Interval Analysis on Directed Acyclic Graphs for Global Optimization

Hermann Schichl and Arnold Neumaier

Institut für Mathematik, Universität Wien
Strudlhofgasse 4, A-1090 Wien, Austria
email: Hermann.Schichl@esi.ac.at, Arnold.Neumaier@univie.ac.at
WWW: http://www.mat.univie.ac.at/~neum/

November 4, 2003

Abstract.

A directed acyclic graph (DAG) representation of optimization problems represents each variable, each operation, and each constraint in the problem formulation by a node of the DAG, with edges representing the flow of the computation.

Using bounds on ranges of intermediate results, represented as weights on the nodes and a suitable mix of forward and backward evaluation, it is possible to give efficient implementations of interval evaluation and automatic differentiation. It is shown how to combine this with constraint propagation techniques to produce narrower interval derivatives and slopes than those provided by using only interval automatic differentiation preceded by constraint propagation.

The implementation is based on earlier work by Kolev [18] on optimal slopes and by Bliek [6] on backward slope evaluation. Care is taken to ensure that rounding errors are treated correctly.

Interval techniques are presented for computing from the DAG useful redundant constraints, in particular linear underestimators for the objective function, a constraint, or a Lagrangian.

The linear underestimators can be found either by slope computations, or by recursive backward underestimation.

For sufficiently sparse problems the work is proportional to the number of operations in the calculation of the objective function (resp. the Lagrangian).

Keywords: global optimization, directed acyclic graphs, automatic differentiation, constraint propagation, slope, interval analysis

2000 MSC Classification: primary 65G40, secondary 90C26

1 Introduction

Deterministic algorithms for solving factorable global optimization problems [11, 20] usually use branch-and-bound like schemes [2, 12, 17, 24, 27]. The success of such a method heavily relies on the quality of the range estimates computed for the functions involved.

This paper discusses a new representation technique for global optimization problems using **directed acyclic graphs** (DAGs). Traditionally, DAGs have been used in automatic differentiation [6, 14] and in the theory of parallel computing [9]. We will show that the DAG representation of a global optimization problem serves many purposes. In some global optimization algorithms [17] and constraint propagation engines (e.g., `ILOG solver`), the computational trees provided by the parsers of high-level programming language compilers (`FORTRAN 90`, `C++`) are used, in others the parsers of modelling languages like `AMPL` [13] or `GAMS` [8] provide the graph representation of the mathematical problem.

In Sections 2 and 3 we will introduce the special DAGs used in problem representation and talk about different interpretations and simplification, and about the difference to computational trees. Section 4 explains the basic evaluation algorithms for computing function values, ranges, derivatives, and slopes, using the DAG representation of a function.

One of the strengths of the DAG concept is that it is suitable both for efficient evaluation and for performing constraint propagation (CP). This method for solving constraint satisfaction problems (CSPs) and global optimization problems (GLOPs) was first developed in the discrete case [15] and later transferred to the continuous case [7, 10, 16, 28]. The basics of constraint propagation on DAGs are outlined in Section 5.

The results of constraint propagation, especially the ranges of the inner nodes, can be used to improve the ranges of the standard evaluation methods for interval derivatives, and slopes. The principles are outlined in Section 6. For global optimization algorithms not only range estimates are relevant but also relaxations by models which are easier to solve. Section 7 describes methods for generating linear relaxations using the DAG representation. (Second order information such as Hessians, second order slopes, quadratic enclosures and convex quadratic relaxations can also be efficiently computed from the DAG representation. Details will be presented elsewhere.) Finally, in Section 8 we will make some statements about implementation issues and performance.

Our notation follows the notation suggested in [22]. In particular, inequalities between vectors are interpreted component-wise, I denotes the identity matrix, intervals and boxes are written in bold face, and $\operatorname{rad} \boldsymbol{x} = \frac{1}{2}(\overline{x} - \underline{x})$ denotes the radius of a box $\boldsymbol{x} = [\underline{x}, \overline{x}] \in \mathbb{IR}^n$.

2 Directed acyclic graphs

This section is devoted to the definition of the graphs used to represent the global optimization problems. Although we will use the term directed acyclic graph (DAG) throughout this paper to reference the graph structure of the problem representation, the mathematical structure used is actually a bit more specialized. Here we will describe the basic properties of the graphs.

2.1 Definition. A **directed multigraph** $\Gamma = (V, E, f)$ consists of a finite set of vertices (nodes) V, a finite set of edges E, and a mapping $f : E \to V \times V$. For every edge $e \in E$ we define the **source of** e as $s(e) := \mathrm{Pr}_1 \circ f(e)$ and the **target of** e as $t(e) := \mathrm{Pr}_2 \circ f(e)$. An edge e with $s(e) = t(e)$ is called a **loop**. Edges $e, e' \in E$ are called **multiple**, if $f(e) = f(e')$.

For every vertex $v \in V$ we define the set of **in-edges**

$$E_i(v) := \{e \in E \mid t(e) = v\}$$

as the set of all edges, which have v as their target, and the set of **out-edges** analogously as the set

$$E_o(v) := \{e \in E \mid s(e) = v\}$$

of all edges with soure v. The **indegree** of a vertex $v \in V$ is defined as the number of in-edges $\mathit{indeg}(v) = |E_i(v)|$, and the the **outdegree** of v as the number of out-edges $\mathit{outdeg}(v) = |E_o(v)|$.

A vertex $v \in V$ with $\mathit{indeg}(v) = 0$ is called a **(local) source** or **leaf** of the graph, and a vertex $v \in V$ with $\mathit{outdeg}(v) = 0$ is called a **(local) sink** or **root** of the graph.

The termini "root" and "leaf" come from directed trees, special directed graphs, which are usually used to represent expressions or functions in algorithms. They are not usually used in the context of directed graphs.

2.2 Definition. Let $\Gamma = (V, E, f)$ be a directed multigraph. A **directed path** from $v \in V$ to $v' \in V$ is a sequence $\{e_1, \ldots, e_n\}$ of edges with $t(e_i) = s(e_{i+1})$ for $i = 1, \ldots, n-1$, $v = s(e_1)$, and $v' = t(e_n)$. A directed path is called **closed** or a **cycle**, if $v = v'$. The multigraph Γ is called **acyclic** if it does not contain a cycle.

An acyclic graph contains at least one source and at least one sink.

2.3 Definition. A **directed multigraph with ordered edges (DMGoe)** $\Gamma = (V, E, f, \leq)$ is a quadruple such that (V, E, f) is a directed multigraph and $(E, \leq)$ is a linearly ordered set. As subsets of E, the in-edges $E_i(v)$ and out-edges $E_o(v)$ for every vertex become linearly ordered as well.

We will represent the global optimization problems as directed acyclic computational multigraphs with ordered edges (in short DAG), where every vertex corresponds to an elementary operation and every edge represents the computational flow. For later use, we define the relationship between different vertices.

The reasons that we need multigraphs is the fact that expression (e.g. x^x) can take the same input more than once. The ordering of the edges is primarily needed for non-commutative operators like division. However, we will see in Section 8 that this also has a consequence for certain commutative operations.

2.4 Definition. Consider the directed acyclic multigraph $\Gamma = (V, E, f)$. For two edges $v, v \in V$ we say that v is a **parent** of v' if there exists an edge $e \in E$ with $s(e) = v'$ and $t(e) = v$, and then we call v' a **child** of v. Furthermore, v will be named an **ancestor** of v' if there is a directed path from v' to v, and v' is then a **descendant** of v.

Now we have all the notions at hand that we will use to represent the optimization problems.

2.5 Proposition. *For every directed acyclic multigraph $\Gamma = (V, E, f)$ there is a linear order $\preceq$ on V such that for every vertex v and every ancestor v' of v we have $v \preceq v'$.*

3 Representing global optimization problems

In this section we will describe how we represent a global optimization problem as a DAG. In Section 3.1 we will talk about simplifying the representation without changing the mathematical model. Later, in Section 3.2 we will show that DAGs can be used to transfer the mathematical problem to various different structures which are needed by specialized optimization and constraint satisfaction algorithms like ternary structure, semi-separable form, and the like. Also sparsity-issues can be tackled by the reinterpretation method described there.

Consider the factorable optimization problem

$$\begin{aligned} &\min \ f(x) \\ &\text{s.t.} \ \ F(x) \in \boldsymbol{F}. \end{aligned} \tag{1}$$

Since it is factorable, the functions f and F can be expressed by sequences of arithmetic expressions and elementary functions. For every arithmetic operation $\circ$ or elementary function involved we introduce a vertex in the graph. Every constant and variable becomes a leaf. If $f \circ g$ is part of one function, we introduce an edge from g to f. The results of f and F become root nodes, of which the result of f is distinguished as the result of the objective function. So with every vertex we associate an arithmetic operation $\{+, *, /, \hat{}\}$ or elementary function $\{1/, \exp, \log, \sin, \cos, \dots\}$. For every edge $e \in E$ we call the vertex $t(e)$ the **result node** and the vertex $s(e)$ the **argument node**.

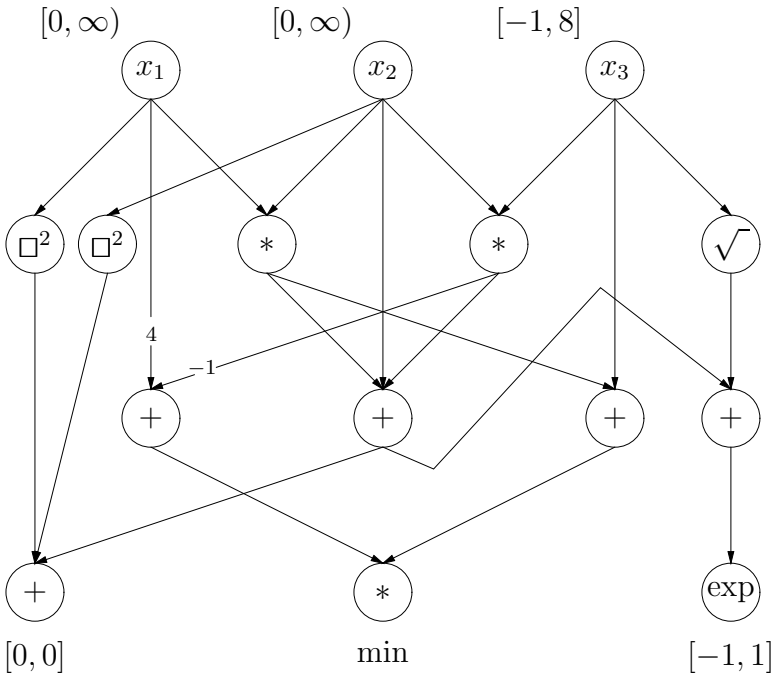

Figure 1: DAG representation of problem 2

When we draw DAG pictures, we write the operation in the interior of the circle representing the node, and mathematically we introduce a map $op : V \to \mathbb{O}$ to the set $\mathbb{O}$ of elementary

operations. We also introduce a mapping $rg : V \to \mathbb{IR}$, the range map, which defines the feasible range of every vertex. In the pictures representing the graphs in this paper, we will write the result of the range map next to every vertex (and leave it out if $r(v) = (-\infty, \infty)$).

Consider for example the optimization problem

$$\begin{aligned} \min \ & (4x_1 - x_2x_3)(x_1x_2 + x_3) \\ \text{s.t.} \ & x_1^2 + x_2^2 + x_1x_2 + x_2x_3 + x_2 = 0 \\ & \exp(x_1x_2 + x_2x_3 + x_2 + \sqrt{x_3}) \in [-1, 1]. \end{aligned} \tag{2}$$

This defines the DAG depicted in Figure 1. Here, we have introduced further notation, the coefficient map $cf : E \to \mathbb{R}$. It multiplies the value of the source of e with $cf(e)$ before feeding it to the operation (or elementary function) $t(e)$. If the coefficient $cf(e)$ is different from 1, we write it over the edge in the picture. In some sense, the DAG in Figure 1 is optimally small, because it contains every subexpression of the functions f and F only once.

3.1 DAG Transformations - Simplification

If we start translating a function to a DAG, we introduce for every variable, every constant, and every operation involved a vertex and connect them by the necessary edges. The resulting DAG, however, usually is too big. Every subexpression of f which appears more than once will be represented by more than one node (e.g. v_1 and v_2). So, the subexpression will be recomputed too often in the evaluation routines, and during constraint propagation (see Secton 5) the algorithms will not make use of the implicit equation $v_1 = v_2$.

Of course, variables usually appear more than once, and many algorithms for constraint propagation [1, 3, 25] use the principle that the variable nodes of identical variables can be identified, hereby reducing the size of the graph. However, this principle can be generalized.

3.1 Definition. Two vertices v_1 and v_2 of the DAG $\Gamma = (V, E, f, \leq)$ are called **simply equivalent** if they represent the same operation or elementary function (i.e. $op(v_1) = op(v_2)$), and there is a monotone increasing bijectiv map $g : E_i(v_1) \to E_i(v_2)$ with the property $s(e) = s(g(e))$ for all $e \in E_i(v_1)$. If there are no distinct simply equivalent vertices in the DAG Γ, we call Γ a **reduced** DAG.

The existence of the map g means nothing else than the fact that v_1 and v_2 represent the same expression. They are the same operation taking the same arguments in the same order. Therefore, any two simply equivalent vertices can be identified without changing the functions represented by Γ.

In particular, every DAG Γ can be transformed to an equivalent reduced DAG. We can start by identifying the equivalent leafs and continue to identify distinct simply equivalent nodes of Γ until all nodes are pairwise simply inequivalent. The resulting DAG Γ' is reduced. Note that this does **not** mean that the graph does not contain any mathematically equivalent subexpressions. This **only** implies that no computationally equivalent subexpressions exist.

These simple graph theoretic transformations can be complemented by additional mathematical transformations. These come in three categories:

Constant Evaluation/Propagation: If all children $v_1, \ldots, v_k$ of a vertex v are leafs representing constants, it can be replaced by a leaf representing the constant which is the result of evaluating the operation $op(v)$ on the children: $v' := \text{const}(op(v)(v_1, \ldots, v_k))$. In a validated computation context, however, you have to make very sure that no roundoff errors are introduced in this step.

Mathematical Equivalences: Typically, properties of elementary functions are used to change the DAG layout. E.g., the rule

$$\log(v_1 \ldots v_k) = \log(v_1) + \cdots + \log(v_k)$$

replaces one log–node and one $*$–node by a $+$–node and a number of log–nodes (or vice versa).

Substitution: Equations of the form

$$-v_0 + v_1 + \cdots + v_k = 0$$

can be used to replace the node v_0 by $v_1 + \cdots + v_k$.

3.2 DAG Interpretation

One strength of the DAG representation is that the mathematical formulation of a problem can be transformed to an equivalent mathematical description which serves the specific needs of some optimization algorithms without having to change the DAG itself; just its **interpretation** is changed.

Consider again problem (2). The following problem is an equivalent formulation

$$\begin{array}{ll} \min & x_{10} \\ \text{s.t.} & x_1^2 + x_2^2 + x_7 = 0 \\ & \exp(x_7 + \sqrt{x_3}) \in [-1, 1] \\ & x_2 x_3 - x_4 = 0 \\ & x_6 + x_3 - x_5 = 0 \\ & x_1 x_2 - x_6 = 0 \\ & x_8 + x_2 - x_7 = 0 \\ & x_4 + x_6 - x_8 = 0 \\ & 4x_1 - x_4 - x_9 = 0 \\ & x_9 x_5 - x_{10} = 0 \end{array} \tag{3}$$

of much higher dimension but with the property that the objective function is linear and that all constraints are **ternary**, i.e. involve at most three variables. This is the required problem formulation for a variety of CP algorithms.

Without changing the DAG we can get this representation just by changing the interpretation of the nodes. All intermediate nodes with more than one child and the objective function node are just regarded as variables, and an equation is added which connects the value of the variable with the value of the node as it is computed from its children. No change of the data structure is necessary.

Adding equations and changing the interpretation of intermediate nodes to variable nodes increases the dimension of the problem but also increases the sparsity. By carefully balancing the number of variables this method can be used, e.g., to optimize the sparsity structure of the Hessian of the Lagrangian.

4 Evaluation

There are several pieces of information which have to be computed for the functions involved in the definition of an optimization problem:

- function values at points,
- function ranges over boxes,
- gradients at points,
- interval gradients over boxes,
- slopes over boxes with fixed center,
- linear enclosures.

To illustrate the techniques, we will use throughout the simple example

$$\begin{array}{ll} \min & f(x_1, x_2, x_3) = (4x_1 - x_2x_3)(x_1x_2 + x_3) \\ \text{s.t.} & x_1 \in [1, 2], \quad x_2 \in [3, 4], \quad x_3 \in [3, 4], \end{array} \tag{4}$$

whose DAG representation can be found in Figure 2.

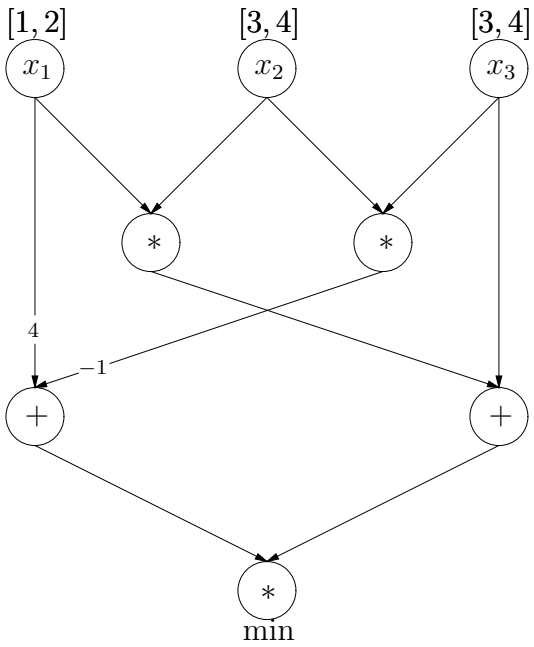

Figure 2: Directed Acyclic Graph representation of (4)

4.1 Forward Evaluation Scheme

The standard method of evaluating expressions works by feeding values to the leafs and propagating these values through the DAG in direction of the edges. This is the reason why this evaluation method is called **forward mode**.

Computing the function value $f(2, 4, 4)$ proceeds as depicted in Figure 3. Here, we have written the results for all nodes to the right of the circle representing them.

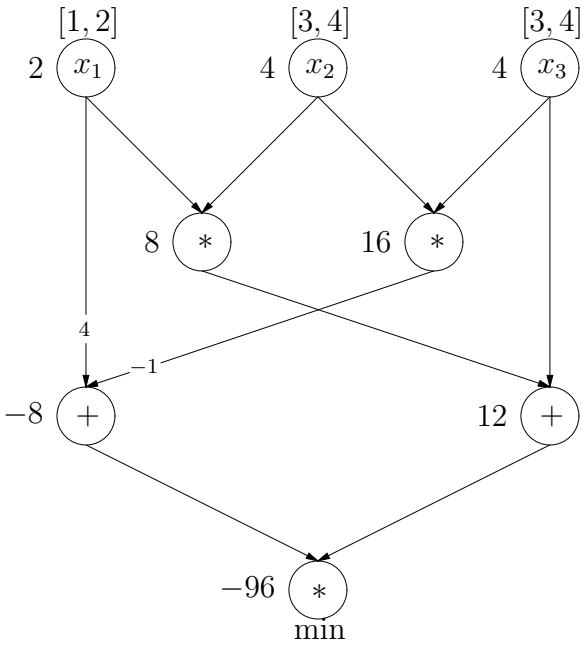

Figure 3: Function evaluation for (4)

In a completely analogous way we can compute a range estimate of f on the initial box $[1, 2] \times [3, 4] \times [3, 4]$. Instead of using real numbers we plug intervals into the leafs and use interval arithmetic instead of real arithmetic and interval extensions of elementary functions instead of their real versions. Again, we show the process in Figure 4 by placing the ranges computed for the nodes next to them

4.2 Backward Evaluation Scheme

Calculating derivatives or slopes could be done by the forward mode as well but then we would need to propagate vectors through the graph, and at every node we would have to perform at least one full vector addition, so the effort to calculate a gradient would be number of variables times the effort of calculating a function value.

However, it is well known from automatic differentiation that the number of operations can be reduced to be of the order of one function evaluation by reversing the direction of evaluation.

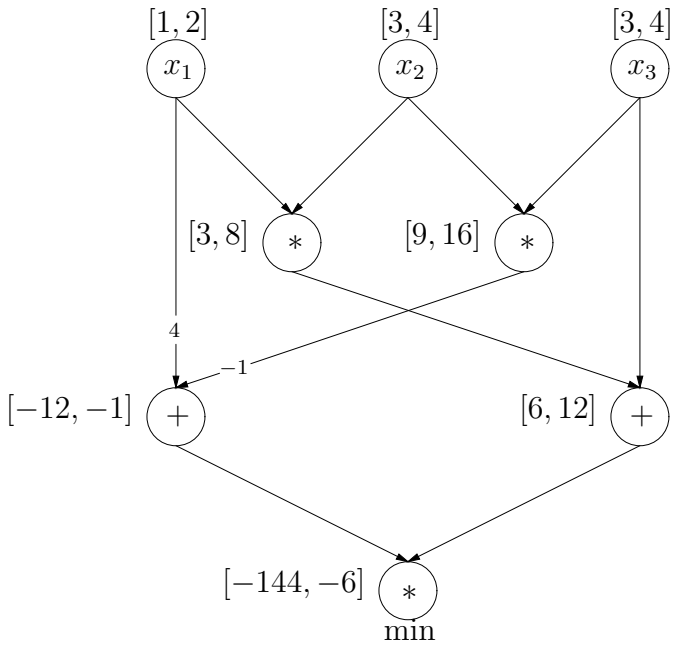

Figure 4: Interval evaluation for (4)

First, we consider the well known fact that the chain rule

$$\frac{\partial}{\partial x_i}(f \circ g)(x) = \sum_k \frac{\partial}{\partial x_k} f(g(x)) \cdot \frac{\partial}{\partial x_i} g(x).$$

holds. So in a first step, during the computation of the function value, we construct a map $dm : E \to \mathbb{R}$ which associates with each edge the value of the partial derivative of the result node with respect to the corresponding argument node. Then we start at the root nodes and walk towards the leafs in the opposite direction of the graph edges, multiplying by $dm(e)$ as we traverse e. When we reach the leaf representing variable x_i, we add the resulting product to the ith component of the gradient vector. The gradient at $(2, 4, 4)$ is calculated as in Figure 5. Here the values of dm are written next to the edges, and the results are next to the nodes. The components of the gradient can be found next to the leafs. We have $\nabla f(2, 4, 4) = (16, -64, -56)$.

There is hardly any difference in computing the interval gradient of f over a given box $\boldsymbol{x}$. Since the chain rule looks exactly the same as for real gradients, the evaluation scheme is the same, as well. We only have to replace real arithmetic by interval arithmetic, and the map $idm : E \to \mathbb{IR}$ becomes interval valued. In Figure 6 we compute $\nabla f(\boldsymbol{x})$ for $\boldsymbol{x} = [1, 2] \times [3, 4] \times [3, 4]$.

A very useful tool for calculating enclosures of the range of f over a box is a **slope**. This is a linear approximation of the form

$$f(x) = f(z) + f[z, x](x - z), \tag{5}$$

see [26, 18]. In one dimension the slope is unique, if it is continuous, and we have

$$f[z, x] = \begin{cases} \dfrac{f(x) - f(z)}{x - z} & x \neq z \\ f'(z) & x = z. \end{cases}$$

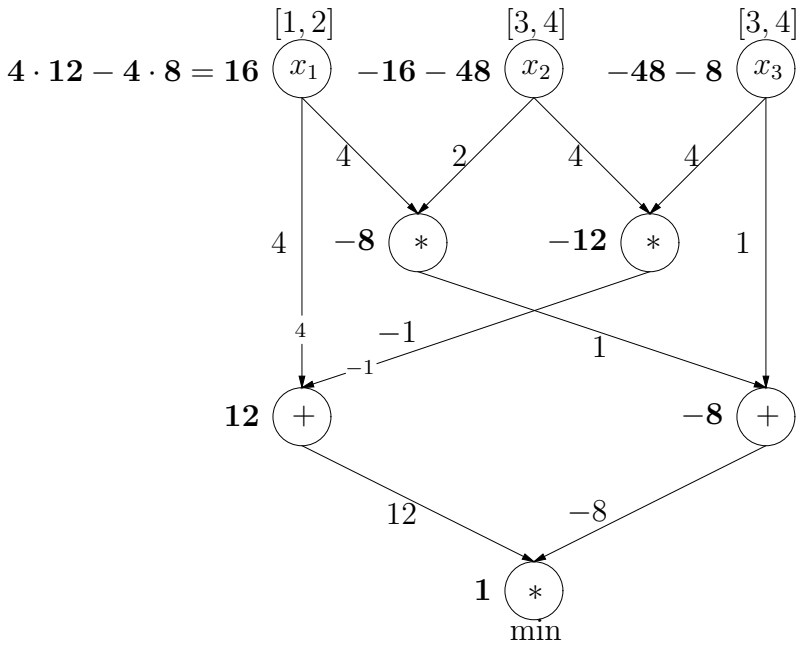

Figure 5: Gradient evaluation for (4)

In higher dimensions the slope is non-unique (see, e.g., Section 8), but it exists always if f is locally Lipschitz.

Using (5) we find an enclosure of the range of f over the box $\boldsymbol{x}$ by

$$f(x) \in f(z) + f[z, \boldsymbol{x}](\boldsymbol{x} - z), \quad \text{for all } x \in \boldsymbol{x}.$$

This is a centered form and has the **quadratic approximation property** (cf. [21]). The most general slope definition is the one with interval center

$$f(\boldsymbol{x}) \subseteq f(\boldsymbol{z}) + f[\boldsymbol{z}, \boldsymbol{x}](\boldsymbol{x} - \boldsymbol{z}),$$

and the special case $\boldsymbol{x} = \boldsymbol{z}$ gives $f[\boldsymbol{z}, \boldsymbol{z}] = f'(\boldsymbol{z})$ the interval derivative. Slopes can be calculated automatically like derivatives, and a chain rule holds:

$$(f \circ g)[\boldsymbol{z}, \boldsymbol{x}] = f[g(\boldsymbol{z}), g(\boldsymbol{x})] \cdot g[\boldsymbol{z}, \boldsymbol{x}]. \tag{6}$$

So, as was noticed by Bliek [6] for computational trees, we can use the backward mode to compute the slopes on the DAG. The arithmetic operations and the elementary functions look like depicted in Figure 7. There z_f denotes the center of f, and s_f the slope of f.

We see from the pictures that for the elementary functions, the slopes $\varphi[z, \boldsymbol{x}]$ have to be computed. It was shown by Kolev [18] that for convex and concave functions the optimal slope is given by

$$\varphi[z, \boldsymbol{x}] = \square\{\varphi[z, \underline{x}], \varphi[z, \overline{x}]\}.$$

For the other functions, the case is more difficult, but we always have

$$\varphi[z, \boldsymbol{x}] \subseteq \varphi'(\boldsymbol{x}).$$

To compute general slopes, we first compute the values of the centers in forward mode, which is an ordinary (interval) function evaluation. Then we change the map dm to $slm : E \to \mathbb{IR}$

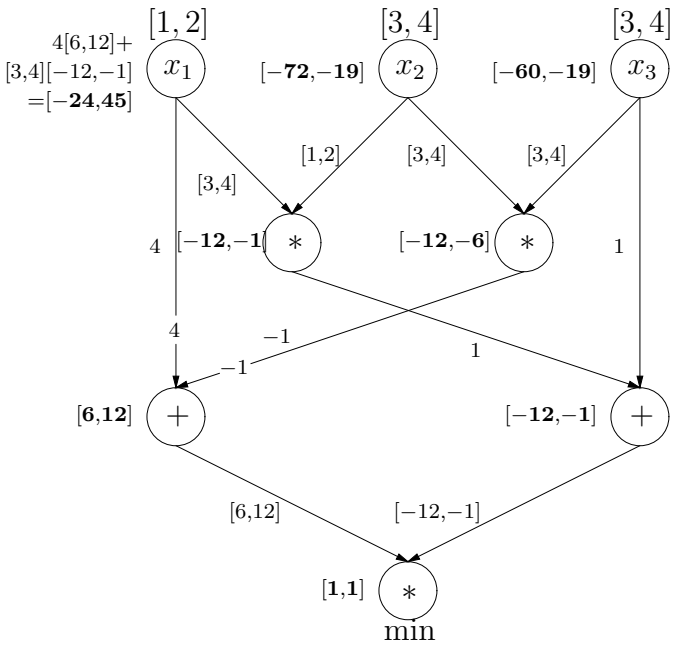

Figure 6: Interval gradient evaluation for (4)

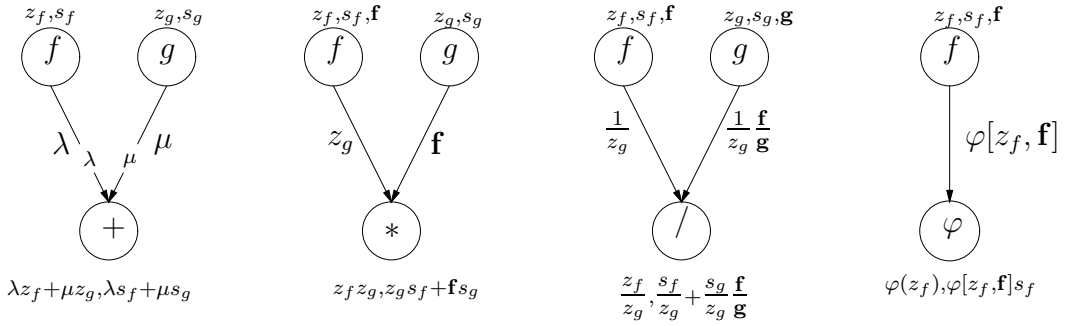

Figure 7: Slopes for elementary operations

storing the slope of the result node with respect to the argument node during the forward pass, and then we use interval arithmetic to compute the slope in backward mode. This can be seen in Figure 8, where we keep at each node the centers and the slopes separated by a comma.

The result $f[z, \boldsymbol{x}] = ([-8, 24], [-64, -34], [-56, -32])$ is clearly slimmer than the interval derivative $f'(\boldsymbol{x}) = ([-24, 45], [-72, -19], [-60, -19])$ as it was expected, since slopes provide better enclosures than interval derivatives.

5 Constraint Propagation on DAGs

As already mentioned, one strength of the DAG concept for global optimization is that knowlege of feasible points and the constraints can be used to narrow the possible ranges of the variables, cf. [3, 25, 28].

If we have a feasible point x_{best} with function value f_{best} we can introduce the new constraint $f(x) \leq f_{\text{best}}$ without changing the solution of the optimization problem (1). Then the ranges

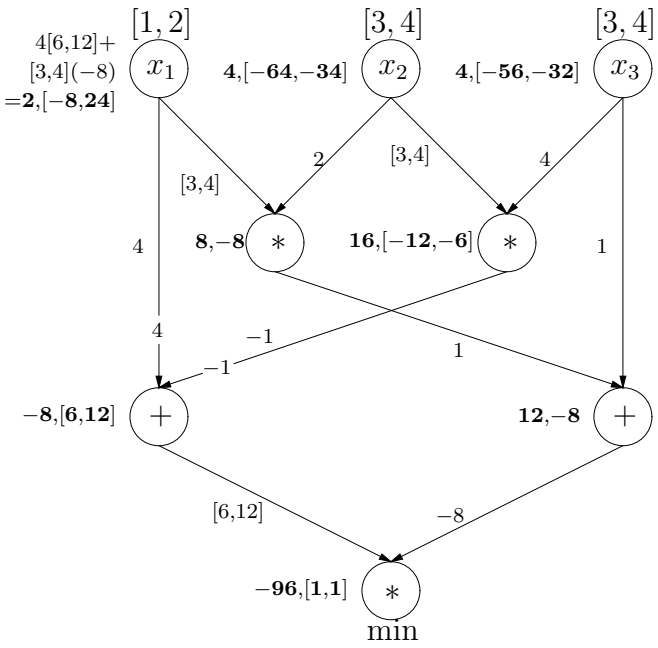

Figure 8: Slope evaluation for (4)

of the nodes can be propagated through the DAG, refining the range map $rg : V \to \mathbb{IR}$ in every step of the constraint propagation (i.e. $rg^{(n+1)}(v) \subseteq rg^{(n)}(v)$ forall v, if $rg^{(n)}$ denotes the range map at step n). We stop when the reductions become too small.

Constraint propagation has two directions, forward and backward. For the elementary functions the propagation steps are as follows.

$\mathbf{h} = \lambda\mathbf{f} + \mu\mathbf{g}$:

forward propagation

$$\boldsymbol{h}^{(n+1)} := (\lambda \boldsymbol{f}^{(n+1)} + \mu \boldsymbol{g}^{(n+1)}) \cap \boldsymbol{h}^{(n)},$$

backward propagation

$$\boldsymbol{f}^{(n+1)} := \tfrac{1}{\lambda}(\boldsymbol{h}^{(n+1)} - \mu \boldsymbol{g}^{(n)}) \cap \boldsymbol{f}^{(n)},$$
$$\boldsymbol{g}^{(n+1)} := \tfrac{1}{\mu}(\boldsymbol{h}^{(n+1)} - \lambda \boldsymbol{f}^{(n)}) \cap \boldsymbol{g}^{(n)}.$$

$\mathbf{h} = \mathbf{fg}$:

forward propagation

$$\boldsymbol{h}^{(n+1)} := (\boldsymbol{f}^{(n+1)} \boldsymbol{g}^{(n+1)}) \cap \boldsymbol{h}^{(n)},$$

backward propagation

$$\boldsymbol{f}^{(n+1)} := (\boldsymbol{h}^{(n+1)} / \boldsymbol{g}^{(n)}) \cap \boldsymbol{f}^{(n)},$$
$$\boldsymbol{g}^{(n+1)} := (\boldsymbol{h}^{(n+1)} / \boldsymbol{f}^{(n)}) \cap \boldsymbol{g}^{(n)}.$$

$\mathbf{h} = \mathbf{f}/\mathbf{g}$:

forward propagation

$$\boldsymbol{h}^{(n+1)} := (f^{(n+1)} / \boldsymbol{g}^{(n+1)}) \cap \boldsymbol{h}^{(n)},$$

backward propagation

$$\boldsymbol{f}^{(n+1)} := (\boldsymbol{h}^{(n+1)} \boldsymbol{g}^{(n)}) \cap \boldsymbol{f}^{(n)},$$
$$\boldsymbol{g}^{(n+1)} := (\boldsymbol{f}^{(n)} / \boldsymbol{h}^{(n+1)}) \cap \boldsymbol{g}^{(n)}.$$

$\mathbf{h} = \varphi(\mathbf{f})$**:**

forward propagation

$$\boldsymbol{h}^{(n+1)} := \varphi(\boldsymbol{f}^{(n+1)}) \cap \boldsymbol{h}^{(n)},$$

backward propagation

$$\boldsymbol{f}^{(n+1)} := \varphi^{-1}(\boldsymbol{h}^{(n+1)}) \cap \boldsymbol{f}^{(n)}.$$

Note that for the DAG representation we refine the range map for **all** nodes not only for the leaf nodes. This is an important step because that will help us in Section 6 to improve the ranges of interval derivatives, slopes, interval Hessians, and second order slopes.

In Figure 9 we show the result of constraint propagation to our example, if we use the function value -96 of the feasible point $(2, 4, 4)$ to introduce the constraint $f(x) \leq -96$. Note that the ranges of the variable nodes do not change, so the traditional method of calculating interval related results is not improved. The new ranges are printed in the picture in bold face.

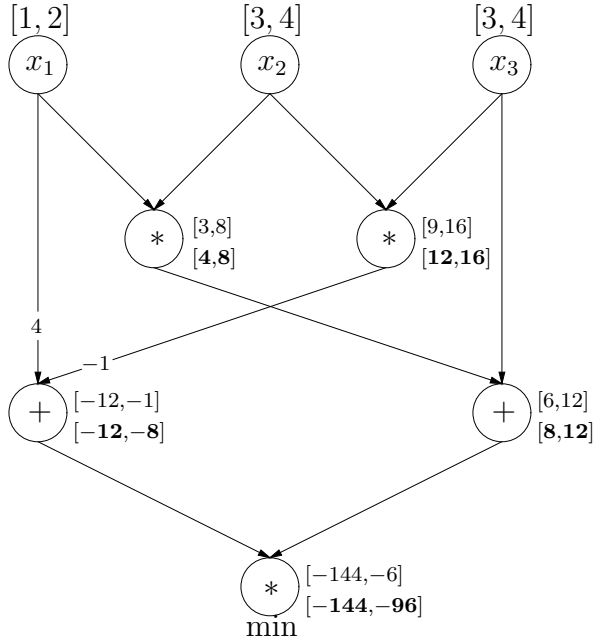

Figure 9: Constraint propagation for (4)

6 Combining CP and Evaluation

In this section we will use the range map $rg : V \to \mathbb{IR}$ improved by constraint propagation to recompute the interval derivative, the slope, and the interval Hessians. This improves the ranges, in some examples tested the improvement was several orders of magnitude.

Figure 10 contains the result of the interval gradient after constraint propagation, and in Figure 11 we recompute the slope.

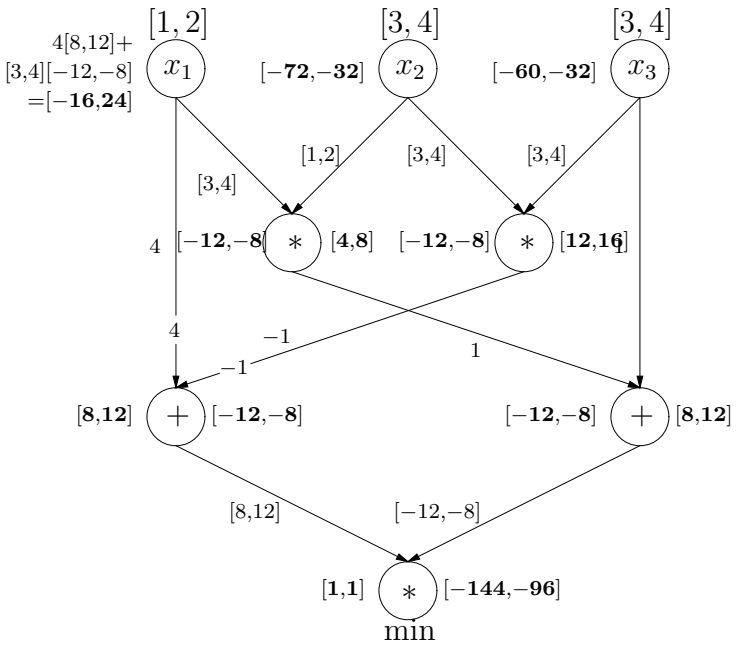

Figure 10: Interval gradient evaluation for (4) after constraint propagation

Both results are clearly an improvement over what we had before:

$$f'(\boldsymbol{x}) \subseteq \begin{pmatrix} [-16,24] \\ [-72,-32] \\ [-60,-32] \end{pmatrix} \subsetneq \begin{pmatrix} [-24,45] \\ [-72,-19] \\ [-60,-19] \end{pmatrix}, \quad f[z,\boldsymbol{x}] \subseteq \begin{pmatrix} [0,24] \\ [-64,-48] \\ [-56,-32] \end{pmatrix} \subsetneq \begin{pmatrix} [-8,24] \\ [-64,-34] \\ [-56,-32] \end{pmatrix}.$$

7 Slopes and linear enclosures

The linear approximation (5) of a function f provided by slopes can be used to construct an enclosure of f by linear functions. This in turn can be used to construct a linear relaxation of the original problem.

7.1 Proposition. *Let $s := f[z,\boldsymbol{x}]$ be a slope of the function $f : \mathbb{R}^n \to \mathbb{R}$. If $z \in \boldsymbol{x}$ then the function*

$$\underline{f}(x) = \underline{f} + \sum \overline{s}_i(\underline{x}_i - z_i) + \frac{\underline{s}_i(\overline{x}_i - z_i) - \overline{s}_i(\underline{x}_i - z_i)}{\overline{x}_i - \underline{x}_i}(x_i - \underline{x}_i)$$

is a linear function which underestimates f on $\boldsymbol{x}$, i.e.,

$$\underline{f}(x) \le f(x) \quad \text{for all } x \in \boldsymbol{x},$$

and the function

$$\overline{f}(x) = \overline{f} + \sum \underline{s}_i(\underline{x}_i - z_i) + \frac{\overline{s}_i(\overline{x}_i - z_i) - \underline{s}_i(\underline{x}_i - z_i)}{\overline{x}_i - \underline{x}_i}(x_i - \underline{x}_i)$$

is a linear overestimating function for f over $\boldsymbol{x}$.

Proof. Everything can be reduced to a series of one dimensional problems, and for those the proof is easy. □

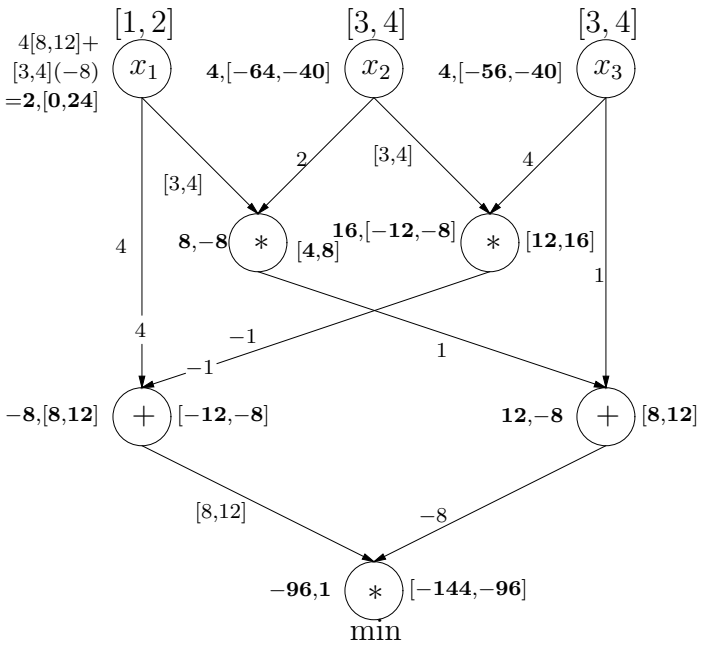

Figure 11: Slope evaluation for (4) after constraint propagation

For problem (1) we have to consider the constraints componentwise. For every component $F_j(x) \in \boldsymbol{F}_j$ the constraints $\underline{F}_j(x) \leq \overline{F}_j$ and $\overline{F}_j(x) \geq \underline{F}_j$ are valid linear constraints. They can be added as redundant constraints to the problem without affecting the solution.

Alternatively, one could also compute the underestimating function $\underline{l}$ for the objective function f. Then the linear program

$$\begin{aligned} \min \ & \underline{f}(x) \\ \text{s.t.} \ & \underline{F}(x) \leq \overline{F} \\ & \overline{f}(x) \geq \underline{F} \\ & x \in \boldsymbol{x}, \end{aligned}$$

where $\underline{F}(x)$ denotes the vector of all underestimating functions $\underline{F}_j$ for all components F_j, is a linear relaxation of (1).

For the example given by (2), we already computed the slope for center $(2, 4, 4)$ in Section 6. Calculating a linear underestimating function for the objective, as above, leads to the constraint

$$-24(x_1 - 2) - 48(x_2 - 4) - 32(x_3 - 4) \leq 0.$$

Performing constraint propagation again on the problem with this additional redundant constraint leads to the domain reduction $x_{2,3} \in [3.4, 4]$. With previously known techniques but without (expensive) higher order consistency, such a reduction would have required a split of the box.

Alternatively, it is possible to construct linear enclosures of the form

$$f(x) \in \boldsymbol{f} + s(x - z), \quad \text{for } x \in \boldsymbol{x},$$

with thin slope $s \in \mathbb{R}^n$ and thick constant term. This approach corresponds to first order Taylor arithmetic as, e.g., presented in [4, 5, 23]. Since linear Taylor expression also obey a chain rule similar to slopes, these enclosures can be computed by backward evaluation with little effort quite similar to "thick" slopes. Kolev [19] showed that propagating them in

forward mode leads to better enclosures; however, the effort for computing in forward mode is n times higher.

8 Implementation Issues

8.1 Multiplication and Division

As has been mentioned earlier, slopes in dimensions greater than one are usually not unique. Two elementary operations, multiplication and division, therefore provide us with a choice for implementation.

All possible slopes for multiplication are

$$x_1 x_2 \in z_1 z_2 + \begin{pmatrix} \lambda \boldsymbol{x}_2 + (1-\lambda) z_2 \\ \lambda z_1 + (1-\lambda) \boldsymbol{x}_1 \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{x}_1 - z_1 \\ \boldsymbol{x}_2 - z_2 \end{pmatrix}$$

for some $\lambda \in \mathbb{R}$ (possibly dependent on the arguments), and for division they are

$$\frac{x_1}{x_2} \in \frac{z_1}{z_2} + \begin{pmatrix} \dfrac{\lambda}{z_2} + \dfrac{1-\lambda}{\boldsymbol{x}_2} \\ -\dfrac{\lambda}{z_2}\dfrac{\boldsymbol{x}_1}{\boldsymbol{x}_2} - \dfrac{1-\lambda}{\boldsymbol{x}_2}\dfrac{z_1}{z_2} \end{pmatrix} \cdot \begin{pmatrix} \boldsymbol{x}_1 - z_1 \\ \boldsymbol{x}_2 - z_2 \end{pmatrix}.$$

The best choice for division is $\lambda = 1$, because we can use the term $\frac{\boldsymbol{x}_1}{\boldsymbol{x}_2}$ after constraint propagation, which is the range enclosure of the division node, for which the slope is being computed, and in addition there is no subdistributivity problem during slope backward evaluation. So the proper choice for division is

$$s_/ = \frac{1}{z_2} \begin{pmatrix} 1 \\ -\dfrac{\boldsymbol{x}_1}{\boldsymbol{x}_2} \end{pmatrix}.$$

For multiplication we can choose λ such that it minimizes the width of the resulting range. A short computation shows that the minimal width is produced for

$$\lambda = \begin{cases} 0 & \text{if } \operatorname{rad}(\boldsymbol{x}_1)|z_2| \le \operatorname{rad}(\boldsymbol{x}_2)|z_1|, \\ 1 & \text{otherwise.} \end{cases}$$

To avoid a case distinction in computing products, it is advisible to find a good heuristics. Considering the Horner scheme for polynomial evaluation gives the following hint: Sort the product by ascending complexity of the factors (i.e., roughly, by increasing overestimation). Then set $\lambda = 0$, hence choose the slope

$$s_* = \begin{pmatrix} z_2 \\ \boldsymbol{x}_1 \end{pmatrix}.$$

8.2 Rounding errors

Since enclosures of the form

$$f(x) \in f(z) + f[z, \boldsymbol{x}](\boldsymbol{x} - z),$$

are computed numerically, the direct evaluation of the thin term $f(z)$ generally does not produce guaranteed enclosures. Hence, it is important to take care for the rounding errors, in order to avoid the occasional loss of solutions in a branch and bound scheme. There are two possible approaches.

The first possibility is to change all calculations involving the center into interval operations, providing a linear interval enclosure

$$f(x) \in \boldsymbol{f}(\boldsymbol{z}) + f[\boldsymbol{z}, \boldsymbol{x}](\boldsymbol{x} - \boldsymbol{z})$$

with generally thick center $\boldsymbol{z}$. This needs slopes of the form $f[\boldsymbol{z}, \boldsymbol{x}]$ with $\boldsymbol{z} \subseteq \boldsymbol{x}$ for all elementary operations.

The second possibility is to allow approximate point evaluations at the centers and elementary slopes with point centers $f[z, \boldsymbol{x}]$, but to take care of the rounding errors in computing $f(z)$ during propagation, by adapting the chain rule appropriately. If

$$\begin{aligned} f(x) &\in \boldsymbol{f} + f[z_f, \boldsymbol{x}](\boldsymbol{x} - z_f), \quad f(z_f) \in \boldsymbol{f},\ x \in \boldsymbol{x} \\ g(y) &\in \boldsymbol{g} + g[z_g, \boldsymbol{y}](\boldsymbol{y} - z_g), \quad g(z_g) \in \boldsymbol{g},\ y \in \boldsymbol{y}, \end{aligned}$$

then, for arbitrary $z_g \approx f(z_f)$,

$$\begin{aligned} g(f(x)) &\in \boldsymbol{g} + g[z_g, f(\boldsymbol{x})](\boldsymbol{f} + f[z_f, \boldsymbol{x}](\boldsymbol{x} - z_f) - z_g) \\ &\subseteq \boldsymbol{g} + g[z_g, f(\boldsymbol{x})](\boldsymbol{f} - z_g) + g[z_g, f(\boldsymbol{x})]f[z_f, \boldsymbol{x}](\boldsymbol{x} - z_f). \end{aligned}$$

The remaining decision is what to compute in forward, and what in backward mode. Taking a third component provides the important hint:

$$h(t) \in \boldsymbol{h} + h[z_h, \boldsymbol{t}](\boldsymbol{t} - z_h),$$

and we find

$$\begin{aligned} h(g(f(x))) \in \boldsymbol{h} &+ h[z_h, g(f(\boldsymbol{x}))](\boldsymbol{g} - z_h + g[z_g, f(\boldsymbol{x})](\boldsymbol{f} - z_g)) \\ &+ h[z_h, g(f(\boldsymbol{x}))]g[z_g, f(\boldsymbol{x})]f[z_f, \boldsymbol{x}](\boldsymbol{x} - z_f) \end{aligned}$$

if the center term is computed in forward mode. If it is computed backward, the term is

$$h[z_h, g(f(\boldsymbol{x}))](g(f(\boldsymbol{x})) - z_h) + h[z_h, g(f(\boldsymbol{x}))]g[z_g, f(\boldsymbol{x})](\boldsymbol{f} - z_g).$$

Because of subdistributivity, this is a worse (or identical) enclosure of the center. Therefore, computing the center in forward mode gives generally tighter result.

References

[1] *ILOG Solver 5.1*, 2001.

[2] I.P. Androulakis, C.D. Maranas, and C.A. Floudas. αBB: a global optimization method for general constrained nonconvex problems. *J. Global Optim.*, 7:337–363, 1995.

[3] F. Benhamou and W. Older. Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*, 1997.

[4] M. Berz and K. Makino. Verified integration of odes and flows using differential algebraic methods on high-order taylor models. *Reliable Computing*, 4:361–369, 1998.

[5] Martin Berz. COSY INFINITY version 8 reference manual. Technical report, National Superconducting Cyclotron Lab., Michigan State University, East Lansing, Mich., 1997. MSUCL–1008.

[6] C. Bliek. *Computer methods for design automation*. PhD thesis, Dept. of Ocean Engineering, Massachusetts Institute of Technology, 1992.

[7] C. Bliek, P. Spellucci, L.N. Vicente, A. Neumaier, L. Granvilliers, E. Monfroy, F. Benhamouand, E. Huens, P. Van Hentenryck, D. Sam-Haroud, and B. Faltings. Algorithms for Solving Nonlinear Constrained and Optimization Problems: The State of the Art. Report of the European Community funded project COCONUT, Mathematisches Institut der Universität Wien, http://www.mat.univie.ac.at/∼neum/glopt/coconut/StArt.html, 2001.

[8] Anthony Brooke, David Kendrick, and Alexander Meeraus. *GAMS - A User's Guide (Release 2.25)*. Boyd & Fraser Publishing Company, Danvers, Massachusetts, 1992.

[9] Chandra Chekuri, Richard Johnson, Rajeev Motwani, B. Natarajan, B. Ramakrishna Rau, and Michael S. Schlansker. Profile-driven instruction level parallel scheduling with application to super blocks. In *International Symposium on Microarchitecture*, pages 58–67, 1996.

[10] S. Dallwig, A. Neumaier, and H. Schichl. GLOPT - A Program for Constrained Global Optimization. In I. M. Bomze, T. Csendes, R. Horst, and P.M. Pardalos, editors, *Developments in Global Optimization*, pages 19–36. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997.

[11] L.C.W. Dixon and G.P. Szegö. *Towards global optimization*. Elsevier, New York, 1975.

[12] C. A. Floudas, Deterministic Global Optimization: Theory, Algorithms and Applications, Kluwer, Dordrecht 1999.

[13] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL — A Mathematical Programming Language*. Thomson, second edition, 2003.

[14] A. Griewank and G. F. Corliss. *Automatic Differentiation of Algorithms*. SIAM Publications, Philadelphia, 1991.

[15] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[16] R. B. Kearfott. Decomposition of Arithmetic Expressions to Improve the Behavior of Interval Iteration for Nonlinear Systems, *Computing*, 47:169-191, 1991.

[17] R. B. Kearfott. *Rigorous Global Search: Continuous Problems*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1996.

[18] L.V. Kolev. Use of interval slopes for the irrational part of factorable functions. *Reliable Computing*, 3:83–93, 1997.

[19] L.V. Kolev An improved interval linearization for solving non-linear problems, Manuscript (2002)

[20] G. P. McCormick. Computability of global solutions to factorable nonconvex programs: Part I – Convex underestimating problems *Math. Programming*, 10:147–175, 1976.

[21] A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge Univ. Press, Cambridge, 1990.

[22] A. Neumaier. *Introduction to Numerical Analysis*. Cambridge Univ. Press, Cambridge, 2001.

[23] A. Neumaier. Taylor forms - use and limits. *Reliable Computing*, 9:43–79, 2002.

[24] N.V. Sahinidis. BARON: A general purpose global optimization software package. *J. Global Optim.*, 8:201–205, 1996.

[25] D. Sam-Haroud and B. Faltings. Consistency techniques for continuous constraints. *Constraints*, 1(1&2):85–118, Sep 1996.

[26] Z. Shen and A. Neumaier. The krawczyk operator and kantorovich's theorem. *J. Math. Anal. Appl.*, 149:437–443, 1990.

[27] M. Tawarmalani. and N.V. Sahinidis, Convexification and Global Optimization in Continuous and Mixed-Integer Nonlinear Programming: Theory, Algorithms, Software, and Applications, Kluwer, Dordrecht 2002.

[28] P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica - A Modeling Language for Global Optimization*. MIT Press, Cambridge, MA, 1997.