myjournal manuscript No.
(will be inserted by the editor)

Bound constrained interval global optimization in the COCONUT Environment

Mihály Csaba Markót, Hermann Schichl*

Faculty of Mathematics, University of Vienna, Nordbergstr. 15, 1090 Vienna, Austria

Received: date / Revised version: date

Abstract We introduce a new interval global optimization method for solving bound constrained problems. The method originates from a small standalone software and is implemented in the COCONUT Environment, a framework designed for the development of complex algorithms, containing numerous state-of-the-art methods in a common software platform. The original algorithm is enhanced by various new methods implemented in COCONUT, regarding both interval function evaluations (such as first and second order derivatives with backward automatic differentiation, slopes, slopes of derivatives, bicentered forms, evaluations on the Karush-John conditions, etc.) and algorithmic elements (inclusion/exclusion boxes, local search, constraint propagation). This resulted in a substantial performance increase as compared to the original code. During the selection of the best combination of options, we performed comparison tests that gave empirical answers to long-lasting algorithmic questions (such as whether to use interval gradients or use slopes instead), that have never been studied numerically in such detail before. The new algorithm, called coco_gop_ex, was tested against the prestigious BARON software on an extensive set of bound constrained problems. We found that in addition to accepting a wider class of bound constrained problems and providing more output information (by locating all global minimizers), coco_gop_ex is competitive with BARON in terms of the solution success rates (with the exception of a set of nonlinear least squares problems), and it often outperforms BARON in running time. In particular, coco_gop_ex was around 21% faster on average over the set of problems solved by both software systems.

* This research was supported by the Austrian Science Found (FWF) Grants Nr. P18704-N13 and P22239-N13.

Keywords. global optimization, bound constrained optimization, interval arithmetic, branch–and–bound

1 Introduction

GOP_ex is a bound constrained general purpose interval branch–and–bound (B&B) algorithm for global optimization (GO). It is originating back to the GOP_ex sample program of the C–XSC Toolbox for Verified Computing [8], further improved by M. C. Markót since 1997 and used successfully in various scientific studies (see, e.g. [13–15]). The COCONUT Environment [4], developed under the leadership of H. Schichl, is an open-source environment for developing and testing algorithms for global optimization. The COCONUT Environment is highly modular, that is, it provides numerous commercial and open-source solver components (inference engines), as building blocks of complex algorithms. Since the GOP_ex code has the main disadvantage that it is hard to extend and hard to interface with other interval related methods for global optimization, it is a straightforward idea to implement the GOP_ex algorithm as a new solver in the COCONUT framework and extend it with novel tools available in COCONUT. The paper introduces this implementation and improvement process.

The implementation of GOP_ex within COCONUT thus has two goals: on the one hand, it greatly facilitates the improvement of the original GOP_ex algorithm, and on the other hand, it demonstrates how COCONUT can promote the development of high-quality solvers using novel components.

The paper is organized as follows: in Section 2 we introduce the GOP_ex algorithm and discuss its components. Section 3 gives a brief introduction to the COCONUT Environment. In Section 4 we discuss the first group of improvements on the solver: advanced methods for evaluating the objective and its higher order derivatives, and methods for tightening the respective range enclosures. The comparison studies of the cost and efficiency of the different routines (like forward vs. backward AD for gradients, slopes vs. interval derivatives) and the discussion on the possible trade-offs between them is presented in Section 5. In Section 6, the second group of improvements, namely, newly added algorithmic tools are discussed, also one-by-one. Our new solver is based on the proposed best combination of the new evaluation and algorithmic methods. This is compared with the BARON solver in Section 7.

Notation and problem setting. The set of real compact intervals is denoted by $\mathbb{I}$. Intervals and interval vectors (boxes) are denoted in boldface. The lower bound, the upper bound, the midpoint, the radius, and the width of an interval $\boldsymbol{x} \in \mathbb{I}$ are denoted by $\inf(\boldsymbol{x})$, $\sup(\boldsymbol{x})$, $\mathrm{mid}(\boldsymbol{x})$, $\mathrm{rad}(\boldsymbol{x})$, and $\mathrm{wid}(\boldsymbol{x})$, resp. The interior of $\boldsymbol{x}$ is denoted by $\mathrm{int}(\boldsymbol{x})$. For interval vectors the above functions are all defined componentwise.

We solve bound constrained global optimization problems of the form

$$\begin{aligned} &\min\ f(x), \\ &\text{s.t.}\ \ x \in \boldsymbol{x}_0, \end{aligned} \tag{1}$$

where $\boldsymbol{x}_0 \in \mathbb{I}^n$ is the search box, and the objective function $f : \mathbb{R}^n \to \mathbb{R}$ is twice continuously differentiable on $\boldsymbol{x}_0$.

Our goal is to provide *mathematically rigorous* interval enclosures of *all* global minimizers and the global minimum, i.e., provide enclosures that are correct even in the presence of the rounding errors caused by the computer arithmetic. This is achieved by tools using *interval arithmetic* (IA). For details on the theory and practice of interval calculations, we refer to textbooks like [17,19].

The range of a function f over a box $\boldsymbol{x}$ is denoted by $\mathrm{rg}(f, \boldsymbol{x})$. Throughout the paper, $\boldsymbol{f}(\boldsymbol{x}) \in \mathbb{I}$ denotes an *interval enclosure* of f over $\boldsymbol{x}$; that is, by definition, $\mathrm{rg}(f, \boldsymbol{x}) \subseteq \boldsymbol{f}(\boldsymbol{x})$. Except of some special cases, usually $\mathrm{rg}(f, \boldsymbol{x}) \subsetneq \boldsymbol{f}(\boldsymbol{x})$ holds; that is, the constructed interval enclosure *overestimates* the real range. We primarily use $\boldsymbol{f}(\boldsymbol{x})$ as a result of a single interval evaluation method, e.g., the application of naive interval arithmetic. Equivalently, $\boldsymbol{f}(\boldsymbol{x})$ also denotes the updated enclosure during the execution of the algorithm, that is, the intersection of enclosures obtained by different evaluation methods. The interval enclosures of the gradient and the Hessian are denoted in a similar fashion by $\nabla \boldsymbol{f}(\boldsymbol{x})$ and $\nabla^2 \boldsymbol{f}(\boldsymbol{x})$, resp.

The interval evaluation on a point c is denoted by $\boldsymbol{f}(c)$. Of course, in practice, such an evaluation is done by expanding c to an interval $\boldsymbol{c}$, but we prefer the notation $\boldsymbol{f}(c)$ to emphasize that the argument is actually a point.

2 The GOP_ex algorithm

We begin by specifying a pseudo-code of the algorithm together with a high level description of the algorithm. The in-depth details of the function and derivative evaluations, the management of the B&B search tree, the box subdivision, the accelerating tests, and the stopping criterion are given in the subsequent Sections 2.2 to 2.6.

2.1 High level description

The pseudo-code of GOP_ex is given by Algorithms 1 and 2. During the algorithm we manage a list $\mathcal{L}$ of working boxes waiting for further processing – these correspond to the open leaves of the B&B search tree. $\mathcal{L}$ is initialized with the search box $\boldsymbol{x}_0$ (Alg. 1, Step 1). In Step 2, we initialize $\tilde{f}$, the current best-known upper bound for the global minimum, to ∞.

Steps 3 to 22 of Alg. 1 contain the main B&B loop. We start by picking a box from $\mathcal{L}$ and subdividing it into s subboxes (Step 6). For each

Algorithm 1: GOP_ex

input : $\boldsymbol{x}_0$ – the search box
output: $\mathcal{R}$ – list of candidates for all global minimizers
output: $\boldsymbol{m}$ – enclosure of the global minimum value

1 $\mathcal{L} = \{\boldsymbol{x}_0\}$; `/* working list of boxes */`
2 $\tilde{f} = \infty$; `/* best-known upper bound for the glob. min. */`
3 **do**
4 $\quad \boldsymbol{y} = Head(\mathcal{L})$;
5 $\quad$ remove $\boldsymbol{y}$ from $\mathcal{L}$;
6 $\quad Split(\boldsymbol{y}, \boldsymbol{u}^1, \ldots, \boldsymbol{u}^s)$;
7 $\quad$ **for** $i = 1, \ldots, s$ **do**
8 $\quad\quad$ **if** $ProcessBox(\boldsymbol{u}^i, \mathcal{L}, \tilde{f})$ **then** next i;
9 $\quad\quad$ **if** $\boldsymbol{u}^i \subseteq \mathrm{int}(\boldsymbol{x}_0)$ **then**
10 $\quad\quad\quad fghEval(\boldsymbol{u}^i, \boldsymbol{f}(\boldsymbol{u}), \nabla \boldsymbol{f}(\boldsymbol{u}), \nabla^2 \boldsymbol{f}(\boldsymbol{u}))$;
11 $\quad\quad\quad c = \mathrm{mid}(\boldsymbol{u}^i)$;
12 $\quad\quad\quad fgEval(c, \boldsymbol{f}(c), \nabla \boldsymbol{f}(c))$;
13 $\quad\quad\quad$ **if** $NewtonTest(\boldsymbol{u}^i, \nabla^2 \boldsymbol{f}(\boldsymbol{u})), \nabla \boldsymbol{f}(c), \boldsymbol{v}^1, \ldots, \boldsymbol{v}^l)$ **then** next i;
14 $\quad\quad\quad$ **for** $j = 1, \ldots, l$ **do**
15 $\quad\quad\quad\quad$ **if** $ProcessBox(\boldsymbol{v}^j, \mathcal{L}, \tilde{f})$ **then** next j;
16 $\quad\quad\quad\quad$ **if** $Stopping\_crit(\boldsymbol{v}^j)$ **then** add $\boldsymbol{v}^j$ in $\mathcal{R}$; **else** add $\boldsymbol{v}^j$ in $\mathcal{L}$;
17 $\quad\quad\quad$ **end**
18 $\quad\quad$ **else**
19 $\quad\quad\quad$ **if** $Stopping\_crit(\boldsymbol{u}^i)$ **then** add $\boldsymbol{u}^i$ in $\mathcal{R}$; **else** add $\boldsymbol{u}^i$ in $\mathcal{L}$;
20 $\quad\quad$ **end**
21 $\quad$ **end**
22 **while** $\mathcal{L} \neq \emptyset$;
23 $\boldsymbol{m} = [\min_{y \in \mathcal{R}} \inf(\boldsymbol{f}(\boldsymbol{y})), \tilde{f}]$;

subbox $\boldsymbol{u}^i$ the `ProcessBox()` procedure is executed first. The primary goal of `ProcessBox()` is to erase those parts of $\boldsymbol{u}^i$ that are guaranteed to be suboptimal, using function and derivative information *up to first order*. In addition, by the update of $\tilde{f}$ suboptimal boxes from $\mathcal{L}$ are also removed. The pseudo-code of `ProcessBox()` is given in Algorithm 2; a detailed description of it will be given later in this section.

If `ProcessBox()` in Step 8 returns false (meaning that $\boldsymbol{u}^i$ was not fully erased), we turn to using second order function information. In GOP_ex the only accelerating test that uses second order derivatives is the interval Newton test [19]. In its present implementation (see Sec. 2.5) the Newton test applies only for boxes that are in the interior of the search space, so this condition is tested first in Step 9. Then the interval Hessian $\nabla^2 \boldsymbol{f}(\boldsymbol{u})$ and the midpoint gradient $\nabla \boldsymbol{f}(c)$ are calculated (Steps 10–12), and one step of the interval Newton method is executed (Sec. 2.5). The Newton step can either return the answer that $\boldsymbol{u}$ can be fully erased, or it returns $l \leq n+1$ subboxes $\boldsymbol{v}^j$ after dividing and pruning (see [8] for the implementation details). In Step 15, `ProcessBox()` is executed for all $\boldsymbol{v}^j$. In Step 16, the stopping

Algorithm 2: ProcessBox

input/output: $\boldsymbol{u}$ – the current box
input/output: $\mathcal{L}$ – working list of boxes
input/output: $\tilde{f}$ – current best-known upper bound for the global minimum
return value : true iff the whole box is sure to be eliminated

```
1  fgEval(u, f(u), ∇f(u));
2  if f̃ < inf(f(u)) then return true;
3  c = mid(u);
4  fEval(c, f(c));
5  if sup(f(c)) < f̃ then
6      f̃ = sup(f(c));
7      CutoffTest(L, f̃);
8  end
9  f(u) := f(u) ∩ (f(c) + ∇f(u)(x − c)) ; /* 1st ord. mean value form */
10 if f̃ < inf(f(u)) then return true;
11 if MonotonicityTest(u, ∇f(u)) then return true;
12 if u is pruned by the MonotonicityTest then
13     fEval(u, f(u));
14     if f̃ < inf(f(u)) then return true;
15     c = mid(u);
16     fEval(c, f(c));
17     if sup(f(c)) < f̃ then
18         f̃ = sup(f(c));
19         CutoffTest(L, f̃);
20     end
21 end
22 return false;
```

criterion is checked for the remaining part of $\boldsymbol{v}^j$ (Sec. 2.6): if $\boldsymbol{v}^j$ is proven to be 'small enough', it is placed on the list of result boxes $\mathcal{R}$, otherwise it is inserted into $\mathcal{L}$. If the condition of Step 9 does not hold for $\boldsymbol{u}^i$, we do not proceed with second order procedures, just check the stopping criterion for $\boldsymbol{u}^i$ and store it as detailed above (Step 19).

The main loop is executed until $\mathcal{L}$ becomes empty, that is, the whole search space is processed (Step 22). At the end of the procedure, the enclosure of the global minimum is evaluated (Step 23), and $\mathcal{R}$ contains the interval enclosures of all global minimizers.

The `ProcessBox()` function (Algorithm 2) starts by computing the interval function and gradient values on the current box $\boldsymbol{u}$. In Step 2, a bound test is performed to test the suboptimality of $\boldsymbol{u}$. Then an attempt is made to improve $\tilde{f}$ by midpoint information. If $\tilde{f}$ is updated, a cut-off test is run on $\mathcal{L}$: this removes all boxes $\boldsymbol{x} \in \mathcal{L}$ for which $\tilde{f} < \inf(\boldsymbol{f}(\boldsymbol{x}))$ holds, that is, those that are suboptimal due to their bounds (Alg. 2, Steps 3–8).

In Steps 9–10, an update is tried on $\boldsymbol{f}(\boldsymbol{u})$, using a first order mean value form (see Sec. 2.2), along with a repeated bound test. In Step 11, the monotonicity test is run on $\boldsymbol{u}$ (Sec. 2.5). If $\boldsymbol{u}$ is shrinked by the latter test, in Steps 13–21 we re-evaluate its function enclosure, perform a bound test, then re-evaluate the new center and do a cut-off test with the new values, as above.

2.2 Function and derivative evaluations.

Interval evaluations on f are done by naive interval arithmetic: substituting real-type arithmetic operators and elementary functions with their interval versions and computing the function enclosure with IA. Interval derivatives and Hessians are evaluated with *automatic differentiation* (AD) [7] in forward mode. A derivative evaluation in forward AD is done the following way: during the computation of f over $x \in \mathbb{R}^n$, for each subexpression q of f the whole gradient vector $\nabla q(x)$ is also computed. For interval arguments the evaluation is done the same way with IA. The evaluation of real and interval Hessians are done similarly, computing the whole Hessian for each occurring subexpression.

The advantage of the forward AD is that it does not require any special representation of f, the derivatives are computed in parallel to the function evaluation (up to the required order). The disadvantage of forward IA is its cost: the time cost of computing $\nabla f(x)$ or $\nabla \boldsymbol{f}(\boldsymbol{x})$ is expected to be equal to n times the cost of computing $f(x)$ or $\boldsymbol{f}(\boldsymbol{x})$, resp. For Hessians the cost is n^2 times the cost of the respective real or interval function evaluation.

In GOP_ex, if $\boldsymbol{f}(\boldsymbol{x})$ and $\nabla \boldsymbol{f}(\boldsymbol{x})$ are known for a box $\boldsymbol{x}$, then the interval enclosure over $\boldsymbol{x}$ can be tightened by a *first order mean value form* update: for $c \in \boldsymbol{x}$, $\operatorname{rg}(f, \boldsymbol{x}) \subseteq \boldsymbol{f}_m(\boldsymbol{x}, c) := \boldsymbol{f}(c) + \nabla \boldsymbol{f}(\boldsymbol{x}) \cdot (\boldsymbol{x} - c)$, thus, one can take $\boldsymbol{f}(\boldsymbol{x}) := \boldsymbol{f}(\boldsymbol{x}) \cap \boldsymbol{f}_m(\boldsymbol{x}, c)$. In the present algorithm the *center* c is chosen as $\operatorname{mid}(\boldsymbol{x})$. Note that in Alg. 2 $\boldsymbol{f}(c)$ will be in hand for the mean value update, since this is the value we use to improve $\tilde{f}$.

In Algorithms 1 and 2, `fEval()`, `fgEval()`, and `fghEval()` denote the interval function evaluation, interval function and gradient evaluation, and interval function, gradient, and Hessian evaluation routines, resp.

2.3 Operations on the B&B 'tree'.

In GOP_ex, the working list $\mathcal{L}$ is implemented as an ordered linked list. There are several list operations to be specified for the algorithm:

- Subbox selection. We select that subbox $\boldsymbol{y}$ from $\mathcal{L}$ for which $\inf(\boldsymbol{f}(\boldsymbol{y}))$ is minimal. In case of a tie, we choose the oldest box among the tied ones. This is known as the Moore-Skelboe type box selection rule. (In GOP_ex also the Hansen type is encoded, which simply chooses the oldest box in $\mathcal{L}$. However, throughout the years we found out that the

Moore-Skelboe rule is usually superior to the latter one, e.g., because the cut-off test below is faster.) The ordering of $\mathcal{L}$ is done according to $\inf(\boldsymbol{f}(\boldsymbol{y}))$ in increasing order, so that the selection operation is done in constant time. That is, we always pick the first box from $\mathcal{L}$, hence the name `Head()` in Step 4 of Alg. 1.
- Storing the remaining part of a subbox. This operation (occurring in Steps 16 and 19 of Alg. 1) is done according to the criterion discussed above, so that the ordering of the list is respected.
- Cut-off test. Whenever $\tilde{f}$ is updated, all boxes $\boldsymbol{u} \in \mathcal{L}$ with $\inf(\boldsymbol{f}(\boldsymbol{u})) > \tilde{f}$ can be erased by suboptimality. For the Moore-Skelboe type working list this means that elements at the end of $\mathcal{L}$ can be detached and deleted (hence the name cut-off).

2.4 Box subdivision.

The current box is subdivided in the procedure `Split()`, which is specified by the following two criteria:

- number of subboxes: GOP_ex offers bisection and multisection (in one or two directions) [14], the default is bisection;
- subdivision direction selection: GOP_ex contains the widest component rule and the first order merit function rules by Csendes and Ratz [5], the default setting is the first order rule named 'Rule C' in [5].

2.5 Accelerating tests.

The accelerating tests are used to eliminate parts of the current box $\boldsymbol{x}$ that are proved to contain only suboptimal points. GOP_ex has the following three accelerating tests (in increasing order of the derivative information required):

- Bound test: if $\inf(\boldsymbol{f}(\boldsymbol{x})) > \tilde{f}$, then the whole box $\boldsymbol{x}$ can be erased. This is essentially the same as the cut-off test above, applied to the current box.
- Monotonicity test: if $0 \notin \nabla_i \boldsymbol{f}(\boldsymbol{x})$ for some i, then $\boldsymbol{x}$ cannot contain a global minimizer *in its interior*. Thus $\boldsymbol{x}$ can be fully erased, or its ith component can be reduced to the respective lower or upper bound. In the pseudo codes, the first parameter of `MonotonicityTest(`$\boldsymbol{u}, \nabla \boldsymbol{f}(\boldsymbol{u})$`)` thus serves as an input-output argument. `MonotonicityTest()` returns true if and only if the whole $\boldsymbol{x}$ was erased.
- Newton test: in GOP_ex, this test is used to search for all stationary points of f in $\boldsymbol{x}$, and is currently restricted to boxes that are in the interior of $\boldsymbol{x}_0$. The method included in GOP_ex is from the family of the so-called *interval Newton-methods* and is referred to as the interval Newton-Gauss-Seidel method in [8] and as the Hansen-Sengupta operator in a general theoretical treatment in [19].

If $x \in \boldsymbol{x}$ is a solution of $\nabla f(x) = 0$ and $c \in \boldsymbol{x}$, then it is easy to see that $x - c$ will be in the set S given by

$$S = \{y \in \boldsymbol{x} - c : Ay = b \text{ for some } A \in C\nabla^2 \boldsymbol{f}(\boldsymbol{x}), b \in -C\nabla \boldsymbol{f}(c)\}.$$

Informally, S is the solution set of the (preconditioned) interval linear system of equations with coefficient matrix $C\nabla^2 \boldsymbol{f}(\boldsymbol{x})$ and right-hand side $-C\nabla \boldsymbol{f}(c)$, truncated to the box $\boldsymbol{x} - c$. GOP_ex uses a midpoint inverse preconditioner, that is, $C = (\text{mid}(\nabla^2 \boldsymbol{f}(\boldsymbol{x})))^{-1}$ in the formulas above. Starting from $\boldsymbol{x}$ and $\boldsymbol{y} := \boldsymbol{x} - c$, one then applies an *interval Gauss-Seidel iteration* to the above system to get an updated interval enclosure of the truncated solution set. Denoting the output of the iteration step by $\boldsymbol{y}'$ and $\boldsymbol{x}' = \boldsymbol{y}' + c$, the following statements holds:

- every stationary point of f in $\boldsymbol{x}$ is also located in $\boldsymbol{x}'$;
- if $\boldsymbol{x} \cap \boldsymbol{x}' = \emptyset$, then f has no stationary point in $\boldsymbol{x}$;
- if $\boldsymbol{x}' \subseteq \text{int}(\boldsymbol{x})$, then f has a unique stationary point in $\boldsymbol{x}'$.

In Algorithm 1, `NewtonTest()` returns true if and only if the box contains no stationary point and thus the whole box can be erased.

2.6 Stopping criterion.

A box $\boldsymbol{x}$ is placed to the list of result boxes if $\text{wid}(\boldsymbol{f}(\boldsymbol{x}))$ is smaller than a pre-given tolerance value.

3 The COCONUT Environment

The COCONUT Environment [4] (shortly called COCONUT), developed at the University of Vienna under the leadership of Hermann Schichl, is a modular open-source environment for global optimization and constraint satisfaction problems. The original COCONUT project, started in 2000, was funded by an IST programme of the European Community (IST-200-26063, 2000–2004). Since then, COCONUT has been developed in smaller scale projects of the Austrian Science Fund.

The modularity of COCONUT means that it can be expanded by commercial and open-source solver components (inference engines). Thus, it integrates existing tools and methods, and promotes the development of new, state-of-the-art solvers. Since the GOP_ex code has the main disadvantage that it is hard to interface and improve with other interval related methods for global optimization, it was a straightforward idea to implement the GOP_ex algorithm as a new solver in the COCONUT framework.

Below we give a very brief introduction to the basic concepts of COCONUT. For more details on its structure and its components not used in the present bound constrained settings, see [4,16,23].

COCONUT represents the optimization problems in the form of *directed acyclic graphs* (DAGs). The leaves of the graphs are the variables,

with initial interval bounds representing the bound constraints. The order of evaluations is represented by directed edges in a natural way. The *model functions* (objective function(s), constraints, auto-generated optimality conditions) are located on the top level of the DAG. To reduce the size of the DAG, multiplicative and additive constants of expressions are stored at the respective nodes, and for many arithmetic operators and elementary functions more flexible expression types are used (such as a 'sum' node is used for addition with an arbitrary number of terms).

Furthermore, the B&B procedure is also represented in COCONUT as a DAG, in the so-called *search graph*. Here each node of the search graph actually corresponds to a node of the B&B search tree. The advantage of this approach is that in each node it is enough to store the difference of the actual node from its parent(s), in the form of data structures called *deltas*. For bound constrained problems, most deltas are bound deltas (representing the pruning of the search region) or split deltas (representing branching).

The B&B elements of Sec. 2 are all implemented in a straightforward way in the COCONUT Environment. The only significant difference from the original implementation of GOP_ex is – as mentioned above – the data structure representing the search procedure, i.e., the access to the open leaves. Since with the search graph representation one loses the advantage of keeping the open leaves in an ordered form, we additionally developed a so-called *leaves cache*, that maintains a representation of the leaves in parallel to their search graph representation. Furthermore, in contrast to the original GOP_ex, the leaves are stored in a binary search tree (using the *set* container of Standard Template Library), thus, allowing optimized performance for the insertion and deletion operations.

4 Improving the GOP_ex algorithm in COCONUT I. – new evaluation methods

4.1 Gradient evaluations with backward AD

In COCONUT, the DAG representation of the model functions makes it possible to implement AD methods in the more efficient backward mode [22]. In addition, for comparison reasons we have implemented gradient evaluation with forward AD as well, as described in Sec. 2.2.

The cost of one first order derivative calculation with backward AD is in general only a small constant multiple of one function evaluation (while with forward AD it is around n-times the cost of a function evaluation), so it is much more economic in both time and storage than the forward mode. Therefore, the backward mode is always considered as the preferred method of calculation, especially for interval arguments. In Sec. 5.2.1 we will show that this superiority is not universally true in practice.

4.2 Hessian evaluations with backward AD

In COCONUT, the basic way of evaluating Hessians is the computation of Hessian-vector products with backward AD [25]. The whole Hessian is evaluated by using the n unit vectors. Nevertheless, when the Hessian has a sparser structure, the number of necessary Hessian-vector product evaluations can usually be reduced. This reduction is done by a so-called Hessian coloring algorithm, (Algorithm 4.1 in [6]), that is a heuristic that constructs $0-1$ vectors so that all nonzero Hessian entries are computed explicitly. By default, Hessian coloring is attempted when the sparsity ratio of the Hessian is below 0.9.

According to [25], as a by-product of the backward Hessian evaluation one also gets an enclosure of the gradient, which can be used to update the current $\nabla \boldsymbol{f}(\boldsymbol{x})$ value (or newly set it if it was not yet evaluated).

4.3 Acquiring derivatives from the Karush–John conditions

The *KJgen* module of COCONUT computes the DAGs of the Karush-John first order necessary optimality conditions of any general constrained problem [23]. The generated conditions can be attached to the original DAG and can be evaluated like any other model functions (objectives or constraints). For bound constrained problems, a subset of these conditions is of the form

$$\kappa \cdot \nabla_j f(x) - \lambda_j = 0, \quad j = 1, \ldots, n\,,$$

where κ and λ_j, $j = 1, \ldots, n$ are the objective and constraint multipliers, respectively. One can immediately recognize that the partial derivative functions of the objective become available as subgraphs in these DAGs, that is, evaluating the function/derivative/Hessian of the left-hand side of the above equations at $(\boldsymbol{x}, \kappa, \lambda) = (\boldsymbol{x}, 1, 0)$ by the ordinary COCONUT evaluators, we gain access to $\nabla \boldsymbol{f}(\boldsymbol{x}), \nabla^2 \boldsymbol{f}(\boldsymbol{x})$, and $\nabla^3 \boldsymbol{f}(\boldsymbol{x})$, resp. For $\nabla \boldsymbol{f}(\boldsymbol{x})$, this method is usually an inferior alternative to backward AD, since it requires n function evaluations. However, for $\nabla^2 \boldsymbol{f}(\boldsymbol{x})$ it is a reasonable alternative of the Hessian-vector product routines (although Hessian-coloring cannot be used), and for $\nabla^3 \boldsymbol{f}(\boldsymbol{x})$ this was actually the only way to compute third order problem information before the recent implementation of explicit third order evaluation routines. (Studying the use of third order information is under development for COCONUT; methods utilizing it will be discussed in forthcoming papers, such as in [26].)

4.4 Bicentered forms

As it is known from theory (see [1] and Theorem 2.3.6 in [19]), the first order mean value form $\boldsymbol{f}_m(\boldsymbol{x}, c) := \boldsymbol{f}(c) + \nabla \boldsymbol{f}(\boldsymbol{x}) \cdot (\boldsymbol{x} - c)$ gives an interval enclosure of minimal radius when the center c is chosen as the midpoint of

$\boldsymbol{x}$. Furthermore, $\inf(\boldsymbol{f}_m(\boldsymbol{x}, c))$ can be maximized and $\sup(\boldsymbol{f}_m(\boldsymbol{x}, c))$ can be minimized by choosing different centers. Namely, by defining

$$d_i = \operatorname{mid}(\nabla_i \boldsymbol{f}(\boldsymbol{x}))/\operatorname{rad}(\nabla_i \boldsymbol{f}(\boldsymbol{x})),$$

$$p_i = \begin{cases} 1 & \text{if } d_i > 1; \\ -1 & \text{if } d_i < -1; \\ d_i & \text{otherwise,} \end{cases}$$

$$z_{*,i} = \operatorname{mid}(\boldsymbol{x}_i) - p_i \operatorname{rad}(\boldsymbol{x}_i), \quad z_i^* = \operatorname{mid}(\boldsymbol{x}_i) + p_i \operatorname{rad}(\boldsymbol{x}_i),$$

for $i = 1, \ldots, n$, $\inf(\boldsymbol{f}_m(\boldsymbol{x}, c))$ attains its maximum at the center $c = z_*$, and $\sup(\boldsymbol{f}_m(\boldsymbol{x}, c))$ attains its minimum at the center $c = z^*$. That is, by taking these two centers and evaluating the intersection of the resulting two mean value enclosures, a better range estimate can be obtained as compared to the case of a single center. This is, of course, reached for the cost of an extra interval function evaluation.

4.5 Slopes

Given two points x and c in the domain of f, the (row) vector $f[x, c] \in \mathbb{R}^n$ is called a *slope* of f between x and c if $f(x) = f(c) + f[x, c](x - c)$ holds. Fixing c and bounding $f[x, c]$ over all $x \in \boldsymbol{x}$ by the *interval slope* $\boldsymbol{f}[\boldsymbol{x}, c] \in \mathbb{I}^n$, the formula

$$f(c) + \boldsymbol{f}[\boldsymbol{x}, c](\boldsymbol{x} - c) \tag{2}$$

is an enclosure of $\operatorname{rg}(f, \boldsymbol{x})$ [12]. Just like the mean value form, (2) is also a centered form, called *slope form*. Slopes can be calculated on DAGs in backward mode in a similar fashion as the interval derivatives [2, 22].

The advantage of slope forms over mean value forms is that they usually provide better range enclosures, in particular, if the center of the slope is carefully chosen. (But the theoretical convergence rate of the slope form to the exact range is quadratic, just like the mean value form [19].) In the COCONUT Environment, we determine the center by the following heuristic method. We use interval derivative information to select a point of $\boldsymbol{x}$ from which the slope is expected to have the smallest width. For instance, if f is proven to be monotone in its ith component, a reasonable choice, for each $i = 1, \ldots, n$, is to set

$$c_i = \begin{cases} \sup(\boldsymbol{x}_i) & \text{if } \sup(\nabla_i \boldsymbol{f}(\boldsymbol{x})) < 0; \\ \inf(\boldsymbol{x}_i) & \text{if } \inf(\nabla_i \boldsymbol{f}(\boldsymbol{x})) > 0, \end{cases}$$

i.e., to set c_i to the respective lower or upper bound. Otherwise, if $0 \in \nabla_i \boldsymbol{f}(\boldsymbol{x})$, for each $i = 1, \ldots, n$ we can apply the heuristic

$$c_i = \begin{cases} \sup(\boldsymbol{x}_i) & \text{if } \sup(\nabla_i \boldsymbol{f}(\boldsymbol{x})) \leq -\inf(\nabla_i \boldsymbol{f}(\boldsymbol{x})); \\ \inf(\boldsymbol{x}_i) & \text{if } \sup(\nabla_i \boldsymbol{f}(\boldsymbol{x})) > -\inf(\nabla_i \boldsymbol{f}(\boldsymbol{x})), \end{cases}$$

so that c is set to a corner of $\boldsymbol{x}$.

A problem arising here is that in our algorithms $\nabla \boldsymbol{f}(\boldsymbol{x})$ is often not available at the time of computing the slope (because the slope itself is constructed in place of it). But in many cases the interval gradient is obtained on the *parent* of $\boldsymbol{x}$ as a by-product of second order evaluation methods (see Sec. 4.8), which can be used in place of $\nabla \boldsymbol{f}(\boldsymbol{x})$. When the interval gradient on the parent is also not available, we choose the midpoint as the center of the slope. (An alternative would be to use $\nabla \boldsymbol{f}(c)$) instead of $\nabla \boldsymbol{f}(\boldsymbol{x})$) for the heuristic; however, this idea also suffers from the possible lack of availability of $\nabla \boldsymbol{f}(c)$).)

There is, however, a big disadvantage of interval slopes as compared to interval derivatives, when applied in interval B&B algorithms: interval slopes do not bound the gradient, therefore they cannot directly provide monotonicity information. Thus, the monotonicity test must be ignored in the first order evaluation phase when interval derivatives are replaced with slopes.

Algorithmically, the slope evaluation is done in place of the `fgEval()` function of `ProcessBox()` in Alg. 2, Step 1. Note that since the slope requires a center, it must be computed before `fgEval()`, which means that Steps 1–2 and Steps 3–8 of Alg. 2 have to be swapped in the slope-based algorithm variants.

In the next two subsections (Sec. 4.6 and 4.7) we introduce a new acceleration test and a new evaluation method combining the available interval derivative and slope data. These methods are developed and implemented in the COCONUT Environment for the particular purpose of enhancing the GOP_ex algorithm.

4.6 Centered form pruning test

According to the first order mean value form, for a box $\boldsymbol{x}$, for all $x, c \in \boldsymbol{x}$ and for any $i = 1, \ldots, n$ it is true that

$$f(x) \in \boldsymbol{f}(c) + \sum_{\substack{j=1 \\ j \neq i}}^{n} \nabla_j \boldsymbol{f}(\boldsymbol{x})(\boldsymbol{x}_j - c_j) + \nabla_i \boldsymbol{f}(\boldsymbol{x})(\boldsymbol{x}_i - c_i) =$$
$$= \boldsymbol{u}_i + \nabla_i \boldsymbol{f}(\boldsymbol{x})(\boldsymbol{x}_i - c_i). \tag{3}$$

Whenever $f(x) > \tilde{f}$, then x is surely suboptimal. That is, we can restrict our attention to those $x_i \in \boldsymbol{x}_i$ points that are solutions of the interval inequality

$$\boldsymbol{u}_i + \nabla_i \boldsymbol{f}(\boldsymbol{x})(\boldsymbol{x}_i - c_i) \leq \tilde{f},$$

i.e., for which there exist real numbers $u \in \boldsymbol{u}_i$ and $d \in \nabla_i \boldsymbol{f}(\boldsymbol{x})$ such that $u + d(x_i - c_i) \leq \tilde{f}$. The set of solutions can be found easily after rearrangements, in a fashion similar to the so-called constraint propagation (Section 6.3). Depending on the signs of $\nabla_i \boldsymbol{f}(\boldsymbol{x})$ and $\tilde{f} - \boldsymbol{u}_i$, the result is either an empty

set, or the entire range $\mathbb{R}$, or an interval with an infinite bound; and in the case when $\nabla_i \boldsymbol{f}(\boldsymbol{x})$ contains 0 in its interior, it is $[-\infty, a] \cup [b, \infty]$ for some $a, b \in \mathbb{R}$. This solution set can then be intersected with $\boldsymbol{x}_i$, leading to a reduction in that component.

The test is easily seen to be superseded by the monotonicity test when $0 \notin \nabla_i \boldsymbol{f}(\boldsymbol{x})$, but in other cases it often proves to be useful. Furthermore, the test is equally valid for slope forms as well, using $\boldsymbol{f}_i[\boldsymbol{x}, c]$ in place of $\nabla_i \boldsymbol{f}(\boldsymbol{x})$. An advantage of the test for slopes is that it provides a good alternative to the (otherwise inapplicable) monotonicity test (see Sec. 4.5).

The above procedure of reducing $\boldsymbol{x}_i$ can be applied for all components, and since the interval gradients and slopes are inclusion isotone, the already reduced components can be used to construct $\boldsymbol{u}_i$ in (3).

4.7 Slopes of the gradient

When the partial derivative functions of the objective are available in forms of DAGs according to Sec. 4.3, it is possible to evaluate the first order slopes $\nabla_i \boldsymbol{f}[\boldsymbol{x}, c]$ of these functions. Then for a box $\boldsymbol{x}$ and for all c and $x \in \boldsymbol{x}$,

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

i.e., the right-hand side is an enclosure of $\mathrm{rg}(\nabla_i f, \boldsymbol{x})$. In vector form, we have

$$\nabla f(x) \in \nabla \boldsymbol{f}(c) + S_f[\boldsymbol{x}, c](\boldsymbol{x} - c), \tag{4}$$

where $S_f[\boldsymbol{x}, c] \in \mathbb{I}^{n \times n}$ is the matrix with the slopes $\nabla_i \boldsymbol{f}[\boldsymbol{x}, c]$ in the ith row, called the *slope matrix of the derivative of* f.

An important observation here is that with this formula the solution set of $\nabla f(x) = 0$ can be bounded, just as with the ordinary Newton method, but using the slope matrix $S_f[\boldsymbol{x}, c]$ instead of the interval Hessian $\nabla^2 \boldsymbol{f}(\boldsymbol{x})$. Since the 'first order part' of obtaining $S_f[\boldsymbol{x}, c]$ is symbolic and first order slopes provide better enclosures than interval derivatives, we expect that the width of $S_f[\boldsymbol{x}, c]$ will be significantly smaller than that of $\nabla^2 \boldsymbol{f}(\boldsymbol{x})$, leading to a more efficient Newton method.

4.8 Second order centered forms

The *second order mean value form* (see, e.g., [20]) is constructed in a similar way as its first order version, but from the Taylor expansion with second order remainder term: for a box $\boldsymbol{x}$ and for $c \in \boldsymbol{x}$

$$f(x) \in \boldsymbol{f}(c) + \nabla \boldsymbol{f}(c)(\boldsymbol{x} - c) + \frac{1}{2}(\boldsymbol{x} - c)^T \nabla^2 \boldsymbol{f}(\boldsymbol{x})(\boldsymbol{x} - c) := \boldsymbol{f}_{m,2}(\boldsymbol{x}, c).$$

In practice, $\boldsymbol{f}_{m,2}(\boldsymbol{x}, c)$ is evaluated as

$$\boldsymbol{f}(c) + \left(\nabla \boldsymbol{f}(c) + \frac{1}{2} \nabla^2 \boldsymbol{f}(\boldsymbol{x})(\boldsymbol{x} - c) \right) (\boldsymbol{x} - c),$$

in order to reduce the overestimation due to the so-called *subdistributivity* [17,19]. Furthermore, during evaluation it is possible to compute an enclosure of $\mathrm{rg}(\nabla f, \boldsymbol{x})$ by

$$\nabla \boldsymbol{f}(c) + \nabla^2 \boldsymbol{f}(\boldsymbol{x})(\boldsymbol{x} - c), \tag{5}$$

and update the current interval gradient enclosure $\nabla \boldsymbol{f}(\boldsymbol{x})$ by taking the intersection with (5). If there is no interval gradient available at the time of evaluation (e.g., because slopes were computed in the first order evaluation phase), we set $\nabla \boldsymbol{f}(\boldsymbol{x})$ to (5). Then, if the interval gradient is improved (or becomes newly available) by the above process, a further enclosure of the objective function is evaluated by the first order mean value form.

In view of Section 4.7, it turns out that there is a simple way of constructing a second order enclosure using the slope matrix of the derivative. For a box $\boldsymbol{x}$ and for all $x, c \in \boldsymbol{x}$ a proper first order enclosure is given by the range of the gradients:

$$f(x) \in \boldsymbol{f}(c) + \mathrm{rg}(\nabla f, \boldsymbol{x})(\boldsymbol{x} - c),$$

and since $\mathrm{rg}(\nabla f, \boldsymbol{x}) \subseteq \nabla \boldsymbol{f}(c) + S_f[\boldsymbol{x}, c](\boldsymbol{x} - c)$ from (4), we obtain

$$f(x) \in \boldsymbol{f}(c) + \left(\nabla \boldsymbol{f}(c) + S_f[\boldsymbol{x}, c](\boldsymbol{x} - c)\right)(\boldsymbol{x} - c) := \boldsymbol{f}_{ms,2}(\boldsymbol{x}, c),$$

called the *second order mixed mean value-slope form*. Clearly, the interval gradient update/construction method discussed above is applicable for this form as well.

Algorithmically, the second order centered form evaluations are best to invoke right before calling the interval Newton method. At this point all required data, including $\nabla \boldsymbol{f}(c)$, is available, thus, the centered form calculation requires no extra evaluation. Of course, after updating $\boldsymbol{f}(\boldsymbol{x})$ with the second order centered form, a bound test on the current box is performed.

5 Numerical comparisons

This section is devoted to present numerous comparative results of the possible algorithm variants. The sequence of the comparisons shows the decision path we actually went through to obtain a suggested set of parameters, that we use as defaults in the competitive versions of the software (and also, in later studies). Although it is not possible to compare and present *all* combinations of options, however, *we designed the sequence of comparisons in such a way, that with some of the tests we give empirical answers to long-lasting questions about alternative methods of evaluations as parts of an interval B&B algorithm (such as interval gradients versus slopes), that have never been studied numerically in such detail before.*

5.1 The methodology of comparing algorithm variants

Throughout the paper we apply a uniform way of comparing different algorithm variants. The test set consists of 164 bound constrained/unconstrained problems from two different sources. First, we took all bound constrained and unconstrained problems up to dimension 50 from the COCONUT test set (that is itself part of the COCONUT Environment) that are the twice continuously differentiable over the whole search domain (or, for unconstrained problems, over the artificial search box defined below), and extended this set by some standard interval GO test problems used in [5]. The problems are classified by their dimension as 'small' ($1 \leq n \leq 5$, 110 problems), 'medium' ($6 \leq n \leq 20$, 44 problems), and 'large' ($21 \leq n \leq 50$, 10 problems). Unconstrained variables were bounded to $[-1000, 1000]$. The running time limit was set for 60, 300, and 600 seconds, respectively, for the three problem groups. The tolerance for the stopping criterion of Sec. 2.6 was set to 10^{-3} for all problems.

5.2 Finding the best interval gradient based variant

In order to make the sequence of comparisons clear, we name the main algorithm variants that are compared to each other; in each comparison we compare two variants that usually differ by enabling/disabling one of the algorithmic elements, or using a different method for evaluating the same kind of data.

5.2.1 Forward vs. backward first order AD. The first and most obvious option we need to investigate is the advantage of backward interval gradient evaluations over forward evaluations. Although theory predicts an all-round advantage of the former methods, it turns out that in practice this is not always the case: as we will see below, during the tests we found that in some cases better performance can be reached with forward evaluation modes. A detailed investigation of the phenomenon gave us also a theoretical explanation that can be considered in later studies. This is a typical example that shows the *importance and the main strength of the COCONUT Environment: the implementation of a wide range of methods in a uniform framework is the only way to recognize such phenomena, trade-offs, etc., between different ideas and algorithm variants.*

Thus, in **Test 1** we compare algorithm version **V0**, denoting the original GOP_ex algorithm (with Hessian-vector products instead of forward mode Hessians), and version **V1**, that differs from V0 by switching from forward to backward mode in interval gradient evaluations. The results are presented in Table 1.

The table (and also the subsequent ones in this section) is built as follows: it has four rows, showing data for the problem instances solved by both, either, or none of the variants. The respective set mark is given in the

first column of the table. The first group of data columns (columns 2–4) shows the number of solved problem instances, grouped by problem size. Column 5 shows the total of the previous 3 columns. Whenever a problem is solved by both solvers, or neither of them, we have the chance to do a more detailed analysis on the performance difference.

If a problem is solved by both variants, we take the execution time as the performance indicator. Let us name the two algorithm variants 'A' and 'B' and denote t_A and t_B the execution times on such a problem instance. Obviously, due to slight variances of the execution times on a computer (that are magnified for problems that are solved very fast) we need to add tolerances to these values: if $|t_A - t_B| < 0.02$ seconds, then the two performances are considered equal. Otherwise, we consider the time ratio $r = \max(t_B, 0.01)/\max(t_A, 0.01)$. If $r < 0.98$ (i.e., variant B improves variant A by at least 2%), then we claim B performs better than A. If $1/r < 0.98$ (i.e., variant A improves variant B by at least 2%), then we claim A performs better than B. In all other cases we treat the performance of A and B equal. The number of problems for which the two variants have equal performance are given in column 'eq'. The number of problems in which A performs better than B and the average of the $1/r$ values *over this particular problem set* is displayed in the columns marked with 'A (av.)'. The columns 'B (av.)' are interpreted in an analogous way, using the average r values over the respective problem set.

If a problem is solved by neither variant (i.e., a timeout occurred), we can still compare the two algorithms by an indicator that shows the overall progress within the common time budget. To get this, one can first compute the total volume V_R of the boxes remaining in the open leaves of the search tree, and divide it by the volume V of the search box one, obtaining the relative unprocessed volume of the search space. The disadvantage of V_R/V is its sensitivity to the dimension, so problems of different sizes are hard to compare. To resolve this issue, for an n-dimensional problem one can take n-th root and obtain $v = (V_R/V)^{1/n}$. Note that for an unsolved problem $v \in (0, 1]$, with $v = 1$ when no progress at all is reached by the algorithm. Intuitively, v is the average decrease in one coordinate direction made by the algorithm when locating the global minimizer(s). We will call this value the *overall progress* measurement. Denoting the respective values by v_A and v_B for the two algorithm variants, the ratio $r = v_B/v_A$ is computed, and the performance improvements are calculated the same way as for the time ratios above. For the overall progress comparisons we also used a tolerance of 2% to mark the range of equality.

If a problem is solved by only one of the variants, then we make no further comparisons, since in such a case the performance difference is qualitative.

From Table 1 we conclude that 79% of the small problems were solved by both algorithms (87 out of 110), but this ratio drops to 27% and 10% for the medium and large problems, resp. Switching from forward to backward mode did not yield any significant qualitative improvement in the sense that only one (small) problem became solvable by the latter method.

Table 1 Test 1: comparing algorithm variants V0 (forward first order AD) and V1 (backward first order AD)

solved by	sm.	med.	lg.	tot.	V0 (av.)	eq.	V1 (av.)
both	87	12	1	100	1 (0.58)	76	23 (0.88)
only V0	0	0	0	0	–	–	–
only V1	1	0	0	1	–	–	–
neither	22	32	9	63	1 (0.88)	59	3 (0.93)

The performance comparison columns show the expected overall advantage of backward AD, with average gains of 12% and 7% for the two types of performance comparisons. (Although not detailed here, we also found the empirical validation of the fact that backward AD becomes more advantageous as the problem dimension grows: we observed a positive correlation of medium strength between the dimension and the performance improvements.)

One can observe, however, that there are rare cases when forward AD results in a much more advantageous method, as shown by the 'V0 (av.)' column. When looking for the reason of this phenomenon, we figured out that for some problems the interval forward AD actually gives much tighter enclosures than the backward one! The reason of this is that certain subexpressions of the form $\boldsymbol{x}(\boldsymbol{y}+\boldsymbol{z})$ are sometimes evaluated as $\boldsymbol{xy}+\boldsymbol{xz}$ in backward mode, causing excess overestimation by subdistributivity, while the forward one naturally avoids this pitfall; for a detailed theoretical explanation, see [25]. (Of course, the phenomenon is not present for real-type AD methods.) We found that this happened in around 5% of the present test problems, but caused a significant performance difference in only the two cases displayed in the table. Unfortunately, the expressions in which this situation may occur are rather hard to detect in advance, but developers of interval algorithms must be aware of it. All in all, in the subsequent algorithm variants of the present study we will use the generally far more efficient backward mode interval gradient evaluations.

5.2.2 Switching on small tests with no extra evaluation needs. In the next test we investigate the effect of enabling some minor tests, namely, second order centered forms (Sec. 4.8), the gradient enclosure updates from second order data (Sec. 4.2 and 4.8), and the centered form pruning test (Sec. 4.6). The common property of these tests is that they need no extra function evaluation effort, thus, they are comparatively cheap. In **Test 2**, algorithm **V1** is the same as in the previous section, while version **V2** is defined as V1 with the above three tests added. The comparison data is presented in Table 2.

The results show that the three minor tests did not yield a big improvement to the algorithm. (Actually, one reason of presenting them together

Table 2 Test 2: comparing algorithm variants V1 (backward first order AD) and V2 (backward first order AD, second order centered forms, gradient updates from second order data, centered form pruning)

solved by	sm.	med.	lg.	tot.	V1 (av.)	eq.	V2 (av.)
both	88	12	1	101	13 (0.92)	83	5 (0.75)
only V1	0	0	0	0	–	–	–
only V2	0	0	0	0	–	–	–
neither	22	32	9	63	1 (0.86)	58	4 (0.67)

Table 3 Test 3: comparing algorithm variants V2 (backward first order AD, second order centered forms, gradient updates from second order data, centered form pruning) and V3 (V2 with bicentered forms)

solved by	sm.	med.	lg.	tot.	V2 (av.)	eq.	V3 (av.)
both	87	12	1	100	17 (0.85)	78	5 (0.40)
only V2	1	0	0	1	–	–	–
only V3	0	0	0	0	–	–	–
neither	22	32	9	63	2 (0.58)	53	8 (0.80)

is that individually none of them changed the behaviour of the algorithm significantly.) Although V1 is slightly better on the solved problems, but the gain (or in this case, the loss due to the newly added tests) is only 8% on average. V2 may cause bigger improvements (but more rarely) on solved problem instances, and it is also proved to be slightly better in the performance of the unsolved problems. Another reason of preferring V2 is that for real-life problems, when the evaluation cost is often big, the relative effort of invoking these tests further decreases, so there is just no particular reason to skip them. Because of these facts we decided to opt for V2 and continue with this variant in the subsequent testing.

5.2.3 Midpoints vs. bicentered forms. Next we consider the question of how to evaluate the centers for the various tests. In **Test 3**, we consider version **V2** of the previous section, which contains simple midpoint evaluations. In version **V3** we apply all tools of V2 but exchange the midpoint evaluations to the bicenter method of Sec. 4.4. The results are shown in Table 3. The target of this test is to investigate an obvious algorithmic trade-off, namely, whether the better function updates of the bicentered forms can compensate the cost of the extra center evaluation.

The results display a pretty mixed picture: on problems solved by both algorithms, V2 performs better on many more instances, but with less gains on average, while for the unsolved problems just the opposite holds. In addition, there was one instance that was solved by algorithm V2 only.

Table 4 Test 4: comparing algorithm variants V4 (first order slopes, second order centered forms, gradient updates from second order data, centered form pruning) and V5 (V4 with corner-based slope centers)

solved by	sm.	med.	lg.	tot.	V4 (av.)	eq.	V5 (av.)
both	85	11	1	97	29 (0.61)	48	20 (0.55)
only V4	1	0	0	1	–	–	–
only V5	0	1	0	1	–	–	–
neither	24	32	9	65	14 (0.76)	45	6 (0.87)

Our decision about selecting what version to proceed with finally came after analyzing the improvements of V3: it turned out that the reason for the better behaviour of V3 was caused essentially not by the better interval enclosures, but instead, by the better capability of updating $\tilde{f}$ through more frequent point evaluations. We found that this gain will quickly disappear on sophisticated algorithm variants that target a fast update of $\tilde{f}$ (see Sec. 6), but the extra function evaluation can burden the bicenter-based algorithm on harder problems. Thus, in the sequel we will stick to version V2; noting that the bicentered form can still be an interesting algorithmic switch for the experimenting user, which may cause gains in some cases.

5.3 Algorithm variants with first order slopes

5.3.1 First order slopes with midpoint and corner centers. Next we discuss some comparative tests on algorithm variants that use slopes (Sec. 4.5 and 4.7) in place of interval gradients and Hessians.

In **Test 4** we compare the following two algorithm variants: version **V4**, that consists of the original GOP_ex algorithm, with interval gradients exchanged with first order slopes, and the additional minor algorithmic elements of Sec. 5.2.2. (Note that the latter tests are all also applicable in the slope context; furthermore their overall performance improvements were similar to those observed for the interval gradient versions, hence, are not detailed here.) This version uses the midpoints as the slope centers. Version **V5** is identical to V4, except that the midpoints are replaced with box corners, based on the heuristic method presented in Sec. 4.5. Note that both variants use the interval Hessians for second order evaluations. The results are given in Table 4.

Our first observation is that, in contrast to the comparisons of the previous section, the preferred algorithm variant becomes much less predictable when slopes are used. In any case, the results show a clear advantage of V4 against V5, i.e., one can see no clear gain by switching from midpoints to corners. A reason for this is that the interval gradient needed for picking the 'best' corner is often coming only from the parent of the current box, thus it is valid but somewhat outdated.

Table 5 Test 5: comparing algorithm variants V4 (first order slopes, second order centered forms, gradient updates from second order data, centered form pruning) and V6 (V4 with slopes of derivatives)

solved by	sm.	med.	lg.	tot.	V4 (av.)	eq.	V6 (av.)
both	86	11	1	98	11 (0.64)	49	38 (0.57)
only V4	0	0	0	0	–	–	–
only V6	2	1	0	3	–	–	–
neither	22	32	9	63	10 (0.75)	45	8 (0.70)

Table 6 Test 6: comparing algorithm variants V2 (interval gradients and Hessians) and V6 (first order slopes, slopes of derivatives)

solved by	sm.	med.	lg.	tot.	V2 (av.)	eq.	V6 (av.)
both	88	12	1	101	18 (0.59)	66	17 (0.58)
only V2	0	0	0	0	–	–	–
only V6	0	0	0	0	–	–	–
neither	22	32	9	63	12 (0.74)	45	6 (0.73)

5.3.2 Slopes of derivatives vs. interval Hessians. The next, important test is devoted to check the differences between our two main methods to acquire second order information. In **Test 5**, we compare version **V4** above with version **V6**, obtained by switching from interval Hessian-vector products to slope evaluations on the gradient expressions acquired from the KJ conditions (Sec. 4.7).

Table 5 shows the clear advantage of slopes of derivatives over the interval Hessians. There may be two explanations of this: first, as mentioned before, the slope matrix $S_f[\boldsymbol{x}, c]$ is usually smaller in width than $\nabla^2 \boldsymbol{f}(\boldsymbol{x})$, so the second order updates and the interval Newton step may work more efficiently. Second, according to Sec. 4.8, the interval gradient can be acquired during the second order calculation, which enables us to apply the usually very efficient monotonicity test also for the slope based algorithm. Consequently, we propose V6 as the most promising algorithm variant involving slopes.

5.4 Interval vs. slope based algorithms

In this section we present the comparison test (**Test 6**) between the best found algorithm of Sec. 5.2 and 5.3, namely, **V2** and **V6**. The difference between the two variants is that V2 has interval gradient and interval Hessian evaluations, while V6 has the respective slope evaluations for both orders.

The results of Table 6 show that the improvements reached along the interval gradient and the slope experiments of Sec. 5.2 and 5.3 are almost

Table 7 Test 7: comparing algorithm variants V2 (interval gradient/Hessians) and V7 (interval gradients, slopes of derivatives)

solved by	sm.	med.	lg.	tot.	V2 (av.)	eq.	V7 (av.)
both	88	12	1	101	2 (0.77)	72	27 (0.67)
only V2	0	0	0	0	–	–	–
only V7	0	0	1	1	–	–	–
neither	22	32	8	62	0 (–)	59	3 (0.80)

Table 8 Test 8: comparing algorithm variants V6 (first order slopes, slopes of derivatives) and V7 (interval gradients, slopes of derivatives)

solved by	sm.	med.	lg.	tot.	V6 (av.)	eq.	V7 (av.)
both	88	12	1	101	9 (0.66)	66	26 (0.62)
only V6	0	0	0	0	–	–	–
only V7	0	0	1	1	–	–	–
neither	22	32	8	62	3 (0.94)	44	15 (0.78)

equal. (There is a slight advantage of the interval gradient version, mainly due to its better performance on the unsolved problems.) A natural question arising here is whether it is possible to combine the advantages of these two versions by mixing the most promising algorithmic elements. More specifically, if switching from interval Hessians to slopes of derivatives turned out to help improving the first order slope-based method, then is it possible that the same switch would improve the interval gradient method as well? This question led us to a mixed interval derivative – slope based method, version **V7**: V7 is the same as V2, except that the interval Hessians are replaced with the slopes of the derivatives.

Tables 7 and 8 compare algorithm **V7** with its 'parents' **V2** and **V6**, respectively, in **Test 7** and **Test 8**. The outcome is clear: in both comparisons V7 solved one more (large) problem instance, and proved to be better than either V2 or V6 in *all* performance indicator categories. Therefore, after investigating the possible ways of function evaluations and the combinations of the related tools, we will continue the comparison tests with the options included in version V7, namely, with backward AD, centered form pruning, interval Hessians computed from the slopes of the derivatives, second order centered forms, and gradient updates from second order information.

6 Improving the GOP_ex algorithm in COCONUT II. – new algorithmic elements

In the next steps we further improve the interval B&B method starting from V7, with new algorithmic elements.

Table 9 Test 9: comparing algorithm variants V7 and V8 (V7 + local search)

solved by	sm.	med.	lg.	tot.	V7 (av.)	eq.	V8 (av.)
both	88	12	2	102	2 (0.78)	84	16 (0.78)
only V7	0	0	0	0	–	–	–
only V8	0	0	0	0	–	–	–
neither	22	32	8	62	0 (–)	50	12 (0.81)

6.1 Local optimization

The primary goal of adding a local search method to an interval B&B algorithm is to update $\tilde{f}$, the best known upper bound of the global minimum; with a better $\tilde{f}$ value the bound test and the cut-off test are working more efficiently. The candidate minimizer returned by the solver is checked, and if necessary, modified to by surely feasible (this is easy when only bound constraints are present), then it is extended to a point interval on which an interval function evaluation is done. Another tool that utilizes a good $\tilde{f}$ value is constraint propagation; this will be introduced in Sec. 6.3.

COCONUT has several local solvers as inference modules and it contains a heuristic prediction mechanism for selecting an appropriate solver based on the problem characteristics. For more details, see [16]. In that work we found that for bound constrained problems the limited memory quasi-Newton solver LBFGS-B by Byrd et al. [3] is the one with the best performance in almost all cases. Thus, in the present study we do not invoke the solver selection algorithm, just call LBFGS-B for all test problems.

A trade-off to resolve here is related to the frequency of invoking local search: rare calls to it may slow down the update, but calling it too often may become time-consuming. The strategy we included in the present method is that whenever $\tilde{f}$ is improved by the evaluation of a center, a local search is run from this point – guessing that the algorithm discovered a basin of attraction of a new local minimizer.

Test 9 compares version **V7** from the previous section, and version **V8**, with the local search added (Table 9). As expected, a significant improvement can be observed for both solved and unsolved problems by adding local search, while a slowdown was experienced in two problems only.

It is worth to mention that in 80% of the problems the local search was started only once; this means that the algorithm was able to locate the global minimizer easily (at least on the solvable problems). Although the test set is known to contain many problems with a large number of local minimizers, there were only six problems with at least four calls to the local search. The largest number of calls was eight, occurring for one problem only. Our consequence is that since the cost of local optimization is very small relative to the main algorithm, it may be worth to enable a more

thorough exploration of the search space initially and/or on problems on which the B&B algorithm shows small or no improvement.

6.2 Exclusion and inclusion boxes

Interval B&B methods for global optimization often suffer from the difficulty that subboxes near the global or local minimizers cannot be easily eliminated. The resulting excessive subdivision generated in these regions is called the *cluster effect* [10].

A possible solution to this problem is the construction of so-called *exclusion boxe*s that are guaranteed to contain a unique minimizer, optionally together with an *inclusion box* that is a high precision enclosure of the particular minimizer. Then, the set of points forming the difference of the exclusion and inclusion box can be discarded from each subbox. (In particular, if a subbox is contained in the exclusion box and contains the inclusion box, then it can be shrinked to the inclusion box.)

A method for creating exclusion boxes, known to behave good for well-conditioned minimizers is called backboxing. The backboxing method implemented in COCONUT uses the interval Newton step to test the contraction of a box (and thus, the uniqueness of a stationary point in its interior). Care is taken to limit the number of calls to the Newton step (and thus, the number of Hessian evaluations needed) in order to stay within a reasonable runtime budget. The sketch of the algorithm is the following:

1. Start from an approximate local minimizer $\tilde{x}$, returned, e.g., by local search, and create $\boldsymbol{x}_{out}$ as the largest possible symmetric box centered at $\tilde{x}$ and contained in $\boldsymbol{x}_0$.
2. Create $\boldsymbol{x}_{in}$ with center $\tilde{x}$ and width $\mathrm{wid}(\boldsymbol{x}_{in}) = \mathrm{wid}(\boldsymbol{x}_{out})/10^4$. (We consider $\boldsymbol{x}_{in}$ as the smallest possible exclusion box with a reasonably large size.)
3. Run a Newton step on $\boldsymbol{x}_{out}$; if $\boldsymbol{x}_{out}$ contracts *and* the bounds of $\boldsymbol{x}_0$ are softcoded (e.g., set for unconstrained problems in this study), then set $\boldsymbol{x}_{out} = \boldsymbol{x}_0 + [-\varepsilon, \varepsilon]$ with some small positive ε, and run the Newton step on $\boldsymbol{x}_{out}$, otherwise go to Step 5.
4. If again, a contraction is reached, then $\boldsymbol{x}_{excl} = \boldsymbol{x}_{out}$, and $\boldsymbol{x}_0$ can be shrinked to the inclusion box of the method (to be found later). Try the Newton step on $\boldsymbol{x}_{in}$ (presumably, $\boldsymbol{x}_{in}$ will contract), and go to Step 8.
5. If $\boldsymbol{x}_{out}$ does not contract in Step 3, then try the Newton step to $\boldsymbol{x}_{in}$.
6. If there is no contraction on $\boldsymbol{x}_{in}$ either, then STOP with a failure.
7. At this point, we have a contracting $\boldsymbol{x}_{in}$ and a non-contracting $\boldsymbol{x}_{out}$. Find the largest contracting box between these two iteratively: set a trial box $\boldsymbol{x}_t$ centered at $\tilde{x}$ with $\mathrm{wid}(\boldsymbol{x}_t) = (\mathrm{wid}(\boldsymbol{x}_{in}) \cdot \mathrm{wid}(\boldsymbol{x}_{out}))^{1/2}$. If contraction is reached on $\boldsymbol{x}_t$, then $\boldsymbol{x}_{in} = \boldsymbol{x}_t$, otherwise $\boldsymbol{x}_{out} = \boldsymbol{x}_t$. The iteration limit is set to 5 to bound the number of Hessian and Newton step calls. At the end of the iteration loop, let $\boldsymbol{x}_{excl} = \boldsymbol{x}_{in}$.

Table 10 Test 10: comparing algorithm variants V8 and V9 (V8 + exclusion/inclusion boxes)

solved by	sm.	med.	lg.	tot.	V8 (av.)	eq.	V9 (av.)
both	88	12	2	102	10 (0.80)	88	4 (0.26)
only V8	0	0	0	0	–	–	–
only V9	0	8	2	10	–	–	–
neither	22	24	6	52	0 (–)	51	1 (0.02)

8. Set the inclusion box $\boldsymbol{x}_{incl}$ to the smallest *contracted* box found so far. Refine it by successive Newton steps until either 5 steps are called, or $\mathrm{wid}(\boldsymbol{x}_{incl})$ drops below 10^{-6}.
9. Output $\boldsymbol{x}_{excl}$ and $\boldsymbol{x}_{incl}$ as the exclusion and inclusion boxes, resp.

An alternative method to construct exclusion regions consists of the existence and uniqueness tests from Schichl and Neumaier [24], based on the Krawczyk operator and the Kantorovich theorem. Recently these results have been further refined with the use of third order evaluation information, and a respective implementation was done in COCONUT [11,26]. We will present the numerical study of the latter methods in a forthcoming paper.

Table 10 displays the comparative results of **Test 10**, between version **V8** and the improved **version V9**, with the additional backboxing-based exclusion/inclusion box method. The most important observation is that there were ten new medium and large problem instances, that became solvable with V9. (Actually, these problems turned out to be well-conditioned convex problems, that were unsolvable with the 'careful' original application of the Newton step and the use of first order tools, but caused no difficulty to the Newton step when it was run on the whole search space.) Apart from these problems there were a few more ones on which the exclusion boxes helped significantly. The problems in which V9 proved to be slightly less efficient were only very easy-to-solve ones, where the total solution time was actually comparable with the time of the few additional Hessian evaluations.

6.3 Constraint propagation

Interval constraint propagation (CP) is a technique that attempts to reduce the current box in the presence of problem-related constraints, see e.g. [9, 18]. In particular, given a (factorable) expression $p(x)$, with known bounds for the variables and for the range of the expression itself (i.e., having $x \in \boldsymbol{x}$, and $\mathrm{rg}(p, \boldsymbol{x}) \subset \boldsymbol{p}(\boldsymbol{x})$), each $\boldsymbol{x}_i$ is tried to be tightened (iteratively) using $\boldsymbol{p}(\boldsymbol{x})$, and $\boldsymbol{x}_j$, $j \neq i$.

COCONUT contains a particularly efficient method for CP, designed specifically for computations on DAGs. It is called the PAID propagator, developed jointly by EPFL Lausanne and the University of Vienna [22,28].

Table 11 Test 11: comparing algorithm variants V9 and V10 (V9 + constraint propagation)

solved by	sm.	med.	lg.	tot.	V9 (av.)	eq.	V10 (av.)
both	88	20	4	112	7 (0.73)	46	59 (0.20)
only V9	0	0	0	0	–	–	–
only V10	7	1	1	9	–	–	–
neither	15	23	5	43	1 (0.95)	13	29 (0.40)

PAID is based on an iterative application of forward evaluation and backward propagation steps on the DAG. The steps are repeated until either the variable bounds become tight, a slow convergence or no more improvement in the reduction of the bounds is experienced, or infeasibility according to the specified constraints is proven.

Note, that although interval CP is developed for solving CSPs and general constrained optimization problems, it is still applicable in the present bound constrained context. First, since every global minimizer f^* has to satisfy $f^* \leq \tilde{f}$, we can pose the additional constraint $\sup(\boldsymbol{f}(\boldsymbol{x})) \leq \tilde{f}$. Technically, this means that the actual bound $\boldsymbol{f}(\boldsymbol{x})$ of the objective node is temporarily set to $[\inf(\boldsymbol{f}(\boldsymbol{x})), \tilde{f}]$ and CP is run on the resulting DAG. Second, CP is run also on the automatically generated KJ-constraints of the problem (Sec. 4.3).

After some experiments we found that since CP is a relatively cheap process (the total cost of one CP call is usually a small constant multiple of the cost of a function evaluation on the respective model functions), but with potentially large gains, it is worth to apply it in the present method in an aggressive fashion, namely, to run it every time in the `ProcessBox()` function, as its first step. In **Test 11**, the comparison of algorithm variants **V9**, and **V10** was made, where V10 was constructed by adding CP to V9.

According to Table 11, CP improves the performance of the algorithm substantially. Nine new problem instances were solved with the aid of CP. Actually, we have found that the average of the relative speedups $r = \max(t_{V10}, 0.01)/\max(t_{V9}, 0.01)$ and the average of the relative overall improvements $r = v_{V10}/v_{V9}$ (thus, the figures marked with 'av.' in the V10 columns) are not descriptive anymore: these averages are usually useful only when the individual ratios are close to 1. However, for CP the improvements are so large that many of the measurements fall into the range $0-0.1$. Thus, instead, we can consider the inverse r values, showing *how much faster the V10 variant, and how much bigger the overall progress of V10 as compared to V9*, respectively. Table 12 contains the average r^{-1} values computed for all solved problems (column 'all') and for all problems where V10 actually improved (column 'V10'). Thus, we found that *CP improves the algorithm so that it is on average 21 times faster on the solved problems, and it results in a 10 times increase in the overall progress on the unsolved ones.*

Table 12 r^{-1} values for comparing variants V9 and V10

solved by	all	V10
both	21.19	39.36
neither	10.22	14.68

Table 13 Test 12: comparing algorithm variants V0 and V10

solved by	sm.	med.	lg.	tot.	V0 (av.)	eq.	V10 (av.)
both	87	12	1	100	2 (0.65)	35	63 (0.17)
only V0	0	0	0	0	–	–	–
only V10	8	9	4	21	–	–	–
neither	15	23	5	43	0 (–)	11	32 (0.38)

Table 14 r^{-1} values for comparing variants V0 and V10

solved by	all	V10
both	45.15	71.09
neither	11.99	15.77

6.4 Summary of the algorithmic comparisons

As an overall comparison of the developed algorithm variants, in **Test 12** we show the improvement reached by the best proposed one, **V10**, compared to the initial GOP_ex variant **V0**. The comparisons and the respective r^{-1} values are given in Tables 13 and 14. According to the tables, we conclude that the additions provided by the COCONUT Environment resulted in 21 newly solved problem instances, increasing the success rate from 79% to 86% on small, from 27% to 48% on medium, and from 10% to 50% on large problems. In addition, with the exception of a few rare instances, a drastic speedup was reached in about two-third (63 out of 100) of the problems solved by both variants, and also, a substantial improvement in the overall progress was experienced in about 75% (32 out of 43) of the problems that still remained unsolved.

Comparing the respective entries of Tables 12 and 14, we arrive at the following main conclusions regarding the algorithmic improvements: *on the solvable problems, the techniques of Sec. 4 regarding function evaluations and the addition of local search and exclusion boxes altogether made the initial algorithm about twice as fast on average* (from the ratio of 45.15 and 21.19), *while CP alone resulted in an additional speedup of around 20 times. For unsolved problems, the gain of all additions except CP was about 15% only (the ratio of 11.99 and 10.22), while CP caused an about 10 times improvement in the overall progress.*

Table 15 Algorithm variants for the performance profile comparison

name	features
V0	basic GOP_ex algorithm implemented in COCONUT
V1	V0 with backward AD
V2	V1 + centered form pruning, second order centered forms, and gradient updates from second order information
V7	V2 with interval Hessians computed from slopes of derivatives
V8	V7 + local search
V9	V8 + exclusion/inclusion boxes
V10	V9 + constraint propagation

As our final comparison, demonstrating the improvements reached by the more and more sophisticated algorithm variants, we present a performance profile of the most relevant versions, namely, V0, V1, V2, V7, V8, V9, and V10, with respect to the running times. A summary of these algorithms is given in Table 15. The profile is created as follows: let us denote the execution time of algorithm A on problem P by $t_{A,P}$. To be able to create runtime ratios, we set $t'_{A,P} = \max(t_{A,P}, 0.01)$. If a problem is unsolved by an algorithm, then we assign $t'_{A,P} := \infty$. Then on each problem P, we compute a time performance ratio $r_{A,P}$ for all A: if a problem is unsolved by all algorithms, then let $r_{A,P} := \infty$ for all A. Otherwise, we determine the running time of the fastest successful algorithm on P, $t'_{best,P}$, and set the performance ratios as the particular solver runtime relative to $t'_{best,P}$, i.e., $r_{A,P} := t'_{A,P}/t'_{best,P}$, very similarly to the relative speedup measurements introduced in Sec. 5.2.1. To stay unbiased with the CPU time measurements, we declare an algorithm equal to the best one (and thus assign a performance ratio of 1 to it), if its running time is within the absolute and relative tolerances (0.02 seconds and 2%, respectively) defined as in Sec. 5.2.1.

Now, for each given performance ratio r and for each algorithm, one can count the number of problems on which *the performance ratio of the particular algorithm is at most* r. The performance profile graph of each algorithm, shown in Figure 1, is constructed by displaying the resulting relative problem frequencies evaluated for different r values. In particular, at $r = 1$ the profile shows the relative problem frequencies for which an algorithm performs the best (or equally best). This grows from around 13% (by V0) to around two third of the problems (by V10). On the other hand, at $r = \infty$ the graphs give the relative amount of problems solved by a given algorithm. This quantity grows from around 55% to about 75%. In general, for a fixed r value a data point shows the frequency for which the respective solver performs by at most r times worse than the best one. Thus, an algorithm is preferred against others if its performance profile graph is located higher.

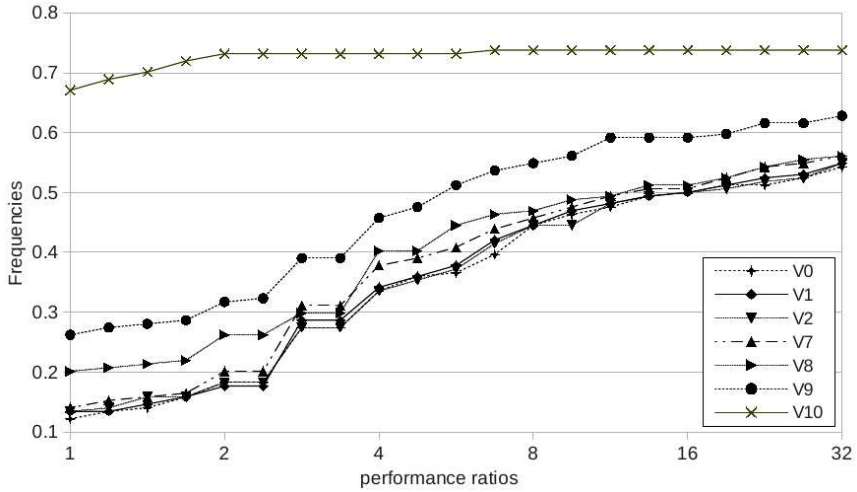

Fig. 1 Performance profile of the selected algorithm variants.

The performance profile shows the substantial advance of V10 against the other algorithms in all measurements. It also shows that the more basic algorithmic variants with different function and derivative methods and the related tools (i.e., improving to V1 and then to V2 from V0) result in relatively small performance increase, while adding local search (V8) and the exclusion/inclusion box method (V9) boost the algorithmic performance much more. Finally, note that for our other performance indicator, namely, for the remaining relative volume of the search space, a very similar performance profile can be drawn.

7 Comparisons with BARON

In this section, we compare the proposed competitive version V10 of our algorithm (called 'coco_gop_ex' in the sequel) with the BARON global solver. BARON (Branch and Reduce Optimization Navigator), the winner of the 2006 Beale-Orchard-Hays Prize, is developed by Nick Sahinidis's optimization group at the Carnegie Mellon University [21,27]. It is designed for general constrained global optimization including mixed integer programming; it has a branch–and–bound framework with sophisticated bound reduction techniques incorporating interval analysis, convex relaxations and duality. Due to its successes in numerous real-life applications and in various optimization benchmarks, BARON is considered by many as the currently best available deterministic global solver.

When running BARON on the test problems, the absolute tolerance regarding the global optimum value was set to 10^{-3} (as we will see later, this is only similar, but not equivalent to the stopping criterion of coco_gop_ex);

and the time limits were set to the same values as discussed in Sec. 5. For the other settings we used the BARON defaults. Table 16 contains a summary of the comparison. Since BARON cannot deal with trigonometric expressions, it accepted only 130 out of the total of 164 problems; the table is thus grouped by the respective acceptance categories. According to the table we can observe that on the accepted problems BARON performs slightly better, with 107 vs. 88 solved problems in total. It is significantly better in the medium category, where there are actually a bunch of problems with similar characteristic – nonlinear least squares problems with exponential and rational terms – that caused difficulties to coco_gop_ex. However, coco_gop_ex was more successful in the large problem category, and it was able to solve a total of seven problems (at least one in each size category) that BARON could not cope with. Furthermore, coco_gop_ex was very efficient on the problem set that BARON did not accept: it solved 33 out of the 34 problems.

On problems solved by both software systems, a runtime performance comparison can be carried out using the r relative speedup values introduced in Sec. 5.2.1. Similarly to those earlier tests, we again used an absolute difference of 0.02 seconds marking 'equal' performance with $r := 1$. Otherwise we set $t'_{BARON} = \max(t_{BARON}, 0.01)$, $t'_{coco\_gop\_ex} = \max(t_{coco\_gop\_ex}, 0.01)$ as before, and evaluated $r = t'_{BARON}/t'_{coco\_gop\_ex}$. A histogram of the $log_{10}(r)$ values are displayed by the dark gray bars of Figure 2. The logarithmic ratios show a clear advantage of our method. In particular, on 15 instances coco_gop_ex was faster than BARON by about one magnitude or more, while it was slower by the same amount for only six problems. In two cases we observed the advantage of coco_gop_ex with two orders of magnitudes or more.

To obtain a descriptive statistic of the average improvement reached by coco_gop_ex, we compute the mean μ and standard deviation σ of the logarithmic ratios, cut off outliers by considering values in $[\mu - 3\sigma, \mu + 3\sigma]$ only, and compute the average μ' of the remaining values. Raising the result to the power of 10, we get $10^{\mu} = 1.45$ and $10^{\mu'} = 1.35$, thus, the average time ratio (relative to BARON) is $1/1.35 \approx 0.74$. That is, we can conclude that *coco_gop_ex is around* 26% *faster on average over the set of problems solved by both software systems.*

It is important to note that BARON attempts to bound the global minimum value within the prescribed tolerance value and to provide a candidate optimizer with the best found function value so far. Instead, coco_gop_ex performs an exhaustive search to find *all* possible global minimizers. Furthermore, it is easy to see that if the stopping tolerance on the global minimum for BARON is set to the same ε value as the stopping tolerance of coco_gop_ex on the boxes (i.e., $\mathrm{wid}(\boldsymbol{f}(\boldsymbol{y})) < \varepsilon$), then the width of the enclosure of the global minimum by coco_gop_ex will be at least as tight as that of by BARON: from Alg. 1, $\mathrm{wid}(\boldsymbol{m}) = \mathrm{wid}([\min_{y \in \mathcal{R}} \inf(\boldsymbol{f}(\boldsymbol{y})), \tilde{f}]) < \mathrm{wid}(\boldsymbol{f}(\boldsymbol{z})) < \varepsilon$, with $\boldsymbol{z} \in \mathcal{R}$. Thus, we can conclude that the output of coco_gop_ex is more informative than BARON regarding the global mini-

Table 16 Comparing coco_gop_ex against BARON

solved by	sm.	med.	lg.	tot.
on problems accepted by BARON (130)				
both	64	14	3	81
only coco_gop_ex	4	1	2	7
only BARON	10	15	1	26
neither	5	7	4	16
coco_gop_ex (total)	68	15	5	88
BARON (total)	74	29	4	107
on problems not accepted by BARON (34)				
only coco_gop_ex	27	6	0	33

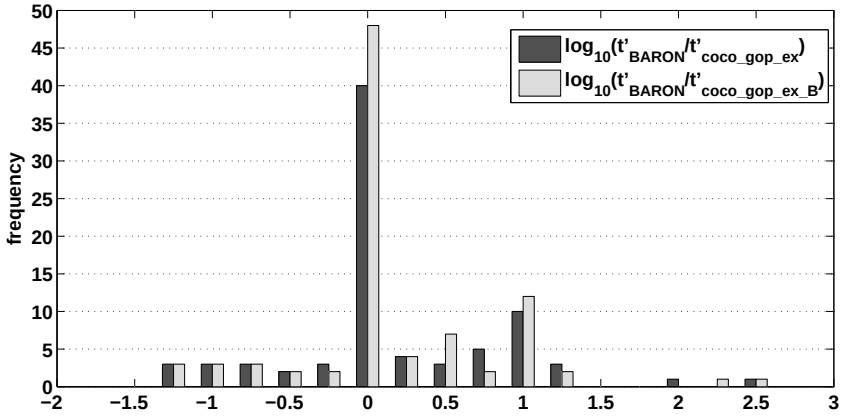

Fig. 2 Runtime comparisons between coco_gop_ex / coco_gop_ex_B and BARON, on problems solved by both methods.

mizers, and at least as informative as BARON regarding the global minimum.

The line of thoughts above leads to the natural conclusion that another comparison of coco_gop_ex and BARON can be made if we ignore exhaustiveness in coco_gop_ex and stop the search if wid($\boldsymbol{m}$) becomes smaller than ε, somewhat 'simulating' the behaviour of BARON. This modification can be easily done in the coco_gop_ex algorithm; we call this new algorithm 'coco_gop_ex_B'.

Table 17 contains the comparison of coco_gop_ex_B and BARON in the same way as shown in Table 16. The difference between the two comparisons is that coco_gop_ex_B solved nine previously unsolved instances, all of which were solvable by BARON. As a result, on the accepted set of problems

Table 17 Comparing coco_gop_ex_B against BARON

solved by	sm.	med.	lg.	tot.
on problems accepted by BARON (130)				
both	71	15	4	90
only coco_gop_ex_B	4	1	2	7
only BARON	3	14	0	17
neither	5	7	4	16
coco_gop_ex_B (total)	75	16	6	97
BARON (total)	74	29	4	107
on problems not accepted by BARON (34)				
only coco_gop_ex_B	27	6	0	33

coco_gop_ex_B had better success rates in both the small and the large classes than BARON, with the overall success difference decreasing to 10 problems (97 vs. 107). On the problem set that BARON did not accept, coco_gop_ex_B performed similar to coco_gop_ex, by solving 33 out of the 34 problems.

The runtime performance comparison over all problems solved by both methods is displayed by the light gray bars of Figure 2. The histogram shows essentially the same results as we experienced for coco_gop_ex. *The average improvement of coco_gop_ex_B as compared to BARON was found to be approximately 21%.*

8 Conclusion

We developed coco_gop_ex, an interval B&B solver for bound constrained global optimization, written in the COCONUT Environment. The solver contains a wide range of state-of-the-art interval methods with numerous options w.r.t. interval evaluations and algorithmic accelerating tools. It is a completely rigorous solver which is still competitive in speed with BARON on the bound constrained problem class, showing significant advantages in many cases, and it possesses a large potential for further improvements. These improvements (third order tools, advanced exclusion box methods, etc.) will be investigated in our ongoing and future studies.

Acknowledgments

We are grateful to Nick Sahinidis (Carnegie Mellon University) and to Michael R. Bussieck (GAMS) for providing us licenses to BARON and GAMS to carry out our numerical tests. We also thank Arnold Neumaier

(University of Vienna) for his numerous suggestions during the development of our software.

References

1. E. Baumann, Optimal Centered Forms, BIT Numer. Math. **28** (1988), 80–87.
2. C. Bliek, *Computer methods for design automation.* PhD thesis, Dept. of Ocean Engineering, Massachusetts Institute of Technology, 1992.
3. R.H. Byrd, P. Lu, J. Nocedal, and C. Zhu, A limited memory algorithm for bound constrained optimization, SIAM J. Sci. Comput. **16** (1995), 1190–1208.
4. COCONUT Environment. Software, University of Vienna. `http://www.mat.univie.ac.at/coconut-environment`.
5. T. Csendes and D. Ratz, Subdivision Direction Selection In Interval Methods For Global Optimization, SIAM J. Numer. Anal. **34** (1997), 922–938.
6. A.H. Gebremedhin, F. Manne, and A. Pothen, What Color Is Your Jacobian? Graph Coloring For Computing Derivatives, SIAM Rev. **47** (2005), 629–705.
7. A. Griewank and G. F. Corliss, *Automatic Differentiation of Algorithms.* SIAM Publications, Philadelphia, 1991.
8. R. Hammer, M. Hocks, U. Kulisch, and D. Ratz, *C++ Toolbox for Verified Computing.* Springer-Verlag, Heidelberg, New York, 1995.
9. P. Van Hentenryck, Numerica: A Modeling Language for Global Optimization. In: Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI- 97), 1997.
10. R.B. Kearfott and K. Du, The cluster problem in multivariate global optimization, J. Global Optim. **5** (1994), 253–265.
11. M. Kieffer, M. C. Markót, H. Schichl, and E. Walter. Verified global optimization for estimating the parameters of nonlinear models. In: Modeling, Design, and Simulation of Systems with Uncertainties (ed.: A. Rauh and E. Auer), Chapter 7, Mathematical Engineering, Springer, 2011.
12. L.V. Kolev, Use of interval slopes for the irrational part of factorable functions, Reliab. Comput. **3** (1997), 83–93.
13. M.C. Markót and T. Csendes, A New Verified Optimization Technique for the "Packing Circles in a Unit Square" Problems, SIAM J. Optim. **16** (2005), 193–219.
14. M.C. Markót, T. Csendes, and A.E. Csallner, Multisection in Interval Branch-and-Bound Methods for Global Optimization II. Numerical Tests, J. Global Optim. **16** (2000), 219–228.
15. M.C. Markót, J. Fernández, L. G. Casado, and T. Csendes, New interval methods for constrained global optimization, Math. Program. **106** (2006), 287–318.
16. M.C. Markót and H. Schichl, Comparison and automated selection of local optimization solvers for interval global optimization methods, SIAM J. Optim. **21** (2011), 1371 –1391.
17. R.E. Moore, R.B. Kearfott, and M.J. Cloud, *Introduction to Interval Analysis.* SIAM, Philadelphia, USA, 2009.
18. A. Neumaier, The enclosure of solutions of parameter-dependent systems of equations. In: Reliability in Computing (ed. by R.E. Moore). Acad. Press, San Diego, 269–286, 1988.

19. A. Neumaier, *Interval Methods for Systems of Equations.* Vol. 37 of Encyclopedia of Mathematics and its Applications, Cambridge Univ. Press, Cambridge, 1990.
20. H. Ratschek, Centered Forms, SIAM J. Numer. Anal. **17**, (1980), 656–662.
21. N. V. Sahinidis and M. Tawarmalani, BARON 9.0.4: Global Optimization of Mixed-Integer Nonlinear Programs, User's manual, 2010. Available at `http://www.gams.com/dd/docs/solvers/baron.pdf`.
22. H. Schichl and A. Neumaier, Interval Analysis on Directed Acyclic Graphs for Global Optimization, J. Global Optim. **33** (2005), 541–562.
23. H. Schichl, *Mathematical Modeling and Global Optimization.* Habilitation Thesis, draft of a book, Cambridge Univ. Press, to appear.
24. H. Schichl and A. Neumaier, Exclusion regions for systems of equations, SIAM J. Numer. Anal. **42** (2004), 383–408.
25. H. Schichl and M. C. Markót, Algorithmic Differentiation Techniques for Global Optimization in the COCONUT Environment, Optim. Methods Softw. **27** (2012), 359–372.
26. H. Schichl, M. C. Markót, and A. Neumaier, Exclusion Regions for Optimization Problems. In preparation.
27. M. Tawarmalani and N. V. Sahinidis, A polyhedral branch-and-cut approach to global optimization, Math. Program. **103** (2005), 225–249.
28. X-H. Vu, H. Schichl, and D. Sam-Haroud, Interval Propagation and Search on Directed Acyclic Graphs for Numerical Constraint Solving, J. Global Optim. **45** (2009), 499–531.