



# Lazy Algorithms for Exact Real Arithmetic

Pietro Di Gianantonio <sup>1,2</sup>

*Dipartimento di Matematica e Informatica  
Università di Udine  
via delle Scienze 206, I-33100 Udine, Italy*

Pier Luca Lanzi <sup>3</sup>

*Dipartimento di Elettronica e Informazione  
Politecnico di Milano  
Piazza Leonardo da Vinci 32, I-20133 Milano, Italy*

---

## Abstract

In this article we propose a new representation for the real numbers. This representation can be conveniently used to implement exact real number computation with a lazy programming languages. In fact the new representation permits the exploitation of hardware implementation of arithmetic functions without generating the granularity problem. Moreover we present a variation of the Karatsuba algorithm for multiplication of integers. The new algorithm performs exact real number multiplication in a lazy way and has a lower complexity than the standard algorithm.

*Keywords:* Exact real arithmetic, Karatsuba algorithm, lazy functional programming.

---

## 1 Introduction

Computation on real numbers is commonly solved, with a computer, using a subset of rationals. A real is represented using the classical floating point notation by a fixed size exponent and mantissa. The arithmetic based on this notation is intrinsically inexact. Errors are first introduced on data and then are propagated during computation. To cope with errors introduced by

---

<sup>1</sup> Research supported by Italian MIUR project COFIN 2001013518 CoMETA.

<sup>2</sup> Email: [digianantonio@dimi.uniud.it](mailto:digianantonio@dimi.uniud.it)

<sup>3</sup> Email: [lanzi@elet.polimi.it](mailto:lanzi@elet.polimi.it)

floating point representation a wide variety of methods of numerical analysis, relatively insensitive to rounding errors, have been devised. Due to the success of numerical analysis only in the latest years the alternative solutions to floating point arithmetic, dubbed *exact real arithmetic* have been studied[2,9,13].

Exact real arithmetic is the implementation of real arithmetic based on the constructive approach to classical analysis. A real number is not simply represented with an approximated value but with a function that can compute every approximation of the number. Thus the resulting arithmetic is error free because every result can be required with any number of fractional digits. Although computation using exact arithmetic will almost certainly require more computational resources than the corresponding computation using floating-point arithmetic the plumbing cost of computer resources suggests that exact real arithmetic may become an attractive option in some scientific and engineering applications. For this reason it is already used in computation where efficiency is not the primal goal such as formal testing of numerical algorithms.

Approaches to exact arithmetic can be classified in two main branches according to the representation used for reals and to the language used to implement the arithmetical algorithms. A real number can be represented, using rational converging series, with the characteristic function which generate the succession. The corresponding implementation [2] is dubbed *functional implementation*. A real can also be represented with an infinite sequence of digits that can be conveniently implemented using a language able to represent infinite lists such as lazy languages (e.g. Haskell, Scheme, Russell). The corresponding implementation is dubbed *lazy implementation*. One should remark that the digit notation is not the only notation used by lazy implementations, there are lazy implementations based on continuous fraction notation [12] or on the Moebius transformation [9].

This classification was first proposed by Boehm and Cartwright[3] who first realized implementations using both approaches and compared the resulting performance. Their research showed that even if real representations used in lazy implementations are more natural and elegant the resulting implementation fails to be as efficient as the functional one. This is mainly caused by coarse granularity of the representation and by arithmetical algorithms used in lazy implementation that have a higher complexity order. A representation has a coarse granularity when the finite approximations of mantissa approximated can have a number of binary digits that is multiple of some large number (typically 16 or 32), in a representation with coarse granularity many finite approximations of an arbitrary length are not allowed.

In this paper we propose a new lazy implementation for exact real arith-

metic which gives a solution to problems evidenced by previous authors. Our study starts to examine experiences and results that authors presented [1,3]. This lead us to a new representation for computation over real numbers dubbed “*Digit-Error*” notation which turns out to be insensitive to the representation granularity. The algorithms for arithmetic and transcendental functions are then given. Then we focus on multiplication algorithm. Classical algorithm commonly used in lazy implementations is, in general, less efficient than the algorithm used in functional implementations. We asked to ourselves if the new representation could lead to a better algorithm. The answer to this question is given by the new multiplication algorithm presented in this paper. This is inspired to Karatsuba’s algorithm for integer multiplication and have a complexity order lower than previous algorithms used in lazy implementations.

Section 2 introduce the mathematical foundations of exact arithmetic. The previous experiences on functional and lazy implementation made by Boehm and Cartwright, are the subject of Section 3. In Section 4 the new “*Digit-Error*” representation is introduced and it is shown how it solves granularity problem. Sections 5 is devoted to algorithms developed to implement the fundamental operation with “*Digit-Error*” notation, while in Section 6 the new multiplication algorithm is presented. A survey of all the implementations realized with “*Digit-Error*” notation is made in Section 7, while in Section 8 the results we obtained and future developments are discussed.

## 2 Formal Foundations

It is not possible to represent every real number in a computer, thus to implement exact real arithmetic only the subset of *Representable Reals* (dubbed also as *Constructive*, *Computable* or *Recursive Reals*) must be taken. This set contains every real for which there exists an algorithm that compute every approximation of the number. The first definition of this set was given by Turing [11] in 1937 and it has been the starting point for a great number of different approaches to the study of classical analysis concepts from a constructive viewpoint. This research field has been dubbed in different ways: *Constructive Analysis*, *Computational Analysis* or *Recursive Analysis*. The intuition underlying Constructive Analysis is simple: every mathematical statement must have a numerical meaning, for this reason the classical real set  $\mathbb{R}$  is substituted with the set of constructive reals. This set contains every real number commonly used and it is closed under all functions that commonly arise in analysis.

**Definition 2.1** [Representable Real] A real number  $r$  is **computable** if there

exists a computable function  $f_r : \mathbb{N} \rightarrow \mathbf{Q}$  such that

$$\forall n \in \mathbb{N} \quad |f_r(n) - r| < \frac{1}{n}$$

function  $f_r$  above is called **functional representation** for  $r$ .

Some interesting aspects of computability theory over constructive reals are given by the following proposition. Let  $r \in \mathbb{R}$  represent an arbitrary computable real number. No algorithm can compute in finite time:

- (i) if  $r = 0$ ;
- (ii) if  $r > r'$  for any fixed  $r' \in \mathbf{R}$ ;
- (iii) if  $r \in \mathbf{Q}$ ;
- (iv) the first digit of the decimal expansion of  $r$ ;
- (v) the first term  $[r] \in \mathbf{Z}$  of the regular continued fraction expansion of  $r$ ;
- (vi) the value  $f(r) \in \mathbf{R}$  of any function  $f$  which is not continuous at  $r$ .

### 3 Real Representations

An important difference between classical analysis and constructive analysis lies in the way real numbers are represented. In classical analysis different representations for real numbers define the same set  $\mathbb{R}$ . Thus analysis, as we know it, can be built without ever having to worry about the representation chosen. This is not true for constructive analysis where, even if many representations are still equivalent, there are some exceptions. For example Dedekind cuts and Cauchy sequences do not induce the same subset of computable functions on real numbers ([10]). The choice of representation become an important matter because it the first defines the subset of real numbers used and set of computable functions.

Between the various constructive representations for real numbers used in exact real arithmetic, the one presented in Definition 1 can be taken as reference because it has been the base for the first implementation proposed ([1]). But this representation is not the best choice from the efficiency point of view. If we want to implement the arithmetic operation in a computer is convenient to exploit the integer arithmetic already implemented in the hardware. This can be obtain representing real numbers using sequence of integers in fact:

**Proposition 3.1** *For every representable real  $r$  there exists a computable function  $g_r : \mathbb{N} \rightarrow \mathbf{Z}$  such that*

$$\forall n \in \mathbb{N} \quad |g_r(n) - r \times 4^n| < 1$$

Using (prop.1) exact real arithmetic based on representable reals is realized implementing reals by functions from  $\mathbb{N}$  to  $\mathbb{Z}$ , while algorithms are implemented by functionals. For instance the addition algorithm for two real numbers  $x$  and  $y$  is simply defined with a function  $f_{x+y}$  such that:

$$\forall n \in \mathbb{N} |f_{x+y}(n) - (x + y) \times 4^n| < 1$$

thus  $f_{x+y}$  can be defined as:

$$(1) \quad f_{x+y}(n) = \lfloor \frac{f_x(n+1) + f_y(n+1)}{4} \rfloor$$

An alternative representation proposed in literature is based on the positional radix representation and represents a real number as potentially infinite stream of digits. Thus a constructive real is represented by a computable sequence of integers  $d_0.d_1\dots$ . The first term  $d_0$  specifies the signed whole part of the number, the remaining terms  $(d_1d_2\dots)$  specify the fractional part of the number.

The lazy evaluation mechanism produces result digits on demand starting from digit  $d_0$ . At the first time only the first digit of the result is given. Subsequent digits can be computed when needed. This type of evaluation saves computational time against classical functional evaluation mechanism. First because only strictly needed digits are computed, second and most important lazy evaluation mechanism is monotonic: once a digit is computed it never changes, then computed digits can be stored in memory and do not need to be recalculated. Instead with functional implementation subsequent computation with growing accuracy must recalculate all the digits every time. Thus computational time is wasted. We refer to this fact as the *recalculation problem* for functional implementations.

It is a well-known fact that monotonicity of lazy evaluation mechanism makes standard arithmetic operations uncomputable with the classical decimal representation. Consider the following situation in the radix positional notation with decimal base. Let  $x$  denotes the numbers  $0.333\dots$ . There are two possible results for the multiplication by 3, namely  $1.000\dots$  and  $0.999\dots$ . If the algorithm generates 1 as first digit, this happen after the algorithm has examined a finite number of digits of the argument. Suppose that the first  $n$  digits have been examined before generating 1. The addition algorithm is going to generate 1 as a first digit also when the argument is  $0.3\dots 30\dots$ . The given result is incorrect, because it should be  $0.9\dots 90\dots$ . An analogous consideration can be done if the algorithm generates 0 as first digit. Multiplication by 3 should collect an infinite amount of information before it can generate the first digit of the answer. This is not an isolated phenomenon, similar examples can also demonstrate that the other arithmetical operations are not computable on classical representation.

A simple solution to this computability problem consists in representing a real number by a stream of positive and negative digits. Consider again the example above. The addition algorithm based on the new representation can generate 1 as first digit, if after  $n$  digits the argument becomes  $0.3\dots30$  then the algorithm can generate  $-1$  at position  $n + 1$ . The result become  $1.0\dots0(-1) = 0.9\dots9$ .

**Definition 3.2** A real number  $r$  is represented in *negative-digit* representation by an integer succession  $s_0\dots s_i\dots$  such that:

- (i)  $\forall i \in \mathbb{N}^+ - 10 < s_i < 10$
- (ii)  $r = \sum_{i=1}^{\infty} s_i 10^{-i}$

An exact real arithmetic based on negative-digit representation is best implemented when the base equals to the largest integer that the underlying hardware can manage. This permits faster operations on digits and best memory usage. Unfortunately exact real arithmetic based on negative-digit representation and large base evidence a *granularity problem* [2], that is best explained with an example. Given two real numbers  $x$  and  $y$  using negative-digit notation, in order to determine the first  $n$  fractional digits of the result  $x + y$ , it is necessary to determine the first  $n + 1$  digits of  $x$  and the first  $n + 1$  digits of  $y$ . Consider now the computation flow to evaluate the integer part of the result  $(x_1 + (x_2 + (\dots (x_{k-1} + x_k) \dots))$ . The program will perform  $k - 1$  additions in the order defined by the parenthesis, and thus it will need to examine  $i$  fractional digits of  $x_i$  to determine the integer part of the result. Thus  $x_n$  will be evaluated to  $n$  fractional digits that is typically  $16n$  or  $32n$  fractional bits if digits  $d_i$  are stored respectively as a one word or two words of memory. Instead it is clear that  $n$  fractional bits would be sufficient. Phenomena of this kind can be evidenced in many other contexts.

Another problem is evidenced using this type of representation. Algorithms used with lazy implementations are adapted from classical algorithms taught in grammar school. Thus complexity order is linear for addition and subtraction but quadratic for multiplication and division. This result is worse than the one obtained for implementations based on (def.1) where addition and subtraction algorithms are still linear but complexity of multiplication and division algorithms can be lowered to  $O(n^{1.59})$ <sup>4</sup>.

---

<sup>4</sup> Due to recalculation problem it is not possible to give an exact valuation of complexity order for functional multiplication.  $O(n^{1.59})$  must be taken as lower bound.

## 4 A “Digit-Error” representation for real numbers

As stated in the previous section the granularity problem, evidenced in lazy approaches, does not depend on the nature of the arithmetic operations, neither on the algorithms used. It only depends on the representation used for real numbers. For this reason our proposal for a new lazy approach to exact arithmetic begins with the definition of a new representation for computable reals.

The “Digit-Error” notation has been introduced by the first author in [4] as a first solution to granularity. This representation differs from the previous used [3] for two main features. First “Digit-Error” notation uses an enlarged set of digits, that is, from  $(-\mathbf{b}, \mathbf{b})$  to  $(-\mathbf{b}^2, \mathbf{b}^2)$  where  $\mathbf{b}$  is the base used.

Second, an integer is associated to every digit of the representation. This integer, called *error digit*, ranging over the interval  $[0, \mathbf{b}]$ , gives an upper bound to the numbers represented by the remaining, less significative, digits. This additional information is used by arithmetical algorithms to measure approximation errors that influence digits.

**Definition 4.1** A real number  $r$  is represented using “Digit-Error” notation, given a base  $\mathbf{b} \in \mathbb{N} : \mathbf{b} > 1$ , by a pair  $(e_m, m)$  where  $e_m \in \mathbb{Z}$  represent the exponent and  $m$  the mantissa given by the succession

$$\langle d_0, e_0 \rangle, \langle d_1, e_1 \rangle \dots$$

of integer pairs such that:

- (i)  $-\mathbf{b} < d_0 < \mathbf{b}$
- (ii)  $\forall i \in \mathbb{N}^+. -\mathbf{b}^2 < d_i < \mathbf{b}^2$
- (iii)  $\forall i \in \mathbb{N}. 0 \leq e_i \leq \mathbf{b}$
- (iv)  $\forall i \in \mathbb{N}. e_i \geq |\sum_{j \in \mathbb{N}^+} d_{i+j} \mathbf{b}^{-j}|$
- (v)  $r = \sum_{i \in \mathbb{N}} d_i \mathbf{b}^{-i}$

Note that with the mantissa part we can only represent numbers in the interval  $[-2\mathbf{b} + 1, 2\mathbf{b} - 1]$ . To represent arbitrary large real numbers the exponent is needed. To keep arguments simple we present the algorithms only for the mantissa part of the representation. Algorithms for the “mantissa–exponent” representation can be derived with minor technical problems.

It is easy to show that this definition is totally equivalent to the functional representation given in [3]. To show how the granularity problem is solved by (def.3) consider again the sum  $x + y$ : to evaluate  $n$  fractional digit of the result the algorithm for the “Digit-Error” notation will evaluate  $x$ , and  $y$  to the first  $n$  digits beyond the decimal point and the  $n - th$  error digits  $e_{x_n}, e_{y_n}$  of the arguments. If error  $e_{x_n} + e_{y_n}$  on digit  $n$  of the result is small enough

the result can be considered correct and no additional computation is needed, otherwise, if the error is too large the next digit of  $x$  and  $y$  will be evaluated to lower approximation errors on digits  $n$ . The granularity problem is then solved, in fact, using “*Digit-Error*” notation real numbers are evaluated only at the strictly requested accuracy.

## 5 Standard Algorithms

We present the main algorithms devised for “*Digit-Error*” representation these are similar to those presented in [3], derived from classical algorithms. A new multiplication algorithm with a complexity order less than classical is presented in Section 6 while in this section the implementation proposed in [4] is given. For sake of simplicity we present algorithms only for the mantissa part, the complete version of the algorithms can be found in [8].

Algorithms are written using Gofer-like syntax [6].

### 5.1 The norm function

Function *norm* has been introduced to lower the approximation of the first digit of the mantissa, augmenting the known accuracy. *norm* is unary and corresponds to identity functions: it only modifies the representation of the argument not the represented real value itself. This function reduce error  $e_0$  of the first digit  $x_0$  adding the possible carry of digit  $x_1$  to  $x_0$  by replacing  $x_1$  with the value  $x_1 \bmod \mathbf{b}$ .

**Algorithm 1 (Function  $norm_r$ )**

$$norm_r(\langle x_0, e_0 \rangle : \langle x_1, e_1 \rangle : xt\text{tail}) = \langle \overline{x_0}, 2 \rangle : \langle \overline{x_1}, e_1 \rangle : xt\text{tail}$$

where

$$\begin{aligned} \overline{x_0} &= x_0 + (x_1 \mathbf{div} \mathbf{b}) \\ \overline{x_1} &= x_1 \bmod \mathbf{b} \end{aligned}$$

The new value for the first digit error  $e_0$  is 2, that is the upper bound for the possible error on the first digit of the new representation.

### 5.2 Addition (Subtraction) Algorithm

As pointed out in Section 4 the addition algorithm for negative digit notation, to produce the result, has one digit of lookahead that causes granularity problems.

The algorithm presented here solves this problem using error digits.

**Algorithm 2 (Addition  $+_r$ )**

$$\begin{aligned}
 &+_r(\langle x_0, e_{x_0} \rangle : xtail, \langle y_0, e_{y_0} \rangle : ytail) = \\
 &|\langle x_0 + y_0, e_{x_0} + e_{y_0} \rangle : +_r(xtail, ytail), \\
 &\quad \text{if } e_{x_0} + e_{y_0} \leq \mathbf{b} \\
 &|+_r(norm_r(\langle x_0, e_{x_0} \rangle : xtail), norm_r(\langle y_0, e_{y_0} \rangle : ytail)), \\
 &\quad \text{if } e_{x_0} + e_{y_0} > \mathbf{b}
 \end{aligned}$$

The *norm* function is used to lower accuracy error on the arguments when the result digit turn to be not enough accurate or equivalently when  $e_{x_0} + e_{y_0} > \mathbf{b}$ . First digit of the sum of two reals is computed as follows. First the sum of the first error digits is computed. If  $e_{x_0} + e_{y_0}$  is small enough the first digit of the sum  $x_0 + y_0$  is produced. Otherwise, if  $e_{x_0} + e_{y_0}$  is too large, that is greater than the base, function *norm<sub>r</sub>* is applied to arguments and (alg.2) is applied again. As shown in Section 4 this algorithm does not suffer of the granularity problem.

Subtraction algorithm is similar.

5.3 Classical Multiplication

Given two real numbers  $x, y$  expressed by

$$x = \sum_{i=0}^{\infty} x_i \mathbf{b}^{-i} \quad y = \sum_{i=0}^{\infty} y_i \mathbf{b}^{-i}$$

the product  $x \times y$  can be written as

$$\begin{aligned}
 x \times y &= (x_0 + \mathbf{b}^{-1}x_{t_1}) \times (y_0 + \mathbf{b}^{-1}y_{t_1}) \\
 &= x_0y_0 + \mathbf{b}^{-1}(y_0x_{t_1} + x_0y_{t_1}) + \mathbf{b}^{-2}y_{t_1}x_{t_1}
 \end{aligned}$$

where

$$x_{t_1} = \sum_{i=0}^{\infty} x_{i+1} \mathbf{b}^{-i} \quad y_{t_1} = \sum_{i=0}^{\infty} y_{i+1} \mathbf{b}^{-i}$$

Thus result can be computed using more simple operations such product of a real number and a digit. Implementations presented in literature are all based on this type of algorithm ([4,3]). Multiplication algorithm for “Digit-Error” notation presented in [4] is given using functions  $\times_r$  and  $\times_{aux}$ :

- $\times_r(x, y)$  computes  $(x \times y) / \mathbf{b}^2$
- $\times_{aux}(d, x, r)$  computes  $(d \times x) / \mathbf{b}^2 + r / \mathbf{b}$ , where  $d, r$  are digits (integers) and  $x$  is a real.

**Algorithm 3 (Function  $\times_r$ )**

$$\begin{aligned} & \times_r(\langle x_0, e_{x_0} \rangle : x_{tail}, \langle y_0, e_{y_0} \rangle : y_{tail}) = \\ & | \langle x_0 y_0 \mathbf{div} \mathbf{b}^2, e_{xy_0} \rangle : +_r(+_r(\times_{aux}(x_0, y_{tail}, x_0 y_0 \bmod \mathbf{b}^2), \\ & \quad \times_{aux}(y_0, x_{tail}, x_0 y_0 \bmod \mathbf{b}^2)), \\ & \quad \langle 0, 1 + x_0 y_0 \bmod \mathbf{b}^2 \rangle : \times_r(x_{tail}, y_{tail})), \\ & \quad \text{if } e_{xy_0} \leq \mathbf{b} \\ & | \times_r(\text{norm}_r(\langle x_0, e_{x_0} \rangle : x_{tail}), \text{norm}_r(\langle y_0, e_{y_0} \rangle : y_{tail})) \\ & \quad \text{if } e_{xy_0} > \mathbf{b} \\ & \quad \text{where } e_{xy_0} = 2 + (x_0 e_{y_0} + y_0 e_{x_0}) \mathbf{div} \mathbf{b}^2 \end{aligned}$$

**Algorithm 4 (Function  $\times_{aux}$ )**

$$\begin{aligned} & \times_{aux}(d, \langle x_0, e_0 \rangle : x_{tail}, r) = \\ & | \langle value \mathbf{div} \mathbf{b}^2, error \rangle : \times_{aux}(d, x_{tail}, value \bmod \mathbf{b}^2) \\ & \quad \text{if } error \leq \mathbf{b} \\ & | \times_{aux}(d \text{norm}(\langle x_0, e_0 \rangle : x_{tail}), r) \\ & \quad \text{otherwise} \\ & \quad \text{where} \\ & \quad \quad error = 2 + (d * e_0) \mathbf{div} \mathbf{b}^2 \\ & \quad \quad value = x_0 * d + r * \mathbf{b} \end{aligned}$$

5.4 Division

There are two algorithms that can be used to implement division between two real numbers. The former uses the Newton’s method to compute the inverse of the divisor and then multiply the result by the dividend. This has been used with functional implementations. The second algorithm is derived from the Euclidean division algorithm adapted to work with an infinite divisor. This is the algorithm commonly used with lazy implementations, and it is the one we also used to realize division on “Digit-Error” notation.

Our implementation uses a set of auxiliary functions, namely  $\div_{aux}$ ,  $remainder_r$ , and  $shift$ .

$\div_r(x, y)$  initializes the computation calling the function  $\div_{aux}$ .

$\div_{aux}(x, y, e)$  is the main part of the algorithm for division between  $x$  and  $y$ , the argument  $e$  is used to calculate the errors.

$remainder_r(x, y, d)$  returns the value  $x - yd$ , where the argument  $d$  is a digit.

$shift(x)$  returns the value  $x\mathbf{b}$ .

The formal definition is the following.

**Algorithm 5 (Quotient:  $\div_r$ )**

$$\begin{aligned} & \div_r(\langle x_0, e_0 \rangle : x_{tail}, \langle y_0, f_0 \rangle : y_{tail}) = \\ & | \div_{aux}(\langle x_0, e_0 \rangle : x_{tail}, norm(\langle y_0, f_0 \rangle : y_{tail}), y_0 * (y_0 - 2)), \\ & \quad \text{if } \mathbf{b} < y_0 \\ & | \div_r(shift(\langle x_0, e_0 \rangle : x_{tail}), shift(\langle y_0, f_0 \rangle : y_{tail})), \\ & \quad \text{if } \mathbf{b} \geq y_0 \end{aligned}$$

**Algorithm 6 (Quotient:  $\div_{aux}$ )**

$$\begin{aligned} & \div_{aux}(\langle x_0, e_0 \rangle : x_{tail}, \langle y_0, f_0 \rangle : y_{tail}, e] = \\ & | (\langle x_0 \div y_0, e_{xy_0} \rangle : \\ & \quad \div_{aux}(remainder(\langle x_0, e_0 \rangle : x_{tail}, \langle y_0, f_0 \rangle : y_{tail}, (x_0 \div y_0)), \\ & \quad \quad \langle y_0, f_0 \rangle : y_{tail}, x), \\ & \quad \text{if } e_{xy_0} < \mathbf{b} \\ & | \div_{aux}(norm(\langle x_0, e_0 \rangle : x_{tail}), \langle y_0, f_0 \rangle : y_{tail}, x), \\ & \quad \text{if } e_{xy_0} \geq \mathbf{b} \\ & \text{where } e_{xy_0} = 4 + (((x_0 * f_0) + (y_0 * e_0)) \div e) \end{aligned}$$

**Algorithm 7 (Division:  $remainder_r$ )**

$$\begin{aligned} & remainder_r(\langle x_0, e_0 \rangle : x_{tail}, \langle y_0, f_0 \rangle : y_{tail}, d) = \\ & shift(-_r(\langle x_0, e_0 \rangle : x_{tail}, shift(shift(\times_{aux}(d, \langle y_0, f_0 \rangle : y_{tail}, 0)))))) \end{aligned}$$

Note that the presented algorithm for division uses only the first digits of the arguments to compute the first digit of the result while it is the remainder, obtained using the computed digit, that keeps trace of both the infinite arguments.

5.5 Transcendental Functions:  $\sin x, \cos x, e^x$

Other functions realized for “Digit-Error” notation are sine, cosine and exponential. We compute these functions using Mac Laurin series approximation:

$$\begin{aligned} e^x &= 1 + x + \frac{x^2}{2} + \dots + \frac{x^n}{n!} + R_n(e, x) \\ \sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} + \dots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + R_n(\sin, x) \\ \cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \dots + (-1)^n \frac{x^{2n}}{(2n)!} + R_n(\cos, x) \end{aligned}$$

To compute these series in the “*Digit-Error*” notation is quite simple. Consider first how an approximated result for a suitable  $f(x)$  is obtained with a given accuracy from Mac Laurin Series. Given a real number  $r$  and the required precision, using the remainder  $R(f, x)$  the number of  $n$  terms that guarantee the expected accuracy is computed. Then the first  $n$  terms are added and the result is given.

Almost the same algorithm can be applied to “*Digit-Error*” notation: the first digit of the result is computed adding first  $n_1$  terms that assure its exactness. For every subsequent digit  $j$  the precision is improved adding to previous result  $n_j$  terms that guarantee the exactness of the next digit. If all these operations are made using algorithms just proposed, the result produced is represented in “*Digit-Error*” notation.

Nevertheless, implementation of this algorithm is complicated by the computation of approximation error  $R(f, x)$  that in general is:

$$R(f, x) = f^{(n)}(\theta) \frac{x^{n+1}}{(n+1)!}$$

For  $e^x$  the formula becomes:

$$(2) \quad R_n(x) = e^\theta \frac{x^{n+1}}{(n+1)!} \theta \in (-x, x)$$

A problem is evidenced: to compute approximation error for function  $e^x$  the value  $e^\theta$  must be evaluated. An upper bound for  $e^\theta$  is needed. In our implementation we used:

$$R_n(e, x) = 3^{\lceil x \rceil} \frac{x^{n+1}}{(n+1)!}$$

Thus approximation error is computed with the underlying integer arithmetic only. Unfortunately the number of terms computed using the upper bound, when  $x$  is large, is greater than the number of terms really needed with (2).

## 6 Multiplication Algorithm

As evidenced in [3] arithmetic algorithms for multiplication and division used in lazy implementations are less efficient than the ones used with functional implementations. Although the former approach does not waste computational time for recalculation, due to algorithm complexity it turns out to be less efficient than the latter approach.

Owing to the similarity between (def.3) and classical positional notation the arithmetical algorithms used with lazy implementations have always been derived from classical ones. Thus complexity order is linear for addition and subtraction but quadratic for multiplication and division. Instead with func-

tional implementation is possible to use other algorithms whose complexity is considered to be less than quadratic. Multiplication algorithm for functional implementation, for example, can be derived from Karatsuba’s algorithm that is  $O(n^{1.59})$ .

By “Digit-Error” notation it has been possible to devise a new multiplication algorithm also inspired by the Karatsuba’s algorithm that has a complexity order equal to  $O(n^{1.83})$ , thus less than quadratic.

In this section we present a simplified version of the algorithm devised for multiplication on “Digit-Error” representation that is completely commented in [8].

To understand how this algorithm works first it is necessary to analyze Karatsuba’s algorithm[7].

### 6.1 Karatsuba’s Algorithm

The Karatsuba’s algorithm for binary multiplication presented in [7] can be easily adapted to work with rational numbers represented using a given base  $\mathbf{b}$ .

Given two real numbers  $x,y$  and a base ( $\mathbf{b} > 1$ ) such that:

$$x = \sum_{i=1}^{2^{n+1}} x_i \mathbf{b}^{-i} \qquad y = \sum_{i=1}^{2^{n+1}} y_i \mathbf{b}^{-i}$$

let

$$\begin{aligned} \overline{x}_M &= \sum_{i=1}^{2^n} x_i \mathbf{b}^{-i} & \overline{x}_L &= \sum_{i=1}^{2^n} x_{2^n+i} \mathbf{b}^{-i} \\ \overline{y}_M &= \sum_{i=1}^{2^n} y_i \mathbf{b}^{-i} & \overline{y}_L &= \sum_{i=1}^{2^n} y_{2^n+i} \mathbf{b}^{-i} \end{aligned}$$

the product  $xy$  can be expressed by

$$\begin{aligned} xy &= (1 + \mathbf{b}^{-2^n}) \overline{x}_M \overline{y}_M + \\ &\mathbf{b}^{-2^n} (\overline{x}_L - \overline{x}_M) (\overline{y}_M - \overline{y}_L) + \\ &(\mathbf{b}^{-2^n} + \mathbf{b}^{-2^{n+1}}) \overline{x}_L \overline{y}_L \end{aligned}$$

The result is computed using only three multiplications and three additions while classical algorithm use four multiplications. If  $T(n)$  is the time needed to multiply two numbers of  $n$  digits we have  $T(2n) \leq 3T(n) + cn$  for a suitable constant  $c$ . By induction it can be proved that:

$$T(n) = O(n^{\log_2 3})$$

$$= O(n^{1.59})$$

### 6.2 Multiplication for “Digit-Error” Representation

There are at least two observations that can be done about the algorithm above. First it only applies to finite representations. Second the kind of recursion used is not suitable for lazy evaluation mechanism. In fact Karatsuba’s algorithm starts splitting the addends from the middle and repeat this process until elementary computation on single digits are done. Instead lazy evaluation mechanism have to examine arguments from the first digit producing result digits proportionally to the number of argument digits examined. Thus Karatsuba’s algorithm cannot be implemented, as it is, for the “Digit-Error” notation.

The modified algorithm works as follows, to compute the product of two real numbers  $x$  and  $y$  it lazily evaluates a series of values  $\times_k(x, y, i)_{i \in \mathbb{N}}$  each one lazily evaluating the product of the approximation to the first  $2^i$  digits of  $x$  and  $y$  ( $x_0.x_1 \dots x_{2^i-1} \times y_0.y_1 \dots y_{2^i-1}$ ). The function  $\times_k(x, y, i)$  can then be implemented using the Karatsuba’s algorithm.

The approximated result computed by  $\times_k(x, y, i)$  can be used to compute  $xy$  as follows. Digit  $j$  produced by  $\times_k(x, y, i)$  can be given as an approximation of digit  $j$  from  $xy$  if the accuracy error computed on digit  $j$  is small enough. If it is not a more accurate approximation of digit  $j$  is taken from  $\times_k(x, y, i + 1)$ . In implemented the function  $\times_k$  we must assure that the computation performed to evaluate the first digits of  $\times_k(x, y, i)$  can be reused (does not need to be repeated) when evaluated the first digits of  $\times_k(x, y, i + 1)$ . Only in this way we can obtain an algorithm with a complexity lower than  $O(n^2)$ .

Let us consider now a lazy definition of the function  $\times_k$  obtained from Karatsuba’s algorithm. Given two real number  $x$  and  $y$  represented by “Digit-Error” notation using a base  $\mathbf{b}$ , product  $xy$  can be written as

$$\begin{aligned} \times_k(x, y, i + 1) &= \overline{x_M}(i)\overline{y_M}(i) + \\ &\mathbf{b}^{-2^i} [\overline{x_M}(i)\overline{y_M}(i) + (\overline{y_L}(i) - \overline{y_M}(i))(\overline{x_M}(i) - \overline{x_L}(i)) + \overline{x_L}(i)\overline{y_L}(i)] + \\ &\mathbf{b}^{-2^{i+1}} \overline{x_L}(i)\overline{y_L}(i) \end{aligned}$$

where  $\overline{x_M}(i)$  and  $\overline{x_L}(i)$  are the numbers represented by the first and the second group of  $2^i$  digit in  $x$ . If we define

$$\begin{aligned} t(i) &= \overline{x_M}(i)\overline{y_M}(i) + (\overline{y_L}(i) - \overline{y_M}(i))(\overline{x_L}(i) - \overline{x_M}(i)) + \overline{x_L}(i)\overline{y_L}(i) \\ s(i) &= \overline{x_L}(i)\overline{y_L}(i) \end{aligned}$$

then  $\times(x, y, i + 1)$  can be written as:

$$(3) \quad \times_k(x, y, i + 1) = \times_k(x, y, i) + \mathbf{b}^{-2^i} t(i) + \mathbf{b}^{-2^{i+1}} s(i).$$

The values  $t(i)$  and  $s(i)$  have an upper bound given a  $\mathbf{b} > 6$ . The result of  $\times_k(x, y, i + 1)$  can be used to produce  $2^{i+2}$  digits of the product between  $x$  and  $y$ . The computation  $\times_k(x, y, i + 1)$  starts generating digits from  $\times_k(x, y, i)$  whose error digits keep trace of the underlying approximation. If approximation error on digit  $j$  is too large means that  $\times_k(x, y, i)$  is no more sufficiently accurate to be taken as an approximation of  $xy$ . Thus remaining digits computed using  $\times_k(x, y, i + 1)$  are added to next digits produced by  $\times_k(x, y, i)$ . Computation continue until  $2^{i+1}$  digits from  $\times_k(x, y, i + 1)$  are produced adding to result from  $\times_k(x, y, i)$  digits from  $t(i + 1)$  and  $s(i + 1)$ .

The computation of  $\times_k(x, y, i)$  can be exploited to obtain a better formulation for  $\times_k(x, y, i + 1)$ :

$$\begin{aligned} \times_k(x, y, i + 1) = & \times_k(x, y, 0) + \mathbf{b}^{-1}t(0) + \mathbf{b}^{-2}s(0) + \dots \\ & + \mathbf{b}^{-2^i}t(i + 1) + \mathbf{b}^{-2^{i+1}}s(i + 1) \end{aligned}$$

It is not difficult to prove that the complexity order of the above algorithm can be bound by the formula:

$$T(n) = O(n^{1.83})$$

The complexity order for this multiplication algorithm is still greater than the complexity order for Karatsuba’s algorithm. This is awarded to the different evaluation mechanism used. The complexity could be lowered [7] but the algorithm will became more involved and we think this is not worth.

## 7 Implementation

Algorithms just proposed in functional style have been realized with different programming languages to test not only performance but also expressiveness of different programming languages.

The first implementation has been done using Scheme language[5]. Starting from the work of [4] the original notation has been changed and the new set of mathematical function has been developed. At first the original multiplication algorithm has been used. The actual code of the implementation can be find at the URL [www.dimi.uniud.it/~pietro/code](http://www.dimi.uniud.it/~pietro/code).

## References

- [1] H.-J. Boehm. Constructive real interpretation of numerical programs. *SIGPLAN Notice*, 22, 7:214–221, July 87.

- [2] H.-J. Boehm and R. Cartwright. Exact real arithmetic: formulating real numbers as functions. In David Turner, editor, *Research topics in functional programming*, pages 43–64. Addison-Wesley, 1990.
- [3] H.-J. Boehm, R. Cartwright, M. Riggie, and M.J. O’Donell. Exact real arithmetic: a case study in higher order programming. In *ACM Symposium on lisp and functional programming*, 1986.
- [4] P. Di Gianantonio. *A functional approach to real number computation*. PhD thesis, University of Pisa, 1993.
- [5] Chris Hansen et al. *MIT Scheme Reference Manual*, 1991.
- [6] Mark P. Jones. *An introduction to Gofer*.
- [7] D. E. Knuth. *The art of computer programming.*, volume 2/Seminumerical algorithms. Addison-Wesley, 1969.
- [8] P. Lanzi. Complessità degli algoritmi per l’aritmetica reale esatta. Master’s thesis, Università di Udine, 1994.
- [9] P. J. Potts, A. Edalat, and M. H. Escardo. Semantics of exact real arithmetic. In *IEEE Symposium on Logic in Computer Science*, 1997.
- [10] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics*. North-Holland, Amsterdam, 1988.
- [11] A.M. Turing. On computable numbers, with an application to the entscheidungs problem. In *Proc. London Math. Soc.* 42, pages 230–265, 1937.
- [12] J. Vuillemin. Exact real computer arithmetic with continued fraction. In *Proc. A.C.M. conference on Lisp and functional Programming*, pages 14–27, 1988.
- [13] K. Weihrauch. *”Computable Analysis, An Introduction”*. Springer-Verlag, 2000.