Printing Floating-Point Numbers Quickly and Accurately

Robert G. Burger* R. Kent Dybvig

Indiana University Computer Science Department
Lindley Hall 215
Bloomington, Indiana 47405
(812) 855-3608
{*burger, dyb*}*@cs.indiana.edu*

Abstract

This paper presents a fast and accurate algorithm for printing floating-point numbers in both free- and fixed-format modes. In free-format mode, the algorithm generates the shortest, correctly rounded output string that converts to the same number when read back in, accommodating whatever rounding mode the reader uses. In fixed-format mode, the algorithm generates a correctly rounded output string using special # marks to denote insignificant trailing digits. For both modes, the algorithm employs a fast estimator to scale floating-point numbers efficiently.

Keywords: floating-point printing, run-time systems

1 Introduction

In this paper we present an efficient floating-point printing algorithm, which solves the *output problem* of converting floating-point numbers from an input base (usually a power of two) to an output base (usually ten).

The algorithm supports two types of output, *free format* and *fixed format*. For free-format output the goal is to produce the shortest, correctly rounded output string that converts to the same internal floating-point number when read by an accurate floating-point input routine [1]. For example, $\frac{3}{10}$ would print as `0.3` instead of `0.2999999`. The algorithm accommodates any input rounding mode, including IEEE unbiased rounding, for example.

For fixed-format output the goal is to produce correctly rounded output to a given number of places without "garbage digits" beyond the point of significance. For example, the floating-point representation of $\frac{1}{3}$ might print as `0.3333333148` even though only the first seven digits are significant. The algorithm uses special # marks to denote insignificant trailing digits so that $\frac{1}{3}$ prints as `0.3333333###`. These marks are useful when printing denormalized numbers, which may have only a few digits of precision, or when printing to a large number of digits.

Our algorithm is based on an elegant floating-point printing algorithm developed by Steele and White [5]. Their algorithm also supports both free- and fixed-format output, although it does not properly handle input rounding modes or distinguish between significant and insignificant trailing zeros. Furthermore, it is unacceptably slow for practical use. An important step in the conversion algorithm is to scale the floating-point number by an appropriate power of the output base. Steele and White's iterative algorithm requires $O(|\log x|)$ high-precision integer operations to scale x, which results in poor performance for floating-point numbers with very large and very small magnitudes. We developed an efficient estimator that always produces an estimate within one of the correct power, so our algorithm scales all floating-point numbers in just a few high-precision integer operations.

Section 2 develops a basic floating-point printing algorithm in terms of exact rational arithmetic. Section 3 describes an implementation of the algorithm using high-precision integer arithmetic and our efficient scaling-factor estimator. Section 4 extends the algorithm to handle fixed-format output and introduces # marks. Section 5 summarizes our results and discusses related work.

2 Basic Algorithm

In describing the basic algorithm, we first explain how floating-point numbers are represented, using the IEEE double-precision floating-point specification as an example [3]. Second, we develop an output algorithm based on a key feature of the representation, the gaps between floating-point numbers. Finally, we prove that our algorithm generates the shortest, correctly rounded output string from which the original floating-point number can be recovered when input.

2.1 Floating-Point Representation

An important goal in the design of floating-point numbers is to provide a representation that can approximate real numbers to a certain number of digits of accuracy. Consequently, a floating-point representation embodies the notions of the first few significant digits and the location of the decimal point.

A floating-point number is modeled mathematically by a mantissa, which corresponds to the first few significant digits, and an exponent, which corresponds to the location of the decimal point. For example, suppose v is a floating-point number in base b (usually two). The mantissa, f, and exponent, e, are base-b integers such that $v = f \times b^e$ and

*Supported in part by a National Science Foundation Graduate Research Fellowship

To appear in *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation.*

$|f| < b^p$, where p is the fixed size of the mantissa in base-b digits. Moreover, v is called *normalized* if $b^{p-1} \leq |f|$, *i.e.*, the mantissa begins with a non-zero digit.

An un-normalized, non-zero floating-point number can be normalized by shifting the mantissa to the left and reducing the exponent accordingly. Because the exponent has a fixed size, however, some numbers cannot be normalized. Such numbers are called *denormalized* floating-point numbers.

If the input base is two, the mantissa of normalized, non-zero numbers always begins with a one. Consequently, this initial bit is often omitted from the representation and is called the *hidden bit*. These representations often reserve an exponent bit pattern to signal denormalized numbers.

The IEEE specification [3] also provides representations for -0.0, positive infinity (+inf), negative infinity (−inf), and "not a number" (NaN).

An IEEE double-precision floating-point number, v, is represented as a 64-bit datum composed of three fields: a one-bit sign, an eleven-bit unsigned biased exponent (be), and a 52-bit unsigned mantissa (m) with a hidden bit.

If $1 \leq be \leq 2046$, v is a normalized floating-point number whose value is $sign\ (2^{52} + m) \times 2^{be-1075}$. If $be = 0$, v is a denormalized floating-point number whose value is $sign\ m \times 2^{-1074}$, which includes $+0.0$ and -0.0. If $be = 2047$ and $m = 0$, v is +inf or −inf, depending on the sign. If $be = 2047$ and $m \neq 0$, v is NaN.

This type of representation produces uneven gaps between floating-point numbers. Floating-point numbers are most dense around zero and decrease in density as one moves outward along the real number line in either direction.

Given a floating-point number, v, it is useful to define its floating-point successor, denoted by v^+, and predecessor, denoted by v^-. All real numbers between $\frac{v^- + v}{2}$ and $\frac{v + v^+}{2}$ round to v.

Suppose $v = f \times b^e$ as before. We consider the case where $f > 0$; the case for $f < 0$ is completely analogous. For all v, v^+ is $(f+1) \times b^e$. If $f+1$ no longer fits in the fixed-size mantissa, *i.e.*, if $f + 1 = b^p$, then v^+ is $b^{p-1} \times b^{e+1}$. If e is the maximum exponent, v^+ is +inf.

For most v, v^- is $(f-1) \times b^e$. For the remaining v, the gap is narrower. If $f = b^{p-1}$ and e is greater than the minimum exponent, v^- is $(b^p - 1) \times b^{e-1}$.

2.2 Algorithm

We now develop an algorithm that takes advantage of the gaps between floating-point numbers in order to produce the shortest, correctly rounded output string from which the original floating-point number can be recovered when input.

For purposes of discussion, we limit the input to positive floating-point numbers. Given a positive floating-point number v in terms of its mantissa and exponent, the algorithm uses v^- and v^+ to determine the exact range of values that would round to v when input. Because input rounding algorithms use different strategies to break ties (*e.g.*, round up or round to even), we initially assume that neither end point of the rounding range can be guaranteed to round to v when input. In Section 3 we show how to relax this constraint based on knowledge of a particular input rounding algorithm.

The algorithm uses exact rational arithmetic to perform its computations so that there is no loss of accuracy. In order to generate digits, the algorithm scales the number so that it is of the form $0.d_1 d_2 \ldots$, where d_1, d_2, ..., are base-B digits. The first digit is computed by multiplying the scaled number by the output base, B, and taking the integer part. The remainder is used to compute the rest of the digits using the same approach.

The rounding range determines when the algorithm stops generating digits. After each digit is generated, the algorithm tests to see if either the resulting number or the resulting number with the last digit incremented is within the rounding range of v. If one or both of these numbers is within range, the number closer to v is chosen. In the case of a tie, any strategy can be used to decide, since both possibilities would round to v when input. By testing the output number at each digit, the algorithm produces the shortest possible output string that would correctly round to v when input. Moreover, it generates digits from left to right without the need to propagate carries.

The following is a more formal description of the algorithm. We use $\lfloor x \rfloor$ to denote the greatest integer less than or equal to x, $\lceil x \rceil$ to denote the least integer greater than or equal to x, and $\{x\}$ to denote $x - \lfloor x \rfloor$. We always indicate multiplication with the $\times$ sign, because we use juxtaposition to indicate digits in a place-value notation.

Input: output base B and positive floating-point number $v = f \times b^e$ of precision $p > 0$

Output: $V = 0.d_1 d_2 \ldots d_n \times B^k$, where $d_1, \ldots, d_n$ are base-B digits and n is the smallest integer such that:

(1) $\frac{v^- + v}{2} < V < \frac{v + v^+}{2}$, *i.e.*, V would round to v when input, regardless of the input rounding algorithm, and

(2) $|V - v| \leq \frac{B^{k-n}}{2}$, *i.e.*, V is correctly rounded.

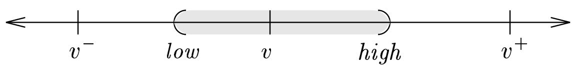
Procedure:

1. Determine v^- and v^+, the floating-point predecessor and successor of v, respectively.

$$v^- = \begin{cases} v - b^e & \text{if } e = \text{min. exp. or } f \neq b^{p-1} \\ v - b^{e-1} & \text{if } e > \text{min. exp. and } f = b^{p-1} \end{cases}$$

$$v^+ = v + b^e$$

Let $high = \frac{v^- + v}{2}$ and $low = \frac{v + v^+}{2}$. All numbers between low and $high$ round to v, regardless of the input rounding algorithm.



2. Find the smallest integer k such that $high \leq B^k$; *i.e.*, $k = \lceil \log_B high \rceil$. k is used to scale v appropriately.

3. Let $q_0 = \frac{v}{B^k}$. Generate digits as follows:

$$\begin{array}{ll} d_1 = \lfloor q_0 \times B \rfloor & q_1 = \{q_0 \times B\} \\ d_2 = \lfloor q_1 \times B \rfloor & q_2 = \{q_1 \times B\} \\ \vdots & \vdots \end{array}$$

4. Stop at the smallest n for which

(1) $0.d_1 \ldots d_n \times B^k > low$, *i.e.*, the number output at point n would round up to v, or

(2) $0.d_1 \ldots d_{n-1}[d_n{+}1]$[1] $\times B^k < high$, *i.e.*, incrementing digit d_n would make the number round down to v.

If condition (1) is true and condition (2) is false, return $0.d_1 \ldots d_n \times B^k$.

If condition (2) is true and condition (1) is false, return $0.d_1 \ldots d_{n-1}[d_n{+}1] \times B^k$.

If both conditions are true, return the number closer to v. If the two are equidistant from v, use some strategy to break the tie (*e.g.*, round up).

2.3 Correctness

We now prove that our algorithm is correct. We begin by showing that the algorithm generates valid base-B digits, the first of which is non-zero, and that there is no need to propagate carries in the case of incrementing the last digit.

Because $0 \le q_i < 1$ for all $0 \le i \le n$, all the d_i are valid base-B digits. Termination condition (2) guarantees that if d_n is incremented, no carry will be introduced; for if there were a carry, termination condition (2) would have held at the previous step. By the minimality of k and termination condition (2), the first digit must be non-zero. (See Theorem 1 in Appendix A for a complete proof.)

Next we show that output condition (1) holds, *i.e.*, the algorithm always terminates with a number that correctly rounds to v when input, satisfying the goal of information preservation. In order to prove this, we first prove an invariant of the digit-generation loop by induction: for all i, $0.d_1 \ldots d_i \times B^k + q_i \times B^{k-i} = v$. In other words, the number generated at step i is $q_i \times B^{k-i}$ below v. (See Lemma 2 in Appendix A.) This invariant leads to a more concise version of the termination conditions (see the corollary to Lemma 2 in Appendix A):

(1) $q_n \times B^{k-n} < v - low$, and

(2) $(1 - q_n) \times B^{k-n} < high - v$.

Since $0 \le q_n < 1$ and B^{k-n} becomes arbitrarily close to zero as n increases, termination condition (1) eventually holds; thus, the algorithm always terminates. Moreover, the invariant and the above termination conditions guarantee that the algorithm terminates with a number strictly between *low* and *high*. (See Theorem 3 in Appendix A.)

Having shown that the output rounds to v when input, we now show that the last digit of the output is correctly rounded, *i.e.*, output condition (2) holds. Because the algorithm chooses the closer of $0.d_1 \ldots d_n \times B^k$ and $0.d_1 \ldots d_{n-1}[d_n{+}1] \times B^k$, the last digit is correctly rounded. (See Theorem 4 in Appendix A.)

Finally, we show that no shorter output string rounds to v when input. Equivalently, no $(n-1)$-digit number (trailing zeros are allowed) also rounds to v when input. Suppose such a number, V', exists. Using the invariant, one can easily show that $0.d_1 \ldots d_{n-1} \times B^k$ and $0.d_1 \ldots d_{n-2}[d_{n-1}{+}1] \times B^k$ are the two $(n-1)$-digit numbers closest to v. Without loss of generality, we assume that V' is one of them. If V' is the first, termination condition (1) would have held at step $n-1$, a contradiction. If V' is the second, termination condition (2) would have held at step $n-1$, a contradiction. Therefore, no shorter output string rounds to v when input. (See Theorem 5 in Appendix A.)

[1] The notation $0.d_1 \ldots d_{n-1}[d_n{+}1]$ denotes $\dfrac{1}{B^n} + \sum_{i=1}^{n} \dfrac{d_i}{B^i}$. Informally, this represents the number formed by incrementing the last digit.

3 Implementation

The basic output algorithm presented in the preceding section can be implemented directly in Scheme using built-in exact rational arithmetic. The resulting code, however, runs slowly, especially for floating-point numbers with large exponents. The two main sources for the inefficiency are the high-precision rational arithmetic and the iterative search for the scaling factor k. Because the algorithm does not need the full generality of rational arithmetic (*i.e.*, there is no need to reduce fractions to lowest terms or to maintain separate denominators), it is more efficient to convert the algorithm to use high-precision integer arithmetic with an explicit common denominator. In this section we modify the algorithm to use high-precision integer arithmetic and a fast estimator for determining k.

3.1 Integer Arithmetic

In order to eliminate the high-precision rational arithmetic, we introduce an explicit common denominator so that the algorithm can use high-precision integer arithmetic. We also make use of the more concise termination conditions given in the preceding section.

Procedure:

1. Initialize r, s, m^+, and m^- such that $v = \frac{r}{s}$, $\frac{v - v^-}{2} = \frac{m^-}{s}$, and $\frac{v^+ - v}{2} = \frac{m^+}{s}$ according to Table 1.

2. Find the smallest integer k such that $\frac{r + m^+}{s} \le B^k$; *i.e.*, $k = \left\lceil \log_B \frac{r+m^+}{s} \right\rceil$.

3. If $k \ge 0$, let $r_0 = r$, $s_0 = s \times B^k$, $m_0^+ = m^+$, and $m_0^- = m^-$.

   If $k < 0$, let $r_0 = r \times B^{-k}$, $s_0 = s$, $m_0^+ = m^+ \times B^{-k}$, and $m_0^- = m^- \times B^{-k}$.

   Generate digits as follows:

$$
\begin{array}{lll}
d_1 = \left\lfloor \dfrac{r_0 \times B}{s_0} \right\rfloor & r_1 = \left\{ \dfrac{r_0 \times B}{s_0} \right\} & s_1 = s_0 \\
 & m_1^+ = m_0^+ \times B & m_1^- = m_0^- \times B \\
d_2 = \left\lfloor \dfrac{r_1 \times B}{s_1} \right\rfloor & r_2 = \left\{ \dfrac{r_1 \times B}{s_1} \right\} & s_2 = s_1 \\
 & m_2^+ = m_1^+ \times B & m_2^- = m_1^- \times B \\
\vdots & \vdots & \vdots
\end{array}
$$

   Invariants:

   (1) $v = \dfrac{r_n}{s_n} \times B^{k-n} + \sum_{i=1}^{n} d_i \times B^{k-i}$

   (2) $\dfrac{v - v^-}{2} = \dfrac{m_n^-}{s_n} \times B^{k-n}$

   (3) $\dfrac{v^+ - v}{2} = \dfrac{m_n^+}{s_n} \times B^{k-n}$

	$e \geq 0$		$e < 0$	
	$f \neq b^{p-1}$	$f = b^{p-1}$	$e = \min \exp$ or $f \neq b^{p-1}$	$e > \min \exp$ and $f = b^{p-1}$
r	$f \times b^e \times 2$	$f \times b^{e+1} \times 2$	$f \times 2$	$f \times b \times 2$
s	2	$b \times 2$	$b^{-e} \times 2$	$b^{-e+1} \times 2$
m^+	b^e	b^{e+1}	1	b
m^-	b^e	b^e	1	1

Table 1: Initial values of r, s, m^+, and m^-

4. Stop at the smallest n for which

   (1) $r_n < m_n^-$, or

   (2) $r_n + m_n^+ < s_n$

   If condition (1) is true and condition (2) is false, return $0.d_1 \ldots d_n \times B^k$.

   If condition (2) is true and condition (1) is false, return $0.d_1 \ldots d_{n-1}[d_n{+}1] \times B^k$.

   If both conditions are true, return the number that is closer to v, using some strategy to break ties. If $r_n \times 2 < s_n$, $0.d_1 \ldots d_n \times B^k$ is closer. If $r_n \times 2 > s_n$, $0.d_1 \ldots d_{n-1}[d_n{+}1] \times B^k$ is closer.

The invariants, which can be verified by a straightforward proof by induction, are useful in establishing the equivalence of this algorithm with the basic algorithm proved correct in Section 2.3.

If the input routine's rounding algorithm is known, V may be allowed to equal *low* or *high* or both. If *low* would round up to v when input, termination condition (1) would be $r_n \leq m_n^-$. If *high* would round down to v when input, termination condition (2) would be $r_n + m_n^+ \leq s_n$, and k would be the smallest integer such that $\frac{r+m^+}{s} < B^k$ (*i.e.*, $k = 1 + \left\lfloor \log_B \frac{r+m^+}{s} \right\rfloor$).

For IEEE unbiased rounding, if the mantissa, f, is even, then both *low* and *high* would round to v; otherwise, neither *low* nor *high* would round to v. For example, 10^{23} falls exactly between two IEEE floating-point numbers, the smaller of which has an even mantissa; thus, 10^{23} rounds to the smaller when input. By accommodating unbiased rounding, the algorithm prints this number as `1e23` instead of `9.999999999999999e22`.

Figure 1 gives Scheme code for the algorithm. It uses an iterative algorithm (*scale*) similar to the one presented in [5] to find k. It assumes the input routine uses IEEE unbiased rounding. In the case of a tie in determining d_n, it always rounds up by choosing $d_n + 1$. The function *flonum→digits*returns a pair whose first element is k and whose second element is the list of digits.

3.2 Efficient Scaling

Steele and White's iterative algorithm requires $O(|\log v|)$ high-precision integer operations to compute k, r_0, s_0, m_0^+, and m_0^-. An obvious alternative is to use the floating-point logarithm function to approximate k with $\lceil \log_B v \rceil$ and then use an efficient algorithm to compute the appropriate power of B by which to multiply either s or r, m^+, and m^-. Because the floating-point logarithm may be slightly smaller or larger than the true logarithm, a small constant (chosen to be slightly greater than the largest possible error) is subtracted from the floating-point logarithm so that the ceiling of the result will be either k or $k-1$. Consequently, the estimate must be checked and adjusted by one if necessary.

Figure 2 shows Scheme code that finds k and scales the numbers using just a few high-precision integer operations. The new *scale* procedure takes an additional argument, v. The code uses a table to look up the value of 10^k for $0 \leq k \leq 325$, which is sufficient to handle all IEEE double-precision floating-point numbers. It also uses a table to look up the value of $\frac{1}{\log B}$ for $2 \leq B \leq 36$ in order to speed up the computation of $\log_B v$.

If the cost of the floating-point logarithm function is fairly high, it may be more efficient to compute a less accurate approximation to the logarithm. Because in almost all floating-point representations the input base, b, is two (or a power of two), we assume that $b = 2$ for our discussion of logarithm estimators. We also assume that $B > 2$, because there is no reason to use a conversion algorithm if the output base is the same as the input base.

Since $v = f \times 2^e$, $\log_2 v = \log_2 f + e$. If we compute the integer s and floating-point number x such that $v = x \times 2^s$ and $1 \leq x < 2$, we get $\log_2 v = \log_2 x + s$, where $0 \leq \log_2 x < 1$. In other words, s is the integer part of the base-2 logarithm of v. Let $len(f)$ be the length of f in bits. Then $s = e + len(f) - 1$. For normalized floating-point numbers, we have $s = e + p - 1$.

In order to estimate $\log_B v = \frac{\log_2 v}{\log_2 B}$, we use $\frac{s}{\log_2 B}$. This estimate never overshoots $\log_B v$, and it undershoots by no more than $\frac{1}{\log_2 3} < 0.631$. Once again, floating-point arithmetic does not compute the exact value of $\frac{s}{\log_2 B}$, so we subtract a small constant in order to preserve the property that the estimate never overshoots. Assuming the estimate is computed using IEEE double-precision floating-point arithmetic, $\frac{1}{\log_2 B}$ can be represented with an error of less than 10^{-14}. Since s is between -1074 and 1023, the floating-point result of $s \times \frac{1}{\log_2 B}$ has an error of less than 10^{-10}. Because our estimate never overshoots k and the error is less than one, $\left\lceil \frac{s}{\log_2 B} \right\rceil$ is k or $k-1$. This result also holds if k is $1 + \left\lfloor \log_B \frac{r+m^+}{s} \right\rfloor$.

Whereas the floating-point logarithm estimate was almost always k, our simpler estimate is frequently $k-1$. Having the estimate off by one introduces extra overhead, but this overhead can be eliminated. When the estimate is $k-1$, *fixup* multiplies s by B and then calls *generate* to generate the digits. On entry to *generate*, r, m^+, and m^- are multiplied by B. By moving these multiplications back into the call sites of *generate*, the multiplications can be eliminated in *fixup* when the estimator returns $k-1$. The

```
(define flonum→digits
  (lambda (v f e min-e p b B)
    (let ([round? (even? f)])
      (if (>= e 0)
          (if (not (= f (expt b (- p 1))))
              (let ([be (expt b e)])
                (scale (* f be 2) 2 be be 0 B round? round?))
              (let* ([be (expt b e)] [be1 (* be b)])
                (scale (* f be1 2) (* b 2) be1 be 0 B round? round?)))
          (if (or (= e min-e) (not (= f (expt b (- p 1)))))
              (scale (* f 2) (* (expt b (- e)) 2) 1 1 0 B round? round?)
              (scale (* f b 2) (* (expt b (- 1 e)) 2) b 1 0 B round? round?))))))

(define scale
  (lambda (r s m+ m- k B low-ok? high-ok?)
    (cond
     [((if high-ok? >= >) (+ r m+) s) ; k is too low
      (scale r (* s B) m+ m- (+ k 1) B low-ok? high-ok?)]
     [((if high-ok? < <=) (* (+ r m+) B) s) ; k is too high
      (scale (* r B) s (* m+ B) (* m- B) (- k 1) B low-ok? high-ok?)]
     [else ; k is correct
      (cons k (generate r s m+ m- B low-ok? high-ok?))])))

(define generate
  (lambda (r s m+ m- B low-ok? high-ok?)
    (let ([q-r (quotient-remainder (* r B) s)]
          [m+ (* m+ B)]
          [m- (* m- B)])
      (let ([d (car q-r)]
            [r (cdr q-r)])
        (let ([tc1 ((if low-ok? <= <) r m-)]
              [tc2 ((if high-ok? >= >) (+ r m+) s)])
          (if (not tc1)
              (if (not tc2)
                  (cons d (generate r s m+ m- B low-ok? high-ok?))
                  (list (+ d 1)))
              (if (not tc2)
                  (list d)
                  (if (< (* r 2) s)
                      (list d)
                      (list (+ d 1))))))))))
```

Figure 1: Scheme code that implements the basic conversion algorithm with an iterative scaling procedure and IEEE unbiased rounding (round to even). For other rounding modes, *scale* and *generate* may be called with different values for *low-ok?* and *high-ok?*.

result is that there is no penalty for an estimate that is off by one. Figure 3 gives a Scheme implementation of our estimator and the modified digit-generation loop. It modifies the original *scale* function to take additional arguments f and e, and it uses a table to look up the value of $\frac{1}{\log_2 B}$ for $2 \leq B \leq 36$.

Table 2 gives the relative CPU times for Steele and White's iterative scaling algorithm [5] and the floating-point logarithm scaling algorithm with respect to our simple estimate and scaling algorithm. The timings were performed using *Chez Scheme* on a DEC AXP 8420 running Digital UNIX V3.2C. The input was a set of 250,680 positive normalized IEEE double-precision floating-point numbers, and the output base was ten. This set was generated according to the forms Schryer developed for testing floating-point units [4]. As expected, the timings show that the iterative scaling algorithm is almost two orders of magnitude slower than either estimate-based algorithm.

Scaling Algorithm	*Relative CPU Time*
Steele & White	70.0
floating-point log	1.03
log approximation	1.00

Table 2: Relative CPU times for three different scaling algorithms

```
(define scale
  (lambda (r s m+ m- k B low-ok? high-ok? v)
    (let ([est (inexact->exact (ceiling (- (logB B v) 1e-10)))])
      (if (>= est 0)
          (fixup r (* s (exptt B est)) m+ m- est B low-ok? high-ok?)
          (let ([scale (exptt B (- est))])
            (fixup (* r scale) s (* m+ scale) (* m- scale) est B low-ok? high-ok?))))))

(define fixup
  (lambda (r s m+ m- k B low-ok? high-ok?)
    (if ((if high-ok? >= >) (+ r m+) s) ; too low?
        (cons (+ k 1) (generate r (* s B) m+ m- B low-ok? high-ok?))
        (cons k (generate r s m+ m- B low-ok? high-ok?)))))

(define exptt
  (let ([table (make-vector 326)])
    (do ([k 0 (+ k 1)] [v 1 (* v 10)])
        ((= k 326))
      (vector-set! table k v))
    (lambda (B k)
      (if (and (= B 10) (<= 0 k 325))
          (vector-ref table k)
          (expt B k)))))

(define logB
  (let ([table (make-vector 37)])
    (do ([B 2 (+ B 1)])
        ((= B 37))
      (vector-set! table B (/ (log B))))
    (lambda (B x)
      (if (<= 2 B 36)
          (* (log x) (vector-ref table B))
          (/ (log x) (log B))))))
```

Figure 2: Scheme code that uses the floating-point logarithm function to estimate k and then adjusts the result to the exact value of k

4 Fixed-Format Output

Up to this point we have addressed the free-format output problem. We now describe how to modify the basic algorithm to generate fixed-format output. A key property of the output conversion algorithm is its use of the rounding range of v, determined by computing v^+ and v^-. For fixed-format output, this range is conditionally modified to indicate the requested precision. If a floating-point number has enough precision to be printed to the given digit position, the rounding range is expanded so that the output will stop at the given position. If a floating-point number has insufficient precision, the rounding range is not expanded, and the output will contain # marks past the last significant digit.

There are two ways of specifying how many digits to print in fixed-format mode: by absolute digit position and by relative digit position. An absolute digit position is the distance from the radix point in base-B digits at which one wants the output to stop. A relative digit position is the number of base-B digits to print.

Suppose an absolute digit position is given. Let j be the digit position and v be a positive floating-point number. In order for the output, V, to be correctly rounded, $v - \frac{B^j}{2} \leq V \leq v + \frac{B^j}{2}$. Because of the gaps in the floating-point representation, all numbers between $\frac{v^- + v}{2}$ and $\frac{v+v^+}{2}$ are indistinguishable from v. The algorithm uses the larger range in order to determine when to stop generating digits. In other words, let *low* be the lesser of $\frac{v^- + v}{2}$ and $v - \frac{B^j}{2}$, and let *high* be the greater of $\frac{v+v^+}{2}$ and $v + \frac{B^j}{2}$.

After *low* and *high* are computed, the scaling factor k is determined as before. If the end point *high* is in the rounding range (*i.e.*, if $high = v + \frac{B^j}{2}$), k is the smallest integer such that $high < B^k$; *i.e.*, $k = 1 + \lfloor \log_B high \rfloor$. Otherwise, k is the smallest integer such that $high \leq B^k$; *i.e.*, $k = \lceil \log_B high \rceil$.

The digits are generated as before. Termination condition (1) is extended to include equality when the end point *low* is in the rounding range. Similarly, termination condition (2) is extended to include equality when the end point *high* is in the rounding range.

Let n be the smallest integer for which one of the termination conditions holds. As before, digit d_n is incremented when $0.d_1 \ldots d_{n-1}[d_n{+}1] \times B^k$ is closer to v than $0.d_1 \ldots d_n \times B^k$ (or possibly in the case of a tie). If $j = k - n$, the algorithm stopped at the desired digit position, so the algorithm simply returns the result. Because of the way we defined the termination conditions, the algorithm cannot generate too many digits. Therefore, if $j \neq k - n$, $j < k - n$, so the algorithm must generate the remaining digits.

Unfortunately the algorithm cannot simply print # marks

```
(define scale
  (lambda (r s m+ m- k B low-ok? high-ok? f e)
    (let ([est (inexact->exact (ceiling (- (* (+ e (len f) -1) (invlog2of B)) 1e-10)))])
      (if (>= est 0)
          (fixup r (* s (exptt B est)) m+ m- est B low-ok? high-ok?)
          (let ([scale (exptt B (- est))])
            (fixup (* r scale) s (* m+ scale) (* m- scale) est B low-ok? high-ok?))))))

(define fixup
  (lambda (r s m+ m- k B low-ok? high-ok?)
    (if ((if high-ok? >= >) (+ r m+) s) ; too low?
        (cons (+ k 1) (generate r s m+ m- B low-ok? high-ok?))
        (cons k (generate (* r B) s (* m+ B) (* m- B) B low-ok? high-ok?)))))

(define generate
  (lambda (r s m+ m- B low-ok? high-ok?)
    (let ([q-r (quotient-remainder r s)])
      (let ([d (car q-r)]
            [r (cdr q-r)])
        (let ([tc1 ((if low-ok? <= <) r m-)]
              [tc2 ((if high-ok? >= >) (+ r m+) s)])
          (if (not tc1)
              (if (not tc2)
                  (cons d (generate (* r B) s (* m+ B) (* m- B) B low-ok? high-ok?))
                  (list (+ d 1)))
              (if (not tc2)
                  (list d)
                  (if (< (* r 2) s)
                      (list d)
                      (list (+ d 1))))))))))

(define invlog2of
  (let ([table (make-vector 37)]
        [log2 (log 2)])
    (do ([B 2 (+ B 1)])
        ((= B 37))
      (vector-set! table B (/ log2 (log B))))
    (lambda (B)
      (if (<= 2 B 36)
          (vector-ref table B)
          (/ log2 (log B))))))
```

Figure 3: Scheme code that uses our fast estimator and modified digit-generation loop

from here until position j. Suppose 100 were printed to absolute position 0, for example. Termination condition (1) would hold after generating the first digit, but the remaining digit positions are significant and must therefore be zero, not #. Consequently, the algorithm must generate zeroes as long as they are significant and then generate # marks. A digit is insignificant when it and all the digits after it can be replaced by any base-B digits without altering the value of the number when input. In other words, a digit is insignificant if incrementing the preceding digit does not cause the number to fall outside the rounding range of v.

If $low = v - \frac{B^j}{2}$ and $high = v + \frac{B^j}{2}$, the remaining digit positions are all significant, so the algorithm fills them with zeroes and returns. Otherwise, the precision of the output is limited by the floating-point representation. The algorithm generates zeroes until incrementing the preceding digit would result in a number less than or equal to *high*, at which point it fills the remaining digit positions with # marks. For example, when printing 100 in IEEE double-precision to digit position -20, the algorithm prints `100.000000000000000#####`.

Now suppose a relative position is given instead. Let $i > 0$ be the number of digits requested. In order to compute the corresponding absolute digit position, j, the algorithm first computes the absolute position of the first digit. Unfortunately, the position of the first digit, $k-1$, may depend on the upper bound of the rounding range of v, which in turn may depend on j. This cycle is resolved by using an initial estimate for k that does not depend on j and then refining it when necessary. The initial estimate, $\hat{k}$, is $\left\lceil \log_B \frac{v+v^+}{2} \right\rceil$, which can be computed efficiently using the techniques described in Section 3.2. If $v + \dfrac{B^{\hat{k}-i}}{2} < B^{\hat{k}}$, the initial estimate was correct, so $k = \hat{k}$; otherwise, the initial estimate was off by one, so $k = \hat{k} + 1$. At this point the algorithm proceeds

System	*Free* / *Fixed*	*Fixed* / `printf`	*Incorrect*
Alpha AXP	1.66	2.94	242
HP 9000	1.61	2.19	317
Linux	1.63	0.58	0
RS/6000	1.75	4.46	0
SGI 32	1.61	5.69	186
SGI 64	1.81	3.12	6280
Solaris	1.59	0.68	0
Sun4c	1.66	0.54	0
Sun4d	1.62	0.38	0
Geom. mean	1.66	1.51	N/A

Key:

Alpha AXP—DEC AXP 8420, Digital UNIX V3.2C
HP 9000—HP 9000/715/E, HP-UX A.09.05
Linux—AMD 80486DX2/80, Linux 1.3.32
RS/6000—IBM RS/6000 7013/560, AIX 3.2
SGI 32—SGI IP22, IRIX 5.3
SGI 64—SGI IP21, IRIX64 6.1
Solaris—Sun SPARCstation 2, SunOS 5.5 (Solaris)
Sun4c—Sun SPARCstation 2, SunOS 4.1.3
Sun4d—Sun SPARCstation 5, SunOS 4.1.3

Table 3: Ratio of CPU time for free-format versus straightforward fixed-format, fixed-format versus `printf`, and the count of incorrectly rounded output from `printf` on 250,680 floating-point numbers

as though it were given the absolute digit position $k - i$.

The rational arithmetic used in fixed-format printing can be converted into high-precision integer arithmetic by introducing a common denominator as before. Because there are several more cases to consider, however, the resulting code is lengthy and has therefore been omitted from this paper.

5 Conclusion

We have developed an efficient algorithm for converting floating-point numbers from an input base to an output base. For free-format output, it provably generates the shortest, correctly rounded number that rounds to the original floating-point number when input, taking the input rounding algorithm into account if desired. For fixed-format output, it generates a correctly rounded number with # marks in the place of insignificant trailing digits. These # marks are useful when the requested number of digits may exceed the internal precision. Our algorithm employs a fast estimator to compute scaling factors. By modifying our algorithm slightly, we eliminated the penalty of having the estimate off by one, which enabled us to make our estimator very inexpensive.

We have compared an implementation of our free-format algorithm for base-10 output against an implementation of a straightforward fixed-format algorithm on several different systems. For this test, we used a set of 250,680 positive normalized IEEE double-precision floating-point numbers [4]. The fixed-format algorithm printed them to 17 significant digits, the minimum number guaranteed to distinguish among IEEE double-precision numbers. In all cases the numbers were printed to `/dev/null` in order to factor out I/O performance. The average number of digits needed is 15.2, so the free-format algorithm has no particular advantage over the fixed-format algorithm.

Table 3 shows that our free-format algorithm takes 66% more CPU time on average than the straightforward fixed-format algorithm. To provide a basis of comparison against a standard fixed-format algorithm for each system, the table also compares the C library's `printf` function against the straightforward fixed-format algorithm and gives the number of floating-point numbers that were rounded incorrectly by `printf`. For the systems where `printf` is considerably faster, we suspect that our implementation could be tuned to achieve comparable results. (In particular, our current implementation uses 64-bit arithmetic and performs poorly on systems without efficient 64-bit support.) While the cost of free-format output may be significant for some applications, the cost is justified for many others by the reduced verbosity of free-format output.

Our algorithm is based on Steele and White's conversion algorithm [5]. Ours is dramatically more efficient, primarily due to our use of a fast estimator for computing scaling factors. Their algorithm does not distinguish between significant and insignificant trailing zeros, nor does it take into account input rounding modes. In addition, their fixed-format algorithm introduced a slight inaccuracy in the computation of the rounding range.

David Gay independently developed an estimator similar to ours [2]. It uses the first-degree Taylor series to estimate $\log_{10} v$. Although our estimator is less accurate than his, it is less expensive as well, requiring two rather than five floating-point operations. Furthermore, since our scaling algorithm incurs no additional overhead when the estimate is off by one, the loss of accuracy is unimportant, and scaling is more efficient in all cases.

Gay also developed an excellent set of heuristics for determining when more efficient digit-generation techniques can be employed for fixed-format output. In particular, he showed that floating-point arithmetic is sufficiently accurate in most cases when the requested number of digits is small. The fixed-format printing algorithm described in this paper is useful when these heuristics fail.

An implementation of the algorithms described in this paper is available from the authors. A version of the free-format algorithm has been used in *Chez Scheme* since 1990; in fact, the ANSI/IEEE Scheme standard requirement for accurate, minimal-length numeric output and the desire to do so as efficiently as possible in *Chez Scheme* motivated the work reported here.

References

[1] William D. Clinger. How to read floating-point numbers accurately. *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, 25(6):92–101, June 1990.

[2] David M. Gay. Correctly rounded binary-decimal and decimal-binary conversions. Numerical Analysis Manuscript 90-10, AT&T Bell Laboratories, Murray Hill, New Jersey 07974, November 1990.

[3] IEEE standard for binary floating-point arithmetic. ANSI/IEEE Std 754-1985, Institute of Electrical and Electronics Engineers, New York, 1985.

[4] N. L. Schryer. A test of a computer's floating-point arithmetic unit. In W. Cowell, editor, *Sources and Development of Mathematical Software*. Prentice-Hall, 1981.

[5] Guy L. Steele Jr. and Jon L. White. How to print floating-point numbers accurately. *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, 25(6):112–126, June 1990.

A Proofs of Correctness

This section presents correctness proofs of the free-format printing algorithm described in Section 2.2. See Section 2.3 for a less formal presentation.

Theorem 1: Each d_i is a valid base-B digit, $d_1 > 0$, and if d_n is incremented, no carry is generated.

Proof: $0 < q_0 = \frac{v}{B^k} < 1$ since $0 < v < high \le B^k$. For $i \ge 1$, $0 \le q_i < 1$ by definition. Thus for all $i \ge 0$, $0 \le q_i \times B < B$, so $d_{i+1} = \lfloor q_i \times B \rfloor$ is a valid base-B digit.

Suppose $d_1 = 0$. Then $0.[d_1{+}1] \times B^k = B^{k-1} < high$ by the minimality of k, so termination condition (2) holds. Termination condition (1) cannot hold since $0.d_1 \times B^k = 0$. Thus digit d_1 will be incremented to 1.

Suppose $d_1 = B - 1$. Then $0.[d_1{+}1] \times B^k = B^k \ge high$, so termination condition (2) will not hold, and digit d_1 will not be incremented.

Assume by way of contradiction that final digit d_n ($n > 1$) is incremented to B, which would introduce a carry. Then $0.d_1 \ldots d_{n-1}[d_n{+}1] \times B^k = 0.d_1 \ldots d_{n-2}[d_{n-1}{+}1] \times B^k < high$, so termination condition (2) would have held at step $n - 1$, a contradiction. □

Lemma 2: $v = q_n \times B^{k-n} + \sum_{i=1}^{n} d_i \times B^{k-i}$

Proof: By induction on n.

Basis: $v = q_0 \times B^k$ by definition of q_0.

Induction: Suppose the result holds for n.

$$\begin{aligned}
v &= q_n \times B^{k-n} + \sum_{i=1}^{n} d_i \times B^{k-i} \\
&= (q_n \times B) \times B^{k-(n+1)} + \sum_{i=1}^{n} d_i \times B^{k-i} \\
&= (\lfloor q_n \times B \rfloor + \{q_n \times B\}) \times B^{k-(n+1)} + \\
&\quad \sum_{i=1}^{n} d_i \times B^{k-i} \\
&= (d_{n+1} + q_{n+1}) \times B^{k-(n+1)} + \sum_{i=1}^{n} d_i \times B^{k-i} \\
&= q_{n+1} \times B^{k-(n+1)} + \sum_{i=1}^{n+1} d_i \times B^{k-i}
\end{aligned}$$

□

Corollary: The following conditions are equivalent to the termination conditions:

(1) $q_n \times B^{k-n} < v - low$

(2) $(1 - q_n) \times B^{k-n} < high - v$

Theorem 3: (Information Preservation) The algorithm always terminates with $low < V < high$.

Proof: By Lemma 2, termination condition (1) is equivalent to $q_n \times B^{k-n} < v - low$. Since $0 \le q_n < 1$, the left-hand side becomes arbitrarily small as n increases, so there will be some n for which the algorithm terminates.

Suppose the algorithm stops at point n. There are two cases to consider:

1. $V = 0.d_1 \ldots d_n \times B^k > low$
   By Lemma 2, $V = v - q_n \times B^{k-n}$. Since $0 \le q_n < 1$, $low < V \le v < high$.
2. $V = 0.d_1 \ldots d_{n-1}[d_n{+}1] \times B^k < high$
   By Lemma 2, $V = v + (1 - q_n) \times B^{k-n}$. Since $0 \le q_n < 1$, $low < v < V < high$.

In both cases $low < V < high$. □

Theorem 4: (Correct Rounding) $|V - v| \le \dfrac{B^{k-n}}{2}$

Proof: There are two cases to consider:

1. $V = 0.d_1 \ldots d_n \times B^k$
   Since d_n was not incremented, $0.d_1 \ldots d_{n-1}[d_n{+}1] \times B^k$ was no closer to v than V.
   Thus $0.d_1 \ldots d_{n-1}[d_n{+}1] \times B^k - v = (V + B^{k-n}) - v \ge v - V$, which is equivalent to $v - V \le \dfrac{B^{k-n}}{2}$.
2. $V = 0.d_1 \ldots d_{n-1}[d_n{+}1] \times B^k$
   Since d_n was incremented, $0.d_1 \ldots d_n \times B^k$ was no closer to v than V. Thus $v - 0.d_1 \ldots d_n \times B^k = v - (V - B^{k-n}) \ge V - v$, which is equivalent to $V - v \le \dfrac{B^{k-n}}{2}$.

In both cases $|V - v| \le \dfrac{B^{k-n}}{2}$. □

Theorem 5: (Minimum-Length Output) There is no $(n-1)$-digit base-B number V' such that $low < V' < high$.

Proof: Assume by way of contradiction that V' exists. By Lemma 2, $v - 0.d_1 \ldots d_{n-1} \times B^k = q_{n-1} \times B^{k-(n-1)}$, so $v - 0.d_1 \ldots d_{n-1} \times B^k < \frac{1}{B^{n-1}}$ and $0.d_1 \ldots d_{n-2}[d_{n-1}{+}1] \times B^k - v < \frac{1}{B^{n-1}}$. Thus $0.d_1 \ldots d_{n-1} \times B^k$ and $0.d_1 \ldots d_{n-2}[d_{n-1}{+}1] \times B^k$ are the two closest $(n-1)$-digit base-B numbers to v.[2] Consequently, there are two cases to consider:

1. $V' = 0.d_1 \ldots d_{n-1} \times B^k$
   Since the algorithm did not stop at point $n - 1$, $V' = 0.d_1 \ldots d_{n-1} \not> low$, a contradiction.
2. $V' = 0.d_1 \ldots d_{n-2}[d_{n-1}{+}1] \times B^k$
   Since the algorithm did not stop at point $n - 1$, $V' = 0.d_1 \ldots d_{n-2}[d_{n-1}{+}1] \times B^k \not< high$, a contradiction.

□

[2] Note that if incrementing the last digit introduces a carry, the resulting number may extend to the left by one digit. This does not cause a problem, however, since the last digit would be 0 and can be eliminated.