AMO - Advanced Modeling and Optimization Volume 1, Number 2, 1999

Global Optimization Using the DIRECT Algorithm in Matlab [1]

Mattias Björkman[2] and Kenneth Holmström[3]
Center for Mathematical Modeling
Department of Mathematics and Physics
Mälardalen University, P.O. Box 883, SE-721 23 Västerås, Sweden

Abstract

We discuss the efficiency and implementation details of an algorithm for finding the global minimum of a multi-variate function subject to simple bounds on the variables. The algorithm DIRECT, developed by D. R. Jones, C. D. Perttunen and B. E. Stuckman, is a modification of the standard Lipschitzian approach that eliminates the need to specify a Lipschitz constant. We have implemented the DIRECT algorithm in Matlab and the efficiency of our implementation is analyzed by comparing it to the results of Jones's implementation on nine standard test problems for box-bounded global optimization. In fifteen out of eighteen runs the results are in favor of our implementation. We also present performance results for our implementation on the more challenging test set used in the first contest on evolutionary optimization (ICEO). An application from computational finance is also discussed.

Our DIRECT code is available in two versions. One, *glbSolve*, is integrated in the Matlab optimization environment TOMLAB, as part of the toolbox NLPLIB TB for nonlinear programming and parameter estimation. The other, *gblSolve*, is a stand-alone version. Both TOMLAB and *gblSolve* are free for academic use and downloadable at the home page of the Applied Optimization and Modeling group, see the URL: **http://www.ima.mdh.se/tom**.

Keywords: Global Optimization, Lipschitzian Optimization, DIRECT, Matlab, Software Engineering, Mathematical Software, Optimization, Algorithms.

AMS Subject Classification: 90C26, 90C30, 90C99

1 Introduction

TOMLAB [4, 7], developed by the Applied Optimization and Modeling group (TOM) at Mälardalen University, is an open Matlab environment for research and teaching in optimization. TOMLAB is based on NLPLIB TB [5], a Matlab toolbox for nonlinear programming and parameter estimation, and OPERA TB [6], a Matlab toolbox for linear and discrete optimization. Although TOMLAB includes more than 65 different optimization algorithms, until recently there has been no routine included that handles global optimization problems. Therefore the DIRECT algorithm focused our interest.

DIRECT is an algorithm developed by Donald R. Jones et al. [9] for finding the global minimum of a multi-variate function subject to simple bounds, using no derivative information. The algorithm is a modification of the standard Lipschitzian approach that eliminates the need to specify a Lipschitz constant. The idea is to carry out simultaneous searches using all possible constants from zero to infinity. In [9] they introduce a different way of looking at the Lipschitz constant. Really, the

[1] Financed by the Mälardalen University Research Board, project *Applied Optimization and Modeling (TOM)*.
[2] E-mail: **mbk@mdh.se**.
[3] E-mail: **hkh@mdh.se**; URL: **http://www.ima.mdh.se/tom**.

Lipschitz constant is viewed as a weighting parameter that indicate how much emphasis to place on global versus local search. In standard Lipschitzian methods, this constant is usually large because it must be equal to or exceed the maximum rate of change of the objective function. As a result, these methods place a high emphasis on global search, which leads to slow convergence. In contrast, the DIRECT algorithm carries out simultaneous searches using all possible constants, and therefore operates on both the global and local level.

Now define some notation to be used throughout this paper, in order to avoid confusion. By DIRECT we mean the algorithm described in [9]. When we talk in comparative terms we will by DIRECT mean the implementation made by Jones, Perttunen and Stuckman. Our implementation of the DIRECT algorithm (including modifications) will be referred to as *glbSolve*, which also is the name of the corresponding Matlab routine included in the NLPLIB TB. The stand-alone version *gblSolve* is identical to *glbSolve*, except for the input and output format.

This paper is organized as follows. In Section 2 we present some of the key ideas behind the DIRECT algorithm. We will also give a formal description, following the implementation in *glbSolve*. An important phase in the algorithm is to find the extreme points on the lower convex hull of a set of points in the plane. A formal description of the technique we use to solve this problem, and a presentation of other implementation details, are also given in Section 2. In Section 3 we define the part of the test problems used in the comparison to Jones's implementation. In Section 4 we present the results of the comparison of *glbSolve* versus DIRECT together with our performance results for the ICEO problems. We also present a graphical illustration of the search behavior. A practical application from computational finance is discussed in Section 5. Section 6 summarizes the results from our point of view. In Appendix A an example of the use of the stand-alone version *gblSolve* is described and in Appendix B the full Matlab code is given.

2 The Algorithm

In the first part of this section we discuss some of the key ideas of the DIRECT algorithm. We will not give a complete description and motivation for those ideas, instead see [9], where a complete and well-written presentation is given. Some details in our implementation, which are not described in [9], will also be pointed out. In the second part of this section a formal description of the algorithm is given, which is close to our real implementation in *glbSolve*.

DIRECT deals with problems on the form

$$\begin{array}{llcccc} \min\limits_{x} & f(x) & & & & \\ \text{s.t.} & x_L & \leq & x & \leq & x_U \end{array},$$

where $f \in \mathbb{R}$ and $x, x_L, x_U \in \mathbb{R}^n$. It is guaranteed to converge to the global optimal function value, if the objective function f is continuous or at least continuous in the neighborhood of a global optimum. This could be guaranteed since, as the number of iterations goes to infinity, the set of points sampled by DIRECT form a dense subset of the unit hypercube. In other words, given any point x in the unit hypercube and any $\delta > 0$, DIRECT will eventually sample a point (compute the objective function) within a distance δ of x.

The first step in the DIRECT algorithm is to transform the search space to be the unit hypercube. The function is then sampled at the center-point of this cube. Computing the function value at the center-point instead of doing it at the vertices is an advantage when dealing with problems in higher dimensions. The hypercube is then divided into smaller hyperrectangles whose center-points are also sampled. Instead of using a Lipschitz constant when determining the rectangles to sample next, DIRECT identifies a set of *potentially optimal* rectangles in each iteration. All *potentially optimal* rectangles are further divided into smaller rectangles whose center-points are sampled. When no Lipschitz constant is used, there is no natural way of defining convergence (except when the optimal function value is known as in the test problems). Instead, the procedure described above is performed for a predefined number of iterations. In our implementation it is possible to **restart** the optimization with the final status of all parameters from the previous run.

As an example, apply *glbSolve* on a certain problem for 50 iterations. Then run e.g. 40 iterations more. The result is the same as a run for 90 iterations in the first place.

The problem of finding the extreme points on the lower convex hull of a set of points in the plane, is introduced as a subproblem when to determine the set of all *potentially optimal* rectangles. In *glbSolve*, the extreme points on the lower convex hull is identified by use of the subroutine *conhull*. As a result of the technique we use in our implementation of DIRECT, *conhull* will assume that the points are sorted by increasing abscissas. The implementation of *conhull* is based on the algorithm GRAHAMHULL in [10, page 108], with the modifications proposed on page 109. The initial step in GRAHAMHULL is to sort the points by increasing abscissas. As mentioned above this is already done so this initial step is skipped. The Algorithm *conhull* is stated as follows.

Algorithm conhull

The points (x_i, y_i), $i = 1, 2, 3, ..., m$ are given with $x_1 \leq x_2 \leq ... \leq x_m$.
Set $h = (1, 2, ..., m)$.
if $m \geq 3$ **then**
  Set $START = 1$, $v = START$, $w = m$ and $flag = 0$.
  while $next(v) \neq START$ **or** $flag = 0$ **do**
    if $next(v) = w$ **then**
      Set $flag = 0$.
    end if
    Set $a = v$, $b = next(v)$ and $c = next(next(v))$.
    Set $A = \begin{pmatrix} x_a & y_a & 1 \\ x_b & y_b & 1 \\ x_c & y_c & 1 \end{pmatrix}$.
    if $\det(A) \geq 0$ **then**
      Set $leftturn = 0$.
    else
      Set $leftturn = 1$.
    end if
    if $leftturn$ **then**
      Set $v = next(v)$.
    else
      Set $j = next(v)$.
      Set $x = (x_1, x_2, ..., x_{j-1}, x_{j+1}, ..., x_m)$, $y = (y_1, y_2, ..., y_{j-1}, y_{j+1}, ..., y_m)$ and
        $h = (h_1, h_2, ..., h_{j-1}, h_{j+1}, ..., h_m)$.
      Set $m = m - 1$, $w = w - 1$ and $v = pred(v)$.
    end if
  end while
end if

The index vector h contains the indices to the points which lies on the boundary of the convex hull. The function $next(v)$ returns $v + 1$ if $v < m$ and 1 if $v = m$. The function $pred(v)$ returns $v - 1$ if $v > 1$ and m if $v = 1$.

Below is a formal description of our DIRECT algorithm, the Algorithm *glbSolve*, which in close detail describes our implementation in the Matlab code *glbSolve*, see the code in Appendix B. Instead of using the tree structure proposed in [9] for storing the information of each rectangle, we use a straightforward matrix/index-vector technique. The notation used will be explained in Table 1.

Algorithm glbSolve

Set the global/local search weight parameter ϵ, e.g. $\epsilon = 10^{-4}$.
Set $C_{i1} = \frac{1}{2}$ and $L_{i1} = \frac{1}{2}$, $i = 1, 2, 3, ..., n$.
Set $F_1 = f(x)$, where $x_i = x_{L_i} + C_{i1}(x_{U_i} - x_{L_i})$, $i = 1, 2, 3, ..., n$.
Set $D_1 = \sqrt{\sum_{k=1}^{n} L_k^2}$.
Set $f_{min} = F_1$ and $i_{min} = 1$.

Table 1: Notation used in the *glbSolve* algorithm description.

Parameter	Explanation
C	Matrix with all rectangle center-points.
D	Vector with distances from center-point to the vertices.
F	Vector with function values.
I	Set of dimensions with maximum side length for the current rectangle.
L	Matrix with all rectangle side lengths in each dimension.
S	Index set of *potentially optimal* rectangles.
T	The number of iterations to be performed.
f_{min}	The current minimum function value.
i_{min}	Rectangle index.
ϵ	Global/local search weight parameter.
δ	New side length in the current divided dimension.

for $t = 1, 2, 3, \ldots, T$ **do**

Set $\hat{S} = \left\{ j : D_j \geq D_{i_{min}} \wedge F_j = \min_i \{F_i : D_i = D_j\} \right\}$.

Define α and β by letting the line $y = \alpha x' + \beta$ pass through the points $(D_{i_{min}}, F_{i_{min}})$ and $\left(\max_j (D_j), \min_i \{F_i : D_i = \max_j (D_j)\} \right)$.

Let $\tilde{S}$ be the set of all rectangles $j \in \hat{S}$ fulfilling $F_j \leq \alpha D_j + \beta + 10^{-12}$.

Let S be the set of all rectangles in $\tilde{S}$ being extreme points in the convex hull of the set $\left\{ (D_j, F_j) : j \in \tilde{S} \right\}$.

while $S \neq \emptyset$ **do**

Select j as the first element in S, set $S = S \setminus \{j\}$.

Let I be the set of dimensions with maximum rectangle side length, i.e. $I = \left\{ i : D_{ij} = \max_k (D_{kj}) \right\}$.

Let δ equal two-thirds of this maximum side length, i.e. $\delta = \frac{2}{3} \max_k (D_{kj})$.

for all $i \in I$ **do**

Set $c_k = C_{kj}$, $k = 1, 2, 3, \ldots, n$.

Set $\hat{c} = c + \delta e_i$ and $\tilde{c} = c - \delta e_i$, where e_i is the ith unit vector.

Compute $\hat{f} = f(\hat{x})$ and $\tilde{f} = f(\tilde{x})$ where $\hat{x}_k = x_{L_k} + \hat{c}_k \left(x_{U_k} - x_{L_k} \right)$ and $\tilde{x}_k = x_{L_k} + \tilde{c}_k \left(x_{U_k} - x_{L_k} \right)$.

Set $w_i = \min(\hat{f}, \tilde{f})$.

Set $C = \begin{pmatrix} C & \hat{c} & \tilde{c} \end{pmatrix}$ and $F = \begin{pmatrix} F & \hat{f} & \tilde{f} \end{pmatrix}$.

end for

while $I \neq \emptyset$ **do**

Select the dimension $i \in I$ with the lowest value of w_i and set $I = I \setminus \{i\}$.

Set $L_{ij} = \frac{1}{2}\delta$.

Let $\hat{j}$ and $\tilde{j}$ be the indices corresponding to the points $\hat{c}$ and $\tilde{c}$, i.e. $F_{\hat{j}} = \hat{f}$ and $F_{\tilde{j}} = \tilde{f}$

Set $L_{k\hat{j}} = L_{kj}$ and $L_{k\tilde{j}} = L_{kj}$, $k = 1, 2, 3, \ldots, n$.

Set $D_j = \sqrt{\sum_{k=1}^{n} L_{kj}^2}$.

Set $D_{\hat{j}} = D_j$ and $D_{\tilde{j}} = D_j$.

end while

end while

Set $f_{min} = \min_j (F_j)$.

Set $i_{min} = \text{argmin} \left(\frac{F_j - f_{min} + E}{D_j} \right)$, where $E = \max \left(\epsilon |f_{min}|, 10^{-8} \right)$.

end for

3 The Test Problems

In this section we first give a complete description of the test problems used in the comparisons in Section 4.1. We use the same problem names and abbreviations as in [9]. Table 2 gives a compact description of the test problems including information about the abbreviation used, the problem dimension and the number of local and global minima.

Table 2: Compact description of the test problems.

Problem	Abbreviation	Problem dimension	Number of local minima	Number of global minima
Shekel 5	S5	4	5	1
Shekel 7	S7	4	7	1
Shekel 10	S10	4	10	1
Hartman 3	H3	3	4	1
Hartman 6	H6	6	4	1
Branin RCOS	BR	2	3	3
Goldstein and Price	GP	2	4	1
Six-Hump Camel	C6	2	6	2
Two-Dimensional Schubert	SHU	2	760	18

The problem definitions of problem S5, S7, S10, H3, H6 and GP are taken from [1], problem BR from [8, page 468] and problem C6 and SHU from [11]. Information about the optimal function values, denoted f_{global}, were kindly supplied by D. R. Jones.

- Shekel 5, Shekel 7 and Shekel 10:

$$\begin{array}{ll} \min\limits_{x} & f(x) = -\sum\limits_{i=1}^{m} \frac{1}{(x-a_i)^T(x-a_i)+c_i} \\ \text{s.t.} & 0 \leq x_j \leq 10, \quad j = 1,2,...,n \ , \end{array}$$

where $n = 4$, a_i is the ith row in A and c_i is the ith element in c. A and c is defined by

$$A = \begin{pmatrix} 4 & 4 & 4 & 4 \\ 1 & 1 & 1 & 1 \\ 8 & 8 & 8 & 8 \\ 6 & 6 & 6 & 6 \\ 3 & 7 & 3 & 7 \\ 2 & 9 & 2 & 9 \\ 5 & 5 & 3 & 3 \\ 8 & 1 & 8 & 1 \\ 6 & 2 & 6 & 2 \\ 7 & 3.6 & 7 & 3.6 \end{pmatrix}, c = \begin{pmatrix} 0.1 \\ 0.2 \\ 0.2 \\ 0.4 \\ 0.4 \\ 0.6 \\ 0.3 \\ 0.7 \\ 0.5 \\ 0.5 \end{pmatrix}.$$

Consider the three cases with $m = 5, 7$ and 10,

$$\begin{array}{ll} m = 5: & f_{global} = -10.1531996790582, \\ m = 7: & f_{global} = -10.4029405668187, \\ m = 10: & f_{global} = -10.5364098166920. \end{array}$$

- Hartman 3 and Hartman 6:

$$\begin{array}{ll} \min\limits_{x} & f(x) = -\sum\limits_{i=1}^{m} c_i \exp\left(-\sum\limits_{j=1}^{m} a_{ij}(x_j - p_{ij})^2\right) \\ \text{s.t.} & 0 \leq x_j \leq 1, \quad j = 1,2,...,n \ . \end{array}$$

For Hartman 3, $n = 3$ and A, c and P is given by

$$A = \begin{pmatrix} 3 & 10 & 30 \\ 0.1 & 10 & 35 \\ 3 & 10 & 30 \\ 0.1 & 10 & 35 \end{pmatrix}, c = \begin{pmatrix} 1 \\ 1.2 \\ 3 \\ 3.2 \end{pmatrix}, P = \begin{pmatrix} 0.3689 & 0.1170 & 0.2673 \\ 0.4699 & 0.4387 & 0.7470 \\ 0.1091 & 0.8732 & 0.5547 \\ 0.03815 & 0.5743 & 0.8828 \end{pmatrix},$$

and for Hartman 6, $n = 6$ and A, c and P is given by

$$A = \begin{pmatrix} 10 & 3 & 17 & 3.5 & 1.7 & 8 \\ 0.05 & 10 & 17 & 0.1 & 8 & 14 \\ 3 & 3.5 & 1.7 & 10 & 17 & 8 \\ 17 & 8 & 0.05 & 10 & 0.1 & 14 \end{pmatrix}, c = \begin{pmatrix} 1 \\ 1.2 \\ 3 \\ 3.2 \end{pmatrix},$$

$$P = \begin{pmatrix} 0.1312 & 0.1696 & 0.5569 & 0.0124 & 0.8283 & 0.5886 \\ 0.2329 & 0.4135 & 0.8307 & 0.3736 & 0.1004 & 0.9991 \\ 0.2348 & 0.1451 & 0.3522 & 0.2883 & 0.3047 & 0.6650 \\ 0.4047 & 0.8828 & 0.8732 & 0.5743 & 0.1091 & 0.0381 \end{pmatrix},$$

$$\begin{array}{ll} n = 3: & f_{global} = -3.86278214782076, \\ n = 6: & f_{global} = -3.32236801141551. \end{array}$$

- Branin RCOS:

$$\begin{array}{ll} \min\limits_x & f(x) = \left(x_2 - \frac{5x_1^2}{4\pi^2} + \frac{5x_1}{\pi} - 6\right)^2 + 10\left(1 - \frac{1}{8\pi}\right)\cos(x_1) + 10 \\ \text{s.t.} & \begin{array}{rcccl} -5 & \leq & x_1 & \leq & 10 \\ 0 & \leq & x_2 & \leq & 15, \end{array} \end{array}$$

$$f_{global} = 0.397887357729739.$$

- Goldstein and Price:

$$\begin{array}{lll} \min\limits_x \quad f(x) & = & \left[1 + (x_1 + x_2 + 1)^2 \left(19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2\right)\right] \\ & \times & \left[30 + (2x_1 - 3x_2)^2 \left(18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2\right)\right] \\ \text{s.t.} & & -2 \leq x_j \leq 2, \quad j = 1, 2\ , \end{array}$$

$$f_{global} = 3.$$

- Six-Hump Camel:

$$\begin{array}{ll} \min\limits_x & f(x) = \left(4 - 2.1x_1^2 + \frac{1}{3}x_1^4\right)x_1^2 + x_1x_2 + \left(-4 + 4x_2^2\right)x_2^2 \\ \text{s.t.} & \begin{array}{rcccl} -3 & \leq & x_1 & \leq & 3 \\ -2 & \leq & x_2 & \leq & 2, \end{array} \end{array}$$

$$f_{global} = -1.0316284535.$$

- Two-Dimensional Schubert:

$$\begin{array}{ll} \min\limits_x & f(x) = \left[\sum\limits_{i=1}^{5} i\cos\left[(i+1)x_1 + i\right]\right] \times \left[\sum\limits_{i=1}^{5} i\cos\left[(i+1)x_2 + i\right]\right] \\ \text{s.t.} & -10 \leq x_j \leq 10, \quad j = 1, 2\ , \end{array}$$

$$f_{global} = -186.730908831024.$$

The problems S5, S7, S10, H3, H6 and GP from [1] has often been criticized for being easy test problems. A more challenging test set was used in the first contest on evolutionary optimization (ICEO) at the ICEO'96 conference. This test bed contains five problems, each in a 5-dimensional and a 10-dimensional version. For these problems the value to reach f_{reach} is given instead of f_{global}. In Section 4.2 we will present the results obtained when applying *glbSolve* to the ICEO test problems.

- Sphere model:

$$\begin{array}{llcccl} \min\limits_x & f(x) = \sum\limits_{i=1}^{n} (x_i - 1)^2 \\ \text{s.t.} & -5 & \le & x_i & \le & 5, \quad i = 1,2,...,n \ , \end{array}$$

$$f_{reach} = 10^{-6}.$$

- Griewank's function:

$$\begin{array}{llcccl} \min\limits_x & f(x) = \frac{\sum\limits_{i=1}^{n} (x_i-100)^2}{4000} - \prod\limits_{i=1}^{n} \cos(\frac{x_i-100}{\sqrt{i}}) + 1 \\ \text{s.t.} & -600 & \le & x_i & \le & 600, \quad i = 1,2,...,n \ , \end{array}$$

$$f_{reach} = 10^{-6}.$$

- Shekel's foxholes:

$$\begin{array}{llcccl} \min\limits_x & f(x) = -\sum\limits_{i=1}^{30} \frac{1}{(x-a_i)^T(x-a_i)+c_i} \\ \text{s.t.} & 0 & \le & x_j & \le & 10, \quad j = 1,2,...,n \ , \end{array}$$

where a_i is the ith row in A and c_i is the ith element in c. A and c is defined by

$$A^T = \begin{pmatrix}
9.681 & 0.667 & 4.783 & 9.095 & 3.517 & 9.325 & 6.544 & 0.211 & 5.122 & 2.020 \\
9.400 & 2.041 & 3.788 & 7.931 & 2.882 & 2.672 & 3.568 & 1.284 & 7.033 & 7.374 \\
8.025 & 9.152 & 5.114 & 7.621 & 4.564 & 4.711 & 2.996 & 6.126 & 0.734 & 4.982 \\
2.196 & 0.415 & 5.649 & 6.979 & 9.510 & 9.166 & 6.304 & 6.054 & 9.377 & 1.426 \\
8.074 & 8.777 & 3.467 & 1.863 & 6.708 & 6.349 & 4.534 & 0.276 & 7.633 & 1.567 \\
7.650 & 5.658 & 0.720 & 2.764 & 3.278 & 5.283 & 7.474 & 6.274 & 1.409 & 8.208 \\
1.256 & 3.605 & 8.623 & 6.905 & 0.584 & 8.133 & 6.071 & 6.888 & 4.187 & 5.448 \\
8.314 & 2.261 & 4.224 & 1.781 & 4.124 & 0.932 & 8.129 & 8.658 & 1.208 & 5.762 \\
0.226 & 8.858 & 1.420 & 0.945 & 1.622 & 4.698 & 6.228 & 9.096 & 0.972 & 7.637 \\
7.305 & 2.228 & 1.242 & 5.928 & 9.133 & 1.826 & 4.060 & 5.204 & 8.713 & 8.247 \\
0.652 & 7.027 & 0.508 & 4.876 & 8.807 & 4.632 & 5.808 & 6.937 & 3.291 & 7.016 \\
2.699 & 3.516 & 5.874 & 4.119 & 4.461 & 7.496 & 8.817 & 0.690 & 6.593 & 9.789 \\
8.327 & 3.897 & 2.017 & 9.570 & 9.825 & 1.150 & 1.395 & 3.885 & 6.354 & 0.109 \\
2.132 & 7.006 & 7.136 & 2.641 & 1.882 & 5.943 & 7.273 & 7.691 & 2.880 & 0.564 \\
4.707 & 5.579 & 4.080 & 0.581 & 9.698 & 8.542 & 8.077 & 8.515 & 9.231 & 4.670 \\
8.304 & 7.559 & 8.567 & 0.322 & 7.128 & 8.392 & 1.472 & 8.524 & 2.277 & 7.826 \\
8.632 & 4.409 & 4.832 & 5.768 & 7.050 & 6.715 & 1.711 & 4.323 & 4.405 & 4.591 \\
4.887 & 9.112 & 0.170 & 8.967 & 9.693 & 9.867 & 7.508 & 7.770 & 8.382 & 6.740 \\
2.440 & 6.686 & 4.299 & 1.007 & 7.008 & 1.427 & 9.398 & 8.480 & 9.950 & 1.675 \\
6.306 & 8.583 & 6.084 & 1.138 & 4.350 & 3.134 & 7.853 & 6.061 & 7.457 & 2.258 \\
0.652 & 2.343 & 1.370 & 0.821 & 1.310 & 1.063 & 0.689 & 8.819 & 8.833 & 9.070 \\
5.558 & 1.272 & 5.756 & 9.857 & 2.279 & 2.764 & 1.284 & 1.677 & 1.244 & 1.234 \\
3.352 & 7.549 & 9.817 & 9.437 & 8.687 & 4.167 & 2.570 & 6.540 & 0.228 & 0.027 \\
8.798 & 0.880 & 2.370 & 0.168 & 1.701 & 3.680 & 1.231 & 2.390 & 2.499 & 0.064 \\
1.460 & 8.057 & 1.336 & 7.217 & 7.914 & 3.615 & 9.981 & 9.198 & 5.292 & 1.224 \\
0.432 & 8.645 & 8.774 & 0.249 & 8.081 & 7.461 & 4.416 & 0.652 & 4.002 & 4.644 \\
0.679 & 2.800 & 5.523 & 3.049 & 2.968 & 7.225 & 6.730 & 4.199 & 9.614 & 9.229 \\
4.263 & 1.074 & 7.286 & 5.599 & 8.291 & 5.200 & 9.214 & 8.272 & 4.398 & 4.506 \\
9.496 & 4.830 & 3.150 & 8.270 & 5.079 & 1.231 & 5.731 & 9.494 & 1.883 & 9.732 \\
4.138 & 2.562 & 2.532 & 9.661 & 5.611 & 5.500 & 6.886 & 2.341 & 9.699 & 6.500
\end{pmatrix}, \quad c = \begin{pmatrix}
0.806 \\ 0.517 \\ 0.100 \\ 0.908 \\ 0.965 \\ 0.669 \\ 0.524 \\ 0.902 \\ 0.531 \\ 0.876 \\ 0.462 \\ 0.491 \\ 0.463 \\ 0.714 \\ 0.352 \\ 0.869 \\ 0.813 \\ 0.811 \\ 0.828 \\ 0.964 \\ 0.789 \\ 0.360 \\ 0.369 \\ 0.992 \\ 0.332 \\ 0.817 \\ 0.632 \\ 0.883 \\ 0.608 \\ 0.326
\end{pmatrix},$$

$$f_{reach} = -9.$$

- Michalewicz's function:

$$\begin{array}{llcccl} \min\limits_x & f(x) = -\sum\limits_{i=1}^{n} \left(\sin(x_i) \sin^{20}(\frac{ix_i^2}{\pi}) \right) \\ \text{s.t.} & 0 & \le & x_i & \le & \pi, \quad i = 1,2,...,n \ , \end{array}$$

$$\begin{array}{ll} n = 5: & f_{reach} = -4.687, \\ n = 10: & f_{reach} = -9.66, \\ else: & f_{reach} = -0.966n. \end{array}$$

- Langerman's function:

$$\begin{array}{ll} \min\limits_{x} & f(x) = -\sum\limits_{i=1}^{30} \left(c_i \exp\left(-\frac{(x-a_i)^T(x-a_i)}{\pi} \right) \cos\left((x-a_i)^T(x-a_i)\pi \right) \right) \\ \text{s.t.} & 0 \quad \leq \quad x_j \quad \leq \quad 10, \quad j = 1, 2, ..., n \ , \end{array}$$

where a_i is the ith row in A and c_i is the ith element in c. A and c are the same as in Shekels's foxholes but with $c_3 = 1.5$.

$$f_{reach} = -1.4.$$

4 Computational Results

In this section we present some numerical experience that have been obtained with the implementation of DIRECT in *glbSolve*.

To illustrate the search behavior we have, as in [9], chosen to present scatter plots for the Branin's problem after 16 and 45 iterations with 231 and 1017 function evaluations respectively, see Figure 1 and 2. Branin's function have three global optimum and the sampled points clearly cluster around them.

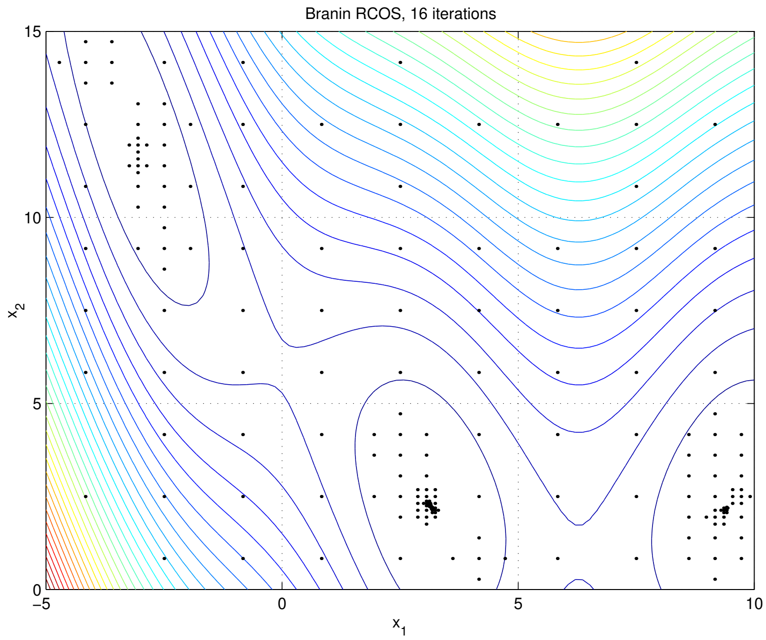

Figure 1: Scatter plot of Branin RCOS after 16 iterations and 231 function evaluations.

4.1 Comparison Results

In Table 3 we compare the efficiency of *glbSolve* versus DIRECT. The efficiency is measured as the number of function evaluations needed for convergence. We use the same definition of convergence as in [9], where convergence is defined in terms of percent error from the globally optimal function value. Let f_{global} denote the known optimal function value and let f_{min} denote the best function value at some point in the search, then the percent error is defined by

$$E = 100 \frac{f_{min} - f_{global}}{|f_{global}|}.$$

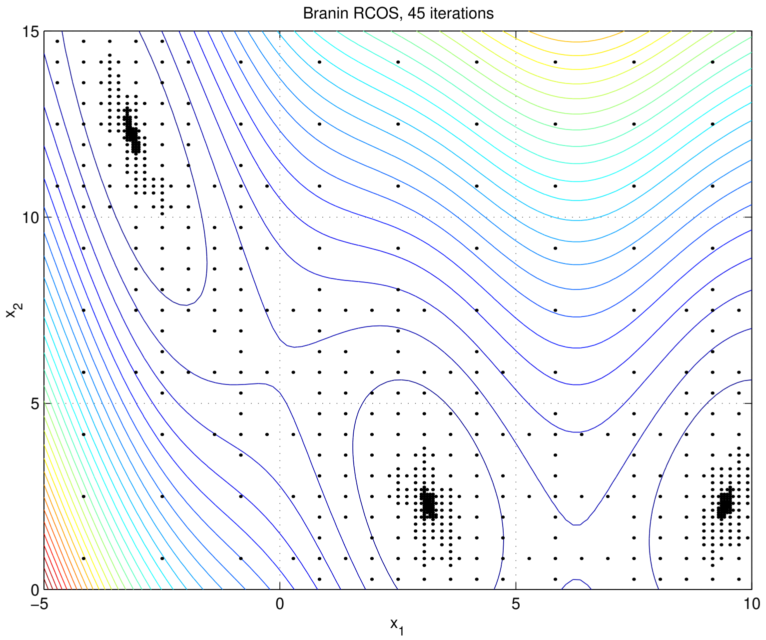

Figure 2: Scatter plot of Branin RCOS after 45 iterations and 1017 function evaluations.

In all cases, the parameter ϵ was set to 10^{-4}, the value of ϵ used in [9].

Table 3: Number of function evaluations, *glbSolve* vs. DIRECT.

Routine		S5	S7	S10	H3	H6	BR	GP	C6	SHU	Av.	Tot.
glbSolve	$E < 1\%$	100	94	94	70	198	65	83	77	3193	442	3974
DIRECT	$E < 1\%$	103	97	97	83	213	63	101	113	2883	417	3753
glbSolve	$E < 0.01\%$	153	143	143	178	529	165	167	146	3274	544	4898
DIRECT	$E < 0.01\%$	155	145	145	199	571	195	191	285	2967	539	4853

As seen in table 3, *glbSolve* needs fewer function evaluations in 15 of the 18 runs. For the first six problems, the differences are small but that is not the fact for problem C6 and SHU. It is difficult to give a clear explanation of those great differences, but for problem SHU with 760 local minima and 18 global minima, we think that the numerical tolerances used in the implementations play an important role. The average and total number of function evaluations are here a bit misleading since the SHU problem affects those values strongly. It is important to mention that we have spent no time and no effort adjusting or tuning the numerical tolerances.

4.2 Computational Results for the ICEO Test Problems

For the ICEO test problems, we can not provide comparison results as in the previous section so we restrict to just present our results. In Table 4, the number of function evaluations needed to reach the value f_{reach} is presented. Note that we also consider the case $n = 2$ for each problem.

5 Practical Applications

In our research on prediction methods in computational finance, we study the prediction of various kinds of quantities related to stock markets, like stock prices, stock volatility and ranking measures.

Table 4: Number of function evaluations needed for *glbSolve* to reach f_{reach}. † indicates that the value was not reached after 15000 evaluations.

Problem	$n=2$	$n=5$	$n=10$
Sphere model	281	7477	†
Griewank's function	6252	†	†
Shekel's foxholes	45	770	†
Michalewicz's function	†	13911	†
Langerman's function	27	†	†

In one project we instead of the classical time series approach used the more realistic prediction problem of building a multi-stock artificial trader (ASTA). The behavior of the trader is controlled by a parameter vector which is tuned for best performance. Here, *glbSolve* is used to find the optimal parameters for the noisy functions obtained, when running on a large database of Swedish stock market data [3].

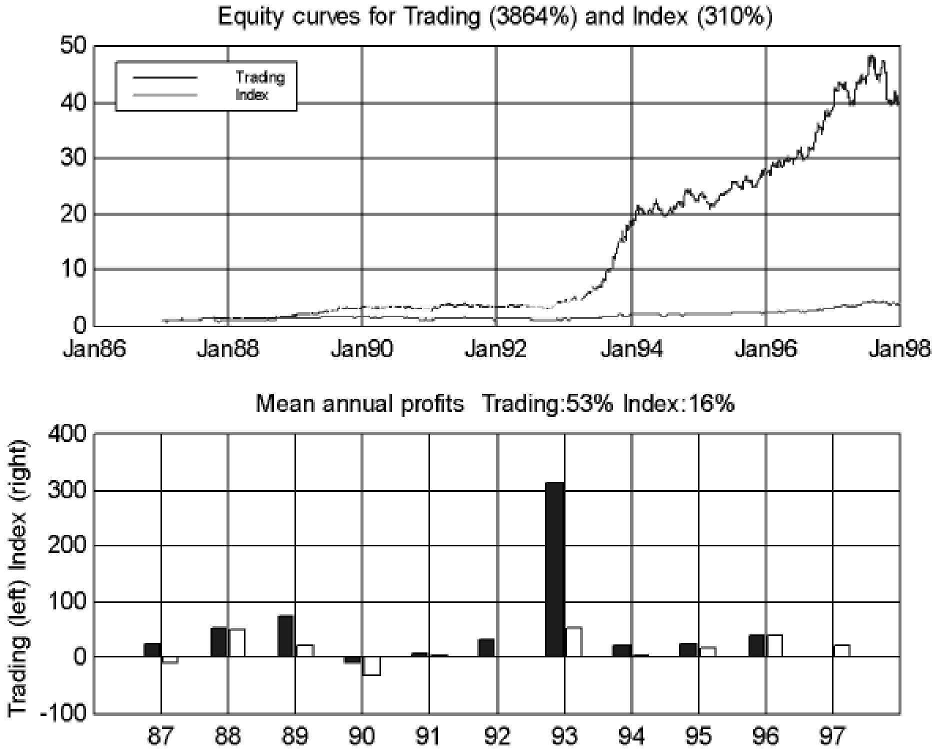

Figure 3: Performance of the trading function $Stoch(30, 3, 3, 20, 80)$ based on the Stochastics indicator.

The Stochastics Indicator is a classical technical trading rule. We have obtained very good results in ASTA using this rule to select buy and sell rules in a multi-stock trading algorithm, see Figure 4 for a performance diagram that compares the trading results with the stock market index. We tried to tune two of the parameters in this trading rule. In Figure 4 we see the points sampled when trying to find the optimal buy and sell rules in the Stochastics Indicator. They cluster around $(40, 78)$, which seems to be the global optimum. In Figure 5 one-dimensional views of the *Net profit* (with reversed sign) versus the *Buylevel* and the *Sellevel* are shown. The optimum is more well-determined and distinct in the *Buylevel*. The global optimum is in fact very close to the standard values used in technical analysis. Further testing and analysis are needed to establish robustness properties of the parameters found.

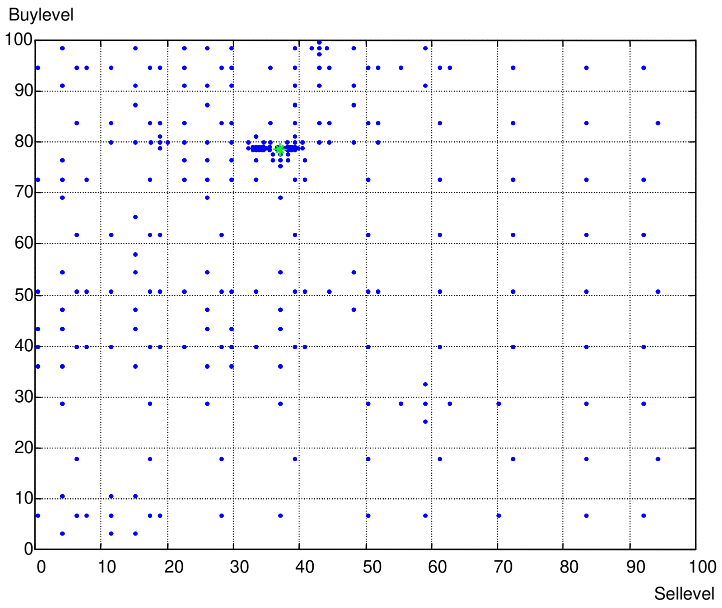

Figure 4: Sampled points by *glbSolve* in the parameter space when optimizing the buy and sell levels for the trading function $Stoch(30, 3, 3, Sellevel, Buylevel)$.

6 Conclusions

We have presented details and some numerical results on the implementation of the DIRECT algorithm in Matlab. The Matlab environment gives access to several utility functions, which made the implementation rather straightforward. Further, having *glbSolve* as part of the TOMLAB toolbox for nonlinear programming NLPLIB TBand also possible to run using the Graphical User Interface (GUI) [2], it could be of use for many purposes.

The numerical results obtained are pleasant. In [9], DIRECT is shown to perform well compared to other algorithms. We compare our implementation to DIRECT and the results are to our favor in most of the cases. On the ICEO test problems, *glbSolve* showed some limitations, but these problems are considered to be hard problems. *glbSolve* has been successfully used in practical applications.

Acknowledgements

We would like to thank Dr. Donald R. Jones who has, with the DIRECT algorithm, inspired us to include routines for global optimization in our optimization environment TOMLAB. He has been very helpful in discussions on details of the algorithm, as well as providing information concerning

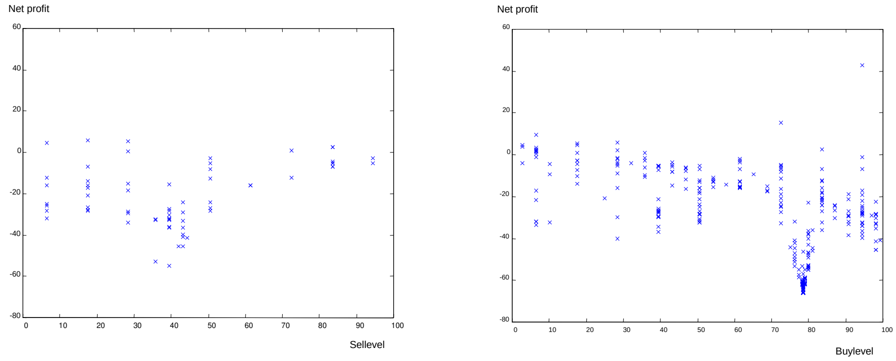

Figure 5: One-dimensional views of the global optimization of the parameters in the trading function $Stoch(30, 3, 3, Sellevel, Buylevel)$. The left graph shows the *Net profit* versus the *Buylevel* for an equidistant grid of values of the *Sellevel*. The right graph shows the *Net profit* versus the *Sellevel* for an equidistant grid of values of the *Buylevel*.

some of the test problems. We would also like to thank Prof. Arnold Neumaier, who has kindly supplied Matlab code for the ICEO test problems.

Appendix A. An Example of the Use of *gblSolve*

Here we give an example of how to run the *gblSolve* solver, the stand-alone version of the TOMLAB NLPLIB TB solver *glbSolve*. In Appendix B the full code of *glbSolve* is given. Assume you want to solve the problem Branin RCOS defined in Section 3 using *gblSolve*.

1. Create a Matlab m-file function for computing the objective function f.

```
function f = funct1(x);
f = (x(2)-5*x(1)^2/(4*pi^2)+5*x(1)/pi-6)^2+10*(1-1/(8*pi))*cos(x(1))+10;
```

2. Define the input arguments at the Matlab prompt:

```
fun = 'funct1';
x_L = [-5  0]';
x_U = [10 15]';
GLOBAL.iterations = 20;
PriLev = 2;
```

3. Now, you can call *gblSolve*:

```
Result = gblSolve(fun,x_L,x_U,GLOBAL,PriLev);
```

4. Assign the best function value found to f_{opt} and the corresponding point(s) to x_{opt}:

```
f_opt = Result.f_k

f_opt =

    0.3979
```

```
x_opt = Result.x_k

x_opt =

    3.1417
    2.2500
```

Note that the number of iterations and the printing level are not necessary to supply (they are by default set to 50 and 1 respectively). Also note that *gblSolve* has no explicit stopping criteria and therefore it runs a predefined number of iterations.

It is possible to **restart** *gblSolve* with the current status on the parameters from the previous run. Assume you have run 20 iterations as in the example above, and then you want to restart and run 30 iterations more (this will give exactly the same result as running 50 iterations in the first run). To use the restart option do:

```
...
...
Result = gblSolve(fun,x_L,x_U,GLOBAL,PriLev);   % First run

GLOBAL = Result.GLOBAL;
GLOBAL.iterations = 30;

Result = gblSolve(fun,x_L,x_U,GLOBAL,PriLev);   % Restart
```

If you want a scatter plot of all sampled points in the search space, do:

```
C = Result.GLOBAL.C;
plot(C(1,:),C(2,:),'.');
```

Appendix B. The Matlab Code for *gblSolve*

```
% This is a standalone version of glbSolve.m which is a part of the
% optimization environment TOMLAB, see  http://www.ima.mdh.se/tom/
%
% gblSolve implements the algorithm DIRECT by D. R. Jones,
% C. D. Perttunen and B. E. Stuckman presented in the paper
% "Lipschitzian Optimization Without the Lipschitz Constant",
% JOTA  Vol. 79, No. 1, October 1993.
% All page references and notations are taken from this paper.
%
% gblSolve solves problems of the form:
%
%    min   f(x)
%     x
%    s/t   x_L <= x <= x_U
%
% Calling syntax:
%
% function Result = gblSolve(fun,x_L,x_U,GLOBAL,PriLev)
%
% INPUT PARAMETERS
%
% fun      Name of m-file computing the function value, given as a string.
```

```
% x_L      Lower bounds for x
% x_U      Upper bounds for x
%
% GLOBAL.iterations   Number of iterations to run, default 50.
% GLOBAL.epsilon      Global/local search weight parameter, default 1E-4.
%
% If restart is wanted, the following fields in GLOBAL should be defined
% and equal the corresponding fields in the Result structure from the
% previous run:
% GLOBAL.C          Matrix with all rectangle centerpoints.
% GLOBAL.D          Vector with distances from centerpoint to the vertices.
% GLOBAL.L          Matrix with all rectangle side lengths in each dimension.
% GLOBAL.F          Vector with function values.
% GLOBAL.d          Row vector of all different distances, sorted.
% GLOBAL.d_min      Row vector of minimum function value for each distance
%
% PriLev            Printing level:
%                   PriLev >= 0   Warnings
%                   PriLev >  0   Small info
%                   PriLev >  1   Each iteration info
%
% OUTPUT PARAMETERS
%
% Result            Structure with fields:
%    x_k            Matrix with all points fulfilling f(x)=min(f).
%    f_k            Smallest function value found.
%    Iter           Number of iterations
%    FuncEv         Number of function evaluations.
%    GLOBAL.C       Matrix with all rectangle centerpoints.
%    GLOBAL.D       Vector with distances from centerpoint to the vertices.
%    GLOBAL.L       Matrix with all rectangle side lengths in each dimension.
%    GLOBAL.F       Vector with function values.
%    GLOBAL.d       Row vector of all different distances, sorted.
%    GLOBAL.d_min   Row vector of minimum function value for each distance
%
% Mattias Bjorkman, Optimization Theory, Dep of Mathematics and Physics,
% Malardalen University, P.O. Box 883, SE-721 23 Vasteras, Sweden.
% E-mail: mattias.bjorkman@mdh.se
% Written               Sep 1,  1998.              Last modified May 4,  1999.
% K. Holmstrom modified Oct 13, 1998. K. Holmstrom Last modified Nov 28, 1998.
%

function Result = gblSolve(fun,x_L,x_U,GLOBAL,PriLev)

if nargin < 5;
   PriLev = [];
   if nargin < 4
      GLOBAL = [];
      if nargin < 3
         x_U = [];
         if nargin < 2
            x_L = [];
            if nargin < 1
               fun = [];
            end
         end
      end
   end
end
```

```
if isempty(PriLev),   PriLev=1;  end
if isempty(fun) | isempty(x_L) | isempty(x_U)
   disp(' gblSolve requires at least three nonempty input arguments')
   return;
end
if isempty(GLOBAL)
   T       = 50;     % Number of iterations
   epsilon = 1e-4;   % global/local weight parameter.
   tol     = 0.01;   % Error tolerance parameter.
else
   if isfield(GLOBAL,'iterations') % Number of iterations
      T = GLOBAL.iterations;
   else
      T = 50;
   end
   if isfield(GLOBAL,'epsilon') % global/local weight parameter
      epsilon = GLOBAL.epsilon;
   else
      epsilon = 1E-4;
   end
   if isfield(GLOBAL,'tolerance') % Convergence tolerance
      tol = GLOBAL.tolerance;
   else
      tol = 0.01;
   end
end

nFunc=0;
convflag=0;

x_L = x_L(:);
x_U = x_U(:);
n = length(x_L);  % Problem dimension

tolle  = 1E-16;
tolle2 = 1E-12;

%
%  STEP 1, Initialization
%
if isfield(GLOBAL,'C') & ~isempty(GLOBAL.C)
   % Restart with values from previous run.
   F = GLOBAL.F;
   m = length(F);

   if PriLev > 0
      fprintf('\n Restarting with %d sampled points from previous run\n',m);
   end

   D = GLOBAL.D;
   L = GLOBAL.L;
   d = GLOBAL.d;
   d_min = GLOBAL.d_min;

   f_min = min(F);
   E = max(epsilon*abs(f_min),1E-8);
   [dummy i_min] = min( (F - f_min + E)./D );

   % Must transform Prob.GLOBAL.C back to unit hypercube
   for i = 1:m
```

```
      C(:,i) = ( GLOBAL.C(:,i) - x_L )./(x_U - x_L);
   end
else
   % No restart, set first point to center of the unit hypercube.
   m = 1;              % Current number of rectangles
   C = ones(n,1)./2;  % Matrix with all rectangle centerpoints
   % All C_coordinates refers to the n-dimensional hypercube.

   x_m = x_L + C.*(x_U - x_L);   % Transform C to original search space
   f_min = feval(fun, x_m);  % Function value at x_m
   f_0 = f_min;
   nFunc=nFunc+1;
   i_min = 1; % The rectangle which minimizes (F - f_min + E)./D where
              % E = max(epsilon*abs(f_min),1E-8)

   L = ones(n,1)./2;  % Matrix with all rectangle side lengths in each dimension
   D = sqrt(sum(L.^2));  % Vector with distances from centerpoint to the vertices
   F = [f_min];            % Vector with function values

   d     = D;     % Row vector of all different distances, sorted
   d_min = f_min; % Row vector of minimum function value for each distance
end


% ITERATION LOOP
t = 1;  % t is the iteration counter
while t <= T  &  ~convflag

   %
   %  STEP 2  Identify the set S of all potentially optimal rectangles
   %
   S = [];  % Set of all potentially optimal rectangles

   S_1 = [];
   idx = find(d==D(i_min));
   %idx = find(abs( d-D(i_min) ) <= tolle );
   if isempty(idx)
      if PriLev >= 0
         fprintf('\n WARNING: Numerical trouble when determining S_1\n');
      end
      return;
   end
   for i = idx : length(d)
      idx2 = find( ( F==d_min(i) ) & ( D==d(i) ) );
      %idx2 = find( (abs( F-d_min(i) ) <= tolle ) & ( abs( D-d(i) ) <= tolle ) );
      S_1 = [S_1 idx2];
   end
   % S_1 now includes all rectangles i, with D(i) >= D(i_min)
   % and F(i) is the minimum function value for the current distance.

   % Pick out all rectangles in S_1 which lies below the line passing through
   % the points: ( D(i_min), F(i_min) ) and the lower rightmost point.
   S_2 = [];
   if length(d)-idx > 1
      a1 = D(i_min);
      b1 = F(i_min);
      a2 = d(length(d));
      b2 = d_min(length(d));
      % The line is defined by: y = slope*x + const
      slope = (b2-b1)/(a2-a1);
```

```
   const = b1 - slope*a1;
   for i = 1 : length(S_1)
      j = S_1(i);
      if F(j) <= slope*D(j) + const + tolle2
         S_2 = [S_2 j];
     end
   end
   % S_2 now contains all points in S_1 which lies on or below the line

   % Find the points on the convex hull defined by the points in S_2
   xx = D(S_2);
   yy = F(S_2);
   h  = conhull(xx,yy); % conhull is an internal subfunction
   S_3 = S_2(h);
else
   S_3 = S_1;
end
S = S_3;


%  STEP 3, 5  Select any rectangle j in S
for jj = 1:length(S)  % For each potentially optimal rectangle
   j = S(jj);

   %
   %  STEP 4  Determine where to sample within rectangle j and how to
   %          divide the rectangle into subrectangles. Update f_min
   %          and set m=m+delta_m, where delta_m is the number of new
   %          points sampled.

   %  4:1 Identify the set I of dimensions with the maximum side length.
   %      Let delta equal one-third of this maximum side length.
   max_L = max(L(:,j));
   I = find( L(:,j)==max_L );
   % I = find( abs( L(:,j) - max_L ) < tolle);
   delta = 2*max_L/3;

   %  4:2 Sample the function at the points c +- delta*e_i for all
   %      i in I.
   w=[];
   for ii = 1:length(I) % for each dimension with maximum side length
      i = I(ii);
      e_i = [zeros(i-1,1);1;zeros(n-i,1)];

      c_m1 = C(:,j) + delta*e_i;       % Centerpoint for new rectangle

      % Transform c_m1 to original search space
      x_m1 = x_L + c_m1.*(x_U - x_L);
      f_m1 = feval(fun, x_m1);    % Function value at x_m1
      nFunc=nFunc+1;

      c_m2 = C(:,j) - delta*e_i;       % Centerpoint for new rectangle
      x_m2 = x_L + c_m2.*(x_U - x_L);  % Transform c_m2 to original search space
      f_m2 = feval(fun, x_m2);    % Function value at x_m2
      nFunc=nFunc+1;

      w(ii) = min(f_m1,f_m2);

      C = [C c_m1 c_m2];  % Matrix with all rectangle centerpoints
      F = [F f_m1 f_m2];  % Vector with function values
```

```
      end

      %  4:3 Divide the rectangle containing C(:,j) into thirds along the
      %      dimension in I, starting with the dimension with the lowest
      %      value of w(ii)
      [a b] = sort(w);
      for ii = 1:length(I)
         i = I(b(ii));

         ix1 = m + 2*b(ii)-1; % Index for new rectangle
         ix2 = m + 2*b(ii);   % Index for new rectangle

         L(i,j) = delta/2;

         L(:,ix1) = L(:,j);
         L(:,ix2) = L(:,j);

         D(j)   = sqrt(sum(L(:,j).^2));
         D(ix1) = D(j);
         D(ix2) = D(j);

      end
      m = m + 2*length(I);
   end

   % UPDATE:
   f_min = min(F);
   E = max(epsilon*abs(f_min),1E-8);
   [dummy i_min] = min( (F - f_min + E)./D );

   d = D;
   i = 1;
   while 1
      d_tmp = d(i);
      idx = find(d~=d_tmp);
      d = [d_tmp d(idx)];
      if i==length(d)
         break;
      else
         i = i + 1;
      end
   end
   d = sort(d);

   d_min = [];
   for i = 1:length(d);
      idx1 = find(D==d(i));
      %idx1 = find( abs( D-d(i) ) <= tolle );
      d_min(i) = min(F(idx1));
   end

   if PriLev > 1
      fprintf('\n Iteration: %d   f_min: %15.10f   Sampled points: %d',...
         t,f_min,nFunc);
   end

   t = t + 1;
end % ITERATION LOOP
```

```
% SAVE RESULTS

Result.f_k  = f_min;    % Best function value
Result.Iter = T;        % Number of iterations

CC = [];
for i = 1:m % Transform to original coordinates
   CC = [CC x_L+C(:,i).*(x_U-x_L)];
end
Result.GLOBAL.C = CC;   % All sampled points in original coordinates
Result.GLOBAL.F = F;    % All function values computed
Result.GLOBAL.D = D;    % All distances
Result.GLOBAL.L = L;    % All lengths
Result.GLOBAL.d = d;
Result.GLOBAL.d_min = d_min;

% Find all points i with F(i)=f_min
idx = find(F==f_min);
Result.x_k = CC(:,idx);    % All points i with F(i)=f_min

Result.FuncEv=nFunc;



function h = conhull(x,y);
% conhull returns all points on the convex hull, even redundant ones.
%
% conhull is based on the algorithm GRAHAMSHULL pages 108-109
% in "Computational Geometry" by Franco P. Preparata and
% Michael Ian Shamos.
%
% Input vector x must be sorted i.e. x(1) <= x(2) <= ... <= x(length(x)).
%
x = x(:);
y = y(:);
m = length(x);
if length(x) ~= length(y)
   disp('Input dimension must agree, error in conhull-gblSolve');
   return;
end
if m == 2
   h = [1 2];
   return;
end
if m == 1
   h = [1];
   return;
end
START = 1;
v = START;
w = length(x);
flag = 0;
h = [1:length(x)]'; % Index vector for points in convex hull
while ( next(v,m)~=START ) | ( flag==0 )
   if next(v,m) == w
      flag = 1;
   end
   a = v;
   b = next(v,m);
```

```
   c = next(next(v,m),m);
   if det([ x(a) y(a) 1 ; x(b) y(b) 1 ; x(c) y(c) 1 ]) >= 0
      leftturn = 1;
   else
      leftturn = 0;
   end
   if leftturn
      v = next(v,m);
   else
      j = next(v,m);
      x = [x(1:j-1);x(j+1:m)];
      y = [y(1:j-1);y(j+1:m)];
      h = [h(1:j-1);h(j+1:m)];
      m=m-1;
      w=w-1;
      v = pred(v,m);
   end
end

function i = next(v,m);
if v==m
   i = 1;
else
   i = v + 1;
end

function i = pred(v,m);
if v==1
   i = m;
else
   i = v - 1;
end
```

References

[1] L. C. W. Dixon and G. P. Szegö. The global optimisation problem: An introduction. In L. Dixon and G Szego, editors, *Toward Global Optimization*, pages 1–15, New York, 1978. North-Holland Publishing Company.

[2] Erik Dotzauer and Kenneth Holmström. The TOMLAB Graphical User Interface for Nonlinear Programming. *Advanced Modeling and Optimization*, 1(2):9–16, 1999.

[3] Thomas Hellström and Kenneth Holmström. Parameter Tuning in Trading Algorithms using ASTA. In Y. S. Abu-Mostafa, B. LeBaron, A. W. Lo, and A. S. Weigend, editors, *Computational Finance – Proceedings of the Sixth International Conference, Leonard N. Stern School of Business, January 1999*, Cambridge, MA, 1999. MIT Press.

[4] Kenneth Holmström. The TOMLAB Optimization Environment in Matlab. *Advanced Modeling and Optimization*, 1(1):47–69, 1999.

[5] Kenneth Holmström and Mattias Björkman. The TOMLAB NLPLIB Toolbox for Nonlinear Programming. *Advanced Modeling and Optimization*, 1(1):70–86, 1999.

[6] Kenneth Holmström, Mattias Björkman, and Erik Dotzauer. The TOMLAB OPERA Toolbox for Linear and Discrete Optimization. *Advanced Modeling and Optimization*, 1(2):1–8, 1999.

[7] Kenneth Holmström, Mattias Björkman, and Erik Dotzauer. TOMLAB v1.0 User's Guide. Technical Report IMa-TOM-1999-01, Department of Mathematics and Physics, Mälardalen University, Sweden, 1999.

[8] Reiner Horst and Panos M. Pardalos. *Handbook of Global Optimization.* Kluwer Academic Publishers, Dordrecht Boston London, 1995.

[9] D. R. Jones, C. D. Perttunen, and B. E. Stuckman. Lipschitzian optimization without the Lipschitz constant. *Journal of Optimization Theory and Applications*, 79(1):157–181, October 1993.

[10] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry.* Springer-Verlag, New York, 1985.

[11] Yong Yao. Dynamic tunneling algorithm for global optimization. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(5):1222–1230, 1989.