Computing 66, 309–320 (2001)

Computing



# A Remarkable Example of Catastrophic Cancellation Unraveled

**A. Cuyt, B. Verdonk, S. Becuwe,** and **P. Kuterna,** Antwerp

Received December 22, 1999; revised May 5, 2000

**Abstract**

In this paper we reinvestigate a well-known expression first published in [7], which is often used to illustrate catastrophic cancellation as well as the fact that identical output in different precisions does not imply reliability. The purpose of revisiting this expression is twofold. First, we show in Section 2 that the effect of the cancellation is very different on different IEEE 754 compliant platforms, and we unravel the underlying (hardware) reasons which are unknown to many numerical analysts. Besides illustrating cancellation, this expression also counters the common misbelief among many numerical analysts that a same program will deliver identical results on all IEEE conforming systems. Second, in Section 3 we use, illustrate and comment upon the cross-platform didactical tool Arithmetic Explorer developed at the University of Antwerp, by means of which we performed the bit level analysis of the expression evaluation under investigation on the different machines. We believe that this tool, which is freely available from the authors, can be of use to all of us teaching a first numerical analysis course.

*AMS Subject Classifications:* 65-04, 65G10, 65Y99.

*Key Words:* Computer arithmetic, IEEE floating-point, catastrophic cancellation.

## 1. Catastrophic Cancellation

Subtraction of nearby floating-point operands with equal signs can be a dangerous operation. Several examples hereof can be found in the literature and are often used when teaching the principles of rounded arithmetic in a numerical analysis class.

Let us denote by $\mathbb{F}(\beta, t, L, U)$ the set of normalized floating-point numbers $\pm d_0 \cdot d_1 \ldots d_{t-1} \times \beta^e$ where

$$\begin{aligned} 0 \leq & d_i \leq \beta - 1 \\ & d_0 \neq 0 \\ & L \leq e \leq U \end{aligned}$$

In order to be able to take care of overflow, the set is enlarged with the representations $\pm 1.0 \ldots 0 \times \beta^{U+1}$ for signed infinity. In order to take care of underflow the set is also enlarged with the denormalized numbers $\pm 0.d_1 \ldots d_{t-1} \times \beta^L$, where $\pm 0.0 \ldots 0 \times \beta^L$ represents signed zero. This enlarged set will be denoted by

$$\bar{\mathbb{F}} \equiv \bar{\mathbb{F}}(\beta, t, L, U)$$

Because the set of floating-point numbers is only a discrete approximation of the real number set, each arithmetic operation in $\bar{\mathbb{F}}$ usually involves a rounding error. Let us introduce the notation $x \circledast y$ for the computer arithmetic approximation of the mathematical operation $x * y$ where $* \in \{+, -, \times, \div\}$. IEEE-compliant arithmetic guarantees exactly rounded results, meaning that for $x, y \in \bar{\mathbb{F}}$

$$x \circledast y = \bigcirc(x * y)$$

where

$$\bigcirc : \bar{\mathbb{R}} \to \bar{\mathbb{F}} : x \to \bigcirc(x)$$

rounds $x$ to its nearest floating-point neighbour $\bigcirc(x)$. Other roundings besides the usual round-to-nearest are possible, but rounding to the nearest neighbouring float with break to even is an unbiased and monotone rounding. It is therefore the default. If $x * y$ is undefined in $\bar{\mathbb{R}}$ (e.g. $+\infty - (+\infty), 0/0, \ldots$), then the representation of $\bigcirc(x * y)$ is NaN (Not-a-Number).

In the case of cancellation during subtraction one has to distinguish between benign and catastrophic cancellation. The former occurs when two nearby, exact operands $x$ and $y$ are subtracted. For instance, with $\beta = 2$ and $t = 4$, the subtraction of $x = \bigcirc(x) = 1.000 \times 2^e$ and $y = \bigcirc(y) = 1.111 \times 2^{e-1}$ yields $z = x \ominus y = x - y = 1.000 \times 2^{e-4}$. The latter occurs when at least one of the operands is inexact. For example, if $x = \frac{1}{15}$, $y = \sin\left(\frac{1}{15}\right)$, $\beta = 10$ and $t = 10$, one obtains

$$\begin{aligned} x \approx \bigcirc(x) &= 6.666666667 \times 10^{-2} \\ y \approx \bigcirc(y) &= 6.661729492 \times 10^{-2} \\ \bigcirc(x) \ominus \bigcirc(y) &= 4.937175000 \times 10^{-5} \end{aligned}$$

Although the subtraction is exact, the last 3 digits in the approximation for $x - y$ are meaningless because they do not originate from reliable digits in $x$ and $\sin(x)$. They are only meaningful for $\bigcirc(x) - \bigcirc(y) = \bigcirc(x) \ominus \bigcirc(y)$.

## 2. A Remarkable Example

The following example was found in [7] where it was used to illustrate that it is not mathematically correct to conclude that a floating-point result is reliable if the same leading digits are delivered in single and double or in double and extended precision. For the expression given below this is the case, while in fact none of the leading digits in the output is correct. But a lot more can be told about this example, which is really a remarkable illustration of catastrophic cancellation.

When we want to compute, for $a = 77617$ and $b = 33096$, the expression

$$y = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b} \tag{1}$$

$$\stackrel{?}{=} 5.76461\ldots \times 10^{17}$$
$$\stackrel{?}{=} 6.33825\ldots \times 10^{29}$$
$$\stackrel{?}{=} 1.1726\ldots$$
$$\stackrel{?}{=} -0.827396\ldots$$

we obtain quite a lot of different results depending on the hardware platform on which we run our program or the precision used for the literals, variables and for the evaluation of the expression. Note that in a C-program all literals are, by default, converted to double precision at compile-time, while in FORTRAN constants with exponent letter `E` or without an exponent part are by default of the kind `KIND (0.0)`, meaning that they are converted to single precision [4]. Luckily, this is of no importance for the evaluation of $y$ since all literals in the expression for $y$ are exactly representable in the smallest basic format of single precision (`float` in C, `REAL` in FORTRAN), which corresponds to the floating-point set $\mathbb{F}(2, 24, -126, 127)$. For completeness we shall also discuss the output obtained using Matlab or using interval packages and compilers such as PROFIL/BIAS [6] and Sun Forte Fortran.

Let us have a look at what happens and why it is happening. In the sequel, we assume that the expression for $y$ is evaluated in one single compound statement, rewriting the integer powers of $a$ and $b$ as multiplications. In this way we only use the basic operations.

One distinguishes 4 different floating-point architectures, namely the extended-based (such as the Intel PC-family), the double-based (such as the IBM RISC machines), the single-double based (such as the Sun Sparc machines) and the single-double-extended based (such as VAX). It is therefore important that we analyze the effect of evaluating $y$, using on one hand architectures that support one single widest precision and on the other hand those that support all different precisions in hardware. We concentrate our detailed analysis on the architecture of the Intel PC-family and the Sun Sparc, although the test programs were also executed on other architectures, without adding different output to the list.

### *2.1. Intel*

When we perform our computations on an Intel personal computer equipped with a Pentium chip or with an older mathematical coprocessor, we have a choice between several ways to evaluate this expression. The Intel is an extended-based machine, meaning that its floating-point unit consists of extended precision registers $(\beta = 2, t = 64, L = -16382, U = 16383)$ and primarily produces extended format results. If one wants to perform certain computations in single or double precision, it is not sufficient to declare the variables and occurring literals accordingly. To mimic pure single and double precision (in the absence of overflow and underflow, which do not occur here), one also has to set the

rounding precision (alternatively called precision control) to 24, respectively 53 bits. During the evaluation of a single compound statement, intermediate results are then rounded to 24, respectively 53 bits precision, rather than to the default 64 bits. This of course has an effect on the final result for $y$. How to set and get the rounding precision is language and compiler dependent and is very often poorly documented. The results below were obtained using a Borland C++ compiler:

(2.1a) if the variables $a, b$ and $y$ and the occurring literals are declared single precision and if the default rounding precision of 64 bits is not changed, then one obtains for $y$ the value $5.76461\ldots \times 10^{17}$;

(2.1b) if the variables $a, b$ and $y$ and the occurring literals are declared double precision and if the default rounding precision of 64 bits is not changed, then one also obtains for $y$ the value $5.76461\ldots \times 10^{17}$;

(2.1c) if the variables $a, b$ and $y$ and the occurring literals are declared single precision and if the rounding precision is set to 24 bits, then one obtains for $y$ the approximation $6.33825\ldots \times 10^{29}$;

(2.1d) if the variables $a, b$ and $y$ and the occurring literals are declared double precision and if the rounding precision is set to 53 bits, then one obtains for $y$ the totally different value $1.1726\ldots$;

(2.1e) if the variables $a, b$ and $y$ and the occurring literals are declared extended precision and if the default rounding precision of 64 bits is not changed, then one also obtains for $y$ the value $5.76461\ldots \times 10^{17}$.

From the above results, one would think that the more 'precise' results are (2.1a), (2.1b) and (2.1e) because of the larger precision for the intermediate results and because of the fact that the outcome is the same. A naive programmer is tempted to decide that the result obtained in (2.1a), (2.1b) and (2.1e) is a reasonable approximation for $y$. He or she could not be more wrong! A suspicious programmer may turn to more powerful tools and repeat the experiment on a Sun Sparc workstation.

### *2.2. Sun*

The Sun Sparc is a single–double based architecture that supports single and double precision floating-point arithmetic in hardware and also offers a library for quadruple precision arithmetic, i.e. $\beta = 2$, $t = 113$, $L = -16382$, and $U = 16383$. When computing $y$ the following results were obtained using the Sun Forte C++ compiler:

(2.2a) if all literals and variables are declared single precision, then all intermediate results are also single precision and the value obtained for $y$ is $6.33825\ldots \times 10^{29}$;

(2.2b) if all literals and variables are declared double precision, then all intermediate results are also double precision and the value obtained for $y$ is $1.1726\ldots$;

(2.2c) the only possible attempt left is to use the quadruple 113 bit precision for literals and variables, which delivers for $y$ the approximation $1.1726\ldots$.

Now the single and double precision results don't exhibit the same leading digits but double and the much more precise quadruple do, so one is tempted to conclude that (2.2b) and (2.2c) provide a reasonable approximation for $y$. Wrong again! Not even the sign is correct.

### *2.3. Matlab, Computer Algebra Systems and Interval Computations*

When using Matlab Version 5.3.0 on either Intel Pentium or Sun Sparc, the outcome of the computation is always $1.1726\ldots$. The explanation of this phenomenon can be found in [5]: current versions of Matlab set the precision control bits of the Intel PC processors such that internally double precision is mimicked instead of using the full extended precision at their disposal. Hence the hardware architecture doesn't really make a difference anymore. This on one hand causes the build-up of larger rounding errors but on the other hand guarantees identical output on different platforms.

Maple V Release 5.1 on Intel always returns $-1.18059\ldots \times 10^{21}$ when using the underlying hardware floats. The result $-1.18059\ldots \times 10^{21}$ is also obtained in Matlab when programming the expression for $y$ using integer exponents instead of multiplications. So we assume Maple algebraically preprocesses our expression for $y$.

Mathematica Version 4.0.2, when computing with hardware floats on either Intel Pentium or Sun Sparc, returns one of $1.1726\ldots$ or $-1.18059\ldots \times 10^{21}$, depending on some small syntactic differences when inputting (1). Whether the integer coefficients in the expression for $y$ are entered "exactly" (without decimal point) or "inexactly" (with .0 added) actually makes a difference. In fact, only replacing 11 by 11.0 and leaving the other integer coefficients unchanged is sufficient to obtain one or the other result. Even if all integer coefficients in the expression are entered exactly, the result returned is different, depending on whether the values for $a$ and $b$ are entered as integers or with a decimal point. Apparently, small (irrelevant) syntactic changes have large semantic consequences.

The multiprecision arithmetic implemented in Mathematica and Maple is not very useful for the evaluation of $y$, because the working precision must be specified by the user and this naturally implies some guess work. At least one significant digit is returned for precisions starting at 37 decimal digits. Before that, both systems' results are different, unpredictable and unreliable.

The double precision interval output for the expression under consideration is

$$[-8.264\ldots \times 10^{21}, 7.0835\ldots \times 10^{21}]$$

using either PROFIL/BIAS (with the gcc-compiler version 2.95.1), the Fortran compiler (option `-xia=strict`) from Sun Forte or the Arithmetic Explorer tool

discussed in section 3. This clearly signals that something is wrong and that one should worry about the reliability of the output!

### 2.4. Mathematical Analysis

Although all the computed results mentioned above are precise, they are not at all accurate. It is not difficult to find out what causes this confusion. Why exactly things go so wrong on the different platforms is explained in the next section. When we rewrite $y$ as

$$z = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) \tag{2}$$

$$x = 5.5b^8 \tag{3}$$

$$y = z + x + \frac{a}{2b}$$

then

$$z = -79171113406689613611011347015249428 50$$
$$x = +79171113406689613611011347015249428 48$$
$$z + x = -2$$

In other words, $z$ and $x$ have 35 out of 37 decimal digits in common. Consequently, whenever the precision is not large enough, rounding errors will contaminate the last few bits in the floating-point representation of $z$ and $x$. This contamination can be such that either $z \oplus x = 0$, in which case $y \approx \frac{a}{2b} = 1.1726\ldots$, or $z$ and $x$ differ in the last few bits of their floating-point representation, in which case $y \approx z \oplus x$. One needs at least 122 bits precision to get $z + x$ right ($2^{122} \approx 10^{36}$) and the significant digits in the final result $y$ then follow automatically. Actually

$$y = -\frac{54767}{66192} \approx -0.827396\ldots$$

### 2.5. Bit Level Analysis

Why do both architectures exhibit a similar behaviour while delivering totally different results, namely respectively $5.76461\ldots \times 10^{17}$ and $1.1726\ldots$, misleading the programmer in both cases?

Let us first consider the Intel platform. The fact that this platform is extended-based has implications for expressions evaluated in one single compound statement. Let us denote by

$$E(x_1, \ldots, x_n; c_1, \ldots, c_n)$$

a mathematical expression involving the basic operations $+, -, \times, \div,$ rem and $\sqrt{}$, the variables $x_1, \ldots, x_n$ and the literals $c_1, \ldots, c_n$. As we have seen in Section 2.1, a mathematical expression $E$ can be evaluated in many ways on an Intel platform, dependent on the rounding precision and the precision of the variables and literals. We shall therefore denote by

$$E_{t,m}^{(I)} \qquad t \in \{24, 53, 64\}, \quad m \in \{s, d, e\}$$

the floating-point value of the mathematical expression $E$ when evaluated in a single statement on an Intel platform with the rounding precision set to $t$ and with $m$ the precision used for the variables and literals. Note that, since we assume absence of underflow and overflow during the evaluation of $E$, the exponent range has no influence on the evaluation and hence only a precision-dependent notation is introduced. The rounding precision can be either 24 for single, 53 for double or 64 for the default extended precision. The possible values for $m$ are $s, d$ or $e$ to indicate that the variables and literals are declared single precision as in (2.1a) and (2.1c), double precision as in (2.1b) and (2.1d), or extended precision as in (2.1e). Clearly, one should choose the rounding precision $t$ at least as large as the precision of the variables and constants, meaning that in $E_{t,s}^{(I)}$, $t \geq 24$ and in $E_{t,d}^{(I)}$, $t \geq 53$. It is now quite easy to see that

$$E_{64,s}^{(I)} = E_{64,d}^{(I)} = E_{64,e}^{(I)} \Leftrightarrow \begin{cases} x_1, \ldots, x_n \in \mathbb{F}(2, 24, -126, 127) \\ c_1, \ldots, c_n \in \mathbb{F}(2, 24, -126, 127) \end{cases}$$

Since these conditions are fulfilled for the expression $y$ given by (1), this explains why (2.1a), (2.1b) and (2.1e) return the same value on the Intel platform. In fact, in all three cases the value of $y$ is evaluated in precisely the same way, with intermediate results rounded to 64 bits precision. The fact that variables and literals are declared single or double precision may mislead the numerical analyst into thinking that the evaluation of the expression is performed in single, respectively double precision. This is only the case if also the rounding precision is set to 24, respectively 53, as in (2.1c) and (2.1d).

In contrast to the Intel personal computer platform, the Sun Sparc platforms are single–double based architectures. This implies that during the evaluation of an expression involving only single precision variables and constants, all intermediate results are rounded to 24 bits. If an expression involves only double precision operands, then all intermediate results are rounded to 53 bits precision. This is also the case when single and double precision operands are mixed, since then the principle of type promotion comes into play as soon as a double precision operand is involved. For a Sun platform, the floating-point value of a mathematical expression $E$ is therefore only dependent on the precision of the variables and literals in the expression. We therefore introduce

$$E_m^{(S)} \quad m \in \{s, d, q\}$$

to denote the floating-point value of the mathematical expression $E$ when evaluated on a Sun platform with $m$ the precision used for variables and literals. The possible values for $m$ are $s$ for single precision as in (2.2a), $d$ for double precision as in (2.2b) and $q$ for quadruple precision as in (2.2c).

Taking these architectural aspects into consideration, we can explain why, when $E = y$ as given by (1), then $y_{24,s}^{(I)}$ on Intel is equal to on Sun and $y_s^{(S)}$, while $y_{53,d}^{(I)}$ is equal to $y_d^{(S)}$.

It still does not explain, however, why $y_s^{(S)}$ and $y_d^{(S)}$ are so different, nor why $y_d^{(S)}$ equals $y_q^{(S)}$, the result obtained on Sun in quadruple precision. If we look at what happens at bit level, then the effect of finite precision rounded arithmetic on this expression can be further detailed.

Let us look at the machine representations for $z$ and $x$, respectively defined by (2) and (3) and obtained on the different platforms and in different precisions. One can check that [3] when rounding in double and quadruple precision

$$\begin{aligned} z_d^{(S)} &= -x_d^{(S)} \\ z_q^{(S)} &= -x_q^{(S)} \end{aligned}$$

and hence

$$\begin{aligned} y_d^{(S)} &= 0.5 \otimes a \oslash b \approx 1.1726\ldots \\ y_q^{(S)} &= 0.5 \otimes a \oslash b \approx 1.1726\ldots \end{aligned}$$

which explains why $y_d^{(S)}$ and $y_q^{(S)}$ have the same leading digits. However, due to the catastrophic cancellation in $z_d^{(S)} + x_d^{(S)}$ and in $z_q^{(S)} + x_q^{(S)}$, none of these leading digits is correct. On the other hand, when rounding in single precision, one can check that

$$\begin{aligned} z_s^{(S)} &= -1.011\,1110\,1001\,1000\,1111\,0111 \times 2^{122} \\ x_s^{(S)} &= +1.011\,1110\,1001\,1000\,1111\,1000 \times 2^{122} \\ z_s^{(S)} + x_s^{(S)} &= 1 \times 2^{122-23} \\ &= 2^{99} \end{aligned}$$

Since

$$2^{99} = \frac{1}{2}2^{100} = \frac{1}{2} \times 1024^{10} \approx 6.3 \times 10^{29}$$

one can understand why

$$y_s^{(S)} = z_s^{(S)} \oplus x_s^{(S)} \oplus 0.5 \otimes a \oslash b \approx 6.3 \times 10^{29}$$

On the Intel platform a similar situation occurs when computing in extended precision. One can again check that

$$
\begin{aligned}
&z_{64,e}^{(I)} \\
&\quad = -1.011\,1110\,1001\,1000\,1111\,0111\,0011\,1100\,1110\,0101\,0010\,0010\,0101\,1011\,0010\,0001 \times 2^{122} \\
&x_{64,e}^{(I)} \\
&\quad = +1.011\,1110\,1001\,1000\,1111\,0111\,0011\,1100\,1110\,0101\,0010\,0010\,0101\,1011\,0010\,0010 \times 2^{122} \\
&z_{64,e}^{(I)} + x_{64,e}^{(I)} = 1 \times 2^{122-63} \\
&\qquad\qquad = 2^{59}
\end{aligned}
$$

Hence

$$
\begin{aligned}
y_{64,e}^{(I)} = 2^{59} \oplus 0.5 \otimes a \oslash b &= \frac{1}{2} \times 1024^6 \oplus 0.5 \otimes a \oslash b \\
&\approx \frac{1}{2} 10^{18} \oplus 0.5 \otimes a \oslash b \approx 5.0 \times 10^{17}
\end{aligned}
$$

Both in single and extended precision, the rounding error in computing $z$ and $x$ is such that $z \oplus x$ becomes the dominant term in the expression $y$, yielding a totally erroneous final result for $y$.

## 3. Arithmetic Explorer

The mathematical and bit level analysis in the respective sections 2.4 and 2.5 were performed using the tool Arithmetic Explorer developed at the University of Antwerp (UIA). For expression (1), the exact result was computed as in Fig. 1 while the effect of catastrophic cancellation is illustrated in Fig. 2.

This tool was initially developed for didactical purposes, in the framework of the course Computer Arithmetic and Numerical Techniques [3]. It is now being further extended into a full-fledged, performant and powerful arithmetic environment called Arithmos.

In the Arithmetic Explorer, it is possible to specify that computations be performed, not by default in the hardware IEEE singles or doubles, but if desired in a user-defined set of floating-point numbers. By choosing a small precision $t$ and a limited exponent range, students can better follow the computations at the bit level. Moreover, in a low-precision floating-point set, one can more easily zoom in on the unmistakable effects of data and rounding error, cancellation, ill-conditioning and numerical instability, and one can develop a better feeling for computer arithmetic issues.

We have taken care to implement the tools' floating-point arithmetic in full compliance with the philosophy of the IEEE 754 and 854 standards for floating-point arithmetic [1, 2]. Except that users can freely specify the precision and exponent range, all aspects of the IEEE standards are implemented and can be

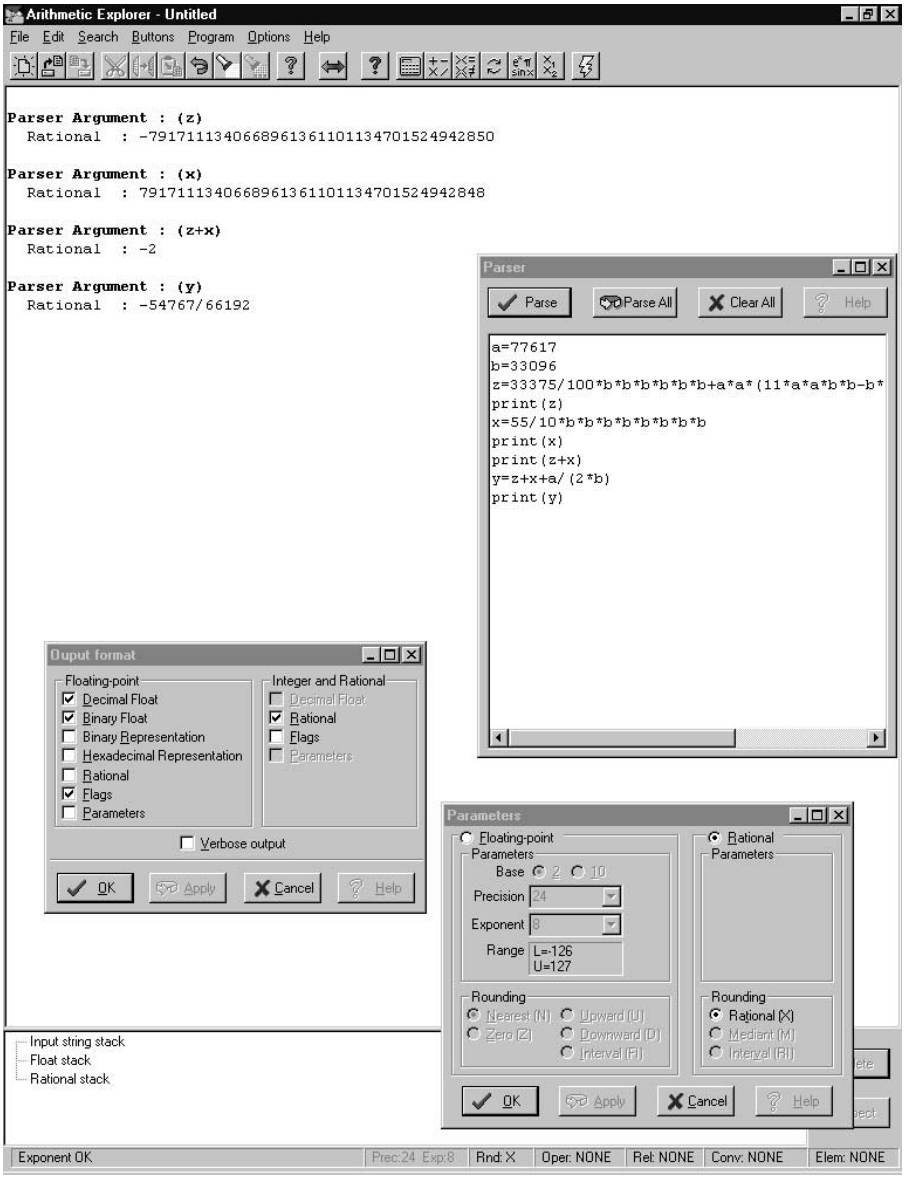


**Figure 1.** Exact result

visualized, including exact rounding, denormals, signed zeroes and infinities, Not-a-Numbers and exception flags. In this way, students can discover the intriguing details of floating-point arithmetic. While a similar analysis can be done with other tools, such as Mathematica or by direct programming using a traditional compiler, the result is often a time-consuming and confusing task, obscuring more points than one is trying to make.

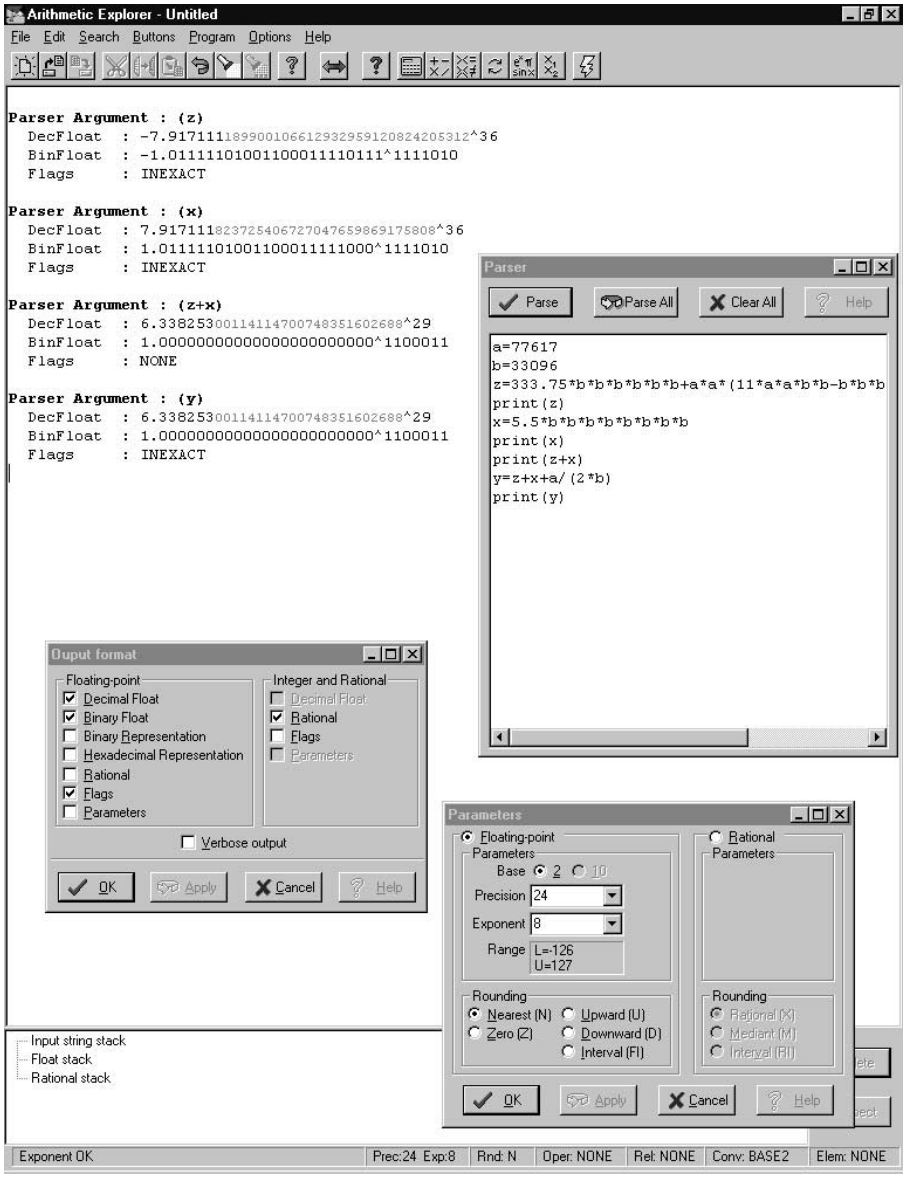


**Figure 2.** Effect of catastrophic calculation

The implementation of Arithmetic Explorer was realized via C++ classes. Full use of operator overloading enables a user, in addition to using the environment to evaluate simple expressions, to run C/C++ code from within the Arithmetic Explorer, with a user-specified precision and exponent range, rather than with the IEEE single or double precision hardware floats.

Except for floating-point arithmetic, the current version of Arithmetic Explorer also supports rational arithmetic and interval arithmetic (where the intervals have endpoints in the user-defined set of floating-point numbers) as well as all type conversions. In a next release, the underlying floating-point implementation (developed for didactical purposes) will be replaced by a fast IEEE compliant multi-precision floating-point library, for use with large precisions and exponent ranges. This implementation will be the basis for sharp multi-precision floating-point interval arithmetic in Arithmos v1.0. In the near future, interval arithmetic with rational endpoints will also be included. Future plans for Arithmos v2.0 include polynomial arithmetic (with polynomial coefficients of all basic types), complex arithmetic, reliable graphics and hybrid expression evaluation. More information on the Arithmetic Explorer and Arithmos can be found at http://win-www.uia.ac.be/u/cant/ or can be obtained from the authors.

## References

[1] ANSI/IEEE Std 754-1985. IEEE standard for binary floating-point arithmetic. ACM SIGPLAN *22*, 9–25 (1987).
[2] ANSI/IEEE Std 854-1987. IEEE standard for radix-independent floating-point arithmetic. New York, 1987.
[3] Cuyt, A., Verdonk, B.: Computational science and engineering at Belgian universities – Scientific computing for the CS student. IEEE Comput. Sci. Eng. *4*, 79–83 (1997).
[4] ISO/IEC. International Standard: Information Technology – Programming Languages – Fortran. 1991. Reference number ISO/IES 1539, 1991.
[5] Kahan, W.: Matlab's loss is nobody's gain. Technical report, UCB, 1998.
[6] Knüppel, O.: PROFIL/BIAS – a fast interval library. Computing *53*, 277–287 (1994).
[7] Rump, S. M.: Algorithms for verified inclusions – theory and practice. In: Reliability in Computing (Moore, R. E., ed.), pp. 109–126. New York: Academic Press, 1988.

Annie Cuyt
Research Director FWO-Vlaanderen
Department of Mathematics
and Computer Science
Universiteit Antwerpen (UIA)
Universiteitsplein 1
B-2610 Antwerp (Wilrijk)
Belgium
e-mail: cuyt@uia.ua.ac.be

Brigitte Verdonk
Postdoctoral Fellow FWO-Vlaanderen
Department of Mathematics
and Computer Science
Universiteit Antwerpen (UIA)
Universiteitsplein 1
B-2610 Antwerp (Wilrijk)
Belgium
e-mail: verdonk@uia.ua.ac.be

Stefan Becuwe
Research Assistant
Department of Mathematics
and Computer Science
Universiteit Antwerpen (UIA)
Universiteitsplein 1
B-2610 Antwerp (Wilrijk)
Belgium
e-mail: sbecuwe@uia.ua.ac.be

Peter Kuterna
Research Student
Department of Mathematics
and Computer Science
Universiteit Antwerpen (UIA)
Universiteitsplein 1
B-2610 Antwerp (Wilrijk)
Belgium
e-mail: kuterna@uia.ua.ac.be