automaton contains a table indexed by e. For each e, the entry for e is an enormous table containing an entry for every floating point number that can result from that value of e. The entries in this subtable are indexed by representations of the numbers that lie exactly halfway between two floating point numbers. These midpoints are expressed as sequences of input digits, so they might not always be expressible as finite sequences, but they are rational so they can be encoded as sub-automata. (If $\beta = 6$ and $f = \beta^{-1}$ is expressed in base 10, for example, then f is a repeating but not a terminating decimal fraction.) Associated with each midpoint are the floating point numbers that it separates, together with an indication of how ties should be broken when f is equal to the midpoint.

The automaton reads e first and uses it as an index into the table. Then it reads f, finds the midpoints that f lies between, and reads off the answer. ■

Corollary 13. If floating point exponents are bounded, then the best binary floating point approximation to $f \times 10^e$ can be found by approximating f using $\lceil \log_2 10^{n+1} \rceil$ bits of precision and performing an enormous table lookup.

	$\lvert\epsilon_{\mathbf{x}}\rvert$	$\lvert\epsilon_{\mathbf{y}}\rvert$	$\lvert\epsilon\rvert$
$f < 2^p \land 0 \le e < h$	0	0	$\le \frac{1}{2}$
$f < 2^p \land (e < 0 \lor e \ge h)$	0	$< \frac{3}{2}$	$< \frac{7}{2}$
$f \ge 2^p \land 0 \le e < h$	$\le \frac{1}{2}$	0	$< \frac{3}{2}$
$f \ge 2^p \land (e < 0 \lor e \ge h)$	$\le \frac{1}{2}$	$< \frac{3}{2}$	$< \frac{9}{2}$

Figure 5. Error bounds in units of the least significant bit.

To convert decimal scientific notation to IEEE double precision using the 180-bit precision implied by the corollary, the table used in the proof would have nearly 10^{20} entries, most of which would contain over 50 decimal digits. The table can be compressed by several orders of magnitude, but it is hard to believe that this approach can be made practical.

On the other hand the existence of such an algorithm implies that, if Algorithm `Bellerophon` is in any sense optimal for practical floating point formats, then a proof of its optimality must be at least as difficult as showing that this table cannot be compressed by more than a few orders of magnitude.

Instead of comparing Algorithm `Bellerophon` against all possible algorithms, therefore, it makes sense to compare it against all algorithms that work by multiplying β-ary floating point approximations to f and 10^e, where the precision of these approximations is a function of f and e. When Algorithm `Bellerophon` fails and must resort to a less efficient algorithm, any other algorithm of this class that uses the same error bounds available to `Bellerophon` must also resort to a higher precision, because `Bellerophon` makes optimal use of the error bounds available to it. It is possible to improve upon `Bellerophon` by using a more efficient algorithm for the failure case, however.

Instead of using `AlgorithmR` for the failure case, Algorithm `Bellerophon` may itself be used with a higher precision, say twice the precision. This refinement guarantees that the precision used is within a constant factor of the smallest possible precision, at the cost of storing a table of the powers of Δ for each precision that might be used.

9. EXPERIMENTAL RESULTS

For inputs generated by IEEE-conforming output routines to the maximum output precisions specified in [IEEE85], Algorithm `Bellerophon` never has to resort to the failure algorithm provided p is at least as large as the extended precisions specified by [Coonen80].

Higher precision arithmetic may be needed to compute the best approximation to inputs generated by the algorithms in [Steele90], because minimizing the number of output digits inevitably increases the error in the printed values. This has the effect of moving those values closer to the midpoints between adjacent floating point numbers. Even so, `Bellerophon` is much less likely to fail on inputs generated by the algorithms in [Steele90] than on uniformly distributed inputs.

As a simple test of `Bellerophon` on more uniformly distributed inputs, 64-bit IEEE extended precision arithmetic was used to find the best IEEE double precision approximation for over ten million sample inputs spanning a wide range of f and e. On these inputs, the algorithm avoided higher precision arithmetic over 99.6% of the time.

In a classic example of local optimization leading to global pessimization, I attempted to save an instruction or two by choosing $h = 16$ as the size of the table of small powers of ten instead of using $h = \lceil \log_5 2^{53} \rceil = 23$. As a result, the algorithm failed systematically for $e = 18$ and odd f beginning with $f = 2363$, changing to every fourth f at $f = 4726$. Such systematic failures will occur for all nonnegative $e < \log_5 2^p$ as f becomes just large enough to shift the rightmost nonzero bit of 10^e into the bit field being tested by `Bellerophon`. These systematic failures can be eliminated by storing exact values of Δ^e for all such e in the table of small powers.

Some compilers do not implement IEEE arithmetic correctly. For example, the Motorola 68881/68882 floating point coprocessors perform extended precision IEEE arithmetic faster than double or single precision IEEE arithmetic. By default, therefore, some compilers that appear to support IEEE single or double precision arithmetic may actually perform single or double precision calculations using extended precision, rounding to single or double only when a result is stored in a variable. Somewhat counterintuitively, this makes individual floating point operations less accurate, and does not meet the error bounds required by the IEEE specification for the default rounding mode using single or double precision arithmetic.

Suppose, for example, that Algorithm `Bellerophon` is used to compute the best IEEE double precision approximation to `1.448997445238699`. The correct result is

$$6525704354437805 \times 2^{-52} \doteq 1.448997445238699$$

obtained by dividing 1448997445238699 by 10^{15} using double precision arithmetic. If this division is performed using 64-bit extended precision arithmetic instead, and the extended precision result rounded to double precision, then the incorrect result

$$6525704354437806 \times 2^{-52} \doteq 1.4489974452386991$$

will be obtained.

10. RELATED WORK

Mathematical properties of the best approximation function have been investigated by Matula, who does not consider algorithms for computing it [Matula68, 70].

Theorem 4 strengthens an observation by Matula and others [Matula68]. Calculations similar to Lemma 2 and

Corollary 3 appear in [Knuth81] and in most books on numerical analysis, though the results are seldom stated as they appear here. Theorem 5 expresses well-known folklore, but to my knowledge this is the first proof of it.

`AlgorithmM` is essentially the same as Method (2a) in Section 4.4 of [Knuth81]. The solution to Exercise 3 of that section contains a forward reference to [Steele90].

A draft of [Clinger90] required the standard routine for numerical output to print floating point numbers using the fewest digits that allow the number to be read back in without loss of accuracy. Although this can be done by extending an IEEE-conforming but imperfect implementation, reference was made to a draft of [Steele90], which assumes a perfect input routine.

On 1 November 1989 Chris Hanson expressed concern over this requirement in electronic mail sent to Steele, White, and myself. Hanson described `AlgorithmM` but noted its inefficiency and asked whether any other perfect algorithms, especially perfect and efficient algorithms, were published or known. The matter was urgent because Hanson was editing a draft IEEE standard for Scheme to be voted on in January. After checking with Steele to confirm that he did not know of an efficient solution to the input problem, I set to work, keeping the others informed of my progress.

Jon L White was out of town and unable to read his mail. On 10 November 1989, after I had announced the basic idea of Algorithm `Bellerophon`, White reported that Lucid Common Lisp has for several years used a similar algorithm of his invention. This algorithm has not been published, and was known only to a handful of people at Lucid. From subsequent telephone conversations, it appears that the algorithm in use at Lucid is essentially the same as `Bellerophon` but uses twice as many bits, primarily because the error bounds were not calculated very tightly.

`Bellerophon` is so named because it inverts the Dragon3 and Dragon4 algorithms of [Steele90]. Unlike its namesake, the algorithm reads its fate and acts accordingly.

ACKNOWLEDGEMENTS

Chris Hanson helped in many ways, and I am indebted also to David Wise, Jon L White, Guy L Steele Jr, and Anne Hartheimer.

REFERENCES

[Clinger90] Clinger, William, and Jonathan Rees [editors]. Revised^4 report on the algorithmic language Scheme. Technical Report CIS-TR-90-02, Department of Computer and Information Science, University of Oregon, 1990.

[Coonen80] Coonen, Jerome T. An implementation guide to a proposed standard for floating-point arithmetic. *Computer* **13**, 1, January 1980, pages 68–79.

[Goldberg67] Goldberg, I. B. 27 bits is not enough for 8-digit accuracy. *CACM* **10**, 2, February 1967, pages 105–106.

[HW60] Hardy, G. H., and E. M. Wright. *An Introduction to the Theory of Numbers, Fourth Edition.* Oxford University Press, 1960.

[IEEE85] *IEEE Standard 754-1985. IEEE Standard for Binary Floating-Point Arithmetic.* IEEE, New York, 1985.

[Knuth81] Knuth, Donald E. *The Art of Computer Programming, Second Edition, Volume 2, Seminumerical Algorithms.* Addison-Wesley, 1981.

[Matula68] Matula, David W. In-and-out conversions. *CACM* **11**, 1, January 1968, pages 47–50.

[Matula70] Matula, David W. A formalization of floating-point numeric base conversion. *IEEE Transactions on Computers*, **C-19**, 8, August 1970, pages 681–692.

[Rees86] Rees, Jonathan, and William Clinger [editors]. Revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices* **21**, 12, December 1986, pages 37–79.

[Steele90] Steele Jr, Guy Lewis, and Jon L White. How to print floating point numbers accurately. Proceedings of this conference.