Effective real numbers in Mmxlib

Joris van der Hoeven
Mathématiques, CNRS (bât. 425)
Université Paris-Sud
91405 Orsay Cedex
France
joris@texmacs.org

ABSTRACT

Until now, the area of symbolic computation has mainly focused on the manipulation of algebraic expressions. Based on earlier, theoretical work, the author has started to develop a systematic C++ library Mmxlib for mathematically correct computations with more analytic objects, like complex numbers and analytic functions. While implementing the library, we found that several of our theoretical ideas had to be further improved or adapted. In this paper, we report on the current implementation, we present several new results and suggest directions for future improvements.

Categories and Subject Descriptors

F.2.1 [**Theory of Computation**]: Analysis of algorithms and problem complexity—*Numerical algorithms and problems*

General Terms

Algorithms

1. INTRODUCTION

Although the field of symbolic computation has given rise to several softwares for mathematically correct computations with algebraic expressions, similar tools for analytic computations are still somewhat inexistent.

Of course, a large amount of software for numerical analysis does exist, but the user generally has to make several error estimates by hand in order to guarantee the applicability of the method being used. There are also several systems for interval arithmetic, but the vast majority of them works only for fixed precisions. Finally, several systems have been developed for certified arbitrary precision computations with polynomial systems. However, such systems cannot cope with transcendental functions or differential equations.

The central concept of a systematic theory for certified computational analysis is the notion of an *effective real number* [17, 22, 4]. Such a number $x \in \mathbb{R}$ is given by an *approximation algorithm* which takes $\varepsilon \in \mathbb{D} = \mathbb{Z}2^{\mathbb{Z}}$ with $\varepsilon > 0$ on input and which produces an ε-*approximation* $\tilde{x} \in \mathbb{D}$ for x with $|\tilde{x} - x| < \varepsilon$. One defines effective complex numbers in a similar way.

Effective real and complex numbers are a bit tricky to manipulate: although they can easily be added, multiplied, etc., there exists no test for deciding whether an effective real number is identically zero. Some interesting questions from a complexity point of view are also raised: if we want to compute an ε-approximation of $y = x_1 + x_2$, how to determine $\delta_1 + \delta_2 = \varepsilon$ so that the computation of δ_i-approximations of the x_i is most efficient?

Concrete approaches and implementations for computations with effective real numbers have been proposed, often independently, by several authors [12, 3, 2, 13, 11, 14, 20]. A first step in these approaches is often to implement some interval arithmetic [1, 16, 7, 15, 18]. As an optional second step, one may then provide a class `real` for which a real number x is given by an *interval approximation algorithm* which, given $\varepsilon \in \mathbb{D}^{>}$, computes a closed interval $\boldsymbol{x} \ni x$ with endpoints in $\mathbb{D}$, of radius $< \varepsilon$.

In this paper, we report on the implementation of a C++ class for effective real numbers in the Mmxlib library [21]. This implementation is based on [20], but it also contains some new ideas.

In section 2, we start by quickly reviewing interval arithmetic and the computationally more efficient variant of "ball arithmetic" (see also [1, 16, 2]). We also try to establish a more precise semantics for this kind of arithmetic in the multi-precision context and discuss the use of interval classes as parameters of template classes such as complex numbers, matrices or polynomials. Our implementation relies on the Gmp and Mpfr libraries [6, 8].

In section 3, we mainly review previous work: equivalent definitions of effective real numbers, representation by dags and the different techniques for *a priori* and *a posteriori* error estimations. We also state improved versions of the "global approximation problem", which serve as a base for the complexity analysis of our library. We finally correct an error which slipped into [20].

In section 4, we describe the current implementation, which is based on the sole use of *a posteriori* error estimates.

In section 5, we prove that our implementation is optimal up to a linear overhead in the input size (4) and a logarithmic overhead in the time complexity (3). It is interesting to compare these results with previous theoretical work on the complexity of interval arithmetic [5, 9].

In the last section, we indicate how to use *a priori* error estimates in a more efficient way than in [20]. In a forthcoming paper, we hope to work out the details and remove the linear overhead (4) in the input size, at least under certain additional assumptions.

2. INTERVAL ARITHMETIC

Since real numbers cannot be stored in finite memory, a first approach to certified computations with real numbers is to compute with intervals instead of numbers. For instance, when using interval arithmetic, a real number like $x = \pi$ would be approximated by an interval like $\boldsymbol{x} = [\underline{\boldsymbol{x}}, \overline{\boldsymbol{x}}] = [3.141592, 3.141593]$. Evaluation of a real function f at a point $x \in \boldsymbol{x} = [\underline{\boldsymbol{x}}, \overline{\boldsymbol{x}}]$ then corresponds to finding an interval $\boldsymbol{y} = [\underline{\boldsymbol{y}}, \overline{\boldsymbol{y}}]$ with $f(x) \in \boldsymbol{y}$ for all $x \in \boldsymbol{x}$. When all functions under consideration are Lipschitz continuous and when all computations are performed using a sufficiently high precision, this technique provides arbitrarily good approximations for the real numbers we want to compute.

2.1 Representation issues

Our current interval library is based on MPFR [8], which implements a generalization of the IEEE 754 standard. More precisely, let $W \in \{32, 64, 128\}$ be the word precision of the computer under consideration. Given a bit precision $l < 2^W$ and a maximal exponent $m < 2^{W-1}$ fixed by the user, the MPFR library implements arithmetic on l-bit floating point numbers and exponents in the range $[-m, \ldots, m-1, m]$ with exact rounding. We recall that the IEEE 754 norm (and thus MPFR) includes special numbers ± 0, $\pm\infty$ and NaN (not a number). Assuming that m has been fixed once and for all, we will denote by $\mathbb{F}_l$ the set of l-bit numbers. Several representations are possible for the computation with l-bit intervals:

Endpoint representation. In this representation, an l-bit interval $\boldsymbol{x} = [\underline{\boldsymbol{x}}, \overline{\boldsymbol{x}}]$ is determined by its end-points $\underline{\boldsymbol{x}} \leqslant \overline{\boldsymbol{x}} \in \mathbb{F}_l \setminus \{\mathrm{NaN}\}$. We also allow for the exceptional value $\boldsymbol{x} = \mathrm{NaI} = [\mathrm{NaN}, \mathrm{NaN}]$. Since the MPFR library provides exact rounding, it is particularly simple to base an interval library on it, when using this representation: whenever a function f is monotonic on a certain range, it suffices to consider the values at the end-points with opposite rounding modes. However, every high precision evaluation of an interval function requires two high precision evaluations of floating point numbers.

Ball representation. In this representation, an l-bit interval $\boldsymbol{x} \neq \mathrm{NaI}$ is represented by a ball $\boldsymbol{x} = \mathcal{B}(c_{\boldsymbol{x}}, r_{\boldsymbol{x}})$ with center $c_{\boldsymbol{x}} \in \mathbb{F}_l \setminus \{\mathrm{NaN}\}$ and radius $r_{\boldsymbol{x}} \in \mathbb{F}_W^{\geqslant}$. If $l > W$ and $\boldsymbol{x} \neq \mathrm{NaI}$, then we also require that the ball $\mathcal{B}(c_{\boldsymbol{x}}, r_{\boldsymbol{x}})$ is *normal* in the sense that $r_{\boldsymbol{x}} \leqslant 2^{W-l} |c_{\boldsymbol{x}}|$. The underlying idea is that the endpoints of a high precision interval are usually identical apart from a few bits at the end, whence it is more efficient to only store the common part and the difference. As a consequence, one high precision operation on balls reduces to one high precision operation on floating point numbers and several operations on low precision numbers.

However, the ball representation has essentially less expressive power. For instance, it is impossible to represent the interval $[+0, +\infty]$. Also, positivity is not naturally preserved by addition if $l = W$. This may be problematic in the case we want to compute quantities like $\mathrm{hypot}(\boldsymbol{x}, \boldsymbol{y}) = \sqrt{\boldsymbol{x}^2 + \boldsymbol{y}^2}$.

Remark 1 The normality condition admits several variants. For instance, given $x \in \mathbb{F}_l \setminus \{\pm 0, \pm\infty, \mathrm{NaN}\}$, we define the *step* σ_x as the exponent of x minus l. A ball $\boldsymbol{x} = \mathcal{B}(c_{\boldsymbol{x}}, r_{\boldsymbol{x}})$ may then be called normal if $\sigma_{r_{\boldsymbol{x}}} \leqslant \sigma_{c_{\boldsymbol{x}}}$ instead of $r_{\boldsymbol{x}} < 2^{W-l}|c_{\boldsymbol{x}}|$. Alternatively, one may require $r_{\boldsymbol{x}} \in \{0, \ldots, 2^W - 1\} 2^{\sigma_{c_{\boldsymbol{x}}}}$.

Hybrid representation. In MMXLIB, we currently use the endpoint representation for low precision numbers $l \leqslant W$ and the ball representation for high precision numbers $l > W$. Modulo some additional overhead, this combines the advantages of both representations while removing their drawbacks.

2.2 Precision changes and semantics

Let $\mathbb{I}_{=l}$ denote the set of l-bit intervals for the hybrid representation. If $l > W$, then it should be noticed that the set $\mathbb{I}_{=l}$ is not stable under the usual arithmetic operations, due to the phenomenon of *precision loss*. Indeed, the sum $\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{y}$ of $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{I}_l$ is typically computed using $c_{\boldsymbol{z}} = c_{\boldsymbol{x}} +_{\mathbb{F}_l} c_{\boldsymbol{y}}$ and $r_{\boldsymbol{z}} = r_{\boldsymbol{x}} +_{\mathbb{F}_W} r_{\boldsymbol{y}} +_{\mathbb{F}_w} \varepsilon$, where ε is a small bound for the rounding error. However, if $\boldsymbol{x} \approx -\boldsymbol{y}$, then $\boldsymbol{z}$ is not necessarily normal.

Nevertheless, the set $\mathbb{I}_l = \bigcup_{W \leqslant k \leqslant l} \mathbb{I}_{=l}$ is stable under all usual arithmetic operations. Indeed, any ball or interval $\boldsymbol{z}$ may be normalized by replacing $c_{\boldsymbol{z}}$ by a lower bit approximation. More precisely, consider an abnormal interval $[\underline{\boldsymbol{z}}, \overline{\boldsymbol{z}}]$ with $c_{\boldsymbol{z}} \in \mathbb{F}_l$ and $r_{\boldsymbol{z}} \in \mathbb{F}_W$. Given $k \in \mathbb{Z}$, let $\underline{\boldsymbol{u}}_k = 2^k \lfloor 2^{-k} \underline{\boldsymbol{z}} \rfloor$ and $\overline{\boldsymbol{u}}_k = 2^k \lceil 2^{-k} \overline{\boldsymbol{z}} \rceil$, so that $c_{\boldsymbol{u}_k}, r_{\boldsymbol{u}_k} \in 2^{k-1}\mathbb{Z}$. Let $k \in \mathbb{Z}$ be minimal such that $c_{\boldsymbol{u}_k} \in \mathbb{F}_{l'}$, $r_{\boldsymbol{u}_k} \in \mathbb{F}_W$ and $r_{\boldsymbol{u}_k} \leqslant 2^{W-l'} |c_{\boldsymbol{u}_k}|$ for some $l' \in \{W+1, \ldots, l\}$. Then $\boldsymbol{u}_k = \mathrm{norm}(\boldsymbol{z})$ is called the *normalization* of $\boldsymbol{z}$. If no such k exists, then $\mathrm{norm}(\boldsymbol{z})$ is defined to be the smallest interval with endpoints in $\mathbb{F}_W$, which contains $\boldsymbol{z}$. It can be shown that $k \approx \sigma_{r_{\boldsymbol{z}}}$ and $r_{\mathrm{norm}(\boldsymbol{z})} \leqslant (1 + 2^{3-W}) r_{\boldsymbol{z}}$.

Remark 2 The normalization procedure is not very canonical and it admits several variants (in particular, it has to be adapted whenever we change the definition of normality). Nevertheless, all good normalization procedures share the property that $r_{\mathrm{norm}(\boldsymbol{z})} \leqslant (1 + B 2^{-W}) r_{\boldsymbol{z}}$ for some small fixed constant $B \in \mathbb{N}$.

Dually, it may occur that the result of an operation can be given with more precision than the argument. For instance, if $\boldsymbol{x} = \mathcal{B}(1, 0.5) 2^{100}$, then $\arctan \boldsymbol{x}$ can be computed with a precision of about 100 binary digits. Similarly, $\log \boldsymbol{x}$ can be computed with a precision of about 7 binary digits. We call this the phenomenon of *precision gain*. The point here is that it is not necessarily desirable to compute the results of $\arctan \boldsymbol{x}$ and $\log \boldsymbol{x}$ with the maximal possible precision.

Taking into account the phenomena of precision loss and gain, we propose the following "ideal" semantics for operations on intervals. First of all, a default precision l for computations is fixed by the user. Now assume that we wish to evaluate an n-ary function $f : \hat{\mathbb{R}}^n \to \hat{\mathbb{R}}$ at intervals $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{I}_l$, where $\hat{\mathbb{R}} = \mathbb{R} \cup \{\pm\infty, \mathrm{NaN}\}$. Then there exists a smallest interval $\boldsymbol{u}$ with $\underline{\boldsymbol{u}}, \overline{\boldsymbol{u}} \in \mathbb{F}_l$, which satisfies either

- $f(x_1, \ldots, x_n) \in \boldsymbol{u}$ for all $x_1 \in \boldsymbol{x}_1, \ldots, x_n \in \boldsymbol{x}_n$.
- $\boldsymbol{u} = \mathrm{NaIn}$ and $f(x_1, \ldots, x_n) = \mathrm{NaN}$ for some $x_1 \in \boldsymbol{x}_1, \ldots, x_n \in \boldsymbol{x}_n$.

Whenever $\{\mathrm{NaN}, \pm 0, \pm\infty\} \cap \boldsymbol{u} \neq \varnothing$, then $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ is taken to be the smallest interval of $\mathbb{F}_W$ which contains $\boldsymbol{u}$. Otherwise, we take $\boldsymbol{y} = \mathrm{norm}(\boldsymbol{u})$.

Remark 3 For some purposes one may use the alternative convention for exceptions that $\boldsymbol{u}$ is the smallest interval which contains all non exceptional values.

Since the normalization procedure is somewhat arbitrary (see remark 2), the ideal semantics may be loosened a little bit for implementation purposes. Instead of requiring the optimal return value $\boldsymbol{y} = \mathcal{B}(c_{\boldsymbol{y}}, r_{\boldsymbol{y}})$, we rather propose to content oneself with a return value $\tilde{\boldsymbol{y}} = \mathcal{B}(c_{\tilde{\boldsymbol{y}}}, r_{\tilde{\boldsymbol{y}}})$ with $|c_{\tilde{\boldsymbol{y}}} - c_{\boldsymbol{y}}| \leqslant (1 + B2^{-l})|c_{\boldsymbol{y}}|$ and $|r_{\tilde{\boldsymbol{y}}} - r_{\boldsymbol{y}}| \leqslant (1 + B2^{-l})|c_{\boldsymbol{y}}|$, for some fixed small constant $B \in \mathbb{N}$, in the case when $\boldsymbol{y}$ has precision $l > W$. This remark also applies to the underlying MPFR layer: exact rounding is not really necessary for our purpose. It would be sufficient to have a "looser" rounding mode, guaranteeing results up to B times the last bit.

2.3 Complex numbers and other template types

A tempting way to implement the complex analogue of interval arithmetic in C++ is to use the `complex` template class from the standard library. Unfortunately, this approach leads to a lot of overestimation for the bounding rectangles. In order to see this, consider the complex rectangle $x = \mathcal{B}_{1,\varepsilon} + \mathcal{B}_{1,\varepsilon}\mathrm{i}$ and the sequence $a_0 = x$, $a_{n+1} = x a_n$. Because multiplication with $1 + \mathrm{i}$ "turns" the bounding rectangle, the error ε is roughly multiplied by $\sqrt{2}$ at each step. In other words, we loose one bit of precision every two steps.

The above phenomenon can be reduced in two ways. First of all, one may use a better algorithm for computing a_n, like repeated squaring. In the case of complex numbers though, the best solution is to systematically use complex ball representations. However, standardization of the operations requires more effort. Indeed, given an operation f on balls $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n$ of precision W, it can be non-trivial to design an algorithm which computes a ball $\boldsymbol{u} \supseteq f(\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n)$ of almost minimal radius (up to $1 + B2^{-W}$).

The precision loss phenomenon is encountered more generally when combining interval arithmetic with template types. The best remedy is again to modify the algorithms and/or data types in a way that the errors in the data are all of a similar order of magnitude. For instance, when computing a monodromy matrix M as a product $M = \Delta_1 \cdots \Delta_k$ of connection matrices, it is best to compute this product by dichotomy $M = (\Delta_1 \cdots \Delta_{\lfloor k/2 \rfloor})(\Delta_{\lfloor k/2 \rfloor + 1} \cdots \Delta_k)$. Similarly, when computing the product $f(z)g(z)$ of two truncated power series, it is good to first perform a change of variables $z \to z/\rho$ which makes the errors in the coefficients of f and g of the same order of magnitude [19, Section 6.2]

3. EFFECTIVE REAL NUMBERS

For users of computer algebra systems, it would be convenient to provide a data type for real numbers which can be used in a similar way as the types of rational numbers, polynomials, etc. Since interval arithmetic already provides a way to perform certified computations with "approximate real numbers", this additional level of abstraction should mainly be thought of as a convenient interface. However, due to the fact that real numbers can only be represented by infinite structures like Cauchy sequences, their manipulation needs more care. Also, the actual implementation of a library of functions on effective real numbers raises several interesting computational complexity issues. In this section, we review some previous work on this matter.

3.1 Definitions and theoretical properties

Let $\mathbb{D} = \mathbb{Z}2^{\mathbb{Z}}$ denote the set of dyadic numbers. Given $x \in \mathbb{R}$ and $\varepsilon \in \mathbb{D}^{>}$, we recall from the introduction that an *ε-approximation* of x is a dyadic number $\tilde{x} \in \mathbb{D}$ with $|\tilde{x} - x| < \varepsilon$. We say that x is *effective*, if it admits an *approximation algorithm*, which takes $\varepsilon \in \mathbb{D}^{>}$ on input and which returns an *ε-approximation* for x. The *asymptotic time complexity* of such an approximation algorithm is the time it takes to compute a 2^{-n}-approximation for x, when $n \to \infty$. We denote by $\mathbb{R}^{\mathrm{eff}}$ the set of effective real numbers.

The above definition admits numerous variants [22, section 4.1]. For instance, instead of require an approximation algorithm, one may require the existence of an algorithm which associates a closed interval $\boldsymbol{x}_n = [\underline{\boldsymbol{x}}_n, \overline{\boldsymbol{x}}_n]$ with end-points in $\mathbb{D}$ to each $n \in \mathbb{N}$, such that $\boldsymbol{x}_0 \supseteq \boldsymbol{x}_1 \supseteq \cdots$ and $\lim_{n \to \infty} r_{\boldsymbol{x}_n} = 0$ (an interval $\boldsymbol{x} \ni x$ with end-points in $\mathbb{D}$ will also be called an *ε-bounding interval* for x). Similarly, one may require the existence of an effective and rapidly converging Cauchy sequence $\mathbb{N} \to \mathbb{D}; n \mapsto x_n$ for which there exists a number $M \in \mathbb{D}^{>}$ with $|x_n - x| \leqslant M2^{-n}$ for all n.

All these definitions have in common that an effective real number x is determined by an algorithm which provides more and more precise approximations of x on demand. In an object oriented language like C++, this can be implemented by providing an abstract representation class `real_rep` with a purely virtual method `approximate` which corresponds to this approximation algorithm. The class `real` is implemented as a pointer to `real_rep`.

Since effective real numbers should be thought of as algorithms, the zero-test problem in $\mathbb{R}^{\mathrm{eff}}$ can be reduced to the halting problem for Turing machines. Consequently, there exist no algorithms for the basic relations $=$, $\neq$, $<$, $>$, $\leqslant$ and $\geqslant$ on $\mathbb{R}^{\mathrm{eff}}$.

Given an open domain Ω of $(\mathbb{R}^{\mathrm{eff}})^n$, a real function $f : \Omega \to \mathbb{R}^{\mathrm{eff}}$ is said to be *effective* if there exists an algorithm $\check{f}$ which takes an approximation algorithm $\check{x} = (\check{x}_1, \ldots, \check{x}_n)$ for $x = (x_1, \ldots, x_n) \in \Omega$ on input and which produces an approximation algorithm $\check{y}$ for $y = f(x_1, \ldots, x_n)$. Here we understand that $\check{y}' = \check{f}(\check{x}')$ approximates the same number y, if $\check{x}'$ is another approximation algorithm for x.

Most common operations, like $+$, $-$, $\times$, $/$, exp, log, min, max, etc., can easily shown to be effective. On the other hand, without any of the operations for comparison, it seems more difficult to implement functions like $x \mapsto \lfloor x \rfloor$. In fact, it turns out that effective real functions are necessarily continuous [22, theorem 1.3.4].

3.2 Dag models

A concrete library for computations with effective real numbers consists of a finite number of functions like 0, 1, $+$, $-$, $\times$, exp, etc. Given inputs $x_1, \ldots, x_n$ of type `real`, such an operation should produce a new instance $y = f(x_1, \ldots, x_n)$ of `real`. Usually, the representation class for y in particular contains members for $x_1, \ldots, x_n$, which can then be used in the method which implements the approximation algo-

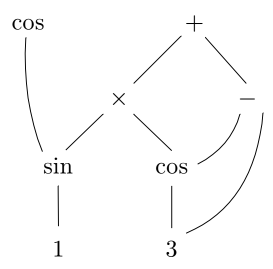

Figure 1: Example of a dag with 2 roots. The dag has *size* 8 (i.e. the total number of nodes) and *depth* 3 (i.e. the longest path from a root to a leaf). The *weight* of the dag corresponds to the sum of the sizes of the trees obtained by "copying" each of the roots. In our example, the weight is 13. Finally, the *ancestrality* of the dag is defined to be the maximum number of ancestors of a leaf. In our example, the ancestrality is 5.

rithm for y. For instance, a very simple implementation of addition might look as follows:

```
class add_real_rep: public real_rep {
  real x, y;
public:
  add_real_rep (const real& x2, const real& y2):
    x (x2), y (y2) {}
  dyadic approximate (const dyadic& err) {
    return x->approximate (err/2) + y->approximate (err/2);
  }
};
```

When implementing a library of effective real functions $f_1, f_2, \ldots$ in this way, we notice in particular that any effective real number computed by the library reproduces the expression by which it was computed in memory. Such effective real numbers may therefore be modeled faithfully by rooted dags (directed acyclic graphs) G, whose nodes are labeled by $f_1, f_2, \ldots$. More generally, finite sets of effective real numbers can be modeled by general dags of this type. Figure 1 shows an example of such a dag, together with some parameters for measuring its complexity.

Since storing entire computations in memory may require a lot of space, the bulk of computations should not be done on the effective real numbers themselves, but rather in their approximation methods. In particular, `real` should not be thought of as some kind of improved `double` type, which can be plugged into existing numerical algorithms: the `real` class rather provides a user-friendly high-level interface, for which new algorithms need to be developed.

3.3 The global approximation problems

Let $f_1, f_2, \ldots$ be a library of effective real functions as in the previous section, based on a corresponding library $\boldsymbol{f}_1, \boldsymbol{f}_2, \ldots$ of functions on intervals. In order to study the efficiency of our library, it is important to have a good model for the computational complexity. In this section, we will describe a static and a dynamic version of the *global approximation problem*, which are two attempts to capture the computational complexity issues in a precise manner.

In its static version, the input of the global approximation problem consists of

- A dag G, whose nodes are labeled by $f_1, f_2, \ldots$
- A challenge $\varepsilon_\alpha \in \mathbb{D}^> \cup \{+\infty\}$ for each node $\alpha \in \mathbb{N}$.

Denoting by f_α the function associated to the node α and by $\alpha_1, \ldots, \alpha_{|\alpha|}$ its children, we may recursively associate a real value $x_\alpha = f_\alpha(x_{\alpha_1}, \ldots, x_{\alpha_{|\alpha|}})$ to α. On output, we require for each node $\alpha \in G$ an interval $\boldsymbol{x}_\alpha$ with endpoints in $\mathbb{D}$, such that

- $x_\alpha \in \boldsymbol{x}_\alpha$ and $r_{\boldsymbol{x}_\alpha} < \varepsilon_\alpha$.
- For certain $\boldsymbol{x}'_{\alpha_1} \supseteq \boldsymbol{x}_{\alpha_1}, \ldots, \boldsymbol{x}'_{\alpha_{|\alpha|}} \supseteq \boldsymbol{x}_{\alpha_{|\alpha|}}$, we have
$$\boldsymbol{x}_\alpha = \boldsymbol{f}_\alpha(\boldsymbol{x}'_{\alpha_1}, \ldots, \boldsymbol{x}'_{\alpha_{|\alpha|}}).$$

Notice that the second condition implies in particular that $\boldsymbol{x}_\alpha \supseteq \boldsymbol{f}_\alpha(\boldsymbol{x}_{\alpha_1}, \ldots, \boldsymbol{x}_{\alpha_{|\alpha|}})$.

The dynamic version of the global approximation problem consists of a sequence of static global approximation problems for a fixed labeled dag G, when we incrementally add challenges ε_α for nodes α. More precisely, we are given

- A dag G, whose nodes are labeled by $f_1, f_2, \ldots$
- A finite sequence $(\alpha_1, \varepsilon_1), \ldots, (\alpha_k, \varepsilon_k)$ of pairs $(\alpha_i, \varepsilon_i) \in G \times (\mathbb{D}^> \cup \{\infty\})$.

On output, we require for each $i \in \{1, \ldots, k\}$ a solution to the i-th static global approximation problem, which consists of the labeled dag G with challenges $\varepsilon_\beta = \min\{\varepsilon_j : j \leqslant i, \alpha_i = \beta\}$. Here we understand that a solution at the stage $i+1$ may be presented as the set of changes w.r.t. the solution at stage i.

Let us explain why we think that the dynamic global approximation problem models the complexity of the library in an adequate way. For this, consider a computation by the library. The set of all effective real numbers constructed during the computation forms a labeled dag G. The successive calls of the approximation methods of these numbers naturally correspond to the sequence $(\alpha_1, \varepsilon_1), \ldots, (\alpha_k, \varepsilon_k)$.

It is reasonable to assume that the library itself does not construct any real numbers, i.e. all nodes of G correspond to explicit creations of real numbers by the user. Indeed, if new numbers are created from inside an approximation method, then all computations which are done with these numbers can be seen as parts of the approximation method, so they should not be taken into account during the complexity analysis. Similarly, if the constructor of a number $f(x_1, \ldots, x_n)$ induces the construction of other real numbers, then $f(x_1, \ldots, x_n)$ may be expressed in terms of more basic real functions, so we may consider f as a function outside our library.

Now assume that another, possibly better library were used for the same computation. It is reasonable to assume that the corresponding dag G' and challenges (a'_1, ε'_1), $\ldots$, $(a'_{k'}, \varepsilon'_{k'})$ coincide with the previous ones. Indeed, even though it might happen that the first and second library return different bounding intervals $\boldsymbol{x}_\alpha$ and $\boldsymbol{x}'_\alpha$ for a given challenge (α, ε), the libraries cannot know what the user wants to do with the result. Hence, for a fair comparison between the first and second library, we should assume that the user does not take advantage out of possible differences between $\boldsymbol{x}_\alpha$ and $\boldsymbol{x}'_\alpha$. This reduces to assuming that $G' = G$, $k' = k$ and $(a'_i, \varepsilon'_i) = (a_i, \varepsilon_i)$ for all i.

Finally, it is reasonable to assume that all actual approximations $\boldsymbol{x}_\alpha$ of the x_α are done using a fixed interval library $\boldsymbol{f}_1, \boldsymbol{f}_2, \ldots$. This means for instance that the second library has no better algorithms for multiplication, exponentiation, etc. than the first one. When putting all our "reasonable assumptions" together, the time of the computation which was spent in the library now corresponds to the time which was required to solve the corresponding dynamic global approximation problem.

3.4 A priori error estimates

Let us now consider the problem of obtaining an ε-approximation for the result of an operation $y = f(x_1, \ldots, x_n)$. For simplicity, we will focus on the case of addition $y = x_1 + x_2$. In this and the next section, we briefly recall several strategies, which are discussed in more detail in [20].

In the case of *a priori* error estimates, the tolerance ε is distributed *a priori* over x_1 and x_2. In other words, we first determine ε_1 and ε_2 with $\varepsilon_1 + \varepsilon_2 \leqslant \varepsilon$, next compute ε_i-approximations for the x_i, and finally add the results. The systematic choice of $\varepsilon_1 = \varepsilon_2 = \varepsilon/2$ can be very inefficient: in the case of badly balanced trees like in figure 2 (this occurs in practice when evaluating polynomials using Horner's rule), this requires the approximation of a_d with a much lower tolerance than a_1 ($\varepsilon/2^{d-1}$ versus $\varepsilon/2$).

This problem can be removed by balancing the error according to the weights w_1 and w_2 of x_1 and x_2 (i.e., by taking $\varepsilon_i = \varepsilon w_i/(w_1 + w_2)$). For "non degenerate" cases of the global approximation problem for a dag of weight w and size s, it can be shown this technique requires tolerances which are never worse than $\log w$ times the optimal ones.

Unfortunately, while implementing the algorithms from [20], it turned out that $\log w$ is often of the same order as s and therefore far from good enough. This is for instance the case when the expressions are obtained *via* some iterative process or as the coefficient of a lazy power series. For this reason, we have currently abandoned the use of *a priori* error estimates in our implementation. However, this situation is quite unsatisfactory, since this technique is still most efficient in many cases. We will come back to this problem in section 6.

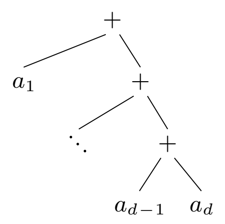

Figure 2: A badly balanced tree.

3.5 A posteriori error estimates

A second strategy consists of computing error estimates *a posteriori*: if we want to compute an ε-approximation for $y = x_1 + x_2$, we start with the computation of a bounding interval for y at precision W. As long as the obtained result is not sufficiently precise, we keep doubling the precision and repeating the same computation.

As explained in [20], this strategy can be optimized in two ways. First of all, the strategy may be carried out locally, by storing a "best available approximation" (together with the corresponding precision) for each instance of `real`. Indeed, when increasing the precision for the computation of y, sufficiently precise approximations for x_1 and x_2 might already be known, in which case their recomputation is unnecessary.

Secondly, instead of doubling the precision at each step, it is better to double the expected computation time. For instance, consider the computation of $y = f(x)$, where f has time complexity $\sim \lambda n^\alpha$ (i.e. y admits an $\sim \lambda n^\alpha + T(n + O(1))$ approximation algorithm, whenever x admits a $T(n)$ approximation algorithm). Evaluate $y = f(x)$ at successive precisions $1, 2^{1/\alpha}, 2^{2/\alpha}, \ldots, 2^{k/\alpha}$, where $k = \lceil \alpha \log_2 n \rceil$ and n is the smallest precision at which the evaluation yields a sufficiently precise result. Then the total computation time $\sim \lambda + 2\lambda + \cdots + 2^k \lambda \leqslant 2^{k+1}\lambda$ never exceeds $\lambda n^\alpha > 2^{k-1}\lambda$ by a factor more than 4 (see also [10]).

Unfortunately, an error slipped into [20], because the successive recursive approximations of x may not be sufficiently precise in order to allow for evaluations of $y = f(x)$ at successive precisions $1, 2^{1/\alpha}, 2^{2/\alpha}, \ldots, 2^{k/\alpha}$. For instance, if x is given by an algorithm of exponential time complexity 2^n, then successive approximations of x will only yield one more bit at every step. This error can be repaired up to a logarithmic factor in two ways. First of all, we notice that the error only concerns the cumulative cost of the successive reevaluation of $y = f(x)$. In section 5, we will prove that the total cost of reevaluating *all* nodes of the dag remains good.

Secondly, it is possible to adapt the technique of relaxed formal power series to real numbers. Roughly speaking, this approach relies on the recursive decomposition of a "relaxed mantissa" x of length l into a fixed part x_1 of length $2^p \geqslant l/2$ and a relaxed remainder x_2 (so that $x = x_1 + x_2$). Given an operation $y = f(x)$, we then compute $f(x_1)$ and $f'(x_1)$ at precision 2^{p+1} and obtain a formula for the relaxed decomposition $y = y_1 + y_2$, since y_1 is a truncation of $f(x_1)$ and $y_2 = f(x_1) - y_1 + f'(x_1)x_2$. As soon as the precision of x_2 exceeds $l/2$, we take a new value for x_1 and recompute $f(x_1)$ and $f'(x_1)$ at a doubled precision. Working out the details of this construction shows that most common real functions can be evaluated in a relaxed way with the same complexity as usual, multiplied by an $O(\log l)$ overhead.

However, the relaxed strategy accounts for a lot of additional implementation work and no noticeable improvement with respect to the global bound (3) which will be proved in section 5. Therefore, it is mainly interesting from a theoretical point of view.

4. EFFECTIVE NUMBERS IN MMXLIB

4.1 The classes `real` and `real_rep`

Inside MMXLIB, dyadic numbers in $\mathbb{D}$ are represented using generalized floating point numbers in $\mathbb{F}_l$, where l is bounded by a precision of the order of 2^{32} or 2^W.

Effective real numbers (of type `real`) are implemented as pointers to an abstract representation class `real_rep` with a virtual method for the computation of ε-bounding intervals. Usually, such a number is of the form $y = f(x_1, \ldots, x_n)$, where f is an effective real function and $x_1, \ldots, x_n$ are other effective real numbers. The number y is concretely represented by an instance of a class `f_real_rep` which derives from `real_rep` and with fields corresponding to $x_1, \ldots, x_n$.

The current implementation is based on the technique of *a posteriori* error bounds from section 3.5 with the two optimizations mentioned there: remembering the best currently available approximations for each real number and doubling computations times instead of precisions. These strategies are reflected as follows in the `real_rep` data type:

```
class real_rep {
protected:
  double   cost;
  interval best;
  real_rep (): cost (1.0) { compute (); }
  virtual int as_precision (double cost);
  virtual interval compute ();
public:
  interval improve (double new_cost);
  interval approximate (const dyadic& err);
};
```

The field `best` corresponds to the best currently available bounding interval for y. The value of `best` is recomputed several times by the purely virtual method `compute` at increasing *intended* costs, the last one of which is stored in `cost`. More precisely, `best` is recomputed as a function of approximations $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ of $x_1, \ldots, x_n$ at the same costs. When these approximations are sufficiently precise, then the cost of the computation of `best` will be more or less equal to `cost`. Otherwise, the actual computation may take less time (see the discussion at the end of section 3.5).

The costs are normalized (we start with `1.0`) and doubled at each iteration. The purely virtual method `as_precision` is used to convert an intended cost to the corresponding intended precision.

The user interface is given by the routines `improve` and `approximate`. The first one computes an approximation of y at intended cost `new_cost`:

```
interval real_rep::improve (double new_cost) {
  if (new_cost <= cost) return best;
  cost= max (new_cost, 2.0 * cost);
  set_precision (as_precision (cost));
  best= compute ();
  restore_precision ();
  return best;
}
```

The method `approximate` returns an ε-bounding interval $\boldsymbol{y}$ for y as a function of ε:

```
interval real_rep::approximate (const dyadic& eps) {
  while (radius (best) >= eps)
    (void) improve (2.0 * cost);
  return best;
}
```

Remark 4 In practice, the method `improve` also avoids the call of `compute` if the new precision associated to `cost` is equal to the old one. This may indeed happen if the cost of the operation increases more than linearly as a function of the bit precision.

4.2 Examples of derived classes of `real_rep`

Let us illustrate the mechanism from the previous section in the case of exponentiation. The exponential $y = \exp(x)$ of a number x is represented by an instance of

```
class exp_real_rep: public real_rep {
  real x;
  int as_precision (double cost);
  interval compute ();
public:
  exp_real_rep (const real& x2): x (x2) {}
};
```

The computation of n bits of y takes a time proportional to n^2 for small values of n and a more or less linear time for large values of n. Therefore, a simple implementation of `as_precision` would be as follows:

```
int exp_real_rep::as_precision (double cost) {
  if (cost <= 256.0) return (int) sqrt (cost);
  return min (MAX_PREC, (int) cost/16.0);
}
```

Of course, this is a very rough approximation of the real time complexity of exp. For the theoretical bounds in the next sections, better approximations are required. In practice however, a simple implementation like the above one is quite adequate. If necessary, one may implement a more precise algorithm, based on benchmarks. One may also gradually increase precisions and use a timer. The actual approximation of y is done using the overloaded function `exp` on intervals:

```
interval exp_real_rep::compute () {
  return exp (x->improve (cost));
}
```

In the case of functions with arity more than one it is often possible to avoid unnecessarily precise computations of one of the arguments, when the approximations of the other argument are far less precise. For instance, in the case of addition, `compute` may be implemented as follows:

```
interval add_real_rep::compute () {
  dyadic eps= pow (2.0, -BITS_IN_WORD)
  if (radius (y->best) < eps * radius (x->best)) {
    (void) x->improve (cost);
    while (y->cost < cost &&
           radius (y->best) >= eps * radius (x->best))
      (void) y->improve (2.0 * y->cost);
  }
  else
  else return x->improve (cost) + y->improve (cost);
}
```

4.3 Further notes

Comparisons. Even though there exists no reliable zero-tests for real numbers, concrete implementations might at least want to provide a heuristic one. Probably, it is best to provide an additional template parameter which allows for the customization of `==`, `!=`, `<`, etc. Currently, given a real number x, we systematically store $\mu_x = \max |\boldsymbol{x}^{\text{init}}|$, where $\boldsymbol{x}^{\text{init}}$ is the approximation of x at minimal cost 1. Whenever we want to test whether $x = 0$, we compute an ε-bounding interval $\boldsymbol{x} \ni x$ with $\varepsilon = \mu_x \delta$ for some fixed $\delta \ll 1$ like $\delta = 2^{-100}$, and test whether $0 \in \tilde{\boldsymbol{x}}$. Alternatively, one could approximate x at a fixed cost. Yet another approach is to keep a trace of all successful zero-tests and to try proving them simultaneously by symbolic methods at the end of the computation. Here we notice that a negative answer to a heuristic zero-test usually proves that $x \neq 0$, while a positive answer only indicates that $x = 0$ might hold.

Overflows. Our current implementation does not yet deal with overflows. Probably, an exception should be raised in this case and similarly for expressions like `exp(exp(1000))`.

Effective complex numbers. In practice, one may factor code by implementing `real` and `complex` as specializations `certify<interval>` resp. `certify<ball>` of a template type `certify`.

5. COMPLEXITY ANALYSIS

Notice that our Mmxlib implementation, as outlined in the previous section, naturally solves both the static and the dynamic versions of the global approximation problem: we first construct the dag G and then either compute an ε_α-approximation for each x_α, or successive ε_i-approximations for each x_i $(i = 1, \ldots, k)$. In this section, we examine the efficiency of this approach.

5.1 Total versus final complexity

Since x_α is approximated several times during our algorithm, let us first study the difference between the total computation time and the time taken by the final and most precise approximations of the x_α.

For each node α, let $t_{\alpha,0}, \ldots, t_{\alpha,p_\alpha}$ be the successive timings for the approximation of x_α. We will also denote by $T_{\alpha,0} < \cdots < T_{\alpha,p_\alpha}$ the intended computation times and precisions. By construction, we have $t_{\alpha,i} \leqslant T_{\alpha,i}$ for all i and $T_{\alpha_1} = 1, \ldots, T_{\alpha,p_\alpha} = 2^{p_\alpha}$. For each α, let $t_\alpha = t_{\alpha,0} + \cdots + t_{\alpha,p_\alpha} \leqslant T_\alpha = T_{\alpha,0} + \cdots + T_{\alpha,p_\alpha}$ and $t_\alpha^{\text{fin}} = t_{\alpha,p_\alpha}$. We define $t = \sum_{\alpha \in G} t_\alpha$, $T = \sum_{\alpha \in G} T_\alpha$ and $t^{\text{fin}} = \sum_{\alpha \in G} t_\alpha^{\text{fin}}$.

We already warned against the possibility that $t_{\alpha,i} < T_{\alpha,i}$. Nevertheless, we necessarily have $t_{\alpha,i} = T_{\alpha,i}$ if α is a leaf. Also, any operation f_α of cost $T_{\alpha,i}$ triggers an operation of cost $T_{\alpha,i}$ for one of the children of α. By induction, it follows that there exists at least one leaf $\lambda_{\alpha,i}$ descending from α which *really* spends a time $t_{\lambda_{\alpha,i},i} = T_{\lambda_{\alpha,i},i} = T_{\alpha,i}$. Hence, denoting by a the ancestrality of the dag and by Λ its subset of leaves, we have

$$T \leqslant \sum_{\alpha \in G, i} T_{\lambda_{\alpha,i},i} \leqslant a \sum_{\lambda \in \Lambda, i} T_{\lambda,i} = a \sum_{\lambda \in \Lambda, i} t_{\lambda,i} \leqslant at. \qquad (1)$$

We also have $T \leqslant 2T^{\text{fin}}$ since $T_{\alpha,1} + \cdots + T_{\alpha,p_\alpha} \leqslant 2T_{\alpha,p_\alpha}$ for all α. Consequently,

$$\frac{1}{a} T^{\text{fin}} \leqslant t \leqslant 2T^{\text{fin}}. \qquad (2)$$

The bound (1) is sharp in the case when the dag has only one leaf λ and *a*) the computation of an l digit approximation of x_λ requires exponential time; *b*) all other operations can be performed in linear or polynomial time. A similar situation occurs when cancellations occur during the computation of x_λ, in which case the computation of $\boldsymbol{x}_\lambda$ at many bits precision still produces a W-bit result.

A variant of (2), which is usually better, is obtained as follows. Since the precision of the result of an operation on intervals increases with the precision of the arguments, and similarly for the computation times, we have $t_{\alpha,1} \leqslant \cdots \leqslant t_{\alpha,p_\alpha}$. Let λ be a node (which can be assumed to be a leaf by what precedes) for which $p = p_\lambda$ is maximal. Then

$$t = \sum_{\alpha \in G, i} t_{\alpha,i} \leqslant \sum_{\alpha \in G} p_\alpha t_\alpha^{\text{fin}} \leqslant p t^{\text{fin}}.$$

It follows that

$$t^{\text{fin}} \leqslant t \leqslant (\log_2 t^{\text{fin}}) t^{\text{fin}}, \qquad (3)$$

since $p = \log_2 T_\lambda = \log_2 t_\lambda \leqslant \log_2 t^{\text{fin}}$.

5.2 Final versus optimal complexity

Let us now compare t^{fin} with the computation time t^{opt} for an optimal solution to the global approximation problem. In fact, it suffices to compare with an optimal solution for the static version: in the dynamic case, we consider the last static global approximation problem.

Denote by t_α^{opt} the computation time at each node α for a fixed optimal solution, so that $t^{\text{opt}} = \sum_{\alpha \in G} t_\alpha^{\text{opt}}$. If s is the size of the dag, then we claim that

$$t^{\text{opt}} \leqslant t^{\text{fin}} \leqslant 2s t^{\text{opt}}. \qquad (4)$$

For simplicity, we will assume that $p_\beta \geqslant p_\alpha$ whenever β is a descendant from α. This is no longer the case if we apply the optimization from the end of section 4.1, but the reasoning can probably be adapted to that case.

Assume for contradiction that (4) does not hold. Now consider the moment during the execution at which we first call `improve` with a maximal cost 2^p for some node α. At that point, the current cost c_β of each of the descendants β of α is $c_\beta = 2^{p-1} = T_\beta^{\text{fin}}/2$. When such a β is a leaf, it follows that $c_\beta = t_\beta^{\text{fin}}/2 \geqslant t^{\text{fin}}/(2s) > t^{\text{opt}} \geqslant t_\beta^{\text{opt}}$. By structural induction over the descendants β of α, it follows that $c_\beta \geqslant t_\beta^{\text{opt}}$ and the best available (resp. optimal) approximation $\boldsymbol{x}_\beta$ (resp. $\boldsymbol{x}_\beta^{\text{opt}}$) for x_β satisfies $r_{\boldsymbol{x}_\beta} \leqslant r_{\boldsymbol{x}_\beta^{\text{opt}}} < \varepsilon_\beta$. In particular $r_{\boldsymbol{x}_\alpha} < \varepsilon_\alpha$. On the other hand, the first call of `improve` with a maximal cost 2^p was necessarily triggered by `approximate`, whence $r_{\boldsymbol{x}_\alpha} \geqslant \varepsilon_\alpha$. This contradiction proves our claim.

Up to a constant factor, the bound (4) is sharp. Indeed, consider the case of a multiplication $x_1 \cdots x_n$ of n numbers which are all zero. When gradually increasing the precisions for the computation of $x_1, \ldots, x_n$, it can happen that one of the x_i produces bounding intervals $\boldsymbol{x}_i$ whose radii quickly converge to zero, contrary to each of the other x_j. In that case, the time spent on improving each of the x_j $(j \neq i)$ is a waste, whence we loose a factor n with respect to the optimal solution. On the other hand, without additional knowledge about the functions f_i, it is impossible to design a deterministic procedure for choosing the most efficient index i. In this sense, our current solution is still optimal. However, under additional monotonicity hypotheses on the cost functions, efficient indices i can be found, by taking into account the "cost per digit".

6. BACK TO A PRIORI ESTIMATES

Although the approach from the previous section has the advantage of never being extremely bad and rather easy to implement on top of an existing layer for interval arithmetic, there are even simple cases in which the factor s in the bound (3) is not necessary: in the truncated power series evaluation $b = a_0 + a_1/2 + \cdots + a_s/2^s$ with $|a_i| \leqslant 1$ for all i, the computation of a 2^{-n}-approximation of b induces the computation of n-bit approximations of each of the a_i. If $n \ll s$, this means that we spend a time $\approx ns$ instead of $\approx n^2$.

In order to remedy to this problem, we suggest to improve the balanced *a priori* estimate technique from [20] and cleverly recombine it with the current approach. In this section, we will briefly sketch how this could be done. The results are based on joint ideas with V. Kreinovich, which we plan to work out in a forthcoming paper.

6.1 Rigid dags

Let us start by isolating those situations in which *a priori* error estimates should be efficient. Consider a labeled dag G so that x_α admits an initial interval approximation $\boldsymbol{x}_\alpha \ni x_\alpha$ for each $\alpha \in G$. Assume also that for each node α and each child α_i of α, we have an interval $\boldsymbol{d}_{\alpha,i}$ with $(\partial x_\alpha / \partial x_{\alpha_i})(\boldsymbol{x}_{\alpha_1}, \ldots, \boldsymbol{x}_{\alpha_{|\alpha|}}) \subseteq \boldsymbol{d}_{\alpha,i}$. If $|\boldsymbol{d}_{\alpha,i}| < \infty$, then we say that G (together with the $\boldsymbol{x}_\alpha$) is a *Lipschitz* dag. If, in addition, we have $\boldsymbol{d}_{\alpha,i} \subseteq \mathcal{B}(c_{\boldsymbol{d}_{\alpha,i}}, \varepsilon |c_{\boldsymbol{d}_{\alpha,i}}|)$ for some $0 < \varepsilon < 1$ and all α, i, then we say that G is *ε-rigid*.

A typical obstruction to the Lipschitz property occurs in dags like $\sqrt{0}$. Similarly, a dag like 0×0 is typically Lipschitz, but not rigid. Given a Lipschitz dag, a variant of automatic differentiation provides us with bounds for the error in x_α in terms of the errors in the x_λ, where λ ranges over the leaves below α. If G is ε-rigid, and especially when $\varepsilon < 2^{-W}$, then these bounds actually become very sharp.

For instance, given a rooted Lipschitz dag and a challenge ε at the root ω, one may compute a sufficient precision l for obtaining an ε-approximation of x_ω as follows. Let Δ_λ be the error in x_λ at each leaf λ, when computing with precision l. We have $\Delta_\lambda = r_\lambda 2^{-l}$ for some r_λ which depends on λ. Then we recursively estimate the error Δ_α at each node α by

$$\Delta_\alpha = \overline{|\boldsymbol{d}|}_{\alpha,1}\, \Delta_{\alpha_1} + \cdots + \overline{|\boldsymbol{d}|}_{\alpha,|\alpha|}\, \Delta_{\alpha_{|\alpha|}}\ .$$

This provides us with a bound of the form $\Delta_\omega = r_\omega 2^{-l}$ for the error at the root ω. We may thus take $l = \lceil \log_2 r_\omega / \varepsilon \rceil$. The approach can be further improved using similar ideas as in the implementation of addition in section 4.2.

6.2 Backward error bounds

Instead of working with a fixed precision l, a better idea is to compute the contribution $\nabla_\lambda = \partial\, \Delta_\omega / \partial\, \Delta_\lambda$ of the error Δ_λ at each leaf to the error Δ_ω at ω. This problem is dual to the problem of automatic differentiation, since it requires us to look at the opposite dag $G^\top$ of G, which is obtained by inverting the direction of the edges. Indeed, if $\alpha^1, \ldots, \alpha^{[\alpha]}$ denote all parents of a node α, and if α is the $i^{\alpha,j}$-th child of α^j for each j, then we take

$$\nabla_\alpha = \overline{|\boldsymbol{d}|}_{\alpha^1, i^{\alpha,1}} \nabla_{\alpha^1} + \cdots + \overline{|\boldsymbol{d}|}_{\alpha^{[\alpha]}, i^{[\alpha],1}} \nabla_{\alpha^{[\alpha]}}.$$

Together with the initial condition $\nabla_\omega = 1$, this allows us to compute ∇_λ for all leafs λ. In order to compute x_ω with error $\Delta_\omega < \varepsilon$, we may now balance ε over the leaves λ according to the ∇_α. More precisely, we compute an $\varepsilon_\lambda = \varepsilon/(p \nabla_\lambda)$-approximation of each x_λ, where p is the number of leaves, and recompute all other nodes using interval arithmetic. As an additional optimization, one may try to balance according to the computational complexities of the $\boldsymbol{f}_\alpha$.

The above strategy is a bit trickier to implement in an incremental way. Indeed, in the dynamic global approximation problem, we ask for ε-approximations at different nodes ω and, since good previous approximations may already be present, it is not always necessary to compute the complete dag below ω. A solution to this problem is to keep track of the "creation date" of each node α and to compute the ∇_α from the top down to the leaves, while first considering nodes with the latest creation date (i.e. by means of a heap). Whenever the computed ∇_α is so small that the current error $\Delta_\alpha = r_{\boldsymbol{x}_\alpha}$ at α contributes only marginally to the error $\Delta_\omega < \varepsilon$ at the top (i.e. $\nabla_\alpha\, \Delta_\alpha < \varepsilon 2^{-W}$), then it is not necessary to consider the descendants of α.

Acknowledgment. We would like to thank two of the referees for their detailed and interesting comments, suggestions and references. Unfortunately, due to space limitations, we could not anwer all questions in the revised version.

7. REFERENCES

[1] G. Alefeld and J. Herzberger. *Introduction to interval analysis.* Academic Press, 1983.

[2] J. Blanck. General purpose exact real arithmetic. Technical Report CSR 21-200, Luleå University of Technology, Sweden, 2002. `http://www.sm.luth.se/~jens/`.

[3] J. Blanck, V. Brattka, and P. Hertling, editors. *Computability and complexity in analysis*, volume 2064 of *Lect. Notes in Comp. Sc.* Springer, 2001.

[4] A. Edalat and P. Sünderhauf. A domain-theoretic approach to real number computation. *TCS*, 210:73–98, 1998.

[5] A. Gaganov. Computational complexity of the range of the polynomial in several variables. *Cybernetics*, pages 418–425, 1985.

[6] T. Granlund et al. GMP, the GNU multiple precision arithmetic library. `http://www.swox.com/gmp`, 1991–2006.

[7] M. Grimmer, K. Petras, and N. Revol. Multiple precision interval packages: Comparing different approaches. Technical Report RR 2003-32, LIP, École Normale Supérieure de Lyon, 2003.

[8] G. Hanrot, V. Lefèvre, K. Ryde, and P. Zimmermann. MPFR, a C library for multiple-precision floating-point computations with exact rounding. `http://www.mpfr.org`, 2000–2006.

[9] V. Kreinovich. For interval computations, if absolute accuracy is NP-hard, then so is relative accuracy+optimization. Technical Report UTEP-CS-99-45, UTEP-CS, 1999.

[10] V. Kreinovich and S. Rump. Towards optimal use of multi-precision arithmetic: a remark. Technical Report UTEP-CS-06-01, UTEP-CS, 2006.

[11] B. Lambov. The RealLib project. `http://www.brics.dk/~barnie/RealLib`, 2001–2006.

[12] V. Ménissier-Morain. Arbitrary precision real arithmetic: design and algorithms. Unpublished.

[13] N. Müller. iRRAM, exact arithmetic in C++. `http://www.informatik.uni-trier.de/iRRAM/`, 2000–2006.

[14] R. O'Connor. A monadic, functional implementation of real numbers. Technical report, Institute for Computing and Information Science, Radboud University Nijmegen, 2005.

[15] N. Revol. MPFI, a multiple precision interval arithmetic library. `http://perso.ens-lyon.fr/nathalie.revol/software.html`, 2001–2006.

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

[17] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Maths. Soc.*, 2(42):230–265, 1936.

[18] J. van der Hoeven. GMPX, a C-extension library for gmp. `http://www.math.u-psud.fr/~vdhoeven/`, 1999. No longer maintained.

[19] J. van der Hoeven. Relax, but don't be too lazy. *JSC*, 34:479–542, 2002.

[20] J. van der Hoeven. Computations with effective real numbers. *TCS*, 351:52–60, 2006.

[21] J. van der Hoeven et al. Mmxlib: the standard library for Mathemagix, 2002–2006. `http://www.mathemagix.org/mml.html`.

[22] K. Weihrauch. *Computable analysis.* Springer-Verlag, Berlin/Heidelberg, 2000.