Reliable Computing 2 (3) (1996), pp. 299–320

New slope methods for sharper interval functions and a note on Fischer's acceleration method

João B. Oliveira

This paper presents algorithms evaluating sharper bounds for interval functions $F(X) : IR^n \to IR$. We revisit two methods that use partial derivatives of the function, and develop four other inclusion methods using the set of slopes $S_f(x, z)$ of f at $x \in X$ with respect to some $z \in IR^n$. All methods can be implemented using tools that automatically evaluate gradient and slope vectors by using a forward strategy, so the complex management of reverse accumulation methods is avoided. The sharpest methods compute each component of gradients and slopes separately, by substituting each interval variable at a time. Backward methods bring no great advantage in the sharpest algorithms, since object-oriented forward implementations are easy and immediate.

Fischer's acceleration scheme [2] was also tested with interval variables. This method allows the direct evaluation of the product $f'(x) * (x - z)$ as a single real number (instead of working with two vectors) and we used it to compute $F'(X) * (X - z)$ for an interval vector X. We are led to decide against such acceleration when interval variables are involved.

Новые методы наклонов для более точного вычисления интервальных функций и замечание по поводу метода ускорения Фишера

Ж. Оливейра

Представлены алгоритмы для вычисления более точных границ интервальных функций $F(X) : IR^n \to IR$. Заново рассмотрены два метода, использующие частные производные функции, и разработано еще четыре локализационных метода, в которых применяется множество наклонов $S_f(x, z)$ функции f для $x \in X$ по отношению к некоторому $z \in IR^n$. Все методы можно реализовать с помощью средств, которые автоматически вычисляют векторы градиента и наклона с помощью опережающей стратегии, таким образом исключив трудности, связанные с методами обратного накопления. Самые точные из этих методов вычисляют каждую компоненту градиентов и наклонов отдельно, подставляя по одной интервальной переменной за раз. Методы с запаздыванием не дают большого выигрыша в таких алгоритмах, поскольку объектно-ориентированные методы с опережением реализуются с малыми затратами труда и времени.

Схема ускорения Фишера [2] также была протестирована с интервальными переменными. Этот метод делает возможным прямое вычисление произведения $f'(x) * (x - z)$ как одного вещественного числа (а не двух векторов). Мы использовали этот метод для вычисления $F'(X) * (X - z)$ с интервальным вектором X, но пришли к выводу, что этот метод ускорения не применим к интервальным переменным.

Introduction

Slopes and gradients can be used to compute sharper bounds for interval functions, and this is essential to many applications, as in the case of global optimization, where intervals offer a *safe* way of finding all minima of a function without the danger of missing a minimum, and guarantee that the found minima are the *true* minima of the function. The evaluation of sharper bounds for interval functions can considerably accelerate these interval-based algorithms. The methods that allow us to evaluate sharper bounds through the use of gradients and slopes are based on the Mean Value Theorem, and are quite simple in their basic form. We present some variations on these methods, so that more elaborated versions are able to evaluate sharper bounds than a naive interval evaluation, through the use of different evaluation strategies and a clever management of the already available information. First of all, we present slopes and a forward method to evaluate them.

1. Defining slopes

Definition: Slope. *From Neumaier [9], a slope* $S_f(x,z) : R^n \times R^n \to R^n$ *for a continuous function* $f(x) : D \subseteq R^n \to R$ *with respect to a given point* $z \in D$ *is any function for which the relation*

$$f(x) - f(z) = S_f(x,z) * (x - z) \tag{1}$$

holds for all $x \in D$.

For monovariate functions the slope $S_f(x,z)$ can be reduced to a difference quotient when $x \neq z$. For the multivariate case, however, both $(x - z)$ and $S_f(x,z)$ are vectors, and the multiplication turns into a scalar product. In this case slopes are no more uniquely determined, as different vectors $S_f(x,z)$ can be chosen as long as the scalar product remains constant.

For the case of a monovariate function $f \in C^1$, we notice that when x tends to z the definition of $S_f(x,z)$ reduces to the same as the derivative of f, thus allowing us to rewrite the above definition as

$$S_f(x,z) = \begin{cases} \big(f(x) - f(z)\big)/(x - z), & x \neq z, \\ f'(x), & x = z. \end{cases} \tag{2}$$

This is just a matter of convenience, and in reality $S_f(x,z)$ may assume *any* value for $x = z$, as in this case the equation that defines slopes will be always satisfied.

1.1. Evaluating slopes from factorizations

As computations can be represented by a sequence of unary/binary operations, we can define a large class of functions representable as factorizations:

Definition: Factorization, factorable function. *Let* $L \subset C^1$ *be a set of monadic operators used in the definition of a function* $f : D \subseteq R^n \to R$. *The function* f *is said to be computed by*

some sequence $\langle f \rangle = (f_1, \ldots, f_k)$ *of functions* $f_i : R^n \to R$, *with* $1 \le i \le k$ *satisfying*

$$\begin{array}{lcll} f_i(x) & = & \alpha \in R, & \text{or} \\ f_i(x) & = & x_j, \quad 1 \le j \le n, & \text{or} \\ f_i(x) & = & g\big(f_j(x)\big), \quad j < i,\ g \in L, & \text{or} \\ f_i(x) & = & f_{i_1}(x) \circ f_{i_2}(x), \quad i_1, i_2 < i,\ \circ \in \{+, -, *, /\} & \end{array}$$

and if for all $x \in D$ *the values* $f_i(x)$ *are well-defined and* $f(x) = f_k(x)$.

Additionally, the sequence $\langle f \rangle$ is said to be a *factorization* of f, and has k steps. The function f is also said to be a *factorable* function. For vector valued functions $f(x) = (f^1(x), \ldots, f^s(x))$, the definition of a factorable function can be extended in the following way: $f : D \subseteq R^n \to R^s$ is factorable if all component functions $f^k(x)$, $1 \le k \le s$ are factorable.

Being able to evaluate the broad class of factorizable functions, we can present the rules used to evaluate slopes for these functions:

Definition: Slope sequence. *A sequence* $\langle S_f \rangle = (S_{f_1}(x, z), \ldots, S_{f_k}(x, z))$ *is said to be a* slope sequence *corresponding to some sequence* $\langle f \rangle = (f_1, \ldots, f_k)$ *that computes a function* $f : D \subseteq R^n \to R$ *if* $x, z \in D$ *and each element* $S_{f_p}(x, z) \in R^n$ *from* $\langle S_f \rangle$ *is computed according to the following rules:*

$$f_p(x) = \alpha \in R \ \Rightarrow\ S_{f_p}(x, z) = (0, \ldots, 0), \tag{3}$$

$$f_p(x) = x_j \ \Rightarrow\ S_{f_p}(x, z) = e^j = (\delta_{1j}, \ldots, \delta_{nj}), \tag{4}$$

$$f_p(x) = f_i(x) \pm f_j(x) \ \Rightarrow\ S_{f_p}(x, z) = S_{f_i}(x, z) \pm S_{f_j}(x, z), \tag{5}$$

$$f_p(x) = f_i(x) * f_j(x) \ \Rightarrow\ S_{f_p}(x, z) = f_j(x) * S_{f_i}(x, z) + f_i(z) * S_{f_j}(x, z), \tag{6}$$

$$f_p(x) = f_i(x)/f_j(x) \ \Rightarrow\ S_{f_p}(x, z) = \big(S_{f_i}(x, z) - S_{f_j}(x, z) * f_p(z)\big)/f_j(x), \tag{7}$$

$$f_p(x) = g\big(f_j(x)\big) \ \Rightarrow\ S_{f_p}(x, z) = S_g\big(f_j(x), f_j(z)\big) * S_{f_j}(x, z), \qquad g \in L. \tag{8}$$

At the end of the computing process we have the value of $S_f(x, z)$ for initially given values of x and z. Following the rules, arises the necessity of keeping the partial values of $f_p(z)$ as well as former values of $f_p(x)$, and we also need a sequence $\langle f_z \rangle$ to compute $f(z) : D \subseteq R^n \to R$. (As $z \in D$, the sequence f_z is also computable and has as many steps as $f(x)$.)

This set of rules is very similar to the set that evaluates gradients in forward mode [3], and the correctness of the rules can be proved in a similar way. The only difference from the gradient algorithm is a small change in the evaluation rules to consider the values of $f_p(z)$ and the necessity of evaluating and storing $f(z)$: this means the extra effort of having another factorization evaluated, but as $f_p(z)$ is a real number it also assures that the computed values are sharper (the gradient evaluation uses intervals where the slope evaluation uses real values $f_p(z)$). The rules presented above can also be easily generalized to be used with interval variables, and a consequence of this generalization is the possibility of computing sharper interval inclusions, as will be shown in the following section.

Factorizations can be easily extended through the use of interval variables: if a function f has a corresponding factorization $\langle f \rangle$, then if we substitute all real operations in $\langle f \rangle$ by its corresponding interval operations the new sequence (denoted $\langle F \rangle$) will be a factorization of F, thus evaluating an interval extension of f. In the same way, slopes and gradients can be evaluated if the variables and operations being used are interval operations.

2. Gradient and slope inclusions for interval functions

We start with two possibilities that are based on the partial derivatives of the function (in the rest of this paper we call these methods *gradient* methods due to the use of partial derivatives, assuming that gradients are row vectors), and develop others that are based on slopes. The aim of these variations is to sharpen the obtained inclusion within a moderate amount of effort. In this sense, slopes are a very interesting source of variations and different versions for methods already existing, since we are able to vary the evaluation method, the values of z (possibly using more than one value of z), and a few other characteristics.

2.1. The inclusions

The first inclusion is obtained from the Mean Value Theorem. Then a second, possibly sharper method is presented, still using gradients. Thereafter we present methods that use slopes: a classical method and three variations on it. The methods here presented make use of slopes and gradients, but it is not important whether these are calculated in forward or backward mode.

2.1.1. Gradient inclusion I

From the Mean Value Theorem ([1, 8, 9] and references therein) we know that, given a function $f : R^n \to R$ defined and continuously differentiable in $z \cup X$ with $z \in R^n$, $X \subseteq IR^n$, the following relation holds:

$$f(x) \in f(z) + f'(z \cup X) * (x - z) \subseteq f(z) + f'(z \cup X) * (X - z), \quad \text{for all } x \in X.$$

This gradient inclusion would begin by evaluating the row vector $f'(z \cup X)$ and use this vector to obtain an inclusion by performing an scalar product with the column vector $(X - z)$. The evaluation of $f(z)$ is also necessary for the process, but the point is that any method of gradient evaluation could be used here, either forward or backward. In the process of computing a gradient we also evaluate a naive interval evaluation $F(X)$, so that at the end the intersection between this partial result and the final inclusions can be built:

$$f(X) \subseteq F(X) \cap \left(f(z) + f'(z \cup X) * (X - z)\right).$$

2.1.2. Gradient inclusion II

In this second inclusion the same relationship as above is used, but the way we evaluate the gradients changes. In fact we use ideas presented by Hansen [5, 6], in a strategy aimed to reduce the diameter of the final result, even if the effort increases. This method will be stated as a theorem, originated from Hansen:

Theorem 1. *Let $f : D \subseteq R^n \to R$ continuously differentiable in D and $z \in D$. Let $X \in IR^n$, $X \subseteq D$. Then for any arbitrary $x \in X$, $x = (x_1, \ldots, x_n)$ it holds that*

$$f(x) = f(z) + \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}(x_1, \ldots, x_{i-1}, \xi_i, z_{i+1}, \ldots, z_n) * (x_i - z_i)$$

for values $\xi_i \in z_i \cup x_i$.

Proof. Let us define a family of functions $f_i : R \to R$, $1 \leq i \leq n$, given by $f_i(w) := f(x_1, \ldots, x_{i-1}, w, z_{i+1}, \ldots, z_n)$. Then it follows from the definition of the f_i that for some $\xi_i \in z_i \cup x_i$

$$\begin{aligned} f_i(x_i) &= f_i(z_i) + f_i'(\xi_i) * (x_i - z_i) \\ &= f_i(z_i) + \partial f/\partial x_i(x_1, \ldots, x_{i-1}, \xi_i, z_{i+1}, \ldots, z_n) * (x_i - z_i). \end{aligned}$$

From the definition of the functions we also have that $f(z) = f_1(z_1)$ and $f(x) = f_n(x_n)$. We use the inclusion above for $f(x)$ to obtain

$$f(x) = f_n(x_n) = f_n(z_n) + \partial f/\partial x_n(x_1, \ldots, x_{n-1}, \xi_n) * (x_n - z_n).$$

Knowing that in general

$$f_i(x_i) = f_i(z_i) + \partial f/\partial x_i(x_1, \ldots, x_{i-1}, \xi_i, z_{i+1}, \ldots, z_n) * (x_i - z_i)$$

and that $f_i(z_i) = f_{i-1}(x_{i-1})$, it is possible to make successive substitutions starting from the inclusion for $f_n(x_n)$ and getting at the end

$$\begin{aligned} f(x) &= f_1(z_1) + \partial f/\partial x_1(\xi_1, z_2, \ldots, z_n) * (x_1 - z_1) \\ &\quad + \cdots \\ &\quad + \partial f/\partial x_n(x_1, \ldots, x_{n-1}, \xi_n) * (x_n - z_n). \end{aligned}$$

Joining all terms in a sum, we finally have

$$f(x) = f(z) + \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}(x_1, \ldots, x_{i-1}, \xi_i, z_{i+1}, \ldots, z_n) * (x_i - z_i)$$

for values $\xi_i \in z_i \cup x_i$. □

As the estimation $F(X)$ also needs to be computed to perform the gradient evaluation, at the end we may include $f(X)$ using

$$f(X) \subseteq F(X) \cap f(z) + \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}(X_1, \ldots, X_{i-1}, \xi_i, z_{i+1}, \ldots, z_n) * (X_i - z_i)$$

for values $\xi_i \in z_i \cup X_i$.

The strategy behind this algorithm is the following: as the gradient evaluation is now transformed into a sequence of n derivatives, we notice that many (in fact, $n - 1$) of these derivatives have less than n interval variables to be used in its evaluation. That is, the derivatives $\partial f/\partial x_i$ calculated in this way are possibly sharper than the corresponding elements in the vector $f'(X)$, and we are able to evaluate sharper inclusions.

2.1.3. Remarks on gradients

At this point, some remarks about the use of gradients to achieve inclusions can be made, these remarks being valid also for the two methods above.

These algorithms can only be applied if the function to be included is differentiable in $z \cup X$. Although this does not seem to be a severe restriction, it may turn up to a problem when trying to achieve inclusions for functions defined by algorithms. These are relatively complex expressions obtained by running programs, and cannot be easily defined in an explicit

form. It is also possible that at the end of the evaluation algorithm no information about the differentiability of the function exists (after an unknown number of IFs, GOTOs or loops), so this exigence may be too strong in practice.

As we compute the inclusion with the term $\xi_i \in z_i \cup X_i$, the derivatives evaluated are inclusions for all existing derivatives in $z_i \cup X_i$. In other words, we are able to substitute the used z_i by any $\xi_i \in z_i \cup X_i$ and the above inclusions would still be valid. So, the derivatives are intervals with a relatively large diameter, maybe too large to be really useful.

As an extra nuisance, if we need to use some z_i outside X_i, then we still have to compute the gradient for the region $z_i \cup X_i$. This brings two major problems:

1. The diameter turns to be even larger.
2. To evaluate this gradient we have to evaluate the function $F(z_i \cup X_i)$, thereafter evaluating the derivative. Unfortunately the function $F(X_i)$ is no more evaluated, and thus we are not able to perform an effective "error-reducing" intersection as in the methods above.

2.1.4. Slope inclusion I

Similarly to the classical gradient inclusion, the basic slope inclusion is also very simple:

$$f(D) \subseteq f(z) + S_F(D, z) * (D - z). \tag{9}$$

Again, $f(z)$ must be evaluated and a slope $S_F(D, z)$ is necessary to compute the inclusion. Once more, when evaluating the slopes we also compute the common interval evaluation $F(X)$, and at the end they may be intersected, reducing the final diameter. This is all, and in the following methods we will try to enhance the sharpness of the final answer taking different variations on this process.

2.1.5. Slope inclusion II

The slope evaluation depends in subtle ways from the value z used as reference to the slopes, and there remains the question about what are the better choices for z for a given function in a given domain.

This dependance suggests other possibilities to effectively use slopes to evaluate safe bounds for a function $f(X) : X \in IR^n \to IR$. Here we depict one of such variations: as an effective strategy we could use *two* slopes taken with respect to two different values $z_1, z_2 \in R^n$ to compute estimations for the range of $f(X)$. Thereafter, the intersection between such estimates is built, achieving a possibly better value. Natural candidates for such values z_1, z_2 would be $z_1 = \min(X)$ and $z_2 = \max(X)$.

We point out that this idea has cost lower than the costs of the evaluation methods that will be immediately proposed, even if its results are possibly not so sharp. In this sense we have a kind of balance between cost and performance.

2.1.6. Slope inclusion III

To develop this method we use the same functions f_i defined in the proof of the gradient method III. For the moment, suppose that we may evaluate slope functions $S_{f_j}(x_j, z_j) : R \times R \to$

R that will expand $f_j(x_j)$ with respect to z_j in the form below:

$$f_j(x_j) = f_j(z_j) + S_{f_j}(x_j, z_j) * (x_j - z_j).$$

If such slope functions exist, we have the following theorem relative to the evaluation of $f(x)$:

Theorem 2. *If there are functions $S_{f_j}(x_j, z_j) : R \times R \to R$ as above, then for all $x \in X$, $x = (x_1, \ldots, x_n)$ it follows that*

$$f(x) = f(z) + \sum_{i=1}^{n} S_{f_i}(x_i, z_i) * (x_i - z_i).$$

Proof. Also in this case we have the identities $f(x) = f_n(x_n)$ and $f(z) = f_1(z_1)$, as well as $f_j(z_j) = f_{j-1}(x_{j-1})$. Then we may still include $f_j(x_j)$ using the value of $f_{j-1}(x_{j-1})$. This succeeds as follows: the evaluation for $f_j(x_j)$ is given by

$$f_j(x_j) = f_j(z_j) + S_{f_j}(x_j, z_j) * (x_j - z_j)$$

and now we make use of the equality $f_j(z_j) = f_{j-1}(x_{j-1})$ to substitute

$$f_j(z_j) = f_{j-1}(x_{j-1}) = f_{j-1}(z_{j-1}) + S_{f_{j-1}}(x_{j-1}, z_{j-1}) * (x_{j-1} - z_{j-1})$$

in the inclusion for $f_j(x_j)$, getting at the end

$$f_j(x_j) = f_{j-1}(z_{j-1}) + S_{f_{j-1}}(x_{j-1}, z_{j-1}) * (x_{j-1} - z_{j-1}) + S_{f_j}(x_j, z_j) * (x_j - z_j).$$

Starting from $f_n(x_n)$ and substituting successively each $f_j(z_j)$, we obtain the following inclusion:

$$\begin{aligned} f(x) \;=\; & f_1(z_1) + S_{f_1}(x_1, z_1) * (x_1 - z_1) \\ & + \cdots \\ & + S_{f_n}(x_n, z_n) * (x_n - z_n). \end{aligned}$$

Joining all terms in a sum, we finally have

$$f(x) = f(z) + \sum_{i=1}^{n} S_{f_i}(x_i, z_i) * (x_i - z_i). \qquad \square$$

This is the slope version of the gradient method that breaks up the gradient evaluation into smaller pieces to achieve interval inclusions with smaller final diameters. The terms $S_{f_i}(X_i, z_i)$ are evaluated with i interval variables each, and the results are sharper than in other methods where slopes are calculated with the maximum number n of interval variables. Unfortunately, the price to be paid for such sharpness is the impressive amount of computations that are necessary to perform the complete process, since for a function $f(X) : IR^n \to IR$ the effort approaches to $2n$ times the cost of computing the function just once.

A good characteristic of these methods is that the value of $f(X_1, \ldots, X_i, z_{i+1}, \ldots, z_n)$ to be used when evaluating $\frac{\partial f}{\partial x_i}(X_1, \ldots, X_{i-1}, \xi, z_{i+1}, \ldots, z_n)$ or the slope $S_{f_i}(X_i, z_i)$ can be easily updated from $f(X_1, \ldots, X_{i-1}, z_i, \ldots, z_n)$ if we only recompute the intermediate steps depending on X_i.

Although not explicitly said in the proof, the process of substituting real by interval variables does not depend on a predefined or constant sequence. For our proof we used the quite natural sequence $1 \ldots n$, but we are free to substitute the variables in any order.

A subtle consequence of this method is that after substituting the first variable by an interval it is no more true that all slopes in the factorization will be evaluated between some interval X and a real z. It will be sometimes necessary to evaluate the slope between one interval and another interval. To give an example, we will use the following function:

$$f(X) = \sqrt{X_1^2 + X_2^2}.$$

To begin with, we evaluate the function for $z = (z_1, z_2)$, then insert the first interval variable. Denoting by $f^s(X)$ a function with s interval variables, we start by evaluating $f^0(X)$ (that is, the true $f(z)$) and a function $f^1(X)$. Using these two, we evaluate a slope between them to make $f^1(X)$ sharper. Thereafter we evaluate some $f^2(X)$ and use it with $f^1(X)$ to evaluate a slope and make $f^2(X)$ sharper. Performing these operations in the above function, we have a sequence like the following (the slope evaluation between the functions is not shown, to enhance readability):

Factorization	$f^0(X)$	$f^1(X)$
$f_1 = X_1$	z_1	X_1
$f_2 = X_2$	z_2	z_2
$f_3 = f_1 * f_1$	z_1^2	X_1^2
$f_4 = f_2 * f_2$	z_2^2	z_2^2
$f_5 = f_3 + f_4$	$z_1^2 + z_2^2$	$X_1^2 + z_2^2$
$f_6 = \sqrt{f_5}$	$\sqrt{z_1^2 + z_2^2}$	$\sqrt{X_1^2 + z_2^2}$

The point here is that the steps f_1, f_3, f_5, and f_6 of the new function $f^1(X)$ are intervals. This would be not important for f_1 and f_3, since the other variable does not appear in their evaluation, so they do not need to be evaluated in further steps. But f_5 and f_6 need to be reevaluated when we substitute the second interval variable, and thus we will need to compute a slope between the corresponding intervals from $f^1(X)$ and $f^2(X)$. This leads us to the computation of slopes between intervals.

2.1.7. Slope inclusion IV

Up to this point, the approach to the use of slopes was more or less the same for any methods: first evaluate an inclusion for $f(X)$ with interval arithmetic (obtaining, say, $F_I(X)$), then $f(z)$, then a slope and the corresponding inclusion (say, $F_S(X)$) obtained by using it. Thereafter, we join both results in the final inclusion

$$f(X) \in F_I(X) \cap F_S(X).$$

It is interesting to notice that this kind of enhancement can only make the last term of the factorization sharper and no other terms take advantage from the slope–evaluation process, due do the rather intuitive process of first evaluating all steps of the factorization for $f(X)$, then all steps for $f(z)$ and finally computing all steps of the gradient or slope. This is a very

natural implementation separating $f(X)$, $f(z)$ and $S_f(X,z)$, but we can make much better by interleaving these computing steps and make all possible intermediate steps sharper.

To start, we remind the reader of an interesting property of a factorization: If we delete any number of steps from the end of a factorization, the remaining sequence is also a factorization. Conversely, any number of steps from the beginning of a factorization also form a factorization, and we will use this fact to work gradually from the beginning, sharpening the intermediate results as we advance to the end.

This new method works as follows[1]:

- First, create an array $S = \emptyset$ of already evaluated slopes.
- For every step f_i in the factorization, do:
  - Evaluate $f_i(X)$.
  - Evaluate $f_i(z)$.
  - If $S_{f_i}(X,z) \notin S$, then evaluate $S_{f_i}(X,z)$ (using other slope values from S) and insert it in S.
  - After that, do $f_i(X) = f_i(X) \cap \Big(f_i(z) + S_{f_i}(X,z) * (X - z)\Big)$.

This seems to be quite an improvement, and may be used either with forward or backward slope evaluation. As intermediate results are enhanced and their slopes are stored in S for further use, it is clear that the slope for any step is evaluated with operations on at most two other slope vectors, namely the slopes of its operands, that were already evaluated and are ready to use. When the slopes are evaluated in forward form, the method has almost the same performance as the forward slopes method, but a much increased sharpness since all intermediate results are possibly enhanced.

3. An interesting speedup

Here we describe a very interesting idea to have a faster evaluation of both gradients and slopes in forward mode. The method was elegantly described by Fischer [2], but we reproduce here its essence, already adapted to gradients and slopes[2]:

It is not uncommon that in the course of some computation we need to evaluate the product $g(u) * v$, where u, $g(u)$ and v are vectors somehow defined. The typical case to be explored in this section would be the evaluation of an interval valued $f(X)$ as the inclusion below:

$$f(X) \in f(z) + \nabla f(X) * (X - z).$$

In the above case, $u = X$, $g = \nabla f$ and $v = X - z$. Common sense says that we evaluate first $g(u)$, then evaluate v and finally evaluate the scalar product $g(u) * v$.

Both slope and gradient evaluations implement exactly this "common sense" approach. They store whole vectors and operate on them progressively, reaching at the end a final vector

[1] We assume that some function $f(x) : R^n \to R$ has a factorization $\langle f \rangle$ with length k.

[2] In this description we talk mainly about gradients since they produce better examples and more readily understandable situations. The situations described are easily adapted to slopes.

that represents $g(u)$. But let us take a closer look at the rules that specify the computation of gradients from a factorization:

$$\begin{aligned}
f_i(X) = \alpha \in R &\Rightarrow \nabla f_i = (0, \ldots, 0), \\
f_i(X) = X_j &\Rightarrow \nabla f_i = e^j = (\delta_{1j}, \ldots, \delta_{nj}), \\
f_i(X) = g\big(f_j(X)\big) &\Rightarrow \nabla f_i = g'\big(f_j(X)\big) * \nabla f_j, \\
f_i(X) = f_{i_1}(X) \pm f_{i_2}(X) &\Rightarrow \nabla f_i = \nabla f_{i_1} \pm \nabla f_{i_2}, \\
f_i(X) = f_{i_1}(X) * f_{i_2}(X) &\Rightarrow \nabla f_i = f_{i_2}(X) * \nabla f_{i_1} + f_{i_1}(X) * \nabla f_{i_2}, \\
f_i(X) = f_{i_1}(X)/f_{i_2}(X) &\Rightarrow \nabla f_i = \big(\nabla f_{i_1} - f_i(X) * \nabla f_{i_2}\big)/f_{i_2}(X).
\end{aligned}$$

Looking at that set of rules, we are tempted to forget about storing whole vectors and to store and operate between the final products. That is, we have for each stored step $f_i(X)$ the vector $\nabla f_i(X)$. Would it then be possible to replace it by the interval $\nabla f_i(X) * (X - z)$? Possibly yes, if we are able to operate between these values. This approach would save a lot of effort and storage.

Interestingly, we can prove that the rules computing $\nabla f(X)$ are also valid to compute the scalar product $\nabla f(X) * (X - z)$. In other words, the rules are practically the same, the only difference being the rule that defines ∇f_i for steps that introduce the elements X_j. This rule was given as

$$f_i(X) = X_j \Rightarrow \nabla f_i = e^j$$

and now this will need to be changed into

$$f_i(X) = X_j \Rightarrow \nabla f_i = X_j - z_j.$$

Instead of working with unit vectors e^j, we have now the result of the scalar product of these vectors with $X - z$, namely $X_j - z_j$. Being all other rules valid, we have no vectors anymore, computing with a single interval associated to each step all through the computation. Doing this, the problem of evaluating the scalar product turns to be indepedent of the number of variables, depending only on the number of operations.

As a consequence, to work with slopes we need to change the same rule in the same way. The other rules for computation with slopes are immediately valid. The above method is very promising: computations using this method are *really* faster and spare lots of storage, but now we should examine what we are really doing when using this method with real and interval variables.

- Real variables.

  Working with real variables no problems of any type arise, and we may easily compute results much faster and with the same exactness as when operating with whole vectors.

- Interval variables.

  Unfortunately, there is a major problem arising when we have to deal with interval variables. To present it, we give the following reasoning:

  In the original method we used unit vectors e^j (that is, real vectors), that in the course of the computation were transformed into interval vectors. That is, the computation was started with real numbers as vector components, and these real numbers were progressively

operated with the intervals originated as intermediate values from the evaluation of $F(X)$. Thus, we have as many intervals being used in the computation as there are steps in the evaluation of $F(X)$. This is the point: the only intervals used in the computation of the slope vector are the ones associated to $F(X)$. After the gradient or slope vector is evaluated, we multiply it by $(X - z)$, another interval vector, and the inclusion is ready. This acceleration method does things a bit differently:

Adapting the new method we already start from intervals (we start from $X - z$, by using the starting values $X_1 - z_1, \ldots, X_n - z_n$), and operate them among themselves and among the intervals originated from the evaluation of $F(X)$. That is, instead of using the difference $X - z$ only at the end, we use these intervals from the start, inserting new intervals in the process and increasing the chance of enlarging the overestimation of the final result.

3.1. Example

As an example, we take $f(x, y) = (x^2 + y^2)/y$. To evaluate a slope inclusion for $f(x, y)$ with $x = [1, 3]$ and $y = [2, 4]$, we use $z = (3, 4)$. Doing the computation using vectors, we have the factorization below (for the sake of readability we insert a new column in the table, namely the value of $f_i(z) + S_{f_i}(x, z) * (x - z)$ for each step):

Factorization	Value	$f(z)$	$S_f(x, z)$	$f(z) + S_f(x, z) * (x - z)$
$f_1 = x$	$[1, 3]$	3	$(1, 0)$	$[1, 3]$
$f_2 = y$	$[2, 4]$	4	$(0, 1)$	$[2, 4]$
$f_3 = f_1 * f_1$	$[1, 9]$	9	$([4, 6], 0)$	$[-3, 9]$
$f_4 = f_2 * f_2$	$[4, 16]$	16	$(0, [6, 8])$	$[0, 16]$
$f_5 = f_3 + f_4$	$[5, 25]$	25	$([4, 6], [6, 8])$	$[-3, 25]$
$f_6 = f_5/f_2$	$[1.25, 12.5]$	6.25	$([1, 3], [-0.125, 0.875])$	$[-1.5, 6.5]$

This slope vector leads us to the inclusion $[-1.5, 6.5]$. Now we evaluate the forward slope using the new method:

Factorization	Value	$f(z)$	$S_f(x, z) * (x - z)$	$f(z) + S_f(x, z) * (x - z)$
$f_1 = x$	$[1, 3]$	3	$[-2, 0]$	$[1, 3]$
$f_2 = y$	$[2, 4]$	4	$[-2, 0]$	$[2, 4]$
$f_3 = f_1 * f_1$	$[1, 9]$	9	$[-12, 0]$	$[-3, 9]$
$f_4 = f_2 * f_2$	$[4, 16]$	16	$[-16, 0]$	$[0, 16]$
$f_5 = f_3 + f_4$	$[5, 25]$	25	$[-28, 0]$	$[-3, 25]$
$f_6 = f_5/f_2$	$[1.25, 12.5]$	6.25	$[-14, 6.25]$	$[-7.75, 12.5]$

This leads to the inclusion $[-7.75, 12.5]$. We see that both factorizations have the same results up to the step f_5, and then suddenly get different. In this example, the division step leads to the overestimation in the new method. This is a consequence of using intervals since from the start.

4. Implementations

We describe the implementation of the previously presented methods, being able to use them to evaluate inclusions for a continuous, differentiable real function $f(x) : R^n \to R$ in an interval domain $X \subset IR^n$ in the following ways:

Common interval evaluation:

This method simply includes $f(X)$ by substituting the variables $x_1, \ldots, x_n$ in $f(x)$ by its interval correspondents $X_1, \ldots, X_n$ from X. Thereafter, the steps of the factorization are computed and the result is ready. This is the most common form of interval evaluation.

Classic gradient:

This method implements an idea previously presented in Section 2.1.1 (the gradient inclusion I), and using it we obtain an inclusion by expanding the function around z using its gradient to obtain the final inclusion:

$$f(X) \subseteq (f(z) + f'(z \cup X) * (X - z)).$$

Thereafter we are able to build an intersection with the inclusion for $f(z \cup X)$ computed by the common interval evaluation, as this value is obtained as an intermediate result in the process of evaluating the gradient in the domain. More formally, we have a pseudo-code like this:

```
interval gradient_inclusion(function f, vec_interval X, vec_point z)
{
        interval res, tmp, tmp2;
        int i;

        tmp = f(X1 ∪ z1, X2 ∪ z2, ..., Xn ∪ zn);
        tmp2 = gradient(X1 ∪ z1, X2 ∪ z2, ..., Xn ∪ zn);
        res = f(z1, z2, ..., zn);

        for i = 1...n
                res += tmp2i * (Xi − zi);
        return res ∩ tmp;
}
```

The intersection between `res` and `tmp` represents exactly the effort of increasing the sharpness by intersecting an older, possibly not so good inclusion (`tmp`, the result of a common evaluation) and a possibly better one (computed by `res`, the inclusion using a gradient). If this were not tried, we would simply return `res`. The use of this intersection assures that this method evaluates inclusions *at most as large* as the common interval evaluation of $f(z \cup X)$. As this is the first algorithm presented, we use this opportunity to explain the following points:

- When reading these algorithms, we must keep in mind that they were implemented operating on factorizations of functions, *and not on program pieces*. That is, after evaluating f for some set of variables, the intermediate set of values used in its computation is preserved, while it would be lost if f were implemented as a piece of compiled code. Thus, after computing f we may compute its gradient without re-evaluating any values from f, just by "finishing" the gradient computation.

- Of course, it is necessary to include some error handling to return appropriate values or set appropriate flags if some function cannot be evaluated due to a division by zero, etc. This error handling is not shown, as it varies on each application.
- The separation between the evaluation of f and its gradient was made only for clarity purposes, but usually this operation is performed at once by some automatic routine. In other algorithms, similar operations will be done separately with the same purpose.
- Experience shows that if $z \notin X$, then the gradients evaluated are usually large enough to make the estimate computed by the algorithm worse than the intermediate value $f(X)$.

Gradient inclusion II:

This method implements the idea embedded in the first theorem of section 2.1, thus originating the gradient inclusion II:

$$f(X) \subseteq f(z) + \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}(X_1, \ldots, X_{i-1}, z_i \cup X_i, z_{i+1}, \ldots, z_n) * (X_i - z_i).$$

To perform the above calculations, we implement the following pseudo-code:

```
interval gradient_inclusionII(function f, vec_interval X, vec_point z)
{
        interval res, tmp1, tmp2;
        int i;

        res = f(z_1, z_2, ..., z_n);

        for i = 1...n {
              tmp1 = f(X_1, ..., X_{i-1}, X_i ∪ z_i, z_{i+1}, ..., z_n);
              tmp2 = gradient(X_1, ..., X_{i-1}, X_i ∪ z_i, z_{i+1}, ..., z_n);
              res += tmp2 * (X_i - z_i);
              }
        return res;
}
```

The process is similar to the others already presented, but in this implementation we do not perform any intersections with previously obtained estimations. To implement this, we include an intersection as follows:

```
res += tmp1 ∩ (tmp2 * (X_i - z_i));
```

It is important to notice that on each iteration just *one* variable is changed. That is, at each time only the i-th variable will have its value changed from z_i to $z_i \cup X_i$. Thus, we are able to examine the factorization and recompute just that steps that depend of this variable, avoiding the effort of recomputing any steps that do not change value. The same is true for the slope computation in the next method.

Classic slopes:

This method implements the basic slope inclusion from equation (1): we compute the slope for the function $f(X)$ with respect to z, finally obtaining the inclusion as

$$f(X) \subseteq f(z) + S_f(X, z) * (X - z)$$

and thereafter we build an intersection with the common inclusion for $f(X)$. (This value was also an intermediate result from the slope evaluation.)

In this case, the final results must be at least as sharp as the values obtained by the forward gradients, since the set of slopes is always a subset of the gradient, but as before nothing can be said about the quality of the results if $z \notin X$. The pseudo-code would be:

```
interval slope_inclusion(function f, vec_interval X, vec_point z)
{
        interval res, tmp, slp;
        int i;

        tmp = f(X_1, X_2, ..., X_n);
        res = f(z_1, z_2, ..., z_n);
        slp = slope(f, X, z);

        for i = 1...n
                res += slp_i * (X_i - z_i);
        return res ∩ tmp;
}
```

In the case of this routine, we use the factorizations to evaluate $f(X)$ and $f(z)$, also storing its intermediary steps. Thereafter, a simple call to a routine returning the slope vector $S_f(X, z)$ used in the loop to compute the scalar product $S_f(X, z) * (X - z)$. After that we have a final intersection with the common evaluation and the result is ready.

Slope inclusion III–IV:

Here we develop *two* methods, and use the second one as implementation. The differences between them are related to the better use of available information, as will be explained in the text. Informally speaking, the first method will implement the slope inclusion III from Section 2.1, and the second method will adapt it to use all information possible, since that method does not fully use it. This produces slope inclusion IV, the one that has presented the best results.

We implement the idea presented in the second theorem of Section 2.1, namely

$$f(X) \subseteq f(z) + \sum_{i=1}^{n} S_{f_i}(X_i, z_i) * (X_i - z_i).$$

To perform the computation, we could implement the following pseudo-code:

```
interval slope_inclusionIII(function f, vec_interval X, vec_point z)
{
        interval res, tmp;
        vec_interval now, previous;
        int i;

        now = (z_1, z_2, ..., z_n);
        res = f(now);

        for i = 1...n {
```

```
                previous = now;
                now_i = X_i;
                tmp = f(now);
                tmp = slope(f, now, previous);
                res += tmp * (X_i − z_i);
                }
        return res;
}
```

There are two sets of variables, to keep track of the already used interval values. We start with a set of real variables, and as each new X component is used, this new component is included in the now set and we evaluate the slope of the function between the two sets of interval variables. Doing this, we are silently computing the functions $f_i(w)$ that were presented in section 2.1, as well as the slope between $f_i(X_i)$ and $f_{i-1}(X_{i-1})$, as the method requires.

In this method, we recompute in the factorization only the steps that are directly dependent of the i-th variable, that is, the only variable that changed its value. This idea applies both to the evaluation of f as well as its slope.

This was indeed a very clear pseudo-code, since it implements literally the method previously presented. Looking at it more carefully, we notice that the computation of f occurs completely before the computation of its slope, that is, all steps of f are recomputed before any steps of its slope are evaluated. This may lead us to an important quality improvement: interleave the steps from f and from its slope, to get advantages from both with a very little cost. To perform this new idea, we change the code to run like this:

```
interval new_inclusionIV(function f, vec_interval X, vec_point z)
{
        interval res, tmp;
        vec_interval now, previous, base;
        int i, j;

        now = (z_1, z_2, ..., z_n);
        res = f(now);

        for i = 1...n {
                for j = 1...steps base_j = f_j(now);
                previous = now;
                now_i = X_i;

                for j = 1...steps
                        if f_j depends on X_i {
                                tmp = slope(f_j, now, previous);
                                f_j = f_j∩ (base_j + tmp * (X_i − z_i));
                        }
                }
        return value_last_step(f);
}
```

The idea behind is: as we interleave the steps of the function and of its slope (say, in the j-th step), we are immediately able to sharpen the value computed to f_j by using the slope and making the intersection $f_j \cap \mathtt{base}_j + S_{f_j}(\mathtt{now}, \mathtt{previous}) * (X_i - z_i)$. That is, we make intermediate steps sharper, and this will make the following steps also sharper, their slopes are sharper... and so on.

In a formal sense, we are no more considering f_j as a step in a factorization, but as a function itself, and sharpening it with a slope computation, before turning to its next step, f_{j+1}. This increase in sharpness may lead to surprising results when compared to the first implementation presented. Although not so obvious, this version has another important advantage: it runs through the factorization only n times, instead of $2n$ times as for the previous version. This represents a considerable speedup if the factorization is long, as for example factorizations generated by running programs, because we are able to do a sharper job with much less memory swaps. The more important conclusion is that this idea may be applied to all methods that compute inclusions by considering one variable at each time, either with slopes or gradients.

5. Examples

In this example we will use the following function:

$$f(X,Y) = \left((X+3Y)(X-Y) + \frac{X-Y}{X+Y}\right) \times \left(\frac{5X-Y}{2X-Y} - \frac{Y}{Y-X}\right)$$

with the variables being $X = [10.708010, 11.274770]$, $Y = [9.301460, 9.583840]$. When slope methods need a value to be used as z, we used $z = (10.666667, 9.333333)$. Maybe some algebraic simplification could be done reducing the differences among the various inclusion methods, but we evaluate the function in this form since it is probably nearer to the results obtained by using these inclusion methods on functions defined by algorithms as more redundancies and dependences are expected to appear. Testing the methods on the function, we obtained the following results[3]:

Method	Inclusion	Diameter	Effort
Common interval evaluation	[349.0581, 988.8205]	639.7624	51
Gradient inclusion If	[335.2089, 935.1833]	599.9744	265
Gradient inclusion Ib	[335.2089, 954.1338]	618.9249	213
Gradient inclusion II	[481.8828, 767.136]	285.2532	274
Slope inclusion I	[487.8473, 772.0643]	284.217	233
Slope inclusion IV	[484.795, 759.1293]	274.3343	436

Gradient Inclusion I is presented in two versions: If and Ib. This means that the gradient vector was evaluated in forward and backward mode, respectively[4]. As we can easily see, there

[3]The effort measured in the table is the total amount of arithmetical operations used in the computation, automatically computed by the methods.

[4]There are differences in the results obtained from both methods when interval variables are used, and usually the gradient obtained in backward mode is an inclusion for the gradient obtained in the forward version. Once again, a consequence of using interval variables in the computation.

is a great improvement when more advanced methods are used. The table presents the relative diameters of the obtained inclusions for the different methods:

Method	Diameter ratio
Common interval evaluation	2.33
Gradient inclusion If	2.19
Gradient inclusion Ib	2.26
Gradient inclusion II	1.04
Slope inclusion I	1.04
Slope inclusion IV	1

As a second example we use

$$f(X,Y) = 2X\left(9 + 2X + \left(\frac{X}{10} - Y\right)^2\right)\left(\frac{X}{10} + \frac{Y}{X}\right)^2.$$

Using this description of the function and evaluating it for $X = [9.7, 10.4]$, $Y = [8.8, 9.6]$ and $z = (\mathrm{Mid}(X), \mathrm{Mid}(Y))$, the different inclusions give the following results:

Method	Inclusion	Diameter	Effort
Common interval evaluation	[5670.5735, 8935.342]	3264.7685	36
Gradient inclusion If	[5809.2835, 8461.7934]	2652.5099	212
Gradient inclusion Ib	[5724.5959, 8546.4809]	2821.885	164
Gradient inclusion II	[5909.7141, 8361.3628]	2451.6486	214
Slope inclusion I	[5918.6285, 8352.4483]	2433.8198	184
Slope inclusion IV	[6096.3915, 8289.8854]	2193.4939	316

Finally, we try the different inclusions with the function given by

$$f(X,Y) = \frac{2X\,(18.4 - 2XY)\,Y}{2X\,(-9.2 + 2XYY - 2XY)}.$$

The results obtained evaluating the function for $X = [0.25, 1.25]$, $Y = [8.5, 9.2]$ and $z = (\mathrm{Mid}(X), \mathrm{Mid}(Y))$ are given below:

Method	Inclusion	Diameter	Effort
Common interval evaluation	[−53.9108, 165.8344]	219.7452	37
Gradient inclusion If	[−53.9108, 165.8344]	219.7452	211
Gradient inclusion Ib	[−53.9108, 165.8344]	219.7452	157
Gradient inclusion II	[−53.9108, 165.8344]	219.7452	226
Slope inclusion I	[−53.9108, 165.8344]	219.7452	183
Slope inclusion IV	[−21.336, 48.7762]	70.1122	328

Supposing that the function is not determined by a running program but known in advance, the repeated term $2X$ can be elimined from the expression. Evaluating the inclusions with this simplification, we have the intervals below:

Method	Inclusion	Diameter	Effort
Common interval evaluation	$[-10.7822, 33.1669]$	43.949	29
Gradient inclusion If	$[-10.7822, 33.1669]$	43.949	163
Gradient inclusion Ib	$[-10.7822, 33.1669]$	43.949	125
Gradient inclusion II	$[-10.7822, 33.1669]$	43.949	170
Slope inclusion I	$[-10.7822, 33.1648]$	43.9469	143
Slope inclusion IV	$[-4.2672, 13.1263]$	17.3935	248

We think that the extra management and computing effort that are applied in the more complicated approaches using both gradients and slopes are a good price to be paid for the better quality provided by their results.

6. Implementation

The major problem with more sophisticated methods based on introducing an interval variable at a time is that they need to store the whole sequence of evaluation steps, so that all computations can be retraced several times, one for each interval variable. This leds directly to serious management problems, very similar to the ones found in the backward gradient evaluation. In fact, these management problems are so similar that it would be very easy to embed these gradient inclusions in any already developed system that performs reverse accumulation.

For the simplest methods, we can very easily implement forward versions of the inclusions, using for example support systems like Profil/BIAS and others. These systems provide properly rounded interval routines, vector and matrix facilities, as well as an object-oriented environment that allows easy customization over a wide range of Unix and PC platforms. The routines that evaluate slopes were programmed under Profil/BIAS as an extra package (Profil already provides a gradient package)[5].

As an example, the program that evaluates inclusions for the last function above from its gradient or slope is given by the code in Figure 1, and it can be seen that the changes needed to get either gradients or slopes as results are quite small. The data types can be automatically changed with macros, as well as the access functions. The only extra care (in this example) is the separate evaluation of the function at the reference point z, since gradients have no automatic evaluation of this value, while slopes perform the evaluation automatically. So, the value of the function at the development point needs to be separately evaluated in the case of gradients, introducing some extra management. On the other hand, if only the gradient and slope vectors were needed, this evaluation could be left out of the program.

In the BIAS/Profil package used in the above example we implemented the slope computation as presented in the Classic Slopes, that is, a rather straightforward implementation very similar to forward differentiation. As a late development, we also implemented an algorithm

[5]Profil and BIAS are available for anonymous ftp at the site `ti3sun.ti3.tu-harburg.de`.

```
// To be defined below: "GRADIENT" or "SLOPE"
// This automatically adapts the whole code to operate
// with gradients or slopes.

#define GRADIENT

#ifdef GRADIENT
#define TheVector GradientValue
#define INTERVAL_TYPE INTERVAL_GRADIENT
#else
#define TheVector SlopeValue
#define INTERVAL_TYPE INTERVAL_SLOPE
#endif

#include "IntervalSlopes.h"
#include "IntervalGradient.h"

INTERVAL_TYPE f(INTERVAL_VECTOR & v) {
   INTERVAL_TYPE x(v);
   INTERVAL_TYPE aux1 = 2 * x(1) * x(2);
   return ((18.4 - aux1) * x(2)) / (-9.2 + aux1 * x(2) - aux1);
}

void main() {
   INTERVAL_VECTOR x(2);
   x(1) = INTERVAL(0.25, 1.25);
   x(2) = INTERVAL(8.25, 9.2);

   VECTOR z(2);
   z(1) = 0.75;
   z(2) = 8.85;
   SetDevelopmentPoint(z);

   INTERVAL_TYPE y = f(x);

#ifdef GRADIENT
   INTERVAL_VECTOR Iz(2) = z;
   INTERVAL RefValue = FunctionValue(f(Iz));
#else
   INTERVAL RefValue = DevValue(y);
#endif

   cout << "x      = " << x                                  << endl;
   cout << "F(x)   = " << FunctionValue(y)                   << endl;
   cout << "z      = " << z                                  << endl;
   cout << "F(z)   = " << RefValue                           << endl;
   cout << "Vector = " << TheVector(y)                       << endl;
   cout << "Result = " << RefValue + TheVector(y)*(x-z) << endl;
}
```

Figure 1. A simple implementation using slopes or gradients under Profil

that evaluates Slope Method IV **without** having to store the computational graph, that is, in a forward version similar to the one suggested in [9]. This was done in a rather simple and elegant way that allows us to avoid all the memory management problems of a backward approach. As the slope computations are implemented using C++, there would be no changes in the program above if this new version were to be used, the only difference that would be noticed is that the slopes evaluated would be sharper.

To implement this new version we need to associate some more information to each variable. It is important to notice that although the amount of memory needed for each variable markedly increases, even more memory is made free by avoiding to store the sequence of operations, and the development effort to be invested in this algorithm is practically insignificant if compared to the effort of implementing a backward system with all its management. Further still, an object-oriented version of this algorithm can be made in very short time.

6.1. New implementation

To obtain a complete version of the algorithm, we present it substituting of real variables for interval ones from first variable to the last. A slightly changed version using other sequencing is easy to obtain. We associate two vectors to a variable, performing computations with them as each operation is performed with the variables. One of these vectors will contain the function values that would be obtained by inserting a few interval variables in the operation being performed, and the other vector will store the slopes obtained for these partial values. More formally, we have:

Definition: Associated vectors. *Let $X = (X_1, \ldots, X_n) \in IR^n$ and $z = (z_1, \ldots, z_n) \in R^n$ be the values of the variables used in the computation of $S_f(X, z)$, and suppose that the function f has a factorization $\langle f \rangle$. Then we associate vectors $P \in IR^{n+1}$ and $V \in IR^n$ to each evaluation step, defined as follows:*

$$f_l(X) \equiv \gamma \in R \;\Rightarrow\; \begin{cases} P_l \equiv (\gamma, \gamma, \ldots, \gamma); \\ V_l \equiv (0, 0, \ldots, 0); \end{cases}$$

$$f_l(X) \equiv X_j \;\Rightarrow\; \begin{cases} P_l \equiv (\alpha_0, \ldots, \alpha_n), \\ \quad \alpha_i \equiv \begin{cases} z_j, & 0 \le i < j, \\ X_j, & i \ge j; \end{cases} \\ V_l \equiv (\alpha_1, \ldots, \alpha_n), \\ \quad \alpha_i \equiv \delta_{ij}; \end{cases}$$

$$f_l(X) = f_j(X) \circ f_k(X) \;\Rightarrow\; \begin{cases} P_l = (\alpha_0, \ldots, \alpha_n), \\ \quad \alpha_i \equiv \begin{cases} u_0 \circ v_0, & i = 0, \\ (u_i \circ v_i) \cap \left(\alpha_{i-1} + S_i * (X_i - z_i)\right), & i > 0, \\ \quad \text{with } u = P_j, v = P_k, S = V_l; \end{cases} \\ V_l = (\alpha_1, \ldots, \alpha_n), \\ \quad \alpha_i \equiv \begin{cases} u_i \circ v_i, & \circ \in \{+, -\}, \\ t_i * u_i + s_{i-1} * v_i, & \circ = *, \\ (u_i - v_i * r_{i-1})/t_i, & \circ = /, \\ \quad \text{with } u = V_j, v = V_k, s = P_j, t = P_k, r = P_l; \end{cases} \end{cases}$$

$$f_l(X) = g\big(f_j(X)\big) \;\Rightarrow\; \begin{cases} P_l = (\alpha_0, \ldots, \alpha_n), \\ \qquad \alpha_i \equiv \begin{cases} g(u_0), & i = 0, \\ g(u_i) \cap \big(\alpha_{i-1} + S_i * (X_i - z_i)\big), & i > 0, \\ \qquad \textit{with } u = P_j, S = V_l; \end{cases} \\ V_l = (\alpha_1, \ldots, \alpha_n), \\ \qquad \alpha_i \equiv \begin{cases} S_g(r_i, r_{i-1}) * u_i, & i > 0, \\ \qquad \textit{with } r = P_j, u = V_j. \end{cases} \end{cases}$$

The above algorithm works as follows: as each interval variable X_j is used, its associated vector P represents the evaluations that would be obtained by substituting the j-th interval variable, and V stores the slopes between consecutive elements of P. So, V has one element less than P. After each variable is introduced, monadic and dyadic operations are performed accordingly to the known rules, and intersections are built consecutively. For each step, the final value of the inclusion is stored in the last element of P.

It is important to notice that for each step f_l that represents an arithmetical operation the vectors P_l and V_l are computed practically together, since elements of one are necessary to compute elements of the other. At the end of a factorization with k steps, the last element of P_k contains the value of the desired inclusion. Although it may seem to be a rather complicated algorithm, it can be easily programmed.

The memory requirements are clear: to each one of the n variables (and any temporary variable) or arithmetic operation we associate $2n + 1$ intervals. This may be a large memory amount, but probably less than the amount that would be used to store the list of already performed operations, plus intermediary data, pointers and any other necessary information. In any case, the vectors associated to the arithmetical operations are volatile: after some operation is performed and its result is used (either stored in a temporary variable or used as argument in other operation) the vectors can be deleted and so no memory remains allocated to any already performed operation. So, if a routine has 400 operations, 16 variables and 24 temporary variables, then $2 * (400 + 16 + 24)$ vectors will be created during the computation, but the number of stored vectors at a given time is bounded by $2 * (16 + 24)$ plus the maximal number of arithmetical operations used in an expression.

An object-oriented implementation will gracefully allocate memory automatically for each new partial result or new arithmetical operation, and this memory will be immediately deallocated after no more references to it exist, thus automatically using the minimum possible amount of memory, providing a rather elegant implementation of the whole algorithm.

Acknowledgements

The author thanks the financial support of the DAAD (Deutsches Akademisches Austauschdienst), and is deeply indebted to the referees for corrections and many helpful suggestions.

References

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

[2] Fischer, H. *Fast method to compute the scalar product of gradient and given vector.* Computing **41** (1989), pp. 261–265.

[3] Griewank, A. *On automatic differentiation.* In: Iri, M. and Tanabe, K. (eds) "Mathematical Programming: Recent Developments and Applications", Kluwer Academic Publishers, Dordrecht, 1989, pp. 83–108.

[4] Griewank, A. and Corliss, G. (eds) *Automatic differentiation of algorithms: theory, implementation and application.* SIAM Proceeding Series, Philadelphia, 1991.

[5] Hansen, E. *On solving systems of equations using interval arithmetic.* Mathematics of Computing **22** (1968), pp. 374–384.

[6] Hansen, E. *Topics in interval analysis.* Oxford University Press, London, 1969.

[7] Kedem, G. *Automatic differentiation of computer programs.* ACM Trans. on Mathematical Software **6** (2) (1980), pp. 150–165.

[8] Moore, R. E. *Methods and applications of interval analysis.* SIAM Studies in Applied Mathematics, Philadelphia, 1979, pp. 24–31.

[9] Neumaier, A. *Interval methods for systems of equations.* Cambridge University Press, 1989.

Received: October 26, 1995
Revised version: December 13, 1995

Technische Informatik III
TU–Hamburg-Harburg
21073 Hamburg
Germany
E-mail: oliveira@tu-harburg.d400.de