THE CLASS LIBRARY FOR EXACT RATIONAL ARITHMETIC IN ARIΘMOΣ

A. Cuyt [*] P. Kuterna B. Verdonk [†] J. Vervloet

Dept Mathematics and Computer Science
University of Antwerp (UIA)
Universiteitsplein 1, B–2610 Antwerpen, Belgium

Tel: +32 3 8202401, Fax: +32 3 820 24 21
Email: {cuyt, kuterna, verdonk, jvvloet}@uia.ua.ac.be
Homepage: win-www.uia.ac.be/u/cant/

Abstract

ARIΘMOΣ consists of a family of class libraries (fully IEEE compliant multiprecision floating-point, sharp multiprecision floating-point interval and exact rational arithmetic in V1.0, floating-slash, rational interval and complex arithmetic in V2.0) that are available at programming language level as well as through a GUI with parser. In this paper we discuss the rational class library from ARIΘMOΣ. Besides offering additional functionality when compared to other rational arithmetic software, the implementation is fully cross-platform (no assembler), object-oriented (operator overloading) and currently the most performant rational library available (on 64-bit platforms even outperforming well-known C libraries).

1 Introduction

In the past decade the interest in alternative implementations of computer arithmetic, besides the well-known IEEE 754 hardware floating-point arithmetic, has grown tremendously. Among others we mention in particular interval arithmetic, multiprecision arithmetic and rational arithmetic.

Rational arithmetic can mostly be found in computer algebra systems, where the user has access to the exact arithmetic via a GUI simulating the usual pencil and paper style of calculations. Of course these interpreters are much less performant than some C- or C++-libraries that offer a similar functionality and operate on an abstract data type. The most popular of these libraries [8, 13, 23, 9] will be discussed in Section 2.

Interval arithmetic [16, 12, 11, 10] has been very successful at solving some hard problems and is now even available directly through a Sun f90 compiler. One of the main drawbacks of this type of automatic error analysis and result verification is the

[*]Research Director FWO–Vlaanderen
[†]Postdoctoral Researcher FWO–Vlaanderen

fact that the interval enclosures tend to grow very fast, especially when calculated naively. Hence the success of interval arithmetic ultimately lies in the combination with multiprecision arithmetic. This is in sharp contrast with most interval implementations that are around and make use of the underlying hardware.

As far as multiprecision arithmetic is concerned, several packages can be found [3, 8], even for extremely large precisions [19, 2], but none of them adheres to the principles of the IEEE 754 floating-point standard that exists for the smaller standard precisions, such as exact rounding of the basic operations, special representations for signed zeroes and infinities, denormals and exception handling. Hence the packages cannot offer the same reliability and portability and certainly do not allow for an interval implementation. Moreover, the big number arithmetic offered in for instance Mathematica follows an obscure formal model that is largely undocumented.

In short, for the numerical programmer who wants to make use of several of the alternatives above, there is no single platform that offers the best of all worlds. The new rational class library that we describe in this paper must be seen in that context. It accompanies a recently developed and very performant multiprecision class library, that fully complies with the principles of the IEEE 754-854 standards for floating-point arithmetic [21]. Offering all the required rounding modes, the latter allows to build a sharp interval library on top. Besides the rational class library described here in Section 3, future work will include mediant rounding in rational arithmetic and the implementation of rational interval arithmetic. Complex and polynomial arithmetic built on each of the basic number types are also planned.

We have chosen to keep the implementation fully cross-platform and not include any platform-specific optimizations. Notwithstanding this choice, the rational class library in ARIΘMOΣ is the fastest library currently available, as will be indicated in Section 4. A simple GUI with parser for the quick evaluation of expressions, complements the set of class libraries and is presented in Section 5. ARIΘMOΣ aims at offering a fast, fully portable, object-oriented alternative for numerical computing, inbetween the computer algebra systems on one hand and the plain C-libraries on the other hand.

2 Overview and functionality of existing implementations

In this section we briefly describe some of the more popular implementations offering rational arithmetic. Detailed information on the functionality offered by each implementation, as well as on the added functionality offered by the new implementation in ARIΘMOΣ V1.0, is given in the next section. It is typical for most implementations that can be found, that little or no communication exists between the supported rational number type and other number types such as multiprecision or hardware floats, hardware integers, complex numbers, intervals etc. As explained before, this is precisely one of the goals of the ARIΘMOΣ project.

2.1 Mathematica

The computer algebra system Mathematica is available for all major operating systems. The core of Mathematica is built into a kernel which can be accessed by means of a GUI. Mathematica is a powerful and easy-to-use environment which is being applied to a wide range of sectors.

Mathematica offers a wide range of functions, algorithms and graphics as well as a programming language. Its greatest drawback however is its performance.

2.2 GNU MP

The GNU Multiprecision arithmetic library [8] is a portable C-library for arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers. GNU MP supports different implementations for each of the operations, depending on the size of the operands in order to achieve a reasonably good performance in all cases. Furthermore it uses optimized assembly code on many different CPU's for the most common inner loops of the algorithms.

For rational arithmetic this library offers initialization and assignment of operands, the basic arithmetic and relational operators, and functions operating on the numerator and denominator of a rational operand.

2.3 SIMATH

SIMATH [23], which stands for SInix-MATHematik, is essentially a computer algebra system focusing mainly on algebraic number theory. SIMATH is known to run on most 32-bit UNIX platforms. The arithmetic library in SIMATH contains algorithms to perform computations over the integers $\mathbb{I}$ (representing the mathematical set $\mathbb{Z}$), the rational numbers $\mathbb{Q}$, (fixed precision) multiprecision floating-point numbers, the finite rings $\mathbb{I}/m\mathbb{I}$, the finite Galois fields, the p-adic number fields, algebraic number fields and function fields. For all these domains, basic arithmetic as well as higher algorithms are available.

The SIMATH-library can be accessed in two ways. Either by calling the SIMATH functions from a SIMATH-program, for which the SIMATH-preprocessor generates a corresponding C-program, or by using the interactive calculator Simcalc, which features many of the SIMATH algorithms, comprehensive error checking, and detailed help facilities.

2.4 MIRACL

MIRACL stands for Multiprecision Integer and Rational Arithmetic C/C++ Library [17, 18]. MIRACL is in fact a C-library, but a C++-wrapper is provided. Optimized assembly code is used on certain processors for the most time-critical routines, but a portable C-version of these routines is still available. The developers state that it is currently the fastest library available on an 80x86 or Pentium platform.

MIRACL uses irreducible floating-slash numbers to approximate data and results. For fixed k, each signed floating-slash number occupies $k + \lceil \log_2 k \rceil + 1$ bits of memory to store the sign, an unsigned i-bit integer numerator and an unsigned j-bit integer denominator with $i + j = k$, the position of the slash separating numerator and denominator, and the status of the fraction which can be exact or approximate. In what follows we will denote the set of floating-slash numbers by $\mathbb{Q}_k$.

For multiprecision integer arithmetic MIRACL includes all the primitives that are necessary for public key cryptography. On the rational side MIRACL supports functions for initialization and assignment of rational numbers, as well as the basic arithmetic and relational operators. Besides this, MIRACL also supports the power function, the square root, the extraction of the numerator and denominator, and elementary functions like sine, cosine, tangent.

2.5 CLN

CLN stands for Class Library for Numbers [9] and is a portable C++-library. CLN is capable of representing integers, rational numbers, floating-point numbers, complex numbers, modular integers and univariate polynomials. CLN claims to be memory efficient by using immediate allocation for small integers and short floats and has an automatic, non-interruptive garbage collection for heap-allocated memory. Furthermore CLN claims to be speed efficient by using assembly language in the kernel for some processors, low-level routines from GNU MP, the Karatsuba multiplication and the Schönhage-Strassen multiplication for very large numbers.

For rational arithmetic the following operations are supported: initialization and assignment, the basic arithmetic and relational operators, the square, the exponential, the absolute value and the sign of an operand. Four rounding functions of an operand x are available: the floor function delivering the largest integer smaller than or equal to x, the ceiling function delivering the smallest integer greater than or equal to x, the trunc function for the nearest integer to x among 0 and x (inclusive), and the round function for the nearest integer to x with break to even. The square root and the n-th root are also supported but a result is only delivered when the root is exact.

2.6 libg++

The implementation libg++ is a library for the GCC-compiler [13] with support for integer and rational arithmetic. As of version 2.8.1 the library is an add-on, because the required portion of libstdc++ is separated from libg++. The libray is still available, but enhancements to it should not be expected.

2.7 Overview

From the above it should be clear that a fully portable, object-oriented yet performant set of class libraries, offering several number representations and complemented with a simple GUI to offer the look and feel of writing mathematical formulas, would fill a void.

	Mathematica	GNU MP	SIMATH	MIRACL	CLN	libg++	ARIΘMOΣ
Implementation and number types							
C		√	√	√			
C++				√	√	√	√
multiprecision IEEE compliant							√
rational	√	√	√	√	√	√	√
multiprecision interval							√
complex			√		√		
GUI	√						√

3 Functionality offered in the new rational class library

In the next sections we discuss one by one the extra functionality that is available in the new implementation. The fact that the new class library is part of an integrated computational system, motivates for the following new features:

- exactly rounded conversions from and to hardware and multiprecision floats (see Section 3.5) and hence more constructors;
- special values like signed infinities, non-integer and non-rational values (see Sections 3.3 and 3.7);
- the unordered relation to compare with non-integer and non-rational values (see Section 3.4);
- the power function for rational operands (see Section 3.6);
- a rational control word and status word (see Sections 3.8 and 3.9), among other things for exception handling (see Section 3.10) and to prepare for rational interval arithmetic.

3.1 Notations

The number sets that are available to the user in the ARIΘΜΟΣ V1.0 programming environment, and that hence appear in our discussion besides the set of rational numbers, are the following:

- The sets of binary hardware floats $\mathbb{F}_s$, $\mathbb{F}_d$ and $\mathbb{F}_e$, where s, d and e refer to the IEEE 754 single, double and double extended precision respectively [1]. Of course a single extended precision inbetween single and double precision can also be included, but such a fourth precision is rare.
- The set of multiprecision software floats with base β and precision t, denoted by $\mathbb{F}_t^{(\beta)}$. Usually β is 2^k or 10^ℓ and in case $\beta = 2$, the superscript will be omitted if no confusion can arise. The set $\mathbb{F}_t^{(\beta)}$ is also dependent on a range $[L, U]$ for the exponent which we shall usually not specify explicitly.
- The sets of decimal floating-point numbers $\mathbb{F}_s^{(10)}, \mathbb{F}_d^{(10)}$ and $\mathbb{F}_e^{(10)}$ where the subscripts s, d and e now respectively refer to the single, double and double extended precisions specified in the IEEE 854 radix-independent standard.
- The sets of signed hardware integers $\mathbb{I}_{16}$, $\mathbb{I}_{32}$ and $\mathbb{I}_{64}$ where the subscript denotes the number of bits provided to represent the integer and its sign in one or other notation.
- The set of signed big integers denoted by $\mathbb{I}_\infty$. Here the subscript ∞ indicates that as much memory as needed can be allocated to represent the integer, as long as the limit on machine memory is not exceeded. In the sequel this subscript will usually be omitted.
- The set of rational numbers $\mathbb{Q}_\infty$ that can be represented in machine memory. In the sequel, we shall often omit the subscript ∞ too. We assume, without loss of generality, that the rational numbers are represented as a tuple of irreducible big integers in $\mathbb{I}_\infty$ and that the denominator is positive.
- The set $\mathbb{Q}_k$ of floating-slash rational numbers [14]. The elements of $\mathbb{Q}_k$ are the signed irreducible fractions with an unsigned $i-$bit integer numerator and an unsigned $j-$bit integer denominator where $i+j \leq k$. Each element of $\mathbb{Q}_k$ occupies $k + \lceil \log_2 k \rceil + 1$ bits of memory in which are stored the unsigned numerator and denominator, the position of the slash separating numerator and denominator, the sign of the fraction and the status of the fraction which can be exact or approximate.

Each of the sets listed so far is an exactly representable finite subset of the set of real numbers $\mathbb{R}$. The graph given in Figure 1, is the analogue for machine numbers of the inclusion relation between the mathematical sets $\mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R}$. In contrast with these mathematical sets however, we remark that there is only a partial order relation between the different sets of machine numbers.

$$\begin{tikzcd}
 & \mathbb{F}_s^{(10)} \arrow[rr] & & \mathbb{F}_d^{(10)} \arrow[r] & \mathbb{F}_e^{(10)} \arrow[r, "t>e"] & \mathbb{F}_t^{(10)} \arrow[drr] & & \mathbb{I} \arrow[d] & \\
\mathbb{I}_{16} \arrow[ur] \arrow[rr] \arrow[dr] & & \mathbb{I}_{32} \arrow[ur] \arrow[dr] \arrow[rrr, "m>32"] & & & \mathbb{I}_m \arrow[u, "t\geq\lceil\log_{10} 2^{m-1}\rceil"] \arrow[urr] \arrow[r, "k\geq m-1"] \arrow[d, "t\geq m-1"] & \mathbb{Q}_k \arrow[r] & \mathbb{Q} \arrow[r] & \mathbb{R} \\
 & \mathbb{F}_s \arrow[rr] & & \mathbb{F}_d \arrow[r] & \mathbb{F}_e \arrow[r, "t>e"] & \mathbb{F}_t \arrow[urr] & & &
\end{tikzcd}$$

Figure 1: Inclusion relation for the participating subsets of $\mathbb{R}$

Each node in this directed acyclic graph represents one of the number sets listed above. Each arrow corresponds to the strict inclusion relation $\subset$. If an arrow is accompanied by a condition, the inclusion relation only holds in case the condition holds. Note that the inclusion of $\mathbb{I}_{16}$ in $\mathbb{F}_s^{(10)}$ and of $\mathbb{I}_{32}$ in $\mathbb{F}_d^{(10)}$ is guaranteed by the requirement in the IEEE 854 standard that the single precision decimal floating-point format has a significand of at least 6 decimal digits wide and the double precision decimal floating-point format one of at least 13.

It is well-known that the operations discussed in the next sections are not necessarily closed in the number sets listed above. Yet when adding for instance two single precision floating-point numbers, everybody is expecting a single precision result as return value, not a real number. Hence the notions of rounding and machine operation have to be introduced and are indispensable.

A rounding $\bigcirc_C : \mathbb{R} \longrightarrow C$ that associates with each real value its representation in the set $C \subset \mathbb{R}$, is called an exact rounding if it is monotone and exact for the elements of C, meaning that

$$\forall x \in C : \bigcirc_C(x) = x \tag{1}$$

$$\forall x, y, z \in \mathbb{R} : x \leq y \leq z \Rightarrow \bigcirc_C(x) \leq \bigcirc_C(y) \leq \bigcirc_C(z) \tag{2}$$

The machine operation $\circledast_C$ denotes the operation $*$ applied to two elements of a machine number set C and yielding an element of C as result. Hence the operation $\circledast_C$ is closed in C. Here $*$ can be any one of $+, -, \times, /, \mathrm{rem}, \surd$ or the supported conversions. Clearly we want $x \circledast_C y$ to be as accurate as possible. The principle of exactly rounded machine operation achieves this. The adoption of this principle for floating-point computation in the IEEE 754 standard is one of the standard's key achievements: it states that, if C is any of $\mathbb{F}_s, \mathbb{F}_d$ or $\mathbb{F}_e$ then

$$\circledast_C := \bigcirc_C \circ * \tag{3}$$

3.2 Constructing a rational number

A rational object can be constructed in several ways. The default constructor creates a rational object which is initialized with the value zero. All the other constructors give the programmer the possibility to initialize the rational object with a value which is passed on to the constructor via parameters.

When two parameters are passed to the constructor, the first parameter stands for the numerator and the second parameter stands for the denominator. In this case the parameters can be of the big integer type or the hardware integer type. A hardware double, a string representing a fraction, or a multiprecision float can also be passed to the constructor for creation of a rational object.

	Mathematica	GNU MP	SIMATH	MIRACL	CLN	libg++	ARIΘΜΟΣ
Construction of a rational							
from a hardware integer			√	√	√	√	√
from a hardware long integer		√	√	√	√	√	√
from a multiprecision integer	√		√	√	√	√	√
from a hardware float							√
from a hardware double		√		√		√	√
from a multiprecision floating-point	√				√		√
from a string	√		√	√	√		√

3.3 Basic operations and square root

Exact rounding of the basic operations in $\mathbb{Q}$ is easy because these binary operations are closed in $\mathbb{Q}$. The overview shows that each implementation offers the four basic operations but not necessarily the square root.

	Mathematica	GNU MP	SIMATH	MIRACL	CLN	libg++	ARIΘΜΟΣ
Basic operations and Square root							
addition	√	√	√	√	√	√	√
subtraction	√	√	√	√	√	√	√
multiplication	√	√	√	√	√	√	√
division	√	√	√	√	√	√	√
square root	√			√	√		√

In a floating-point context, the implementation of the square root is required to be exactly rounded. This is achieved by defining the machine implementation of the square root for $C \neq \mathbb{Q}$ as

$$\textcircled{\surd}_C := \bigcirc_C \circ \sqrt{\ } \tag{4}$$

The above definition cannot be applied when $C = \mathbb{Q}$, because when for instance $q \in \mathbb{Q}$ is such that $\sqrt{q} \in \mathbb{R} \setminus \mathbb{Q}$, then the exactly rounded value of $\sqrt{q}$ in $\mathbb{Q}$ is mathematically not well-defined.

In order to define $\textcircled{\surd}_{\mathbb{Q}}$, we observe that for other number sets such as for example $\mathbb{F}_t$, the definition of $\textcircled{\surd}_{\mathbb{F}_t}$ is such that if $\bigcirc_{\mathbb{F}_t}(\sqrt{x})$ is mathematically well-defined in $\mathbb{F}_t$, then $\textcircled{\surd}_{\mathbb{F}_t}$ returns this value. If, on the other hand, this value is mathematically not well-defined, which can only occur if $x < 0$, then $\textcircled{\surd}_{\mathbb{F}_t}$ returns NaN (Not a Number). Applying a similar philosophy to $C = \mathbb{Q}$, the following definition of $\textcircled{\surd}_{\mathbb{Q}}$ seems appropriate if it is essential that all computations in $\mathbb{Q}_\infty$ be implemented exactly:

$$\textcircled{\surd}_{\mathbb{Q}}(q) := \begin{cases} \sqrt{q} & \text{if } \sqrt{q} \in \mathbb{Q} \\ +\text{IaR (positive Is a Real)} & \text{if } \sqrt{q} \notin \mathbb{Q}, q > 0 \\ \text{NaN (Not a Number)} & \text{if } q < 0 \end{cases} \tag{5}$$

Indeed, in all cases where $\sqrt{q} \notin \mathbb{Q}$, the value $\bigcirc_{\mathbb{Q}}(\sqrt{q})$ is not well-defined. The fundamental reason to return a special value such as IaR or ±IaR (and not return an undefined NaN or abort the program) comes from the fact that we found it useful to return as much information on the result as possible, in case a representation cannot be given. The underlying reason being that in subsequent computations such special values can sometimes vanish while mathematically undefined results persist. Consider for instance

$$\begin{aligned}(+\text{IaR}) \otimes (-\infty) &= -\infty & (6)\\ \gcd(1, \lfloor +\text{IaQ} \rfloor) &= 1 & (7)\end{aligned}$$

Generally speaking, returning IaD or ±IaD (where D can be Z for integers, Q for rationals and R for reals) for the operation $\circledast_C$, means that the result of the operation $*$ cannot be represented in the set C (taking the specifications in the control word into account) but does exist in the mathematical set $D \supset C$. Here the set D is the smallest of $\mathbb{Z}, \mathbb{Q}, \mathbb{R}$.

It is clear, however, that a definition such as (5) is not very informative for the user. Rather than returning a signed or unsigned Ia{Z, Q, R} in many cases, a more useful implementation of the square root can be given in interval arithmetic (let us denote the set of intervals with rational endpoints by $I\mathbb{Q}$):

$$\oslash_{I\mathbb{Q}}(q) := \begin{cases} \{\sqrt{q}\} & \text{if } \sqrt{q} \in \mathbb{Q} \\ [q_1, q_2] \quad q_1, q_2 \in \mathbb{Q}_k & \text{if } \sqrt{q} \notin \mathbb{Q}, q > 0, \sqrt{q} \in [q_1, q_2] \\ \text{NaN (Not a Number)} & \text{if } q < 0 \end{cases} \tag{8}$$

As announced in the introduction, this last implementation will be the subject of future work. It is clear that at any time during the computation, the programmer must be able to specify which of the two definitions (5) or (8) should be applied, in the same way as it must be possible to specify the rounding mode at any time during the computation. How this can be achieved in a uniform manner will be detailed in Section 3.9.

3.4 Relations

The supported relational operators are $<, \leq, >, \geq, =, \neq$, and unordered. The last one is used when at least one operand is undefined or unspecified because it is either NaN, IaR, IaQ or IaZ. Every undefined or unspecified value compares unordered with everything, including itself. The unordered relation is not available in any of the other implementations.

	Mathematica	GNU MP	SIMATH	MIRACL	CLN	libg++	ARIΘΜΟΣ
Relational operators							
$<, >$	✓	✓	✓	✓	✓	✓	✓
$=$	✓	✓	✓	✓	✓	✓	✓
$\leq, \geq$	✓	✓	✓	✓	✓	✓	✓
$\neq$	✓	✓	✓	✓	✓	✓	✓
unordered							✓

3.5 Conversions

With hardware integers and floating-point numbers of different precisions available in rounded arithmetic, and big integers and rationals available in exact arithmetic, a lot of conversions are possible. We only want to focus in somewhat more detail on the conversions between $\mathbb{F}_t^{(\beta)}$ and $\mathbb{Q}$.

Since $\mathbb{F}_t^{(\beta)} \subset \mathbb{Q}$, it follows that the exactly rounded conversion of a floating-point number to a rational number (not to a floating-slash number with fixed k), always equals the number itself. For example, if f is the single precision hardware representation (24 binary digits) of the decimal constant 0.1, then:

$$\begin{aligned} f &= 1.100\,1100\,1100\,1100\,1100\,1101 \times 2^{-4} \\ &= 13421773/134217728 = q \end{aligned}$$

is its exact conversion to a rational value. It is however well-known that the fraction q usually has rather large numerator and denominator, as in the above example. For this reason, a non-exact conversion from $\mathbb{F}_t^{(\beta)}$ to $\mathbb{Q}$, which delivers simpler approximating fractions, is recommendable in a computing environment supporting many number types. Such a conversion could map every element $f \in \mathbb{F}_t^{(\beta)}$ to a floating-slash number in $\mathbb{Q}_k$, for some small value k independent of f and t. The value k could even be a parameter of the conversion. Both the exact and approximate conversions are, for example, already supported in LISP and will be included in a next release of ARIΘMOΣ.

The conversion from $\mathbb{Q}$ to $\mathbb{F}_t^{(\beta)}$ in the current release is done using the exactly rounded division in $\mathbb{F}_t^{(\beta)}$ with t being the precision necessary to represent numerator and denominator of the rational operand exactly, without any additional rounding.

	Mathematica	GNU MP	SIMATH	MIRACL	CLN	libg++	ARIΘMOΣ
Conversions							
to a hardware integer					√		√
to a hardware long integer					√		√
to a multiprecision integer	√				√	√	√
to a hardware float					√		√
to a hardware double				√	√	√	√
to a multiprecision floating-point	√				√		√
to a string	√		√	√			√

3.6 Miscellaneous functions

The sign, the numerator and the denominator of a rational object can be extracted. The floor, the ceiling, the absolute value and the inverse of a rational object can be calculated. The power function of two rational objects is also supported. The special value IaZ is for instance returned when computing $\lfloor \text{IaQ} \rfloor$ where IaQ can be the result of a division of operands from $\mathbb{I}$.

	Mathematica	GNU MP	SIMATH	MIRACL	CLN	libg++	ARIΘΜΟΣ
Miscellaneous							
extracting sign	√	√	√	√	√	√	√
extracting numerator / denominator	√	√	√	√	√	√	√
floor and ceiling	√		√		√	√	√
absolute value	√		√	√	√	√	√
inverse		√	√			√	√
n^{th} power	√		√	√	√	√	√
n^{th} root	√			√	√		√

3.7 Special values

In the same way as the IEEE 754 and 854 standards would not have been complete without a description of all exceptional situations and the introduction of the necessary exceptional values (such as $\pm\infty$, NaN and ± 0), the ARIΘΜΟΣ programming environment is not complete without a listing of the exceptional cases that can occur and the actions that need to be taken.

A lot has been said and written about the representation of 0. Depending on the number set one considers, three different representations for 0 are in use: the exact mathematical or unsigned zero and the signed zeroes $+0$ and -0. In exact arithmetic, when computing in $\mathbb{I}_\infty$ and $\mathbb{Q}_\infty$, only the exact value of zero needs to be represented.

In inexact or rounded arithmetic, one not only needs to represent the exact zero value but, much more often, $\bigcirc(\pm\epsilon)$, with ϵ too small to be itself represented as a machine number. Floating-point implementations that follow the principles of the IEEE standards use ± 0 to represent $\bigcirc(\pm\epsilon)$. The values ± 0 are then identified with the exact zero value. In the set of floating-slash rational numbers, the situation is slightly different since computations in $\mathbb{Q}_k$ can yield both exact rational as well as approximate rational values. Hence an implementation of arithmetic in $\mathbb{Q}_k$ should support all three representations of 0, as is the case in [14]. The inexact, signed representation of zero is needed in case of underflow, when ϵ is too small to be represented in $\mathbb{Q}_k$. The exact representation of zero is needed to represent the result of $x \ominus x = x - x$ when x itself is exact. Since in the future the three values of zero have to coexist in our system environment, the rule $+0 = -0$ of rounded arithmetic should be extended for conversion purposes to $+0 = -0 = 0$. Nevertheless, the unsigned representation of zero has a stronger mathematical meaning than the signed representation of 0.

The need to represent $\pm\infty$ is clear when overflow can occur, as is the case in $\mathbb{F}_t^{(\beta)}$, $\mathbb{Q}_k$ and also in the sets of hardware integers $\mathbb{I}_t$, $t \neq \infty$, although in the latter sets overflow is ignored in current hardware implementations and usually a negative, meaningless integer number is returned. While overflow cannot occur in $\mathbb{I}_\infty$ and $\mathbb{Q}_\infty$, providing a representation for $\pm\infty$ in these sets is necessary, for example when the argument of a conversion to $\mathbb{I}_\infty$ or $\mathbb{Q}_\infty$ is itself $\pm\infty$. Furthermore, when moving to the interval plane and considering intervals with exact integer or rational endpoints, one is able to represent a halfline only if the representations for $\pm\infty$ are provided.

Last but not least, the introduction of Not-a-Number is thoroughly motivated in [7] for floating-point arithmetic. Many of these arguments also apply here. The need to also introduce $\pm$IaZ (Is an Integer), $\pm$IaQ (Is a Rational) and IaR (Is a Real) has already been touched upon in Section 3.2 and will become apparent as we discuss exceptions in Section 3.10. Several rules for the propagation of Ia{Z, Q, R} and NaN (which should actually be split into Is-a-Complex and Not-a-Number) can be formulated [22]. Since this discussion is a subject in its own, it lies beyond the scope of this paper.

Besides supporting the representation of these special values for exceptional results, a rational object can also be initialized with any of these special values and the special values can be checked for.

	Mathematica	GNU MP	SIMATH	MIRACL	CLN	libg++	ARIΘΜΟΣ
Special value support							
setting and testing infinities	$\surd$						$\surd$
setting and testing NaN's, Ia{Z, Q, R}'s							$\surd$
signed zeroes				$\surd$			
exception handling (Section 3.10)							$\surd$

3.8 Status words

The notion of status word was introduced in the framework of the IEEE standardization in order to keep track of exceptions in floating-point arithmetic. The five exceptions that, according to IEEE 754 and IEEE 854, should be signaled when detected, are (i) Zero divide, (ii) Overflow, (iii) Underflow, (iv) Inexact result and (v) Invalid operation.

Each flag in the status word of IEEE compliant hardware signals the exception it corresponds to, independent of whether that exception occurs in $\mathbb{F}_s$, $\mathbb{F}_d$ or $\mathbb{F}_e$. In a hybrid computing environment like ARIΘΜΟΣ, encompassing all the sets in Figure 1, providing only one status word would imply a severe loss of information. It makes it for example impossible to retrace whether the exception has occurred during the computations in exact or rounded arithmetic. Moreover, in each of the computer arithmetic implementations from Figure 1, the set of exceptions that can arise is different. Whether a separate status word should be provided for each arithmetic implementation (multiprecision software floating-point, hardware floating-point, exact rational, floating-slash rational, hardware integer, variable precision software integer, ...) or whether a status word could be commonly used by some of the arithmetic implementations, is not a clear cut issue. The implementation under discussion here has provided some insight in this matter and the conclusion is that at least two status words are required: one for exact integer and rational arithmetic (in $\mathbb{I}_\infty$ and in $\mathbb{Q}_\infty$) and one for rounded arithmetic (in $\mathbb{F}_t^{(\beta)}$ and $\mathbb{Q}_k$), with a separate status word for floating-point and for floating-slash rational arithmetic as a valid option in the next release. The exceptions arising in rational arithmetic are further discussed in Section 3.10 while those from floating-point arithmetic were thoroughly discussed in [1]. A status word for the latter is provided in software in ARIΘΜΟΣ for the sake of the multiprecision computations.

The only sets that preclude a faithful representation of the special values in their format are the hardware integers. Hence the use of a hardware integer type for numerical results is not advisable. The parser available via the GUI of ARIΘΜΟΣ therefore works in rounded arithmetic only with integers faithfully represented in floats, and in exact arithmetic only with so-called big integers.

In any case, when an operation $\circledast_C$ generates an exception, this shall be signaled in the status word corresponding to the set C in which the operation is performed.

This causes no ambiguity, even when operands of different types are involved in the operation, when the principle of type promotion is applied. In the same way, for conversions that raise an exception, it is the status word of the destination's format that should be set. Thus, exceptions arising in big integer arithmetic can be signaled in the corresponding status word. This is more desirable than the situation where exceptions arising from floating-point to hardware integer conversions, are signaled in the floating-point status word because it is impossible otherwise.

3.9 Control words

In parallel with status flags to signal exceptions, an implementation should provide control bits for trap handling and rounding specification, if applicable. We restrict ourselves here to the rounding specifications, and the result specification in trap-disabled mode for all exceptional situations.

In the same way as we have introduced different status words to signal exceptions occurring in different number sets, it is appropriate that in a hybrid system also different control words be supported. In line with the above, we provide at least two control words: one for exact arithmetic in $\mathbb{I}_\infty$ and in $\mathbb{Q}_\infty$ and one for rounded arithmetic in $\mathbb{F}_t^{(\beta)}$ and $\mathbb{Q}_k$, with a separate control word for floating-point and for floating-slash rational arithmetic as a recommendable option for the future.

Note that, in a control word for floating-point arithmetic only four out of the five exception control bits are meaningful as trap enabling/disabling bits. Indeed, enabling the inexact trap handler in floating-point arithmetic is not really an option, because the system would trap on almost every floating-point operation. We therefore use the inexact control bit in both control words not as a trap enable/disable bit, but in a 'diluted' way. It should always be set in floating-point arithmetic, to indicate that inexact operations are 'allowed', while clearing or setting the inexact bit in the control word for exact arithmetic will indicate whether only exact (in $\mathbb{Q}$) or also approximate (in $\mathbb{Q}_k$) results are allowed. This use of the inexact control bit would then solve the problem we have indicated in Section 3.3 concerning the two implementations of the square root operation in $\mathbb{Q}_\infty$ and in $\mathbb{I}_\infty$. If the inexact bit in the control word corresponding to $\mathbb{Q}_\infty$ is set, definition (8) will be used, else (5) will be used.

This usage of the inexact control bit raises two new issues. First, when the bit is set and approximate results are allowed, a rounding mode needs to be specified. Therefore a control word should also provide bits to let the user specify the active rounding mode. Here one needs to distinguish between rounding in $\mathbb{I}_\infty$ on one hand and rounding in $\mathbb{Q}_\infty$ on the other. In the former case, one of the four well-known rounding modes, to nearest, up, down and to zero, can be applied. In the latter case these four rounding modes have to be interpreted in $\mathbb{Q}_k$ as mediant, up, down and to zero. When rounding to elements in $\mathbb{Q}_k$ additional precision control bits are needed to specify the granularity of the set $\mathbb{Q}_k$. These can be compared to the precision control bits for temporary results available on the INTEL PC hardware.

Second, whereas operations in $\mathbb{F}_t^{(\beta)}$ are almost always inexact, this is not necessarily the case for operations in $\mathbb{Q}_k$. A programmer may want to disable inexact operations in $\mathbb{Q}_k$ while enabling inexact operations in $\mathbb{F}_t^{(\beta)}$. This is only possible if two inexact control bits are available in two different control words, one for floating-point and one for floating-slash rational arithmetic.

In Section 3.10 we describe in more detail the exceptions that can occur in exact arithmetic and the result that should be returned when an exceptional situation arises. As already mentioned, we restrict ourselves to trap-disabled mode for all exceptions.

Zero divide: If the divisor is (unsigned) zero and the dividend is a finite nonzero number, then the 'zero divide' exception shall be signaled. The result shall be NaN, since division of a nonzero number by an exact zero is mathematically undefined.

Overflow in $\mathbb{I}_t, t \neq \infty$: that the inexact control bit is set (inexact operations are allowed), this exception shall be signaled when the destination's format largest finite number is exceeded in magnitude by what would have been the (rounded) integer result, were t unbounded. Depending on the rounding mode and the sign of the intermediate result, the result shall be $\pm\infty$ or the largest/most negative finite number in $\mathbb{I}_t$.

Inexact result: Under the condition that the inexact control bit is set, this exception shall be signaled if the rounded result of an operation is inexact or if it overflows. It is clear that inexact without overflow can only occur for the operations that are not closed in $\mathbb{Z}$ (in casu / and $\sqrt{}$) and in $\mathbb{Q}$ (in casu $\sqrt{}$), as well as certain conversions to $\mathbb{I}_t$, $\mathbb{I}_\infty$ or $\mathbb{Q}_\infty$. In all cases the rounded or overflowed result shall be delivered.

Invalid operation: Whenever the result of an operation is mathematically undefined, the invalid exception should be signaled and a NaN returned. This is the case in all arithmetic number representations. Moreover, in integer and exact rational arithmetic, we propose to also raise the invalid operation whenever the inexact control bit is cleared (no inexact operations allowed) and the result of the operation overflows or cannot be delivered exactly. In that case the result returned is the proper special value Ia{Z,Q,R}, possibly signed.

Table 1: Exception handling in $\mathbb{I}_t$, $\mathbb{I}_\infty$ and $\mathbb{Q}_\infty$

3.10 Exception handling

In this section we discuss the exceptions that can occur in the number sets $\mathbb{I}_t$ (where usually $t = 16, 32$ or 64), in $\mathbb{I}_\infty$ and in $\mathbb{Q}_\infty$, in other words in integer and rational arithmetic. These exceptions are: (i) Zero divide, (ii) Overflow (only for $\mathbb{I}_t$), (iii) Memory overflow (only for $\mathbb{I}_\infty$ and $\mathbb{Q}_\infty$), (iv) Inexact result and (v) Invalid operation.

As pointed out in 3.8, the exceptional situations arising in $\mathbb{I}_t$ cannot be intercepted and hence one has to limit oneself to the discussion of exceptions in $\mathbb{I}_\infty$ and $\mathbb{Q}_\infty$.

Except for overflow (which can only occur in $\mathbb{I}_t$) and memory overflow (which can only occur in $\mathbb{I}_\infty$ and $\mathbb{Q}_\infty$), the exceptions (i–v) can arise in all three number sets. Furthermore, these sets have in common that they need and provide only the unsigned representation for zero. Hence we have grouped the status and control bits for $\mathbb{I}_\infty$ and $\mathbb{Q}_\infty$ in a single status, respectively control word.

Table 1 describes in detail the conditions for the raising of each arithmetic exception in exact integer and rational arithmetic, and the result returned in that case. When any of these exceptions is detected, the corresponding status flag is set and the specified result returned. In case of Memory overflow, the system halts.

4 Implementation of the class library

The rational class library is based on a big integer class library for storing and manipulating integer numbers. As will be indicated in 4.1, the rational class library in ARIΘΜΟΣ is currently the most performant implementation of rational arithmetic, when compared to the libraries described in Section 2. In addition, as described in Section 3, it offers several new features such as exception handling and support for special representations. This functionality is indispensable from the point of view of a hybrid computational system.

4.1 Performance

One must take great care in designing the core of the C++-library to benefit from the advantages of C++ and OO design, without incurring too much overhead.

Such overhead can occur every time a function returns a value, since this involves constructing an object of the return value type. This (temporary) object is passed to the calling routine and is typically destroyed at the end of the statement in which the function was called. In fact, the lifetime of such an object is typically shorter than that of an object explicitly declared in the calling routine.

Consider for example the expression `A = B + C`. The subexpression `B + C` returns a temporary object which is then used as the operand for the assignment operator. The assignment operator places a verbatim copy of its operand into the target operand, possibly returning a reference of the target object. The temporary object is then destroyed.

The overhead incurred by the creation of a temporary object can be considerable: allocation of the required memory, initialization of the object, copying of the information before the temporary is destroyed and deallocation of memory at the time of destruction. This is especially the case if the objects that need to be constructed are large integers or rationals.

To avoid this considerable overhead, the principle of 'copy-by-assumption' [6] can be applied whereby at every function call the memory allocated to a temporary object can be reused. The data of a temporary object is transferred to the target object in three steps. First, any data that is currently pointed to by the target object is released. Second, the data pointer in the temporary object is copied to the data pointer in the target object. And third, the data pointer in the temporary object is set to the null pointer.

Nevertheless, copy-by-assumption will never be as performant as 'bare' C programming, because when evaluating `C = A + B`, the previous data of `C` has to be freed before it will be replaced by the new result. The C-style expression `sum (C, A, B)` allows the `sum` function to reuse the memory which might already have been allocated for `C`. Keeping this in mind, C-style expressions were used to generate intermediate results for the implementation of the classes.

When the memory needed for these intermediate results can be allocated on the stack instead of the heap, expensive heap memory management will be avoided. This also improves the computation time, but possible stack overflow must be taken into account.

Use of the above in the design of our classes for rational arithmetic, results in a significantly better performance than a straightforward C++ implementation of the same algorithms. In the Figures 2 and 3, the performance of our new 'all C++ without assembler' library is compared to the other libraries.

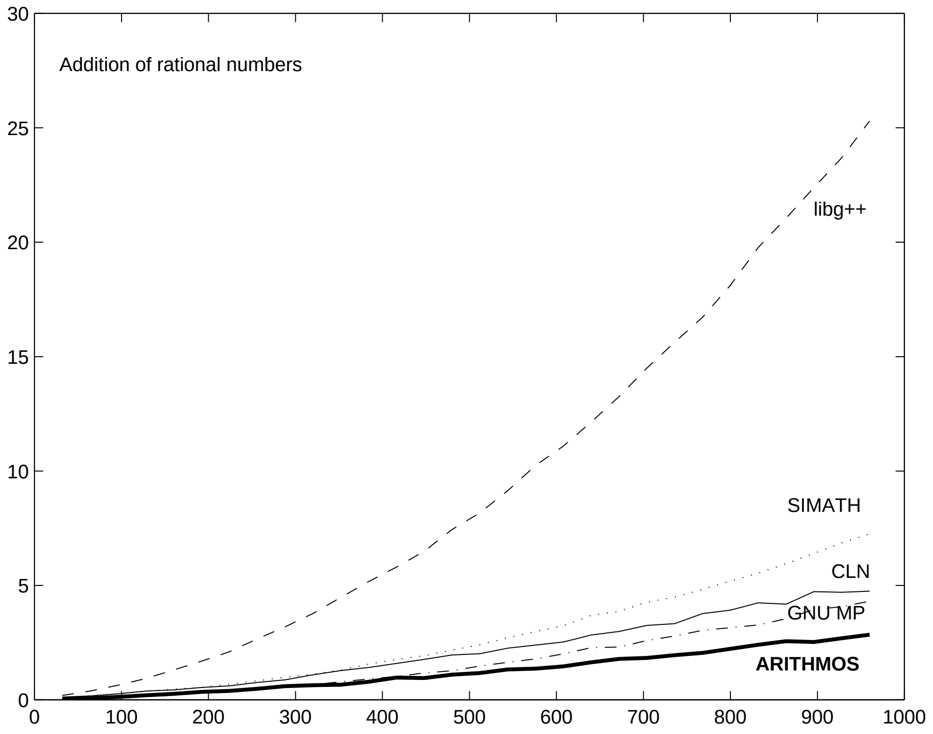


Figure 2: Addition of rationals

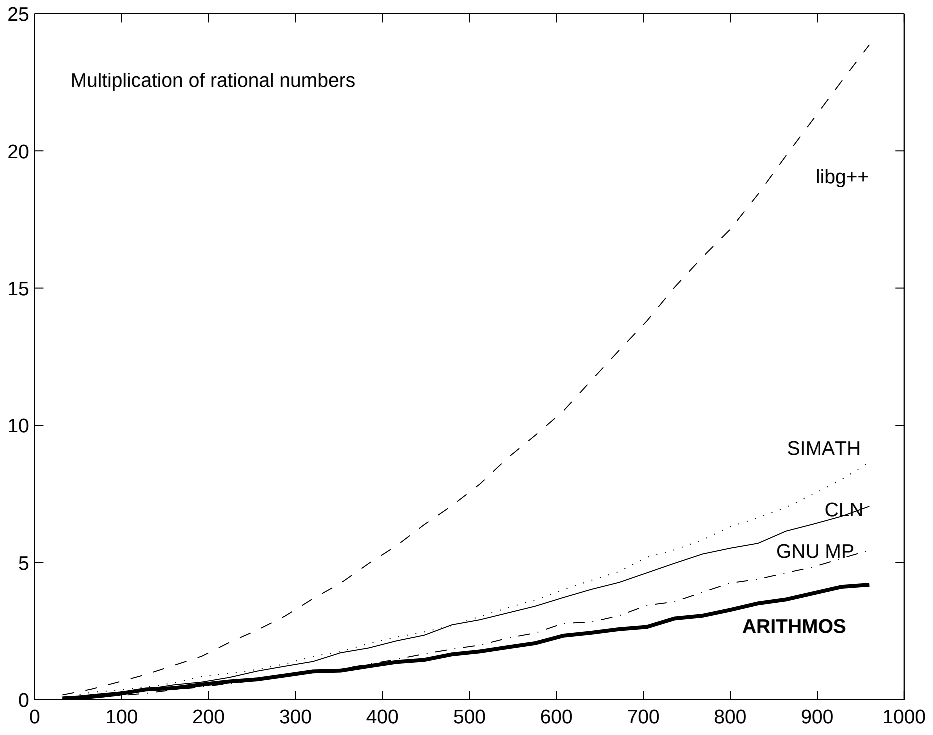


Figure 3: Multiplication of rationals

As can be seen, the rational library of ARIΘΜΟΣ is the most performant of all implementations discussed in Section 2. It outperforms both libg++ as well as the speed efficient rational library CLN, two C++ implementations of rational arithmetic.

The latter is remarkable, especially if one takes into account that the performance of CLN is boosted by assembly language code in the kernel. Furthermore, our object-oriented implementation performs better than GNU MP, which is the fastest C library for rational arithmetic.

All performance tests were carried out on a Sun 5.6 platform equipped with two Ultrasparc 167 Mhz processors. For each integer s between 32 to 960, fourty rationals were generated with random s-bit numerator and denominator. Addition and multiplication was performed for all 1600 combinations of these 40 rationals and the CPU time was measured in seconds.

4.2 Class structure

The most important class in the library is the `bigint` class; it is used to represent (almost) arbitrarily large integers. The most interesting class members of `bigint` are listed below :

```
class bigint
{
 protected:
  unsigned long size;
  unsigned long datasize;
  signed char sign;
  unsigned char special;
  atom* data;

 /* member functions and static members are omitted */
}
```

An `atom` is just an integer, which can be 32-bit, 16-bit or 8-bit. This is automatically determined at compile time, depending on the availability of respectively 64-bit, 32-bit or 16-bit integers on the platform.

The `special` byte is used to represent special values, which are described in section 3.7 while `sign` indicates whether the integer is strictly positive (`sign = 1`), strictly negative (`sign = -1`) or unsigned (`sign = 0`), which is the case for zero and some special values.

The magnitude of a `bigint` number is given by

$$\sum_{i=0}^{\texttt{size}-1} \texttt{data}[i] \cdot 2^{i\cdot(\#\text{bits per atom})}$$

Finally, the member `datasize` indicates how many `atom`s are allocated for the array `data`. When a result becomes too big to fit in `datasize` atoms, the array will be resized. Since `datasize` is a `long` integer, a `bigint` has to be smaller than

$$2^{(\#\text{bits per atom})\cdot(2^{\texttt{LONG\_BITS}}-1)}$$

where `LONG_BITS` is the number of bits in a `long` integer (usually 32).

Two classes inherit from `bigint` : `tbigint` and `stackbigint`. The `tbigint` class represents a temporary large integer, which typically appears as return value of a function. When a `tbigint` is used as the right hand side in an assignment, its value will disappear, as can be read in the previous section. When one writes a function which has a big integer as return value, it is recommended to let this value be a `tbigint`.

A `stackbigint` is a bigint which has its `data` on the call stack of the program. This avoids expensive memory management, but does not allow the integer to 'grow'. Since this class is designed for internal use only, the user should not bother about it.

Finally the `rational` class is a record, consisting of two `bigints` (numerator and positive denominator) with no common non-trivial divisors. Similar to the above, a `trational` is a temporary instance of the `rational` class.

4.3 Cross-platform interface

The reliable arithmetic library was developed while keeping two other goals in mind: speed and availability on a large range of platforms. Therefore the library was developed in ANSI C/C++ and makes use of the latest techniques to reduce the C++-overhead. No assembly code was used to boost the performance, but great care has been taken when choosing the data structure and the algorithms that work on this structure.

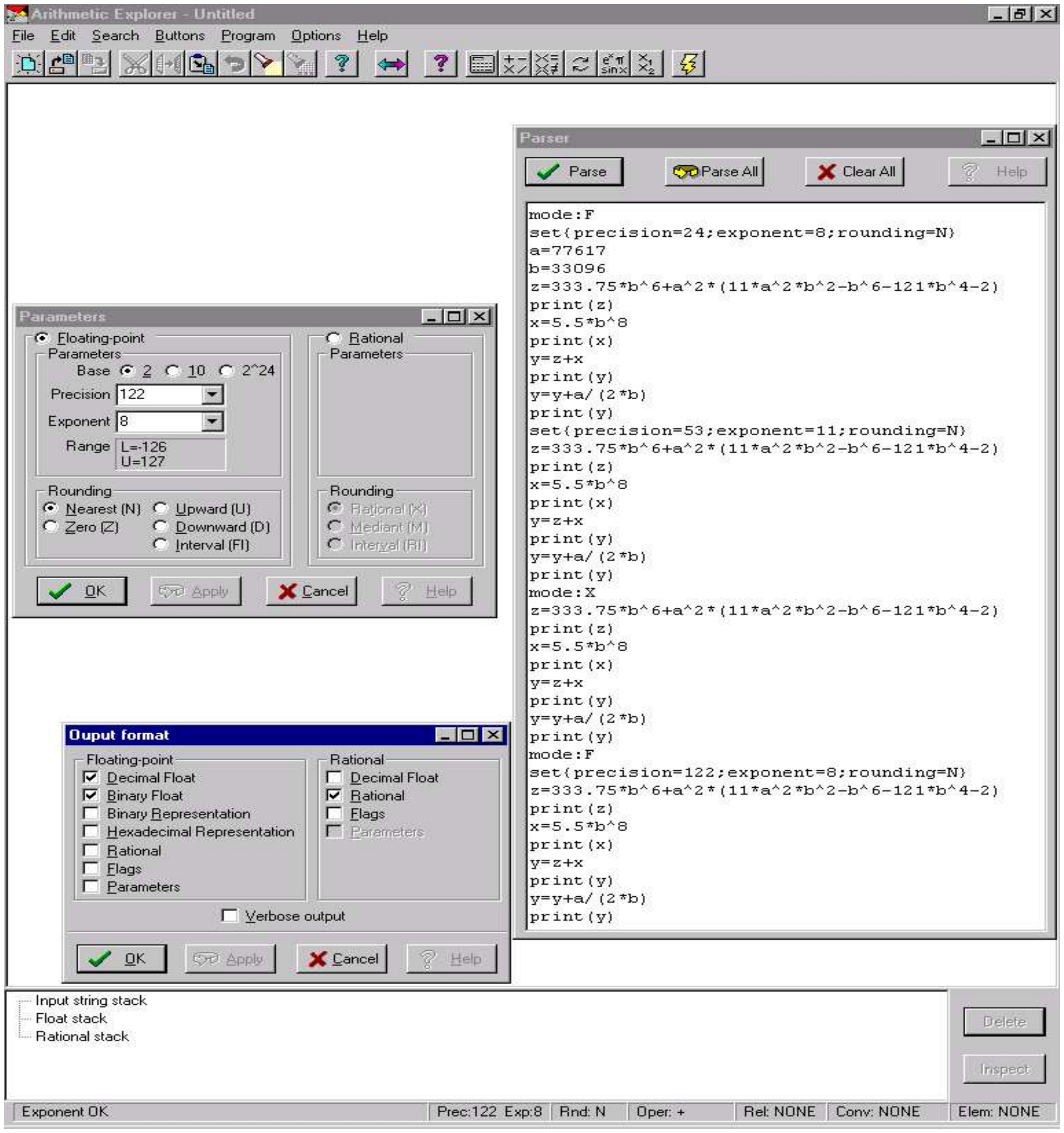
In parallel with the implementation of the class library, a cross-platform graphical user interface (GUI) has been developed in Qt [20]. The Qt-library makes it possible to develop GUI software for both the X Window System and Microsoft Windows in one sweep. By using Qt, our GUI is automatically available on a wide range of platforms without the need of recoding. Qt is also used by leading software companies like HP, IBM, Intel, Siemens, and it is the basis of the KDE desktop environment in Linux.

5 ARIΘΜΟΣ V1.0

The implementation was initially developed for didactical purposes, in the framework of the course Computer Arithmetic and Numerical Techniques [4], which is taught at the University of Antwerp. But it was soon further extended into a full-fledged, performant and powerful arithmetic environment called ARIΘΜΟΣ.

To summarize, the following number representations are provided in ARIΘΜΟΣ V1.0: exactly rounded multiprecision floating-point arithmetic in the four traditional rounding modes (up, down, trunc, nearest), interval arithmetic with multiprecision floating-point endpoints and exact rational arithmetic. The complete functionality of ARIΘΜΟΣ is available both at the class library/programming language level (for use with a C++-compiler) as well as through a GUI (the easiest way to parse and evaluate some expressions). To illustrate the latter some screen dumps of the GUI can be found in the Figures 4 and 5 which we shall discuss now.

As shown in Figure 4, one can easily specify in which of the number sets the computations should be performed through the 'Parameters' dialog box. When computing in floating-point or interval arithmetic, the base, the precision and the exponent range of the floating-point numbers, as well as the rounding mode (nearest, zero, up, down or interval) can be specified. In rational mode, at this time only exact rational arithmetic is supported. Floating-slash rational arithmetic (based on mediant rounding) as well as rational interval arithmetic, which are grayed in the 'Parameters' dialog box, will be provided in ARIΘΜΟΣ V2.0. Through the 'Output format' dialog box, the user can specify the required output format. For didactical purposes for example, when choosing a small precision and exponent range, the binary representation of (intermediate) floating-point results may be very helpful to zoom in on computer arithmetic issues. Furthermore, as has been described in the previous sections, the exceptions occurring during the computations are signaled through flags. Whether or not these flags should be output can also be specified in the 'Output format' dialog box.


Arithmetic Explorer - Untitled
File Edit Search Buttons Program Options Help
Parser
Parse
Parse All
Clear All
Help

```
mode:F
set(precision=24;exponent=8;rounding=N)
a=77617
b=33096
z=333.75*b^6+a^2*(11*a^2*b^2-b^6-121*b^4-2)
print(z)
x=5.5*b^8
print(x)
y=z+x
print(y)
y=y+a/(2*b)
print(y)
set(precision=53;exponent=11;rounding=N)
z=333.75*b^6+a^2*(11*a^2*b^2-b^6-121*b^4-2)
print(z)
x=5.5*b^8
print(x)
y=z+x
print(y)
y=y+a/(2*b)
print(y)
mode:X
z=333.75*b^6+a^2*(11*a^2*b^2-b^6-121*b^4-2)
print(z)
x=5.5*b^8
print(x)
y=z+x
print(y)
y=y+a/(2*b)
print(y)
mode:F
set(precision=122;exponent=8;rounding=N)
z=333.75*b^6+a^2*(11*a^2*b^2-b^6-121*b^4-2)
print(z)
x=5.5*b^8
print(x)
y=z+x
print(y)
y=y+a/(2*b)
print(y)
```

Parameters

[✓ Floating-point] Rational

Floating-point Parameters: Base [✓ 2] 10 2^24; Precision 122; Exponent 8; Range L=-126 U=127

Rounding: [✓ Nearest (N)] Upward (U) Zero (Z) Downward (D) Interval (FI)

Rational Rounding: [✓ Rational (X)] Mediant (M) Interval (RI)

OK Apply Cancel Help

Ouput format

Floating-point: [✓ Decimal Float] [✓ Binary Float] Binary Representation Hexadecimal Representation Rational Flags Parameters

Rational: Decimal Float [✓ Rational] Flags Parameters

Verbose output

OK Apply Cancel Help

Input string stack
Float stack
Rational stack

Delete Inspect

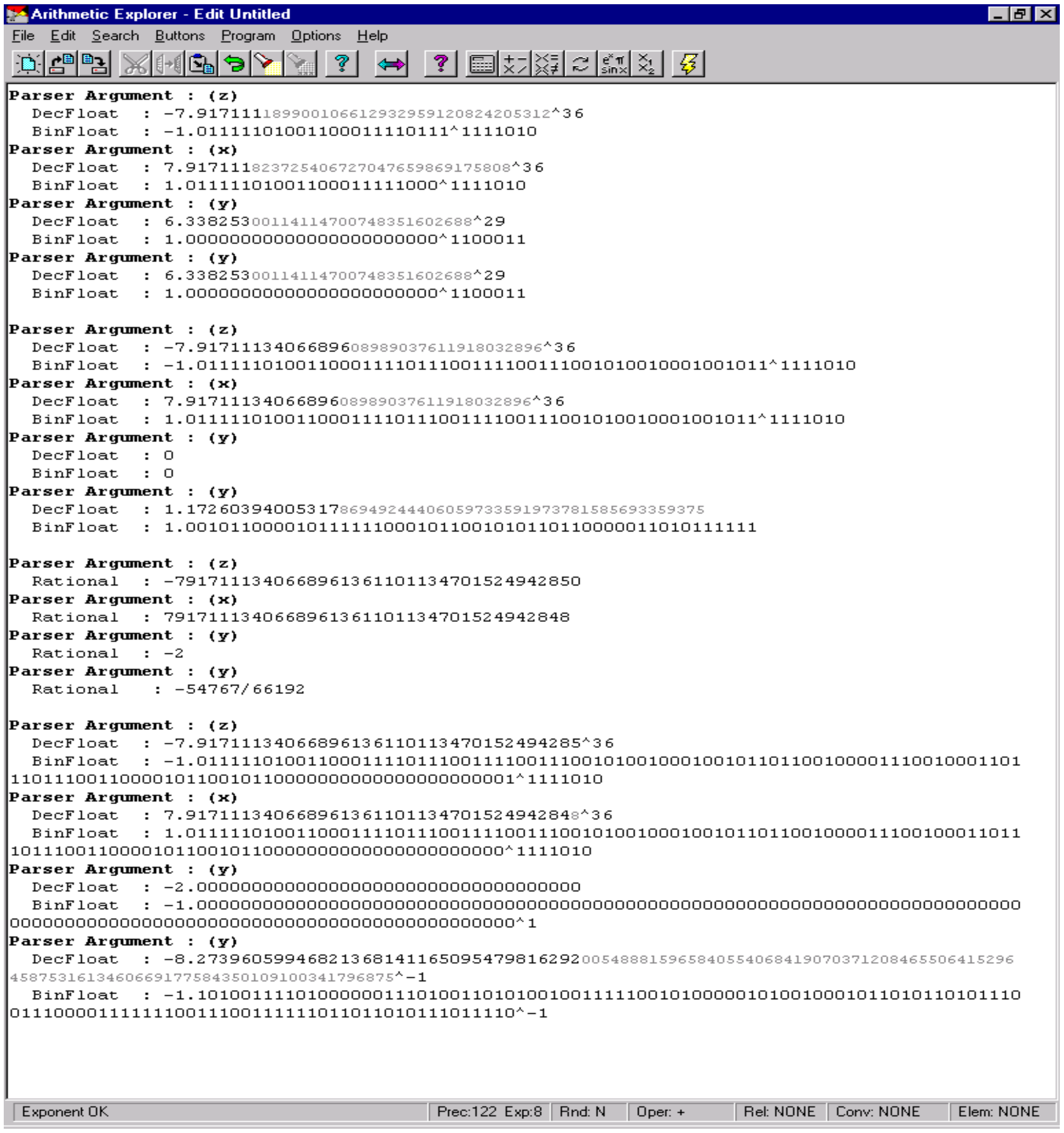
Exponent OK | Prec:122 Exp:8 | Rnd: N | Oper: + | Rel: NONE | Conv: NONE | Elem: NONE

Figure 4

The expression that is evaluated in Figure 4 is a remarkable example of catastrophic cancellation that was first published in [15]:

$$
\begin{aligned}
a &= 77617 \\
b &= 33096 \\
y &= 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \tfrac{a}{2b} \\
&= -0.827396\ldots
\end{aligned}
$$

The output generated by ARIΘMOΣ V1.0 for y, in different number representations, is displayed in Figure 5. It confirms what is already known [15, 5]: it requires either exact rational arithmetic or floating-point arithmetic with at least 122 bits of binary precision to obtain any significant digits in the computed result.


Arithmetic Explorer - Edit Untitled
File Edit Search Buttons Program Options Help

```
Parser Argument : (z)
  DecFloat  : -7.917111189900106612932959120824205312^36
  BinFloat  : -1.01111101001100011110111^1111010
Parser Argument : (x)
  DecFloat  : 7.917111823725406727047659869175808^36
  BinFloat  : 1.01111101001100011111000^1111010
Parser Argument : (y)
  DecFloat  : 6.338253001141147007483516026 88^29
  BinFloat  : 1.00000000000000000000000^1100011
Parser Argument : (y)
  DecFloat  : 6.33825300114114700748351602688^29
  BinFloat  : 1.00000000000000000000000^1100011

Parser Argument : (z)
  DecFloat  : -7.917111340668960898903761191803289 6^36
  BinFloat  : -1.0111110100110001111011100111100111001010010001001011^1111010
Parser Argument : (x)
  DecFloat  : 7.9171113406689608989037611918032896^36
  BinFloat  : 1.0111110100110001111011100111100111001010010001001011^1111010
Parser Argument : (y)
  DecFloat  : 0
  BinFloat  : 0
Parser Argument : (y)
  DecFloat  : 1.172603940053178694924440605973359197378158569335 9375
  BinFloat  : 1.0010110000101111111000101100101011011000001101011 1111

Parser Argument : (z)
  Rational  : -79171113406689613611011347015 24942850
Parser Argument : (x)
  Rational  : 7917111340668961361101134701524942848
Parser Argument : (y)
  Rational  : -2
Parser Argument : (y)
  Rational   : -54767/66192

Parser Argument : (z)
  DecFloat  : -7.91711134066896136110113470152494285^36
  BinFloat  : -1.0111110100110001111011100111100111001010010001001011011001000011100100011 01
110111001100001011001011000000000000000000000001^1111010
Parser Argument : (x)
  DecFloat  : 7.917111340668961361101134701524942848^36
  BinFloat  : 1.0111110100110001111011100111100111001010010001001011011001000011100100011011
1011100110000101100101100000000000000000000000000^1111010
Parser Argument : (y)
  DecFloat  : -2.00000000000000000000000000000000000
  BinFloat  : -1.0000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000^1
Parser Argument : (y)
  DecFloat  : -8.2739605994682136814116509547981629200548881596584055406841907037120846550641529 6
458753161346066917758435010910034179 6875^-1
  BinFloat  : -1.101001111010000001110100110101001001111100101000001010010001011010110101110
0111000011111110011100111111101101101011101 1110^-1
```

Figure 5

Future plans for ARIΘMOΣ include polynomial arithmetic (with coefficients of all real number types as well as intervals), complex arithmetic, circular arithmetic, reliable graphics and hybrid expression evaluation. More information on the ARIΘMOΣ environment can be found at http://win-www.uia.ac.be/u/cuyt/ or can be obtained from the authors.

References

[1] ANSI/IEEE Std 754-1985. IEEE standard for binary floating-point arithmetic. *ACM SIGPLAN*, 22(2):9–25, 1987.

[2] D.H. Bailey. A FORTRAN-90 Based Multiprecision System. *ACM TOMS*, 21(4):379–387, 1995.

[3] R.P. Brent. A FORTRAN multiple-precision arithmetic package. *ACM Trans. Math. Software*, 4:57–70, 1978.

[4] A. Cuyt and B. Verdonk. Computational science and engineering at Belgian universities. *IEEE Computational Science and Engineering*, 4(4):79–83, 1997.

[5] A. Cuyt and B. Verdonk. A remarkable example of catastrophic cancellation unraveled. Technical report, 1999. Submitted for publication in Computing.

[6] A. Dingle and T. Hildebrandt. Improving C++ performance using temporaries. *IEEE Computer*, 31(3):31–41, 1998.

[7] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surveys*, 23:5–48, 1991.

[8] T. Granlund. *The GNU Multiple Precision Arithmetic Library Edition 2.0.2.* Free Software Foundation, Inc., June 1996.

[9] B. Haible. CLN, a Class Library for Numbers. Available at http://clisp.cons.org/~haible/packages-cln.html.

[10] R.B. Kearfott, M. Dawande, K. Du, and Ch. Hu. Algorithm 737:INTLIB: a portable FORTRAN-77 elementary function library. *ACM Trans. Math. Software*, 20(4):447–459, 1994.

[11] R. Klatte, U. Kulisch, A. Wiethoff, et al. *C-XSC: a C++ class library for extended scientific computing.* Springer Verlag, Berlin, 1993.

[12] O. Knüppel. PROFIL/BIAS-a fast interval library. *Computing*, 53:277–287, 1994.

[13] D. Lea. Libg++. Copyright (C) 1988 Free Software Foundation.

[14] D. Matula and P. Kornerup. Finite precision rational arithmetic: slash number systems. *IEEE Trans. Comput.*, C-34:3–18, 1985.

[15] S. Rump. Algorithm for verified inclusion - theory and practice. In R.E. Moore, editor, *Reliability in Computing*, pages 109–126, 1988.

[16] S. Rump. Fast and parallel interval arithmetic. *BIT*, 39(3):534–554, 1999.

[17] M. Scott. Fast Rounding in Multiprecision Floating-Slash Arithmetic. *IEEE Trans. Comput.*, 38(7):1049–1052, 1989.

[18] Shamus Software Ltd. MIRACL, Multiprecision Integer and Rational Arithmetic C/C++ Library. Available at http://indigo.ie/~mscott.

[19] D. Smith. A FORTRAN package for floating-point multiple-precision arithmetic. *ACM Transactions on Mathematical Software*, 17(2):273–283, 1991.

[20] Troll Tech AS. QT, Cross-platform C++ GUI Application Framework. Available at ftp://ftp.troll.no/qt/pdf/qt-whitepaper-v10.pdf, 1999.

[21] D. Verschaeren and A. Cuyt. A class library for multiprecision IEEE floating-point arithmetic. Technical report, University of Antwerp, Dept. Math. and Comp. Science, 1999. In preparation.

[22] J. Vervloet. Rationale en polynomiale aritmetiek: een implementatie. University of Antwerp (UIA), 1999. Master's Thesis.

[23] H. G. Zimmer. SIMATH, a computer algebra system for number theoretic applications. Available at http://emmy.math.uni-sb.de/~simath/.