CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Online Research Database In Technology

Non-linear Global Optimization using Interval Arithmetic and Constraint Propagation

Steffen Kjøller, Pavel Kozine, Kaj Madsen
Informatics and Mathematical Modelling, Technical University of Denmark
Lyngby, Denmark. *km@imm.dtu.dk*
Ole Stauning
Saxo Bank, Denmark

December 2005

Abstract

In this Chapter a new branch-and-bound method for global optimization is presented. The method combines the classical interval global optimization method with constraint propagation techniques. The latter is used for including solutions of the necessary condition $f'(x) = 0$. The constraint propagation is implemented as an extension of the automatic differentiation library FADBAD [24], which implements forward and backward differentiation. Thus the user of the integrated programme only has to implement the function expression.
For illustration purposes the performance is illustrated through a couple of numerical examples.

1 Introduction

We consider the problem of finding the global minimum of a function $f : D \to \mathbb{R}$ where $D \subseteq \mathbb{R}^n$ is a compact right parallelepiped parallel to the coordinate axes:

$$\boldsymbol{x}^* = \underset{\boldsymbol{x} \in D}{\operatorname{argmin}} \ f(\boldsymbol{x}) \tag{1}$$

In the following a compact right parallelepiped parallel to the coordinate axes is denoted a *box*.

Methods for solving global optimization problems have been investigated for many years, see for instance [3], [5], [7], [19], [26]. For some classes of problems, e.g. analytically defined functions with a modest number of variables, interval methods have been very successful, [5].
In this chapter we describe a new branch-and-bound type method for solving (1). The method is an extension of the classical interval global optimization method (see for instance [5], [16], [25]), which is often denoted the *Moore-Skelboe algorithm.* This method iteratively investigates sub-boxes of D using monotonicity tests and interval Newton methods for reducing the set guaranteed to contain all solutions. The extension to be described uses constraint propagation (CP) in each iteration to further reduce this set, without loosing solutions. Such a combination has previously been used by several authors, for instance [8], [4], [6], [14]. To the best of our knowledge we are the first, however, to apply CP for finding rigorous bounds for the set of stationary points, i.e., enclosing the solutions to the non-linear set of equations $f'(x) = 0$.
In the classical interval global optimization method such an inclusion is also applied, using some variation of the interval Newton method, [15]. However the two inclusion methods CP and Newton are of quite different natures.
Under non-singularity conditions the classical (iterative) interval method for solving non-linear equations has a quadratic asymptotic convergence rate, however the initial box $X_{(0)}$ for the iteration often has to be quite narrow. If a box X contains more than one solution then $f''(x)$ is singular for some $x \in X$, and then the interval Newton method cannot be applied.
The CP method for enclosing solutions to a set of non-linear equations is normally not so sensitive to narrow starting boxes and to singularities in f''. However its ultimate convergence rate is often slower than the interval Newton method.
Therefore CP may be be used to provide an initial reduction whereas the classical interval method provides the ultimate convergence.
We describe the two methods and their combination in Section 2. The implementation and two numerical illustrations are described in Section 3.

2 Description of the method

The method is a combination of a version of the Moore-Skelboe algorithm for interval global optimization and a version of the constraint propagation

method for solving a system of non-linear equations.
In Subsection 2.1 we describe the interval global optimization algorithm which is a branch-and-bound algorithm combined with the Krawczyk algorithm [10] for solving the non-linear equation $f'(x) = 0$. In Subsection 2.2 the constraint propagation algorithm for finding bounds for the solutions to $f'(x) = 0$ is described, and in Subsection 2.3 this constraint propagation algorithm is incorporated into the Moore-Skelboe algorithm.

2.1 The basic interval method

The algorithm is rigorous, i.e., it is guaranteed that all solutions are located. It is a branch and bound type method. At any stage of the algorithm we have the *candidate set* S. This is a finite set of sub-boxes $S_{(k)} \subseteq D$ having the property that the set of solutions to (1), X^*, is contained in the union of $\{S_{(k)}\}$. The aim is to reduce the candidate set, and to do that, let F be an interval extension of f (see [15]), and notice that

$$\min_k \{L(S_{(k)})\} \ \leq \ f^* \ \leq \ \min_k \{f(x_k)\} \ \equiv \ \tau \ , \tag{2}$$

where $L(S_{(k)})$ is the lower bound of $F(S_{(k)})$ and x_k is a random point in $S_{(k)}$. Therefore, if

$$L(S_{(k)}) > \tau \ , \tag{3}$$

then $S_{(k)}$ can be discarded from the candidate set. Otherwise, we can get sharper bounds by splitting $S_{(k)}$ into two smaller subregions. If $n = 1$ then the splitting is done by simple bisection, and in multiple dimensions we bisect in the direction of the largest component of the radius of $S_{(k)}$.
Suppose that we not only have access to the interval extension F, but also to an interval extension F' of the *gradient* $f' = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right)$ and the *Hessian* $f' = \left(\frac{\partial^2 f}{\partial x_i \partial x_j}\right)$. This information can be used in two ways:

1. Monotonicity. If $0 \notin \big(F'(S_{(k)})\big)_i$, then f is monotone in the component x_i on the subdomain $S_{(k)}$. Therefore, we can reduce the ith component of the interval vector $S_{(k)}$ to its lower (respectively upper) bound if $\big(F'(S_{(k)})\big)_i > 0$ (respectively $\big(F'(S_{(k)})\big)_i < 0$). Furthermore, if this reduced candidate is interior to the original domain D, then we can discard it from S because all

components of the gradient at an interior minimizer are zero.

2. Stationary points. Since an interior minimizer is a solution of the equation $f'(x) = 0$ we can use an interval equation solver to locate this minimizer. We prefer Krawczyk's method, [1], [10], which is a version of the interval Newton method based on the operator

$$K(x,X) \equiv x - H\,f'(x) + (I - H\,J(X))(X - x) \tag{4}$$

Here, $x \in X$, $I = \mathrm{diag}(1,\ldots,1)$ is the unit matrix, J is an interval extension the Jacobian of f' (i.e., the Hessian f'') and H is an arbitrary matrix in $\mathbb{R}^{n\times n}$. For efficiency reasons H should be chosen close to the inverse of $f''(x)$.
K has the following properties

- If x^* is a solution then $x^* \in X \Rightarrow x^* \in K(x,X)$.
- If $K(x,X) \subseteq X$ then there exists a solution x^* in $K(x,X)$.

Therefore the nested sequence

$$X_{(s+1)} \;=\; X_{(s)} \cap K(x_{(s)}, X_{(s)}) \;\text{ with }\; x_{(s)} \in X_{(s)} \qquad s = 0,1,\ldots \tag{5}$$

has the properties

- If $x^* \in X_{(0)}$ then $x^* \in X_{(s)}$, $s = 1, 2, \ \ldots$.
- If $X_{(s)} = \emptyset$ then no solution exists in $X_{(s)}$, and thus no solution exists in $X_{(0)}$.

Furthermore it has been proved that

- If $\{X_{(\cdot)}\}$ is convergent to x^* and $f''(x^*)$ is non-singular then the rate of convergence of (5) is quadratic.

Using (5) from $X_{(0)} = X = S_{(k)}$ there are three possible results:

a) If $X_{(s)} = \emptyset$ for some value of s then X contains no root. If X is interior in D, then we can discard X from the candidate set, otherwise we can reduce it to the union of its non-interior edges.

b) $\{X_{(\cdot)}\}$ converges to x^*. If (5) is stopped after iteration number s then any solution contained in X is also contained in $X_{(s+1)}$. Therefore $X \setminus X_{(s+1)}$ can be discarded from further search.

c) The iteration (5) stalls, maybe because X is too wide.
Use the splitting strategy to reduce the width.

Now we are ready to outline the algorithm. Let $w(X)$ denote the width of the interval X. We use the condition

$$w(S_{(k)}) \;\leq\; \delta \tag{6}$$

to decide when no further search should be done. Sub-boxes satisfying (6) are stored in the result set R. S is the Candidate Set. τ is the threshold value used in (2),(3), and we choose x_k as the mid point of the interval $S_{(k)}$. In each iteration we wish to pick the most promising box from S. This is chosen as the interval $S_{(k)}$ which has the smallest lower bound $L(S_{(k)})$.
For simplicity we assume that all minimizers are interior in D. Then the algorithm has the following structure:

Algorithm MS:

$S_{(1)} := D$
$S := \{S_{(1)}\}$
$\tau := f(\text{mid}(S_{(1)}))$
while $S \neq \emptyset$ **do**
  $X :=$ the most promising box in S
  remove X from S
  if Monotone(X) **then** { see 1. above }
    $R_X := \emptyset$
    else if (2a **or** 2b) **then** { see 2. above }
      X is reduced to R_X
    else { split }
      $R_X := (X_1, X_2)$ where $X = X_1 \cup X_2$
  end
  { *R_X contains 0, 1 or 2 elements* }
  with all $Z \in R_X$
    $\tau := \min\{\tau, f(\text{mid}(Z))$
    if $w(F(Z)) \leq \delta$ **then** $R := R \cup Z$
            else $S := S \cup Z$
  end
end { while }

This algorithm locates all solutions to (1), i.e., all solutions are contained in R which is the union of sub-boxes each of which having width less than δ. Algorithm MS has proven to be very efficient for problems with a rather modest number of variables (up to 15-20, say). If the number of variables is higher then the computing time may be severe since the worst case complexity of the algorithm is exponential. Under special circumstances, however, the algorithm may be efficient, even for a large number of variables.

2.2 Constraint propagation

Constrained propagation is here considered as an interval method for solving equations, as described in [9] and [22]. It is used as a method to reduce the set of possible candidates for a solution. The reduction is done rigorously, i.e., the method guarantees that no solution contained in the candidate set is lost.

The method is based on the *sub-definite calculations*. In order to solve a desired equation or inequality, the constraint corresponding to the value of the expression needs to be *propagated* through the expression. For instance, if we wish to find which values of $x \in \mathbb{R}$ satisfy the inequality $3x - 2 \geq 5$ then the feasible interval of function values $[5, \infty]$ is propagated through the expression $3x - 2$ until the set of feasible values of x is calculated.

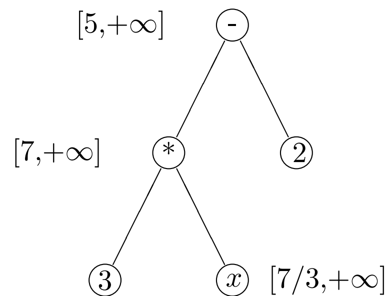

Figure 1 The calculus tree corresponding to $3x - 2$. The initial bound on the expression and the propagated bounds are shown as intervals.

Knowing the bounds of the whole expression, we first calculate the bounds of $3x$ and then the bounds of x. A very convenient way of illustrating how the method works (and actually the way to implement it too) is by constructing a *calculus tree* for the expression and propagating the constraint through it, see Figure 1. The calculus tree is made in such a way that each node corresponds to an operator in the expression of the function and each leaf in the tree being either a variable or a constant.
The operators can be both binary and unary, i.e., the nodes of the tree can have either one or two children.
In the context of global optimization constraint propagation is used to locate the set of stationary points, i.e., solving the equation $f'(x) = 0$. This equation is called the *propagated constraint*.
Thus the initial feasible set of function values, i.e., the interval attached to the root of the calculus tree representing $f'(x)$, is $[0, 0]$. Furthermore, the ranges of values for the independent variables are assigned. Finally interval values of the other nodes are assigned; if nothing is known a priori then the value $[-\inf, +\inf]$ is assigned to these nodes. Thus, all nodes in the tree are assigned an interval value. The constraint propagation method intends to reduce the interval ranges without loosing any solution.
Based on the propagated constraint the method works by walking through the calculation tree and updating the ranges of values attached to the nodes. For each calculation the intersection with the previous range is used. The method continues until no further changes in the values attached to the nodes can be made. The method does not necessarily provide precise bounds for the solutions, however no solution inside the initial interval of the variables is lost.
Following [22] we base the propagation on a so-called *code list*. For each node in the expression tree the code list displays the interaction between the node and its neighbours. Thus each function in the code list consists of just one operator, the one that defines the corresponding node in the tree, or its inverse operator.
The technique is illustrated through the following example.

Example 1. We wish to solve the equation

$$(f'(x) \equiv \,) \quad e^x - x - 2 = 0 \tag{7}$$

The calculus tree for the function $e^x - x - 2$ is shown in Figure 2. Since the

exponential is positive the initial value of the corresponding node, denoted by t_1, is included in the interval $]0, \infty]$. Equation (7) implies that the node denoted by t_3 has the value 0 attached. The variables t_2 and x are a priori only known to belong to the real line $[-\infty, \infty]$.

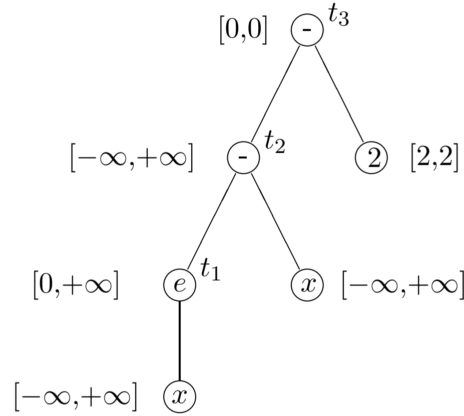

Figure 2 Calculus tree for the function $e^x - x - 2$ including the initial intervals for the nodes. The node names t_1, t_2, t_3 refer to the code list in Table 1.

The code list corresponding to the calculus tree is the following:

$$\begin{array}{l} f_1 : t_1 := exp(x) \cap t_1 \\ f_2 : x := ln(t_1) \cap x \\ f_3 : t_2 := (t_1 - x) \cap t_2 \\ f_4 : t_1 := (t_2 + x) \cap t_1 \\ f_5 : x := (t_1 - t_2) \cap x \\ f_6 : t_2 := (t_3 + 2) \cap t_2 \\ f_7 : t_3 := (t_2 - 2) \cap t_3 \end{array}$$

Table 1 The code list for the function $e^x - x - 2$.

As one can immediately see the code list is more than just another form for writing the mathematical expression for the function. There are only three operators in (7), however the code list consists of seven expressions. The reason for it is that the functions f_2, f_4, f_6 and f_5 are inferred from f_1, f_3 and

Step num.	Working function	Sub-definite values $\mathbf{t_3}$	$\mathbf{t_2}$	$\mathbf{t_1}$	$\mathbf{x}$
0		[0,0]	$[-\infty, +\infty]$	$[0, +\infty]$	$[-1000, +1000]$
1	f_6		$[2, 2]$		
2	f_5				$[-2, 1000]$
3	f_4			$[0, 1002]$	
4	f_2				$[-2, 6.91]$
5	f_1			$[0.13, 1002]$	
6	f_5				$[-1.87, 6.91]$
7	f_4			$[0.13, 8.91]$	
8	f_2				$[-1.87, 2.19]$
9	f_1			$[0.15, 8.91]$	
10	f_5				$[-1.85, 2.19]$
11	f_4			$[0.15, 4.19]$	
12	f_2				$[-1.85, 1.43]$
13	f_1			$[0.15, 4.18]$	
14	f_4			$[0.15, 3.43]$	
15	f_2				$[-1.85, 1.24]$

Table 2 The first fifteen steps for the example (7).

f_7, thus giving the opportunity to move up and down the calculus tree and propagate the constraint.
Now the idea is to reduce the intervals using the code list and the known inclusions. This goes on until no further reduction takes place.
We start from below in the code list (Table 1) and move upwards: Since $t_3 = 0$ we obtain $t_2 = 2$. Next f_5 gives the interval]-2, ∞] for x. This implies that t_1 is reduced to]exp(-2),∞]=]0.135,∞] by f_1. If we repeat this sweep we obtain further reductions in t_1 and x, however the rate of reduction is rather slow.
In the global optimization problem (1) the variables are finitely bounded. Therefore, let us repeat the example starting with finite bounds on x, $x \in [-1000, 1000]$. Again we obtain $t_3 = 0$ and $t_2 = 2$ initially. Then we obtain the values given in Table 2 (in order to be rigorous we round intervals outwards).

In practice the code list is easily derived from the corresponding calculus tree. The picture of the tree actually represents how the tree or DAG (Directed

Acyclic Graph) looks in our implementation. This is illustrated in Table 1 and Figure 2.
Schematically the constraint propagation method as described in [9], [22] is the following:

Algorithm CP

1. Execute all the functions of the constructed code list.
2. If the value of any variable has been changed after the execution of one of the functions, then mark all functions having this variable as an argument.
3. If the set of marked functions is empty then stop.
4. Select and execute an arbitrary function from the set of marked functions and go to 2.

The algorithm corresponds to going up and down the calculus tree, executing the functions of the code list, until no more changes can be made in the node values. As the propagation can go both upwards and downwards a node that can tentatively be changed is included into the set of marked functions, *the active function set*. It keeps the track of the updates of the node values.
The selection in (4) can be done in numerous ways. If a node value in the calculus tree has been changed, then one could for instance update the values of its children or one could update the values of all of the neighbours of the node. Such a small difference can play a big role in the performance of constraint propagation, however it is not clear which choice is generally the best.
However, our experience indicates that in connection with the global optimization algorithm the most efficient is to perform only one sweep and stop, i.e., in Example 1 we would stop after step number 5. This is because the major changes often take place in the first sweep.

2.3 Algorithm MS extended with constraint propagation

The *Krawczyk method* (5) and the *Algorithm CP* both intend to solve a nonlinear set of equations ($f'(x) = 0$). However they perform quite differently.

Krawczyk provides fast convergence to a narrow (machine precision) box when the necessary condition is satisfied. Unfortunately the box $X_{(0)}$ often has to be quite narrow around the solution in order for Krawczyk to converge. The CP is normally not as quickly convergent as Krawczyk and the limit box cannot be expected to be as narrow as that provided by Krawczyk, especially when the function expression is complex. However CP does not necessarily need a narrow initial box. Therefore CP may often be be used to provide an initial reduction whereas Krawczyk provides the ultimate convergence. CP may even reduce boxes with more than one stationary point. This is demonstrated in Figure 3:

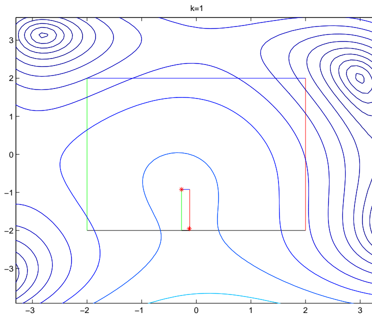

Figure 3 Contour plots. Constraint propagation tightening the box $[-2,2] \times [-2,2]$ to a narrow box around the two stationary points marked as dots.

Figure 3 shows the contour plot for the function

$$f(x) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2 + 3$$

If we apply CP with the starting box $[-2,2] \times [-2,2]$, then the algorithm will reduce the box to the small one, just containing the two stationary points.
These observations indicate that a combination of the two methods may exploit the strong sides of each of them. Therefore we use Algorithm CP if the Krawczyk method does not provide any reduction in Algorithm MS. In other words we use both methods as root finding methods in Algorithm MS. This is done by first employing one of them; if no progress has been made then we apply the other. If the candidate has still not been reduced then it is split into two. Thus the only difference compared with Algorithm MS is that instead of one interval method for solving $f'(x) = 0$ two interval methods are tried.

The combined algorithm is the following:

Algorithm MSCP:

```
S_(1) := D
S := {S_(1)}
τ := f(mid(S_(1)))
while S ≠ ∅ do
   X := the most promising box in S
   remove X from S
   if Monotone(X)  then     { see 1. in Subsection 2.1 }
      R_X := ∅
      else if ( 2a or 2b )  then     { see 2. in Subsection 2.1 }
            X is reduced to R_X
      else if ( CP works )  then
            X is reduced to R_X
      else { split }
         R_X := (X1,X2) where X = X1 ∪ X2
   end
   { R_X contains 0, 1 or 2 elements }
   with all Z ∈ R_X
      τ := min{τ, f(mid(Z))
      if  w(F(Z)) ≤ δ then R := R ∪ Z
                         else  S := S ∪ Z
   end
end { while }
```

3 Implementation and numerical results

Algorithm MSCP has been implemented in C++ using SUN's Interval package (Suninterval). We have tested the programme on several problems. Here we illustrate the results by two examples only.

3.1 Implementation

When implementing Constraint Propagation a tree-structure is used. In fact, the constraint programming implementation is an extension of the au-

tomatic differentiation library FADBAD [2], [24], which implements forward and backward differentiation. FADBAD has been extended in such a way that the expression trees used to store functions and derivatives are reused for implementing the constraint propagation.
Thus the integrated programme automatically calculates first and second derivatives as well as the CP code, and the user only has to programme the expression for $f(x)$.
A discussion of some implementation issues can be found in [9], Section 7. Here we shall only mention one difficulty which is connected with power functions. We illustrate the problem by considering the square operator, $y = x^2$. The problem occurs because the inverse operator splits into two, $+\sqrt{x}$ and $-\sqrt{x}$. Thus in such a case a more complicated tree structure is needed which might give rise to exponential growth in the amount of space needed to store the tree.

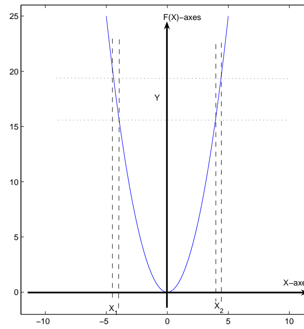

Figure 4 Illustration of the problem occurring when implementing the inverse of the function $y = x^2$.

The situation is illustrated in in Figure 4. When intersecting the two new intervals $X_1 = +\sqrt{X}$ and $X_2 = -\sqrt{X}$ with the previous interval value X_{prev} we have the following three cases:
If $X_{prev} \cap X_1 = \emptyset$ then $X_{new} := X_{prev} \cap X_2$, and no problem occurs.
If $X_{prev} \cap X_2 = \emptyset$ then $X_{new} := X_{prev} \cap X_1$, and no problem occurs.
If $X_{prev} \cap X_1 \neq \emptyset$ and $X_{prev} \cap X_2 \neq \emptyset$ then X_{new} splits into two unless $0 \in X$. In order to avoid this problem we let X_{new} be the hull of the two non-empty intersections.
Our implementation can be formulated as follows:

$$X_{new} := (X_{old} \cap X_1) \bigcup (X_{old} \cap X_2)$$

where $\bigcup$ is the interval hull of the union.
This implementation is of course rigorous, however it is pessimistic, since we throw away information by taking the hull. This may cause a slower convergence rate of the CP algorithm, and a sharp implementation of inverse power functions is planed in the future.

3.2 Some numerical tests

In order to investigate the efficiency of Algorithm MSCP it has been tested on several problems. Some of these tests are described in [9] where different variations of the implementation are examined.
The most important results of these tests are the following:

- Since Algorithm CP as described in Subsection 2.2 is running the same simple equations in each CP iteration, it can be expected to provide most of the reduction during the beginning. Does it matter whether Algorithm CP is run until the end, where no further reduction of the interval is possible (as it is described in Subsection 2.2), or is it better to run Algorithm CP just once down the tree, i.e., to execute each working function in the code list only once, as described in [14]?
  On the test examples tried it turned out that Algorithm MSCP was about twice as fast when the one-loop implementation of Algorithm CP was used rather than running full CP in each iteration.

- Krawczyk's method (4) was tested in a loop as described in (5) as well as in just one iteration of (5). Like Algorithm CP Krawczyk's method gives the largest reduction in the first iteration. And on the test examples tried in [9] it also turned out that Algorithm MSCP was fastest when only one loop of (5) was used, rather than running full iteration each time.

In the test examples below we have used these results: For (5) as well as for Algorithm CP only one loop is run in each iteration of Algorithm MSCP.
The first illustration is the so-called Schwefel test problem [21] which is tested for several values of n:

$$f(x) = \sum_{i=1}^{n} \{-x_i \sin(\sqrt{x_i})\} \qquad D = [1, 500]^n$$

Figure 5 shows the performance of the two algorithms for $n = 1, 2, \ldots, 18, 30$. It is seen that although the computing time is exponential in n for both algorithms, Algorithm MSCP is far better than Algorithm MS. The numbers of splits show the same tendency as the CPU time. The computing times for $n = 15$ are approximately 10 hours for Algorithm MS and 18 seconds for Algorithm MSCP.

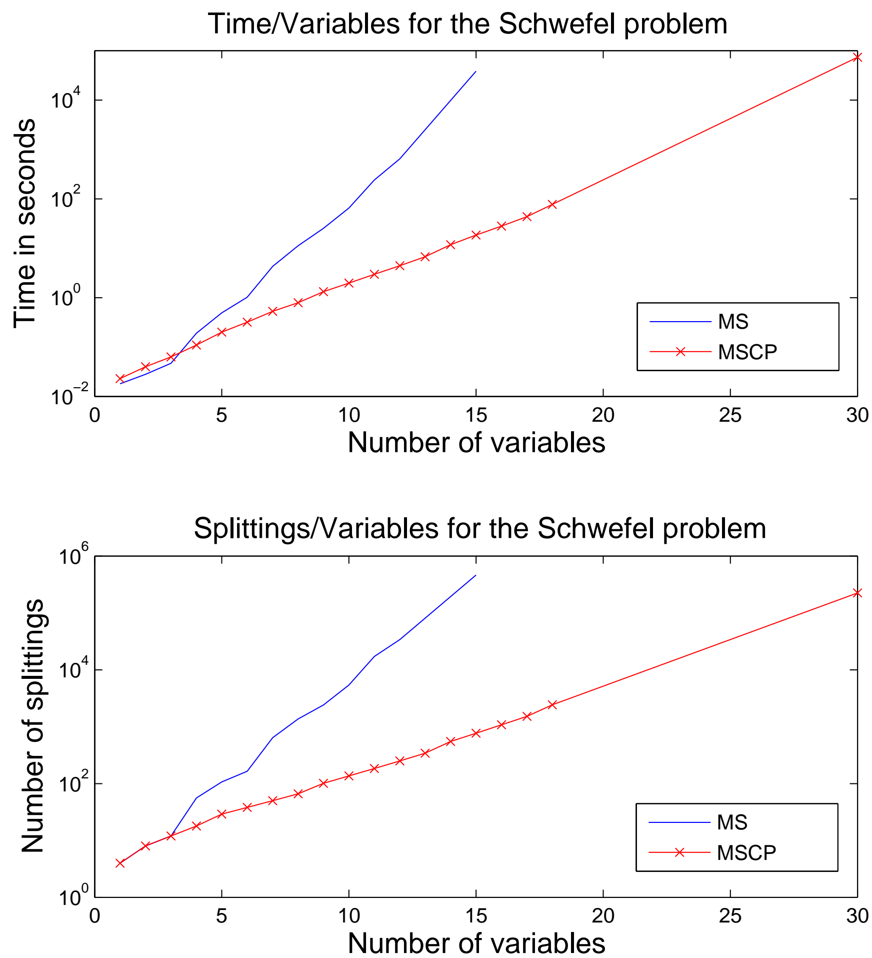

Figure 5 Plot showing the CPU times and the number of interval splits when applying the two algorithms Algorithm MS and Algorithm MSCP to the Schwefel problem for $n = 1, 2, \ldots, 18, 30$.

Figure 5 illustrates some tendencies we also have seen in other examples: The constraint programming is very efficient when power functions are involved, and it is not so sensitive as Algorithm MS to an increasing number of variables, i.e., the factor in the exponential growth is much smaller. In other words: The more variables the better Algorithm MSCP compares with the classical Algorithm MS.
It is easily seen that the number of splits in Algorithm MS cannot be less than the number of splits in Algorithm MSCP. However, the amount of work per split is higher for Algorithm MSCP, and we have seen examples where the CPU time for the latter method is higher. This is for instance the case for the so-called Levy function, [5], when the number of variables is less than 6. It is the following:

$$f(x) = \sin(3\pi x_1) + \sum_{i=1}^{n-1} \{(x_i - 1)^2 (1 + \sin^2(3\pi x_{i+1}))\} + (x_n - 1)(1 + \sin^2(3\pi x_{i+1}))$$

where the initial box is $D = [-5, 5]^n$. This function has a very large number of local minimizers: For $n = 5, 6, 7$ it is approximately $10^5, 10^6, 10^8$, respectively. In this case the number of splits for Algorithm MSCP is about half of the number of splits in Algorithm MS. The computing times for Algorithm MS are 11, 83 and 537 seconds for $n = 5, 6, 7$, respectively, whereas the corresponding computing times for Algorithm MSCP are 12, 80 and 330 seconds, respectively.

We have made an experimental comparison with R. Baker Kearfott's optimization package GlobSol [8], release November 2003, which also uses a combination of interval global optimization and constraint propagation. GlobSol has a lot of features and is a much more complex programme than Algorithm MSCP. Like Algorithm MSCP GlobSol includes automatic differentiation. In order to make a fair comparison we used the mode in GlobSol where it is assumed that no solution exists at the boundary of D.
In general it turned out that for many problems GlobSol was faster than Algorithm MSCP. When power function are involved in calculating f, however, Algorithm MSCP is much faster than GlobSol. For problems where we could increase the number of variables n, both programmes illustrated an exponential growth similar to that displayed in Figure 5.
The two examples in this paper are typical for the comparisons: For the Levy function, $n = 7$, GlobSol used 54 seconds to find the solution with

high accuracy whereas Algorithm MSCP used 5.5 minutes. For the Schwefel problem Algorithm MSCP was much faster when n is large. For $n = 10$ GlobSol used 2.5 minutes to to find the solution with high accuracy whereas Algorithm MSCP used 1 second. For $n = 13$ the CPU times were 50 minutes for GlobSol and 6 seconds for Algorithm MSCP.

4 Conclusion

The classical interval global optimization algorithm has proved to be very efficient for a large class of problems, especially when the number of variables is modest, [5]. We combine this method with constrained propagation, as a tool for enclosing the set of stationary points. The combination has been implemented and tested, and two typical test examples are displayed in this chapter. An important fact that can be concluded from the tests in [9] is, that the use of constraint propagation makes each iteration of the global optimization algorithm more time consuming, however the number of iterations and bisections is often reduced, sometimes drastically. This fact is crucial when dealing with the problems with high number of variables.

References

[1] Ole Caprani, Kaj Madsen, and Hans Bruun Nielsen. *Introduction to Interval Analysis*. Available at http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/1462/pdf/imm1462.pdf,
Informatics and Mathematical Modelling, Technical University of Denmark, DK-2800 Lyngby, Denmark, 2002.

[2] G. Corliss, C. Faure, A. Griewank, L. Hascoët and U. Naumann (editors). *Automatic differentiation of algorithms* . Springer Verlag, New York, 2002.

[3] R. Fletcher. *Practical Methods of Optimization*. Second Edition, John Wiley and Sons, 1987.

[4] Laurent Granvilliers, Frédéric Benhamou, and Etienne Huens. *Constraint Propagation*. Chapter 5, in "COCONUT Deliverable D1 : Algorithms for

Solving Nonlinear Constrained and Optimization Problems : The State of The Art." The COCONUT Project, 2001.

[5] E. Hansen and G.W. Walster. *Global Optimization using Interval Analysis.* Marcel Dekker Inc., New York, 2004.

[6] Pascal Van Hentenryck, Laurent Michel, and Yves Deville. *Numerica.* MIT Press, 1997.

[7] R. Horst, P.M. Pardalos and N.V. Thoai. *Introduction to Global Optimization.* Nonconvex optimization and its applications Vol. 48, 2. edition, Kluwer Academic Publishers, 2000.

[8] R. Baker Kearfott. *The GlobSol Project.* http://interval.louisiana.edu/private/downloading_instructions.html, release date 22. november 2003.

[9] S. Kjøller and P. Kozine. *Global Optimization using Constraint Propagation and Krawczyk's method.* Informatics and Mathematical Modelling, Technical University of Denmark, DK-2800 Lyngby, Denmark, 2005.

[10] R. Krawczyk, *Newton-Algorithmen zur Bestimmung von Nulstellen mit Fehlerschranken*, Computing **4**, pp 187-201, 1969.

[11] K. Madsen. *Real versus Interval Methods for Global Optimization.* Presentation at the Conference 'Celebrating the 60th Birthday of M.J.D. Powell', Cambridge, July 1996.

[12] K. Madsen and J. Žilinskas. *Testing of branch-and-bound methods for global optimization.* IMM-REP-2000-05, Department of Mathematical Modelling, Technical University of Denmark, DK-2800 Lyngby, Denmark, 2000.

[13] K. Madsen and J. Žilinskas. *Evaluating performance of attraction based subdivision method for global optimization.* Second International Conference 'Simulation, Gaming, Training and Business Process Reengineering in Operations', RTU, Latvia, 38-42, 2000.

[14] F. Messine, *Deterministic global optimization using interval constraint propagation techniques.* RAIRO Operations Research, **38**, 277-293, 2004.

[15] R.E. Moore, *Interval Analysis.* Prentice Hall, Englewood Cliffs, N.J., 1966.

[16] R.E. Moore. *On computing the range of values of a rational function of n variables over a bounded region.* Computing **16**, pp 1-15, 1976.

[17] A. Neumaier. *Interval methods for systems of equations.* Cambridge University Press, 1990.

[18] A. Neumaier. *Introduction to Numerical Analysis.* Cambridge University Press, 2001.

[19] J. Nocedal and S.J. Wright. *Numerical Optimization.* Springer Verlag, 1999.

[20] V.J. Rayward-Smith, S.A. Rush and G.P. McKeown. *Efficiency considerations in the implementation of parallel branch-and-bound.* Annals of Operations Research 43, 123-145, 1993.

[21] H. Schwefel. *Numerical Optimization of Computer Models.* John Wiley and Sons, 1981.

[22] A. Semenov, *Solving integer/real equations by constraint propagation.* Technical report, Informatics and Mathematical Modelling, Technical University of Denmark, 1994.

[23] O. Stauning, *Automatic Validation of Numerical Solutions.* Technical report, Informatics and Mathematical Modelling, Technical University of Denmark, 1994.

[24] O. Stauning and C. Bendtsen, *Flexible Automatic differentiation using templates and operator overloading in ANSI C++.* http://www2.imm.dtu.dk/~km/FADBAD/, Technical University of Denmark, 2003.

[25] S. Skelboe (1974), *Computation of rational interval functions* . BIT **14**, pp 87-95.

[26] A. Törn and A. Žilinskas, *Global Optimization.* Lecture Notes in Computer Science, 350, Springer Verlag, 1987.