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. 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, such programs are straightforward. On machines that offer a feature that I call "Presubstitution" such programs are even pleasant. But programming continued fractions on other machines is a chore I prefer to leave to someone else.

Introduction:

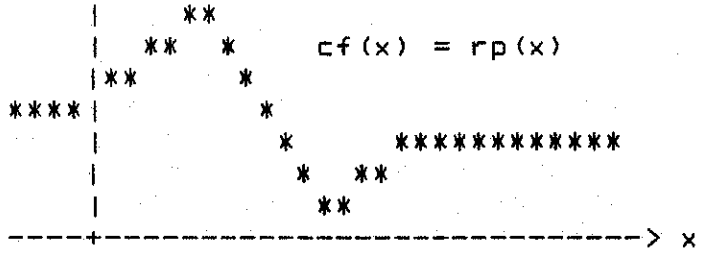
A typical continued fraction is

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

for which an algebraically equivalent ratio of polynomials is

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

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



Although $rp(x) = cf(x)$ as rational functions go, they are not computationally equivalent ways to compute that function. For instance,
$rp(1) = 7$, $rp(2) = 4$, $rp(3) = 8/5$, $rp(4) = 5/2$;
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
$\pm(\text{nonzero})/0 = \pm\infty$, $\infty \pm (\text{finite}) = \infty$, $(\text{finite})/\infty = 0$,
computed values of $cf(x)$ and $rp(x)$ agree at those arguments too. 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 $rp(x)$ encounters $(\text{overflow})/(\text{overflow})$, which yields something else. And at arguments x between 1.6 and 2.4 the formula

rp(x) suffers from roundoff usually much worse than cf(x) . For instance, typical values obtained for rp(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
"rp":	8.752378651	4.823132981	2.304822296	.7966166480	.7407074217
"cf":	8.752378523	4.823133133	2.304822346	.7966165780	.7407073780
..f :	8.752378524	4.823133133	2.304822344	.7966165794	.7407073784

That is why cf(x) is preferable to rp(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,

rp(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 rp(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:

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

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

$$f(x) := a_0 + \cfrac{b_0}{x + a_1 + \cfrac{b_1}{x + a_2 + \cfrac{b_2}{x + \ddots + \cfrac{\ddots}{x + a_N}}}} \quad (\text{all } b_j \neq 0) .$$

Continued fractions like this figure in formulas for various

transcendental functions of interest to mathematical physicists and statisticians. For instance, for big $y >> 0$ and $x = y^2$,

$$\int_y^\infty \exp(-t^2/2)\,dt = y\exp(-x/2)\Big/\left(x+1 - \frac{2}{x+5 - \frac{12}{x+9 - \frac{30}{x+13 - \frac{56}{x+17 - \ldots}}}}\right)$$

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

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 ε-trick; replace the statement " d := x + f ;" by " d := (x+f) + ε ;" 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-√ε rounds to 1.0 ;
f' := 0 ;  f := a_N ;
for j = N-1 to 0 step -1 do
                        { d := x + f ;   d' := (1+f') + ε ;
                          q := b_j/d ;
                          if  |d'| = ∞  then  f' := p
                                        else  f' := -(q/d)d' ;
                          f := a_j + q ;  p := b_{j-1}d'/b_j  } .
```

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 $0*\infty$ in examples like the following at $x = 0$:

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

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'| = ∞ ... " is replaced by a vectorized conditional assignment

" f' := if |d'| = ∞ then p else -(q/d)d' ; "

that exploits the computer's ability to select a vector's elements in accordance with a bit-mask derived from the boolean expression " |d'| = ∞ " . 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 to-day's fastest computers, or else to exploit extremely devious perturbations contrived 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 perhaps 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 can be 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 may be needed to remember what was done so it can be undone. Even if archtectural ingenuity provides precise interrupts neatly with little extra hardware, as appears to have been done for the MIPS R2000, compile-time "optimizations" may have so rearranged the operations' order that some exceptions cannot be located precisely relative to the source code, thus vitiating ON ... statements.

A different approach provides most of the benefits of those kinds of error-handling statements at a small fraction of their 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 might well call the exception a *Removable Singularity*. Examples are ...

Operation	Type	Example			
Add/Subtract	∞−∞	cot(x) − 1/x	→ 0	as	x → 0 ,
Multiply	0*∞	x cot(x)	→ 1	as	x → 0 ,
Divide	0/0	x/sin(2x)	→ 1/2	as	x → 0 ,
	∞/∞	x/(3x+1)	→ 1/3	as	x → ∞ .

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 IEEE standards eschew APL's assignment 0/0 = 1 . Only the programmer's special *Action to be taken* can remove the singularity correctly.

An ON <*Condition*> <*Action*> statement is not always preferable to a well-placed test-and-branch. To choose between them programmers must weigh not only the probability of the *Condition* and the relative complexity of the *Action* but also the extent to which the ON ... statement may require inhibition of all concurrency that would interfere with a precise interrupt at any operation the hardware thinks might generate the *Condition*. An explicit test-and-branch encumbers only those (presumably fewer) 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 can be much cheaper; consider a statement of the form

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

that causes any *Condition* drawn from the set

{ ∞−∞ , 0*∞ , 0/0 , ∞/∞ }

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*. And the principle that requires anything written to be readable requires a statement

<*Variable*> := VALUE PRESUBSTITUTED ON <*Condition*>

to be available too. 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:

```
SaveIt := VALUE PRESUBSTITUTED ON "0/0"
ON "0/0" PRESUBSTITUTE 1.0 ;
FOR j IN {1..DIM(v)} DO w_j := sin(v_j)/v_j IN PARALLEL ;
ON "0/0" PRESUBSTITUTE SaveIt .
```

No tests; no branches; no precise interrupts; no bubbles in pipes; no synchronization operation in the loop. Presubstitution acts like a special divide instruction to send the value 1.0 to the divider in anticipation of invalid divides; this value will be used only if some divisor v_j turns out to be 0 .

How well would presubstitution handle the continued fraction f(x) and its derivative f'(x) ? Compare earlier programs with this one:

```
SaveIt() := values presubstituted on  "0/0" or "∞/∞" or "0*∞" ;
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*∞" presubstitute  b_{j-1}d'/b_j } ;
on  "0/0" or "∞/∞" or "0*∞"  presubstitute  SaveIt() .
```

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 has been found helpful 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 $\pm\infty$ 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 can be NaNs for debugging, or zeros for compatibility with certain higher-level language conventions. Presubstituting instead of aborting can simplify beginnings and ends of loops, especially in matrix functions and in programs that search through data structures. On heavily pipelined machines, presubstitution allows compilers to overlap floating-point operations and anticipatory fetches of data without the risk that fetching data destined to be discarded might abort computation unnecessarily on an invalid fetch.

Retrospective Diagnostics:

Presubstitution brings to the fore a dilemma inherent in any policy for handling exceptions. On the one hand, a policy that forces execution to be aborted for every exception in some class (say Division-by-Zero) can be proved (by means lying beyond this paper's scope) to preclude successful searches under circumstances sometimes unavoidable. On the other hand, always to continue execution with, say, presubstitution for every exception is a risky policy too; programs transported from a machine that always stops to a machine that always continues may malfunction terribly.

Passing this dilemma down to applications programmers, forcing them to choose one policy or the other in subprograms that cannot "see the whole picture," is no resolution; they will choose to stop rather than be blamed for allowing their programs to continue on to a calamity.

A reasonable resolution of the dilemma comes from what I have called *Retrospective Diagnostics*. These take automatic note of the occurrence of at least one exception in each class, both by flags accessible to the program that raised the exception, and by other means addressed afterwards to the users of that program. Further discussion would lead beyond the intended scope of this paper; but the issue had to be mentioned first to warn readers that exceptions cannot be handled by presubstitution alone, and second to reassure readers that related problems are tractable.

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 was foreshadowed in exception handling provided for Fortran on IBM /360's as early as 1967 and documented currently in the VS Fortran Version 2 *Programming Guide* SC26-4222-3 and *Language and Library Reference* SC26-4221-3 (1988). See also ch. 2 of *Floating-Point Computation* by P. H. Sterbenz (1973), Prentice-Hall, New Jersey. Presubstitution has been implemented by David Barnett on a DEC VAX™ running 4.3 BSD Berkeley UNIX™, and on a SUN™ III; it works as advertised but has yet to be accepted as a standard component of UNIX . Presubstitution of a NaN for an array reference out of bounds occurs in BASIC on the Hewlett-Packard HP-71B handheld computer, which also has the flags and modes necessary to exploit its conformity with IEEE 854 though it does not provide more general presubstitution nor retrospective diagnostics; see the HP-71 Owner's and Reference Manuals 00071-90001 and -90010 (Oct. 1983).

Rudimentary retrospective diagnostics were first put into IBSYS 12 on an IBM 7094 at the Univ. of Toronto in 1963 (described in "7094 II System Support for Numerical Analysis," item C4537 in pages 1-54 of SHARE SSD 159, Dec. 1966), and on a Burroughs

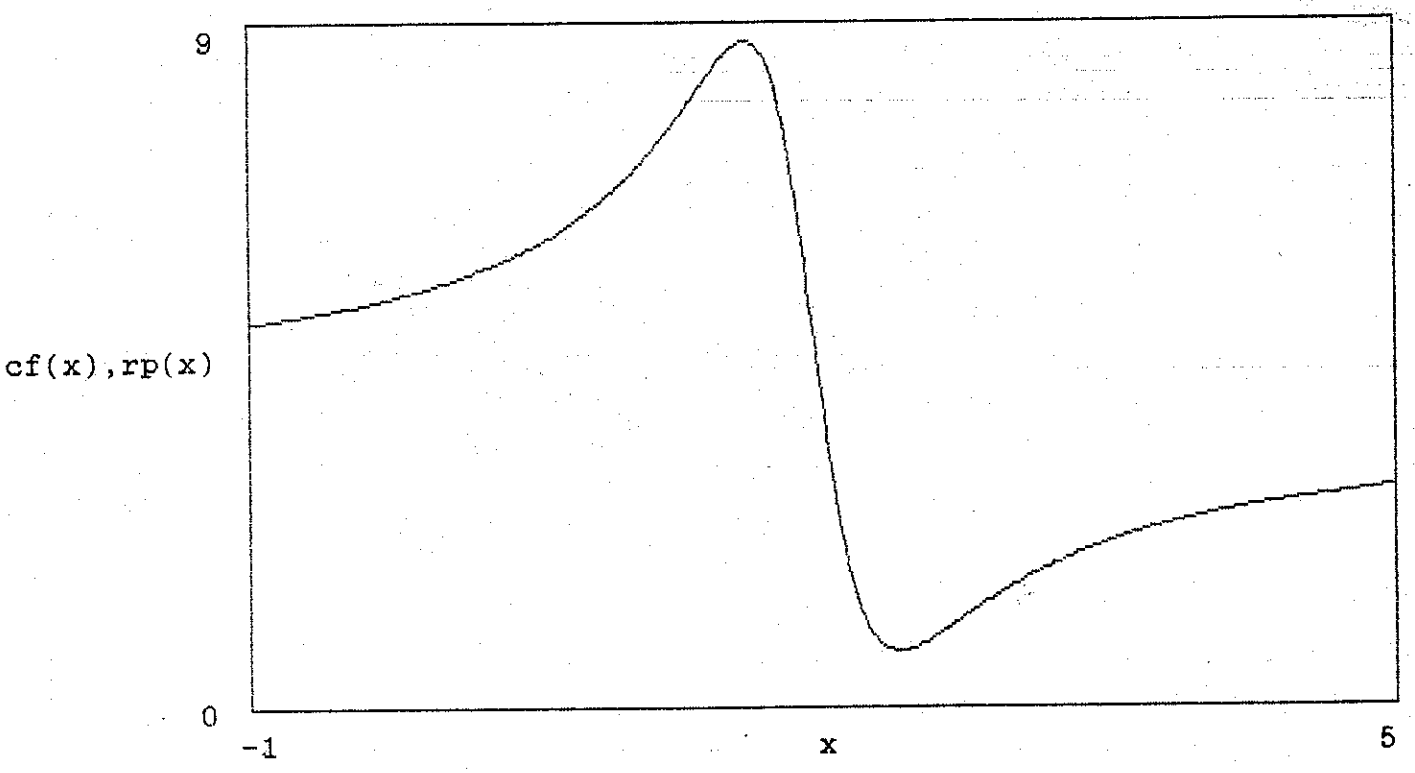
B5500 for the last few months of its stay at Stanford Univ. in 1966, and may be found in SUN's OS4 now.

x := -1,-.98 ..5 FIGURE 1 from MathCAD v. 2.03

Here are two ways to express the same rational function:

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

$$rp(x) := \frac{622 - x\cdot(751 - x\cdot(324 - x\cdot(59 - 4\cdot x)))}{112 - x\cdot(151 - x\cdot(72 - x\cdot(14 - x)))}$$



The coincidence of the graphs obtained by plotting both expressions confirms that they represent the same function, though they treat Roundoff, Overflow and Division-by-Zero differently.

For example, ...

cf(1) = ▪	cf(2) = ▪	cf(3) = ▪	cf(4) = ▪
singularity	singularity	singularity	singularity
rp(1) = 7	rp(2) = 4	rp(3) = 1.6	rp(4) = 2.5

Division-by-Zero cannot happen to rp(x) ; and it would be harmless in cf(x) too if the ∞ supplied by the hardware (it has an INTEL 80x87 that conforms to IEEE standard 754) were used as its designers intended. For instance, computing cf(3) would then produce correctly

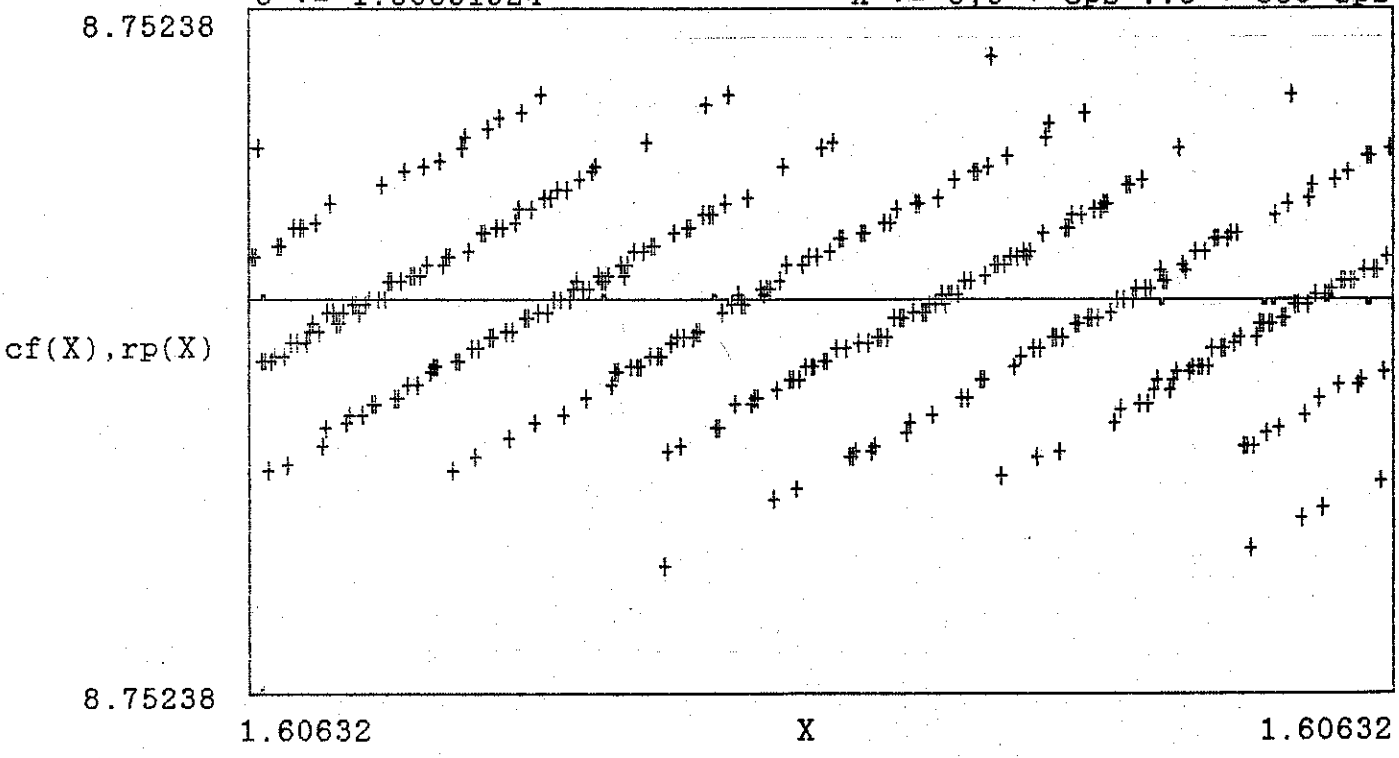
2/(x-3) = ∞ , x-2-∞ = -∞ , 10/∞ = 0 , x - 7 - 0 = -4 , ...

eps := 0.5

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

$$rp(x) := \frac{622 - x\cdot(751 - x\cdot(324 - x\cdot(59 - 4\cdot x)))}{112 - x\cdot(151 - x\cdot(72 - x\cdot(14 - x)))}$$

U := 1.60631924 X := U, U + eps .. U + 360·eps



The nearly smooth graph ---- belongs to cf(x) ; the ragged graph of + 's belongs to rp(x) . Every point on each graph has been plotted to show not only how much worse roundoff affects rp(x) than cf(x) but also that roundoff is not nearly so random as some people think.

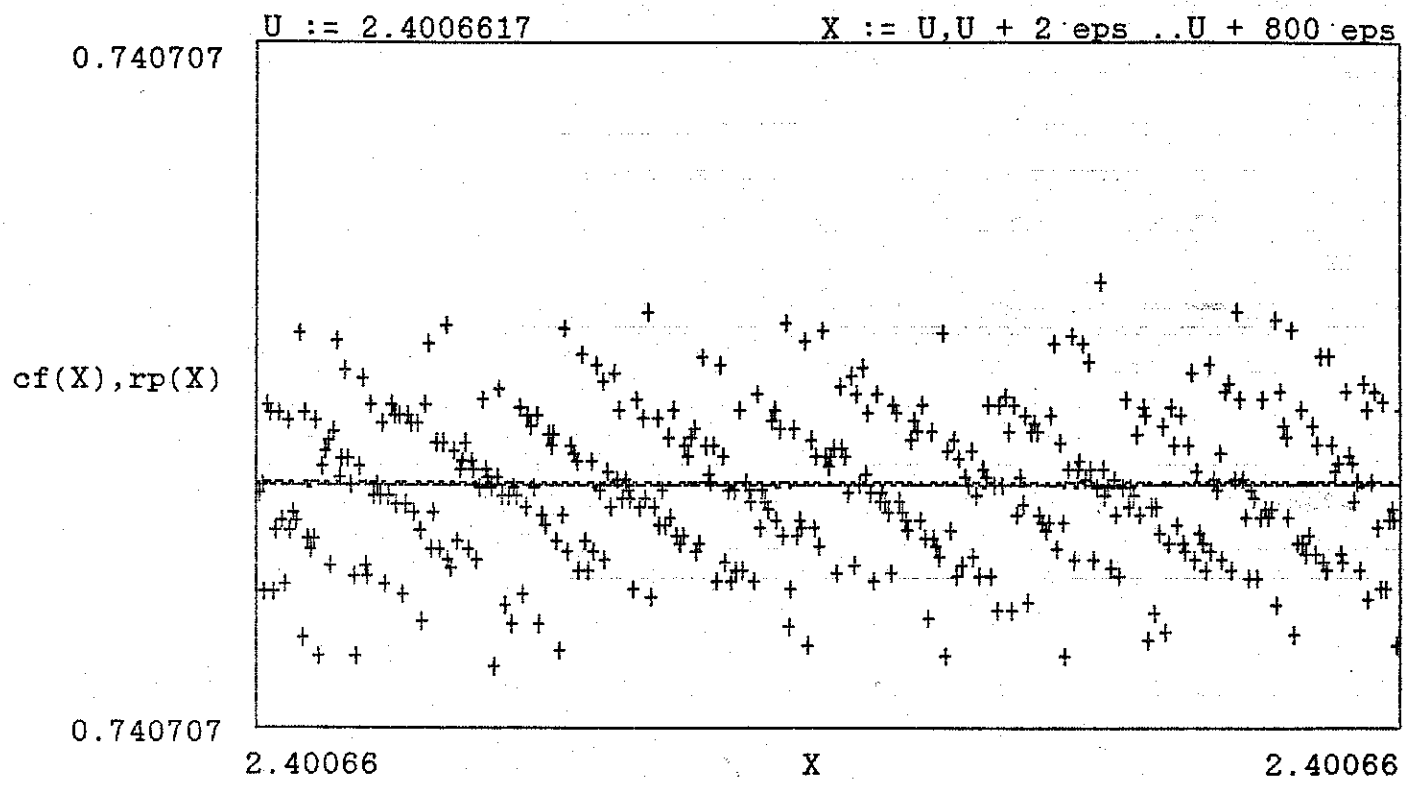
The next example illustrates that cf(x) is invulnerable to Overflow but rp(x) is not :

$cf\left[10^{77}\right] = 4$ correctly.

$rp\left[10^{77}\right] = \blacksquare\ \blacksquare$ overflow

FIGURE 3 from MathCad v. 2.03

The next graphs are included just to show that the previous one was not a fluke. They use different ranges of values for x .



$V := 2 + 240 \cdot eps$

$e(x) := cf(x) - rp(x)$

The next graph shows how roundoff obscures rp(x) , but not cf(x) , by about 24 times as much as that function changes when x changes by one unit in its last place for values x slightly bigger than 2 ; for most other values of x roundoff in rp is much worse than this.

$$d(x) := \frac{cf(x) - cf(V)}{10}$$