Presubstitution, and Continued Fractions

W. Kahan
E. E. & C. S. Dept
Univ. of Calif.
Berkeley CA 94720

Abstract:
This is a case study of attempts to program the computation of a continued fraction and its first derivative in a way that avoids spurious behavior caused by roundoff, over/underflow and, most important, division by zero. The program is easier to find for a machine that does not merely abort computation but continues in a reasonable way after division by zero, as do machines that meet IEEE standards 754 and 854 for floating-point arithmetic. And machines that offered a feature that I call "Presubstitution" would be particularly easy to program. But programming on other machines is a challenge that I prefer to leave to someone else.

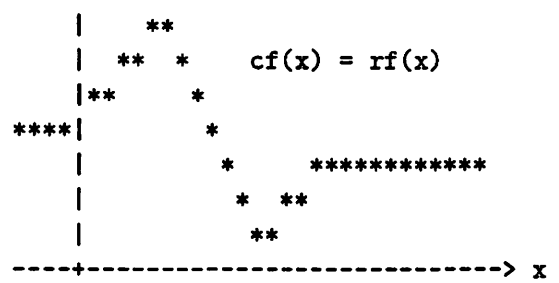
Introduction:
A typical continued fraction is

$$cf(x) := 4 - \cfrac{3}{x - 2 - \cfrac{1}{x - 7 + \cfrac{10}{x - 2 - \cfrac{2}{x - 3}}}} \qquad \text{(4 div.)},$$

for which an algebraically equivalent form is

$$rf(x) := \frac{622 - x(751 - x(324 - x(59 - 4x)))}{112 - x(151 - x(72 - x(14 - x)))} \qquad \text{(7 mul., 1 div.)}.$$

Both expressions represent the same rational function, one whose graph is smooth and unexceptional:



Although $rf(x) = cf(x)$ as rational functions go, they are not computationally equivalent ways to compute that function. For instance,

$$rf(1) = 7, \quad rf(2) = 4, \quad rf(3) = 8/5, \quad rf(4) = 5/2;$$

but the corresponding values of $cf(x)$ encounter divisions by zero that stop many computers. However, provided the computer conforms to IEEE 754 or 854, in which case

$$\pm(nonzero)/0 = \pm\infty, \quad \infty \pm (finite) = \infty, \quad (finite)/\infty = 0,$$

we find that the computed values of $cf(x) = rf(x)$ at those arguments. On the other hand, if $|\ x\ |$ is so big that x^4 must overflow, then the computed value of $cf(x) = cf(\infty) = 4$ but $rf(x)$ encounters (overflow)/(overflow) , which yields something else. And at arguments x between 1.6 and 2.4 the formula $rf(x)$ suffers from roundoff usually much worse than $cf(x)$. For instance, typical values obtained for $rf(x)$ and $cf(x)$ at a few values x are tabulated below, as computed on a calculator that rounds to 10 sig. dec., together with a value computed correctly:

x :	1.6063193	1.959	2.101010101	2.3263	2.4005
"rf":	8.752378*651*	4.8231*32981*	2.30482*2296*	.79661*66480*	.74070*74217*
"cf":	8.752378*523*	4.823133133	2.30482234*6*	.79661657*80*	.74070737*80*
..f :	8.752378524	4.823133133	2.304822344	.7966165794	.7407073784

That is why $cf(x)$ is preferable to $rf(x)$ if division is not too much slower than multiplication and if division by zero produces something huge enough instead of stopping computation.

Many other ways to compute this function are worth considering. For instance,

$$rf(x) := 4 - 3(x-2)((x-5)^2+4)/(((x-5)^2+3)(x-2)^2+x)$$

entails less work (5 mul., 1 div.) and much less trouble with roundoff, and a little less trouble with overflow on those machines that overflow to a huge number instead of just stopping. But if the coefficients of $cf(x)$ were arbitrary floating-point numbers instead of simply integers like 4, 3, 2, 1, 7, ... then the resulting coefficients of $rf(x)$, regardless of which form be chosen, would almost certainly be contaminated by roundoff to an extent difficult to ascertain. Ideally we should prefer to compute $cf(x)$ as it stands, but that may be impractical on a machine that balks at division by zero. What should we do then?

There is another trick to consider. Choose a tiny positive number ε so small that $1.0 \pm \sqrt{\varepsilon}$ rounds to 1.0 when computed in the same floating-point arithmetic as is being used to compute $cf(x)$, but not so small that $10/\varepsilon$ overflows. Then very slightly alter the expression for $cf(x)$ thus:

$$cf(x) := 4 - \cfrac{3}{(x-2) - \cfrac{1}{((x-7) + \cfrac{10}{((x-2) - \cfrac{2}{(x-3)+\varepsilon}) + \varepsilon}) + \varepsilon}}.$$

Both versions of $cf(x)$ yield exactly the same computed values except that division by zero never happens to the latter version ! In general, if $cf(x)$ were more complicated, with coefficients that ran over a very wide range, the values to use for ε might be difficult even for a skilled error-analyst to determine. This is not a trick that typical programmers might be expected to find.

A Continued Fraction and its Derivative:

How should programmers generally deal with continued fractions and similar computations in which divisions by zero that might occur would be harmless if handled properly? Consider

the general "*Jacobi*" continued fraction, which takes the form

$$f(x) := a_0 + \cfrac{b_0}{x + a_1 + \cfrac{b_1}{x + a_2 + \cfrac{b_2}{x + \cfrac{\ldots}{x + a_N}}}}.$$

Continued fractions like this figure in formulas for various transcendental functions of interest to mathematical physicists and statisticians. For instance, for large $y > 0$,

$$\int_y^\infty \exp(-t^2/2)dt = y \times \cfrac{\exp(-y^2/2)}{y^2 + 1 - \cfrac{2}{y^2 + 5 - \cfrac{12}{y^2 + 9 - \cfrac{30}{y^2 + 13 - \cfrac{56}{y^2 + 17 - \ldots}}}}}$$

A Jacobi fraction can be computed by a very simple recurrence:

$f := a_N$;
for $j = N - 1$ to 0 step -1 do $f := a_j + b_j/(x + f)$;

after which $f = f(x)$ provided division by zero, if it occurs, produces a sufficiently huge quotient (like ∞) rather than stop the machine. Any expedient introduced here to preclude division by zero or to handle it some other way would encumber that simple recurrence intolerably. But ∞ is no panacea; it cannot cure all divisions by zero equally easily. Let us turn to a more realistic illustration of the role played by ∞.

Both $f = f(x)$ and its first derivative $f' = f'(x) = df(x)/dx$ are generated simultaneously by the recurrence

$f' := 0$; $\quad f := a_N$;
for $j = N - 1$ to 0 step -1 do
$\{\ d := x + f$;
$\quad q := b_j/d$;
$\quad f' := -(1 + f')q/d$;
$\quad f := a_j + q\quad\}$;

provided the divisor d never vanishes. But if $d = 0$ at some pass around the loop, followed by $q = f = \infty$ and $f' = \infty$, the next pass around the loop will put $d = \infty, q = 0$ and $f = a_j$ correctly, but $f' = \infty * 0$ or ∞/∞, which turns into NaN (Not a Number) when arithmetic conforms to IEEE 754 or 854. This NaN is not the correct value for f'. One way to get f' correctly is to use the $\varepsilon-$ trick; replace the statement "$d := x + f$;" by "$d := (x + f) + \varepsilon$;" for some suitably tiny positive ε that has to be computed differently around each pass of the loop. But this is an expedient for error-analysts, not for programmers who seek algebraic and combinatorial cures for programming maladies. Alas, all other recurrences known to cope with division-by-zero and spurious over/underflow correctly seem obliged to include some kind of test-and-branch. The simplest such scheme I know is this:

Choose a positive ε so tiny that $1.0 - \sqrt{\varepsilon}$ rounds to 1.0;
$f' := 0$; $\quad f := a_N$;

for $j = N - 1$ to 0 step -1 do

$$\{\ d := x + f; \quad d' := (1 + f') + \varepsilon;$$
$$q := b_j/d;$$
$$\text{if } |\ d'\ | = \infty \text{ then } f' := p$$
$$\text{else } f' := -(q/d)d';$$
$$f := a_j + q; \quad p := b_{j-1}d'/b_j \quad \}.$$

Complicated though it may appear, this recurrence is far simpler than the proof that it is correct, which involves taking limits as $d \to 0$ and a verification that $d' \neq 0$. The last condition is assured by a simple version of the ε-trick, which prevents 0/0 or 0*∞ in examples like the following at $x = 0$:

$$f(x) := \ldots$$
$$\ldots + 1 + \cfrac{1}{x + 1 + \cfrac{1}{x - 1 + \cfrac{1}{x + 1}}}$$

On a vectorized computer like the CRAYs the last recurrence is applicable to vectors x, f', f, d, d', q and p elementwise provided the conditional statement "if $|d'| = \infty$..." is replaced by a vectorized conditional assignment

$$\text{"}f' := \text{ if } |\ d'\ | = \infty \text{ then } p \text{ else } -(q/d)d';\text{"}$$

that exploits the computer's ability to select a vector's elements in accordance with a bit-mask derived from the boolean expression "$|\ d'\ | = \infty$". On a heavily pipelined computer with multiple arithmetic units the operations in the recurrence will overlap to an extent indicated partially by the way the statements have been written. But if division by zero is disallowed, or if division is too much slower than multiplication, all programs I know to calculate f and f' robustly seem obliged either to branch in ways that slow down many of today's fastest computers, or else to exploit extremely devious perturbations calculated to vanish amongst the rounding errors.

Presubstitution:
In the past, programming languages have required that exceptions like Overflow and Division-by-Zero be either precluded by apt tests and branches, or else handled by "Error-Handlers" invoked via special statements like

"ON ERROR GOTO <*line*>" or "ON ERROR GOSUB <*line*>" in *BASIC*,
"ON <*Error-Condition*> <*Action to be taken*>" in *PL-1*.

These statements require a *Precise Interrupt* if their error- handling actions are to be followed by resumption of the program from the point where the *Error-Condition* was detected. But a Precise Interrupt is expensive to implement in fast computers that achieve part of their speed by overlapping instructions, by pipelining them, or by vectorizing. The trouble is that several instructions may be executing simultaneously when one of them signals an exception, and then the computer will have to undo whatever was done by instructions that were issued after the one that signalled but before the signal was received. Otherwise some variables referenced by the *Action to be taken* might have changed since the exceptional

instruction was issued. Much extra hardware is needed to remember what was done so it can be undone.

A different approach provides most of the benefits of those kinds of error-handling statements at a small fraction of the cost. The essential insight is the realization that, if an exceptional operation can be so redefined by the *Action to be taken* as would justify resuming execution afterwards, then mathematicians would call the exception a *Removable Singularity*. Examples are ...

Operation	Type	Example			
Add/Subtract	$\infty - \infty$	$\cot(x) - 1/x$	$\rightarrow 0$	as	$x \rightarrow 0$,
Multiply	$0 * \infty$	$x \cot(x)$	$\rightarrow 1$	as	$x \rightarrow 0$,
Divide	$0/0$	$x/\sin(2x)$	$\rightarrow 1/2$	as	$x \rightarrow 0$,
	∞/∞	$x/(3x+1)$	$\rightarrow 1/3$	as	$x \rightarrow \infty$.

IEEE Standards 754 and 854 prescribe NaN as the default result of such operations because any other value, prescribed without knowledge of the exceptional circumstances, would cause confusion more often than help; that is why the standards do not assign 1 to 0/0 as APL does. Only the programmer's special *Action to be taken* can remove the singularity correctly.

If that special *Action to be taken* is complicated enough, a well placed test-and-branch in the program costs little more than what has to be there anyway. It may cost less at run-time than an ON <*Condition*> <*Action*> statement whose *Action* inhibits all concurrency that would interfere with a precise interrupt at any operation that the hardware thinks might encounter the *Condition*. An explicit test-and-branch encumbers only those operations that the programmer thinks might encounter the *Condition*.

When the programmer intends that execution resume after a very simple *Action*, the inhibition of concurrency required to achieve precise interrupts is too high a price to pay. Another mechanism would be cheaper; consider a statement of the form

ON <*Condition*> PRESUBSTITUTE <*Value*>

that caused any *Condition* drawn from the set

$$\{\infty - \infty, \quad 0 * \infty, \quad 0/0, \quad \infty/\infty\}$$

to deliver *Value* instead of NaN. The hardware required to implement this statement entails only presettable registers in lieu of the read-only registers from which a NaN would be drawn. The programmer has to precompute *Value* before initiating any operation that might encounter the *Condition*. Let's see how well this scheme would handle a simple example first:

Define $S(x) := \sin(x)/x$ with the understanding that $S(0) := 1$, and suppose we wish to compute the vector $w := S(v)$ elementwise. A very simple program would suffice:

ON 0/0 PRESUBSTITUTE 1.0 ;
FOR j IN $\{1..DIM(v)\}$ DO $w_j := \sin(v_j)/v_j$ IN PARALLEL.

No tests; no branches; no precise interrupts. The value 1.0 would go to the divider in anticipation of invalid divides, and would be used only if some v_j turned out to be 0 .

How well would presubstitution handle the continued fraction $f(x)$ and its derivative $f'(x)$? Consider this program:

On 0/0 or ∞/∞ presubstitute ∞;
$f' := 0$; $f := a_N$;
for $j = N - 1$ to 0 step -1 do
{ $d := x + f; d' := 1 + f'$;
$q := b_j/d$;
$f' := -(d'/d)q$; $f := a_j + q$;
on $0 * \infty$ presubstitute $b_{j-1}d'/b_j$ }.

No tests; no branches; no ε. This program works well if all variables are scalars and arithmetic is overlapped or concurrent.

But if all unsubscripted variables were vectors interpreted elementwise, then the presubstitution operation would have to be interpreted elementwise too, which is impractical on a machine with the CRAY's architecture where operations upon vector registers are performed in a few pipelined arithmetic elements. On such a machine the most practical program would resemble the earlier one with a vectorized conditional assignment statement. So presubstitution is no panacea for exceptions on vectorized machines. It is a compromise between expensive hardware that interrupts precisely to handle exceptions and cheap hardware that ignores them, between unfettered software allowed to do anything in response to exceptions and software in a straitjacket.

Presubstitution would be useful also for handling some other classes of exceptions, namely

Over/underflow, to a presubstituted magnitude with its sign inherited from the operation's true result;
Division by Zero, or any operation that would produce ∞ exactly from finite operands, to a finite presubstituted magnitude with inherited sign.
Dereferencing a Null Pointer, to a presubstituted entity.
Element Outside an Array (or other data structure), to a presubstituted entity.
Uninitialized variable, to a presubstituted entity.

The last three exceptions' presubstituted entities could be NaNs for debugging, or zeros for compatibility with certain higher- level language interpreters; presubstituting instead of aborting could simplify beginnings or ends of loops, especially in matrix functions and in programs that search through data structures. On heavily pipelined machines, compilers would be allowed to overlap floating-point operations and anticipatory fetches of data, with the expectation that an invalid fetch would not abort computation but would instead fetch a presubstituted value destined for discard.

Further discussion leads beyond the intended scope of this paper.

Acknowledgements etc.:
Parts of this work have been supported at times by research grants from The U. S. Office of Naval Research, from the Air Force Office of Scientific Research, and from DARPA.

The IEEE Standards 754 and 854 are explained in "A Proposed Radix- and Word-Length-Independent Standard for Floating-Point Arithmetic" by W. J. Cody et al. in the IEEE magazine MICRO (Aug. 1984) pp. 86-100.

Continued fractions for various transcendental functions can be found in the *Handbook of Mathematical Functions* edited by M. Abramowitz and Irene Stegun, no. 55 in the National Bureau of Standards Applied Math. Series, issued in 1964 but reprinted now

by Dover, New York. Programs to convert them from one form to another can be found in *Computer Approximations* by J. F. Hart *et al.*, published in 1968 by Wiley but now reprinted by Krieger in Huntington, New York.

Presubstitution has been implemented by David Barnett on a DEC VAX™ running 4.3 BSD Berkeley UNIX™, and on a SUN™ III ; his work is still in progress.