Computations with effective real numbers

Joris van der Hoeven

Dépt. de Mathématiques (Bât. 425)
Université Paris-Sud
91405 Orsay Cedex
France
Email: joris@texmacs.org

January 6, 2004

A real number x is said to be effective if there exists an algorithm which, given a required tolerance $\varepsilon \in \mathbb{Z}\ 2^{\mathbb{Z}}$, returns a binary approximation $\tilde{x} \in \mathbb{Z}\ 2^{\mathbb{Z}}$ for x with $|\tilde{x} - x| < \varepsilon$. In this paper, we review several techniques for computations with such numbers, and we will present some new ones.

1. Introduction

A *dyadic number* is a number of the form $x = k\ 2^p$ with $k, p \in \mathbb{Z}$. We denote by $\mathbb{D} = \mathbb{Z}\, 2^{\mathbb{Z}}$ the set of dyadic numbers and $\mathbb{D}^{>} = \{x \in \mathbb{D}: x > 0\}$. Given $x \in \mathbb{R}$ and $\varepsilon \in \mathbb{D}^{>}$, an ε-approximation for x is a number $\tilde{x} \in \mathbb{D}$ with $|\tilde{x} - x| < \varepsilon$. An *approximation algorithm* for $x \in \mathbb{R}$ is an algorithm which takes a tolerance $\varepsilon \in \mathbb{D}$ on input and which returns an ε-approximation for x. A real number $x \in \mathbb{R}$ is said to be *effective*, if it admits an approximation algorithm. The aim of this paper is to review several recent techniques for computations with effective real numbers and to present a few new ones.

Effective real numbers can be useful in areas of numerical analysis where numerical instability is a major problem, like singularity theory. In such areas, multiple precision arithmetic is often necessary and the required precisions for auxiliary computations may heavily vary. We hope that the development of an adequate computational theory for effective real numbers will both allow us to automatically perform the error analysis and compute the required precisions at which computations have to take place.

We recall that their exists no general zero-test for effective real numbers. Nevertheless, exact or heuristically reliable zero-tests do exist for interesting subfields, which contain transcendental constants. However, this topic will not be studied in this paper and we refer to [Ric97, vdH01] for some recent work on this matter.

In an object-oriented language like C++, a natural way to represent an effective real number is by an abstract object with a method which corresponds to the approximation algorithm. When using this representation, which will be discussed in more detail in section 2, it is natural to perform *a priori* error estimations for common operations on real numbers. In other words, if we want to compute $y = f(x_1, \ldots, x_r)$ with precision ε, then we determine tolerances $\varepsilon_1, \ldots, \varepsilon_r$ such that the multiple precision evaluation of f at any ε_i-approximations for the x_i always yields an ε-approximation for y.

In some cases, *a priori* error estimates are quite pessimistic. An alternative technique for computing with effective real numbers is interval arithmetic. In this case, we approximate an effective real number x by an interval $[\underline{x}, \overline{x}]$ which contains x. Then the evaluation of $y = f(x_1, \ldots, x_r)$ comes down to the determination of an interval $[\underline{y}, \overline{y}]$ with

$$f([\underline{x_1}, \overline{x_1}], \ldots, [\underline{x_r}, \overline{x_r}]) \subseteq [\underline{y}, \overline{y}].$$

For continuous functions f, when starting with a very precise approximation of x (i.e. $\overline{x}-\underline{x}$ is very small), the obtained *a posteriori* error estimate for y will also become as small as desired. In section 3, this technique of *a posteriori* error estimates will be reviewed in more detail, as well as an efficient representation for high-precision intervals.

Unfortunately, in some cases, both *a priori* and *a posteriori* error estimates are quite pessimistic. In sections 4 and 5, we will therefore present two new techniques for the computation of “adaptive error estimates”. These techniques combine the advantages of *a priori* and *a posteriori* error estimates, while eliminating their major disadvantages. Moreover, this new technique remains reasonably easy to implement in a general purpose system for computations with effective real numbers. Under certain restrictions, the new techniques are also close to being optimal. This point will be discussed in section 6.

The approaches presented in sections 2 and 3 have been proposed, often independently, by several authors [BBH01, Bla02, GPR03] and we notice the existence of several similarities with the theory of effective power series [vdH97a, vdH02]. In the past, we experimentally implemented the approaches of sections 2 and 3 in the case of power series (not officially distributed) resp. real numbers [vdH99]. We are currently working on a more robust C++ library based on the ideas in this paper [vdH04]. Currently, this library contains the basic arithmetic operations. Some other implementations with similar objectives are currently known to the author [Mül04, Rev04]. All these implementations are free software and they are mainly based on the Gmp and Mpfr libraries [GO04, HLRZ04].

2. *A priori* error estimates

A natural way to represent an effective real number x is by its approximation algorithm. Conceptually speaking, this means that we view x as a black box which can be asked for approximations up to any desired precision:

$$\text{tolerance } \varepsilon \quad \longrightarrow \quad \boxed{x} \quad \longrightarrow \quad \varepsilon\text{-approximation } \tilde{x} \text{ for } x$$

In an object oriented language like C++, this can be implemented using an abstract base class `real_rep` with a virtual method for the approximation algorithm. Effective real numbers will then be pointers to `real_rep`. For instance,

```
class real_rep {
public:
  virtual dyadic approx (const dyadic& tol) = 0;
};
typedef real_rep* real;
```

Here `dyadic` stands for the class of dyadic numbers. In practice, these may be taken to be arbitrary precision floating point numbers. For simplicity, we also do not care about memory management. In a real implementation, one would need a mechanism for reference counting or conservative garbage collection.

Now assume that we want to evaluate $y=f(x_1,\ldots,x_r)$ for a given tolerance $\varepsilon\in\mathbb{D}^{>}$, where $x_1,\ldots,x_r$ are effective real numbers and f is an operation like $+$, $\times$ or exp. In order to make f *effective*, we need to construct an approximation algorithm for y as a function of $x_1,\ldots,x_r$. This both involves the dyadic approximation of the evaluation of f in dyadic approximations of the x_i and the determination of tolerances ε_1, …, ε_r for the dyadic approximations of the x_i. More precisely, $\varepsilon_1,\ldots,\varepsilon_r$ should be such that for any $\tilde{x}_1,\ldots,\tilde{x}_r\in\mathbb{D}$ with $|\tilde{x}_i-x_i|<\varepsilon_i$ $(i=1,\ldots,r)$, we have

$$|\tilde{f}_\varepsilon(\tilde{x}_1,\ldots,\tilde{x}_r)-f(x_1,\ldots,x_r)|<\varepsilon,$$

where $\tilde{f}_\varepsilon$ stands for a dyadic approximation algorithm for f which depends on the tolerance ε. For instance, in the case when f is the addition, we may use exact arithmetic on $\mathbb{D}$ (so that $\tilde{f}_\varepsilon = f$ for all ε) and take $\varepsilon_1 = \varepsilon_2 = \varepsilon/2$. This yields the following class for representing sums of real numbers:

```
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 approx (const dyadic& tol) {
    return x->approx (tol >> 1) + y->approx (tol >> 1); }
};
```

The addition can now be implemented as follows:

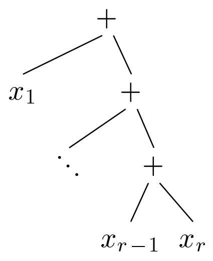
```
inline real operator + (const real& x, const real& y) {
  return new add_real_rep (x, y); }
```

Notice that, in a sense, we have really represented the sum of x and y by the expression $x + y$ (more generally, such expressions are dags). Nevertheless, the representation using an abstract class `real_rep` provides additional flexibility. For instance, we may attach additional information to the class `real_rep`, like the best currently known approximation for the number (thereby avoiding unnecessary recomputations). In practice, it is also good to provide an additional abstract method for computing a rough upper bound for the number. This gives a fine-grained control over potential cancellations.

The above approach heavily relies on the computation of *a priori* error estimates (i.e. the computation of the ε_i). If no additional techniques are used, then this leads to the following disadvantages:

P1. We do not take advantage of the fact that the numeric evaluation of $\tilde{f}_\varepsilon(\tilde{x}_1, \ldots, \tilde{x}_r)$ may lead to an approximation of y which is far better than the required tolerance ε. Indeed, multiple precision computations are usually done with a precision which is a multiple of the number of bits W in a machine word. "On average", we therefore gain something like $W/2$ bits of precision. In section 4, we will show that it may actually profitable to systematically compute more than necessary.

P2. The error estimates may be pessimistic, due to badly balanced expressions. For instance consider the ε-approximation of a sum $x_1 + (x_2 + (x_3 + \cdots + (x_r)))$, which corresponds to a tree



Then the above technique would lead to the computation of an $(\varepsilon/2^i)$-approximation of x_i for $i < r$ and an $(\varepsilon/2^{r-1})$-approximation of x_r. If r is large, then $\varepsilon/2^{r-1}$ is unnecessarily small, since a mere (ε/r)-approximation for each x_i would do. In section 5 we will consider a general technique for computing "balanced error estimates".

3. *A posteriori* error estimates

An alternative technique for computing with effective real numbers is interval arithmetic. The idea is to systematically compute intervals approximations instead of floating point approximations. These intervals must be such that the real numbers we are interested in are certified to lie in their respective interval approximations.

More precisely, given $x \in \mathbb{R}$ and $\varepsilon \in \mathbb{D}^>$, an *$\varepsilon$-interval* for x is a closed interval $[\underline{x}, \overline{x}]$ with $\underline{x}, \overline{x} \in \mathbb{D}^>$ and $\overline{x} - \underline{x} < 2\,\varepsilon$. Concretely speaking, we may represent such intervals by their endpoints $\underline{x}$ and $\overline{x}$. Alternatively, if the precisions of $\underline{x}$ and $\overline{x}$ are large, then it may be more efficient to represent the interval by its center $(\underline{x} + \overline{x})/2$ and its radius $(\overline{x} - \underline{x})/2$. Indeed, the exact endpoints of the interval are not that important. Hence, modulo a slight increase of the radius, we may always assume that the radius can be stored in a "single precision dyadic number" $r \in \{0, \ldots, 2^W - 1\}\, 2^{\mathbb{Z}}$, where W is the number of bits in a machine word. This trick allows to reduce the number of multiple precision computations by a factor of two.

Now assume that we want to compute an ε-approximation for $y = f(x_1, \ldots, x_r)$, where $x_1, \ldots, x_r$ are effective real numbers and where f is a continuous function. Assume also that we have reasonable initial δ_i-intervals for the x_i. Then, starting with a low precision $p = W$, we first compute $(\delta_i/2^p)$-intervals $[\underline{x_i}, \overline{x_i}]$ for the x_i. We next evaluate f using interval arithmetic. This yields an interval $[\underline{y}, \overline{y}]$ with

$$f([\underline{x_1}, \overline{x_1}], \ldots, [\underline{x_r}, \overline{x_r}]) \subseteq [\underline{y}, \overline{y}].$$

If $\overline{y} - \underline{y} < 2\,\varepsilon$, then $(\underline{y} + \overline{y})/2$ is an ε-approximation for y. Otherwise, we increase the precisions p and repeat the computations. Under relatively mild assumptions on the way we evaluate f, this procedure will eventually stop, since f is continuous.

Although this technique of *a posteriori* error estimates does solve the problems **P1** and **P2** raised in the previous section, it also induces some new problems. Most importantly, we have lost the fine-grained control over the precisions in intermediate computations during the evaluation of $f(x_1, \ldots, x_r)$. Indeed, we have only control over the overall starting precision p. This disadvantage is reflected in two ways:

P3. It is not clear how to increase p. Ideally speaking, p should be increased in such a way that the computation time of $[\underline{y}, \overline{y}]$ is doubled at each iteration. In that case (see section 4), the overall computation time is bounded by a constant time the computation time w.r.t. the least precision p which leads to an ε-computation for x. Now the problem is that this "overall computation time" of f can be estimated well by hand for elementary operations like $+$, $\times$, exp, etc., but not necessarily for more complicated functions.

P4. Consider the case when, somewhere during the evaluation of f, we need to compute the sum $u + v$ of a very large number u and a very small number v. Assume moreover that u can be approximated very fast, but that the approximation of v requires a lot of time. Since v is very small w.r.t. u, it will then be possible to skip the computation of v, by setting $u + v \approx u$, unless p is very large. However, the above technique of *a posteriori* error estimates does not allow for this optimization.

4. Relaxed evaluation

Relaxed evaluation can be used in order to "combine the good ideas" of sections 2 and 3. The idea is to endow `real_rep` with an extra field `interval best` which corresponds to the best currently known interval approximation of the real number. Moreover, when requesting for a better approximation, we will actually compute a much better approximation, so as to avoid expensive recomputations when we repeatedly ask for slightly better approximations. This anticipating strategy was first introduced in the case of power series, where it leads to important algorithmic gains [vdH97b, vdH02]. In our context, the proposed method is quite different though, because of the problem with carries.

More precisely, assume that we have an r-ary operation f, such that the n-digit approximation of $f(x_1, \ldots, x_r)$ at dyadic numbers with $\leqslant n$ digits has time complexity $T(n)$. The complexity $T(n)$ for an elementary operation is usually a well-understood, regular function (in general, $T(n)/n$ is increasing, etc.). For instance, we have $T(n) \sim \alpha n$ for addition and $T(n) \sim \alpha n^{\lambda_n}$ for multiplication, where $2 \geqslant \lambda_n > 1$ and λ_n decreases slowly from 2 to 1.

Now assume that we have an n-digit approximation of $y = f(x_1, \ldots, x_r)$ and that we request a slightly better approximation. Then we let $n' > n$ be such that $T(n') \approx 2\,T(n)$ and we replace our n-digit approximation by an n'-digit approximation. This strategy has the property that the successive evaluation of $f(x_1, \ldots, x_n)$ at $1, \ldots, n$ digits requires a time $T'(n) \approx T(n_1) + \cdots + T(n_k)$ where $n_1 < \cdots < n_k$ are such that $n_k \geqslant n$ and $T(n_{i+1}) \approx 2\,T(n_i)$ for $i = 1, \ldots, n-1$. Consequently

$$T'(n) \approx T(n_k) + \frac{1}{2} T(n_k) + \frac{1}{4} T(n_k) + \cdots \approx 2\,T(n_k) \approx 4\,T(n_{k-1}) \leqslant 4\,T(n).$$

More generally, the evaluation of $f(x_1, \ldots, x_r)$ at different precisions $n_1, \ldots, n_k$ requires at most four times as much time as the evaluation of $f(x_1, \ldots, x_r)$ at precision $\max\{n_1, \ldots, n_k\}$.

The combination of *a priori* and *a posteriori* error estimates in the relaxed strategy clearly solves problems **P1** and **P4**. Furthermore, at the level of the class `real_rep`, we have a fine-grained control over how to increase the computational precision. Moreover, we have shown how to perform this increase in an efficient way, as a function of $T(n)$. This will avoid expensive recomputations when y occurs many times in a dag. In other words, the relaxed technique also solves problem **P3**.

Let us illustrate the relaxed strategy on the concrete example of the computation of the sine function. We assume that we have implemented a suitable class `interval` for certified arbitrary precision computations with intervals. First of all, `real_rep` now becomes:

```
class real_rep {
protected:
  interval best;
public:
  inline real_rep (): best (interval::fuzzy) {}
  virtual interval approx (const dyadic& tol) = 0;
};
```

Here `interval::fuzzy` stands for the interval $(-\infty, \infty)$. Sines of numbers are represented by instances of the following class:

```
class sin_real_rep: public real_rep {
  real x;
public:
  inline sin_real_rep (const real& x2): x (x2) {
    best= interval (-1, 1); }
  interval approx (const dyadic& tol);
};
```

The approximation algorithm is given by

```
interval sin_real_rep::approx (const dyadic& tol) {
  if (tol < radius (best)) {
     interval xa= x->approx (tol * (1 - DELTA));
     int required_prec= 2 - expo (tol);
     int proposed_prec= next_prec (-expo (radius (best)));
     xa= truncate (xa, max (required_prec, proposed_prec));
     best= sin (xa);
  }
  return best;
}
```

This algorithm needs some explanations. First of all, `DELTA` stands for a small number like 2^{4-W}, where W is the machine word size. We use `DELTA` for countering the effect of rounding errors in the subsequent computations. Next, `expo` stands for the function which gives the exponent of a dyadic number. We have $2^{\operatorname{expo}(x)} > x \geqslant 2^{\operatorname{expo}(x)-1}$ for all $x \in \mathbb{D}$. The function `next_prec` (see below) applied to a precision n computes the precision $n' > n$ such that $T(n') \approx 2\,T(n)$, where $T(n)$ stands for the complexity of the evaluation of the sine function. Now we recall that `xa` may contain an approximation for `x` which is much better than the approximation we need. We therefore truncate the precision of `xa` at the precision we need, before computing the next approximation for the sine of `x`. The function `next_prec` may for instance be implemented as follows:

```
int next_prec (int prec) {
  if (prec <= (16 * WORD_SIZE)) return 1.41 * prec + WORD_SIZE;
  else if (prec <= 256 * WORD_SIZE) return 1.55 * prec;
  else if (prec <= 4096 * WORD_SIZE) return 1.71 * prec;
  else return 2 * prec;
}
```

The different thresholds correspond to the precisions where we use naive multiplication, Karatsuba multiplication and two ranges of F.F.T. multiplication. We finally have the following algorithm for computing sines:

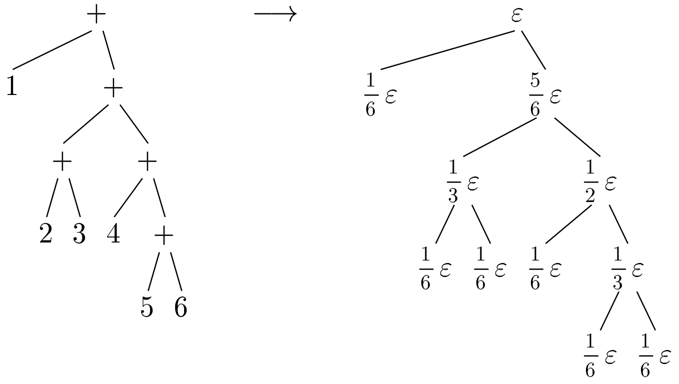
```
inline real sin (const real& x) {
  return new sin_real_rep (x); }
```

5. Balanced error estimates

One of the main problems with *a priori* error estimates that we have mentioned in section 2 are badly balanced expressions, which lead to overly pessimistic bounds. In this section we introduce the general technique of "balanced error estimates" which solve this problem as well as possible in a general purpose system. The idea behind balanced error estimates is to "distribute the required tolerance over the subexpressions in a way which is proportional to their weights". Here leafs have weight 1 and the wt y of an expression $y = f(x_1, \ldots, x_n)$ is given by $\operatorname{wt} y = \operatorname{wt} x_1 + \cdots + \operatorname{wt} x_n$.

Remark 1. We recall that expressions are really dags, so the size of an expression may be exponentially smaller than its weight. Indeed, this can be seen by constructing an expression using repeated squarings `x=x*x`. It is therefore important to represent the weight by a dyadic number with exponent $\leqslant W$, where W is the size of a machine word.

Let us first illustrate the strategy of balanced error estimates in the case of addition. So assume that we want to compute an ε-interval for a sum $x+y$, where x has weight p as an expression and y has weight q. Then we will compute a $\frac{p\,\varepsilon}{p+q}$-interval for x and a $\frac{q\,\varepsilon}{p+q}$-interval for x. For example, in case of the expression $1+((2+3)+(4+(5+6)))$, a tolerance of ε is distributed as follows:



Subtraction is treated in a similar way as addition.

In the case of multiplication, the above additive distribution of tolerances cannot longer be used and one has to resort to "relative tolerances". Here a relative tolerance ρ for x corresponds to an absolute tolerance of $\varepsilon=\rho\,|x|$. Now let x and y be effective real numbers of sizes p and q. Assume for simplicity that we are in the *regular case* when we have good starting approximations $\tilde{x},\tilde{y}\in\mathbb{D}$ for x and y, say of relative tolerances $<2^{-W}$, where W is the machine word size. Then, in order to compute an ε-interval for $x\,y$, we first distribute $\rho=\frac{\varepsilon}{|\tilde{x}\,\tilde{y}|}(1-2^{4-W})$ over $\rho_x=\frac{p\,\rho}{p+q}$ and $\rho_y=\frac{q\,\rho}{p+q}$, and then compute a $(\rho_x\,|x|)$-interval for x and a $(\rho_y\,|y|)$-interval for y. The product of these approximations yields an ε-interval for $x\,y$.

More generally, assume that $x_1,\ldots,x_r$ are effective real numbers of weights $p_1,\ldots,p_r$ and that we want to compute an ε-interval for $y=f(x_1,\ldots,x_r)$, where f is a differentiable r-ary operation for which we are given an arbitrary precision evaluation algorithm at dyadic intervals. Before balancing tolerances as above, we first have to determine the numbers which have to be subdivided (i.e. ε in the case of addition and ρ (really $\rho\,|x|$ and $\rho\,|y|$ as we will see below) in the case of multiplication).

We define the *typical tolerances* for $x_1,\ldots,x_r$ to be the minimal numbers $\tau_1,\ldots,\tau_r$, such that for any intervals $[\underline{x_1},\overline{x_1}],\ldots,[\underline{x_r},\overline{x_r}]$ with

$$f([\underline{x_1},\overline{x_1}],\ldots,[\underline{x_r},\overline{x_r}])\subseteq[y-\varepsilon,y+\varepsilon], \tag{1}$$

we have $\overline{x_i}-\underline{x_i}\leqslant\tau_i$ $(i=1,\ldots,r)$. Let us for simplicity that we are in the *regular case* when there exist (computable) bounds

$$0<\frac{B_i}{2}\leqslant\left|\frac{\partial f}{\partial x_i}(\tilde{x}_1,\ldots,\tilde{x}_r)\right|\leqslant 2\,B_i,$$

which are valid for $\tilde{x}_i\in[x_i-\frac{2\,\varepsilon}{B_i},x_i+\frac{2\,\varepsilon}{B_i}]$ $(i=1,\ldots,r)$. Then (1) implies in particular that $\overline{x_i}-\underline{x_i}\leqslant\frac{4\,\varepsilon}{B_i}$ for $i=1,\ldots,r$. Conversely, $\overline{x_i}-\underline{x_i}\leqslant\frac{\varepsilon}{r\,B_i}$ $(i=1,\ldots,r)$ implies (1). This proves that $\frac{\varepsilon}{2\,r\,B_i}\leqslant\tau_i\leqslant\frac{2\,\varepsilon}{B_i}$. For instance, in the case of addition, we may take $B_1=B_2=1$ and we obtain $\tau_1=\tau_2=1$. In the case of multiplication, we may take $B_1=|\tilde{x}|$ and $B_2=|\tilde{y}|$ and obtain $\tau_1\approx\frac{\varepsilon}{|\tilde{x}|}$ and $\tau_2\approx\frac{\varepsilon}{|\tilde{y}|}$.

Now, assuming that we have sharp lower bounds $\tilde{\tau}_i$ for the τ_i (like $\tilde{\tau}_i = \frac{\varepsilon}{2\,r\,B_i}$), we compute a $\frac{\tilde{\tau}_i\,p_i}{p_1 + \cdots + p_r}$-interval for each x_i. Then we obtain the requested ε-interval for y by evaluating f at these intervals at a sufficient precision.

6. Optimal error estimates and the irregular case

Let us now show that our technique of adaptive error estimates (i.e. the combination of relaxed evaluation and balanced error estimates) is often close to being optimal. In order to make this statement more precise, assume that we have a library of effective real functions f_1, ..., f_l, some of which have arity 0. Then any computation using this library involves only a finite number $\varphi_1, \ldots, \varphi_p$ of expressions constructed using the f_i. These expressions actually correspond to an acyclic graph, since they may share common subexpressions.

Assume that the aim of the computation is to obtain an ε_i-approximation for each φ_i with $1 \leqslant i \leqslant q \leqslant p$. A *certified solution* to this problem consists of assigning a dyadic interval to each node of the acyclic graph, such that

- The interval $[\underline{\varphi_i}, \overline{\varphi_i}]$ associated to each φ_i with $i \leqslant q$ satisfies $\overline{\varphi_i} - \underline{\varphi_i} < 2\,\varepsilon_i$.
- For each subexpression $f_i(\psi_1, \ldots, \psi_r)$ with $r > 0$, the interval associated to f_i is obtained by evaluating f_i at the intervals associated $\psi_1, \ldots, \psi_r$.

The certified solution is said to be *optimal*, if the time needed to compute all interval labels is minimal. Modulo rounding errors, finding such an optimal solution corresponds to determining optimal tolerances for the leafs of the acyclic graph (which is equivalent to determining the lengths of the intervals associated to the leafs).

Remark 2. The above modelization does simplify reality a bit: in practice, some of the expressions φ_1, ..., φ_p may depend themselves on the way we compute things. Also, we may require ε-approximations for the φ_i during the intermediate computations, and in an arbitrary order. Nevertheless, we expect that the above theoretical notion of optimality corresponds quite well to what happens in practice.

Let us now compare the times T^{opt} resp. T we need for the approximation of $\varphi_1, \ldots, \varphi_q$, depending on whether we use an optimal strategy or our strategy of adaptive error estimates. We may decompose $T^{\text{opt}} = \sum_\nu T_\nu^{\text{opt}}$ and $T = \sum_\nu T_\nu$ according to the times which are spent at the computation of each node ν. For each node ν, we denote by $f_\nu = f_{i_\nu}$ the corresponding function and by T_{f_ν} its time complexity. Since we use the relaxed strategy, we have $T_\nu^{\text{opt}} = T_{f_\nu}(k_\nu^{\text{opt}})$ and $T_\nu \leqslant 4\,T_{f_\nu}(k_\nu)$, where k_ν^{opt} and k_ν stand for the (maximal) precision at which we evaluate f_ν, using the respective strategies.

Now let w denote the sum of the weights of the expressions $\varphi_1, \ldots, \varphi_q$. Using structural induction, we observe that the minimal balanced tolerances used at each node of the acyclic graph are at most $O(w)$ times smaller than the tolerances for the optimal solution. In other words, there exists a constant C with $k_\nu \leqslant k_\nu^{\text{opt}} + C \log w$ for all ν. Denoting

$$\lambda = \max_\nu \frac{T_{f_\nu}(k_\nu^{\text{opt}} + C \log w)}{T_{f_\nu}(k_\nu^{\text{opt}})},$$

we obtain $T \leqslant 4\,\lambda\,T^{\text{opt}}$. In particular, if the φ_1, ..., φ_p are fixed and we are interested in obtaining n-digit approximations for φ_1, ..., φ_q, then the regularity hypothesis implies that $k_\nu^{\text{opt}} \sim k_\nu \sim n$ for large n. Consequently, if $T_{f_\nu}(n + \log w) \sim T_{f_\nu}(n)$ for all ν and large n (which is the case for most usual operations like $+$, $\times$, exp, etc.), then $T \leqslant (4 + o(1))\,T^{\text{opt}}$. More generally, if the optimal algorithm essentially spends its time in high precision computations, then one has $T \leqslant 4\,T^{\text{opt}}$.

This analysis shows that the strategy of adaptive error estimates is close to being optimal for high precision computations and in the regular case. In the irregular case, the complexity analysis is more involved and we have only some partial results. It is instructive to well understand the case of multiplication first; the other operations can probably be treated in a similar way.

So assume that we want to compute a product $x y$. Let us first consider the semi-regular case when we still have a good initial approximation for one of the arguments, say x, but not for the other. Then we first compute a $(\frac{\varepsilon}{|\tilde{x}|}(1-2^{4-W}))$-interval for y, with similar notations as before. In case when this yields an approximation $\tilde{y}$ for y with a good relative tolerance, then we may proceed as in the regular case. Otherwise, we obtain at least an upper bound for $|y|$ and we let this upper bound play the role of $|\tilde{y}|$.

This leaves us with the purely irregular case when we neither have strictly positive lower bounds for $|x|$ nor $|y|$. In this case, we have the choice between computing a more precise approximation for x or for y, but it is not clear *a priori* which choice is optimal from the complexity point of view. One solution to this problem may be to continuously improve the precisions of the approximations for both x and y, while distributing the available computation time equally over x and y. It is not clear yet to us how to estimate the complexity of such an algorithm. Another solution, which is easier to implement, is to use rough upper bounds for $|x|$ and $|y|$ instead of $|\tilde{x}|$ and $|\tilde{y}|$, while increasing the precision continuously. However, this strategy is certainly not optimal in some cases.

Bibliography

[BBH01] J. Blanck, V. Brattka, and P. Hertling, editors. *Computability and complexity in analysis*, volume 2064 of *Lect. Notes in Comp. Sc.* Springer, 2001.

[Bla02] J. Blanck. General purpose exact real arithmetic. Technical Report CSR 21-200, Luleå University of Technology, Sweden, 2002. `http://www.sm.luth.se/~jens/`.

[GO04] T. Granlund and Others. GMP, the GNU multiple precision arithmetic library. `http://www.swox.com/gmp`, 1991–2004.

[GPR03] 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.

[HLRZ04] 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`, 2000–2004.

[Mül04] N. Müller. iRRAM, exact arithmetic in C++. `http://www.informatik.uni-trier.de/iRRAM/`, 2000–2004.

[Rev04] N. Revol. MPFI, a multiple precision interval arithmetic library. `http://perso.ens-lyon.fr/nathalie.revol/software.html`, 2001–2004.

[Ric97] D. Richardson. How to recognize zero. *J.S.C.*, 24:627–645, 1997.

[vdH97a] J. van der Hoeven. *Automatic asymptotics*. PhD thesis, École polytechnique, France, 1997.

[vdH97b] J. van der Hoeven. Lazy multiplication of formal power series. In W. W. Küchlin, editor, *Proc. ISSAC '97*, pages 17–20, Maui, Hawaii, July 1997.

[vdH99] J. van der Hoeven. GMPX, a C-extension library for gmp. `http://www.math.u-psud.fr/~vdhoeven/`, 1999. No longer maintained.

[vdH01] J. van der Hoeven. Zero-testing, witness conjectures and differential diophantine approximation. Technical Report 2001-62, Prépublications d'Orsay, 2001.

[vdH02] Joris van der Hoeven. Relax, but don't be too lazy. *JSC*, 34:479–542, 2002.

[vdH04] J. van der Hoeven. Mmxlib, a C++ core library for Mathemagix. `http://www.math.u-psud.fr/~vdhoeven/`, 2003–2004.