Mathematical Programming 58 (1993) 179-199
North-Holland

A deterministic algorithm for global optimization

Leo Breiman*
University of California, Berkeley, CA, USA

Adele Cutler
Utah State University, Logan, UT, USA

Received 4 December 1989
Revised manuscript received 14 January 1991

We present an algorithm for finding the global maximum of a multimodal, multivariate function for which derivatives are available. The algorithm assumes a bound on the second derivatives of the function and uses this to construct an upper envelope. Successive function evaluations lower this envelope until the value of the global maximum is known to the required degree of accuracy. The algorithm has been implemented in RATFOR and execution times for standard test functions are presented at the end of the paper.

Key words: Deterministic global optimization, global maximum, global minimum.

1. Introduction

The problem considered is that of finding f_*, the unconstrained maxima of a differentiable function $f: \boldsymbol{X} \to \mathbb{R}^1$ where $\boldsymbol{X} \subset \mathbb{R}^m$ is a compact polytope.

We develop a space-covering deterministic algorithm similar to that studied by Shubert [15] and Mladineo [10], who assume that the objective function satisfies a Lipschitz condition. Instead we assume that the function satisfies the condition

$$f(\boldsymbol{x}) \leqslant f(\boldsymbol{y}) + \nabla f(\boldsymbol{y})'(\boldsymbol{x}-\boldsymbol{y}) + K\|\boldsymbol{x}-\boldsymbol{y}\|^2 \quad \text{for any } \boldsymbol{x}, \boldsymbol{y} \in \boldsymbol{X}, \tag{1}$$

where K is a known constant. In more than one dimension, this condition leads to simpler geometric properties than the Lipschitz bound.

The idea of the algorithm is to construct an upper envelope for the function using (1). The maximum of the upper envelope is easily found. Then f is evaluated at the maximizer, which reduces the maximum of the upper envelope. As the number of function evaluations increases, the maximum of the upper envelope converges to f_*.

Correspondence to: Prof. Adele Cutler, Department of Mathematics and Statistics, Utah State University, Logan, UT 84322-3900, USA.

* Partially supported by NSF DMS-8718362.

A conceptual description of the algorithm is given in Section 2, illustrated by a 1-dimensional example. We find that the algorithm concentrates on the global maximum of the function and pays very little attention to local maxima. The third section extends the 1-dimensional algorithm to higher dimensions, developing the geometrical results used in implementing the algorithm. A detailed description of the algorithm is given in Section 4.

In Section 5 we evaluate the performance of the algorithm, and compare it to published results for other optimization methods using standard test functions. Some statistical examples are also included.

The results are mixed. However, given sufficient workspace, the present algorithm is guaranteed to find the global optimum. Many of the published results for other algorithms are vague about the proportion of times their algorithm failed to find the global maximum. In addition, we have found that combining the present algorithm with an occasional local search leads to a drastic reduction in computing time (see [4]). This modification will be discussed in a future paper.

2. Background

Piyavskii [11] described a general approach to global optimization requiring continuous functions h_i defined on $\boldsymbol{X}$ such that for any $\boldsymbol{x}_i \in \boldsymbol{X}$,

$$f(\boldsymbol{x}) \leq h_i(\boldsymbol{x}) \quad \text{for all } \boldsymbol{x} \in \boldsymbol{X},$$

$$f(\boldsymbol{x}_i) = h_i(\boldsymbol{x}_i).$$

Suppose the function has already been evaluated at $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$. The *upper envelope at the nth stage* is defined to be

$$\boldsymbol{B}^{(n)}(\boldsymbol{x}) = \min_{i=1,\ldots,n} h_i(\boldsymbol{x}). \tag{2}$$

Now by assumption, $f(\boldsymbol{x}) \leq h_i(\boldsymbol{x})$, for any i, so

$$\begin{aligned} f_* &= \max_{\boldsymbol{x} \in \boldsymbol{X}} f(\boldsymbol{x}) \\ &\leq \max_{\boldsymbol{x} \in \boldsymbol{X}} \left(\min_{i=1,\ldots,n} h_i(\boldsymbol{x}) \right) \\ &= \max_{\boldsymbol{x} \in \boldsymbol{X}} \boldsymbol{B}^{(n)}(\boldsymbol{x}). \end{aligned}$$

Thus, a lower bound $\underline{f}_n$ and an upper bound $\bar{f}_n$ for f_* are given by

$$\underline{f}_n = \max_{i=1,\ldots,n} f(\boldsymbol{x}_i) \leq f_* \leq \max_{\boldsymbol{x} \in \boldsymbol{X}} \boldsymbol{B}^{(n)}(\boldsymbol{x}) = \bar{f}_n.$$

The function is next evaluated at $\boldsymbol{x}_{n+1}$, where

$$\boldsymbol{x}_{n+1} = \arg\max_{\boldsymbol{x} \in \boldsymbol{X}} \boldsymbol{B}^{(n)}(\boldsymbol{x}).$$

Piyavskii [11] proved that $\bar{f}_n - \underline{f}_n$ converges to zero and $\underline{f}_n$ converges to f_*.

It should be noted that the problem of maximizing $\boldsymbol{B}^{(n)}$ and updating $\boldsymbol{B}^{(n)}$ to $\boldsymbol{B}^{(n+1)}$ is potentially as difficult as the original problem of maximizing f. Shubert [15] and Mladineo [10] assume that for all x, y,

$$f(x) \leq f(y) + K\|x - y\|,$$

for a known constant K, leading to the bounding functions

$$h_i(x) = f(x_i) + K\|x - x_i\|.$$

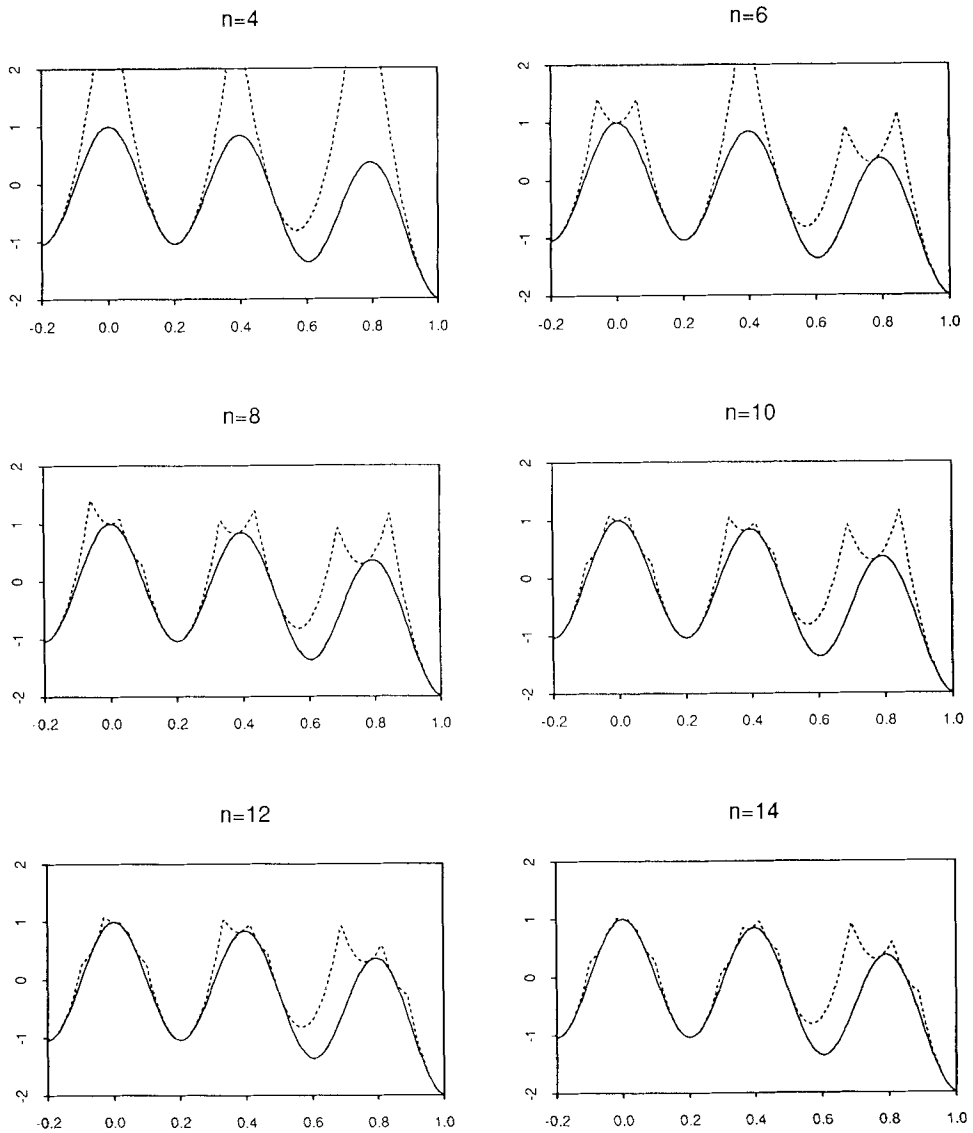

Fig. 1. Lowering of upper envelope, one-dimensional example. —— example function; – – – upper envelope.

Mladineo [10] treated the multidimensional case. The analytic difficulties involved forced her to use approximations of unknown accuracy in evaluating $\boldsymbol{x}_{n+1}$ and updating $\boldsymbol{B}^{(n)}$.

Under assumption (1) we use the bounding functions

$$h_i(\boldsymbol{x}) = f(\boldsymbol{x}_i) + \nabla f(\boldsymbol{x}_i)'(\boldsymbol{x} - \boldsymbol{x}_i) + K\|\boldsymbol{x} - \boldsymbol{x}_i\|^2.$$

Surprisingly, this leads to simple and exact evaluations of $\boldsymbol{x}_{n+1}$ and updates for $\boldsymbol{B}^{(n)}$.

2.1. *One-dimensional illustration*

Let $f(x) = \cos(5\pi x) - x^2$, with $\boldsymbol{X} = [-0.2, 1]$. The global maximum occurs at $x = 0$ and there are two local maxima, one at $x = 0.4$ and the other at $x = 0.8$. We used $K = 12.5\pi^2 - 1$.

If the function has been evaluated at $x_1, \ldots, x_n$, the upper envelope $\boldsymbol{B}^{(n)}$ consists of n quadratic pieces intersecting at local maxima or cusps. One intersection occurs between each pair of adjacent function evaluations. The function is evaluated at x_{n+1}, the largest intersection peak, introducing a new quadratic which intersects $\boldsymbol{B}^{(n)}$ in two places, one on each side of x_{n+1}.

The algorithm's progress in lowering the upper envelope has been plotted in Figure 1. We note that, unlike almost all of its competitors, this algorithm pays little attention to the local maxima of f.

3. The general algorithm

Implementation of the algorithm involves finding $\boldsymbol{x}_{n+1}$, computing $f(\boldsymbol{x}_{n+1})$, and updating $\boldsymbol{B}^{(n)}$. In this section we show that the geometry that made this feasible in the one-dimensional case can be extended to higher dimensions. We begin with a simple illustration, then present the theoretical framework.

Figures 2, 3 and 4 illustrate finding $\boldsymbol{x}_{21}$ and updating $\boldsymbol{B}^{(20)}$ to $\boldsymbol{B}^{(21)}$ for a two-dimensional example. After 20 function evaluations, $\boldsymbol{X}$ has been split into polytopes as shown in Figure 2. The polytope around $\boldsymbol{x}_i$, call it $\boldsymbol{C}_i^{(20)}$, is the set of all $\boldsymbol{x}$ for which $\boldsymbol{B}^{(20)}(\boldsymbol{x}) = h_i(\boldsymbol{x})$. We show that a local maximum of $\boldsymbol{B}^{(n)}$ can only occur at a vertex of some such polytope, so we can find $\boldsymbol{x}_{21}$ by searching for the vertex with the largest value of $\boldsymbol{B}^{(20)}(\boldsymbol{x})$. This is labeled $\boldsymbol{x}_{21}$ in Figure 2.

After evaluating $f(\boldsymbol{x}_{21})$, the polytope structure is updated. The new polytope, $\boldsymbol{C}_i^{(21)}$, is drawn with dashed lines in Figure 2. The vertices inside $\boldsymbol{C}_i^{(21)}$, which we refer to as *dead* vertices, must be removed. Those of $\boldsymbol{C}_i^{(21)}$, which we call *new* vertices, must be located. This gives the updated structure of Figure 4.

To update efficiently, selected vertex information must be stored. For each vertex, we store $\boldsymbol{v}$ and $\boldsymbol{B}^{(n)}(\boldsymbol{v})$ and pointers to vertices on the same edge of some polytope, which we call *adjacent* vertices. We also store pointers to the polytopes containing $\boldsymbol{v}$, called the *index set* of the vertex. These are given in Figure 3.

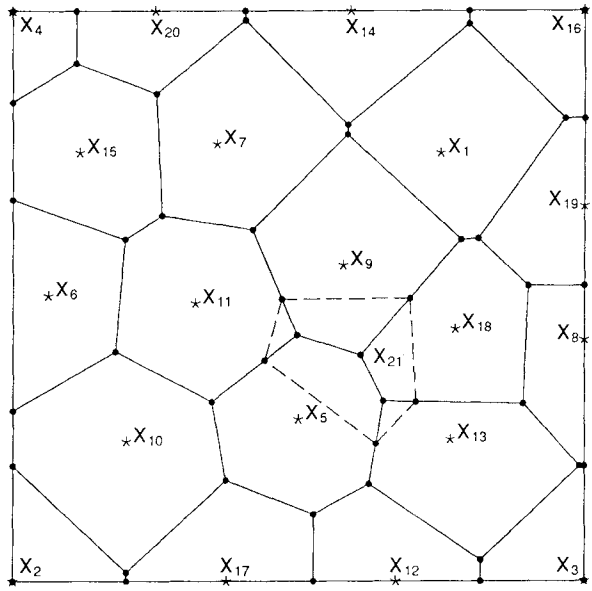

Fig. 2. Vertex structure, $n = 20$.

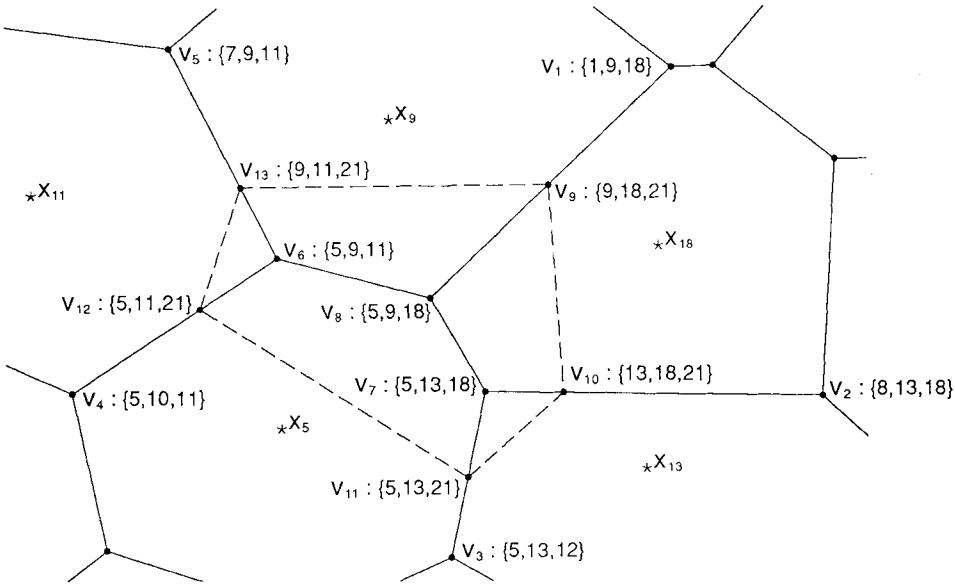

Fig. 3. New polytope $C_{21}^{(21)}$.

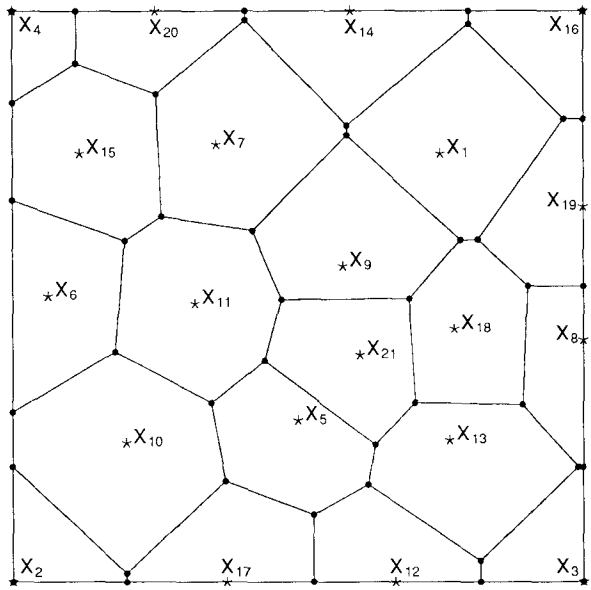

Fig. 4. Vertex structure, $n = 21$.

The dead vertices are those for which $h_{21}(\boldsymbol{v}) < \boldsymbol{B}^{(20)}(\boldsymbol{v})$. These are connected via edges, so they can be located without computing h_{21} for every vertex. Furthermore, each new vertex occurs on an edge between two adjacent (parent) vertices, one of which is dead. This is seen in Figure 3. The location of the new vertices can be found from the parent information. The index set of a new vertex can be obtained by taking the two indices common to both parents and adding the new index "21" to the set. These index sets are then used to determine the adjacencies.

The rest of this section contains more precise formulations of the geometry, culminating in a detailed description of the algorithm.

3.1. Definitions and notation

Assume $\boldsymbol{X}$ is a compact subset of $\mathbb{R}^m$ and can be written $\boldsymbol{X} = \{\boldsymbol{x} \in \mathbb{R}^m : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}\}$, for some $\boldsymbol{A}$, $\boldsymbol{b}$. Let $\boldsymbol{a}_j'$ be the jth row of $\boldsymbol{A}$, and b_j be the jth element of $\boldsymbol{b}$. With h_i defined as above, let $\boldsymbol{C}_i^{(n)}$ be the set of points at which the ith upper bound function is the smallest;

$$\boldsymbol{C}_i^{(n)} = \{\boldsymbol{x} \in \boldsymbol{X} : h_i(\boldsymbol{x}) \leq h_j(\boldsymbol{x}), j = 1, \ldots, n\},$$

and note that any $\boldsymbol{x} \in \boldsymbol{X}$ is a member of at least one $\boldsymbol{C}_i^{(n)}$. If we let

$$s_i = f(\boldsymbol{x}_i) - \boldsymbol{x}_i' \nabla f(\boldsymbol{x}_i) + K \|\boldsymbol{x}_i\|^2,$$

$$\boldsymbol{t}_i = \nabla f(\boldsymbol{x}_i) - 2K\boldsymbol{x}_i,$$

we have $h_i(\boldsymbol{x}) = s_i + \boldsymbol{t}_i'\boldsymbol{x} + K\|\boldsymbol{x}\|^2$, so $h_i(\boldsymbol{x}) = h_j(\boldsymbol{x})$ iff $(\boldsymbol{t}_i' - \boldsymbol{t}_j')\boldsymbol{x} = s_j - s_i$. Letting $\boldsymbol{A}_i^{(n)} = (\boldsymbol{A}', \boldsymbol{t}_i - \boldsymbol{t}_1, \boldsymbol{t}_i - \boldsymbol{t}_2, \ldots, \boldsymbol{t}_i - \boldsymbol{t}_n)'$ and $\boldsymbol{b}_i^{(n)} = (\boldsymbol{b}', s_1 - s_i, s_2 - s_i, \ldots, s_n - s_i)'$ we have $\boldsymbol{C}_i^{(n)} = \{\boldsymbol{x} \in \mathbb{R}^m : \boldsymbol{A}_i^{(n)}\boldsymbol{x} \leq \boldsymbol{b}_i^{(n)}\}$ which gives the result below.

Proposition 3.1. *For $i = 1, \ldots, n$, $\boldsymbol{C}_i^{(n)}$ is a polytope.* □

Definition 3.1 (vertex, vertex set, boundary vertex). A vertex of $\boldsymbol{C}_i^{(n)}$ is an element of $\boldsymbol{C}_i^{(n)}$ which satisfies m linearly independent constraints from the collection $\boldsymbol{A}_i^{(n)}\boldsymbol{x} = \boldsymbol{b}_i^{(n)}$. Let $\boldsymbol{V}^{(n)}$ be the set of all such vertices

$$\boldsymbol{V}^{(n)} = \{\boldsymbol{v}: \boldsymbol{v} \text{ is a vertex of } \boldsymbol{C}_i^{(n)} \text{ for some } i\}.$$

The constraints $\boldsymbol{Ax} = \boldsymbol{b}$ will be called boundary constraints, the others functional constraints since they have the form $h_i(\boldsymbol{x}) = h_j(\boldsymbol{x})$ for some j. A vertex for which the m linearly independent constraints are all boundary constraints will be called a boundary vertex.

Linear independence assumption. Assume that for every n, and for $i = 1, \ldots, n$, any $\boldsymbol{x} \in \boldsymbol{C}_i^{(n)}$ satisfies at most m constraints from the collection $\boldsymbol{A}_i^{(n)}\boldsymbol{x} = \boldsymbol{b}_i^{(n)}$. In particular, any vertex satisfies exactly m constraints.

This assumption will hold provided each boundary vertex satisfies exactly m linearly independent constraints from $\boldsymbol{Ax} = \boldsymbol{b}$, and for each n, $h_{n+1}(\boldsymbol{v}) \neq \boldsymbol{B}^{(n)}(\boldsymbol{v})$ for every $\boldsymbol{v} \in \boldsymbol{V}^{(n)}$. This holds almost invariably in practice.

Definition 3.2 (index set). For $\boldsymbol{x} \in \boldsymbol{X}$ define an index set

$$\boldsymbol{I}^{(n)}(\boldsymbol{x}) = \{-j: \boldsymbol{a}_j'\boldsymbol{x} = b_j\} \cup \{j: h_j(\boldsymbol{x}) = B^{(n)}(\boldsymbol{x})\}.$$

The negative elements of the index set refer to the boundary constraints satisfied by $\boldsymbol{x}$, so they never change. Each positive element j refers to a polytope $\boldsymbol{C}_j^{(n)}$ of which $\boldsymbol{x}$ is a member. These will change whenever a new polytope is created which contains $\boldsymbol{x}$. The following lemma gives a useful characterization of a vertex.

Lemma 3.1. *Let $\boldsymbol{x} \in \boldsymbol{X}$. Then $\boldsymbol{x} \in \boldsymbol{V}^{(n)}$ iff $\boldsymbol{I}^{(n)}(\boldsymbol{x})$ contains exactly $m+1$ elements.*

Proof. If $\boldsymbol{v} \in \boldsymbol{V}^{(n)}$ then $\boldsymbol{v} \in \boldsymbol{C}_i^{(n)}$ for some i, so $i \in \boldsymbol{I}^{(n)}(\boldsymbol{v})$ and $\boldsymbol{v}$ satisfies m linearly independent constraints from the collection $\boldsymbol{A}_i^{(n)}\boldsymbol{x} = \boldsymbol{b}_i^{(n)}$. Each constraint contributes one element to the index set, giving a total of $m+1$ elements. Conversely, $\boldsymbol{x}$ must be in $\boldsymbol{C}_i^{(n)}$ for some i since the polytopes cover $\boldsymbol{X}$. There are m other elements of $\boldsymbol{I}^{(n)}(\boldsymbol{x})$. Each negative element represents a boundary constraint satisfied by $\boldsymbol{x}$, each positive one a functional constraint, so by the linear independence assumption $\boldsymbol{x}$ is a vertex of $\boldsymbol{C}_i^{(n)}$. □

Lemma 3.2. *If $\boldsymbol{v}$ is a vertex and $\boldsymbol{v} \in \boldsymbol{C}_i^{(n)}$, then $\boldsymbol{v}$ is a vertex of $\boldsymbol{C}_i^{(n)}$.*

Proof. If $\boldsymbol{v} \in \boldsymbol{C}_i^{(n)}$ then $i \in \boldsymbol{I}^{(n)}(\boldsymbol{v})$. The m other elements of $\boldsymbol{I}^{(n)}(\boldsymbol{v})$ correspond to m constraints from the collection $\boldsymbol{A}_i^{(n)}\boldsymbol{x} = \boldsymbol{b}_i^{(n)}$ that are satisfied by $\boldsymbol{v}$ so by the linear independence assumption $\boldsymbol{v}$ is a vertex of $\boldsymbol{C}_i^{(n)}$. □

Lemma 3.3. *For any $v \in V^{(n)}$ and for any $i > 0$, v is a vertex of $C_i^{(n)}$ iff $i \in I^{(n)}(v)$.*

Proof. If v is a vertex of $C_i^{(n)}$, then $h_i(v) = B^{(n)}(v)$ so $i \in I^{(n)}(v)$. Conversely, if $i \in I^{(n)}(v)$ then $h_i(v) = B^{(n)}(v)$ so $v \in C_i^{(n)}$ and by Lemma 3.2, v is a vertex of $C_i^{(n)}$. □

Definition 3.3 (adjacent, adjacency list). Let $v_j, v_k \in V^{(n)}$. Then v_j and v_k are adjacent or neighbors if they are on the same edge of some polytope $C_i^{(n)}$, that is, if for some i, there are $m-1$ linearly independent constraints from the collection $A_i^{(n)}x = b_i^{(n)}$ that are satisfied by both v_j and v_k. Let

$$\mathrm{Adj}^{(n)}(v_k) = \{j\colon v_j \text{ is adjacent to } v_k\}.$$

Lemma 3.4. *Two vertices $v_j, v_k \in V^{(n)}$ are adjacent iff $I^{(n)}(v_j) \cap I^{(n)}(v_k)$ contains exactly m elements.*

Proof. If v_j and v_k are adjacent, there are $m-1$ linearly independent constraints from the collection $A_i^{(n)}x = b_i^{(n)}$ that are satisfied by both v_j and v_k. These constraints give m indices that must be in *both* $I^{(n)}(v_j)$ and $I^{(n)}(v_k)$. Conversely, note that $I^{(n)}(v_j) \cap I^{(n)}(v_k)$ must contain some $i > 0$ or the two vertices would satisfy the same m linearly independent boundary constraints, and hence be identical. Then by definition of the index sets we can find $m-1$ constraints from $A_i^{(n)}x = b_i^{(n)}$ which are satisfied by both v_j and v_k, so they are adjacent. □

Lemma 3.5. *If $v_j, v_k \in V^{(n)}$ are adjacent vertices, and v is a vertex for which $I^{(n)}(v_j) \cap I^{(n)}(v_k) \subset I^{(n)}(v)$ then either $v = v_j$ or $v = v_k$.*

Proof. Note that reasoning as in the last proof, $I^{(n)}(v_j) \cap I^{(n)}(v_k)$ must contain some $i > 0$. But then if the vertices were distinct, $C_i^{(n)}$ would have three vertices on the same edge, which is not possible since $C_i^{(n)}$ is a polytope. □

Lemma 3.6. *Let $v \in V^{(n)}$. Then if v is a boundary vertex, it is adjacent to exactly m vertices and if v is not a boundary vertex it is adjacent to exactly $m+1$ vertices.*

Proof. First note that there are $m+1$ subsets of size m from $I^{(n)}(v)$, so by Lemma 3.5 there are at most $m+1$ adjacent vertices. If v is not a boundary vertex, each of these subsets refers to $m-1$ linearly independent constraints which, by continuity, are satisfied by some elements of X, and hence by adjacent vertices. If v is a boundary vertex, one of the subsets refers to the m linearly independent boundary constraints that are satisfied only by v, but by continuity each of the others are satisfied by some elements of X, and hence by adjacent vertices. □

3.2. Finding x_{n+1}

If we have a list of the vertices and the value of $B^{(n)}$ at each vertex, the following theorem allows us to search through the list to find the vertex v_* for which $B^{(n)}$ is

the largest, and set $x_{n+1} = v_*$. In practice it is more efficient to use a heap and update the heap after each function evaluation. This way v_* is simply the vertex at the top of the heap.

Theorem 3.1. *The global max of $B^{(n)}$ can only occur at a vertex $v \in V^{(n)}$.*

Proof. Since the $C_i^{(n)}$'s partition X and $B^{(n)}(x) = h_i(x)$ for $x \in C_i^{(n)}$, it is sufficient to show that for any i the maximum of h_i over $C_i^{(n)}$ occurs at a vertex of $C_i^{(n)}$. This follows immediately from the fact that h_i is convex and $C_i^{(n)}$ is a polytope. □

3.3. Updating the upper envelope

We first consider updating $V^{(n)}$.

Definition 3.4 (dead and alive). A vertex $v \in V^{(n)}$ is dead if $h_{n+1}(v) < B^{(n)}(v)$ and alive if $h_{n+1}(v) \geq B^{(n)}(v)$.

Theorem 3.2 (updating $V^{(n)}$). *Let the linear independence assumption hold. Then*

$$V^{(n+1)} = \{v \in V^{(n)}: v \text{ is alive}\} \cup \{v: v \text{ is a vertex of } C_{n+1}^{(n+1)}\}.$$

Proof. If v is an alive vertex of $V^{(n)}$, $v \in C_j^{(n)}$, then $h_j(v) < h_{n+1}(v)$ by the linear independence assumption. So $I^{(n+1)}(v) = I^{(n)}(v)$, and v is a vertex by Lemma 3.1. Conversely, let $v \in V^{(n+1)}$. If $h_{n+1}(v) < B^{(n)}(v)$ then $B^{(n+1)}(v) = h_{n+1}(v)$ so $n+1 \in I^{(n+1)}(v)$ and by Lemma 3.3, v is a vertex of $C_{n+1}^{(n+1)}$. Otherwise, by the linear independence assumption $h_{n+1}(v) > B^{(n)}(v) = B^{(n+1)}(v)$ so $I^{(n+1)}(v) = I^{(n)}(v)$ and so v is an alive vertex of $V^{(n)}$. □

Theorem 3.2 implies that the dead vertices must be removed. They can be located quickly using the following results.

Definition 3.5 (graph-connected). A set of vertices $V = \{v_1, \ldots, v_p\}$ is graph-connected if for any $v_{j_1}, v_{j_2} \in V$, there exist $v_{i_1}, \ldots, v_{i_p} \in V$ with $v_{j_1} = v_{i_1}$ and $v_{j_2} = v_{i_p}$ and such that v_{i_k} is adjacent to $v_{i_{k+1}}$ for all $k = 1, \ldots, p-1$.

Theorem 3.3. *The set of dead vertices in $V^{(n)}$ is graph-connected.*

Proof. Let v_1, v_2 be dead vertices and consider the line between v_1 and v_2. Since $C_{n+1}^{(n+1)}$ is convex, $h_{n+1}(x) < B^{(n)}(x)$ for every point x on this line. Suppose as we move from v_1 to v_2 the line passes through the polytopes $C_{j_1}^{(n)}, \ldots, C_{j_p}^{(n)}$ respectively. It is sufficient to show that given a dead vertex in $C_{j_k}^{(n)}$ we can reach $C_{j_{k+1}}^{(n)}$ by moving only between adjacent dead vertices.

Note that any dead vertices of $C_{j_k}^{(n)}$ are graph-connected because they are in the intersection of the polytope $C_{j_k}^{(n)}$ with the halfspace $H_{j_k} = \{x: h_{n+1}(x) \leq h_{j_k}(x)\}$. Furthermore, $C_{j_k}^{(n)} \cap C_{j_{k+1}}^{(n)} \cap H_{j_k}$ is non-empty since the line passes through it. But $C_{j_k}^{(n)} \cap C_{j_{k+1}}^{(n)}$ is a polytope and the non-empty intersection of a polytope with a halfspace contains a vertex of that polytope, so there must be a dead vertex $v \in C_{j_k}^{(n)} \cap C_{j_{k+1}}^{(n)}$ which can be reached by moving along the adjacencies of the dead vertices of $C_{j_k}^{(n)}$. □

Theorem 3.3 allows us to find the dead vertices using a stack, starting with $v_* = x_{n+1}$. A vertex is taken off the stack, and its adjacent vertices are examined. Any newly discovered dead vertices are put onto the stack. This continues until all vertices on the stack have been examined. This way h_{n+1} is evaluated only at the dead vertices and their alive neighbors. These values of h_{n+1} are used to find the vertices of $C_{n+1}^{(n+1)}$ as described in Theorem 3.4 and its corollary.

Theorem 3.4 (finding the vertices of $C_{n+1}^{(n+1)}$). *Let $v \in X$. Then v is a vertex of $C_{n+1}^{(n+1)}$ iff either*

(i) *$v \in V^{(n)}$ is a dead boundary vertex, or*

(ii) *there exist adjacent vertices $v_j, v_k \in V^{(n)}$, where v_j is dead, v_k is alive, and $\alpha \in [0, 1)$ such that*

$$v = \alpha v_j + (1-\alpha) v_k,$$

$$h_{n+1}(v) = B^{(n)}(v).$$

Proof. Suppose v is a vertex of $C_{n+1}^{(n+1)}$. If v is a boundary vertex, it has always satisfied the same $m+1$ linearly independent boundary constraints, so it is a vertex of $V^{(n)}$, and since $h_{n+1}(v) = B^{(n+1)}(v)$ it is dead. If v is not a boundary vertex, note that v must be on an edge of some polytope $C_i^{(n)}$, and therefore there exist vertices v_j and v_k on the same edge of $C_i^{(n)}$ such that

$$v = \alpha v_j + (1-\alpha) v_k \quad \text{for some } \alpha,\ 0 \leq \alpha < 1.$$

Now note that $\{x: h_{n+1}(x) \leq h_i(x)\}$ is a halfspace and the edge between v_j and v_k is a line segment. These intersect at v, so one endpoint of the line segment is in the halfpsace, the other is not, and hence one of the adjacent pair is dead, the other alive.

Conversely, suppose (i) holds. Since v is a boundary vertex in $V^{(n)}$, it satisfies m linearly independent boundary constraints, so it is still a vertex. Since v is dead, $h_{n+1}(v) = B^{(n+1)}(v)$ so v must be a vertex of $C_{n+1}^{(n+1)}$. Now suppose (ii) holds. Since v_j and v_k are adjacent, they are on the same edge of some polytope $C_i^{(n)}$. But $h_i(v_j) > h_{n+1}(v_j)$ since v_j is dead, and $h_i(v_k) \leq h_{n+1}(v_k)$ since v_k is alive, so by continuity of h_{n+1} and h_i, there is a point v on the line between v_j and v_k for which $h_{n+1}(v) = h_i(v)$. This point is a vertex since it satisfies the $m-1$ constraints satisfied by v_j and v_k as well as this additional constraint, which must be linearly independent of the others or v_j and v_k would either both be alive or both dead. □

Corollary. *Let* $\boldsymbol{v}$, $\boldsymbol{v}_j$ *and* $\boldsymbol{v}_k$ *be as in Theorem* 3.4(ii). *Then*

$$\alpha = \frac{h_{n+1}(\boldsymbol{v}_k) - \boldsymbol{B}^{(n)}(\boldsymbol{v}_k)}{(h_{n+1}(\boldsymbol{v}_k) - \boldsymbol{B}^{(n)}(\boldsymbol{v}_k)) - (h_{n+1}(\boldsymbol{v}_j) - \boldsymbol{B}^{(n)}(\boldsymbol{v}_j))}.$$

Proof. Since $\boldsymbol{v}_j$, $\boldsymbol{v}_k$, and $\boldsymbol{v}$ are on the same edge of some polytope $\boldsymbol{C}_i^{(n)}$, $h_i(\boldsymbol{v}_j) = \boldsymbol{B}^{(n)}(\boldsymbol{v}_j)$, $h_i(\boldsymbol{v}_k) = \boldsymbol{B}^{(n)}(\boldsymbol{v}_k)$ and $h_i(\boldsymbol{v}) = \boldsymbol{B}^{(n)}(\boldsymbol{v})$. But by the theorem, $h_{n+1}(\boldsymbol{v}) = \boldsymbol{B}^{(n)}(\boldsymbol{v})$ so $h_i(\boldsymbol{v}) = h_{n+1}(\boldsymbol{v})$. Substituting for $\boldsymbol{v}$ in terms of $\boldsymbol{v}_j$ and $\boldsymbol{v}_k$ and simplifying gives the result. □

Theorem 3.4 and its corollary allow us to find the vertices of $\boldsymbol{C}_{n+1}^{(n+1)}$ easily because the relevant values of h_{n+1} were computed and saved when finding the dead vertices.

We also need to find the values of $\boldsymbol{B}^{(n+1)}(\boldsymbol{v})$ for each $\boldsymbol{v} \in \boldsymbol{V}^{(n+1)}$. For the alive vertices, this is just $\boldsymbol{B}^{(n)}(\boldsymbol{v})$. For the dead boundary vertices, this is $h_{n+1}(\boldsymbol{v})$, which has been saved. The value must be computed for the vertices of $\boldsymbol{C}_{n+1}^{(n+1)}$.

Proposition 3.2 (updating index sets). *Let* $\boldsymbol{v} \in \boldsymbol{V}^{(n+1)}$.

(i) *If* $\boldsymbol{v} \in \boldsymbol{V}^{(n)}$ *and* $\boldsymbol{v}$ *is alive then* $\boldsymbol{I}^{(n+1)}(\boldsymbol{v}) = \boldsymbol{I}^{(n)}(\boldsymbol{v})$.

(ii) *If* $\boldsymbol{v}$ *is a new vertex formed between* $\boldsymbol{v}_j$ *and* $\boldsymbol{v}_k$ (*see Theorem* 3.4) *then* $\boldsymbol{I}^{(n+1)}(\boldsymbol{v}) = (\boldsymbol{I}^{(n)}(\boldsymbol{v}_j) \cap \boldsymbol{I}^{(n)}(\boldsymbol{v}_k)) \cup \{n+1\}$.

(iii) *If* $\boldsymbol{v}$ *is a boundary vertex of* $\boldsymbol{C}_{n+1}^{(n+1)}$ *then* $\boldsymbol{I}^{(n+1)}(\boldsymbol{v})$ *can be obtained from* $\boldsymbol{I}^{(n)}(\boldsymbol{v})$ *by replacing the only positive index by the value* $n+1$.

Proof. (i) If $\boldsymbol{v}$ is an alive vertex of $\boldsymbol{V}^{(n)}$, $h_{n+1}(\boldsymbol{v}) \geq \boldsymbol{B}^{(n)}(\boldsymbol{v})$ so $\boldsymbol{I}^{(n)}(\boldsymbol{v}) \subset \boldsymbol{I}^{(n+1)}(\boldsymbol{v})$, and by Lemma 3.1 they are identical.

(ii) Since $\boldsymbol{v}$ is between $\boldsymbol{v}_j$ and $\boldsymbol{v}_k$ and the constraints are linear, any element of $\boldsymbol{I}^{(n)}(\boldsymbol{v}_j) \cap \boldsymbol{I}^{(n)}(\boldsymbol{v}_k)$ will be in $\boldsymbol{I}^{(n+1)}(\boldsymbol{v})$. Since $\boldsymbol{B}^{(n+1)}(\boldsymbol{v}) = h_{n+1}(\boldsymbol{v})$, $n+1$ will also be in $\boldsymbol{I}^{(n+1)}(\boldsymbol{v})$. This gives a total of $m+1$ elements so the result holds by Lemma 3.1.

(iii) Let $\boldsymbol{v}$ be a boundary vertex of $\boldsymbol{C}_{n+1}^{(n+1)}$. The m negative elements of $\boldsymbol{I}^{(n)}(\boldsymbol{v})$ never change, so they will be in $\boldsymbol{I}^{(n+1)}(\boldsymbol{v})$. Now $\boldsymbol{B}^{(n+1)}(\boldsymbol{v}) = h_{n+1}(\boldsymbol{v})$ so $n+1 \in \boldsymbol{I}^{(n+1)}(\boldsymbol{v})$ which gives the required $m+1$ elements. □

We finally turn to the problem of determining adjacencies.

Proposition 3.3 (updating adjacencies). *Let* $\boldsymbol{v} \in \boldsymbol{V}^{(n+1)}$.

(i) *If* $\boldsymbol{v}$ *is an alive vertex from* $\boldsymbol{V}^{(n)}$ *then* $\mathrm{Adj}^{(n+1)}(\boldsymbol{v})$ *can be obtained from* $\mathrm{Adj}^{(n)}(\boldsymbol{v})$ *by replacing any index referring to a dead vertex by the index for the resulting new vertex* (*Theorem* 3.4).

(ii) *If* $\boldsymbol{v}$ *is a new vertex formed between* $\boldsymbol{v}_j$ *and* $\boldsymbol{v}_k$ (*see Theorem* 3.4) *then* $\boldsymbol{v}$ *is adjacent only to* $\boldsymbol{v}_k$ *and to other vertices of* $\boldsymbol{C}_{n+1}^{(n+1)}$.

(iii) *If* $\boldsymbol{v}$ *is a boundary vertex of* $\boldsymbol{C}_{n+1}^{(n+1)}$ *then* $\boldsymbol{v}$ *is adjacent only to other vertices of* $\boldsymbol{C}_{n+1}^{(n+1)}$.

Proof. (i) $I^{(n+1)}(v) = I^{(n)}(v)$ by Proposition 3.2. The index set of any alive adjacent vertex will also have stayed the same, so the vertex will still be adjacent. Any dead adjacent vertex will have given rise to some new vertex, which by Proposition 3.2 shares m of the indices of $I^{(n)}(v) = I^{(n+1)}(v)$, so is adjacent to v.

(ii) By (i), the vertex must be adjacent to the alive vertex from which it was created. Now $n+1 \in I^{(n+1)}(v)$ by Proposition 3.2, so the only m-element subset of $I^{(n+1)}(v)$ that does not contain $n+1$ is the subset shared by the alive vertex. All other subsets contain $n+1$, so all other adjacent vertices are vertices of $C_{n+1}^{(n+1)}$.

(iii) Suppose v is adjacent to v_i, so they are on the same edge of some polytope. By Proposition 3.2, $n+1$ is the only positive element of $I^{(n+1)}(v)$, so they must both be vertices of $C_{n+1}^{(n+1)}$. □

Proposition 3.4 (adjacencies of vertices of $C_{n+1}^{(n+1)}$). *Let v_1, v_2 be distinct vertices of $C_{n+1}^{(n+1)}$. If v_1 is a boundary vertex, set $v_{j_1} = v_1$. Otherwise, let v_{j_1} be the dead vertex which gave rise to v_1 (see Theorem 3.4). Let d_1 be the index which is in $I^{(n)}(v_{j_1})$ but not in $I^{(n+1)}(v_1)$. Similarly define v_{j_2} and d_2. Then the following hold.*

(i) *If $v_{j_1} = v_{j_2}$ then v_1 and v_2 are adjacent.*

(ii) *If v_{j_1} and v_{j_2} are adjacent, v_1 and v_2 are adjacent iff $d_1 = d_2$.*

(iii) *If $I^{(n)}(v_{j_1}) \cap I^{(n)}(v_{j_2})$ contains $m-1$ elements, v_1 and v_2 are adjacent iff $d_1 \notin I^{(n)}(v_{j_2})$ and $d_2 \notin I^{(n)}(v_{j_1})$.*

(iv) *If $I^{(n)}(v_{j_1}) \cap I^{(n)}(v_{j_2})$ contains fewer than $m-1$ elements, v_1 and v_2 are not adjacent.*

Proof. (i) Let $v_{j_1} = v_{j_2} = v_j$. Since $I^{(n+1)}(v_1)$ contains every element of $I^{(n)}(v_j)$ except d_1 and $I^{(n+1)}(v_2)$ contains every element but d_2, each must contain the same $m-1$ element subset. But they also contain the element $n+1$, giving m indices in common, so v_1 and v_2 are adjacent.

(ii) Since v_{j_1} and v_{j_2} are adjacent, we can write $I^{(n)}(v_{j_1}) = \{b_1, a_1, a_2, \ldots, a_m\}$ and $I^{(n)}(v_{j_2}) = \{b_2, a_1, a_2, \ldots, a_m\}$, with $b_1 \neq b_2$. First note that no new vertex can occur on the edge between v_{j_1} and v_{j_2} since both of these vertices are dead, and clearly no boundary vertex can occur on an edge. So $b_1 \neq d_1$ and $b_2 \neq d_2$. Say $d_1 = a_1$. Then if $d_1 = d_2$, $I^{(n+1)}(v_1) \cap I^{(n+1)}(v_2) = \{n+1, a_2, \ldots, a_m\}$ so v_1 and v_2 are adjacent, and if $d_2 \neq d_1$, the intersection contains one less element, so v_1 and v_2 are not adjacent.

(iii) Let $I^{(n)}(v_{j_1}) = \{c_1, b_1, a_1, a_2, \ldots, a_{m-1}\}$ and $I^{(n)}(v_{j_2}) = \{c_2, b_2, a_1, a_2, \ldots, a_{m-1}\}$ with $\{c_1, b_1\} \cap \{c_2, b_2\} = \emptyset$. Now if d_1 is one of the a_i's, $I^{(n+1)}(v_1) \cap I^{(n+1)}(v_2)$ will contain at most $m-1$ elements, namely the other a_i's and $n+1$. Similarly for d_2. So v_1 and v_2 can only be adjacent if neither d_1 nor d_2 are in $\{a_1, a_2, \ldots, a_{m-1}\}$. Conversely, if $d_1 \in \{c_1, b_1\}$ and $d_2 \in \{c_2, b_2\}$, $I^{(n+1)}(v_{i_1}) \cap I^{(n+1)}(v_{i_2}) = \{n+1, a_1, a_2, \ldots, a_{m-1}\}$, which has m elements, so v_{i_1} and v_{i_2} are adjacent.

(iv) If $I^{(n)}(v_{j_1}) \cap I^{(n)}(v_{j_2})$ contains fewer than $m-1$ elements, then $I^{(n+1)}(v_1) \cap I^{(n+1)}(v_2)$ contains at most one more element, namely $n+1$, so v_1 and v_2 cannot be adjacent. □

This result allows us to update the index sets easily provided we save the appropriate index d_i for each new vertex $\boldsymbol{v}_i$. To make the task easier, the elements of each index set are stored in increasing order.

4. Detailed description of the algorithm

The algorithm developed in Section 3 is now presented in more detail. Programming details (e.g. organization of the heap, management of arrays to allow new vertices to occupy the places of dead vertices, etc.) are suppressed in the following description.

The parameters m, *maxit*, K, ε_1, ε_2, $\boldsymbol{x}_1$, and $\boldsymbol{a}_j$, b_j for $j=1,\ldots,$ nc (where nc is the number of boundary constraints) must be specified. The user must also supply the vertices of the domain $\boldsymbol{X}$, and the index sets of these vertices, with each index set in increasing order, the last element equal to 1. Let nv be the number of boundary vertices.

It is necessary to store $\boldsymbol{x}_*$ (the current estimated position of the global maximum), $\underline{f}$, $\bar{f}$ and $f_{\min}$, and the values of $\boldsymbol{v}_j$, $\boldsymbol{B}^{(n)}(\boldsymbol{v}_j)$, $\boldsymbol{I}^{(n)}(\boldsymbol{v}_j)$ and $\mathrm{Adj}(\boldsymbol{v}_j)$ for each vertex.

Step. 1. Initialize.

(i) Calculate $f_1=f(\boldsymbol{x}_1)$, $\nabla f_1=\nabla f(\boldsymbol{x}_1)$. Let $n=1$.

(ii) For every $k=1,\ldots,$ nv let $\boldsymbol{B}^{(1)}(\boldsymbol{v}_k)=f_1+\nabla f_1'(\boldsymbol{v}_k-\boldsymbol{x}_1)+K\|\boldsymbol{v}_k-\boldsymbol{x}_1\|^2$.

(iii) For every $k=1,\ldots,$ nv, put k in $\mathrm{Adj}(\boldsymbol{v}_k)$. For every $k,j=1,\ldots,$ nv with $k<j$, if $\boldsymbol{I}^{(1)}(\boldsymbol{v}_j)\cap\boldsymbol{I}^{(1)}(\boldsymbol{v}_k)$ has m elements, put j in $\mathrm{Adj}(\boldsymbol{v}_k)$ and k in $\mathrm{Adj}(\boldsymbol{v}_j)$.

(iv) Create a heap of the vertices.

(v) Let $\boldsymbol{x}_*=\boldsymbol{x}_1$, $\underline{f}=f_1$ and $f_{\min}=f_1$.

(vi) Let $\boldsymbol{x}_2=\boldsymbol{v}_*$, the vertex from the top of the heap, and $\bar{f}=\boldsymbol{B}^{(1)}(\boldsymbol{v}_*)$.

Step 2. Evaluate the function.

Compute $f_{n+1}=f(\boldsymbol{x}_{n+1})$, $\nabla f_{n+1}=\nabla f(\boldsymbol{x}_{n+1})$.

If $f_{n+1}>\underline{f}$ then $\boldsymbol{x}_*=\boldsymbol{x}_{n+1}$, $\underline{f}=f_{n+1}$, else $f_{\min}=\min\{f_{n+1},f_{\min}\}$.

Step 3. Update the upper envelope.

(i) (*find dead vertices*). Classify $\boldsymbol{v}_*=\boldsymbol{x}_{n+1}$ as dead and put in stack. Calculate $h_{n+1}(\boldsymbol{v}_*)$ and save.

(A) If the stack is empty, go to (ii). Otherwise, take $\boldsymbol{v}$ off the stack.

(B) Find a vertex $\boldsymbol{v}_j$ adjacent to $\boldsymbol{v}$. If $\boldsymbol{v}_j$ is already classified, go to (C), otherwise, calculate $h_{n+1}(\boldsymbol{v}_j)$ and save. If $h_{n+1}(\boldsymbol{v}_j)<\boldsymbol{B}^{(n)}(\boldsymbol{v}_j)$, classify $\boldsymbol{v}_j$ as dead and add it to the list, otherwise, classify $\boldsymbol{v}_j$ as alive.

(C) If all vertices adjacent to $\boldsymbol{v}$ have been classified, return to (A), otherwise return to (B).

(ii) (new vertices) For each dead vertex $\boldsymbol{v}_j$ and adjacent alive vertex $\boldsymbol{v}_k$ do the following.

(A) Let nv = nv + 1. Let $\boldsymbol{v}_{\mathrm{nv}}=\alpha\boldsymbol{v}_j+(1-\alpha)\boldsymbol{v}_k$, where

$$\alpha=\frac{h_{n+1}(\boldsymbol{v}_k)-\boldsymbol{B}^{(n)}(\boldsymbol{v}_k)}{(h_{n+1}(\boldsymbol{v}_k)-\boldsymbol{B}^{(n)}(\boldsymbol{v}_k))-(h_{n+1}(\boldsymbol{v}_j)-\boldsymbol{B}^{(n)}(\boldsymbol{v}_j))}.$$

(B) Let

$$\boldsymbol{I}^{(n+1)}(\boldsymbol{v}_{\mathrm{nv}}) = (\boldsymbol{I}^{(n)}(\boldsymbol{v}_j) \cap \boldsymbol{I}^{(n)}(\boldsymbol{v}_k)) \cup \{n+1\},$$

$$\boldsymbol{B}^{(n+1)}(\boldsymbol{v}_{\mathrm{nv}}) = h_{n+1}(\boldsymbol{v}_{\mathrm{nv}}).$$

(C) Replace j by nv in $\mathrm{Adj}(\boldsymbol{v}_k)$. Let $\mathrm{Adj}(\boldsymbol{v}_{\mathrm{nv}}) = \{k\}$.

(D) Update the heap.

(E) Let $d(\boldsymbol{v}_{\mathrm{nv}})$ be the element of $\boldsymbol{I}^{(n)}(\boldsymbol{v}_j)$ that was replaced by $n+1$ in step (B). Save to use in part (iv).

(iii) (*boundary vertices*) For each dead boundary vertex $\boldsymbol{v}_j$ do the following.

(A) Let $\mathrm{nv} = \mathrm{nv}+1$. Let $\boldsymbol{v}_{\mathrm{nv}} = \boldsymbol{v}_j$.

(B) Obtain $\boldsymbol{I}^{(n+1)}(\boldsymbol{v}_{\mathrm{nv}})$ from $\boldsymbol{I}^{(n)}(\boldsymbol{v}_j)$ be replacing the positive element by $n+1$. Let $B^{(n+1)}(\boldsymbol{v}_{\mathrm{nv}}) = h_{n+1}(\boldsymbol{v}_j)$.

(C) Let $\mathrm{Adj}(\boldsymbol{v}_{\mathrm{nv}}) = \{\mathrm{nv}\}$.

(D) Update the heap.

(E) Let $d(\boldsymbol{v}_{\mathrm{nv}})$ be the element of $\boldsymbol{I}^{(n)}(\boldsymbol{v}_j)$ that was replaced by $n+1$ in step (B). Save to use in part (iv).

(iv) (*update adjacencies*)

(A) For every dead vertex $\boldsymbol{v}_j$, record that all new vertices obtained from $\boldsymbol{v}_j$ in part (ii) or (iii) above are adjacent to each other.

(B) For every adjacent pair of dead vertices $\boldsymbol{v}_{j_1}$ and $\boldsymbol{v}_{j_2}$, look at every pair of new vertices created from these in part (ii) or (iii). Suppose $\boldsymbol{v}_1$ was created from $\boldsymbol{v}_{j_1}$ and $\boldsymbol{v}_2$ was created from $\boldsymbol{v}_{j_2}$. If $d(\boldsymbol{v}_1) = d(\boldsymbol{v}_2)$, then record $\boldsymbol{v}_{i_1}$ and $\boldsymbol{v}_{i_2}$ as adjacent.

(C) For every adjacent pair of dead vertices $\boldsymbol{v}_{j_1}$ and $\boldsymbol{v}_{j_2}$, determine whether $\boldsymbol{I}^{(n)}(\boldsymbol{v}_{j_1})$ and $\boldsymbol{I}^{(n)}(\boldsymbol{v}_{j_2})$ differ by two elements. If so, look at every pair of new vertices created from $\boldsymbol{v}_{j_1}$ and $\boldsymbol{v}_{j_2}$ in part (ii) or (iii). Suppose $\boldsymbol{v}_1$ was created from $\boldsymbol{v}_{j_1}$ and $\boldsymbol{v}_2$ was created from $\boldsymbol{v}_{j_2}$. Then record $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ as adjacent if neither $d(\boldsymbol{v}_1)$ nor $d(\boldsymbol{v}_2)$ is in the set $\boldsymbol{I}^{(n)}(\boldsymbol{v}_{j_1}) \cap \boldsymbol{I}^{(n)}(\boldsymbol{v}_{j_2})$.

Step 4. Find next point.

Find $\boldsymbol{x}_{n+2}$ = the vertex from the top of the heap, and let $\bar{f} = \boldsymbol{B}^{(n+1)}(\boldsymbol{x}_{n+2})$.

Step 5. Test for convergence.

If $n = maxit$ or

$$\bar{f} - \underline{f} \leqslant \varepsilon_1 \quad \text{and} \quad \frac{\bar{f} - \underline{f}}{\underline{f} - f_{\min}} \leqslant \varepsilon_2,$$

then stop: $\boldsymbol{x}_*$ is the estimate of the position of the global maximum and $\underline{f}$ is the function value at this point. Otherwise, set $n = n+1$ and return to Step 2.

5. Computational behavior

In Section 5.1 we consider the performance of the algorithm for an example function. In Section 5.2 we compare our algorithm to published results for other optimization methods using standard test functions. Some statistical examples are also included.

In Dixon and Szegö [7] the authors present the results of several global optimization algorithms for a set of test functions. They report the number of function evaluations and the cpu time measured in units of *standard time.* Standard time is the time required for 1000 evaluations of the Shekel(5) function, defined in [7]. We present results along similar lines. However, although *standard time* was introduced to allow comparison of algorithms timed on different machines, it is highly dependent on the use of a floating point accelerator. This is because the floating point accelerator speeds up the evaluation of the Shekel(5) function considerably, but programs which are less floating-point intensive are not speeded up as much. For this reason, we present *soft standard time, 68881 standard time* and *fpa standard time.* The *soft* results were obtained on a SUN 3/50 without a floating point accelerator. The *68881* results were from the same machine, using a Motorola MC68881 floating-point accelerator. The *fpa* results were obtained on a SUN 3/140 with a Sun-3 floating point accelerator. None of the programs were compiled using the fortran optimizer flags.

5.1. *Performance for a simple example*

Consider the cosine example given in the appendix,

$$f(\boldsymbol{x}) = 0.1 \sum_{i=1}^{m} \cos(5\pi x_i) - \sum_{i=1}^{m} x_i^2.$$

For this function, the program was run with $m = 1, 2, 3, 4, 5$ and 6 and $n = 100$, 200, 300 and 400 iterations (=function evaluations). It was also run for 1 iteration to give an idea of the setup time. We used $\varepsilon_2 = 10^{-6}$ and $K = 10000$ since otherwise the algorithm stopped in fewer than 400 iterations for $m = 1$. Table 1 gives the *fpa standard time* and the number of vertices for each of these situations.

Table 1
Cpu-time and number of vertices for m-dimensional COS function, n is the number of iterations

	n	dimension (m)					
		1	2	3	4	5	6
Time (*fpa standard time*)	1	0.32	0.32	0.32	0.32	0.48	0.52
	100	0.77	2.13	6.61	26.77	140.65	658.06
	200	1.45	3.61	13.87	64.84	380.32	1900.00
	300	2.00	5.48	21.29	107.42	635.16	4360.32
	400	2.58	7.31	29.40	150.00	894.84	15 590.52
Number of vertices	1	2	4	8	16	32	64
	100	101	202	522	1576	4787	14 796
	200	201	402	1103	3454	11 121	39 766
	300	301	602	1675	5388	17 691	67 304
	400	401	802	2265	7446	25 773	97 766

For $m < 6$, cpu time increases roughly in proportion to n, as does the number of vertices, so the time per vertex is roughly constant. This is an indication that our efforts to get efficient updates of $\boldsymbol{B}^{(n)}$ have largely succeeded. For $m = 6$, the amount of memory required was so great that page faults were encountered, which probably accounts for the rather sharp increase in the time per vertex.

On the other hand, comparing across dimensions, the increase in both the number of vertices and the cpu time is roughly exponential. The program requires storage of $2m+6$ integers and $m+2$ reals per vertex. In a 32 megabyte machine, virtual memory was required for 400 function evaluations for $m = 6$. But 400 function evaluations in 6 dimensions is not usually enough to get close to the global maximum for complicated functions.

Table 2 gives the results of running the algorithm to convergence for $m = 1, 2, 3$ and 4 with $\varepsilon_1 = 10^{-2}$, and $\varepsilon_2 = 10^{-4}$. The number of function evaluations and the time increases rapidly with the dimensionality of the function, but the number of local maxima is also increasing. We note that using a floating point accelerator gives very different results.

Table 2

Results of running the algorithm to convergence, COS function, standard time as defined in Dixon and Szegö [7] with floating point accelerators (*68881* and *fpa*) and without a floating-point accelerator (*soft*)

Dimension m	Function evaluations	Number of local max	*soft* std time	*68881* std time	*fpa* std time
1	19	5	0.05	0.38	0.82
2	77	25	0.53	1.06	1.83
3	327	125	7.04	13.75	23.66
4	1392	625	179.88	441.75	660.97

5.2. *Performance on test functions*

The Appendix contains definitions of the less well-known example functions. The examples we consider are the one-dimensional functions WingoA, WingoB and WingoC, the two-dimensional functions EXP2, COS2, S&H, GW, G&P, RCOS and CAM, the three-dimensional functions F&N and H3, and the four-dimensional EXP4 and COS4. The four-dimensional examples S5, S7 and S10 and the six-dimensional H6 given in Dixon and Szegö [7] were not considered because for these functions the algorithm converged slowly and required a large amount of workspace.

Table 3 contains our results. For each example, $\varepsilon_1 = 10^{-2}$ and $\varepsilon_2 = 10^{-4}$. Both *soft* and *fpa* standard time are given, the latter using a floating-point accelerator for all calculations, the former using no floating-point accelerator. As with all optimization routines, this algorithm works better for some functions than others.

Table 3

Results of upper bound algorithm for example functions

function	K	n	$f_{\max}$	*fpa* std time	*soft* std time
WingoA	0.5	16	−15.282	0.72	0.10
WingoB	1.25	21	−44.957	0.77	0.17
WingoC	3.125	391	−261.787	4.49	5.14
EXP2	0.223	24	1.000	0.95	0.14
EXP4	0.223	117	1.000	33.48	8.34
COS2	11.34	77	0.200	1.83	0.53
COS4	11.34	1392	0.400	660.97	179.88
S&H	45.35	667	95.283	19.16	18.80
GW	0.495	939	1.000	17.65	7.63
G&P	1.7e6	$10\,000^N$	−3.002	187.81	57.40
RCOS	8.56	269	−0.398	4.92	1.50
CAM	4.5	112	1.032	2.42	0.58
F&N	13.11	8657	−74.196	869.10	575.60
H3	197.1	2575	3.863	213.42	62.90

Function definitions are given in the Appendix. K is the upper bound constant, n is the number of function evaluations, $f_{\max}$ is the largest function value obtained in the n evaluations.

Wingo [20] used a derivative-free algorithm due to Brent [3] to find the global maximum for the Wingo functions. Brent's method took 71, 107 and 1048 function evaluations for the three functions. The upper bound algorithm took 16, 21 and 391 function evaluations for the same functions, but these should be multiplied by two since with each function evaluation there is a derivative evaluation. We also note that Wingo used a different stopping criterion.

Table 4 gives published results in terms of the number of function evaluations and the units of standard time. It must be noted that it is difficult to compare algorithms in this way because the algorithms employ different stopping rules. If, for example, we had selected different values of ε_1 and ε_2, we could have increased or decreased the number of function evaluations and standard time considerably. Also, for a fair comparison, the number of function evaluations for the upper bound algorithm must be multiplied by $m+1$ since we use derivatives.

In terms of function evaluations, the upper bound algorithm seems to compare reasonably well with the other algorithms for the functions GW, CAM and RCOS. It does not work well for G&P, for which the algorithm does not converge in 10 000 iterations. It is also not particularly good for H3. In terms of time, the upper bound algorithm compares favorably with the other algorithms for the RCOS function but not for any of the other functions.

At this stage, the upper bound algorithm seems more expensive than the methods available from the literature. However, all of the algorithms taken from the literature are stochastic and may fail to locate the global maximum a significant proportion of the time, whereas the upper bound algorithm locates it with certainty when the upper bound constant is known. Moreover, not all the researchers have published

Table 4
Comparison with other methods

Method	Number of function evaluations					Units of standard time				
	Function					Function				
	GW	CAM	G&P	RCOS	H3	GW	CAM	G&P	RCOS	H3
Törn [18]	—	—	2499	1558	2584	—	—	4	4	8
de Biase and Frontini [1]	—	—	378	597	732	—	—	15	14	17
Price [12]	—	—	2500	1600	2400	—	—	3	4	8
Bremermann [2]†	—	—	300	160	420[L]	—	—	0.7	0.5	2[L]
Rinnooy Kan and Timmer [13, 14]	—	—	148	206	197	—	—	0.15	0.25	0.5
Snyman and Fatti [17]	1496	178	474	—	365	1.4	0.1	0.2	—	0.6
Vanderbilt and Louie [19]	—	—	1186	557	1224	—	—	2	1	4
New method	939	112	N	269	2575	7.63	0.58	N	1.50	62.90

† Results taken from Dixon and Szegö [7].
L: the algorithm found a local maximum.
N: the algorithm did not converge in 10 000 iterations.

results on how often the algorithm finds the global maximum, so it is difficult to know how reliable some of these methods are.

Another point in favor of the upper bound algorithm is that although the overhead is high, leading to high values for standard time, the number of function evaluations is often competitive with the existing algorithms. If a function is expensive to evaluate, the overhead is less significant, and the upper bound algorithm may do better than many of its competitors. This is not evident in Table 4 since all the standard test functions are cheap to evaluate.

One of the problems with the algorithm is in the analytic computation of a good upper bound constant. A more difficult problem is that the upper bound constant may differ drastically over different subregions of $\boldsymbol{X}$ and its maximum value on $\boldsymbol{X}$ is very high, but occurs only in a small subregion not containing the global optimum. This seems to be the problem with the G&P function. On the other hand, the upper bound constant for RCOS is "uniformly fairly good" over the entire region. For some functions, much better performance would be achieved by breaking up the domain and using a different upper bound constant for the different regions. Parallel architectures make this idea particularly appealing.

We have also found (see [4]) that combining the upper bound algorithm with an occasional local search usually results in much faster convergence and a subsequent reduction in cpu-time. In fact, this combination is competitive with the best timing results for published algorithms. This modified algorithm will be the subject of a forthcoming report.

A RATFOR or FORTRAN listing of the upper bound algorithm is available from the correspondence author.

Appendix: Example functions

We present only the less well-known test functions. The Griewank function, GW, and the 6-hump camel back function, CAM, are documented in Griewank [9]. The G&P, RCOS and H3 functions are documented in Dixon and Szegö [7].

Cauchy likelihood. WingoA, WingoB, WingoC (Wingo [20]). For a given set of data $y_1 \leq y_2 \leq \cdots \leq y_n$, f is the log likelihood for the one-parameter Cauchy distribution

$$f(x) = -\sum_{i=1}^{n} [\log(\pi) + \log(1 + (y_i - x)^2)].$$

The region of interest is $y_1 \leq x \leq y_n$. We used the starting points 9.5 for WingoA, 13.0 for WingoB, 242.5 for WingoC. The values of y_i are given in Table A.1.

Table A.1
Data for Wingo functions

Wingo A	3	7	12	17						
Wingo B	2	5	7	8	11	15	17	21	23	26
Wingo C	4.1	7.7	17.5	31.4	32.7	92.4	115.3	118.3	119.0	129.6
	198.6	200.7	242.5	255.0	274.7	274.7	303.8	334.1	430.0	489.1
	703.4	978.0	1656.0	1697.8	2745.6					

Exponential. EXP. Let

$$f(x) = \exp\left(-\tfrac{1}{2}\sum_{i=1}^{m} x_i^2\right).$$

The region of interest is $-1 \leq x_i \leq 1$ for $i = 1, \ldots, m$. The starting point was $x_i = 0.2$ for $i = 1, \ldots, m$.

Cosine mixture. COS. Let

$$f(x) = 0.1 \sum_{i=1}^{m} \cos(5\pi x_i) - \sum_{i=1}^{m} x_i^2.$$

The region of interest was $-1 \leq x_i < 1$ for $i = 1, \ldots, m$. The starting point was $x_i = 0.5$ for $i = 1, \ldots, m$.

Poissonian pulse-train likelihood. S&H (Slump and Hoenders [16]). Let

$$f(x) = \sum_{i=1}^{p} (-\lambda_i(x) + \hat{n}_i \log(\lambda_i(x)) - \log(\hat{n}_i!)),$$

where

$$\lambda_i(\boldsymbol{x}) = 2\left[1 + 2.5\exp\left\{-\frac{1}{2}\left(\frac{i - x_1}{x_2}\right)^2\right\}\right] + 3$$

and $p = 21$. The region of interest was $1 \leq x_1 \leq 21$, $1 \leq x_2 \leq 8$, with a starting point of $x_1 = 11.0$, $x_2 = 4.5$. The values of $\hat{n}_i$, $i = 1, \ldots, 21$ are 5, 2, 4, 2, 7, 2, 4, 5, 4, 4, 15, 10, 8, 15, 5, 6, 3, 4, 5, 2, 6.

Binomial likelihood. F&N (Freedman and Navidi [8]). Let

$$f(\boldsymbol{x}) = \sum_{i=1}^{n} [v_i \log(1 - e^{-f_i(\boldsymbol{x})}) - f_i(\boldsymbol{x})(p_i - v_i)],$$

where

$$f_i(\boldsymbol{x}) = x_1 t_i^6 (1 + x_2 d_i)(1 + x_3 d_i),$$

and t_i, d_i, p_i, and v_i are given in Table A2. The region of interest was $100 \leq x_1 \leq 140$, $0.1 \leq x_2 \leq 8$, $0.1 \leq x_3 \leq 8$, with $x_3 \leq x_2$. We used the starting point $x_1 = 120.0$, $x_2 = 4.0$, $x_3 = 2.0$.

Table A.2
Data for F&N function

i	t_i	d_i	p_i	v_i	i	t_i	d_i	p_i	v_i
1	0.18	0.00	401	3	16	0.33	0.60	11	7
2	0.24	0.00	383	5	17	0.17	0.75	134	7
3	0.33	0.00	23	4	18	0.18	0.75	267	12
4	0.18	0.30	1573	25	19	0.24	0.75	311	69
5	0.24	0.30	900	83	20	0.33	0.75	12	9
6	0.33	0.30	92	35	21	0.17	1.00	67	3
7	0.17	0.35	389	9	22	0.18	1.00	131	5
8	0.18	0.35	792	19	23	0.24	1.00	160	44
9	0.24	0.35	639	52	24	0.33	1.00	10	10
10	0.33	0.35	45	21	25	0.15	1.50	90	5
11	0.18	0.45	383	3	26	0.16	1.50	86	6
12	0.24	0.45	445	39	27	0.17	1.50	65	4
13	0.33	0.45	12	5	28	0.18	1.50	121	12
14	0.18	0.60	268	6	29	0.24	1.50	130	50
15	0.24	0.60	415	60					

Acknowledgement

The authors thank Chris Fraley and Gina Mladineo for suggestions and discussions relating to this work.

References

[1] L. de Biase and F. Frontini, "A stochastic method for global optimization: Its structure and numerical performance," in: L.C.W. Dixon and G.P. Szegö, eds., *Towards Global Optimization 2* (North-Holland, Amsterdam, 1978) pp. 85-102.

[2] H. Bremermann, "A method of unconstrained global optimization," *Mathematical Biosciences* 9 (1970) 1-15.

[3] R.P. Brent, *Algorithms for Minimization Without Derivatives* (Prentice-Hall, Englewood Cliffs, NJ, 1973).
[4] A. Cutler, "Optimization methods in statistics," Ph.D thesis, U.C. Berkeley (Berkeley, CA, 1988).
[5] L.C.W. Dixon and G.P. Szegö, eds., *Towards Global Optimization* (North-Holland, Amsterdam, 1975).
[6] L.C.W. Dixon and G.P. Szegö, eds., *Towards Global Optimization 2* (North-Holland, Amsterdam, 1978).
[7] L.C.W. Dixon and G.P. Szegö, "The global optimization problem: An introduction," in: L.C.W. Dixon and G.P. Szegö, eds., *Towards Global Optimization 2* (North-Holland, Amsterdam, 1978) pp. 1-15.
[8] D.A. Freedman and W. Navidi, "On the multistage model for carcinogenesis," Technical Report No. 97, Statistics Department, U.C. Berkeley (Berkeley, CA, 1988).
[9] A.O. Griewank, "Generalized descent for global optimization," *Journal of Optimization Theory and Applications* 34(1) (1981) 11-39.
[10] R.H. Mladineo, "An algorithm for finding the global maximum of a multimodal, multivariate function," *Mathematical Programming* 34 (1986) 188-200.
[11] S.A. Piyavskii, "An algorithm for finding the absolute extremum of a function," *USSR Computational Mathematics and Mathematical Physics* 12(4) (1972) 57-67.
[12] W.L. Price, "Global optimization by controlled random search," *Journal of Optimization Theory and Applications* 40 (1983) 333-348.
[13] A.H.G. Rinnooy Kan and G.T. Timmer, "Stochastic global optimization methods. Part I: Clustering methods," *Mathematical Programming* 39 (1987) 27-56.
[14] A.H.G. Rinnooy Kan and G.T. Timmer, "Stochastic global optimization methods. Part II: Multi level methods," *Mathematical Programming* 39 (1987) 57-78.
[15] B.O. Shubert, "A sequential method seeking the global maximum of a function," *SIAM Journal of Numerical Analysis* 9(3) (1972) 379-388.
[16] C.H. Slump and B.J. Hoenders, "The determination of the location of the global maximum of a function in the presence of several local extrema," *IEEE Transactions on Information Theory* IT-31 (1985) 490-497.
[17] J.A. Snyman and L.P. Fatti, "A multi-start global minimization algorithm with dynamic search trajectories," *Journal of Optimization Theory and Applications* 54 (1987) 121-141.
[18] A.A. Törn, "A search-clustering approach to global optimization," in: L.C.W. Dixon and G.P. Szegö, eds., *Towards Global Optimization 2* (North-Holland, Amsterdam, 1978) pp. 49-62.
[19] D. Vanderbilt and S.G. Louie, "A Monte Carlo simulated annealing approach to optimization over continuous variables," *Journal of Computational Physics* 56 (1984) 259-271.
[20] D.R. Wingo, "Estimating the location of the Cauchy distribution by numerical global optimization," *Communications in Statistics Part B: Simulation and Computation* 12(2) (1983) 201-212.