CONTINUED FRACTIONS FOR HIGH-SPEED AND HIGH-ACCURACY COMPUTER ARITHMETIC

Robert B. Seidensticker

Aydin Computer Systems
Ft. Washington, PA 19034 USA

Abstract

Continued fraction representation has many advantages for fast and high-accuracy computation when compared with positional notation. A continued fraction is a number of the form
$$p_1 + q_1/(p_2 + q_2/(p_3 + \ldots)),$$
where p_i and q_i are integers. Some of the benefits of continued fraction representation for computer arithmetic are: faster multiply and divide than with positional notation, fast evaluation of trigonometric, logarithmic, and other unary functions, easy extension to infinite-precision arithmetic, infinite-precision representation of many transcendental numbers, no roundoff or truncation errors, and improved software transportability because accuracy is not hardware dependent. A unified system for continued fraction arithmetic is given along with an outline of a hardware architecture for evaluating these functions.

1. Introduction and Summary

Research has been done on the use of continued fraction expansions of many unary functions for high-speed computer arithmetic [Tri 77, Bra 74] as well as functions for implementing binary operations on continued fractions [BGS 72]. This article proposes to use previous results plus some new work to build a unified system for practical continued fraction arithmetic. This new work includes error analysis, expansion and clarification of the binary arithmetic algorithms, expansion of the unary algorithms, speed analysis, and a hardware implementation to exploit the advantages of continued fraction arithmetic. Because this paper builds on previous work, some unavoidable recounting is included.

Continued fraction representation is only one of many interesting nonpositional representations. Each of these has its own unique advantages and disadvantages and its own set of application areas in which it performs well (discussed in section 2). The algorithms for conversion between positional, rational, and continued fraction representations are simple. These algorithms, the basic continued fraction identities, and the use of continued fractions in representing numbers are discussed in section 3. An introduction to on-line arithmetic and continued fraction arithmetic, the evaluation of unary functions (such as sin, tan, ln, and sqrt), and the use of floating point are given in section 4. Arithmetic with continued fractions is shown to be simple and fast. Section 5 introduces some application areas; those requiring fast evaluation and high accuracy form only one of several major areas. A hardware architecture is proposed in section 6. The architecture is non von Neumann and allows considerable parallelism.

2. Number Representations

Positional notation is the representation of numbers with strings of digits chosen from a set of digits, as defined by the base. The *position* of each digit determines its value relative to the other digits. Positional representation with a base of 10 (decimal) is widely used, and binary is popular in machine arithmetic. From a computational-effort standpoint, positional representation boasts easy multiplication by powers of the base (shifting), easy comparison and computation of addition and subtraction, and fairly easy computation of multiplication and division. Many other representations exist [OnA 81], each with its own advantages. Most use positional notation to some extent, but all add to it in some way. Several of these representations will be introduced below.

Redundant number systems do not have a unique representation for each integer. These have been used, for example, to speed up arithmetic operations in digit on-line arithmetic [WaE 81] and to provide fault tolerance in the Fibonacci number system [LiN 81]. Mixed radix number systems do not use the same base for all digits. The factorial number system is an example [Knu 69]. Modular ("residue") representation [Knu 69] uses increased parallelism to provide faster addition, subtraction, and multiplication than found with positional representation. It performs less well with division and comparison. Power-of-primes notation [Pra 75] represents a number as the exponents of its prime factors. Multiplication and division are easy, analogous to that performed on a slide rule, but addition, subtraction, and comparison are difficult.

Because the set of integers is closed under addition, subtraction, and multiplication, but not under division, the rational representation [KoM

81] proposes to bypass division by representing every number as a numerator/denominator pair. The four basic binary operations all require about the same amount of time and inversion is simple. Bit-vector machines [Pra 75] use extreme parallelism to calculate addition, subtraction, and multiplication in time $O(\log n)$ and carry-save addition in time $O(1)$, a speed far better than that of positional notation algorithms.

This brief excursion into other number representations is intended only to illustrate the wide variety of alternate representations and the advantages which they can bring to a given problem. This paper intends to make a case for the use of continued fractions in applications requiring fast and precise real arithmetic.

3. Primer on Continued Fractions

Continued fractions notation is another number representation scheme which has its own advantages and disadvantages. Continued fractions are numbers of the form

$$p_1 + \cfrac{q_1}{p_2 + \cfrac{q_2}{p_3 + \dots}} = p_1 + q_1/(p_2 + q_2/(p_3 + \dots)),$$

where p_i and q_i are integers for all i. The fraction can terminate, with q_{n-1}/p_n as the last term, but may not. The characteristics of continued fractions are described in detail elsewhere [Old 63, Knu 69, Wyn 64]; the traits most relevant to computer arithmetic are briefly outlined here.

Research into the theory of continued fractions has been ongoing for hundreds of years. The list of investigators includes the names of Leibnitz, Euler, Lagrange, Laplace, Gauss, Chebyshev, and Klein. While continued fractions were used to a limited extent for the evaluation of constants, the introduction of the computer has increased interest in their computational use.

In the equation above, if $q_i = 1$ for all i, the continued fraction is represented as $[p_1;\ p_2,\ p_3,\ \dots\ p_n]$. If, in addition, $p_1 > 0$, $p_i > 1$, where $1 < i < n$, and $p_n > 2$, the continued fraction is called *regular*. Any arbitrary continued fraction can be converted into a regular continued fraction.

3.1 Conversion Algorithms

Several simple algorithms exist for the conversion between continued fraction, rational, and decimal representations. The following ALGOL-like algorithm will convert the rational number a/b into the regular continued fraction form $[x_1;\ x_2,\ x_3,\ \dots\ x_n]$:

```
$x \leftarrow 1$;
do {$x_i \leftarrow \mathrm{int}(a/b)$;
    $temp \leftarrow b$; $b \leftarrow a - bx_i$; $a \leftarrow temp$;
    $i \leftarrow i + 1$}
until $b = 0$.
```

The x_is are termed "partial quotients." To reverse the process and convert the continued fraction $[x_1;\ x_2,\ x_3,\ \dots\ x_n]$ into the rational number a/b,

```
$a_{-1} \leftarrow 0$; $a_0 \leftarrow 1$;
$b_{-1} \leftarrow 1$; $b_0 \leftarrow 0$;
for $i \leftarrow 1$ to $n$ do
    {$a_i \leftarrow x_i a_{i-1} + a_{i-2}$; $b_i \leftarrow x_i b_{i-1} + b_{i-2}$}.
```

The a_i/b_i fractions are called approximating fractions or convergents. The desired rational number equivalent in value to the continued fraction is the last approximating fraction, a_n/b_n. The set of approximating fractions a_i/b_i is the complete set of best rational approximations, where a "best" approximation is one which comes closest to the actual value with such a small denominator. Additionally, all approximating fractions are in lowest terms (that is, $\gcd(a_i, b_i) = 1$), the fractions alternate between being greater than and less than the actual value (when i is even, the fractions are high), and every approximating fraction is a better approximation than all its predecessors. Another useful function converts a rational number a/b into its equivalent positional representation $d_1.d_2d_3d_4\dots$ in number base B:

```
$d_1 \leftarrow \mathrm{int}(a/b)$;
$a \leftarrow (a - d_1 b)B$;
if $a < 0$ then $a \leftarrow -a$;
$i \leftarrow 2$;
while $a > 0$ do
    {$d_i \leftarrow \mathrm{int}(a/b)$; $a \leftarrow (a - d_i b)B$;
     $i \leftarrow i + 1$}.
```

3.2 Representation of Values

Continued fraction notation represents certain numbers more easily than can positional notation. The positional representation of some rational numbers repeats, while that for others terminates. The base dictates which numbers terminate. For example, $1/7 = 0.\overline{142857}_{10}$, while $1/7 = 0.1_7$. One tenth equals 0.1_{10}, but equals $0.0\overline{0011}_2$. Represented as continued fractions, rational numbers always terminate; there is no base to alter the representation. For example, $1/7 = [1;\ 7]$ and $31/57 = 0.\overline{543859649122807017}_{10} = [0;\ 1,\ 1,\ 5,\ 5]$.

Quadratic irrationalities ("quadratic surds") are numbers of the form $(a + b/\sqrt{D})/c$, where a, b, c, and D are integers, $b \neq 0$, $c \neq 0$, $D > 1$, and D is not a perfect square. Positional notation representations of quadratic irrationalities and all other transcendental numbers are non-repeating and non-terminating, but quadratic irrationalities always repeat when represented as continued fractions and many transcendental numbers are represented as simple infinite series. For example, $\sqrt{2} = 1.414213\dots = [1;\ 2,\ 2,\ 2,\ \dots]$, $e = 2.71828\dots = [2;\ 1,\ 2,\ 1,\ 1,\ 4,\ 1,\ 1,\ 6,\ \dots]$, and $\phi = (1 + 5)/2 = [1;\ 1,\ 1,\ 1,\ \dots]$. Other simple regular or nonregular continued fraction representations exist for π, $\sqrt{n}$, $\sqrt{e}$, and $e^{-1/n}$.

3.3 Error Analysis

The error in an approximating fraction compared to the actual value of a regular continued fraction can be computed by noting that, if i is even--causing the approximating fraction a_i/b_i to be high--the value is bounded on the high side by

$$\frac{x_i a_{i-1} + a_{i-2}}{x_i b_{i-1} + b_{i-2}} = \frac{a_i}{b_i}$$

and on the low side by

$$\frac{(x_i + 1)a_{i-1} + a_{i-2}}{(x_i + 1)b_{i-1} + b_{i-2}} = \frac{a_i'}{b_i'}.$$

The definition of a_i/b_i was given in section 3.1, above. The low bound a_i'/b_i' is computed by noting that x_i is always slightly low, unless x_i is the last term (its inaccuracy is corrected by all the succeeding terms x_j, where $j > i$). The error in x_i is never greater than or equal to 1. Similarly, if i is odd, the error is bounded on the high side by a_i'/b_i' and on the low side by a_i/b_i. The magnitude of the error is then less than $|a_i'/b_i' - a_i/b_i|$. By noting that $a_i = x_i a_{i-1} + a_{i-2}$ and $b_i = x_i b_{i-1} + b_{i-2}$, the error relation simplifies to:

$$|\text{error}| < \frac{b_{i-2}a_{i-1} - a_{i-2}b_{i-1}}{(b_i + b_{i-1})b_i}.$$

However, $b_{i-2}a_{i-1} - a_{i-2}b_{i-1}$ simplifies to $b_{i-4}a_{i-3} - a_{i-4}b_{i-3}$. This produces a numerator for the error relation of $b_{-1}a_0 - a_1 b_0 = 1\times 1 - 0\times 0 = 1$ for i odd and $b_0 a_1 - a_0 b_1 = x_1 \times 0 - 1\times 1 = -1$ for i even. The error relation becomes:

$$|\text{error}| < \frac{1}{(b_i + b_{i-1})b_i}.$$

3.4 Basic Unary Functions

Continued fraction representation allows some basic functions to be computed with almost trivial effort, much as positional notation is amenable to shifting. The multiplicative inverse $1/x$ is computed:

$$1/[x_1;\ x_2,\ \ldots] = [0;\ x_1,\ x_2,\ \ldots].$$

The additive inverse $-x$ is computed:

$$\begin{aligned}-1\times[x_1;\ x_2,\ x_3,\ \ldots] &= [-x_1;\ -x_2,\ -x_3,\ \ldots]\\ &= [-x_1 - 1;\ 1,\ x_2 - 1,\ x_3,\ \ldots].\end{aligned}$$

We can produce $1 - x$ with

$$[-x_1;\ 1,\ x_2 - 1,\ x_3,\ \ldots].$$

Multiplication or division by constants is not simple in regular continued fraction notation, but, using nonregular continued fractions,

$$\begin{aligned}&[p_1 + q_1/(p_2 + q_2/(p_3 + \ldots))]\times A\\ &\quad = Ap_1 + Aq_1/(p_2 + q_2/(p_3 + \ldots))\end{aligned}$$

and

$$\begin{aligned}&[p_1 + q_1/(p_2 + q_2/(p_3 + \ldots))]/A\\ &\quad = p_1 + q_1/(Ap_2 + Aq_2/(p_3 + \ldots)).\end{aligned}$$

These basic functions are computationally very simple. A correspondingly simple positional notation operation might be shifting.

4. Arithmetic on Continued Fractions

4.1 On-Line Arithmetic

"On-line" arithmetic [TrE 77] computes the standard arithmetic functions but requests the operands in pieces or outputs the result in pieces, the most significant piece first. If a function requests the inputs in pieces, it is on-line with respect to input; if it outputs the result in pieces, it is on-line with respect to output. Classical positional arithmetic algorithms are not on-line with respect to either inputs or outputs. In digit on-line arithmetic [WaE 81, Owe 81], the numbers are represented with positional notation and the binary operators (+, −, ×, ÷) request the inputs one digit at a time and output the result one digit at a time. Digit on-line algorithms have been developed for the four basic binary operations and many unary functions. This arithmetic is attractive because it allows increased parallelism in a hardware implementation when compared with standard positional notation arithmetic.

Approximations to positional notation values (i.e., only the first n digits of an $n + m$ digit number) are always low; for this reason, redundant number systems are used to allow digit on-line arithmetic algorithms to be on-line with respect to output. Approximating fractions of a continued fraction, on the other hand, alternate between being too high and too low. Arithmetic functions are able to output continued fraction terms on-line, after the input of a few terms of the arguments, with no need to ever retract them.

4.2 Binary Operators

Binary continued fraction arithmetic has been researched by Gosper; much of this topic is developed in [BGS 72]. Suppose that $z(x,y) = x + y$ were to be computed, where x and y are non-regular continued fractions. The terms of x and y will be requested in an on-line manner, and will be of successively less significance. Given a few terms of x and y, we need state variables in z to record its magnitude so that an occasional on-line output term can be generated. To input a term of x, for example, we can calculate the first term pair (p_1, q_1) and substitute $p_1 + q_1/x'$ in place of x in the expression for z. Unfortunately, when this substitution is made, z simplifies to $(x'y + px' + q)/x'$, which is *not* of the same form as the original $z = x + y$. A more satisfactory and more general function is

$$z(x,y) = (axy + bx + cy + d)/(exy + fx + gy + h),$$

represented as $(a\ b\ c\ d)(e\ f\ g\ h)$ for convenience. After substituting $x = p + q/x'$ into z, we have

$$z(x',y) = \frac{(pa + c)xy + (pb + d)x + qay + qb}{(pe + g)xy + (pf + h)x + qey + qf}$$

or, equivalently,

$$(pa + c\quad pb + d\quad qa\quad qb)(pe + g\quad pf + h\quad qe\quad qf).$$

The form of z is preserved after the substitution. Similar reasoning and similar algebra produce

$$z(x,y') = (pa + b\quad qa\quad pc + d\quad qc)$$
$$(pe + f\quad qe\quad pg + h\quad qg),$$

the form of z after the input of the next term from y. One boundary condition must be noted: substituting $p + q/x'$ for x is valid except when (p,q) is the last term and there is nothing left of x to be called x'. In fact, q itself is meaningless here, since it is a numerator without a denominator. A more accurate equality for the last term is $x = p = p + q/\infty$. The substitution for $z(x',y)$, above, is then accurate except for the substitution of the last (p, q) term. In this case, use

$$z(-,y) = (0\quad 0\quad pa + c\quad pb + d)$$
$$(0\quad 0\quad pe + g\quad pf + h).$$

For inputting the last term of y,

$$z(x,-) = (0\quad pa + b\quad 0\quad pc + d)$$
$$(0\quad pe + f\quad 0\quad pg + h).$$

It is not unreasonable for one operand--x, for example--to terminate before the other. In this case, use the $z(-,y)$ substitution for the last term of x and continue requesting terms from y alone, using $z(x,y')$, until y also terminates.

The state variables of z hold an approximation to the next term of the continued fraction representation of z. At any time, the approximation is $\mathrm{int}[(a + b + c + d)/(e + f + g + h)]$. However, z should not output a continued fraction term (p_i, q_i) until there is confidence that future terms of x or y will not change its value. Actually, z could be premature and output a close approximation to the proper term, using the fact that

$$[\ldots x_i,\ x_{i+1},\ 0,\ x_{i+3},\ \ldots]$$
$$= [\ldots x_i,\ x_{i+1} + x_{i+3},\ \ldots].$$

In other words, by using non-regular continued fractions, an error in an output term of z is not disastrous; the continued fraction representation can recover, due to the fact that non-regular continued fractions are *not* unique representations of real numbers. This is somewhat akin to the effect of using a redundant number system in digit on-line arithmetic. The use of non-regular continued fractions of this sort, however, loses the pleasing property that every term of a regular continued fraction invariably produces a more accurate approximating fraction. Fortunately, it turns out that z can output regular continued fraction terms. The question to be asked of z is: Does z now have enough information to output a reliable term? To answer this, anticipate the changes to z caused by the next term of x and y being either very high or very low. If the unread terms of x and y, even at their extrema, can not affect the integer part of z, then z is ready to output a term. The lowest value of (p, q) (that is, closest to zero) is $(0, 0)$ and the highest is (∞, ∞). If the next term from x was $(0, 0)$, the value of z would become $(c + d)/(g + h)$. If the next term were (∞, ∞) (these terms should really be thought of as very large numbers, rather than infinity), z would be $(a + b)/(e + f)$. If the integer parts of these two terms match, then 1) future terms of x can not affect the next term of z and 2) we know the regular continued fraction value of that next z term. The boundary tests for y are $(b + d)/(f + h)$ for the next term of y being at the low boundary and $(a + c)/(e + g)$ for the next term of y being at the high boundary. Of course, if x has terminated, it is unnecessary to speculate on further terms of x and, instead, only the boundary tests for y are performed; the reciprocal argument for a terminated y also holds. The complete algorithm for determining when and what to output (it is simpler than it looks) is:

```
if $x$ is terminated
 then if $y$ is terminated
       then $z$ is complete. Turn
            the rational number
            $(a + b + c + d)/(e + f + g + h)$
            into a continued fraction
       else if int($(b + d)/(f + h)$) =
               int($(a + c)/(e + g)$)
             then $z$ wants to output
                  (int($(b + d)/(f + h)$), 1)
             else $z$ is not ready to output
 else if $y$ is terminated
       then if int($(c + d)/(g + h)$) =
               int($(a + b)/(e + f)$)
             then $z$ wants to output
                  (int($(c + d)/(g + h)$), 1)
             else $z$ is not ready to output
       else if int($(c + d)/(g + h)$) =
               int($(a + b)/(e + f)$) =
               int($(b + d)/(f + h)$) =
               int($(a + c)/(e + g)$)
             then $z$ wants to output
                  (int($(c + d)/(g + h)$), 1)
             else $z$ is not ready to output.
```

If z is not ready to output a term, the computation of the influence of the unused terms of x and y on z is still helpful in determining which operand should be next consulted for a term. z need not output the terms of a regular continued fraction; however, this seems to be the easiest approach. The term (p, q) to be output is given in the above algorithm. z must be modified to reflect the fact that it has output a term. Noting that $z = p + q/z'$, we can solve for z':

$$z'(x,y) = (qe \;\; qf \;\; qg \;\; qh)\; (a - pe \;\; b - pf \;\; c - pg \;\; d - ph).$$

The choice of initial values for the state variables in z produce different operations on x and y:

for $x + y$, initialize z to $(0\;1\;1\;0)(0\;0\;0\;1)$,
for $x - y$, $z \leftarrow (0\;1\;-1\;0)(0\;0\;0\;1)$,
for $x \times y$, $z \leftarrow (1\;0\;0\;0)(0\;0\;0\;1)$,
for $x \div y$, $z \leftarrow (0\;1\;0\;0)(0\;0\;1\;0)$,
for $x \times a$, $z \leftarrow (0\;a\;0\;0)(0\;0\;0\;1)$,
for $x \div a$, $z \leftarrow (0\;1\;0\;0)(0\;0\;0\;a)$, and
for x, $z \leftarrow (0\;1\;0\;0)(0\;0\;0\;1)$.

The identity function $z(x,y) = x$ can be useful for converting a non-regular continued fraction into a regular representation. More complicated expressions can be computed as easily as a single binary operation. To compute $(5xy - 6)/(3x - 2y + 1)$, for example, we initialize z to $(5\;0\;0\;-6)(0\;3\;-2\;1)$. Whatever the choice of initial values, the computational effort in producing the terms of z is the same. This is an interesting contrast to arithmetic in positional notation, in which addition and subtraction are faster than multiplication and division and considerably faster than the computation of simple equations.

For some unfathomable reason, most (if not all) computer hardware uses positional, instead of continued fraction, representation. The functions in section 3, above, allow conversion between continued fraction and rational representations and between rational and positional representations. To output positional numbers directly from z, use the same boundary checks and output the same p as the next digit, except use

$$z'(x,y) = (B(a - pe) \;\; B(b - pf) \;\; B(c - pg) \;\; B(d - ph))\; (e \;\; f \;\; g \;\; h),$$

where B is the number base of the positional representation.

The eight-variable representation of $z(x,y)$ is adequate for all arithmetic functions, since zs can be combined to produce arbitrarily complex equations. However, it is not unique in maintaining its form after substitution; $z(x) = (ax + b)/(cx + d)$ will work and requires only four state variables, as will $z(x,y,z)$, which requires sixteen variables.

4.3 Comparison

One final binary operation--that of comparison--should be dealt with. Regular continued fractions have the valuable property that they each uniquely represent a real number. Positional notation is not unique in its representations; for example, $7.4_{10} = 7.3\overline{9}$. The value 8/3 in rational notation could equivalently be represented as 16/6 or 32/12, to list only two of an infinite sequence of representations. However, 8/3 can only be represented as [2; 1, 2] in regular continued fraction notation. This trait of unique representation coupled with the fact that regular continued fraction terms are successively less significant gives this simple comparison algorithm for a vs. b:

```
$a \leftarrow 0$;
**while** $a_i = b_i$ **do** $\{i \leftarrow i + 1\}$;
**if** ($i$ is odd and $a_i < b_i$) or
   ($i$ is even and $a_i > b_i$)
 **then** b is greater
 **else** $a$ is greater.
```

The output of the binary operator $z(x,y)$, described above, can produce regular continued fraction terms, which makes it amenable to this kind of comparison. Another valid method which works for all continued fractions and is a common technique for comparison in positional notation, is to subtract the two values and examine the sign of the result. Only one continued fraction output term need be generated.

A list of some of the advantages of continued fraction notation over positional notation for arithmetic would include:

- infinite-precision representation of many transcendental constants
- easily extends to infinite-precision arithmetic. Multi-precision positional arithmetic slows as precision increases
- roundoff or truncation never happen
- accuracy is not hardware dependent with continued fraction machines and the word size is hidden from the user, thus eliminating the source of two common software transportation problems
- an estimate of the error of a truncated continued fraction can be given and, if some operands are known only imprecisely, the one which limited the significance can be identified
- no calculation is performed unnecessarily, such as the computation of a value being multiplied by zero or the computation of a value to a precision greater than required.
- because the operators are on-line, many operations can be in progress concurrently; this leads to a very efficient use of parallel resources.

4.4 Unary Functions

Many important unary functions can be directly represented as continued fractions [Wyn 64, Old 63]; among these are sin, arcsin, tan, tanh, arctan, ln, e^x, $e^{-1/n}$, and $\sqrt{x}$ (see Appendix). Tan(x), for example, is represented as

$$x/(1 - x^2/(3 - x^2/(5 - \ldots))).$$

These functions accept integer arguments very well, but have difficulty with continued fractions as arguments. The problem is that after substituting the continued fraction representation in place of the first instance of x, the first term can be output, but the function becomes the product of the remainder of itself and the remainder of the continued fraction representation of x. This creation of products is merely a nuisance when x only has a few terms or if the required precision of the result is modest, because the total number of ongoing continued fraction multiplications remains low. If x has an infinite number of terms and the required precision is large enough, however, the unary operator will eventually consume all the resources available to it. Using a Taylor series representation of the desired function does not bypass this problem and would be useful only if it converges faster. One simple technique which widens the useful range of the functions is the use of a scaling factor. For tangent, this gives

$$\tan(ax) = \cfrac{x}{1/a - \cfrac{x^2}{3/a - \cfrac{x^2}{5/a - \dots}}} .$$

Letting $a = 10^{-5}$, for example, we can compute the tangent of a wider range of real arguments, and yet continue to operate with integers.

One well-known method for computing square roots which does not expand into products of continued fractions is known as Newton's method. It is a successive approximation ("recurrence relation") technique and computes a better guess to the actual value of the square root with each iteration. The next approximation for $\sqrt{x}$ is:

$$\alpha' \leftarrow \frac{\alpha - x/\alpha}{2} ,$$

where α is the current approximation and α' is the next approximation. After the error becomes less than one, this technique is quadratically convergent (i.e., the number of accurate positional notation digits doubles with each iteration). The recursion in this form can be contrasted with the recursion in a continued fraction. A continued fraction can be defined as $x = p + q/x'$, where the recursion is in the x' term, a value to be expanded in the future. In contrast, Newton's method has the recursion in the α term, a value which has already been computed. This interesting difference makes Newton's method an excellent technique for producing square roots of continued fractions [BGS 72, RaE 81]. It works in a way strangely analogous to that of solving differential equations on an analog computer, which uses integrators and multipliers as some of its building blocks. Given the equation $dx/dy = 10y$, for example, the equation is solved on this computer by first *assuming* the existence of dx/dy. This is integrated (giving y) and multiplied by 10 (giving $10y$), which equals dx/dy, from the original equation. Now that dx/dy is known, it is fed back into the input of the integrator and the task is complete. Newton's method works with continued fractions by providing at each iteration more terms than were available after the previous iteration. Since 1) the equation for α' at each iteration is a function of α--which is a less accurate version of α'--and x, and 2) since both α' and α are on-line with respect to output, it becomes possible to use the output of α' as the input for α. The resulting feedback of data is similar to that in the analog computer. Of course, an initial approximation to $\sqrt{x}$ consisting of a small number of continued fraction terms will be required to provide the first terms for α.

The Newton-Raphson method, of which Newton's method for square roots is a special case, allows the computation of roots to more general equations. The general approximating equation is

$$\alpha' \leftarrow \alpha - \frac{f(\alpha)}{f'(\alpha)},$$

where $f(\alpha)$ is the function. To use this technique to find $\alpha = \sqrt{x}$, convert the equation into a value equal to zero ($f(\alpha) = \alpha^2 - x = 0$), take the derivative to find $f'(\alpha)$, and simplify. $\alpha = \sqrt[n]{x}$ produces the approximation

$$\alpha' \leftarrow \frac{\alpha(n-1) - x/\alpha^{n-1}}{n} .$$

This technique can be used to find roots to other equations, although it should be used with care. The function $\sqrt[n]{x}$, having only one root and being monotonic, produces no problems.

4.5 Speed of Operations

Multiplication and division with continued fractions can be done faster than that with positional notation. The value of the next continued fraction term can be stated probabilistically, but it can be any number. The probability of a term being 1 is 0.41, of it being <= 13 is 0.9, and of it being <= 100 is 0.99 [Knu 69]. Despite this tendency toward low values, the expected value of the next term is infinity; this complicates the calculation of the effort involved with continued fraction arithmetic. The following theorem assists the comparison.

Theorem: Addition, subtraction, multiplication, and division in continued fraction notation are performed faster than multiplication in positional notation.

Proof: The effort involved in positional notation multiplies is a function of the length of the operands, which is the logarithm of the operands. Instead of a value with a single length, continued fraction notation breaks the value into many smaller values with small lengths. By comparing the length of a positional notation value a with that of a continued fraction $[p_1; p_2, \dots p_n]$, it can be shown that

$$\sum_{i=1}^{n} \log p_i \le 2 \log a = 2 \log(p_1 + 1/(p_2 + \dots)).$$

The important fact about this inequality is that the sum of the lengths of the continued fraction terms is proportional to the length of the equivalent positional notation value. Each continued fraction term triggers a fixed number of multiplies in the $z(x,y)$ binary operator discussed above. However, by breaking the large value a into the n continued fraction terms p_i, the multiplication effort is reduced even though the total length is more than the length of the positional notation value. As an example, consider positional notation multiplication being $O(n^x)$, where n is the length of the operands. If there are $2m$ continued fraction terms, each with a length of n/m, the ratio of the effort with continued fractions to that with positional notation is

$$\frac{m(n/m)^x}{n^x} = m^{1-x}.$$

The value m, proportional to the number of continued fraction terms, is a function of n--the larger the value, the more continued fraction terms. However, this relationship is difficult to establish because of the probabilistic nature of the terms. Knuth gives $O(n \log n)$ as the fastest multiply algorithm [Knu 69]. The ratio using this algorithm is

$$\frac{m(n/m) \log (n/m)}{n \log n} = 1 - \log(m/n).$$

Computation of unary functions of scalars, described in section 4.4, above, operates similar to that of the binary operators. However, the comparison is made more difficult by the fact that some unary functions have increasing, not random, terms. Positional notation must use Taylor series expansions or other techniques which converge fairly slowly. One benefit of continued fractions from a speed standpoint is that operations can be easily pipelined. Concurrency in positional notation arithmetic is accomplished only after much more effort.

4.6 Floating Point

Continued fractions best represent numbers in the ranges $\{-1, -\frac{1}{2}\}$ and $\{\frac{1}{2}, 1\}$. With numbers greater than one, the first term, p_1, becomes large, since it holds the integer part of the value. Large values can be inverted, but that puts them in the range $\{\frac{1}{2}, -\frac{1}{2}\}$, which simply shifts the large term from p_1 to p_2. As described above, large terms are rare with continued fractions in the proper range. The low typical term value reduces the load on arithmetic operators. A method for representing large numbers $>> 1$ is to borrow the concept of floating point from positional notation and maintain an exponent where needed. Using this technique, a binary or unary operator which is outputting a first or second term significantly outside the optimum range (perhaps > 100) could multiply or divide the entire value by a suitable power of the base (the use of an exponent requires the selection of some base). This is accomplished by examining the first and second terms, choosing an appropriate correction term, and feeding the stream of terms through a binary operator which performs the correction. The exponent must, of course, remain attached to this new value.

Every regular continued fraction maps uniquely to a real number and vice versa. Some regular continued fractions have unpleasantly large terms (e.g, $2318.59 = [2318; 1, 1, 2, 3, 1, 3]$. The use of an exponent can be thought of as a simple way to get around the unique representation restriction: if we do not like a continued fraction representation, a different exponent allows us to pick another one. We can represent 2318.59 as $2.31859\times10^3 = [2; 3, 7, 4, 1, 12, 2, 2, 13]\times10^3$.

Arithmetic on floating point continued fraction values must be modified to accept the concept of exponents. Little modification need be made for multiplication and division: for multiplication, the new mantissa is the product of the old mantissas, without regard to the exponents and the new exponent is the sum of the old exponents. Division is also computed similar to that in positional notation. Addition and subtraction, as with positional notation, require the alignment of one value to match the exponent of the other. The procedure for this can best be illustrated with an example. Consider the sum $x\times10^5 + y\times10^{13}$, where x and y are continued fraction values. The larger is aligned to the exponent of the smaller: $y\times10^{13}$ becomes $(10^8 y)\times10^5$. The value $10^8 y$ can be computed with the identity

$$[p_1 + q_1/(p_2 + \ldots)]\times A = Ap_1 + Aq_1/(p_2 + \ldots).$$

This will produce a very large first term for y which reflects the difference in magnitudes between x and y. The state variables in z, the result, will also output a very large first term (on the order of 10^8) which must be noticed and reduced to a more reasonable magnitude by modifying the output exponent. Since the difference in the magnitudes of the two powers is known, in this case 10^8, this factor might be used to initialize the appropriate state variable in z to allow z to output terms of an appropriate magnitude. In this example, z could be initialized to $(1\ 0\ 0\ 0)(0\ 0\ 0\ 10^8)$. With this initialization, z both computes the floating point sum or difference, and corrects the output magnitude of z. There is no floating point continued fraction analog to normalization, since there is no finite word size to be conserved.

5. Applications

The most obvious application areas for a continued fraction machine are those requiring high speed, high precision, or both. Many scientific and mathematical areas exhibit these characteristics. Problems in linear algebra are often very error sensitive. Quite a lot has been written on error analysis and its importance [Bar 81, CHH 81, Dem 81]. Those fields requiring unusually high-precision computation with reasonable speed are rare enough that the most popular mainstream computers have failed to provide satisfactory results; these fields might find satisfaction with a machine operating with continued fractions. Some applications make heavy use of transcendental numbers and elementary functions and might appreciate the infinite precision and fast evaluation provided by continued fractions.

Software transportability is another important consideration. Having a type of arithmetic which has no perceptable word size, introduces no error, and produces the same result to arbitrary accuracy regardless of the manufacturer of the computer is of great value. In applications where only accuracy, not speed, is important, these simple algorithms are easily implemented in software.

Before the age of computing machines, mathematicians imagined the exact value of equations, but some of these equations could not be even approximated because of the difficulty in doing so. At that time, mathematicians dealt with the set of all real numbers. With the availabilty of computers, mathematical models are still built, but there is a new tool to work with. While the computer opens many new doors, computer-assisted mathematical speculation is burdened with new constraints: How long will this algorithm take? What will be the error in the result? What can be done to approximate or reduce the error? New techniques for speeding calculations and reducing the effect of computer imprecision have improved its usefulness, but computer scientists are obliged to work with discrete numbers, which are only approximations to real numbers. Hardware to evaluate continued fraction expressions, while not able to evaluate infinite-precision results infinitely quickly, could be a large step in making computer arithmetic more like the continuous mathematics it often attempts to represent.

6. Hardware Architecture for Continued Fraction Processor

Most present-day computers are von Neumann machines, characterized by a single program counter, globally-addressable memory, and several other traits. These can also be termed "control-flow" computers because the elements making up the computer--memory, registers, arithmetic units, etc.--operate after the arrival of control information. This type of operation implies close synchronization among elements. In "data-flow" computers [AgA 82], on the other hand, components await the arrival of data before performing an operation. While data-flow machines are data driven, "reduction machines" are demand driven; where the data-flow machine uses the arrival of data to drive computation, the reduction machine uses the need for data to trigger a search for it. Either type of intercomponent communication seems to fit the needs of a continued fraction computer better than that in a von Neumann machine. The unary and binary operators are executed by components of the form shown in figure 1. Each component can be controlled by a von Neumann processor, although an all-hardware implementation would speed operation considerably. Two operands are received from an external source (although only one would be used in the computation of a unary function) and operated on internally; the result terms are output as soon as possible. Due to the small expected value of the terms, the processor will typically operate on very small numbers (90% of the terms will be 13 or less).

On-line functions of this sort typically experience an "on-line delay" which is the number of input terms required from each operand before the first output term is available. The binary algorithms presented have a typical on-line delay of 2 - 4.

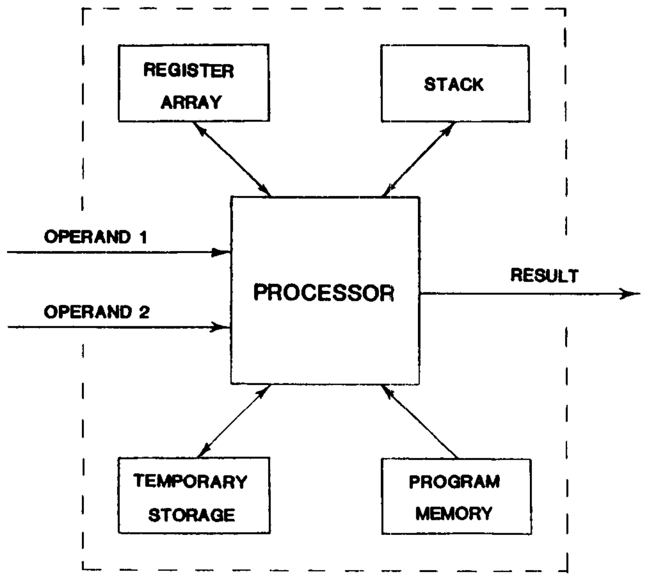

Figure 1: Continued Fraction Processing Element

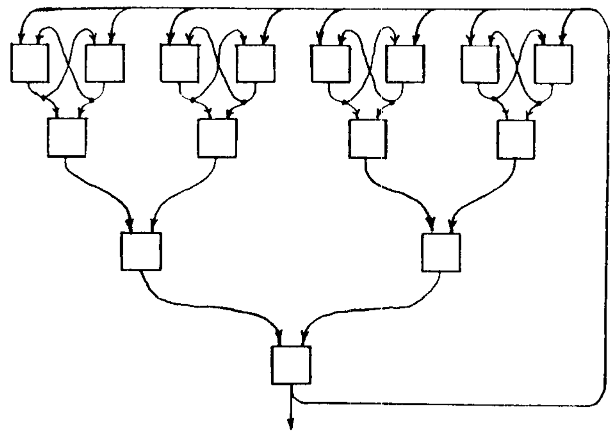

Figure 2: Tree Structure of many Processing Elements

A typical computer might be composed of many processing elements, possibly connected in the form of a tree, as shown in figure 2. The tree structure imitates the form of an equation, to easily represent an equation and maximize the time each processing element is used. The tree structure expands gracefully and easily accepts extra processing elements to fit the needs of the application. If too few processing elements are present to serve the task, each element can expand into a virtual subtree, mimicking a tree with many processing elements. The subtrees fit together to form a virtual tree large enough for the entire task. The state variables of many different virtual processors can be stored in the register array and the stack controls recursion to the different functions. The register array and the stack control communication between the potentially many virtual elements in one physical element.

Memory for the storage of results is not used in the same way as it is with control-flow computers. In fact, data flow computers prohibit side effects. Pure LISP is one computer language which has no side effects. The FORTH language without variables communicates via a single stack and also has no side effects. Allowing side effects has a debilitating effect on some control-flow SIMD and MIMD (Single/Multiple Instruction stream, Multiple Data stream) processors. Typically, many processors and many memories need to communicate in a flexible way. Crossbar switches and banyan networks are two of perhaps a dozen popular interconnection strategies. The I/O problem is so characteristic, however, that systems of this sort which become I/O bound are said to be battling the "von Neumann bottleneck." The machine proposed here uses a compromise between the attitudes toward memory of the von Neumann and data-flow schools of thought.

It has been said that data in transit is no different from data stored in memory--each looks like a communication stream relative to the other and sees itself as memory. This is basic to the philosophy of the data-flow machine. While a von Neumann machine might create five temporary variables before generating a useful result, a data-flow machine would calculate the result directly from the inputs without the use of temporaries. This philosophy works well for the proposed continued fraction machine. However, many applications need to store a value in continued fraction form for use at a later time just as the LISP and FORTH computer languages can be benefitted by the use of variables. The variable-length continued fraction can be stored as a linked list made of small memory words, capitalizing on the fact that the terms are usually very small. Storage of values is a von Neumann operation so, to avoid the von Neumann bottleneck, storage for values is put with each processing element. This can never cause communication bottlenecks and does not limit the expansion of the tree of processing elements.

As discussed above, the binary operations are performed very quickly. Both binary and unary operators produce a fast stream of continued fraction terms which allows for considerable concurrency. Additionally, unlike some dedicated parallel processors, this hardware is very flexible and can compute different expressions with little reconfiguration effort. As in other data-flow machines, repeated and vector operations are encouraged, since they often maximize the usage of the hardware.

The computational elements (the nodes of the execution tree) are simple enough and have sufficiently limited I/O to be considered for VLSI implementation. A VLSI continued fraction processor node could be small and inexpensive. A processor formed from a tree of such nodes would have two main uses: it could be the arithmetic unit of a stand-alone processor (either von Neumann or data flow) or it could be a processing peripheral to a von Neumann machine, much like an array processor.

This design exists only on paper; however, its benefits appear to merit further study.

7. Summary and Suggestions for Further Work

The advantages of the continued fraction arithmetic system described here are:

- infinite-precision representation of many transcendental constants
- extensibility to infinite-precision arithmetic
- no roundoff or truncation errors
- superior software transportability
- excellent control of errors
- multiply and divide faster than with positional notation
- computation of trigonometric, logarithmic, exponential, and other functions simply and quickly
- highly parallel hardware implementation.

The attempt was made here to merge the useful traits of continued fractions from many sources along with some new contributions into a unified system for mathematical computation. One gap in this system is the lack of unary operators which easily take continued fractions as arguments. The successive approximation techniques work very well; perhaps more of such methods can expand the collection of useful functions. A more detailied identification of the application areas is also called for.

Continued fraction unary functions have been developed which replace multiplication with shifts [Tri 77, Bra 74]. A similar speedup for binary operations would be a significant improvement.

The particularly fast operations of a given number representation are exploited by some algorithms to produce unusually fast results. The CORDIC technique [Wal 71], for example, uses the fact that shifts and adds in positional notation are typically much faster than multiplies and divides; the resulting algorithm is simple and produces very fast computation of elementary functions. The peculiar traits of continued fractions hold promise for some equally valuable new algorithms.

A unified and coherent arithmetic has been presented here. Arithmetic on continued fractions is practical; in fact, the algorithms are very simple. The unique benefits of this representation justify more research into analysis of algorithms and the hardware implementation.

8. Acknowledgements

I wish to thank Jeff Rosenthal of Aydin for considerable assistance with some parts of this research and I appreciate the referees' help. Additionally, I would like to applaud MIT's AI memo #239, "HAKMEM," as a compendium of unusual and valuable concepts, including (of course) continued fractions.

9. References

[AgA 82] T. Agerwala and Arvind, "Data Flow Systems," Computer (Feb., 1982), pp. 10-13.

[Bar 81] J. Barlow, "On the Distribution of Accumulated Roundoff Error in Floating Point Arithmetic," Proceedings of 5th Symposium on Computer Arithmetic (May, 1981), pp. 100-105.

[BGS 72] M. Beeler, R. W. Gosper, and R. Schroeppel, "HAKMEM," AI Memo #239 (MIT, Feb., 1972), pp. 36-44.

[Bra 74] A. Bracha-Barak, "Application of Continued Fractions for Fast Evaluation of Certain Functions on a Digital Computer," IEEE Trans. on Computers (March, 1974), pp. 301-309.

[CHH 81] M. Cohen, V. C. Hamacha, and T. E. Hull, "CADAC: An Arithmetic Unit for Clean Decimal Arithmetic and Controlled Precision," Proc. of 5th Symp. on Comp. Arithmetic (May, 1981), pp. 106-112.

[Dem 81] J. Demmel, "Effects of Underflow on Solving Linear Systems," Proc. of 5th Symp. on Comp. Arithmetic (May, 1981), pp. 113-119.

[KoM 81] P. Kornerup and D. W. Matula, "An Integrated Rational Arithmetic Unit," Proc. 5th Symp. on Comp. Arithmetic (May, 1981), pp. 233-240.

[Knu 69] D. E. Knuth, The Art of Computer Programming, vol. 2 (Addison Wesley, 1969).

[LiN 81] P. Ligomenides and R. Newcomb, "Complement Representations in the Fibonacci Computer," Proc. 5th Symp. on Comp. Arithmetic (May, 1981), pp. 6-9.

[Old 63] C. D. Olds, Continued Fractions (Random House, 1963).

[OnA 81] S. Ong and D. E. Atkins, "Towards Quantitative Comparison of Computer Number Systems," Proc. of 5th Symp. on Comp. Arithmetic (May, 1981), pp. 21-33.

[Pra 75] V. Pratt, Course Notes for 6.046 (MIT, 1975).

[RaE 81] C. S. Raghavendra and M. D. Ercegovac, "A Simulator for On-Line Arithmetic," Proc. of 5th Symp. on Comp. Arithmetic (May, 1981), pp. 92-98.

[TrE 77] K. S. Trivedi and M. D. Ercegovac, 'On-Line Algorithms for Division and Multiplication," IEEE Trans. on Computers (July, 1977).

[Tri 77] K. S. Trivedi, "On the Use of Continued Fractions for Digital Computer Arithmetic," IEEE Trans. on Computers (July, 1977), pp. 700-704.

[WaE 81] O. Watanuki and M. D. Ercegovac, "Floating-Point On-Line Arithmetic: Algorithms," Proc. of 5th Symp. on Comp. Arithmetic (May, 1981), pp. 81-86.

[Wal 71] J. S. Walther, "A Unified Algorithm for Elementary Functions," Proceedings of AFIPS 1971 Spring Joint Computer Conference (1971), pp. 379-385.

[Wyn 64] P. Wynn, "On Some Recent Developments in the Theory and Application of Continued Fractions," J. SIAM Numer. Analysis (1964), pp. 177-197.

Appendix

Some useful unary continued fraction functions follow:

$$\sqrt{a^2+b} = a + \cfrac{b}{2a + \cfrac{b}{2a + \cdots}}$$

$$\sin x = \cfrac{x}{1 + \cfrac{x^2}{(2\times 3 - x^2) + \cfrac{2\times 3x^2}{(4\times 5 - x^2) + \cdots}}}$$

$$\tan x = \cfrac{x}{1 - \cfrac{x^2}{3 - \cfrac{x^2}{5 - \cdots}}}$$

$$\tanh x = \cfrac{x}{1 + \cfrac{x^2}{3 + \cfrac{x^2}{5 + \cdots}}}$$

$$\arctan x = \cfrac{x}{1 + \cfrac{1\times x^2}{3 + \cfrac{4\times x^2}{5 + \cfrac{9\times x^2}{7 + \cdots}}}}$$

$$\ln(1+x) = \cfrac{x}{1 + \cfrac{1^2x}{2 + \cfrac{1^2x}{3 + \cfrac{2^2x}{4 + \cdots}}}}$$