AT&T
Bell Laboratories

Numerical Analysis Manuscript 90-10

Correctly Rounded Binary-Decimal and Decimal-Binary Conversions

David M. Gay

November 30, 1990

Correctly Rounded Binary-Decimal and Decimal-Binary Conversions

David M. Gay

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This note discusses the main issues in performing correctly rounded decimal-to-binary and binary-to-decimal conversions. It reviews recent work by Clinger and by Steele and White on these conversions and describes some efficiency enhancements. Computational experience with several kinds of arithmetic suggests that the average computational cost for correct rounding can be small for typical conversions. Source for conversion routines that support this claim is available from *netlib*.

1. Introduction

On computers that use non-decimal floating-point arithmetic, occasion often arises to convert a number from decimal form to the internal floating-point form or vice versa, i.e., to find an internal floating-point number that is near to a given decimal number or to find a decimal number that is near to a given internal floating-point number. Ideally such a conversion would be correctly rounded in a specified sense, e.g., would yield the nearest number expressible with at most a prescribed number of digits of the appropriate kind. It is of interest to ask how much correctly rounded conversions cost computationally. Here we examine this question for some commonly used arithmetics: IEEE binary [1], IBM-mainframe, and VAX. For IEEE arithmetic, we assume that the specified rounding sense is the IEEE round-nearest mode, i.e., unbiased rounding, which yields a nearest floating-point number and, in case there are two nearest numbers, yields the one whose trailing digit is even. For the other arithmetics, we assume biased rounding, which yields the floating-point number of larger magnitude when there are two nearest floating-point numbers. (Note that IBM arithmetic is not ordinarily rounded, but that, as illustrated in Appendix A, extended-precision instructions can be used to compute floating-point products and quotients rounded in this way.)

Recently, Clinger [3] described a scheme that relies on IEEE double-extended arithmetic to perform correctly rounded decimal-to-binary conversion. In the next section we show how to modify Clinger's method so that only double-precision arithmetic is needed. Steele and White [12] have recently described a method for binary-to-decimal conversion that yields the shortest decimal number that, when correctly rounded back to the same binary floating-point precision, results in the original number; they also describe variants that yield correctly rounded decimal strings having at most a prescribed number of digits or a prescribed number of digits after the decimal point (fixed-point notation). In Section 3 we explain several ways to speed up such calculations. Section 4 describes some computational experience, and §5 offers concluding remarks. The rest of this section introduces some notation and assumptions. For other work on conversions, see Coonen's thesis [4] and the works cited in [3, 4, 12].

Since signs are easy to treat, we restrict the discussion to conversion of nonnegative numbers.

We assume that nonnegative normalized internal floating-point numbers have the form

$$b = \sum_{i=0}^{p-1} b_i \times \beta^{e-i}, \tag{1}$$

where the floating-point arithmetic base β and the number p of base β digits are fixed positive integers, e is an integer that satisfies

$$e_{\min} \le e \le e_{\max} \tag{2}$$

for prescribed integers $e_{\min}$ and $e_{\max}$, the digits b_i are integers with $0 \le b_i < \beta$, and $b_0 \ne 0$. For IEEE arithmetic, it is also necessary to consider *denormalized* numbers, in which $e = e_{\min}$, $b_0 = 0$, and $b \ne 0$.

Corresponding to (1), we consider nonnegative decimal numbers of the form

$$d = \sum_{i=0}^{n-1} d_i \times 10^{k-i}, \tag{3}$$

where n is a positive integer and k is an integer that may be positive, negative, or zero. We assume that β is not commensurable with 10 in the sense of Matula [10], i.e., that β is not a power of 10, since conversions are easy if β is a power of 10.

A *floating-point integer* is an integer that can be expressed in the form (1).

Suppose b is the nearest value of the form (1) to d when the bounds (2) are ignored. If $e > e_{\max}$, then d is said to *overflow*, and if $e < e_{\min}$, then d is said to *underflow*. Under IEEE arithmetic, we allow *gradual underflow*: if

$$\beta^{e_{\min}} - \beta^{e_{\min}-p} > d \ge \beta^{e_{\min}-p+1}/2,$$

we consider the desired approximation b to d to be the denormalized number nearest d.

Suppose an (exact) arithmetic operation yields a result r with $\beta^{e_{\min}} - \beta^{e_{\min}-p+1} < r < \beta^{e_{\max}+1} - \beta^{e_{\max}-p+1}$ and that r is not a floating-point number, i.e., cannot be represented in the form (1). Then there are two adjacent floating-point numbers, $\underline{r}$ and $\overline{r}$, with $\underline{r} < r < \overline{r}$. By a *rounded floating-point operation*, we mean a calculation that yields $\underline{r}$ if $r - \underline{r} < \overline{r} - r$, $\overline{r}$ if $\overline{r} - r < r - \underline{r}$, and, if $r - \underline{r} = \overline{r} - r$, $\underline{r}$ if we are using IEEE arithmetic and the least-significant digit of $\underline{r}$ is even and $\bar{r}$ otherwise. With IEEE and VAX arithmetic, the hardware arithmetic operations are rounded floating-point operations in the sense just defined. With IBM arithmetic, one can use extended-precision instructions, such as those shown in Appendix A, to compute rounded floating-point products and quotients.

2. Decimal-to-Binary Conversion

Suppose we wish to find the nearest floating-point number b to a given decimal number d. Initially we assume that

$$\beta^{e_{\min}} - \beta^{e_{\min} - p + 1} \le d \le \beta^{e_{\max} + 1} - \beta^{e_{\max} - p + 1}, \tag{4}$$

so that over- and underflow are not an issue. As Clinger [3] notes, if n is small enough and k is near enough to zero, then d can be expressed as the product or ratio of the floating-point integers $10^{|k - n + 1|}$ and $\sum_{i=1}^{n} d_{i-1} \times 10^{n-i}$. Thus we can compute the correctly rounded b with one rounded floating-point operation. Another case (not noted by Clinger) where a single rounded floating-point operation suffices is that where k is large enough that 10^k is not a floating-point integer, but there is a positive integer $j < k$ such that both 10^{k-j} and $10^j \cdot \left[\sum_{i=1}^{n} d_{i-1} \times 10^{n-i} \right]$ are floating-point integers. Fortunately, these cases are the typical ones in most applications.

If the above cases do not obtain, then we can use a combination of floating-point arithmetic and arithmetic on large integers to compute b. A reasonable approach is to compute a series of approximations, say $b^{(1)}, b^{(2)}, \cdots, b^{(m)}$, with

$$b^{(j)} = \sum_{i=0}^{p-1} b_i^{(j)} \times \beta^{e^{(j)} - i}$$

and $b^{(m)} = b$. Clinger suggests using IEEE binary double-extended arithmetic to compute $b^{(1)}$; he describes a procedure that changes $b^{(j)}$ by one unit in the last place (i.e., that adds or subtracts $\beta^{e^{(j)} - p + 1}$ to or from $b^{(j)}$), and he argues that his choice of $b^{(1)}$ is such that at most one iteration of the correction procedure is needed, i.e., $m \le 2$. Unfortunately, many systems that have IEEE arithmetic do not have hardware support for IEEE double-extended arithmetic, and many other systems have non-IEEE arithmetic. However, it is possible to use the native floating-point arithmetic together with high-precision integer calculations in a correction procedure that yields $b^{(m)} = b$ with $m \le 3$, and that usually has $m \le 2$.

It is easy to use floating-point arithmetic to compute an initial guess $b^{(1)}$ with

$$|b - b^{(1)}| < c\beta^{e^{(1)} - p + 1} \tag{5}$$

for a small constant c. For example, §4 describes an easy way to achieve $c = 6.01$ on machines with IEEE, VAX, or IBM arithmetic; 6.01 is an upper bound on $(1 + \beta^{1-p})^6 - 1$ (as in the roundoff analysis in §20 of [7]).

Let $\underline{b} < b$ and $\overline{b} > b$ be the floating-point numbers adjacent to b. For b to be the desired approximation to d, it is necessary and sufficient that $\underline{b} < d < \overline{b}$ and $|b - d| \le \min\{d - \underline{b}, \overline{b} - d\}$, with b chosen properly if equality holds (i.e., b_{p-1} even for unbiased rounding and $b - d = d - \underline{b}$ for biased rounding). Thus for $b^{(j)} = b$, it is necessary that

$$(6) \qquad |d - b^{(j)}| \le \tfrac{1}{2}\beta^{e^{(j)} - p + 1}.$$

In the special case that $b = \beta^e$, it is also necessary that

$$(7) \qquad d - b^{(j)} \ge -\tfrac{1}{2}\beta^{e^{(j)} - p}.$$

Condition (6) is easily reduced to a test that only requires multiplying and subtracting large integers; it is equivalent to

$$(8) \qquad |2M(d - b^{(j)})| \le M\beta^{e^{(j)} - p + 1},$$

where M is chosen so that $2Md$, $2Mb^{(j)}$, and $M\beta^{e^{(j)} - p + 1}$ are all integers. For example, $M := \max\{1, \beta^{p - e^{(j)} - 1}\} \cdot \max\{10^{n-k-1}, 1\}$ suffices, though, as illustrated in §4, smaller choices of M are often possible. Suppose (8) fails to hold, and consider

$$\tilde{\delta}^{(j)} := \tfrac{1}{2} \times [2M(d - b^{(j)})]/[M\beta^{e^{(j)} - p + 1}];$$

note that $b^{(j)} + \tilde{\delta}^{(j)}\beta^{e^{(j)} - p + 1} = d$, so $\tilde{\delta}^{(j)}$ is the number of units in the last place of $b^{(j)}$ by which we would like to perturb $b^{(j)}$. With floating-point arithmetic that starts by obtaining floating-point approximations to the integers $2M(d - b^{(j)})$ and $M\beta^{e^{(j)} - p + 1}$, we can compute an excellent approximation $\delta^{(j)}$ to $\tilde{\delta}^{(j)}$ and then a new approximation $b^{(j+1)}$ to b, i.e., $b^{(j+1)} = fl(b^{(j)} + \delta^{(j)}\beta^{e^{(j)} - p + 1})$, where $fl(\cdot)$ denotes an approximation to $(\cdot)$ computed with floating-point arithmetic. If $e^{(j+1)} = e$ (the correct exponent for the desired floating-point approximation b) and either $b^{(j)}$ is not a power of β or $\delta^{(j)} > 0$, then $b^{(j+1)}$ will differ from b by at most one unit in the last place, and if $\delta^{(j)}$ differs from the nearest odd multiple of ½ by more than a modest multiple of β^{1-p}, then $b^{(j+1)} = b$.

It could happen that $b^{(j+1)} = b^{(j)} \ne b$ if b differs from $b^{(j)}$ by $\pm\beta^{e^{(j)} - p + 1}$ and d differs from $b^{(j)}$ by very nearly $\tfrac{1}{2}\beta^{e^{(j)} - p + 1}$. To preclude the infinite loop that might otherwise result, it suffices to replace $\delta^{(j)}$ by $\operatorname{sign}(\delta^{(j)}) \max\{|\delta^{(j)}|, \beta^{e^{(j)} - p + 1}\}$. Similarly, if $b^{(j)}$ is a power of β and $b^{(j)} - d$ is very close to $\tfrac{1}{2}\beta^{e^{(j)} - p}$, it could happen that $b^{(j+1)} = b^{(j)}$ unless $\delta^{(j)}$ is suitably modified; in this special case (when $b^{(j)}$ is a power of β and $\delta^{(j)} < 0$), it suffices to replace $\delta^{(j)}$ by $-\max\{|\delta^{(j)}|, \beta^{e^{(j)} - p}\}$.

In the worst case, the procedure just described yields $b^{(m)} = b$ with $m = 3$. This could happen if $|b^{(2)} - b| = \beta^{e-p+1}$, i.e., $b^{(2)}$ differs from b by one unit in the last place of b. Otherwise the procedure yields $b = b^{(1)}$ or $b = b^{(2)}$. (Whenever $\delta^{(j)}$ is modified as described in the previous paragraph, $b^{(j+1)} = b$.)

Consider IEEE arithmetic for a moment and suppose the left-hand side of (4) is relaxed to $\beta^{e_{\min} - p + 1}$. To handle denormalized numbers, it suffices to adjust the exponents of β in (6), (7), and (8) appropriately. Specifically, if L_j is the least integer such that $b^{(j)}_{L_j} \ne 0$, then we replace p by $(p - L_j)$ in (6), (7), and (8).

Correctly detecting over- and underflow requires some extra tests. For example, it is necessary to avoid over- and underflow in the computation of $b^{(1)}$. The implementation discussed in §4 does this by testing n and k in (3) to screen out obvious over- and underflows and checking (and, if necessary, temporarily modifying) the exponent of the intermediate result before the final multiplication or division; in borderline cases, it sets $b^{(1)}$ to the smallest or largest positive floating-point number, as appropriate. It similarly avoids over- and underflow in computing $b^{(j+1)}$ from $b^{(j)}$, detecting over- or underflow if $b^{(j)}$ is the largest or smallest positive floating-point number and $b^{(j+1)}$ would be larger or smaller, respectively, than $b^{(j)}$.

3. Binary-to-Decimal Conversion

Binary-to-decimal conversion is simpler than decimal-to-binary conversion in the sense that over- and underflow are not issues. A straightforward approach that only involves arithmetic on integers (sometimes very large integers) is to maintain an invariant described by Steele and White [12]. Let b denote the floating-point number (1) to be converted. First we determine integers $b^{(0)} \geq 1$, $S \geq 1$ and k such that

$$b = (b^{(0)}/S) \times 10^k \tag{9a}$$

and

$$1 \leq b^{(0)}/S < 10. \tag{9b}$$

Then we compute digits d_j and residuals $b^{(j)}$ to maintain

$$b = (b^{(j)}/S) \times 10^{k-j} + \sum_{i=0}^{j-1} d_i \times 10^{k-i} \tag{10}$$

by setting

$$d_j := \lfloor b^{(j)}/S \rfloor \tag{11}$$

and

$$b^{(j+1)} := 10 \times (b^{(j)} - d_j \times S) = 10 \times (b^{(j)} \bmod S), \tag{12}$$

where $\lfloor x \rfloor$ denotes the greatest integer $\leq x$.

Steele and White [12] suggest an iterative procedure that uses integer arithmetic to compute the k for which (9) holds, but it is generally more efficient to use floating-point arithmetic to compute $\hat{k}$ with $k \leq \hat{k} \leq k + 1$ and, if necessary, to then use integer arithmetic to compute k from $\hat{k}$. For the floating-point arithmetics considered here, it is straightforward to compute an integer ℓ and a floating-point number x such that $1 \leq x < 2$ and $b = x \times 2^{\ell}$. Now

$$k = \lfloor \log_{10}(b) \rfloor = \lfloor \ell \cdot \log_{10}(2) + \log_{10}(x) \rfloor,$$

$$\log_{10}(x) \leq \log_{10}(1.5) + [x - 1.5]/(1.5 \cdot \log(10)),$$

and, for the arithmetics considered here, $|\ell| \leq 1077$, so

$$k \le \hat{k} := \lfloor fl(\ell \times 0.301029995663981 + (x - 1.5) \times 0.289529654602168 + 0.1760912590558) \rfloor,$$

where we have approximated the constant term $\log_{10}(1.5) = 0.176091259055681242...$ by 0.1760912590558 to ensure $\hat{k} \ge k$; it is easy to see from Taylor's theorem that $\hat{k} \le k + 1$.

Using high-precision integer arithmetic in (11) and (12), we can compute an arbitrarily close decimal approximation d to b. Often an approximation of only a few significant decimal places or a few places past the decimal point is desired. The simplest approach is to generate the desired number of digits and then to round the final digit d_{n-1} appropriately. Of course, this rounding sometimes involves propagating carries, i.e., changing d_i for some values of $i < n - 1$. An alternative advocated by Steele and White [12] is to use a stopping test that assures that we compute each d_i correctly. This is convenient for the co-routine structure described in [12], and sometimes it saves work, but other times it costs more work, because it requires computing with larger integers.

Suppose we have chosen n, and we wish to compute the nearest decimal number of the form (3) to b (with ties broken appropriately). From (10), if

$$b^{(j)} \bmod S < 5 \times S \times 10^{j-n}, \tag{13}$$

then (11) delivers the desired d_j and $d_i = 0$ for $j < i < n$. Similarly, if

$$b^{(j)} \bmod S > S \times (1 - 5 \times 10^{j-n}), \tag{14}$$

then d_j from (11) must be increased by one and again $d_i = 0$ for $j < i < n$. It can happen that $j = 0$ and (11) gives $d_0 = 9$, in which case we must increment k and set $d_0 := 1$. The computation of k in [12] precludes this special case, but it is more efficient to compute and correct k as just described. Note that by scaling $b^{(0)}$ and S by a nonnegative integral power of 10, we can arrange that tests (13) and (14) only involve integer arithmetic.

If the left- and right-hand sides of (13) or (14) are equal, then it is necessary to use the appropriate tie-breaking rule to decide whether and how to stop generating digits.

As argued in [12], there are times when it is desirable to compute the shortest decimal string that correctly rounds to a given floating-point number b. For example, this is the ''right'' way to represent numbers as set elements in the AMPL modeling language [8], which was a prime motivation for the present work. For d to correctly round to b, it suffices that

$$\tfrac{1}{2}(\underline{b} + b) < d < \tfrac{1}{2}(b + \overline{b}), \tag{15}$$

where, as before, $\underline{b}$ and $\overline{b}$ are the floating-point numbers adjacent to b, and [12] tells how to compute the shortest d that satisfies (15). This involves stopping tests similar to (13) and (14). By taking account of whether d will to be converted to binary by biased or unbiased rounding, we can allow one of the strict inequalities in (15) to be an equality, which can lead to a shorter d. For example, with binary double-precision IEEE arithmetic, 10^{23} rounds to a binary floating-point number which (15) would render as

$9.999999999999999 \times 10^{22}$, but taking the unbiased rounding of IEEE arithmetic into account, we can render this number as 10^{23}. The implementation described in §4 will honor either (15) or a relaxed version that allows equalities in (15) when the appropriate rounding rule will yield b.

In general it is necessary to use integer arithmetic of high precision in (11) and (12), but often it is possible — and faster — to get by with floating-point arithmetic. In particular, if b is a floating-point integer with

(16) $$\lfloor \log_{10}(b) \rfloor < \log_{10}(\beta^p) - 1,$$

then $S := 10^{\lfloor \log_{10}(b) \rfloor}$, $i \cdot S$, $1 \le i \le 9$, and all the $b^{(j)}$ are floating-point integers, and there is no roundoff error when (11) and (12) are carried out straightforwardly in floating-point arithmetic.

Another case in which floating-point arithmetic often suffices is that where the number n of significant decimal digits required is sufficiently small, say

(17) $$n < \log_{10}(\beta^{p-1}) - 1$$

(e.g., $n \le 14$ for binary double-precision IEEE arithmetic). In that case, it is often worthwhile to compute

(18) $$\tilde{b}^{(0)} := fl(b \cdot 10^{-k}),$$

and, for $j = 0, 1, \cdots, n-1$,

(19) $$\tilde{d}_j := \lfloor \tilde{b}^{(j)} \rfloor$$

and

(20) $$\tilde{b}^{(j+1)} = fl(10 \times [\tilde{b}^{(j)} - \tilde{d}_j]).$$

If (18) involves μ floating-point operations, then

(21a) $$b = (\tilde{b}^{(0)} + \eta_0) \times 10^k$$

with

(21b) $$|\eta_0| < [(1 - \beta^{1-p})^{-\mu} - 1] \tilde{b}^{(0)}.$$

(In fact, for rounded arithmetic, $|\eta_0| < (1 - \tfrac{1}{2}\beta^{1-p})^{-\mu} - 1$.) Calculation (20) is tantamount to a fixed-point computation; the subtraction is always exact, and the multiplication, over all iterations of (20), can lose a cumulative total of at most 3 bits. Indeed, in VAX or binary IEEE arithmetic ($\beta = 2$),

(22a) $$\tilde{b}^{(0)} = (\tilde{b}^{(j)} + \tilde{\eta}_j) \times 10^{-j} + \sum_{i=0}^{j-1} \tilde{d}_i \times 10^{-i},$$

with

(22b) $$|\tilde{\eta}_j| < 7 \times \beta^{1-p} \times 10^j$$

(and $\tilde{\eta}_0 = 0$); in IBM arithmetic ($\beta = 16$), (22a) holds with $\tilde{\eta}_j \equiv 0$. (The right-hand side of (22b) comes from bounding $\tilde{\eta}_j$ by the error that would be introduced by ignoring the three least significant bits of $b^{(0)}$ — with these bits set to 0, (20) would introduce no further rounding errors.) Combining (21) and (22), we have

$$b = (\tilde{b}^{(j)} + \eta_j)\times 10^{k-j} + \sum_{i=0}^{j-1} \tilde{d}_i \times 10^{k-i},$$

with

$$|\eta_j| = |10^j \times \eta_0 + \tilde{\eta}_j| < \left[[(1 - \beta^{1-p})^{-\mu} - 1]\tilde{b}^{(0)} + 7\times\beta^{1-p} \right] \times 10^j;$$

with (18) carried out as in §4, $(1 - \beta^{1-p})^{-\mu} - 1 < (\mu + 1)\times\beta^{1-p}$ and

(23) $$|\eta_j| < \bar{\eta}_j := fl\left[([\mu + 1]\tilde{b}^{(0)} + 7)\times\beta^{1-p}\times 10^j \right].$$

If

(24a) $$\tilde{b}^{(n)} < 5 - \bar{\eta}_n,$$

then (3) with $d_i = \tilde{d}_i$ is the desired decimal approximation to *b*; otherwise, if

(24b) $$\tilde{b}^{(n)} > 5 + \bar{\eta}_n,$$

then we obtain (3) by incrementing $\tilde{d}_{n-1}$ and propagating carries if necessary; otherwise (18–20) and (23) do not conclusively yield the desired rounded decimal approximation *d*, and we must resort to using high-precision integer arithmetic in (11) and (12).

As in [12], we can use an alternative to tests (24) that involves more computation per digit, but that may save time if some of the trailing digits d_i in (3) are zero. (Note that with (24) there is no need to compute $\bar{\eta}_j$ for $j < n$.) The alternative works as follows. We compute

(25a) $$\hat{\eta}_j := fl\left[[5\times 10^{-n} - ((\mu + 1)\tilde{b}^{(0)} + 7)\times\beta^{1-p}]\times 10^j \right]$$

and test whether

(25b) $$\tilde{b}^{(j)} - \tilde{d}_j < \hat{\eta}_j,$$

in which case $d_i = \tilde{d}_i$ for $i \le j$ and $d_i = 0$ for $j < i < n$. If (25b) is not satisfied, we test whether

(25c) $$fl(1 - [\tilde{b}^{(j)} - \tilde{d}_j]) < \hat{\eta}_j,$$

in which case $d_j = \tilde{d}_j + 1$, $d_i = \tilde{d}_i$ for $i < j$, and again $d_i = 0$ for $j < i < n$. Note that the quantity $\tilde{b}^{(j)} - \tilde{d}_j$ that appears in (25) is part of (20), and that (25) with $j = n - 1$ reduces to (24), since (24b) is equivalent to $10 - \tilde{b}^{(n)} < 5 - \bar{\eta}_n$. Thus if we reach $j = n - 1$ and neither (25b) nor (25c) holds, then again we must resort to using high-precision integer arithmetic in (11) and (12).

On average, when (17) holds, i.e., *n* is not too large, it is probably most efficient first to try (18), (19), and (20). If this fails or if *n* is too large (which includes the case where we wish (15) to hold), then it is reasonable to check whether b is a floating-point integer and (16) holds, which means we can use floating-point arithmetic in (11) and (12). Otherwise, resorting to high-precision integer arithmetic may be unavoidable. The test (17) suggested above for n not being too large comes from requiring (23) to yield $\bar{\eta}_n < 1$ with $[\mu + 1]\tilde{b}^{(0)} + 7$ approximated by 10.

4. Computational Experience

We have written functions, `strtod` and `dtoa`, that do the calculations described in §2 and §3, respectively; `strtod` is designed to conform to the ANSI C standard [2] (in "C" locale); `dtoa` has modes that correspond to fixed-point formatting (as in Fortran's F*w.d* or `printf`'s %*w.d*f format specifiers), to floating-point formatting (as in E*w.d* or %*w.d*e), and to (15), i.e., to computing the shortest decimal string that rounds to a given binary floating-point number b. Thus `dtoa` could be used inside the ANSI C routines `fprintf`, `printf`, and `sprintf`, but it can also compute shortest decimal strings as advocated by Steele and White [12]. The source code is compilable by an ANSI C or C++ [6, 13] compiler, and, by defining appropriate preprocessor variables, one can compile the functions so they are suitable for use with binary IEEE, conventional IBM-mainframe, or VAX double-precision arithmetic. Source for the functions is available from *netlib* [5]; for details, send `netlib@research.att.com` the electronic-mail message

```
send index from fp
```

For decimal-to-binary conversions, we obtain $b^{(1)}$ so that (5) holds with $c = 6.01$ by using a mixture of integer and floating-point arithmetic to compute the integer

$$fl\left[\sum_{i=0}^{\min\{17,\, n-1\}} d_i \times 10^{\min\{17,\, n-1\} - i}\right] \tag{26}$$

with at most one rounding error, then multiplying or dividing (26) by the appropriate power of 10, expressed as the product of factors from the set

$$\{10^i : 0 \le i \le 15\} \cup \{10^{(2^i)} : 4 \le i \le 8\}.$$

(For IEEE arithmetic, this involves at most 6 possibly inexact floating-point operations; for IBM arithmetic, at most 4, and for VAX, at most 3.)

The choices of M in (8) and of S and $b^{(0)}$ in (9) deserve brief discussion. Since β is a power of 2 for the arithmetics considered here, the computations involving M, S, and $b^{(0)}$ entail multiplications by factors of the form 2^κ and 5^λ for nonnegative integers κ and λ. By keeping separate track of κ and λ, `strtod` and `dtoa` often can arrange to use smaller values of M, S, and $b^{(0)}$ than the most straightforward choices $M := \max\{1,\ \beta^{p - e^{(j)} - 1}\} \times \max\{10^{n-k-1},\ 1\}$ or $S := \max\{1,\ \beta^{p-e+1}\} \times \max\{1,\ 10^k\}$ would give.

Below are tables showing average microseconds for 10000 or 20000 repetitions of various conversions on three machines:

• *pyxis* is an SGI 4D/240S with 25MHz MIPS R3000 cpu chips, running IRIX System V Release 3.3.1, with binary IEEE arithmetic. Compilation is by the standard vendor-supplied C compiler (`cc -O`).

• *odin* is an Amdahl 5890, running UNIX® UTS System V Release 2.6b, with IBM-mainframe floating-point arithmetic. On *odin*, when a single possibly inexact floating-point operation suffices, `strtod` computes the relevant rounded product or quotient using the assembly-coded routines whose source is shown in Appendix A. The ANSI C routines were converted to old-style C by `cfront` version 2.1 (part of the AT&T C++ translator), then compiled by the vendor-supplied C compiler (`cc -O`).

• *pipe* is a VAX 8550 running a 10th Edition UNIX system. Compilation was by `lcc`, an experimental C compiler by Chris Fraser and Dave Hanson — see [9].

Tables 1 and 2 show approximate times for our `strtod`, based on §2, to do the indicated decimal-to-binary conversions, and they show corresponding times for the `atof` routine from the standard C library on the machine in question. The conversions in Table 1 fall into the ‘‘typical’’ cases, in which `strtod` gets by with floating-point arithmetic; in these cases, it takes roughly the same time as `atof` on *odin* and runs faster than `atof` on *pyxis* and *pipe*. (On the VAX, the ‘‘typical’’ cases include all $d \geq 1$ with $n \leq 16$ significant figures.) The conversions in Table 2 have so large or small an exponent k or so many digits that `strtod` must resort to high-precision integer arithmetic, which makes it take considerably longer than `atof`. It is easy to find examples on all three machines where `strtod` returns the correctly rounded value and `atof` is less accurate. (On *pyxis*, such examples seem to need at least 18 decimal digits.)

	pyxis IEEE arith.		*odin* IBM arith.		*pipe* VAX arith.	
Input	`strtod`	`atof`	`strtod`	`atof`	`strtod`	`atof`
1.23	9	18	7	7	33	82
1.23e+20	11	15	8	8	38	70
1.23e–20	12	19	8	8	43	305
1.23456789	15	27	12	12	57	150
1.23456589e+20	16	24	12	13	63	85
1.23e+30	11	20	7	10	42	102

Table 1. Decimal-to-binary microseconds, ‘‘typical’’ cases.

	pyxis		*odin*		*pipe*	
	IEEE arith.		IBM arith.		VAX arith.	
Input	`strtod`	`atof`	`strtod`	`atof`	`strtod`	`atof`
1.23e–30	101	20	80	10	380	413
1.23456789e–20	130	28	98	13	458	370
1.23456789e–30	134	28	95	15	552	493
1.234567890123456789	156	39	110	22	640	290

Table 2. Decimal-to-binary microseconds, hard cases.

Tables 3–8 show approximate times for `dtoa`, based on §3, to do the indicated binary-to-decimal conversions, and they show corresponding times for the `ecvt` and `sprintf` routines from the standard C library on the machine in question. The numbers to be converted are the binary versions of the numbers in the *Input* column, as computed by `strtod`. Columns 2 and 3 give times for `dtoa` computing $n = 6$ significant figures. The first six input numbers in Tables 3, 5, and 7 are such that `dtoa` gets by with floating-point arithmetic; the seventh allows use of floating-point arithmetic in (11) and (12). The input numbers in Tables 4, 6, and 8 are chosen to force computation with high-precision integers. The computations in column 2 use a mode that may need to propagate carries when rounding to n significant figures, whereas those in column 3 use the approach advocated by Steele and White [12] of avoiding such carry propagations at the cost of extra computation. Column 4 shows times for computing the shortest decimal string that correctly rounds to the input binary floating-point number b. This generally requires computing with large integers, unless b is a floating-point integer that is not too large, such as the last input number in Tables 3, 5, and 7. Column 5 shows the time taken for the C library routine `ecvt` to compute 6 significant figures for the given input number, and column 6 shows the time taken for `sprintf("%g")` to do this computation. The `dtoa` times are generally longer than those for `ecvt` but shorter than those for `sprintf` (which obviously has extra overhead); but `dtoa` is more accurate than `ecvt` or `sprintf` on all three machines.

Input	`dtoa`, $n = 6$ carries possible	no carries	`dtoa` n from (15)	`ecvt`	`sprintf`
1.23	31	29	109	21	77
1.23e+20	33	31	137	25	71
1.23e–20	33	31	195	26	77
1.23456789	30	37	261	17	78
1.23456589e+20	32	38	285	22	85
1.23456789e–20	32	39	415	22	76
1234565	47	53	36	26	68

Table 3. Binary-to-decimal microseconds on *pyxis* (binary IEEE arithmetic), ‘‘typical’’ cases for $n = 6$.

Input	`dtoa`, $n = 6$ carries possible	no carries	`dtoa` n from (15)	`ecvt`	`sprintf`
1.234565	131	268	210	17	76
1.234565e+20	157	285	237	22	74
1.234565e–20	208	341	335	23	80

Table 4. Binary-to-decimal microseconds on *pyxis* (binary IEEE arithmetic), hard cases for $n = 6$.

Input	`dtoa`, $n = 6$ carries possible	no carries	`dtoa` n from (15)	`ecvt`	`sprintf`
1.23	23	20	87	10	39
1.23e+20	23	22	108	13	46
1.23e–20	23	22	136	13	41
1.23456789	20	25	203	9	38
1.23456589e+20	23	25	222	12	41
1.23456789e–20	22	25	298	13	42
1234565	32	33	23	14	37

Table 5. Binary-to-decimal microseconds on *odin* (IBM arithmetic), ‘‘typical’’ cases for $n = 6$.

Input	dtoa, $n = 6$ carries possible	no carries	dtoa n from (15)	ecvt	sprintf
1.234565	88	200	163	9	38
1.234565e+20	108	212	184	12	41
1.234565e–20	137	242	246	13	41

Table 6. Binary-to-decimal microseconds on *odin* (IBM arithmetic), hard cases for $n = 6$.

Input	dtoa, $n = 6$ carries possible	no carries	dtoa n from (15)	ecvt	sprintf
1.23	82	88	428	78	144
1.23e+20	105	98	518	294	168
1.23e–20	100	95	716	107	168
1.23456789	92	108	1024	80	152
1.23456589e+20	98	120	1109	291	174
1.23456789e–20	93	115	1565	112	181
1234565	162	177	125	108	149

Table 7. Binary-to-decimal microseconds on *pipe* (VAX arithmetic), ‘‘typical’’ cases for $n = 6$.

Input	dtoa, $n = 6$ carries possible	no carries	dtoa n from (15)	ecvt	sprintf
1.234565	450	998	804	83	154
1.234565e+20	565	1092	907	288	173
1.234565e–20	737	1223	1273	111	176

Table 8. Binary-to-decimal microseconds on *pipe* (VAX arithmetic), hard cases for $n = 6$.

5. Concluding Remarks

We have shown that it is possible to do binary-to-decimal and decimal-to-binary conversions in ways that always yield the correctly rounded results, with little time penalty in common cases.

In hard cases, we must resort to high-precision integer arithmetic. The conversion functions (`strtod` and `dtoa`) described in §4 have machine-independent routines, written in C, that carry out this arithmetic. Assembly-coded versions of these routines might run significantly faster on some machines.

Under IEEE arithmetic, `strtod` and `dtoa` could be modified to reset, then test the ''inexact'' flag, and they could then avoid high-precision integer computations in some further cases. The extent to which this would be worthwhile obviously depends on the cost of resetting and testing the ''inexact'' flag and on the distribution of numbers presented for conversion.

The Numeric C Extensions Group is drafting a ''standard'' in which correct or nearly correct binary-to-decimal and decimal-to-binary conversions are required. Some people have expressed concern about the cost of such a requirement; this note should serve to demonstrate that the cost can be modest in common cases.

Of course, there are many situations where precise conversions are not needed and where trading speed for accuracy is desirable. For these situations, it would be helpful to have a library of alternate conversion routines that make a reasonable such trade. But the principle of least surprise suggests that correctly rounded conversions should be the default.

Acknowledgments

Howard Trickey [private communication] has written a program for testing conversion routines with numbers of the form that Schryer [11] has found useful in detecting floating-point arithmetic bugs. I thank Howard for his help in testing `strtod` and `dtoa`. I also thank Norm Schryer and Margaret Wright for helpful comments on the manuscript.

References

[1] *IEEE Standard for Binary Floating-Point Arithmetic,* Institute of Electrical and Electronics Engineers, New York, NY, 1985. ANSI/IEEE Std 754-1985.

[2] *American National Standard for Information Systems — Programming Language — C,* American National Standards Institute, New York, NY, 1990. ANSI X3.159-1989.

[3] W. D. Clinger, ''How to Read Floating Point Numbers Accurately,'' pp. 92–101 in *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation. White Plains, NY, June 20-22, 1990.*

[4] J. T. Coonen, ''Contributions to a Proposed Standard for Binary Floating-Point Arithmetic,'' Ph.D. Dissertation (1984), Univ. of California, Berkeley.

[5] J. J. Dongarra and E. Grosse, ''Distribution of Mathematical Software by Electronic Mail,'' *Communications of the ACM* **30** #5 (May 1987), pp. 403–407.

[6] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual,* Addison-Wesley, 1990.

[7] G. Forsythe and C. B. Moler, *Computer Solution of Linear Algebraic Systems,* Prentice-Hall, 1967.

[8] R. Fourer, D. M. Gay, and B. W. Kernighan, ‘‘A Modeling Language for Mathematical Programming,’’ *Management Science* **36** #5 (1990), pp. 519–554.

[9] C. W. Fraser, ‘‘A Language for Writing Code Generators,’’ *SIGPLAN Notices* **24** #7 (1989), pp. 238–245.

[10] D. W. Matula, ‘‘A Formalization of Floating-Point Numeric Base Conversion,’’ *IEEE Trans. Computers* **C-19** #8 (1970), pp. 681–692.

[11] N. L. Schryer, ‘‘A Test of a Computer’s Floating-point Arithmetic Unit,’’ in *Sources and Development of Mathematical Software*, ed. W. Cowell, Prentice-Hall (1981).

[12] G. L. Steele and J. L. White, ‘‘How to Print Floating-Point Numbers Accurately,’’ pp. 112–126 in *Proceedings of the ACM SIGPLAN’90 Conference on Programming Language Design and Implementation. White Plains, NY, June 20-22, 1990.*

[13] B. Stroustrup, *The C++ Programming Language,* Addison-Wesley, 1986.

Appendix A: Rounded Products and Quotients with IBM Extended Precision

The following assembly code, suitable for use on *odin*, defines functions `rnd_prod` and `rnd_quot` that correspond to the ANSI C headers

```
double rnd_prod(double a, double b); /* rounded a*b */
double rnd_quot(double a, double b); /* rounded a/b */
```

These functions compute the rounded product or quotient required in `strtod` when the result can be expressed as a single possibly inexact product or quotient.

```
        entry rnd_prod
rnd_prod:
        using  rnd_prod,15
        ld     0,0+64(13)
        mxd    0,8+64(13)
        lrdr   0,0
        b      2(,14)
        drop
        entry  rnd_quot
rnd_quot:
        using  rnd_quot,15
        ld     0,0+64(13)
        ldr    2,0
        ld     4,8+64(13)
        ddr    2,4
        std    2,32(13)
        mxdr   4,2
        sdr    2,2
        sxr    0,4
        dd     0,8+64(13)
        sdr    2,2
        ld     4,32(13)
        sdr    6,6
        axr    0,4
        lrdr   0,0
        b      2(,14)
```