# High Precision Division and Square Root

Alan H. Karp
Peter Markstein
HP Labs Report 93-93-42 (R.1)
June 1993
Revised October 1994

Multiprecision floating point, quad precision, square root, division; Computing Reviews Categories G.1.0, G.4

We present division and square root algorithms for calculations with more bits than are handled by the floating point hardware. These algorithms avoid the need to multiply two high precision numbers, speeding up the last iteration by as much as a factor of ten. We also show how to produce the floating point number closest to the exact result with relatively few additional operations.

# 1 Introduction

Floating point division and square root take considerably longer to compute than addition and multiplication. The latter two are computed directly while the former are usually computed with an iterative algorithm. The most common approach is to use a division-free Newton-Raphson iteration to get an approximation to the reciprocal of the denominator (division) or the reciprocal square root, and then multiply by the numerator (division) or input argument (square root). Done carefully, this approach can return a floating point number within 1 or 2 ULPs (Units in the Last Place) of the exact result. With more work, sometimes a lot more, the error can be reduced to half an ULP or less.

The usual approach works quite well when the results are computed to no more precision than the hardware addition and multiplication. Typically, division and square root take 10 to 20 machine cycles on a processor that does a multiplication in the same precision in 2 or 3 cycles. The situation is not so good for higher precision results because the cost of the final multiplication is much higher. The key feature of our algorithm is that it avoids this expensive multiplication.

Section 2 discusses some methods used to do division and square root. Section 3 describes the floating point arithmetic and the assumptions we are making about the floating point hardware. Section 4 describes our new algorithms. In Section 5 we present an analysis of the accuracy of the algorithm. Next, we show how to get the correctly rounded results with relatively few additional operations, on average, in Section 6. The procedures we use for testing are described in Section 7. Programs written in the programming language of the Unix desk top calculator utility `bc`[4] that implement these algorithms are included in an appendix.

# 2 Algorithmic Choices

There are a number of ways to perform division and square root. In this section we will discuss these methods with emphasis on results that exceed the precision of the floating point hardware.

One approach is to do school-boy long division, a scheme identical to successive subtraction for base 2 arithmetic[8]. The procedure is straightforward. To compute $B/A$, we initialize a remainder to $R = 0$. Then, for each bit we do the following steps:

1. Shift $R$ concatenated with $B$ left one bit so that the high order bit of $B$ becomes the low order bit of $R$.
2. Subtract $A$ from from $R$.
3. If the result is negative, set the low order bit of $B$ to zero; otherwise set it to 1.

4. If the result of the subtraction is negative, add $A$ to $R$.

At the end of N steps, the N-bit quotient is in $B$, and the remainder is in $R$.

A similar scheme can be used for square root[7]. We start with the input value $A$ and a first guess $Y = 0$. For each bit we compute the residual, $R = A - Y^2$. Inspection of $R$ at the $k$'th step is used to select a value for the $k$'th bit of $Y$. The selection rules give some flexibility in tuning the algorithm.

There are improvements to these approaches. For example, non-restoring division avoids adding $B$ to $R$. The SRT algorithm[8] is an optimizaton based on a carry-sum intermediate representation. Higher radix methods compute several bits per step. Unfortunately, these improvements can't overcome the drawback that we only get a fixed number of bits per step, a particularly serious shortcoming for high precision computations.

Polynomial approximations can also be used[7]. Chebyshev polynomials are most often used because they form an approximation with the smallest maximum error giving a guaranteed bound on the error. However, each term in the series adds about the same number of bits to the precision of the result, so this approach is impractical for high precision results.

Another approach is one of the CORDIC methods which were originally derived for trigonometric functions but can be applied to square root[7]. These methods treat the input as a vector in a particular coordinate system. The result is then computed by adding and shifting some tabulated numbers. However, the tables would have to be inconveniently large for high precision results.

There are two methods in common use that converge quadratically instead of linearly as do these other methods, Goldschmidt's method and Newton iteration. Quadratic convergence is particularly important for high precision calculations because the number of steps is proportional to the logarithm of the number of digits.

Goldschmidt's algorithm[8] is based on a dual iteration. To compute $B/A$ we first find an approximation to $1/A = A'$. We then initialize $x = BA'$ and $y = AA'$. Next, we iterate until convergence.

1. $r = 2 - y$.
2. $y = ry$.
3. $x = rx$.

If $AA'$ is sufficiently close to 1, the iteration converges quadratically. A similar algorithm can be derived for square root. Round-off errors will not be damped out because Goldschmidt's algorithm is not self-correcting. In order to get accurate results, we will have to carry extra

digits through the calculation. While carrying an extra word of precision is not a problem in a multiprecision calculation, the extra digits required will slow down a calculation that could have been done with quad precision hardware.

Newton's method for computing $B/A$ is to approximate $1/A$, apply several iterations of the Newton-Raphson method, and multiply the result by $B$. The iteration for the reciprocal of $A$ is

$$x_{n+1} = x_n + x_n(1 - Ax_n). \tag{1}$$

Newton's method for computing $\sqrt{A}$ is to approximate $1/\sqrt{A}$, apply several iterations of the Newton-Raphson method, and multiply the result by $A$. The iteration for the reciprocal square root of $A$ is

$$x_{n+1} = x_n + \frac{x_n}{2}(1 - Ax_n^2). \tag{2}$$

Newton's method is quadratically convergent and self-correcting. Thus, round-off errors made in early iterations damp out. In particular, if we know that the first guess is accurate to $N$ bits, the result of iteration $k$ will accurate to almost $2^k N$ bits.

# 3 Floating Point Arithmetic

Our modifications to the standard Newton method for division and square root makes some assumptions about the floating point arithmetic. In this section we introduce some terminology, describe the hardware requirements to implement our algorithm, and show how to use software to overcome any deficiencies.

We assume that we have a floating point arithmetic that allows us to write any representable number

$$x = \beta^e \sum_{k=0}^{N} \beta^{-k} f_k, \tag{3}$$

where $\beta$ is the number base, the integer $e$ is the exponent, and the integers $f_k$ are in the interval $[0, \beta)$.

As will be seen in Section 4, we compute a $2N$-bit approximation to our function from an $N$-bit approximation. We will refer to these as full precision and half precision numbers, respectively. In the special case where the operations on half precision numbers are done in hardware, we will refer to 2N-bit numbers as quad precision and N-bit numbers as hardware precision. The term high precision will be used to denote both quad and multiprecision operations.

There are several ways to implement floating point arithmetic. We can round the result of every operation to the floating point number nearest the exact result. We can truncate the result by throwing away any digits beyond the number of digits in the input format. When

```
void prod(a,b,c)   /* All digits of c = a*b */
  double a, b, c[]
{
  long double t;
  t = (long double) a * (long double) b;
  c[0] = t;
  c[1] = t - c[0];
}
```

Figure 1: One way to get all the digits of the product of two double precision numbers on a machine that supports quad precision variables. The result is returned as two double precision numbers.

we truncate, we can use a guard digit to improve the average accuracy. It is possible to do a multiply-add as a multiplication followed by an addition, which has two roundings, or as a fused operation with only one rounding. The fused multiply-add operation can be done with full precision or with only some of the digits in the intermediate result. Our algorithms are insensitive to what choices are made. Surprisingly enough, we are able to get accurate results even if we truncate all intermediate operations.

Our algorithms assume that we can get all the digits in the product of two hardware precision numbers as well as the leading quad precision part of the sum. Some machines have hardware instructions that return the quad precision result of multiplying two double precision numbers[3], but many do not. Figure 1 shows one way to compute the result on a machine that supports quad precision variables. This approach is inefficient since it typically takes 4 to 10 times longer to multiply two quad numbers than two doubles. A hardware operation that returns all the bits of a double times double to quad takes no more time than a hardware multiplication.

If the system doesn't support quad precision, we must use a single-single representation as our hardware format. In this format our numbers are stored in two single precision numbers. Figure 2 shows one way to multiply two single-single precision numbers and return both the high and low order parts of the product. We have assumed that the double precision format holds at least two digits more than in the product of two single precision numbers, a condition met by IEEE floating point[1].

Addition is different. The sum of two floating point numbers can have a very large number of digits in the result if the exponents differ by a lot. Figure 3 shows one way to add two numbers in the single-single format and return the leading quad precision part of the result in four single precision numbers. Each such addition takes 12 floating point operations.

There are significant differences in the implementations of multiprecision and quad precision

```
void prod(a,b,c)   /* All digits of c = a*b */
  float a[], b[], c[]
{
  double t, u, v, w, x, y;
  u = (double) a[0]*(double) b[0];
  v = (double) a[0]*(double) b[1];
  w = (double) a[1]*(double) b[0];
  x = (double) a[1]*(double) b[1];
  y = v + w + (float) x;
  t = u + y;
  c[0] = t;
  t -= c[0];
  c[1] = t;
  t = t - c[1] + x - (float) x;
  c[2] = t;
  c[3] = t - c[2];
}
```

Figure 2: One way to get all the digits of the product of two double precision numbers stored in single-single format. The result is returned as four single precision numbers.

addition and multiplication. Full precision multiplication takes up to 4 times longer than half precision multiplication; quad precision multiplication takes more than 10 times longer than hardware precision multiplication with hardware support and some 100 times longer without. On the other hand, carry propagation is much simpler in quad precision since we are dealing with only two numbers.

Our algorithms use many of the combinations of precisions shown in Table 1. In this table, the subscript indicates the number of digits in the result, $d$ denoting the hardware precision. We can compute $V_d$ in hardware. $U_{2d}$ is available in hardware on some systems. If it is not, we will have to use software. The cost estimates are for double precision operations that return quad results done in hardware ($Q_h$), these operations done in software ($Q_s$), multiprecision numbers of up to a few hundred digits ($C$ since we use the conventional approach), and multiprecision numbers of more than a few hundred digits ($F$ since we use an FFT method).

Henceforth we will assume that our hardware returns all the bits we need. If it doesn't, we will have to implement functions such as the ones shown for all unsupported operations.

```
void sum(a,b,c)   /* Leading 2N digits of c = a+b */
  float a[], b[], c[]
{
  double cl, ch, max, min, t;
  max = (fabs(a[0])>fabs(b[0])?a[0]:b[0]);
  min = (fabs(a[0])<=fabs(b[0])?a[0]:b[0]);
  ch = a[0] + b[0];
  t = min - (ch - max);
  cl = a[1] + b[1] + t;
  t = ch + cl;
  cl = cl - (t - ch);
  c[0] = t;
  c[1] = t - c[0];
  c[2] = cl;
  c[3] = cl - c[2];
}
```

Figure 3: One way to get the leading $2N$ digits of the sum of two numbers stored in the single-single format. We store the high and low order parts of the inputs in separate single precision words, add the high and low order parts, and propagate any carries from the low order part to the high order part of the result. The result is returned as 4 single precision numbers.

## 3.1 Quad

Quad precision is the number format with approximately twice as many digits as the largest format handled in hardware. Some systems have a true quad precision data type; others use a double-double representation. Some do quad arithmetic in hardware; others use software. In almost all cases, quad operations are built up out of operations on hardware precision numbers stored in registers.

We store our quad precision number in two doubles, the double-double representation. Our numbers will be written as $A = A_h + \eta A_l$ where $\eta$ is the precision of our memory format numbers, $\eta = 2^{-53}$ for IEEE double precision. Sample code for the various multiplications we will need are shown in Appendix A. Note that none of the algorithms need all the digits of a full times full, although they could be coded this way.

We do need two additional computations, namely $W_d = C - AB$ and $X_{2d} = C - AB$, where $W_d$ is less than 2 ULPs of $C_h$ and $X_d$ is less than 2 ULPs of $C$. We exploit the special structure of the result by proceeding from high order terms to low order terms, something we can not do without this condition. Our implementation shown in Figure 11 makes use

Table 1: Different multiplications. Entries are listed in order of decreasing cost. $A$ and $B$ are full precision numbers of $2N$ digits while $a$ and $b$ are half precision numbers of $N$ digits. The subscript $kd$ denotes a number consisting of $k$ half precision words.

| Input | Input | Output | Operation | Relative Cost $Q_h$ | $Q_s$ | $C$ | $F$ |
|---|---|---|---|---|---|---|---|
| Full | Full | All bits | $P_{4d} = A * B$ | 11 | 132 | 8 | 4 |
| Full | Full | Full | $R_{2d} = A * B$ | 8 | 84 | 4 | 4 |
| Full | Half | All bits | $Q_{3d} = A * b$ | 7 | 96 | 3 | 2 |
| Full | Half | Full | $S_{2d} = A * b$ | 4 | 48 | 2 | 2 |
| Full | Half | Half | $T_d = A * b$ | 3 | 36 | 2 | 1 |
| Half | Half | Full | $U_{2d} = a * b$ | 1 | 12 | 2 | 1 |
| Half | Half | Half | $V_d = a * b$ | 1 | 1 | 1 | 1 |

of the fact that there is a lot of cancellation of terms. For example, `g[8]` can have at most two nonzero digits.

## 3.2 Multiprecision

Our discussion is based on the public domain package `mpfun`[2]. A multiprecision number is stored in a single precision floating point array. The first word is an integer valued floating point number whose absolute value represents the number of words in the mantissa; the sign of this word is the sign of the multiprecision number. The next word contains an integer valued floating point number representing the exponent of the number base, $\beta$. In contrast to the notation of Equation 3, the decimal point follows the first mantissa word, not the first digit.

Multiplication is based on the fact that virtually every floating point system in existence will return all the digits of the product of two single precision numbers. Converting this double precision result into two single precision numbers gets us ready for the next operation. A similar trick can be used for addition; all intermediate sums are computed in double precision and carries are propagated to single precision numbers.

Since we are interested in the efficiency of our algorithms, we must consider the cost of the basic operations. Adding two $2N$-bit numbers takes about four times as long as adding two $N$-bit numbers. The procedure is straightforward. First, align the numbers by scaling one of them until their exponents are the same. Next, add corresponding elements. Finally, we propagate the carries. The value returned is the leading $2N$ bits of the result.

Multiplication is more complicated. The time to compute the product of an $N$-digit number and an $M$-digit number scales as $NM$ if we use the direct method. Methods based on fast Fourier transforms (FFT) for the product of two $N$-digit numbers take a time proportional

to $NlogNloglogN$[2]. Of course, the coefficient of these scalings are much larger for multiplication than for addition. Hence, algorithms that avoid multiplying full precision numbers together will run faster than those that need such products.

# 4 New Algorithms

We have seen how complicated high precision arithmetic can be. It is clear that our algorithm should avoid such operations whenever possible. The algorithms presented in this Section perform high precision division and square root with no multiplications of numbers of the precision of the result.

First look at the standard Newton method for divison and square root shown in Equations 1 and 2. If we are doing a high precision calculation, we can implement these iterations without doing any operations on two numbers in the longest precision. First of all, in the early iterations, when the approximation is less accurate than the base number format, we use hardware addition and multiplication. At each iteration beyond this accuracy, we double the number of digits carried to match the accuracy of the approximation.[2]

The last iteration needs some care to avoid multiplying two long precision numbers. Look at the square root. First we compute $u = Ax_n$ as a full times a half, then $v = ux_n$ the same way. Since $x_n$ is a very good approximation to $\sqrt{A}$, $1 - v$ will have at most 1 more bit than $x_n$. Hence, we can compute $x_n(1 - v)$ as a half times a half. We don't even need to compute more than half the digits in the product since this quantity is a small correction to the current estimate. Hence, both the last multiplication and the last addition can be done in half precision.

The problem comes in computing the desired result from the last iteration. For division, we must multiply the full $x_{n+1}$ times the full $B$; for square root, we multiply by the full $A$. These operations are expensive, anywhere from 2 to 100 times the time of half precision operations.

There is a simple way to avoid these long multiplications; do them before the last iteration rather than after. Now the last iteration for division becomes

$$y_{n+1} = y_n + x_n(B - Ay_n), \tag{4}$$

where $y_n = Bx_n$, while for square root,

$$y_{n+1} = y_n + \frac{x_n}{2}(A - y_n^2). \tag{5}$$

with $y_n = Ax_n$.

The key to this approach is that, in both cases, these are the Newton iterations for the final result, quotient and square root, respectively. Hence, $y_{n+1}$ is the desired approximate result

accurate to nearly the number of digits in a full precision number. Since the Newton iteration is self-correcting, we don't even need an accurate value for $y_n$. In our implementations we compute $y_n$ as the half precision result of multiplying two half precision numbers.

There is a subtle point in the way the terms have been collected in the final iterations. In both cases, we have brought the multiplicand inside the parentheses. In this way, we compute the residual based on $y_n$, the number we are correcting. If we had made the other choice, we would be correcting $y_n$ with a residual computed from $x_n$. This choice would have forced us to compute $y_n$ as the product of a full and a half precision number in order to get the right answer.

We can now see the savings. For square root, we compute $y_n^2$ as the full precision product of two half precision numbers. After subtracting the result from $A$, we are left with at most one bit more than a half precision number so we can use a half times half to half precision result when we multiply by $x_n$. We do a little bit more work for division. Here we must multiply the full precision number $A$ times the half $y_n$, but the rest of the operations are the same as for square root.

The savings are summarized in Table 2, much of which was supplied by David Bailey. We look at division and square root for three precisions – quad (Q), up to a few hundred digits (C since we use the conventional multiplication algorithm), and more than a few hundred digits (F since we use FFT-based multiplication). Times if hardware returns the quad result of the product or sum of two doubles are in the $Q_h$ column; times without hardware assist are in the $Q_s$ column. In each column, the unit used is the time to compute the half precision product of two half precision numbers. For quad precision, the unit is the hardware multiplication time. Hardware precision addition is assumed to take as long as hardware multiplication, and the time to do addition is ignored for multiple precision calculations. Upper case letters denote full precision numbers; lower case, half precision numbers. If assignment is made to a half precision number, we need calculate only the leading digits of the result. We only count the operations in the last iteration because that is the only place we modify the standard algorithm.

# 5 Analysis

Several of our results depend on properties of quotients, square roots, and Newton Raphson approximation methods. In this section we will sketch proofs of these properties.

**Theorem 1** *In k-digit floating point arithmetic, using a prime radix, a quotient cannot be exact in $k+1$ digits in which the low order result digit is significant.*

**Proof 1** *Suppose that the quotient $c = a/b$ is exactly $c = p^e \sum_{i=0}^{k} c_i p^{-i}$. If $b = p^f \sum_{i=0}^{k-1} b_i p^{-i}$, it must be the case that*

Table 2: Comparison of cost of new versus standard algorithms for three different precisions.

| Division $B/A$ | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Standard Approach<br>$X_{n+1} = x_n + x_n(1 - Ax_n)$<br>$B/A \approx BX_{n+1}$ | | | | | New Approach<br>$y_n = Bx_n$<br>$Y_{n+1} = y_n + x_n(B - Ay_n)$ | | | | |
| Operation | $Q_h$ | $Q_s$ | C | F | Operation | $Q_h$ | $Q_s$ | C | F |
| $T = Ax_n$ | 4 | 48 | 2 | 2 | $y_n = Bx_n$ | 1 | 1 | 1 | 1 |
| $t = 1 - T$ | 1 | 2 | 0 | 0 | $T = Ay_n$ | 4 | 48 | 2 | 2 |
| $t = x_n t$ | 1 | 1 | 1 | 1 | $t = B - T$ | 1 | 2 | 0 | 0 |
| $X_{n+1} = x_n + t$ | 1 | 3 | 0 | 0 | $t = x_n t$ | 1 | 1 | 1 | 1 |
| $T = BX_{n+1}$ | 8 | 96 | 4 | 2 | $Y_{n+1} = y_n + t$ | 1 | 3 | 0 | 0 |
| Total | 15 | 150 | 7 | 5 | Total | 8 | 55 | 4 | 4 |
| Square root $\sqrt{A}$ | | | | | | | | | |
| Standard Approach<br>$X_{n+1} = x_n + \frac{x_n}{2}(1 - Ax_n^2)$<br>$\sqrt{A} \approx AX_{n+1}$ | | | | | New Approach<br>$y_n = Ax_n$<br>$Y_{n+1} = y_n + \frac{x_n}{2}(A - y_n^2)$ | | | | |
| Operation | $Q_h$ | $Q_s$ | C | F | Operation | $Q_h$ | $Q_s$ | C | F |
| $T = x_n^2$ | - | 12 | - | 1 | $y_n = Ax_n$ | 1 | 1 | 1 | 1 |
| $T = AT$ | - | 96 | - | 2 | $T = y_n^2$ | 1 | 12 | 2 | 1 |
| $T = Ax_n$ | 4 | - | 2 | - | $t = A - T$ | 1 | 2 | 0 | 0 |
| $T = Tx_n$ | 4 | - | 2 | - | $t = x_n t/2$ | 2 | 2 | 2 | 2 |
| $t = 1 - T$ | 1 | 2 | 0 | 0 | $Y_{n+1} = y_n + t$ | 1 | 3 | 0 | 0 |
| $t = x_n t/2$ | 2 | 2 | 2 | 2 | | | | | |
| $X_{n+1} = x_n + t$ | 1 | 3 | 0 | 0 | | | | | |
| $T = AX_{n+1}$ | 8 | 96 | 4 | 2 | | | | | |
| Total | 20 | 211 | 10 | 7 | Total | 6 | 20 | 5 | 4 |

$$\begin{aligned} a &= b \times c \\ &= p^{e+f} \sum_{i=0}^{k} c_i p^{-i} \sum_{i=0}^{k-1} b_i p^{-i} \\ &= p^{e+f} (c_0 b_0 + \cdots + c_k b_j p^{-(k+j)}), \end{aligned}$$

*where $j$ is the lowest order non-zero digit in $b$. Since $c_k$ and $b_j$ are both non-zero, $c_k b_j$ is not divisible by the radix, so that this quantity requires at least $k + j + 1$ digits. But $a$ was representable as a $k$-digit number.* •

The same line of reasoning shows that a square root cannot be representable as a $(k+1)$-digit result. (Exercise for the reader: Why does this proof fail for non-prime radices? It does not hold for hex floating point arithmetic, for example).

**Theorem 2** *To round a quotient correctly to k bits, the quotient must be computed correctly to at least $2k + 1$ bits.*

**Proof 2** *A proof of this proposition has been published previously[5]. With certain precautions, $2k$ bits suffice[6].* •

For binary radix, the following empirical evidence suggests this proposition. Consider the quotient $1/(2^k - 1)$. The closest binary fraction to the result, up to $2k$ bits, is $1 + 2^{-k}$ (a $k + 1$ bit number). To get the correctly rounded result (either 1, or $1 + 2^{-k+1}$), we must know whether $1 + 2^{-k}$ is an overestimate or an underestimate, a task which takes almost as much computation as to establish the quotient to $2k + 1$ bits. Only when the quotient is computed to $2k + 1$ bits (or more), is it clear that the closest binary fraction to the quotient is $1 + 2^{-k} + 2^{-2k}$ (up to $3k$ bits), which clearly should be rounded to $1 + 2^{-k+1}$.

**Theorem 3** *To round a square root correctly to k bits, the square root must be computed correctly to at least $2k + 3$ bits.*

**Proof 3** *As before, with certain precautions, $2k$ bits suffice[6].* •

Again, empirical evidence suggested this proposition. Consider the square root of $1 - 2^{-k}$. The closest binary fraction to the result, using $k + 1$ to $2k + 2$ bits, is $1 - 2^{-k-1}$. Just as in the case of division, the decision whether to round up or down depends on whether this approximation is an overestimate or an underestimate, which requires almost as much computation as to establish the quotient to $2k + 3$ bits. Computing further shows that the best approximation is $1 - 2^{-k-1} - 2^{-2k-3}$ (up to $3k + 4$ bits), so that the best $k$-bit approximation is $1 - 2^{-k}$.

Theorems 2 and 3 imply that the trick of looking at a few extra bits in the result to decide on the rounding direction[8] will only work *most* of the time, not *all* of the time as we would like.

**Theorem 4** *Equation 5 always underestimates a square root.*

**Proof 4** *The conventional Newton Raphson square root formula, Equation 2, always overestimates the correct result. However, in Equation 5, we do not divide the residual by $y$, but multiply by $x$, the previous reciprocal square root from which $y$ was computed as $Ax$. Suppose $1 - Ax^2 = e$. Then $Ax^2 = 1 - e$, and*

$$x\sqrt{A} \approx 1 - \frac{e}{2}, \qquad e \ll 1.$$

*If $y = Ax$, an application of Equation 5 gives*

$$\begin{aligned} y' &= y + \tfrac{x}{2}(A - y^2) \\ &= Ax + \tfrac{x}{2}(A - A^2x^2) \\ &= Ax + \tfrac{Ax}{2}(1 - Ax^2) \\ &= Ax(1 + \tfrac{\epsilon}{2}) \\ &= \sqrt{A}(x\sqrt{A})(1 + \tfrac{\epsilon}{2}) \\ &\approx \sqrt{A}(1 - \tfrac{\epsilon}{2})(1 + \tfrac{\epsilon}{2}) \\ &= \sqrt{A}(1 - \tfrac{\epsilon^2}{4}). \quad \bullet \end{aligned}$$

In Section 3, we claim that using truncating arithmetic also will produce acceptable results. For square root, if the reciprocal square root is known to $n$ bits, using $n$ bit arithmetic, how much error can be introduced in the application of Equation 5 if truncated arithmetic is used? First, the computation of $y = Ax_n$ would also be good to $n$ bits of accuracy before truncation. After truncation, an error as great as one unit in the last place may be introduced, so that only $n - 1$ bits of precision remain.

In the absence of additional rounding or truncation errors, we would expect that Equation 5 yields $2n - 2$ bit accuracy. We assume that $y_n^2$ is calculated exactly, so that $2n - 2$ bits of the product are trustworthy. As few as $n - 1$ bits will cancel in the subtraction (in the difficult rounding cases, this is usually the case). Thus, the subtraction yields an $n + 1$ bit difference, which will be truncated to $n$ bits. Since only $n - 1$ bits of the difference was trustworthy, after truncation, only $n - 2$ bits are accurate. The product of this difference by the reciprocal approximation can introduce another bit of error due to truncation. Thus, Equation 5 is expected to yield an approximation good to $2n - 3$ bits.

# 6 Rounding

We have shown how to get the quotient and square root accurate to a few Units in the Last Place (ULPs). This accuracy usually suffices for isolated evaluations, but there are times when more accuracy is needed. For example, if the result is not correct to half an ULP or less, the computed value for the function may not satisfy such algebraic properties as monotonicity. In addition, if the result returned is not the floating point number closest to the exact result, future implementations may not return the same value causing confusion among users.

The need to maintain monotonicity and the identical results for different implementations is important for both quad and high precision arithmetic. In the latter case, simply asking the user to use one more word precision than the application needs is not a major inconvenience. However, if the user takes all the words in the multiprecision result and doesn't round to the needed number of bits, the desired arithmetic properties may be lost. In addition, in the worst case, the rounding can't be done correctly unless twice as many bits as needed are known. In the case of quad precision we have no choice; we can't return more digits than the user asks for so we must do the rounding ourselves.

Quad precision rounding depends on how numbers are stored in registers. Most machines have registers with the same number of digits as the storage format of the numbers; others, such as the Intel x87 floating point co-processors, implement the IEEE standard[1] recommended extended format which keeps extra bits for numbers in the registers. We will consider both these implementations. First, we look at getting the correct value on a machine that keeps more digits in the registers than in the memory format. We will also show how to get the correct result on a machine that keeps no additional digits in the register.

For concreteness, we will look at the problem of computing the quad precision quotient and square root. For simplicity, we ignore error conditions and operations on the characteristics of the floating point numbers since these are the same for both the usual algorithm and ours. We will also assume that the machine has an instruction that returns the quad precision result of arithmetic operations on two double precision numbers. If we do not have this capability, we will have to build up the algorithm using the double precision product of two single precision numbers as shown in Figures 2 and 3.

As shown in Section 5, our algorithm produces a $2N$-digit mantissa with all but the last few digits correct. We also showed that there are some numbers we can't round correctly without computing at least $4N + 3$ digits of the result. The implementations are different if our registers are wider than our words in memory or not so we describe them separately.

## 6.1 Long Registers

Here we will show how to compute the floating point number closest to the exact square root or quotient when the output from our final iteration has more bits than we need return to the user. In a multiprecision calculation we can always carry an additional word of precision but return a result with the same number of digits as the input parameters. For quad results we assume we are running on a machine that does base 2 arithmetic and keeps more bits in the registers than in memory. More specifically, we assume that a double precision number has 53 mantissa bits in memory and 64 in the registers. Our input is assumed to have 113 mantissa bits. Our goal is to return the correctly rounded 113-bit result.

As shown in Section 5 our algorithm produces a mantissa with at least 125 correct bits. We also showed that there are square roots that we can't round correctly unless we know the

result to at least 229 bits. Rather than do an additional, expensive iteration, we use the fact that we have underestimated the correct result. Hence, we know the correctly rounded value is the 128-bit number we have computed truncated to 113 bits or that 113-bit number plus one ULP.

One way to find out which is correct is to compute the residual with the computed value $y_{n+1}$ and with $y_{n+1} + \mu_{113}$, where $\mu_{113}$ is one ULP of a 113-bit number. The smaller residual belongs to the correct result. This approach is expensive, both because we must compute two residuals and because each residual needs the product of two quad precision numbers.

Tuckerman rounding[6] avoids this problem. We need only compute the residual for $y_{n+1} + \mu_{113}/2$. If the sign is positive, the larger number is the desired result; if negative, we want the smaller. As shown in Section 5 the value can not be exactly zero for a machine with a prime number base so we don't have to worry about breaking ties.

The full Tuckerman test for precisions with more digits than the hardware can handle is expensive. For example, for square root, even if we make the simplification that

$$(y_{n+1} + \mu_{113}/2)^2 \geq y_{n+1}(y_{n+1} + \mu_{113}), \tag{6}$$

we must compute a lot of terms. These operations are almost exactly what we must do first to do another iteration, but we need only look at the sign of the result. Finishing the iteration would require a half precision multiplication and an addition.

Computing the residual is expensive, but if we have extra bits in the registers, we can avoid the doing the test most of the time. We know that the first 125 bits of our result are correct and that we have underestimated the correct answer. Hence, if bit 114, the rounding bit, is a 1, we know we must round up. If the rounding bit is a 0 and any of the bits 115 through 125 is a zero, we know that we should return the smaller value. Only if bit 114 is a 0 and bits 115 through 125 are all ones do we need further computation. In other words, there is exactly one pattern of 12 bits that we can't round properly. If the trailing bits are random, we need do extra work for only 1 in 2,048 numbers.

Even in the rare cases where we can't decide which way to round from the 128-bit result, we can often avoid doing the full Tuckerman test. Any time the intermediate result becomes negative or zero we can stop because we know we should return the smaller value, a case which occurs half the time for random bit patterns. We can also stop if the intermediate result is positive and larger in magnitude than a bound on the magnitude of the remaining terms which are all negative.

There are two convenient places to check – after accumulating all terms larger than $\eta^2$, and again after computing all terms larger than $\eta^3$. (Here $\eta = 2^{-64}$, the precision of numbers in the registers. If we are using normalized, IEEE floating point, double precision numbers which have an implicit leading 1, mantissas lie in the interval $[1, 2)$. This means that the coefficient of $\eta^2$ and $\eta^3$ are less than 10. (See Figure 11.) Therefore, we can stop after

computing the first set of terms unless the residual is positive and less than $10\eta$ or after the second set unless the residual is positive and less than $10\eta^2$.

For randomly distributed residuals we need the test only one time out of 2,048 inputs. We can stop after 9 operations, 3 of them multiplications, all but once in 16,384 times. The next test will determine the rounding in all but one case in $2^{64}$ trials. Hence, the average performance for correctly rounded results is almost exactly the same as that for results accurate to one ULP, although in the worst case we must do a large number of additional operations.

The situation is even better for multiprecision calculations. First of all, there are only 4 half precision multiplications in the test; all the rest of the operations are additions. Secondly, it is a simple matter to do all the intermediate calculations with a single extra word. In this case, the only bad situation is when the lowest order word of the result, when normalized to be an extension of the next higher order word, has a leading zero followed by all ones. Since this situation arises only once in $2^{64}$ evaluations for random trailing bits, we almost never need the Tuckerman test. When we do need the test, the early tests catch an even larger percentage of the cases than for quad precision. However, there is no escaping the fact that there are some input values that require us to compute the full Tuckerman test to decide which way to round.

As originally formulated, Tuckerman rounding can be used only for square root, not division. That formulation uses the approximation in Equation 6. We have no such identity for division, but we do have extra bits in our registers so our test is to check the sign of $B - A(y_{n+1} + \mu_{113}/2)$. This version is what appears in Section B.

## 6.2 Short Registers

We can also compute the floating point number closest to the exact result on a machine that does not keep extra bits in the registers. In this case we will assume that our quad precision numbers are stored as two double precision numbers each having 53 mantissa bits in both memory and the registers. Our input is 106 bits long, and we wish to compute the correctly rounded 106-bit result.

We know from Section 5 that the algorithm described in Section 4 has produced a result with at least 102 correct bits. We would have to do two more iterations to get the correct result, a very expensive proposition, but we can't apply standard Tuckerman rounding since there may be four bits in error. Fortunately, we can apply Tuckerman rounding six times at different bit positions at a modest average number of operations.

The procedure is simple. We take the output of the Newton iteration and set the low order 4 bits to zero. Since we have an underestimate of the correct result, we know that the correctly rounded 102-bit result is either the number we have or that number plus $\mu_{102}$, the ULP of

a 102-bit number. If the Tuckerman test tells us to use the larger number, we know that bit 103 must be a one so we add $\mu_{103}$ to set this bit. We now have an underestimate of the correct result but with 103 correct bits. Now we repeat at the 103'rd, 104'th, and 105'th bits. One more application of the test at the 106'th bit does the trick, but now we add $\mu_{106}$ if the test indicates that our result is too small.

We need to be careful at the start. Say that the correctly rounded result ends in the hexadecimal string `8000001`. If we make a 2 ULP underestimate, our working value would be `7FFFFFF`. If we use the procedure just described, we would return `8000000`. Although we have made only a 2 ULP error, the last correct bit is 28 positions from the end. We account for this situation by testing our initial estimate, `7FFFFF0` in our example, plus hexadecimal 10. If the Tuckerman test indicates that we have an underestimate, we continue with the larger value, *e.g.,* `8000000`. Otherwise, we continue with the smaller, *e.g.,* `7FFFFF0`.

Notice that we are repeating a lot of the calculations in the subsequent Tuckerman tests. The formulation of the test in Figure 11 was chosen to minimize the number of operations that must be repeated between applications. Only terms that depend on $\mu_k$, the point at which the test is being applied, must be recomputed. Hence, in the best case where the first test succeeds, the first application takes 9 operations, and each additional takes 1 more for a total of 14 operations. In the worst case, we need 36 operations for the first test and 8 for each remaining test, a total of 76 operations. In the most common case, the second test is definitive so we need 18 operations for the first application and 5 for each additional one for a total of 43. Fortunately, none of the repeated operations is a multiplication.

The alternative to get the error down to a half ULP or less is to do 2 more Newton iterations since we need compute 215 bits to get a 106 bit result. Since we only have 102 bits correct, the first extra iteration only gives us 204-bit accuracy. Repeated application of the Tuckerman test is clearly faster.

Division is a bit trickier. If we had extra bits in the register, we could form $y_{n+1} + \mu_{106}/2$. We don't, so we compute the residual from $B - Ay_{n+1} - A\mu_{106}/2$. Since $y_{n+1}$ is a very good approximation to $B/A$, the first two terms will nearly cancel leaving a positive value since we have underestimated the exact quotient. We have no problem computing $A\mu_{106}/2$, barring underflow, since the result is just a rescaling of $A$.

## 6.3 Other Roundings

All the discussion has assumed we want to return the floating point result closest to the exact answer, round-to-nearest. The IEEE floating point standard[1] includes three other rounding modes. We can also return the result in any of these modes as well.

If we have extra bits in the registers, we handle the different roundings in the following way.

- Round to zero: Return the output from the Newton iteration.

- Round to positive infinity: If the result is positive, add one ULP to the output from the Newton iteration. If the result is negative, return the output of the Newton iteration.
- Round to negative infinity: If the result is negative, subtract one ULP from the output of the Newton iteration. If the result is positive, return the output of the Newton iteration.

If we don't have extra bits in the registers, the following method returns values with these other roundings. Our computed result is the value obtained after the first 5 Tuckerman roundings.

- Round to zero: Return the computed value.
- Round to negative infinity: If the result is positive, add one ULP to the computed result. If the result is negative, return the computed value.
- Round to positive infinity: If the result is negative, subtract one ULP from the computed result. If the result is positive, return the computed value.

These procedures don't handle exact results correctly since they assume we have an underestimate. Simply subtracting one ULP ($\mu_{128}$ or $\mu_{106}$ for the long and short register cases, respectively) from the magnitude of the result before doing the above roundings guarantees an underestimate.

# 7 Test Procedures

To test our algorithms, we generated division and square root problems which presented difficult rounding problems. For division an algorithm is known for generating divisors and dividends so that the correct quotient is almost exactly 1/2 ulp more than a representable floating point number[5]. This is accomplished by solving the diophantine equation

$$2^{k-j}A = BQ + r \bmod 2^k \tag{7}$$

for a given odd divisor $B$, where $r$ is chosen to be an integer near $B/2$, and where $k$ is the precision of the floating point arithmetic. $B$ is chosen to satisfy

$$2^{k-1} \leq B < 2^k,$$

and solutions of Equation 7 are sought for $A$ and $Q$ satisfying the same inequality (with $j = 0$ or 1).

Some difficult square root problems are given by numbers of the form

$$1 + 2^{-k+1}(2j+1) \qquad \text{and} \qquad 1 - 2^{-k}(2j+1)$$

whose square roots are slightly less than

$$1 + 2^{-k+1}(j + 1/2) \qquad \text{and} \qquad 1 - 2^{-k}(j + 1/2),$$

respectively. These almost-exact square roots require $k+1$ bits, making the rounding decision difficult (in all cases the result must be rounded downward). To generate cases requiring close rounding decisions, we attempt to find an integer $x$ satisfying

$$2^{k-1} \leq x < 2^k$$

for which $(x+1/2)^2$ is close to a multiple of $2^k$. We seek solutions of the diophantine equation

$$(x + 1/2)^2 = 2^{k+j}y + \epsilon \bmod 2^k$$

or, multiplying by 4 to remove fractions,

$$(2x + 1)^2 = 2^{k+j+2}y + 1 + 8m \bmod 2^k$$

for various small integer values of $m$, in which $j$ can be 0 or 1. ($4\epsilon$ must be one more than a multiple of eight, since all odd squares are congruent to 1 mod 8). We require $y$ to satisfy the same inequality as $x$. For any $y$ which satisfies the above diophantine equation,

$$\sqrt{2^{k+j}y} = x + 1/2 + o(-m/x),$$

so that the correctly rounded result is $x + 1$ when $m < 0$, and x when $m \geq 0$.

Using these techniques, a large number of test cases were generated both for division and square root, and our implementations of the algorithms presented in this paper successfully rounded each example correctly.

# 8 Conclusions

Division and square root account for a small percentage of all floating point operations, but the time it takes to execute them dominates some calculations. The extra time is even more noticable for quad and multiple precision arithmetic. In this paper we have shown how to speed up these calculations by an appreciable amount.

The primary improvement made to the standard Newton algoirthmn is to multiply by the appropriate factor, the numerator for division and the input argument for square root, before the last iteration instead of after. This trick works because the modified equation is almost exactly the Newton iteration for the desired function instead of a reciprocal approximation. The key observation is that the reciprocal approxmation from the penultimate iteration is sufficiently accurate to be used in the last iteration.

The performance improvement comes from avoiding any multiplications of full precision numbers. An interesting result is that it becomes practical to do quad division and square

root in hardware because we can use the existing floating point hardware. Implementing hardware to multiply two full precision numbers is impractical.

We have also showed how to compute the correctly rounded result with a minimum of additional arithmetic. The method presented is a modification of the Tuckerman test[6] which works for both division and square root. We show how to extend Tuckerman rounding to the case where the registers do not hold any extra bits.

# Acknowledgements

We would like to thank David Bailey, Dennis Brzezinski, and Clemens Roothaan for their help.

# 9 References


[1] American National Standards Institute, Inc. IEEE Standard for Binary Floating-Point Arithmetic. Technical Report ANSI/IEEE Std 754-1985, IEEE, 345 East 47th Street, New York, NY 10017, 1985.

[2] David H. Bailey. A Portable High Performance Multiprecision Package. RNR Technical Report RNR-90-022, NAS Applied Research Branch, NASA Ames Research Center, Moffett Field, CA 94035, May 1992.

[3] Ned Chapin. *360/370 Programming in Assembly Language.* McGraw-Hill, New York, second edition edition, 1973.

[4] Hewlett-Packard, 3404 East Harmony Road, Fort Collins, CO 80525. *HP-UX Reference*, first edition, January 1991.

[5] W. Kahan. Checking Whether Floating-Point Division is Correctly Rounded. Monograph, Computer Science Dept., UC Berkeley, 1987.

[6] Peter Markstein. Computation of Elementary Functions of the IBM RISC System/6000. *IBM J. Res. Develop.*, 34:111–119, 1990.

[7] P. Monuschi and M. Mezzalama. Survey of Square Rooting Algorithms. *IEE Proceedings*, 137(1, Part E):31–40, January 1990.

[8] David A. Patterson and John L. Hennessy. *Computer Architecture A Qualitative Approach.* Morgan Kaufmann, San Mateo, CA, 1990.

# A Basic Operations

The figures in this section contain the code used for the operation counts of the various multiplications in Table 1. These figures assume that the hardware will provide all the bits of the product and the leading quad precision part of the sum of two double precision numbers. If the hardware does not have this capability, the multiplication and addition operations must be replaced with functions such as those shown in Figures 2 and 3, respectively. In this case, each addition counts as 4 operations and each multiplication as 12.

```
void prod(a,b,c)
  double a[], b, c
{
  long double t;
  t = a[0]*b + a[1]*b;
  c = (double t);
}
```

Figure 4: Full times half to half.

```
void prod(a,b,c)
  double a[], b, c[]
{
  long double d, t;
  d = a[0]*b;
  t = d + a[1]*b;
  c[0] = (double t);
  c[1] = t - c[0];
}
```

Figure 5: Full times half to full.

```
void prod(a,b,c)
  double a[], b, c[]
{
  long double s, t, u;
  s = a[0]*b;
  t = a[1]*b;
  u = s + t;
  c[0] = (double u);
  c[1] = u - c[0];
  c[2] = t + ((u-s)-(double t));
}
```

Figure 6: All bits of full times half.

```
void prod(a,b,c)
  double a[], b[], c[]
{
  long double t, u;
  t = a[0]*b[0]+a[0]*b[1]+a[1]*b[0]+a[1]*b[1];
  c[0] = (double t);
  c[1] = t - c[0];
}
```

Figure 7: Full times full to full.

```
void prod(a,b,c)
  double a[], b[], c[]
{
  long double s, t, u, v, w;
  s = a[0]*b[0];
  t = a[0]*b[1]+a[1]*b[0];
  u = a[1]*b[1];
  v = t + s;
  w = t - (v-s) + (u - (double u));
  c[0] = (double t);
  c[1] = t - c[0];
  c[2] = (double w);
  c[3] = w - c[2];
}
```

Figure 8: All bits of full times full.

# B A bc Implementation

These algorithms have been tested using the Unix desk top calculator `bc` with programs written in its C-like language. This utility does integer operations with an arbitrary number of digits and floating point to any desired accuracy. We chose to implement the algorithms with integers because it afforded complete control over the bits included in the operations.

The programs that follow use a few utility routines. The routine `h(a,n)` returns the first `n` base `obase` digits in `a`, where `obase` is the number base used for output; `l(a,n)`, the second `n` base `obase` digits of `a`. The function `table_look_up` is a place holder for the usual procedure for getting the first guess.

Figure 9: Program in `bc` to compute the square root of the full precision number `A` to nearly `2n` bits.

```
define s(A,n){
  auto b,c,d,E,x,y,z
  b = h(A,n)
  x = table_look_up(b)              /* (n/4)-bits     */
  y = h(x*x,n)/n^2
  z = (n^6-h(b*y,n)/2)/n^4
  x = (x*n^2+h(x*z,n))/n^2          /* (n/2)-bits     */
  y = h(x*x,n)/n^2
  z = ((n^6-h(b*y,n))/2)/n^4
  x = h(x*n^2+h(x*z,n),n)/n^2       /* n-bits         */
  /* Last iteration */
  y = h(b*x,n)/n^4                  /* sqrt(h(A))     */
  c = x/2                           /* 1/(2y)         */
  d = h(A-y*y,n)                    /* A - y^2     */
  E = (y*n^4+h(c*d,n))/n^4          /* Almost 2n bits */
  return ( E )
}
```

Figure 10: Program in `bc` to compute the quotient of two full precision numbers, `B/A`, to nearly `2n` bits.

```
define d(B,A,n){
  auto c,d,e,F,x,y,z
  c = h(B,n)
  d = h(A,n)
  x = table_look_up(a)              /* (n/4)-bits     */
  z = (n^4-h(d*x,n))/n^2
  x = (x*n^2+h(x*z,n))/n^2         /* (n/2)-bits     */
  z = (n^4-h(d*x,n))/n^2
  x = x*n^2+h(x*z,n)                /* n-bits         */
  /* Last iteration */
  y = h(c*x,n)/n^4                  /* h(B)/h(A)      */
  e = h(B*n^2-A*y,n)/n^2           /* B - A*h(y)     */
  F = (y*n^4+h(e*x,n))/n^4         /* Almost 2n bits */
  return ( F )
}
```

Figure 11: Tuckerman test. The value of u determines the bit position for the test, $\mu_k$.

```
define t(b,a,y,u){
  g[1] = h(a,n)*h(y,n);           g[2] = h(a,n)*l(y,n)
  g[3] = l(a,n)*h(y,n);           g[4] = h(b,n)-h(g[1],n)
  g[5] = h(g[4],n)-l(g[1],n);     g[6] = h(g[5],n)-h(g[2],n)
  g[7] = h(g[6],n)-h(g[3],n);     g[8] = h(g[7],n)+l(b,n)
  g[9] = h(g[8],n)-h(a,n)*u
  if ( g[9] <= 0 ) return ( 0 )
  if ( g[9] > 8*n^2 ) return ( 1 )
  g[10] = l(a,n)*l(y,n);          g[11] = l(g[2],n)+l(g[3],n)
  g[12] = h(g[10],n)+h(g[11],n);  g[13] = l(g[5],n)-h(g[12],n)
  g[14] = l(g[6],n)+l(g[7],n);    g[15] = h(g[13],n)+h(g[14],n)
  g[16] = l(g[9],n)+h(g[15],n);   g[17] = h(g[9],n)+h(g[16],n)
  g[18] = h(g[17],n)-l(a,n)*u
  if ( g[18] <= 0 ) return ( 0 )
  if ( g[18] > 10*n ) return ( 1 )
  g[19] = l(g[13],n)+l(g[14],n);  g[20] = l(g[10],n)+l(g[11],n)
  g[21] = h(g[20],n)+l(g[12],n);  g[22] = h(g[19],n)-h(g[21],n)
  g[23] = l(g[15],n)+h(g[22],n);  g[24] = l(g[18],n)+h(g[23],n)
  g[25] = l(g[17],n)+h(g[24],n);  g[26] = l(g[16],n)+h(g[25],n)
  g[27] = h(g[18],n)+h(g[26],n)
  if ( g[27] <= 0 ) return ( 0 )
  if ( g[27] > C ) return ( 1 )
  g[28] = l(g[20],n)+l(g[21],n);  g[29] = l(g[19],n)-h(g[28],n)
  g[30] = l(g[22],n)+l(g[23],n);  g[31] = h(g[29],n)+h(g[30],n)
  g[32] = l(g[24],n)+l(g[25],n);  g[33] = l(g[26],n)+h(g[31],n)
  g[34] = h(g[32],n)+h(g[33],n);  g[35] = l(g[27],n)+h(g[34],n)
  g[36] = h(g[27],n)+h(g[35],n)
  if ( g[36] <= 0 ) return ( 0 )
  return ( 1 )
}
```

Figure 12: Getting the error to half an ULP or less on a machine with extra bits in the registers. The functions `u(A)` and `d(B,A)` return the square root and quotient, respectively, with at most four bits in error. Routine `t` performs the Tuckerman test with ULP `u`. `f` and `g` are the square root and quotient with an error no larger than 1/2 ULP.

```
R = u(A)                    /* 1 ULP square root     */
F = R + t(A,R,R,u)          /* 1/2 ULP square root   */
S = d(B,A)                  /* 1 ULP quotient        */
G = S + t(B*nV^2,A,S,u)    /* 1/2 ULP quotient      */
```

Figure 13: Getting the error to half an ULP or less on a machine without extra bits in the registers. We assume an error as large as 15 ULPs.

```
/*        Square Root                              */
R = u(A)          /* Square root                  */
u = 10
V = u*(R/u)       /* Set trailing bits to 0       */
if ( t(A,R,R,2*u) > 0 ) v = v + u  /* Borrow?     */
while ( u > 1 ) {
  V = V + (u/2)*t(A,V,V,u)
  u = u/2
}
F = V + t(A,V,V,1) /* Final rounding              */
```