

RESEARCH ARTICLE

*ICOS: a branch and bound based solver for rigorous global optimization*Lebbah Yahia^{ab*}^a *Laboratoire I3S, UNSA-CNRS - Projet Cep, Polytech'Nice-Sophia - Département Sciences Informatiques, 930, Route des Colles - BP 145, 06903 Sophia Antipolis Cedex;*^b *Département Informatique, Université d'Oran Es-Senia, B.P. 1524 EL-M'Naouar, 31000 Oran, Algeria*

(April 2008)

This paper describes a software package, called ICOS, which implements a branch&bound algorithm for solving rigorously global optimization problems. ICOS library contains algorithms coming from constraint programming, interval analysis, and linear relaxation techniques. It contains an interface to linear programming solvers and local optimization solvers (e.g. Coin/Clp, Cplex, IpOpt). ICOS has also its own AMPL parser, which enables to call ICOS binary code in a unix-like command.

The ICOS strategy language enables combining and parametrising existing algorithms for solving optimization problems. So, the user can develop its own solving strategies without changing anything in the ICOS internal architecture. Various examples are given to show how the strategy language can be used. We give an overview of ICOS design, implementation, and a quick user's guide.

Keywords: global optimization; interval analysis; constraint programming; problem solving; abstract syntax tree;

AMS Subject Classification: 90C26, 65G40, 68T20, 68N01

1 Introduction

ICOS stands for Interval COnstraint Solver. It is a system for finding a rigorous global minimum of constrained optimization problems in the following form:

$$\begin{aligned} & \text{minimize } f(X) \\ & \text{subject to } g_i(X) = 0, \quad i = 1..k \\ & \quad \quad \quad g_j(X) \leq 0, \quad j = k + 1..m \end{aligned} \quad (1)$$

with $X \in \mathbf{X}$, $f : \mathbb{R}^n \rightarrow \mathbb{R}$ and $g_j : \mathbb{R}^n \rightarrow \mathbb{R}$; Functions f and g_j are continuously differentiable on some vector \mathbf{X} of intervals of \mathbb{R} . Nonlinear systems of equations and inequalities may be modeled with (1) by fixing the objective function to a constant value.

Methods for solving the optimization problem (1) may be classified as follows:

- Local methods which finds a local ¹ minimum.

*Corresponding author. Email: lebbah@essi.fr / ylebbah@gmail.com

¹A feasible point x' is a local minimum if for a small ϵ , any feasible point x having $|x_i - x'_i| < \epsilon$, $f(x) \geq f(x')$. We recall that the feasible region of (1) is the set of points that satisfy the m constraints of (1).

ISSN: 1055-6788 print/ISSN 1029-4937 online

© 2008 Taylor & Francis

DOI: 10.1080/03081080xxxxxxx

<http://www.informaworld.com>

- Global methods find a global ² minimum, assuming exact computations. In practice, the solving process is stopped when reaching an approximate solution respecting the user's tolerance.

- Rigorous methods take into account rounding-off errors when finding a minimum. The result of a rigorous algorithm, when run on a computer, is reached respecting the user's tolerance, with certainty. There is an except for near-degenerate cases where the rigorous method terminates but notifies that the tolerances are exceeded (see details in [21]). A non-rigorous method provides a solution which does not necessarily respect the user's tolerance due to the accumulation of rounding-off errors.

There are mathematical and algorithmic difficulties to find the global minimum. Finding a global minimum of (1) is much more difficult than finding a local minimum, because the whole feasible region should be explored with its discontinuities and non-convexities. For a more detailed discussion on these difficulties, see the Neumaier's survey paper [21]. On top of it, taking into account rounding-off errors adds the challenging difficulty of the rigorous computation. Moreover, optimization softwares require usually an expert user interaction.

ICOS system provides an easy to use solver that aims at responding to the above issues posed by solving globally and rigorously optimization problems. To cope with globality, ICOS implements a branch&bound algorithm that combines many techniques coming from

- continuous optimization (e.g. linear relaxation techniques [2, 2, 18, 18, 25, 26, 26], local optimization methods),
- constraint programming (e.g. *2B* [16], *Box* [4, 28], *kB(w)* reduction techniques [5, 13, 16], and **quad**-filtering [15]),
- and interval analysis techniques [9, 19, 20] which make the solver fully rigorous.

ICOS started as an experimentation environment developed during a PhD thesis [12]. Its development continues with a close collaboration with CeP team ¹ at university of Nice - Sophia antipolis. The current version of ICOS is 16000 lines of code written in C++, and 1100 lines in Yacc/Lex. It runs under Linux system. To use ICOS, you just have to invoke the ICOS executable on the optimization problem formulated with the AMPL scalar language [6].

ICOS has been used in the Coconut benchmarking [1] for global optimization solvers². It is also available under Neos server ³ for solving nonlinear systems of equations.

The paper is organised as follows. Section 2 gives a quick overview of using ICOS on an illustrating example. We introduce in section 3 our branch&bound generic algorithm. In section 4, we explore the ICOS strategy language and explain its operators illustrated with various examples. Section 5 gives an overview on the design architecture of ICOS implementation. A quick user's guide is given in section 6. Section 7 concludes this document.

²Any point \tilde{x} in the feasible region for which $f(x) \geq f(\tilde{x})$ for all point x in the feasible region is a global minimum.

¹<http://www.polytech.unice.fr/~rueher/CEP/en/>

²<http://www.mat.univie.ac.at/~neum/glopt/coconut/>

³<http://www-neos.mcs.anl.gov/neos/solvers/go:icos/AMPL.html>

```

optimize(
  tolerance : 1.0e-3,
  reduction : lfp(1e-8, nat),
  lower_bounding : quadopt(quad_mid, rlt_xn),
  upper_bounding : [unewton([borsuk3], [lb])],
  branching : [large]
);

```

Figure 1. The default strategy

2 Using ICOS: an illustrating example

Consider the following optimization problem [14]:

$$\begin{aligned}
 & \text{minimize} && x \\
 & \text{subject to} && y - x^2 \geq 0 \\
 & && y - x^2(x - 2) + 10^{-5} \leq 0 \\
 & && x, y \in [-10, +10]
 \end{aligned} \tag{2}$$

To solve this problem with ICOS, it has to be expressed in the AMPL language:

```

var x1 >= -10, <= 10;
var x2 >= -10, <= 10;
minimize obj: x1;
subject to
  eq1: x2 - (x1^2) >= 0;
  eq2: x2 - (x1^2)*(x1 - 2) + 1.0e-5 <= 0;
solve;

```

This model can be solved by typing

```
icos-clp demo.mod C=opt.cfg
```

where `opt.cfg` is the default strategy script file which defines the algorithms and the parameters that ICOS should use to solve the problem. `opt.cfg` contains the text given in Figure 1.

This strategy uses the optimization branch&bound algorithm [14], where the global optimum has to be found with relative tolerance 10^{-3} fixed in `tolerance` section. In `reduction` section, the reduction step `lfp(1e-8, nat)` is done with a $2B(10^{-8})$ [16] on the natural interval extension `nat` of the constraints. In `lower_bounding` section, the lower bounding `quadopt(quad_mid rlt_xn)` is done with a linear relaxation on the quadrification of the polynomial terms with the middle heuristic `quad_mid` [15, 24], and a special relaxation `rlt_xn` for the power term [15, 27]. In the branching step, we use an adaptation of the maximum violation heuristic [25] denoted by `large`. The upper-bounding is done with an internal local search [8] `unewton`. The found upper-bound is proved with the third version of the Borsuk algorithm [7], fixed with `borsuk3` keyword.

At each step of the branch&bound algorithm, ICOS displays a line informing the user about the current bounds of the objective function. The first display is:

```
BB-Obj in [2.826837737928890e+00, inf], Rel(Ub-Lb)=inf
```

which means that the current lower-bound is $2.826837737928890e + 00$, and the current upper-bound is ∞ . The relative width of the objective function values is also ∞ .

In the second step of the branch&bound, ICOS outputs:

```

Solution 1
modelstatus = 1.00
x1 = [2.82683773792888981546e+00, 3.16227766016837952279e+00];
x2 = [8.23734411919200226748e+00, 1.00000000000000017764e+01];
OBJ2 = [2.82683773792888981546e+00, 3.16227766016837952279e+00];
BB-Obj in [2.981224143061410e+00, 3.162277660168380e+00],
Rel(Ub-Lb)=5.894147113679867e-02

```

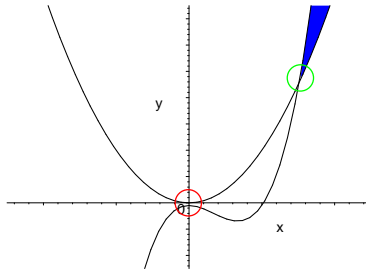


Figure 2. Geometrical representation of problem 2

where ICOS displays the first feasible box¹ followed with an update of the current objective function values whose relative width is now 10^{-2} .

The third and the fourth steps display respectively

```
Solution 2
modelstatus = 1.00
x1 = [2.99455769904863444708e+00, 3.01070617418392671993e+00];
x2 = [8.99550579828946261784e+00, 9.01499537489565483384e+00];
OBJ2 = [2.99455769904863444708e+00, 3.01070617418392671993e+00];
BB-Obj in [2.999455425825419e+00, 3.010706174183927e+00],
Rel(Ub-Lb)=3.743908768938957e-03
```

and

```
Solution 3
modelstatus = 1.00
x1 = [2.99990627425213451929e+00, 3.00010627425213494135e+00];
x2 = [8.99994344564743009585e+00, 9.00014344564742962973e+00];
OBJ2 = [2.99990627425213451929e+00, 3.00010627425213494135e+00];
BB-Obj in [2.999997236412806e+00, 3.000106274252135e+00],
Rel(Ub-Lb)=3.634531942153525e-05
```

In this fourth step, ICOS finds a close feasible box to the global minimum and gives a rigorous enclosure $[2.999997236412806e + 00, 3.000106274252135e + 00]$ of the objective function values at the global minimum.

As shown in figure 5, the solution of problem 2 lies in the neighbourhood of point $x \approx 3$, $y \approx 9$, which confirms the result given by ICOS.

The found feasible box contains the point $(3, 9)$ which is the unique intersection of curve $y = x^2$ and curve $y = x^2(x-2) - 10^{-5}$. However, at point $x = 0$, $y = 0$, the two curves are only separated by a small distance of 10^{-5} . Many local solvers and non rigorous global solvers find $(0, 0)$ as the global minimum even if the precision is enforced up to high. Such a flaw is particularly annoying: as pointed out by Neumaier in [21], there are many situations, like safety verification problems or chemistry, where the knowledge of the real global optima is critical.

3 Branch and bound generic algorithm

Algorithm 1 depicts the branch&bound algorithm [14] that controls the solving process. The branch&bound algorithm we use here combines interval analysis and constraint programming techniques within the well known branch&bound schema described by Horst and Tuy in [10]. Interval analysis techniques enable to introduce safeguards that ensure rigorous and safe computations whereas constraint programming techniques improve the reduction of the feasible space.

¹In interval analysis terminology, a feasible or safe box is a vector of intervals where there is a mathematical proof that the box contains a unique solution.

```

Function BB(IN  $\mathbf{x}$ ,  $\epsilon$ ; OUT  $\mathcal{S}$ ,  $[L, U]$ )
%  $\mathcal{S}$ : set of proved feasible points
%  $\mathbf{f}_{\mathbf{x}}$  denotes the set of possible values for  $f$  in  $\mathbf{x}$ 
 $\mathcal{L} \leftarrow \{\mathbf{x}\}$ ;  $\mathcal{S} \leftarrow \emptyset$ ;  $(L, U) \leftarrow (-\infty, +\infty)$ ;
while  $w([L, U]) > \epsilon$  do
     $\mathbf{x}' \leftarrow \mathbf{x}''$  such that  $\mathbf{f}_{\mathbf{x}''} = \min\{\mathbf{f}_{\mathbf{x}''} : \mathbf{x}'' \in \mathcal{L}\}$ ;  $\mathcal{L} \leftarrow \mathcal{L} \setminus \mathbf{x}'$ ;
     $\bar{\mathbf{f}}_{\mathbf{x}'} \leftarrow \min(\bar{\mathbf{f}}_{\mathbf{x}'}, U)$ ;
STEP 1:  $\mathbf{x}' \leftarrow \text{Reduce}(\mathbf{x}')$ ;
STEP 2:  $\underline{\mathbf{f}}_{\mathbf{x}'} \leftarrow \text{LowerBound}(\mathbf{x}')$ ;
STEP 3:  $(\bar{\mathbf{f}}_{\mathbf{x}'}, \mathbf{x}_p, \text{Proved}) \leftarrow \text{UpperBound}(\mathbf{x}')$ ;
    if  $\text{Proved}$  then  $\mathcal{S} \leftarrow \mathcal{S} \cup \{\mathbf{x}_p\}$ ; endif
    if  $\mathbf{x}' \neq \emptyset$  then
STEP 4:  $(\mathbf{x}'_1, \mathbf{x}'_2) \leftarrow \text{Split}(\mathbf{x}')$ ;  $\mathcal{L} \leftarrow \mathcal{L} \cup \{\mathbf{x}'_1, \mathbf{x}'_2\}$ ;
    endif
    if  $\mathcal{L} = \emptyset$  then
         $(L, U) \leftarrow (+\infty, -\infty)$ ;
    else
         $(L, U) \leftarrow (\min\{\underline{\mathbf{f}}_{\mathbf{x}''} : \mathbf{x}'' \in \mathcal{L}\}, \min\{\bar{\mathbf{f}}_{\mathbf{x}''} : \mathbf{x}'' \in \mathcal{S}\})$ ;
    endif
endif
endwhile

```

The solver (see algorithm 1) computes enclosures for minimizers and safe bounds of the global minimum value within an initial box \mathbf{x} . The algorithm maintains two lists: a list \mathcal{L} of boxes to be processed and a list \mathcal{S} of proved feasible boxes. It provides a rigorous enclosure $[L, U]$ of the global optimum with respect to a given tolerance ϵ .

The algorithm selects the box with the lowest lower bound of the objective function. The *Reduce* function applies reduction techniques to reduce the size of the box \mathbf{x}' . Then, *LowerBound*(\mathbf{x}') computes a rigorous lower bound of the objective within the box \mathbf{x}' using a linear programming relaxation of the initial problem. *UpperBound*(\mathbf{x}) computes a feasible box. If *UpperBound* succeeds to prove feasibility then the box \mathbf{x}_p that contains this proved feasible point is added to the list \mathcal{S} . At this stage, if the box \mathbf{x}' is empty then, either it does not contain any feasible point or its lower bound $\underline{\mathbf{f}}_{\mathbf{x}'}$ is greater than the current upper bound U . In both cases, we say that the box is fathomed. If \mathbf{x}' is not empty, the box is split along one of the problem variables. At each box selection and processing, the algorithm maintains the lowest lower bound L of the remaining boxes \mathcal{L} and the lowest upper bound U of proved feasible boxes. The algorithm terminates when the space between U and L becomes smaller than the given tolerance ϵ . Of course a proved optimum cannot always be found, and thus, algorithm 1 has to be stopped in some cases to get the feasible boxes which may have been found.

Four parameters are needed in algorithm 1:

- (1) STEP 1 (*Reduce*): There are many reduction techniques that could be used in this step to reduce the current box.
- (2) STEP 2 (*LowerBound*): The lower bounding is done with linear relaxation techniques. But there is many ways to relax a nonlinear term. Here also the ICOS strategy proposes a language to choose the techniques above those available.
- (3) STEP 3 (*UpperBound*): The upper bounding could be done by calling a local search [8] available in ICOS library. ICOS contains an interface to AMPL solvers enabling to get an approximate upper bound. At the final stage of the upper bounding, an interval analysis proof procedure, mainly Borsuk proof [7], is used to prove the feasibility of the obtained upper bound.

(4) **STEP 4 (*Split*)**: When the current box has not been proved to be empty and not reduced enough, it is splitted along one variable or several variables. All these parameter can be fixed within the strategy language part dedicated to the splitting strategy.

All these four algorithmic parameters can be configured within the strategy language which is explained in the following section 4.

4 ICOS strategy language

Complex algorithms use many parameters and algorithms. For instance, in branch&bound algorithm, we pointed out four algorithmic parameters. Fixing such parameters inside the implementation, will make its maintenance complex. Specially in an experimental environment, the parameters are usually changed. Moreover, each class of problems requires its own parameters that should be memorized. It is puzzling to memorise these different parameters inside the implementation.

For this reason, we have adopted in ICOS a special design architecture where the parameters of the algorithm are fixed outside within what is commonly called *strategy language*. Thus the parameters are stored within a script that makes easy parameters changing.

The solving process of optimization problems can be viewed as a hierarchical computation where many basic algorithms are used (e.g., solving linear systems, monovariate equations, linear programs, quadratic programs, etc.). Actually, every optimization solver proposes its own branch&bound process. And existing solvers use lower-bounding, upper-bounding, reduction, node selection, and so on. They differ in the algorithms used and the way they use them.

We want to capture the way these basic algorithms are used and exploited, and not the internal implementation of these basic algorithms. Because we think that the performance of basic algorithms are very close between current solvers. What makes the difference between the solvers is the way they are using basic algorithms.

Simply, in a solver architecture, there are generic algorithms, and basic algorithms. Generic algorithms need to call other algorithms, whereas basic algorithms do not call any other algorithm. The branch&bound is a generic algorithm since that it needs upper-bounding, lower-bounding, reduction, and branching algorithms. These four algorithms are also generic algorithms. Upper-bounding needs two algorithms: local optimization solving, and proving the feasibility by inflation. In proving feasibility, there are many basic algorithms that could be used, such as Miranda, Borsuk, or Krawczyck [7]. There are also, a plethora of local solving algorithms. Lower-bounding is also generic, since that it can use a combination of basic relaxation algorithms. The reduction step can be done with algorithms coming from constraint programming. Some of these algorithms are generic such as strong consistency algorithms [13, 16].

The ICOS strategy language enables to combine in several ways existing algorithms. The ICOS strategy script is processed with Lex/Yacc parsers. The text is transformed into an Abstract Syntax Tree ¹ (AST). Thus, the hierarchical structure of a generic algorithm is captured within an AST.

An AST represents the structure of any strategy. Internal nodes are generic algorithms or also operators and the leaf nodes represent the basic algorithms.

¹ASTs are well established in compiler construction where they provide an efficient data-structure to handle the program however complex is it. An abstract syntax tree (AST) is a finite, labelled, directed tree, where each interior node represents a programming language construct and the children of that node represent computation components of the construct.

The solving process of the default strategy given in Figure 1 can be represented with the AST given in Figure 3.

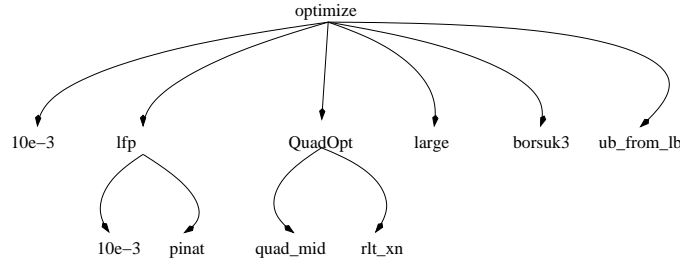


Figure 3. AST of the default strategy

The grammar of the strategy language in a BNF like syntax ¹ is given in Figure 4.

Strategy defines optimization and systems solving operators. **Reduction** defines the main reduction operators. Some reduction algorithms (e.g. `slice`, `fp`), are recursive, since that they call other reduction strategies. Various options can be defined. In the following sections, we describe informally the semantics of these syntactic constructs.

4.1 nat and box basic operators

`nat` (resp. `box`) implements a single step of $2b(w)$ -algorithm [13, 16] (resp. $box(w)$ -algorithm [4]). It reduces the domain of each variable.

Let be the following constraint system:

¹Words with only lower-case letters are terminals. Words beginning with one single capital letter are non-terminals. Repeated terms are compacted with `Term{,Term}` where `Term` is the repeated term, and “,” is the separating symbol. For instance, `LocSolver{,LocSolver}` denotes a sequence of `LocSolver`.

```

Strategy      ::=
  optimize(
    tolerance: Num,
    reduction: Reduction{+Reduction},
    lower_bounding: quadopt(QuadOption{,QuadOption}),
    upper_bounding: [LocSolver{,LocSolver}],
    branching: [BranchOption{,BranchOption}]
  ); |
  search(
    tolerance: Num,
    reduction: Reduction{+Reduction},
    branching: [BranchOption{,BranchOption}]
  );
Reduction    ::=
  nat | box(Num) | newton(Num) |
  slice(Num, Reduction{+Reduction}) |
  fp(Num, Reduction{+Reduction}) |
  lfp(Num, Reduction{+Reduction}) |
  quad(QuadOption{,QuadOption})
QuadOption   ::= rlt_xn | quad_mid | quad_left | ...
BranchOption ::= large | pgd | rr
LocSolver    ::= LocName([LocOption{,LocOption}], [LocPoint{,LocPoint}])
LocName      ::= unewton | ipopt | loqo | ...
LocOption    ::= initialization | borsuk3 | period(Num) | ...
LocPoint     ::= mid | lb | rand(Num) | init
ID           ::= "identifier"
Num          ::= "numeric number"
  
```

Figure 4. Grammar of ICOS strategy language

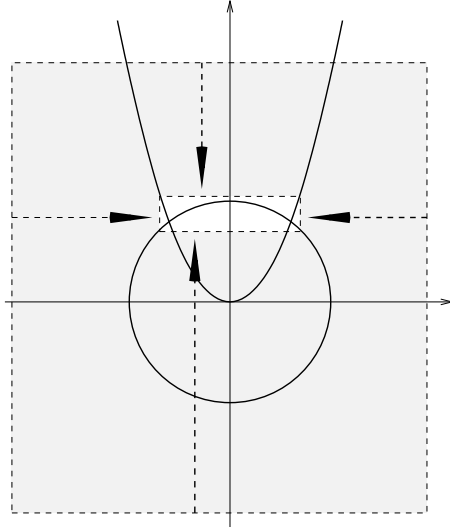


Figure 5. Reduction with `nat` or `box` operators

$$\begin{aligned}
 y - x^2 &= 0 \\
 y^2 + x^2 &= 1 \\
 x, y &\in [-10, +10]
 \end{aligned} \tag{3}$$

`nat` or `box` tries to reduce the initial box $[-10, +10] \times [-10, +10]$ by following the steps:

- (1) `nat` (resp. `box`) takes the constraint $y - x^2 = 0$ and the variable x , and applies the reduction mechanism (the natural extension for `nat`, or the box-algorithm for `box`) in order to tighten the domain of x .
- (2) In the same way, it reduces y with $y - x^2 = 0$.
- (3) It reduces also x with $y^2 + x^2 = 1$.
- (4) And finally y with $y^2 + x^2 = 1$.

At the end, we obtain a tighter domain (see Figure 5).

4.2 newton *basic operator*

`newton(Num)` is the interval newton algorithm with Gauss-Seidel iterations. It can terminate with a feasible box if the existence proof procedure succeeds. For more details see Hansen's book [9].

4.3 slice *operator*

`slice(Num, Reduction{+Reduction})` implements a slicing algorithm that uses the reduction algorithms specified in `Reduction{+Reduction}` on each bound of the variables with tolerance `Num`. The details of the slicing algorithm is given within the strong consistency definition $kB(w)$ -consistency available in [13].

Let be the following constraint system

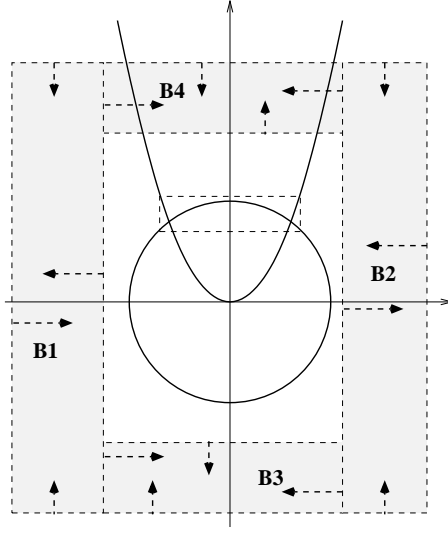


Figure 6. Reduction with `slice` operator

$$\begin{aligned}
 y - x^2 &= 0 \\
 y^2 + x^2 &= 1 \\
 x, y &\in [-10, +10]
 \end{aligned} \tag{4}$$

and a reduction strategy specified as follows

```
slice(1.0e-3, lfp(1.0e-8, nat+box))
```

Let be $D = [-10, +10] \times [-10, +10]$ the current box. As depicted in figure 6, we have $B1 = [-9, -10] \times [-10, +10]$, $B2 = [9, 10] \times [-10, +10]$, $B3 = [-10, +10] \times [-9, -10]$, and $B4 = [-10, +10] \times [9, 10]$. The `slice` operator will proceed as follows (see figure 6):

- (1) It takes the variable x and fixes locally its lower-bound with tolerance 10^{-3} , and get the box $B1$. It provides the domain $B1$ to the reduction algorithm `lfp(1.0e-8, nat+box)`. If it terminates with an empty domain, then $B1$ is removed from the current box D . This reduces the lower-bound of x .
- (2) It proceeds with the same manner on the upper-bound of x with $B2$, lower-bound of y with $B3$, and finally the upper-bound of y with $B4$.

With other words, the slicing operator loops on the bounds of the variables and tries to reduce each bound separately by calling another reduction algorithm. But it loops only once on each bound. If we want to loop on the slicing operator, we have to use the fixed-point operator introduced below.

4.4 fp operator

`fp(Num, Reduction{+Reduction})` is a fixed point algorithm which loops on the algorithms specified in `Reduction{+Reduction}` until reaching a domain where the variables are tightened with less than `Num`.

The following operator

```
fp(1.0e-8, slice(1.0e-3, lfp(1.0e-8, nat+box)))
```

is a reduction algorithm that applies a fixed-point iterations on the slicing re-

duction algorithm

```
slice(1.0e-3, lfp(1.0e-8, nat+box))
```

until reaching a domain where the variables are tightened with less than $1.0e-3$. It is very close to the $3B(1.0e-3)$ -filtering [13, 16].

4.5 lfp operator

`lfp(Num, Reduction{+Reduction})` is a lazy fixed point algorithm which is very similar to `fp`, except that it manages the reduction process with a lazy manner. It is very similar to AC3 [17] known reduction algorithm in constraint programming.

4.6 quad operator

`quad(QuadOption{,QuadOption})` implements the `quad`-filtering algorithm [15]. The `quad` algorithm generates a linear relaxation of the nonlinear constraints. It uses the quadrification process with the middle heuristic [15] (resp. left heuristic) when `quad_mid` (resp. `rlt_xn`) is given in the options. If `rlt_xn` is given in the options, power terms are linearized with a special relaxation technique [15, 27].

With `quad`, ICOS reduces the bounds of the variables by successively minimizing and maximizing each variable. It proceeds by solving $2n$ linear programs, where n is the number of the variables.

Let us take the following constraints

$$\begin{aligned} 2x_1x_2 + x_2 &= 1 \\ x_1x_2 &= 0.2 \\ x_1, x_2 &\in [-10, +10] \end{aligned} \tag{5}$$

The `quad` algorithm generates the linear relaxation of (5) by applying bilinear relaxations [2, 18]. Figure 7 illustrates the relaxation and shows that the set of the feasible points of the relaxation of the constraints (5) is a polyhedron. The four steps of minimizing and maximizing of the two variables x_1 and x_2 reduce the domain to the dotted box depicted in Figure 7.

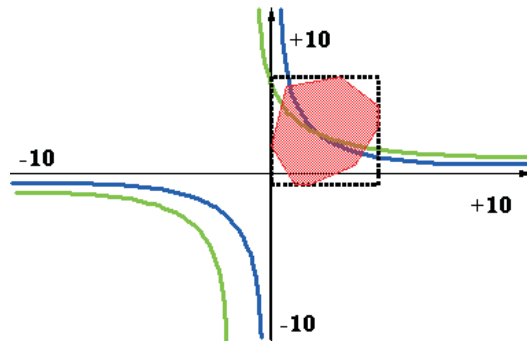


Figure 7. Relaxation of (5) with `quad` algorithm

4.7 quadopt operator

`quadopt` is similar to `quad`, except that it linearises also the objective function. It serves to the lower-bounding step in the branch&bound algorithm.

4.8 Optimize operator

The operator

```
optimize(  
  tolerance: Num,  
  reduction: Reduction{+Reduction},  
  lower_bounding: quadopt(QuadOption{,QuadOption}),  
  upper_bounding: [LocSolver{,LocSolver}],  
  branching: [BranchOption{,BranchOption}]  
);
```

specifies the branch&bound algorithm with its five parameters (see also Figure 4):

- (1) **tolerance**: Num provides the user's tolerance that is the desired width of the objective function values around the global optimum.
- (2) **reduction** part specifies the reduction algorithms. (See previous sections on reduction algorithms.)
- (3) **lower_bounding** fixes the parameters of the **quadopt** operator.
- (4) **upper_bounding** specifies local solvers that should be called in the upper-bounding step of the branch&bound algorithm. Each local solver **LocSolver** is specified by giving (see Figure 4) :
 - **LocName** is the name of the solver. Its particular value **newton** is an internal local search [8]. All the other names are AMPL external solvers names.
 - **LocOption** specifies particular options, such as the proof procedure (e.g. Borsuk, Miranda, etc.). When **initialization** is given, then the solver is called just before the branch&bound algorithm. The option **period(Num)** specifies the number Num of the branch&bound steps required to call the local solver periodically.
 - **LocPoint{,LocPoint}** specifies what are the starting points of the local solver. When **LocPoint** is fixed to **mid** (resp. **lb**, **init**), then the starting point is the middle of the current box (resp. the lower-bound, the starting point given in the AMPL model). When **rand(Num)** is specified, then the local solver is called Num times with a random point in the current box. This last strategy is a kind of a multistart procedure.
- (5) **BranchOption{,BranchOption}** defines branching options. **large** adopts the maximum violation heuristic when choosing the variables to split. This heuristic is very similar to those given in [25]. **pgd** points out that the variable with the greatest interval width has to be chosen in the branching step. And **rr** option adopts a round-robin heuristic in the branching step.

4.9 search operator

The operator

```
search(  
  tolerance: Num,  
  reduction: Reduction{+Reduction},  
  branching: [BranchOption{,BranchOption}]  
);
```

is very similar to the branch&bound operator, except that it does neither need lower-bounding, nor upper-bounding. It implements a branch&reduce algorithm which is similar to the branch&prune algorithm [28].

The default strategy for solving nonlinear equations is

```
search(  
  tolerance: 1.0e-6,  
  reduction: lfp(0.01, nat)+newton(1.0e-8),  
  branching: [pgd]  
);
```

This strategy uses the following algorithms

- (1) `lfp(0.01, nat)` which applies a $2B(0.01)$ -consistency algorithm to reduce the domain with natural interval extension.
- (2) `newton(1.0e-8)` which applies the interval newton algorithm to reduce the domain, and prove the existence of a unique solution inside the considered box.

The `pgd` option indicates that the variable with the largest domain is chosen in the branching step.

4.10 Examples

4.10.1 Moreaux suite

The default solving strategy has been able to solve this constraints satisfaction problem, which is available in Coconut benchmarking ¹.

4.10.2 Reimer suite

With AMPL language, `Reimer(n)`, $n \leq 5$, coming from Verschelde's web site, is defined

```
param n := 5;
set m := 1 .. n;
var x{m} >= -1, <= 1;
subject to
  e1 {k in 2 .. n+1} : -0.5 + sum{i in m} ((prod{j in 1 .. i+1} (-1))*x[i]^k) = 0;
```

The default strategy is not able to find the solutions quickly. We change the strategy and adopt in the reduction step the `quad` and `box` reduction algorithms:

```
search(
  tolerance: 1.0e-8,
  reduction: lfp(1e-1, nat+box(0)+quad(quad_mid, rlt_xn))+newton(1.0e-8),
  options: [pgd]
);
```

For `Reimer(5)`, ICOS finds 24 boxes whose width is less than 10^{-8} . Each of these boxes is proved to contain a unique solution. Such boxes are commonly called safe boxes in interval analysis terminology.

4.10.3 Audet

We have solved `Audet` problems coming from pages 140, 141, 142, 145, 146, and 147 of Audet's thesis [3].

For example, `Audet-146`, with the default strategy, has been solved within 197 seconds. Changing the strategy to the following strategy

```
optimize(
  tolerance: 1.0e-3,
  reduction: quad(quad_mid, rlt_xn) + lfp(1e-8, nat),
  lower_bounding: quadopt(quad_mid rlt_xn),
  upper_bounding: [unewton([borsuk3], [lb])],
  options: [large]
);
```

which adopts the `quad`-algorithm in the reduction step, enables to solve the problem within 36 seconds.

4.10.4 Rump suite

In this example, we tackle a difficult optimization problem `rump(n)` introduced by Rump [22, 23]

¹<http://www.mat.univie.ac.at/~neum/glopt/coconut/Benchmark/Benchmark.html>

$$\begin{aligned}
& \text{minimize} && \|P(x)Q(x)\|^2 \\
& \text{subject to} && \|P(x)\|^2 = \|Q(x)\|^2 = 1 \\
& && p_i, q_i \in [-1, +1], i \in 1..n, \\
& && r_j \in [-1, +1], j \in 2n - 1.
\end{aligned} \tag{6}$$

where $P(x), Q(x)$ are real polynomials (see [22, 23]). A set of upper-bounds for $n \leq 14$, are suggested in [22]. The challenge is to prove these bounds.

`rump(2)` has been solved with the default strategy. When $n > 2$, `icos` was not able to tackle the problem. Experimenting this problem points out that the performance of the lower-bounding is very dependent on the availability of a good upper-bound. Changing the strategy, by adopting the `ipopt/Ampl` solver within the upper-bounding process

```

optimize(
  tolerance: 1.0e-3,
  reduction: lfp(1e-8, nat),
  lower_bounding: quadopt(quad_mid, rlt_xn),
  upper_bounding: [ipopt([initialization, borsuk3],
                        [init, mid, empty, rand(10)]),
                  ipopt([borsuk3, period(100)],
                        [rand(10)]),
                  unewton([borsuk3, period(1)], [1b])
                ],
  branching: [large]
);

```

enables to solve quickly `rump(3)` and `rump(4)` with tolerance 10^{-3} .

5 ICOS design overview

ICOS design architecture derived from extremely simple fact that “*Mathematical solving software applies an algorithm on the problem and get the solution*”. The more we conform to this simple way to view solving algorithms, the more we keep the design simpler. More precisely, it is very important to implement the algorithms separately. If some algorithm A needs another algorithm P , then P should be an input in the parameters of A , and not called directly within A . This helps to keep the algorithms design simple. It makes easy changing the algorithmic parameters. It also enables to exploit it systematically in the strategy language.

In ICOS implementation, we have adopted the AST nomenclature of algorithms given in section 4: basic computations/interfaces, and generic algorithms. Basic interface algorithms manage communication with external solvers, namely `Coin/Clp`, `Cplex`, and `AMPL` (see section 5.1). Generic algorithms implement various algorithms available within the strategy language (see section 5.2).

5.1 Basic computations and interfaces

A basic computation class is a standalone class that implements a basic computation or datatype. Actually, we have two basics:

- (1) *interval arithmetic* `IcInterval`: This class implements all what is needed to compute with intervals. This implementation is mainly based on `Profil/Bias` library [11].
- (2) *CLapack linear algebra library*

Currently, we have two interfaces:

- *Linear programming interface IcMyLP*: This class stores the matrices and vectors of a linear program, and contains the necessary interface methods to call an external linear programming solver. In the current version, we have an interface with Cplex and Coin/Clp.

- *Ampl Solver interface IcINOUT_AMPL*: This class implements the necessary functions to communicate with an Ampl based solver. We just have to fix the name of the solver, and then, this interface can transform any optimization problem into an Ampl text model file, call Ampl solver executable with the intended solver, and finally get the solution.

5.2 Generic algorithms

Internally developed algorithms have as a main class `IcAlgorithm`. First, let us take a brief look at the `IcAlgorithm` interface class:

```
class IcAlgorithm {
public:
    ICOS_OP algo;
    IcAlgorithm** children; int children_size;
    ...
    IcAlgorithm() {};}
    ...
    virtual ICOS_RET solve() {return ICOS_RET_UNCHANGE;}
    ...
};
```

This interface class specifies the minimal properties that we need. `algo` is an identifier of the algorithm. `children` is the algorithmic parameters of the algorithm. `solve` method specifies the solving algorithm; here it does nothing, since it is just an interface class.

The `IcAlgorithm` class is then derived to implement various algorithms:

- `IcRAlgorithmLin` implements algorithms for linear equations.
- `IcRAlgorithmLocal` contains an algorithm doing a local search for continuous optimization.
- `IcIAlgorithm` contains basic reduction and branch&bound algorithm for discrete optimization problems. It is still under development.
- `IcRAlgorithm` is a main class which implements various algorithms for continuous nonlinear optimization.

We will give below more details about `IcRAlgorithm` that implements the branch&bound and other algorithms on real numbers. It is depicted partially below:

```
class IcRAlgorithm: public IcAlgorithm {
    ...
    // interface algorithms
    friend IcRAlgorithm& optimize(...);
    friend IcRAlgorithm& search(...);
    friend IcRAlgorithm& nat(...);
    friend IcRAlgorithm& fp(...);
    ...
    ICOS_RET solve();
    ...
    // algorithms
    ICOS_RET eval_search();
    ICOS_RET eval_optimize();
    ICOS_RET eval_nat();
    ICOS_RET eval_fixpoint();
    ...
};
```

Two categories of methods are specified in `IcRAlgorithm`: interface methods and concrete methods. An interface method fixes the parameters coming from the strategy language, and a concrete method implements the algorithm and uses

the algorithmic parameters available in its children algorithms. For instance, the interface method for branch&bound is given by

```
friend IcRAlgorithm& optimize(...);
```

Parameters of method `optimize` are filled by calling this method within the strategy language parser. The branch&bound is implemented in `eval_optimize` method. For instance, when needing to call lower-bounding, we just invoke the `solve` of the lower-bounding algorithmic parameter.

6 Quick user's guide

6.1 *Installation*

The simplest mechanism for distribution is by downloading from the author's home-pages:

```
http://ylebbah.googlepages.com/icos  
http://www.polytech.unice.fr/~lebbah/
```

After downloading and unzipping, you will find the following files:

- `icos-clp` An executable file, which can be invoked on problems formulated with the AMPL modelling language.
- `libicos-coin.a` An archive file containing the ICOS classes and functions library.
- `makefile` A makefile enabling to generate an executable.
- `cfg` A directory containing various strategy files.
- `cpp` A C++ directory that contains some problems expressed with C++, and solved by calling the library.
- `include` A directory containing C++ header files of ICOS.
- `mod` A directory containing problems modelled with AMPL and solved with ICOS.

6.2 *Using ICOS with AMPL scalar language*

It is easy to use ICOS with AMPL, since that ICOS contains an AMPL scalar parser. Let be `demo.mod` an AMPL scalar model file containing a continuous optimization problem. To run and solve the AMPL model, you just have to issue the command

```
icos-clp demo.mod C=cfg/opt.cfg G="logs" A="/home/user/ampl"
```

where `opt.cfg` is a strategy file that fixes the strategy within the branch&bound ICOS algorithm. Option `G="logs"` fixes the log directories, and `A="/home/user/ampl"` the AMPL solver directory.

During the execution, ICOS generates some log files that contain the found feasible solutions, and the current bounds of the objective function. The following section explains a description of the log files.

6.3 *ICOS output and termination conditions*

In the AMPL interface, given some model file `demo.mod`, once ICOS starts tightening the objective function values and getting feasible solutions, there are three generated files in the `logs` directory:

- `logs/demo/demo.log` A file which contains all the updates on the objective function values, the found feasible solutions and their status, and finally the last objective function bounds respecting the user tolerance. It contains at the end some statistics on the number of solutions found, the number of branchings, the number of simplex calls, ...etc.

- `logs/demo/demo.inf` This file contains the summary information contained in the previous file.

- `logs/demo/demoi.log` Each one of these files contains the i th solution and its status.

Each solution is provided with the following status code `modelstatus`. If it is fixed to 1, then the provided box is feasible or safe; otherwise it is not.

7 Perspectives

In this paper, we have introduced ICOS, a rigorous and global optimization solver. We show that its strategy language enables to fix various algorithmic parameters. It helps the user to parametrize externally the branch&bound algorithm. It also makes easy experimenting algorithms. This language is still under development. Some operators not available, should be added, such as operators for semi-definite relaxations.

Concerning the performance of the ICOS solver, it is well known that global optimization solvers cannot still solve problems with high dimensions. Our various experiments show that global optimization proceeds always in two steps. First, it finds a feasible box near to the global optima within the upper-bounding step. Second, it loops on the lower-bounding step to increase the lower bound toward the initially found upper-bound. We have to work on these bounding steps, and enhance the strategy language to try to cope with problems with high dimensions.

We want also to extend ICOS to solve mixed integer programs. We have to find a way to extend the strategy language with cooperation operators between continuous and discrete solvers.

Acknowledgements

Firt, I want to thank Michel Rueher and Claude Michel for their support and advices. I wish to thank Olivier Lhomme who supported the author during the first development of ICOS system. I wish to thank Oleg Shcherbina for testing ICOS on nonlinear systems within the Coconut project. I thank Mittelman for introducing ICOS in Neos server. I wish also to thank Arnold Neumaier for his many advices and his support in 2006 during the Vienna visit.

References

- [1] W.H. A. Neumaier, O. Shcherbina and T. Vinko, *A comparison of complete global optimization solvers*, Math. Programming B. (2005), pp. 103:335–356.
- [2] F. Al-Khayyal and J. Falk, *Jointly constrained biconvex programming*, Mathematics of Operations Research (1983), pp. 8:2:273–286.
- [3] C. Audet, *Optimisation globale structurée : propriétés, équivalences et résolution*, Ph.D. thesis, École Polytechnique de Montréal (1997).
- [4] F. Benhamou, D. McAllester, and P. Van-Hentenryck, *Clp(intervals) revisited - proceedings of the international symposium on logic programming (1994)*, pp. 124–138.
- [5] H. Collavizza, F. Delobel, and M. Rueher, *Comparing partial consistencies*, Reliable Computing (1999), pp. Vol.5(3),213–228.

- [6] R. Fourer, D. Gay, and B. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, Duxbury Press / Brooks/Cole Publishing Company (2002).
- [7] A. Frommer and B. Lang, *Existence tests for solutions of nonlinear equations using borsuk's theorem*, Tech. Rep. BUW-SC 2004/2, Department of Mathematics, Faculty of Mathematics and Natural Sciences, University of Wuppertal, MathePrisma (2004).
- [8] A. Goldstein et al., *Revisiting the proof process in safe global optimization branch/bound algorithm - submitted* (2008).
- [9] E.R. Hansen, *Global Optimization Using Interval Analysis*, Marcel Dekker, New York (2004).
- [10] R. Horst and H. Tuy, *Global Optimization: Deterministic Approches*, Springer-Verlag (1993).
- [11] O. Knüppel, *PROFIL/BIAS — A fast interval library*, Computing 53 (1994), pp. 277–287, URL PROFIL/BIAS is available at www.ti3.tu-harburg.de/Software/PROFIL/Profil.texinfo_1.html.
- [12] Y. Lebbah, *Contribution à la résolution de contraintes par consistance forte*, Ph.D. thesis, Université de Nantes — 2, rue de la Houssinière, F-44322 Nantes, France (1999).
- [13] Y. Lebbah and O. Lhomme, *Accelerating filtering techniques for numeric CSPs*, Artificial Intelligence 139 (2002), pp. 109–132.
- [14] Y. Lebbah, C. Michel, and M. Rueher, *An efficient and safe framework for solving optimization problems*, Journal of Computational and Applied Mathematics 199 (2007), pp. 372–377.
- [15] Y. Lebbah et al., *Efficient and safe global constraints for handling numerical constraint systems*, SIAM Journal on Numerical Analysis 42(5) (2004), pp. 2076–2097.
- [16] O. Lhomme, *Consistency techniques for numeric CSPs - proceedings of international joint conference on artificial intelligence* (1993), pp. 232–238.
- [17] A. Mackworth, *Consistency in networks of relations*, Journal of Artificial Intelligence (1977), pp. 8(1):99–118.
- [18] G. McCormick, *Computability of global solutions to factorable nonconvex programs – part i – convex underestimating problems*, Mathematical Programming 10 (1976), pp. 147–175.
- [19] R. Moore, *Interval Analysis*, Prentice Hall (1966).
- [20] A. Neumaier, *Interval Methods for Systems of Equations*, *Encyclopedia of Mathematics and its Applications*, vol. 37, Cambridge University Press, Cambridge, UK (1990).
- [21] ———, *Complete search in continuous global optimization and constraint satisfaction*, Acta Numerica (2004).
- [22] S. Rump, *Global optimization: A model problem*, Tech. rep., Hamburg University of Technology (2006), URL <http://www.ti3.tu-harburg.de/rump/Research/ModelProblem.pdf>.
- [23] S. Rump and H. Sekigawa, *The ratio between the toeplitz and the unstructured condition number* (2008), p. to appear, URL <http://www.ti3.tu-harburg.de/paper/rump/RuSe06.pdf>.
- [24] V. Sahinidis and M. Twarmalani, *Baron 5.0 : Global optimisation of mixed-integer nonlinear programs*, Tech. rep., University of Illinois at Urbana-Champaign, Department of Chemical and Biomolecular Engineering (2002).
- [25] ———, *Convexification and Global Optimization in Continuous and Mixed-Integer Nonlinear Programming*, Kluwer Academic Publishers Group (2002).
- [26] H. Sherali and W. Adams, *A Reformulation-Linearization Technique for Solving Discrete and Continuous Nonconvex Problems*, Kluwer Academic Publishing (1999).
- [27] H. Sherali and C. Tuncbilek, *New reformulation linearization/convexification relaxations for univariate and multivariate polynomial programming problems*, Operations Research Letters (1997), pp. 21:1–9.
- [28] P. Van-Hentenryck, D. Mc Allester, and D. Kapur, *Solving polynomial systems using branch and prune approach*, SIAM Journal on Numerical Analysis (1997), pp. 34(2):797–827.