An Interval Filtering Operator for Upper and Lower Bounding in Constrained Global Optimization

Olivier Sans*, Remi Coletta*† and Gilles Trombettoni*
* *LIRMM, University of Montpellier, France*
† *Tellmeplus, Montpellier, France*
{*firstname.lastname*}*@lirmm.fr*

***Abstract*—This paper presents a new interval-based operator for continuous constrained global optimization. It is built upon a new filtering operator, named TEC, which constructs a bounded subtree using a Branch and Contract process and returns the parallel-to-axes hull of the leaf domains/boxes. Two extensions of TEC use the information contained in the leaf boxes of the TEC subtree to improve the two bounds of the objective function value: (i) for the lower bounding, a polyhedral hull of the (projected) leaf boxes replaces the parallel-to-axes hull, (ii) for the upper bounding, a good feasible point is searched for inside a leaf box of the TEC subtree, following a look-ahead principle. The algorithm proposed is an auto-adaptive version of TEC that plays with both extensions according to the outcomes of the upper and lower bounding phases. Experimental results show a significant speed-up on several state-of-the-art instances.**

I. Introduction

Interval $Branch\,\&\,Bound$ algorithms are used to solve constrained global optimization problems[1] in a reliable way [1], [2], [3], [4], [5]. They interleave branching, filtering and bounding steps for calculating an optimal solution and its cost with a bounded error or a proof of infeasibility. The branching step selects a variable and splits its domain into two sub-domains. To limit the combinatorial explosion, the filtering step can contract the domains without loss of solutions. In several interval $Branch\,\&\,Bounds$, the objective function value is handled as an additional variable, so that the filtering step is also used for finding a good lower bound, i.e. a point such that no other feasible point exists with a cost lower than it. The lower bounding step is used to prove the optimality of the solution.

In this paper, we first introduce a new filtering operator, called `TEC` (Tree for Enforcing Consistency), for contracting the domains and improving the lower bound of the objective function value. `TEC` is a generalization to several dimensions of the constructive disjunction used by the operator `CID` [6]. It constructs a subtree, bounded in size by a parameter, by using a $Branch\ \&\ Contract$ process, and returns a parallel-to-axes *hull* of the leaf boxes. Two extensions of `TEC` exploit the information contained in the leaf boxes of the TEC subtree. In a first extension, the basic parallel-to-axes hull is replaced by a polyhedral hull of the (projected) leaf boxes for improving the filtering capacity of `TEC` and the lower bounding. This polyhedral hull is achieved by a rigorous variant of the Graham computational geometry algorithm. A second extension of `TEC` focuses on the *upper bounding* step of global optimization, which searches for a feasible point with a cost better than the cost of the best solution found so far. This extension searches for a good feasible point inside a selected leaf box of the TEC subtree, thus following a look-ahead principle.

Finally, we introduce an advanced version of `TEC` endowed with both extensions. This `LGT` (Lazy Graham TEC) algorithm calls the standard hull or the Graham polyhedral hull algorithm in an adaptive way during the optimization process. When the upper bound has not been improved for long, `LGT` intensifies its effort for proving optimality: it tries to improve the lower bound of the objective function value by switching to the polyhedral hull of the TEC subtree during the next nodes. If a new upper bound is found, `LGT` switches back to the less time-consuming box hull, preferring investing in the upper bounding phase.

After presenting the background in Section II, `TEC` is described in Section III, and `LGT` is detailed in Section IV. Section V carries out an empirical study of these operators implemented in the `Ibex` interval library [7], and compares them to state-of-the-art operators on constrained optimization problems belonging to the Coconut benchmark [8]. Finally, Section VI presents the related work.

II. Background

An interval $[x_i] = [\underline{x_i}, \overline{x_i}]$ defines the set of reals x_i s.t. $\underline{x_i} \leq x_i \leq \overline{x_i}$, where $\underline{x_i}$ and $\overline{x_i}$ are floating-point numbers. $\mathbb{IR}$ denotes the set of all intervals. A box $[X]$ is a cartesian product of intervals $[x_1] \times ... \times [x_i] \times ... \times [x_n]$. $\omega([x_i])$ denotes the width $\overline{x_i} - \underline{x_i}$ of an interval $[x_i]$. The width of a box is given by the width of its largest dimension (i.e., $\max_{i=1..n}\ \omega([x_i])$). Since the union of several boxes is generally not a box, the **box hull** operator returns the minimal box enclosing all of them.

Interval arithmetic [9] has been defined to extend to $\mathbb{IR}$ elementary functions over $\mathbb{R}$. For instance, the interval sum is defined by $[x_1]+[x_2] = [\underline{x_1}+\underline{x_2}, \overline{x_1}+\overline{x_2}]$. When a function

[1]In this paper, we consider minimization, without loss of generality.

f is a composition of elementary functions, an extension of f to intervals must be defined to ensure a conservative image computation.

Definition 1: **Extension of a function** $f: \mathbb{R}^n \rightarrow \mathbb{R}$ **to** $\mathbb{IR}$. $[f] : \mathbb{IR}^n \rightarrow \mathbb{IR}$ is said to be an **extension** of f to intervals if:

$$\forall [X] \in \mathbb{IR}^n \qquad [f]([X]) \supseteq \{f(X), X \in [X]\}$$
$$\forall X \in \mathbb{R}^n \qquad f(X) = [f](X)$$

In our context, the expression of a function f is always a composition of **elementary functions** (e.g., sin, cos, $+$, $\times$, exp, log, ...). The natural extension $[f]_N$ is then simply a composition of the corresponding interval operators.

This paper deals with continuous global optimization under inequality constraints defined by:

$$\min_{X \in [X]} f(X) \ s.t.\ g(X) \leq 0 \qquad (1)$$

where $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is the (non convex) real-valued objective function and $g : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is a (non convex) vector-valued function. $X = (x_1, ..., x_i, ..., x_n)$ is a vector of variables varying in a box $[X]$. A point X is said to be *feasible* if it satisfies the constraints.

An interval $Branch$ & $Bound$ ($B\&B$) scheme for continuous constrained global optimization is described below. Algorithms 1 and 2 correspond to a simplified version of the algorithms implemented in `IbexOpt` [5].

The algorithm is launched with the vector of constraints g, the objective function f and with the input domain initializing a list $Boxes$ of boxes to be handled. ϵ_{obj} is the absolute or relative precision required on the objective function value and is used in the stopping criterion. The algorithm outputs a feasible floating point $X_{\widetilde{f}}$ of cost $\widetilde{f}$ and a proof that no feasible points can be found with a cost lower than f_{min}.

Let X_{f^*} and f^* be the optimal real point and its cost. The algorithm returns $f_{min} \leq f^* \leq \widetilde{f}$ with an absolute ($\widetilde{f} - f_{min} \leq \epsilon_{obj}$) or relative ($\frac{\widetilde{f} - f_{min}}{|\widetilde{f}|} \leq \epsilon_{obj}$) precision.

The variable x_{obj} (corresponding to the objective function value) is added to the problem (to the vector X of variables) with the constraint $f(X) = x_{obj}$. The algorithm maintains two main types of information during the iterations:

- $\widetilde{f}$: the value of the best feasible point $X_{\widetilde{f}}$ found so far,
- f_{min}: the minimal value of the lower bounds $\underline{x_{obj}}$ of the boxes to explore.

The lower bound f_{min} guarantees that, in every box, no feasible point exists with an objective function value lower than it. The procedure `SelectBox` selects the next node to handle. The selected box $[X]$ is then split into two sub-boxes $[X]_1$ and $[X]_2$ along one dimension (selected by a branching strategy). Both sub-boxes are then handled by the `Contract&Bound` procedure (see Algorithm 2).

A constraint $x_{obj} \leq \widetilde{f} - \epsilon_{obj}$ is first added to the problem for decreasing the upper bound of the objective function in

Algorithm 1: `IntervalBranch&Bound`$(f, g, X, box, \epsilon_{obj}, \epsilon_{sol})$

```
1  f_min ← −∞; f~ ← +∞
2  Boxes ← {box}
3  while Boxes ≠ ∅ and f~ − f_min > ε_obj and (f~ − f_min)/|f~| > ε_obj do
4      [X] ← SelectBox (Boxes, criterion)
5      Boxes ← Boxes\{[X]}
6      ([X]_1, [X]_2) ← Bisect ([X])
7      ([X]_1, X_f~, f~, Boxes) ← Contract&Bound ([X]_1, f, g,
         X, ε_obj, X_f~, f~, Boxes)
8      ([X]_2, X_f~, f~, Boxes) ← Contract&Bound ([X]_2, f, g,
         X, ε_obj, X_f~, f~, Boxes)
9      PushIfNonEmpty([X]_1, Boxes)
10     PushIfNonEmpty([X]_2, Boxes)
11     f_min ← min_{[X]∈Boxes} x_obj (lower bound)
```

the box. Algorithm 2 then contracts the handled box without loss of feasible part [7]. In other words, some infeasible parts at the bounds of the domain are discarded using filtering operators. First, `CPContract` uses a filtering operator which comes from constraint programming [10], [6], [1]. Then, `ConvexContract` uses a convexification algorithm [11] which relies on a polyhedral convex relaxation of the system to contract the box. These contractions work on the extended box including the objective function variable x_{obj} and on the associated constraint, so improving $\underline{x_{obj}}$ amounts to improving the lower bound of the objective function image (lower bounding).

Algorithm 2: `Contract&Bound`$([X], f, g, X, \epsilon_{obj}, X_{\widetilde{f}}, \widetilde{f}, Boxes)$

```
1  g' ← g ∪ {x_obj ≤ f~ − ε_obj}
2  [X] ← CPContract ([X], g' ∪ {f(X) = x_obj})
3  [X] ← ConvexContract ([X], g' ∪ {f(X) = x_obj})
4  if [X] ≠ ∅ then
5      (X_f~, cost) ← FeasibleSearch ([X], f, g', ε_obj)
6      if cost < f~ then
7          f~ ← cost
8          Boxes ← FilterOpenNodes (Boxes, f~ − ε_obj)
9  return ([X], X_f~, f~, Boxes)
```

The last part of the procedure carries out upper bounding. `FeasibleSearch` calls one or several heuristics searching for a feasible point $X_{\widetilde{f}}$ that improves $\widetilde{f}$, the best cost found so far. If the upper bound of the objective value is improved, the `FilterOpenNodes` procedure performs a type of garbage collector on all the open nodes by removing from $Boxes$ all the nodes having $\underline{x_{obj}} > \widetilde{f} - \epsilon_{obj}$. After the calls to `Contract&Bound`, Algorithm 1 pushes the two sub-boxes in the set $Boxes$ of open nodes.

Several filtering operators have been proposed by the CP community. The `HC4` algorithm [10] is a well-known constraint propagation algorithm. The `3B` algorithm [12], like `SAC` [13] for finite CSP, relies on the shaving principle to

prune a sub-interval on a bound of an interval. The shaving consists in first restricting an interval to a sub-interval on a bound. If the application of a constraint propagator, like HC4, on the corresponding subproblem leads to a wipeout, the sub-interval is removed from the domain. Otherwise, the sub-interval is kept and the pruning effort performed by the constraint propagator is lost. The constructive disjunction permits to exploit this pruning effort by performing the box hull of all the boxes calculated by the subproblems. CID uses this principle to improve the contraction. First, CID selects a variable x_i and splits its interval $[x_i]$ in s_{cid} sub-intervals. Then, each corresponding subproblem is contracted by a constraint propagator like HC4, and the box hull of the contracted boxes is performed. This process is achieved on each variable while a contraction is obtained. 3BCID [6] is an efficient algorithm that hybridizes 3B and CID. This state-of-the-art variant of CID is used in the experiments shown in Section V.

III. TEC

Contrarily to 3B, CID is able to prune values from all variable domains when it performs shaving tests on one of them. TEC generalizes the CID principle. TEC steps up the shaving process by shaving a subset of variables simultaneously and uses constructive disjunction to be able to contract all variable domains. Indeed, TEC constructs a bounded subtree, called here *TEC tree*, and performs the box hull of all generated leaves (boxes). The TEC tree is constructed in breadth-first search by a $Branch$ & $Contract$ process where each node is handled by constraint propagation. Algorithm 3 describes this process.

A. TEC Algorithm

Algorithm 3 is called at line 2 of Algorithm 2. It is launched with the vector of constraints, and an initial box to contract. The combinatorial process is achieved by the loop at line 4. First, the initial box is contracted by a constraint propagator Φ and is pushed into the list $Boxes$. At each iteration of the loop, the first box $[X]$ of $Boxes$ is extracted, following a "First In, First Out" principle. If the size of $[X]$ is inferior to a given precision ϵ_{sol}, $[X]$ is stored in $AtomicBoxes$ (line 7). Otherwise, $[X]$ is split into two sub-boxes along one dimension. Both are contracted by Φ and are pushed at the end of $Boxes$. This process is repeated until the number of contractions reaches the value given by the $TECnodes$ parameter or until $Boxes$ is empty. Finally, TEC returns the box hull of the leaf boxes of the TEC tree.

B. Implementation Choices for TEC

Several implementation choices have been validated by preliminary experiments. They have been obtained on a subset of the benchmark used in this paper (see Section V).

Algorithm 3: TEC$(box, g'', \epsilon_{sol}, TECnodes, \Phi)$

```
1  Boxes ← {Φ(box, g'', ε_sol)}   /* queue (FIFO) of boxes */
2  AtomicBoxes ← ∅                /* set of atomic boxes */
3  #nodes ← 1
4  while Boxes ≠ ∅ and #nodes ≤ TECnodes do
5      [X] ← Pop(Boxes)
6      if ω([X]) ≤ ε_sol then
7          Push([X], AtomicBoxes)
8      else
9          ([X]_1, [X]_2) ← Bisect([X])
10         PushIfNonEmpty(Φ([X]_1, g'', ε_sol), Boxes)
11         PushIfNonEmpty(Φ([X]_2, g'', ε_sol), Boxes)
12         #nodes ← #nodes + 2
13 return BoxHull(Boxes ∪ AtomicBoxes)
```

Variable Selection Heuristic

TEC selects a subset of variables to contract the box. Indeed, selecting a good subset of variables is very important to apply a strong contraction. The importance of a variable depends on constraints, but also on the box studied. Consequently, the importance of a variable changes during the search. Therefore we compared different dynamical variable selection heuristics available in Ibex [7]. The tests led us to choose the $SmearSumRel$[2] heuristic. This was not a surprise since $SmearSumRel$ is state-of-the-art both in constraint satisfaction and global optimization. Although it is not always the best, this strategy appears to be more robust than its competitors on the tested benchmark.

Subfiltering Operator

To choose the subfiltering operator Φ, we compared the different filtering operators available in Ibex. Experimental studies showed that using HC4 in TEC provides the best results, due to its low computational time.

TECnodes Value

Experiments showed that the most significant impact on performance is brought by the size of the TEC tree. We compared several values of *TECnodes* (see Algorithm 3) for each instance of our benchmark. Depending on the instance considered, the optimal value of $TECnodes$ varied from 10 to 100. Further work could investigate how to automatically adapt *TECnodes* according to the instance. Lacking such an adaptive strategy, we used 25 as a fixed value for all benchmarks in the following experiments, since we identify it as a robust value.

Overall, the different implementation choices led to a current version of TEC that is comparable in performance to 3BCID.

[2] $SmearSumRel$ is a variant of the well-known *smear function* [14]. The $smear(x_i, c_j)$ function measures an impact of the variable x_i on the function of c_j, depending on the width of x_i and on the partial derivative of the c_j function w.r.t. x_i evaluated on the box $[X]$.

IV. LGT

Although TEC is comparable in performance to the state-of-the-art filtering operator 3BCID, its tree (i.e., the TEC tree) contains additional information. LGT uses this additional information in an auto-adaptive way to better improve the two bounds of the objective function value. During the optimization process, LGT can switch between two versions of TEC, namely TEC-UB and Graham-TEC.

For improving the lower bounding and the contraction, Graham-TEC adds new linear constraints in the polytope generated by the polyhedral convexification algorithm, e.g. X-Newton (see the ConvexContract procedure in Algorithm 2). For the upper bounding, TEC-UB makes better use of the In-X-Newton procedure.

Let us first present the convexification algorithms X-Newton and In-X-Newton, available in our optimization code IbexOpt, for supporting the observation that led us to propose TEC-UB (see Section IV-B) and Graham-TEC (see Section IV-C).

A. Brief Description of X-Newton and In-X-Newton

X-Newton [11] is a convexification algorithm which relies on an outer interval linear form of the constraints to contract the box. It can implement the ConvexContract procedure called in Algorithm 2.

A polyhedral convexification is built upon an adaptation of a specific first order interval Taylor form of a nonlinear function [15]. Consider a function $g_j : \mathbb{R}^n \rightarrow \mathbb{R}$ defined on a domain $[X]$. For any variable $x_i \in X$, let $[a_i] = \left[\frac{\partial g_j}{\partial x_i}\right]_N ([X])$ be the natural interval evaluation of the partial derivative of g_j w.r.t. x_i. The idea is to (lower) tighten $g_j(X)$, for every point X in $[X]$, using an interval Taylor form $g_j^l(X)$ expanded at a *vertex* of the box, e.g. $\underline{X}$.

$$\forall X \in [X]\; g_j^l(X) = g_j(\underline{X}) + \sum_i \underline{a_i}\,(x_i - \underline{x_i}) \leq g_j(X) \quad (2)$$

Thus, considering an inequality constraint $g_j(X) \leq 0$, Expression (2) defines a hyper-plane $g_j^l(X) \leq 0$ bounding the solution set from below: $g_j^l(X) \leq g_j(X) \leq 0$. Note that, contrarily to the usual case where the Taylor expansion point is taken in the middle of the box, this interval Taylor form is polyhedral because for all x_i, $(x_i - \underline{x_i}) \geq 0$.

X-Newton defines a hyper-plane for the objective function, and for each constraint to obtain a polytope enclosing the solution set. Then, X-Newton contracts the interval of each variable x_i by calling a Simplex algorithm twice, improving potentially both bounds of $[x_i]$. Improving $\underline{x_{obj}}$ performs lower bounding.

In-X-Newton [5] is a heuristic searching for a feasible point (see Algorithm 2) which relies on an *inner* interval linear form computing an *inner region*. An inner region r^{in} is a feasible subset of the box $[X]$, i.e., $r^{in} \subset [X]$ and all points $X \in r^{in}$ satisfy all the constraints $g(X) \leq 0$. The inner regions computed by In-X-Newton are polytopes. Symmetrically to Eq. (2):

$$\forall X \in [X]\; g_j^l(X) = g_j(\underline{X}) + \sum_i \overline{a_i}\,(x_i - \underline{x_i}) \geq g_j(X) \quad (3)$$

In-X-Newton defines a hyper-plane for each constraint to obtain an inner region. Then, In-X-Newton defines a hyper-plane for the objective function and calls a Simplex algorithm minimizing x_{obj} to find the best solution inside the inner region. This solution is also a solution of the original system. Even if it is generally *not* the optimal one, it can be used to update the upper bound.

Both algorithms use an outer (resp. inner) interval linear form. In fact, the generated polytope can greatly overestimate (resp. underestimate) the set of solutions. In some cases, the overestimation may produce a polytope larger than the box, and the underestimation may produce an empty inner region even if the box contains solutions. In practice, these cases occur more often in large boxes. This is mainly explained by the first order interval Taylor form. Close to the expansion point (e.g., $\underline{X}$ in Eq. (2)), the approximation is tight, even exact at the expansion point. Thus, the smaller the box is, the sharper will be the approximation of the solution set.

B. TEC-UB

To improve the upper bounding, TEC-UB simply calls In-X-Newton to search for a good feasible point in a *small* leaf box of the TEC tree (see Fig. 1). Thus, In-X-Newton is called twice:

- once on the contracted hull box (like in the standard interval $B\&B$), and
- another time on the selected leaf box of the TEC tree.

As explained in Section IV-A, the smaller the box is, the smaller is the underestimation. Therefore, TEC-UB selects the smallest box in the leaf boxes of the TEC tree (i.e., $\min_{[X]} \omega([X])$). To break ties, the selection is made using the objective function, i.e. minimizing $\overline{x_{obj}}$, for finding a better solution.

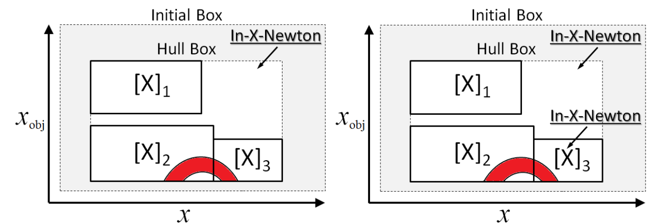

Figure 1. Difference between TEC and TEC-UB on a system with one variable x and the objective variable x_{obj}. The region removed by contraction appears light greyed out and the set of solutions is dark greyed out (or red). $[X]_1$, $[X]_2$ and $[X]_3$ are the leaf boxes of the TEC tree. **Left:** Just one call to In-X-Newton is achieved on the contracted box (i.e., the box hull of the leaf boxes). **Right:** Two calls to In-X-Newton are performed, the second one on the selected leaf box $[X]_3$.

C. Graham-TEC

Another possible way to improve the outer approximation of the solution set is to add hyper-planes to the polytope generated in ConvexContract. Graham-TEC uses the additional information contained in the leaf boxes of the TEC tree to learn implicit binary constraints. In other words, Graham-TEC:

1) constructs the TEC tree in the same way as TEC,
2) computes, for each pair of variables, a polyhedral convex hull of the leaf boxes of the TEC tree (see Algorithm 4),
3) transmits to the ConvexContract operator the polytope made of these binary constraints and the box hull of the TEC tree leaves.

Finally, ConvexContract (e.g., X-Newton, Quad [16]) enriches its polytope with the additional constraints given in input, thus computing generally a sharper contracted box than TEC would do.

Convex Hull of the TEC Tree Leaves

Let us detail hereafter how the polyhedral convex hull of boxes is computed. Algorithm 4 is called at the end of TEC (see Algorithm 3). The algorithm is launched with the set L of the TEC tree leaf boxes and with the box hull $[H]$ of these leaf boxes. γ is a parameter used to decide whether a constraint learned is kept or not. γ measures a gain relative to each interval size of $[H]$.

Algorithm 4: ConvexHull(L, $[H]$, γ)

```
1  S ← ∅                     /* a set of linear constraints */
2  foreach (x_i, x_j) ∈ X^2 with i < j do
3      NE ← ∅; NW ← ∅; SW ← ∅; SE ← ∅ /* quadrant */
4      foreach [X] ∈ L do
5          NW ← NW ∪ (lb(x_i), ub(x_j)); NE ← NE ∪ (ub(x_i), ub(x_j))
6          SW ← SW ∪ (lb(x_i), lb(x_j)); SE ← SE ∪ (ub(x_i), lb(x_j))
7      foreach quadrant ∈ {NE, NW, SW, SE} do
8          if ¬Cover(quadrant, [H], γ) then
9              Front ← GrahamFront(quadrant)
10             S ← S ∪ GetConstraints(Front, x_i, x_j, quadrant)
11 return S
```

For each pair of variables (x_i, x_j), Algorithm 4 constructs the convex hull of the leaf boxes projected on (x_i, x_j) and computes linear constraints from this (2-dimensional) convex hull. The projection of a box on (x_i, x_j) is a rectangle, in which each vertex belongs to a quadrant (see Fig. 2). Thus, at lines 5 and 6, the 4 points of each projected leaf box are pushed into 4 lists corresponding to the 4 quadrants. Then, the 2-dimensional convex hull is constructed quadrant by quadrant. Some quadrants are rapidly treated. Let $[H]_{ij}$ be the rectangle projected from $[H]$ onto the variables x_i and x_j. When a vertex of $[H]_{ij}$ and a point of a projected leaf box in L coincide, no constraint can be found on this (covered) quadrant, as shown in Fig. 2. It is therefore useless

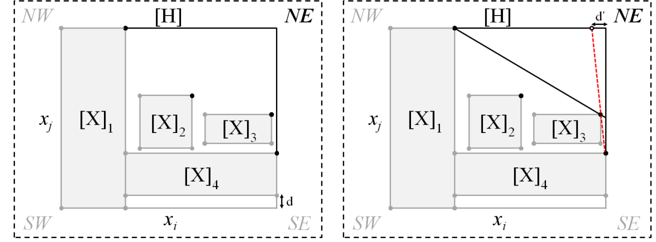

Figure 2. Illustration of ConvexHull on the projection of the leaf boxes $[X]_1$,...,$[X]_4$ and the hull box $[H]$ on (x_i, x_j). NW, SW, SE and NE are the 4 quadrants. **Left:** The quadrants NW and SW are covered. SE is also covered since on the x_j-axis we have $d < \gamma \times \omega([x_j])$, and on the x_i-axis we have $0 < \gamma \times \omega([x_i])$. Only the 4 points of NE are handled to learn constraints, the others are ignored. **Right:** The two lines represent the two constraints learned. The dashed one does not give a sufficient gain on one of the two dimensions ($d' < \gamma \times \omega([x_i])$). It is not learned.

to handle the points of this quadrant. Moreover, we consider that a quadrant is also covered if a point of a projected leaf box is too close (on each dimension) to the corresponding vertex of $[H]_{ij}$ (see Fig. 2). The idea is to learn only impacting constraints.

Then, the GrahamFront function, based on the Graham Scan algorithm, is called on each non covered quadrant. Graham Scan is a classical computational geometry algorithm that constructs a convex hull of given 2-dimensional points [17]. GrahamFront just computes the front of the convex hull relative to the quadrant studied and returns the corresponding list of points, in counterclockwise order. We call this list of points the front of a quadrant. Let r be the number of leaf boxes, the complexity of GrahamFront is dominated by the sort (counterclockwise order) in $O(r.log(r))$ [17].

Finally, GetConstraints is called to compute the constraints corresponding to the front of a quadrant. Still with the objective of only keeping impacting constraints, when a constraint does not give a significant gain (see Fig. 2), it is removed. The impacting constraints are added to the ConvexContract polytope.

Computation of Rigorous Constraints

The algorithm GetConstraints (see Algorithm 5) generates rigorous constraints from the front of a quadrant. Each pair of consecutive points $((x_i, y_i), (x_{i+1}, y_{i+1}))$ of the $Front$ list is iteratively handled. The goal is to rigorously approximate the line $y = ax + b$ that passes through the 2 points. Indeed, $a = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$ and $b = y_i - (x_i \times a)$ are generally not floating-point numbers. The round-off error may generate a constraint (halfspace) which does not include one of both points. To avoid this drawback, a and b are rigorously calculated using interval arithmetic [9]. From the two points (x_i, y_i) and (x_{i+1}, y_{i+1}), interval arithmetic calculates the intervals $[a]$ and $[b]$ enclosing the respective real values. Then, to generate a rigorous constraint, the

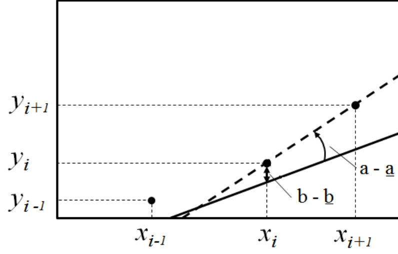

Figure 3. Rigorous calculation on the quadrant SE. The dashed line is the one with a, $b \in \mathbb{R}$. The second line corresponds to the rigorous approximation of the first line with a and b rounded to the floating-point numbers $\underline{a}$ and $\underline{b}$.

adequate bound of $[a]$ and $[b]$ is selected depending on the quadrant (see line 7 to line 13). The reader will be able to easily check the choice of bounds. Fig. 3 shows the principle in the quadrant SE.

Algorithm 5: `GetConstraints`$(Front, x, y, quadrant)$

1 $P \leftarrow \emptyset$ `/* a set of rigorous hyperplanes */`
2 **foreach** $(x_i, y_i) \in Front$ **do**
3 $[a] \leftarrow ([y_{i+1}] - [y_i])/([x_{i+1}] - [x_i])$
4 $[b] \leftarrow [y_i] - ([x_i] \times [a])$
5 **switch** $quadrant$ **do**
6 **case** *SE*
7 $p \leftarrow \underline{a} \times x - y \leq -\underline{b}$
8 **case** *NE*
9 $p \leftarrow \underline{a} \times x - y \geq -\overline{b}$
10 **case** *NW*
11 $p \leftarrow \underline{a} \times x - y \geq -\overline{b}$
12 **case** *SW*
13 $p \leftarrow \underline{a} \times x - y \leq -\underline{b}$
14 **if** the previous points satisfy p **then** $P \leftarrow P \cup \{p\}$
15 return $P' \subseteq P$

The line 14 adds a condition to guarantee that the previous points satisfy the constraint. Theoretically, the round-off of $\underline{a}$ may produce this pathological case when the 3 points are quasi-aligned. This case never occurred in our tests.

D. The Lazy-Graham-TEC Auto-adaptive Algorithm

Lazy-Graham-TEC (`LGT`) is an advanced version of `TEC` switching between `TEC-UB` and `Graham-TEC` in an auto-adaptive way during the optimization process. The main parameter of `LGT` is a number p of nodes in the search tree.

`LGT` starts by using `TEC-UB` in order to better improve the upper bounding. When `LGT` does not find a new upper bound after a given number p of nodes, `LGT` is betting that the best feasible point in a local minimum (or an optimal solution) has been found. Therefore, the stronger filtering operator `Graham-TEC` is called to more quickly reach another region of the tree search, or prove optimality of the solution by improving the lower bound.

When a new upper bound is found, `LGT` switches back to `TEC-UB` for better intensifying the search. `LGT` performs a new phase of at least p nodes with `TEC-UB`, before switching again to `Graham-TEC`, and so on.

V. Experiments

We have run experiments on constrained global optimization instances. Section V-A reports the implementation and the protocol used. Section V-B compares the different versions of `TEC` and their settings. Section V-C presents the results obtained by `LGT`, compared to `HC4` and `3BCID`.

A. Experimental Protocol and Implementation

All the versions of `TEC` have been implemented in the Interval-Based EXplorer `Ibex` [7]. This free C++ library has facilitated the implementation of our filtering operators by providing us a direct access to the interval $Branch$ & $Bound$ `IbexOpt` [5] presented in Section II, the `X-Newton`, `In-X-Newton` algorithms and filtering operators as `HC4` and `3BCID`. All the experiments were run on the same computer (`Intel X64, 2.6GHz, 32Go RAM`).

All the experiments were performed on constrained global optimization instances solved by `IbexOpt`. All the parameters of `IbexOpt` have been fixed to a given set of adequate values common to all the tested instances. Thus, $SmearSumRel$ was used as variable selection heuristic, `X-Newton` as the algorithm used by the procedure `ConvexContract`, and the algorithms `In-HC4` [18] and `In-X-Newton` as `FeasibleSearch` heuristic searching for a good feasible point (upper bounding). The precision settings were fixed to ϵ_{obj} = 1.e-8 and $\epsilon_{sol} = \frac{\epsilon_{obj}}{100}$. A timeout of 1800 seconds was chosen. The parameters used to construct the TEC tree were assigned with the values specified in Section III-B, for all versions of `TEC`.

Tests have been performed on a benchmark of 82 satisfiable instances of constrained optimization problems belonging to the Coconut benchmark [8]. These instances correspond to all the instances from the series 1 and 2 of the Coconut benchmark that have a reasonable size[3] (between 6 and 50 variables) and are significant (solved in more than 1 second by at least one approach).

Note that the equations $h(X) = 0$ in these instances have been relaxed by inequalities -1.e-8 $\leq h(X) \leq +1$.e-8 because `IbexOpt` needs to achieve upper bounding in non empty feasible regions [18] (see Section IV-A). Note that other interval $Branch$ & $Bound$ solvers like `Icos` [3] and `GlobSol` [2] can provide a guaranteed optimum also on problems having non relaxed equations. A comparison between `Ibex` and these solvers can be found in [5].

[3]Instances with more than 50 variables are generally not solved in reasonable time by Interval $Branch$ & $Bound$ algorithms without a tuning of the algorithm specific to each instance.

B. Comparing the Different Versions of TEC

Table I shows a synthetic comparison between TEC and the advanced versions of TEC, i.e. TEC-UB, Graham-TEC and LGT. We report the time and number of nodes gain ratios, comparing each strategy s with the reference strategy TEC. We take into account the CPU time and the number of nodes obtained by a strategy s on an instance i. These experimental results are averaged over 10 trials obtained with different seeds, due to the random choices made by IbexOpt. The following formulas are used to display the time comparison:

- Total Gain(s):

$$\frac{\sum_i Time_{TEC}(i) - \sum_i Time_s(i)}{\sum_i Time_{TEC}(i)} \tag{4}$$

- Max Loss(s):

$$\max_i \left(\frac{Time_s(i) - Time_{TEC}(i)}{Time_s(i)} \right) \tag{5}$$

- Max Gain(s):

$$\max_i \left(\frac{Time_{TEC}(i) - Time_s(i)}{Time_{TEC}(i)} \right) \tag{6}$$

Same formulas are used for comparing the number of nodes.

Table I
COMPARISON OF TEC-UB, GRAHAM-TEC AND LGT VERSUS TEC.

	Times			Nodes		
	Total Gain	Max Loss	Max Gain	Total Gain	Max Loss	Max Gain
TEC-UB	26.4 %	40.7 %	71.8 %	28.1 %	53.1 %	80.7 %
Graham-TEC	15.7 %	64.0 %	41.4 %	39.3 %	26.1 %	69.6 %
LGT	35.4 %	27.3 %	88.3 %	41.2 %	51.5 %	91.3 %

Graham-TEC shows the lowest gain on the total time and the second best gain on the total number of nodes. This confirms that the convex hull computed by Graham-TEC produces a significant gain of contraction compared to the box hull achieved by TEC, but at a high cost. The improvement of the upper bounding brought by TEC-UB also produces a gain on the total number of nodes, at lower computational time than Graham-TEC. Finally, LGT exploits these two approaches and obtains the best total gain both in time and number of nodes. These results seem to confirm the relevance of the adaptation policy proposed and show a complementarity between the improvement of the upper bounding (TEC-UB) and the improvement of the contraction and the lower bounding (Graham-TEC) leading to a drop in the number of nodes.

To reach a good value of the parameter p used by LGT (number of nodes visited with no improvement of the upper bound), we tested several values. Below $p = 500$, the results did not show good performances and above $p = 1500$ too. A robust tested value was 1000 with the γ parameter set to 0.05. Note that first attempts to tune p in an auto-adaptive way during the optimization process were not successful.

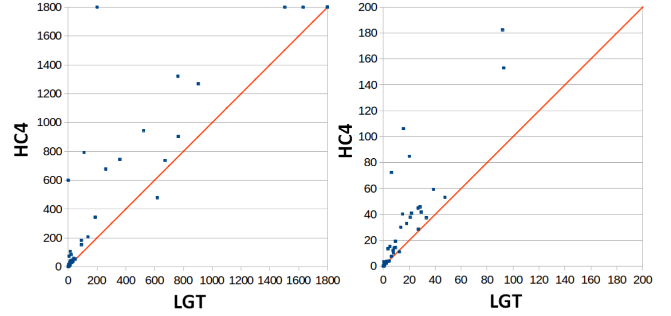

Figure 4. Pairwise comparison between LGT and HC4. The coordinates of each point represents the CPU time (in second) required by both competitors to solve one instance.

C. Results Obtained by LGT

Compared to the HC4 constraint propagation algorithm, LGT obtains better results (see Fig. 4). Three additional instances (haldmads, ex6_2_10 and disc2) are solved by LGT. When we extend the timeout to 7200 s, only ex6_2_10 is solved in 2530 s by HC4.

Compared to the 3BCID state-of-the-art domain shaving algorithm, LGT also obtains good results as shown in Fig. 5. Two instances are solved by LGT whereas 3BCID solves them with an extended timeout (disc2 in 3885 s and ex_6_2_10 in 3463 s). However, one instance (ex8_4_5) is solved by 3BCID but is not solved by LGT, even when the timeout is extended to 7200 s. Among all the filtering operators and all versions of TEC tested, only 3BCID reaches the optimal solution of this instance in the extended timeout 7200 s.

VI. RELATED WORK

Polyhedral convexification of the solution set consists in computing an outer approximation of the solution space of a system by convexifying each inequality constraint individually. A polyhedral enclosure of the solution set leads to a reduction of the search space and/or an improvement of the lower bound by using a call to a linear programming solver.

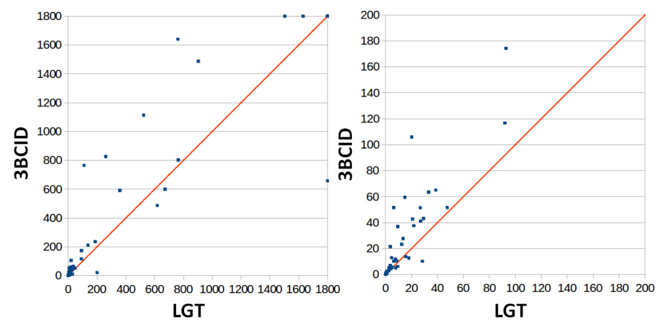

Figure 5. Pairwise comparison between LGT and 3BCID (parameter s=10 splits). The coordinates of each point represents the CPU time (in second) required by competitors to solve an instance.

The idea behind Graham-TEC is significantly different since the linear constraints learned depend on the filtering operation. They do not correspond to the initial constraints.

In Pelleau et al. [19], the notion of box encountered in continuous constraint programming is called into question. Each pair (i, j) of variables is associated with an octagon. This octagon can be viewed as the intersection of two boxes: the initial box and a box rotated by 45 degrees. For each pair (i, j), 4 constraints $\pm x_i \pm x_j \leq c_k$ are added. To translate the constraints in the rotated box, the variables x_i and x_j are replaced by their expressions in the new base. Then, domains are contracted with these additional constraints. The principles are significantly different from LGT and TEC, but a common point resides in the additional binary constraints that are not defined by the user.

An arborescent filtering operator, called Box-k, was proposed in 2009 [20]. Several well-constrained $n \times n$ subsystems of equations were initially selected in the system. At each node of the search tree, a contraction, based on an arborescent exploration, is performed on each subsystem. Unlike the TEC tree, an interval Newton is also called at each node of the tree. Under some conditions, Newton can use the property that the subsystem is well-constrained to converge more quickly towards the solutions (Newton behaves like a global constraint on the subsystem). The main difference between TEC and Box-k is that TEC is called at each node visited on all the system whereas the subsystem handled by Box-k is structured. Box-k provides good results only on very sparse systems.

VII. Conclusion

We have proposed a new filtering operator TEC and several ways to exploit the TEC tree in global optimization: TEC-UB and Graham-TEC which improve the upper and the lower bounding, respectively. The adaptive version LGT can switch between TEC-UB and Graham-TEC during search. The experimental results underline that LGT is better than each of them. Finally, results show that LGT brings a significant speed-up on several state-of-the-art instances compared to the filtering operators HC4 and 3BCID.

LGT is implemented in IbexOpt which is rigorous only for inequalities. It would be interesting to adapt the current algorithm for making it work in rigorous global optimization codes (also for equations) like Icos [3] and GlobSol [2].

References

[1] P. Van Hentenryck, L. Michel, and Y. Deville, *Numerica : A Modeling Language for Global Optimization*. MIT Press, 1997.

[2] R. B. Kearfott, *Rigorous Global Search: Continuous Problems*. Kluwer Academic Publishers, 1996.

[3] Y. Lebbah, C. Michel, and M. Rueher, "An Efficient and Safe Framework for Solving Optimization Problems," *J. Computing and Applied Mathematics*, vol. 199, pp. 372–377, 2007.

[4] J. Ninin, F. Messine, and P. Hansen, "A Reliable Affine Relaxation Method for Global Optimization," *4OR*, vol. 13, no. 3, pp. 247–277, 2015.

[5] G. Trombettoni, I. Araya, B. Neveu, and G. Chabert, "Inner Regions and Interval Linearizations for Global Optimization," in *AAAI*, 2011, pp. 99–104.

[6] G. Trombettoni and G. Chabert, "Constructive Interval Disjunction," in *Proc. CP, LNCS 4741*, 2007, pp. 635–650.

[7] G. Chabert and L. Jaulin, "Contractor Programming," *Artificial Intelligence*, vol. 173, pp. 1079–1100, 2009.

[8] O. Shcherbina, A. Neumaier, D. Sam-Haroud, X. Vu, and T. Nguyen, *Benchmarking Global Optimization and Constraint Satisfaction Codes*. Springer, 2002.

[9] R. E. Moore, *Interval Analysis*. Prentice-Hall, 1966.

[10] F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget, "Revising Hull and Box Consistency," in *Proc. ICLP*, 1999, pp. 230–244.

[11] I. Araya, G. Trombettoni, and B. Neveu, "A Contractor Based on Convex Interval Taylor," in *CPAIOR 2012*, ser. LNCS, no. 7298, May 2012, pp. 1–16.

[12] O. Lhomme, "Consistency Techniques for Numeric CSPs," in *IJCAI*, 1993, pp. 232–238.

[13] R. Debruyne and C. Bessiere, "Some Practicable Filtering Techniques for the Constraint Satisfaction Problem," in *Proc. IJCAI*, 1997, pp. 412–417.

[14] R. Kearfott and M. Novoa III, "INTBIS, a portable interval Newton/Bisection package," *ACM Trans. on Mathematical Software*, vol. 16, no. 2, pp. 152–157, 1990.

[15] Y. Lin and M. Stadtherr, "LP Strategy for the Interval-Newton Method in Deterministic Global Optimization," *Industrial & engineering chemistry research*, vol. 43, pp. 3741–3749, 2004.

[16] Y. Lebbah, C. Michel, M. Rueher, D. Daney, and J. Merlet, "Efficient and safe global constraints for handling numerical constraint systems," *SIAM Journal on Numerical Analysis*, vol. 42, no. 5, pp. 2076–2097, 2005.

[17] R. Graham, "An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set," *Inf. Process. Lett.*, vol. 1, no. 4, pp. 132–133, 1972.

[18] I. Araya, G. Trombettoni, B. Neveu, and G. Chabert, "Upper Bounding in Inner Regions for Global Optimization under Inequality Constraints," *Journal of Global Optimization*, vol. 60, no. 2, pp. 145–164, 2014.

[19] M. Pelleau, C. Truchet, and F. Benhamou, "Octagonal Domains for Continuous Constraints," in *Proc. CP, Constraint Programming, LNCS 6876*. Springer, 2011, pp. 706–720.

[20] I. Araya, G. Trombettoni, and B. Neveu, "Filtering Numerical CSPs Using Well-Constrained Subsystems," in *Proc. CP'09, LNCS 5732*, 2009.