JOURNAL OF OPTIMIZATION THEORY AND APPLICATION: Vol. 79, No. 1, OCTOBER 1993

Lipschitzian Optimization Without the Lipschitz Constant

D. R. JONES,[1] C. D. PERTTUNEN,[2] AND B. E. STUCKMAN[3]

Communicated by L. C. W. Dixon

Abstract. We present a new algorithm for finding the global minimum of a multivariate function subject to simple bounds. The algorithm is a modification of the standard Lipschitzian approach that eliminates the need to specify a Lipschitz constant. This is done by carrying out simultaneous searches using all possible constants from zero to infinity. On nine standard test functions, the new algorithm converges in fewer function evaluations than most competing methods.

The motivation for the new algorithm stems from a different way of looking at the Lipschitz constant. In particular, the Lipschitz constant is viewed as a weighting parameter that indicates how much emphasis to place on global versus local search. In standard Lipschitzian methods, this constant is usually large because it must equal or exceed the maximum rate of change of the objective function. As a result, these methods place a high emphasis on global search and exhibit slow convergence. In contrast, the new algorithm carries out simultaneous searches using all possible constants, and therefore operates at both the global and local level. Once the global part of the algorithm finds the basin of convergence of the optimum, the local part of the algorithm quickly and automatically exploits it. This accounts for the fast convergence of the new algorithm on the test functions.

Key Words. Global optimization, Lipschitzian optimization, space covering, space partitioning.

[1]Staff Research Scientist, General Motors Research and Development Center, Warren, Michigan.

[2]Technical Consultant, Brooks and Kushman, Department of Patent and Computer Law, Southfield, Michigan.

[3]Patent Attorney, Brooks and Kushman, Department of Patent and Computer Law, Southfield, Michigan.

0022-3239/93/1000-0157$07.00/0

1. Introduction

From a theoretical point of view, the Lipschitzian approach to global optimization has always been attractive. By assuming knowledge of a Lipschitz constant (i.e., a bound on the rate of change of the objective function), global search algorithms can be developed and convergence theorems easily proved. Since Lipschitzian methods are deterministic, there is no need for multiple runs. Lipschitzian methods also have few parameters to be specified (besides the Lipschitz constant), and so the need for parameter finite-tuning is minimized. Finally, Lipschitzian methods can place bounds on how far they are from the optimum function value, and hence can use stopping criteria that are more meaningful than a simple iteration limit.

In practice, however, Lipschitzian optimization has three major problems: (i) specifying the Lipschitz constant; (ii) speed of convergence; and (iii) computational complexity in higher dimensions. This paper shows how these problems can be eliminated by modifying the standard approach.

Specifying a Lipschitz constant is a practical problem because a Lipschitz constant may not exist or be easily computed. For example, in optimizing a nonlinear control system, the objective function may be based on a time-consuming simulation or, perhaps, an experiment on the real system (Ref. 1). Similarly, in mechanical engineering applications, designs are often evaluated by a lengthy finite-element analysis. In these cases, no closed-form expression for the objective function is available, and so computing a Lipschitz constant is usually difficult or impossible. The new algorithm eliminates the need to specify the Lipschitz constant by carrying out simultaneous searches using all possible constants from zero to infinity. The exact sense in which this is done will become clear later.

The second problem—speed of convergence—is closely related to the first. As we describe later, the Lipschitz constant can be viewed as a weighting parameter that indicates how much weight to place on global versus local exploration. In standard Lipschitzian methods, this constant is usually large because it must equal (or exceed) the maximum rate of change of the objective function. As a result, these methods place a high emphasis on global search and exhibit slow convergence. In contrast, the new algorithm uses all possible constants, and therefore operates at both the global and local level. Once the global part of the algorithm finds the basin of convergence of the optimum, the local part of the algorithm quickly and automatically exploits it. This is why the new algorithm can converge more quickly than the standard approach.

The third and final problem has to do with computational complexity.

When optimizing a function of n variables subject to simple bounds, the search space is a hyperrectangle in n-dimensional Euclidean space. Most previous Lipschitzian algorithms (Refs. 2–4) partition this search space into smaller hyperrectangles whose vertices are sampled points. Horst and Tuy (Ref. 5) review several such methods. To initialize the search, these algorithms must evaluate all 2^n vertices of the search space. The new algorithm cuts through this computational complexity by sampling the midpoint of each hyperrectangle as opposed its vertices. Whatever the number of dimensions, a rectangle can have only one midpoint.

As mentioned above, the new algorithm does not need a Lipschitz constant to determine where to search. But knowledge of a Lipschitz constant can be helpful in determining when to stop searching (e.g., stop when one is certain to be within ε of the optimum function value). When a Lipschitz constant is not known, the algorithm stops after a prespecified number of iterations.

The new algorithm has only one parameter that must be specified in addition to the iteration limit. Empirical results suggest that the algorithm is fairly insensitive to this parameter, which can be varied by several orders of magnitude without substantially affecting performance. In contrast, many global search methods have several algorithmic parameters that must be carefully adjusted to ensure good results. One of our goals in developing the new algorithm was to eliminate the need to experiment with such algorithmic parameters.

We call the new algorithm DIRECT. This captures the fact that it is a direct search technique and also is an acronym for *di*viding *rec*tangles, a key step in the algorithm. We will introduce DIRECT as a modification and extension of a one-dimensional Lipschitzian algorithm due to Shubert (Ref. 2). We begin in Section 2 by reviewing Shubert's method and discussing why it is hard to extend it to more than one dimension. Section 3 then modifies Shubert's method to make it tractable in higher dimensions and to eliminate the need to specify a Lipschitz constant. This gives us the one-dimensional DIRECT algorithm. Section 4 extends this one-dimensional algorithm to several dimensions. Section 5 proves convergence. Section 6 compares the performance of DIRECT to other algorithms, and Section 7 summarizes our results.

2. Lipschitzian Optimization in One Dimension

Consider the problem of finding the global minimum of a function $f(x)$ defined on the closed interval $[\ell, u]$. Standard Lipschitzian algorithms assume that there exists a finite bound on the rate of change of the

function; that is, they assume that there exists a positive constant K, the Lipschitz constant, such that

$$|f(x)-f(x')| \leqslant K|x-x'|, \qquad \text{for all } x, x' \in [\ell, u]. \tag{1}$$

This assumption can be used to place a lower bound on the function in any closed interval whose endpoints have been evaluated. Figure 1 illustrates this for a hypothetical function on the interval $[a, b]$. If we substitute a and b for x' in (1), we see that $f(x)$ must satisfy the following two inequalities:

$$f(x) \geqslant f(a) - K(x-a), \tag{2}$$

$$f(x) \geqslant f(b) + K(x-b). \tag{3}$$

These inequalities correspond to the two lines with slopes $-K$ and $+K$ shown in Fig. 1. Since the function must lie above the V formed by the intersection of these two lines, the lowest value that $f(x)$ can attain occurs at the bottom of the V. If we denote this point by $X(a, b, f, K)$ and the corresponding lower bound of f by $B(a, b, f, K)$, then we have

$$X(a, b, f, K) = (a+b)/2 + [f(a)-f(b)]/(2K), \tag{4}$$

$$B(a, b, f, K) = [f(a)+f(b)]/2 - K(b-a). \tag{5}$$

These two equations form the core of Shubert's algorithm (Ref. 2). The basic idea is shown in Fig.. 2. We initialize the search by evaluating the function at the endpoints ℓ and u; see part (a) of the figure. We then evaluate the function at $x_1 = X(\ell, u, f, K)$. This divides the search space into two intervals, $[\ell, x_1]$ and $[x_1, u]$; see part (b) of the figure. We now determine which of these intervals has the lowest B-value. In this case, there is a tie, which we break arbitrarily in favor of the interval $[l, x_1]$. We then evaluate the function at $x_2 = X(\ell, x_1, f, K)$. Now the search space is divided into three intervals, $[\ell, x_2]$, $[x_2, x_1]$, $[x_1, u]$; see part (c) of the

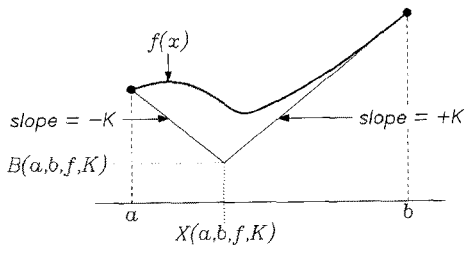

Fig. 1. Computing the Lipschitzian lower bound for an interval.

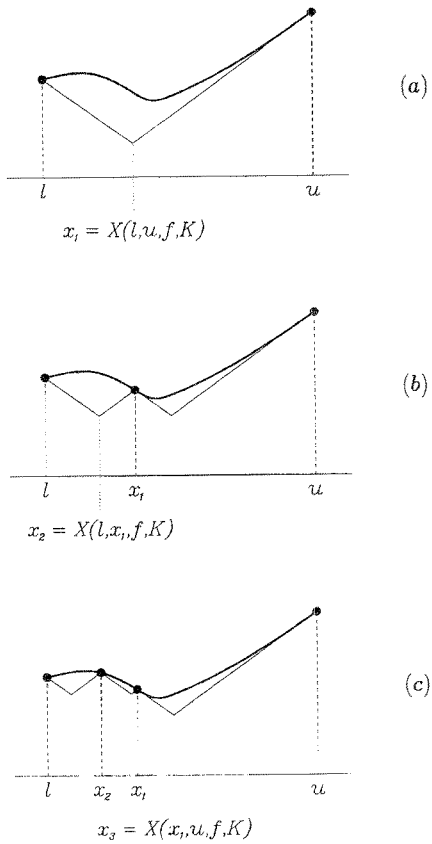

Fig. 2. Shubert's algorithm.

figure. The interval with the lowest B-value is $[x_1, u]$, and so we evaluate the function at $x_3 = X(x_1, u, f, K)$. At any point in Shubert's algorithm, the V's for all the intervals form a piecewise linear function that approximates $f(x)$ from below. The next point sampled is the minimum of this piecewise linear approximation. Shubert's algorithm stops when the minimum of the approximation is within some prespecified tolerance of the current best solution.

As we have seen, Shubert's algorithm selects an interval for further search based on a comparison of lower bounds. Each lower bound, in turn, is the sum of two terms, $[f(a)+f(b)]/2$ and $-K(b-a)$; see Eq. (5). The first term is lower (and therefore better, since we are minimizing) when the function values at the endpoints are low. Thus, this term leads us to select intervals where previous function evaluations have been good, i.e., it leads us to do local search. The second term is lower algebraically the bigger is

the interval, i.e., the bigger is $b-a$. Thus, this term leads us to select intervals with large amounts of unexplored territory, i.e., it leads us to do global search. The Lipschitz constant K serves as a relative weight on global versus local search; the larger K, the higher is the relative emphasis put on global search.

This way of thinking about Shubert's algorithm highlights one of its problems. Since the Lipschitz constant must be an upper bound on the rate of change of the function, it will generally be quite high. In terms of the above discussion, this means that Shubert's algorithm will place a high emphasis on global search and, as a result, may exhibit slow convergence.[4] Once the basin of convergence of the optimum is found, the search would proceed more quickly if K could be reduced, thereby increasing the emphasis on local search. In practice, however, it is difficult to know when and how to reduce K. Thus, one must leave K at its initial value and, if this value is high, one must accept slow convergence as inevitable.

The other problem with Shubert's method is in its initialization phase, where we evaluate the function at the endpoints ℓ and u. Although this is easy to do in one dimension, the natural extension to n dimensions is to evaluate the function at all 2^n vertices of the search space. This is the approach adopted in the multivariate Lipschitzian algorithms of Pinter (Ref. 4) and Galperin (Ref. 3). Mladineo (Ref. 6) has developed an extension of Shubert's algorithm that can be initialized by evaluating the function at a single arbitrary point, but this algorithm is computationally complex for other reasons. In particular, the selection of new points involves solving several systems of n linear equations in $n+1$ unknowns, and the number of such systems goes up rapidly with the number of iterations. For these reasons, the Mladineo algorithm must be modified in order to be applied in more than two dimensions. In summary, Shubert's algorithm has two problems: a potential overemphasis on global search, and the high computational complexity of current multivariate extensions.

3. DIRECT Algorithm in One Dimension

In this section, we modify Shubert's algorithm to alleviate the problems just discussed. The result of these modifications will be the DIRECT algorithm in one dimension.

---

[4]In the extreme case when the Lipschitz constant is infinity, Shubert's algorithm reduces to a grid search. To see this note that, when $K=\infty$, Eq. (5) implies that the biggest interval is selected and Eq. (4) implies that this interval is sampled at its midpoint. It follows that, after $1+2^k$ function evaluations, for any $k=1, 2, \ldots$, the sampled points form a uniform grid over the interval.

The key to making Shubert's algorithm practical in higher dimensions is to modify the way the space is partitioned. Instead of evaluating the function at the endpoints of an interval, we will evaluate it at the center point. In n dimensions, this means that the algorithm can be initialized by sampling just one point (the center of the search space) as opposed to all 2^n vertices of the space. Of course, while center-point sampling enables one to operate in high-dimensional spaces, it does not, by itself, ensure good performance in such spaces.

The shift from sampling endpoints to sampling center points requires some accompanying changes. First, the calculation of an interval lower bound must change. Let $[a, b]$ be an interval with center point $c = (a+b)/2$. Setting x' equal to c in (1), we see that $f(x)$ must satisfy the inequalities

$$f(x) \geqslant f(c) + K(x-c), \qquad \text{for } x \leqslant c, \tag{6}$$

$$f(x) \geqslant f(c) - K(x-c), \qquad \text{for } x \geqslant c. \tag{7}$$

In Fig. 3, these inequalities correspond to the lines with slopes $+K$ and $-K$, and the function must lie above the inverted V formed by their intersection. The lowest value the function can attain occurs at the endpoints a and b. This lower bound is

$$\text{lower bound} = f(c) - K(b-a)/2. \tag{8}$$

Note that the lower bound in Eq. (8) only takes into account the function value at the center of the interval in question. Stronger bounds can sometimes be computed by also considering the function value at the centers of nearby intervals. Unfortunately, computing such stronger bounds becomes intractable in higher dimensions; this is why we use the weaker bound in Eq. (8).

So far, we have said that we will partition the space into intervals whose center points are evaluated and will select intervals based on the

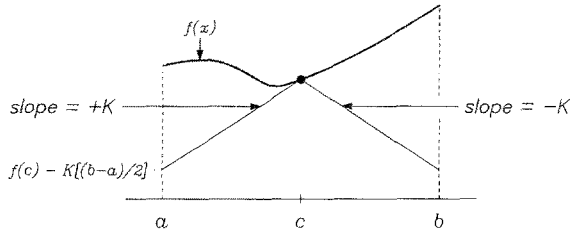

Fig. 3. Computing a lower bound with center-point sampling.

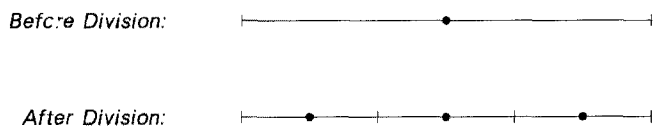

Fig. 4. Subdividing an interval with center-point sampling.

lower bound given in Eq. (8). To complete our shift toward center-point sampling, we must specify where to evaluate the function and how to subdivide the selected interval. In doing this, we must be sure to maintain the property that each interval is sampled at its center. To maintain this property, we have adopted the strategy illustrated in Fig. 4: the interval is divided into thirds, and the function is avaluated at the center points of the left and right thirds. The original center point simply becomes the center of a smaller interval.

Center-point sampling takes us one step toward the DIRECT algorithm. To motivate text next step, let us suppose that we have already partitioned the search space into intervals $[a_i, b_i]$, $i = 1, \ldots, m$, and are in the process of selecting one of these intervals for further sampling. In Fig. 5, we have represented each interval in the partition by a dot with horizontal coordinate $(b_i - a_i)/2$ and vertical coordinate $f(c_i)$. The horizontal coordinate is the distance from the interval's center to its vertices. It captures the goodness of the interval with respect to global search, that is, goodness based on the amount of unexplored territory in the interval. The vertical coordinate is the value of the function at the interval's center. It captures the goodness of the interval with respect to local search, that is, goodness based on known function values. If one passes a line with slope K through any dot in this diagram, the vertical intercept will be the lower bound for

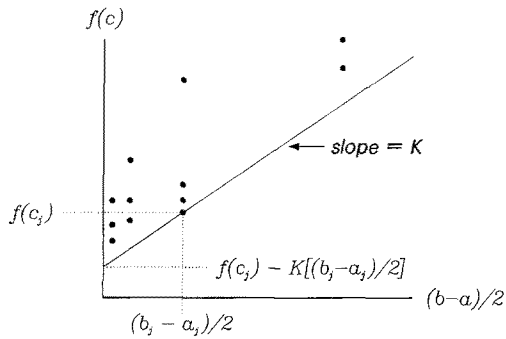

Fig. 5. Graphical interpretation of interval selection.

the corresponding interval. Hence, the interval with the lowest lower bound can be found by positioning a line with slope K below the cloud of dots and shifting it upwards until it first touches a dot. Figure 5 shows how such an optimal dot (and its corresponding interval) is selected.

The Lipschitz constant, reflected in the slope of the line in Fig. 5, determines the relative weighting of global versus local search. In standard methods, this constant is usually high and so tends to overemphasize global search. But what would happen if we used all possible relative weightings? This would correspond to identifying the set of intervals that could be selected using a line with some positive slope. As shown in Fig. 6, these intervals are represented by the dots on the lower right part of the convex hull of the cloud of dots. The basic idea of DIRECT is to select (and sample within) all of these intervals during an iteration. More precisely, we will sample all "potentially optimal" intervals as defined below:

Definition 3.1. Suppose that we have partitioned the interval $[\ell, u]$ into intervals $[a_i, b_i]$ with midpoints c_i, for $i=1, \ldots, m$. Let $\varepsilon > 0$ be a positive constant, and the $f_{\min}$ be the current best function value. Interval j is said to be potentially optimal if there exists some rate-of-change constant $\tilde{K} > 0$ such that

$$f(c_j) - \tilde{K}[(b_j - a_j)/2] \leqslant f(c_i) - \tilde{K}[(b_i - a_i)/2], \qquad \text{for all } i = 1, \ldots, m,$$

$$f(c_j) - \tilde{K}[(b_j - a_j)/2] \leqslant f_{\min} - \varepsilon |f_{\min}|.$$

The first condition in the definition forces the interval to be on the lower right of the convex hull of the dots. The second condition insists that the lower bound for the interval, based on the rate-of-change constant $\tilde{K}$, exceed the current best solution by a nontrivial amount. For example, if

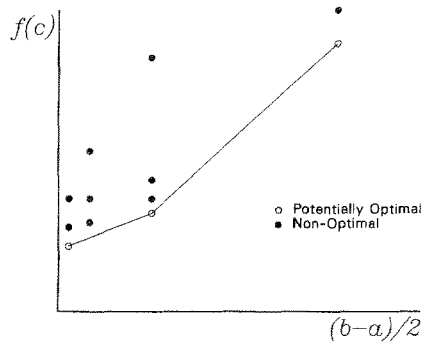

Fig. 6. Set of potentially optimal intervals.

$\varepsilon = 0.01$, then the lower bound for the interval would have to exceed the current best solution by more than 1%. This second condition is needed to prevent the algorithm from becoming too local in its orientation, wasting precious function evaluations in pursuit of extremely small improvements; in terms of Fig. 6, it implies that some of the smaller intervals might not be selected. Later, we will show results suggesting that DIRECT is fairly insensitive to the setting of ε, providing good results for values ranging from 10^{-3} to 10^{-7}. Note that the tilde in $\tilde{K}$ is used to emphasize that $\tilde{K}$ is just a rate-of-change constant and not a Lipschitz constant in the normal sense.

The one-dimensional DIRECT algorithm is essentially Shubert's algorithm modified to use center-point sampling and to sample all potentially optimal intervals during an iteration. If a Lipschitz constant were known, we could also use Shubert's stopping criterion; that is, we could compute a lower bound on the function and stop searching when this bound is within some tolerance of our current best solution.[5] However, we prefer to assume that a Lipschitz constant is not known. Hence, we will stop using a prespecified limit T on the number of iterations. A formal statement of the one-dimensional DIRECT algorithm appears below.

Univariate DIRECT Algorithm.

Step 1. Set $m = 1$, $[a_1, b_1] = [\ell, u]$, $c_1 = (a_1 + b_1)/2$, and evaluate $f(c_1)$. Set $f_{\min} = f(c_1)$. Let $t = 0$ (iteration counter).

Step 2. Identify the set S of potentially optimal intervals.

Step 3. Select any interval $j \in S$.

Step 4. Let $\delta = (b_j - a_j)/3$, and set $c_{m+1} = c_j - \delta$ and $c_{m+2} = c_j + \delta$. Evaluate $f(c_{m+1})$ and $f(c_{m+2})$ and update $f_{\min}$.

Step 5. In the partition, add the left and right subintervals

$$[a_{m+1}, b_{m+1}] = [a_j, a_j + \delta], \quad \text{center point } c_{m+1},$$

$$[a_{m+2}, b_{m+2}] = [a_j + 2\delta, b_j], \text{ center point } c_{m+2}.$$

Then modify interval j to be the center subinterval by setting

$$[a_j, b_j] = [a_j + \delta, a_j + 2\delta].$$

Finally, set $m = m + 2$.

Step 6. Set $S = S - \{j\}$. If $S \neq \varnothing$, go to Step 3.

[5] If a Lipschitz constant K were actually known, one would also modify Definition 3.1 to insist that $\tilde{K} \leq K$.

Step 7. Set $t = t + 1$. If $t = T$, stop; the iteration limit has been reached. Otherwise, go to Step 2.

The order in which the potentially optimal intervals are selected in Step 3 is irrelevant as long as one selects them all. All results reported in this paper will reflect complete iterations; that is, they will reflect values of m and $f_{\min}$ at Step 7.

Although we have not described how to identify the set of potentially optimal intervals, it can be done quite efficiently. For example, an algorithm known as Graham's scan can find the convex hull of a set of m arbitrary points in $O(m \log_2 m)$ time (Ref. 7). If the points are already sorted by their abscissas, it requires only $O(m)$ time. We can do better than this, however, because the points in Fig. 6 are not arbitrary. In particular, since DIRECT always divides intervals into thirds, the only possible interval lengths are $(u-\ell)3^{-k}$, for $k = 0, 1, 2, \ldots$. This means that many of the points in Fig. 5 will have the same abscissa. As a result, if we keep the intervals sorted by function value within groups of intervals with the same length, then we can identify the convex hull in $O(m')$ time, where m' is the number of distinct interval lengths (abscissas in Fig. 6).

4. DIRECT Algorithm in Several Dimensions

In this section, we generalize the one-dimensional DIRECT algorithm to several dimensions. Without loss of generality, we will assume that every variable has a lower bound of zero and an upper bound of one (the scale can always be normalized so that this is true). Thus, the search space will be the n-dimensional unit hypercube. As the algorithm proceeds, this space will be partitioned into hyperrectangles, each with a sampled point at its center. The main issue in extending DIRECT to several dimensions concerns how to divide these hyperrectangles. To keep things simple, we will start our discussion by focusing on the division of hypercubes. Once this is done, we then extend the method to hyperrectangles.

An easy way to divide a hypercube would be to select one dimension arbitrarily and split the hypercube into thirds along this dimension. But selecting a dimension arbitrarily is not conceptually attractive. To avoid such arbitrariness, we have constructed the following approach. We start by sampling the point $\mathbf{c} \pm \delta \mathbf{e}_i$, $i = 1, \ldots, n$, where $\mathbf{c}$ is the center point of the hypercube, δ is one-third the side length of the hypercube, and $\mathbf{e}_i$ is the ith unit vector (i.e., a vector with a one in the ith position and zeros elsewhere). In the two-dimensional example of Fig. 7, this translates into sampling above, below, to the left, and to the right of the center point;

these newly sampled points are shown as open dots with numbers beside them indicating the function's value. By sampling along all dimensions, we have avoided selecting any dimension arbitrarily. But we now must resolve another issue: how are we to divide the hypercube so that each subrectangle has a sampled point at its center?

Figure 7 shows two possible ways to do this for the case $n=2$. In part (a), we first divide the square into thirds along the horizontal dimension and then divide the center rectangle (the one with $\mathbf{c}$) into thirds along the vertical dimension. In part (b), the order is reversed. Both of these division strategies partition the hypercube into subrectangles with sampled points at their centers.

To decide which division order to use, notice that, if we first split on dimension i, then the two points $\mathbf{c} \pm \delta \mathbf{e}_i$ will be at the centers of the biggest subrectangles. For example, in part (a) of Fig. 7, we first divide on dimension 1; as a result, the points with function values 5 and 8 become the centers of the largest subrectangles. This observation leads to the following question: do we want the biggest rectangles to contain the points with the best or the worst function values? The strategy that we have adopted is to make the biggest rectangles contain the best function values. This strategy increases the attractiveness of searching near points with good function values (since bigger rectangles are preferred for sampling, everything else equal). In our experience, the increased emphasis on local search speeds up convergence without sacrificing the global properties of the algorithm, which are ensured by the method of selecting rectangles discussed later.

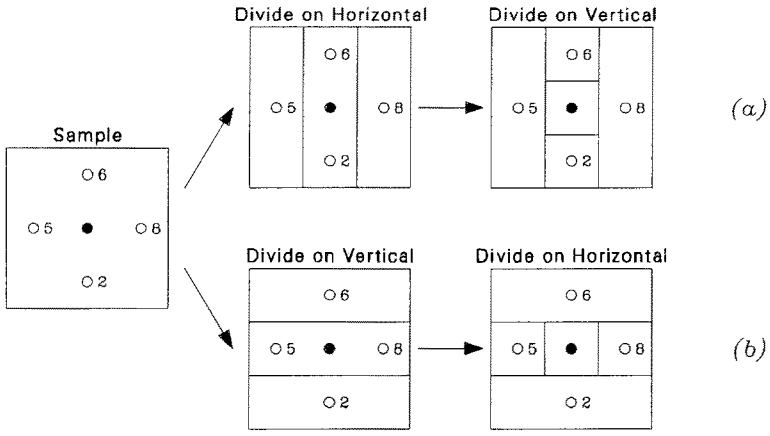

Fig. 7. Sampling and dividing a square in the DIRECT algorithm.

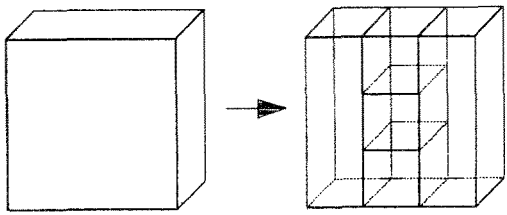

Fig. 8. Dividing a rectangle in the DIRECT algorithm.

More precisely, we have adopted the following rule for subdividing a hypercube. Let

$$w_i = \min\{f(\mathbf{c} + \delta \mathbf{e}_i), f(\mathbf{c} - \delta \mathbf{e}_i)\}$$

be the best of the function values sampled along dimension i. Start by splitting along the dimension with the smallest w value. Once this is done, split the rectangle containing $\mathbf{c}$ into thirds along the dimension with the next smallest w value. Continue in this way until we have split on all dimensions. This splitting rule would select part (b) of Fig. 7.

Once the initial hypercube has been divided, some of the subregions will be rectangular. In dividing such rectangles, we only consider the long dimensions. For example, the three-dimensional rectangle shown in Fig. 8 would be divided along the horizontal and vertical dimensions, but not the shorter depth dimension. By dividing only along the long dimensions, we ensure that the rectangles shrink on every dimension; as we see later, this is essential for proving convergence. A formal description of the rectangle division procedure is given below. Note that this description also covers hypercubes as a special case.

Procedure for Dividing Rectangles.

Step 1. Identify the set I of dimensions with the maximum side length. Let δ equal one-third of this maximum side length.

Step 2. Sample the function at the points $\mathbf{c} \pm \delta \mathbf{e}_i$ for all $i \in I$, where $\mathbf{c}$ is the center of the rectangle and $\mathbf{e}_i$ is the ith unit vector.

Step 3. Divide the rectangle containing $\mathbf{c}$ is into thirds along the dimensions in I, starting with the dimension with the lowest value of

$$w_i = \min\{f(\mathbf{c} + \delta \mathbf{e}_i), f(\mathbf{c} - \delta \mathbf{e}_i)\},$$

and continuing to the dimension with the highest w_i.

The procedure for identifying the set of potentially optimal rectangles in the same as that in one dimension. For each rectangle, we will know the function value at the center point and the distance d from the center point to the vertices. We can now form a diagram like that in Fig. 6, using the distance d for the horizontal axis, and identify the set of potentially optimal intervals as before. Formally, the set of potentially optimal intervals is given by Definition 4.1 below.

Definition 4.1. Suppose that we have a partition of the unit hypercube into m hyperrectangles. Let $\mathbf{c}_i$ denote the center point of the ith hyperrectangle, and let d_i denote the distance from the center point to the vertices. Let $\varepsilon > 0$ be a positive constant. A hyperrectangle j is said to be potentially optimal if there exists some $\tilde{K} > 0$ such that

$$f(\mathbf{c}_j) - \tilde{K} d_j \leqslant f(\mathbf{c}_i) - \tilde{K} d_i, \qquad \text{for all } i = 1, \ldots, m,$$

$$f(\mathbf{c}_j) - \tilde{K} d_j \leqslant f_{\min} - \varepsilon |f_{\min}|.$$

We now have all the ingredients for the DIRECT algorithm in several dimensions. We initialize the search by sampling at the center of the unit hypercube. Each iteration then begins by identifying the set of potentially optimal hyperrectangles. These hyperrectangles are then sampled and subdivided as just described. The process continues until a prespecified iteration limit is reached (one could also stop after a prespecified number of function evaluations). A formal statement of the multivariate DIRECT algorithm is given below.

Multivariate DIRECT Algorithm.

Step 1. Normalize the search space to be the unit hypercube. Let $\mathbf{c}_1$ be the centerpoint of this hypercube and evaluate $f(\mathbf{c}_1)$. Set $f_{\min} = f(\mathbf{c}_1)$, $m = 1$, and $t = 0$ (iteration counter).

Step 2. Identify the set S of potentially optimal rectangles.

Step 3. Select any rectangle $j \in S$.

Step 4. Using the procedure described earlier, 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 Δm is the number of new points sampled.

Step 5. Set $S = S - \{j\}$. If $S \neq \varnothing$ go to Step 3.

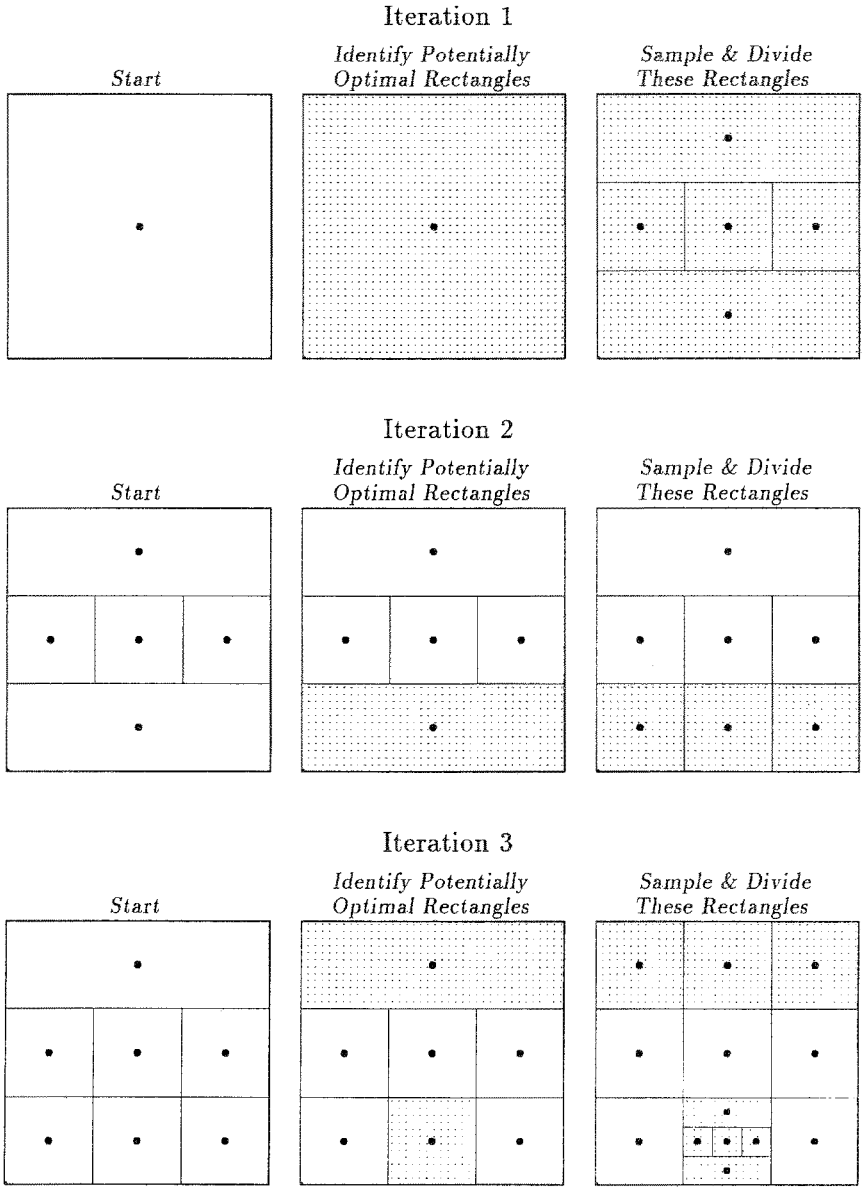

Fig. 9. Three iterations of the DIRECT algorithm on Branin's function.

Step 6. Set $t = t + 1$. If $t = T$, stop; the iteration limit has been reached. Otherwise, go to Step 2.

To provide a more intuitive feeling for how the algorithm works, Fig. 9 shows the first three iterations of the algorithm on the two-dimensional Branin test function (Ref. 8) using $\varepsilon = 0.0001$. For each iteration, column 1 shows the status of the partition at the start of the iteration, column 2 shows the set of potentially optimal rectangles (shaded), and column 3 shows how these rectangles are sampled and subdivided. Figure 10(a) shows a scatter plot of the sampled points after 16 iterations and 195 function evaluations. At this point, the best solution from DIRECT is within 0.01% of the global optimum. The status of the search after 45 iterations and 1003 function evaluations is shown in Fig. 10(b). Branin's function has three global optima, and the sampled points clearly cluster around them.

In our description of the algorithm so far, it would appear necessary to store the center point and side lengths of each rectangle. But one need not store all this information. Instead, one can store information that makes it possible to reconstruct the center points and side lengths when needed. Each time a rectangle is divided, the subrectangles can be considered child rectangles of the original parent rectangle. What we actually store is information on this search tree, such as parent nodes, depth in the tree, child type (left or right), and so on. In this way, the storage requirements of DIRECT become independent of the number of dimensions. However, the storage requirements do increase with the number of function evaluations (i.e., with the number of rectangles being stored).

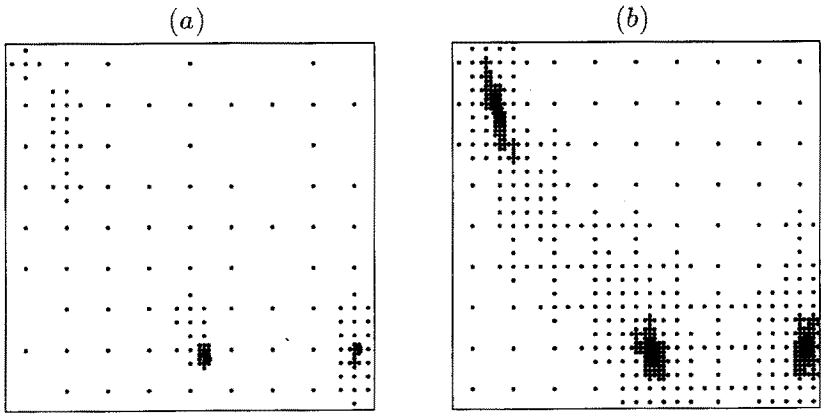

Fig. 10. Scatter plots for Branin's function after 195 and 1003 evaluations.

5. Convergence

DIRECT is guaranteed to converge to the globally optimal function value if the objective function is continuous—or at least continuous in the neighborhood of a global optimum. This follows from the fact that, as the number of iterations goes to infinity, the set of points sampled by DIRECT form a dense subset of the unit hypercube. That is, given any point $\mathbf{x}$ in the unit hypercube and any $\delta > 0$, DIRECT will eventually sample a point within a distance δ of $\mathbf{x}$.

The reason why the iterates of DIRECT are dense is as follows. Recall that the partition is initialized with just one rectangle, the unit hypercube, for which every side has a length of 1. Since new rectangles are formed by dividing existing ones into thirds on various dimensions, the only possible side lengths for a rectangle are 3^{-k} for $k = 0, 1, 2, \ldots$. Moreover, since a rectangle is always divided on its largest side, no side of length $3^{-(k+1)}$ can be divided until all of those of length 3^{-k} have been divided. It follows that, after r divisions, the rectangle will have $j = \text{mod}(r, n)$ sides of length $3^{-(k+1)}$ and $n-j$ sides of length 3^{-k}, where $k = (r-j)/n$. The distance from the center to the vertices is therefore given by

$$d = [j3^{-2(k+1)} + (n-j)3^{-2k}]^{0.5}/2. \tag{9}$$

As the number of divisions approaches infinity, the center-to-vertex distance approaches zero.

Now suppose that we are at the start of iteration t. Each rectangle in the partition will have been divided a certain number of times. Let r_t be the fewest number of divisions undergone by any rectangle. This rectangle would then have the largest center-to-vertex distance. We claim that $\lim_{t \to \infty} r_t = \infty$; that is, the number of divisions of every rectangle approaches infinity. This is easily proved by contradiction. If the $\lim_{t \to \infty} r_t$ is not infinity, then there must exist some iteration t' after which r_t never increases; that is, $\lim_{t \to \infty} r_t = r_{t'}$. Now at the end of iteration t', there will be a finite number of rectangles which have been divided $r_{t'}$ times; let this number be N. All these rectangles will be tied for the largest center-to-vertex distance, but they will differ with respect to the value of the function at the center point. Let rectangle j be the one with the best function value at the center point. In the next iteration, rectangle j will be potentially optimal because the two conditions of Definition 4.1 are satisfied for $\tilde{K} > \max\{K_1, K_2\}$, where

$$K_1 = [f(\mathbf{c}_j) - f_{\min} + \varepsilon |f_{\min}|]/d_j, \tag{10}$$

$$K_2 = \max_{1 \leq i \leq m} [f(\mathbf{c}_j) - f(\mathbf{c}_i)]/(d_j - d_i). \tag{11}$$

But if rectangle j is potentially optimal, it will be subdivided. This will leave $N-1$ rectangles that have been divided only $r_{t'}$ times. Clearly, by iteration $t=t'+N$, all of the original N rectangles will have been divided, implying that $r_t \geq r_{t'}+1$. But this contradicts our assumption that r_t never increases above $r_{t'}$. This contradiction proves that $\lim_{t\to\infty} r_t=\infty$. From this, it follows that the maximum center-to-vertex distance must approach zero as $t\to\infty$. Thus, given any $\delta>0$, there will exist some number of iterations T such that, if $t>T$, then every rectangle has a center-to-vertex distance less than δ. This, in turn, implies that every point in the hypercube will be within a distance δ of some sampled point.

Because the points sampled by DIRECT form a dense subset of the hypercube, DIRECT will converge to the globally optimal function value as long as the function is continuous in the neighborhood of the global minimum. Since any function satisfying a Lipschitz condition is continuous, DIRECT will also converge for any function satisfying a Lipschitz condition, even though the Lipschitz constant may not be known.

6. Performance Comparisons

We have applied DIRECT to nine standard test functions. The first seven were proposed by Dixon and Szego (Ref. 8) as benchmarks for comparing global search methods. The last two are taken from the literature on tunneling algorithms (Ref. 9). Table 1 gives the number of dimensions, local minima, and global minima for each of these functions. All of the test functions are differentiable.

Following Dixon and Szego (Ref. 8), we have compared DIRECT to existing algorithms based on the number of function evaluations required

Table 1. Key characteristics of the test functions.

Test function	Abbreviation	Number of dimensions	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 Shubert	SHU	2	760	18

for convergence as well as the number of units of standard time required, where one unit of standard time corresponds to 1000 evaluations of the Shekel 5 test function at the point (4, 4, 4, 4).

While this may seem like an objective standard, it has some serious problems. First, Dixon and Szego established no definition of convergence, allowing this to be defined by the originators of the methods as they saw fit. Second, no standard was proposed for dealing with stochastic algorithms. It is entirely possible for a stochastic algorithm to converge on some runs but not on others. In these cases, it becomes unclear what one should report as the number of function evaluations required for convergence. Finally, many algorithms have several algorithmic parameters and their performance can be quite sensitive to how these parameters are set. If a large number of runs are spent fine-tuning these parameters, then the performance of the algorithm with the fine-tuned parameters does not truly represent the full effort involved in optimizing the function.

Other methods for comparing optimization algorithms have been proposed. For example, Stuckman and Eason (Ref. 10) have compared several global search algorithms based on percent error after 20, 50, 100, 200, 500, and 1000 function evaluations. Unfortunately, results of this sort are not available for many algorithms.[6]

To use the Dixon/Szego comparison method, we must define what we mean by convergence for DIRECT. Since all the test functions have known global optima, a natural choice is to define convergence in terms of percent error from the globally optimal function value. If we let f_{global} denote this globally optimal function value and let $f_{\min}$ denote the best function value at some point in the search, then the percent error is

$$\text{percent error} = 100(f_{\min} - f_{\text{global}})/|f_{\text{global}}|. \tag{12}$$

We report the number of function evaluations and standard CPU times required to achieve less than 1.0 and 0.01 percent errors. For existing algorithms, we report results based on the definition of convergence used by their authors. In all computer runs, the parameter ε was set to 0.0001.

Tables 2 and 3 summarize these performance comparisons with respect to number of function evaluations and computation time, respectively. The first 11 algorithms all appeared in the 1978 anthology edited by Dixon and Szego (Ref. 8) and therefore are somewhat old. The algorithm by Belisle et al. (Ref. 11) is of the simulated annealing type, while those by Boender et al. (Ref. 12) and Snyman and Fatti (Ref. 13) are variations on the multistart method. The Kostrowicki and Piela algorithm (Ref. 14) uses a local

[6]On request, we will provide full iteration histories for DIRECT on all the test functions (contact D. R. Jones).

optimizer to minimize a smoothed version of the function, with the amount of smoothing being reduced as the algorithm proceeds. Yao's algorithm (Ref. 9) alternates between a local optimization phase and a tunneling phase that attempts to move from the current local minimum into the basin of convergence of a better one. The Perttunen (Ref. 15) and Perttunen/Stuckman (Ref. 16) algorithms are of the Bayesian sampling variety; during each iteration, they sample at a point calculated to have the highest probability of improving upon the current best solution.

To provide some overall perspective, Table 4 summarizes the results as follows. For any test function where both DIRECT and a competing algorithm were applied, we say that DIRECT wins if it converged in fewer function evaluations and loses otherwise. If a competing algorithm converged to a local, it is considered a loss. Similar results are shown for

Table 2. Number of function evaluations in various methods compared to DIRECT.

		Test functions								
Method	Reference	S5	S7	S10	H3	H6	GP	BR	C6	SHU
Bremmerman	8	(a)	(a)	(a)	(a)	(a)	(a)	250	(b)	(b)
Mod Bremmerman	8	(a)	(a)	(a)	(a)	515	300	160	(b)	(b)
Zilinskas	8	(a)	(a)	(a)	8641	(b)	(b)	5129	(b)	(b)
Gomulka–Branin	8	5500	5020	4860	(b)	(b)	(b)	(b)	(b)	(b)
Törn	8	3679	3606	3874	2584	3447	2499	1558	(b)	(b)
Gomulka–Törn	8	6654	6084	6144	(b)	(b)	(b)	(b)	(b)	(b)
Gomulka–V.M.	8	7085	6684	7352	6766	11125	1495	1318	(b)	(b)
Price	8	3800	4900	4400	2400	7600	2500	1800	(b)	(b)
Fagiuoli	8	2514	2519	2518	513	2916	158	1600	(b)	(b)
De Biase–Frontini	8	620	788	1160	732	807	378	587	(b)	(b)
Mockus	8	1174	1279	1209	513	1232	362	189	(b)	(b)
Belisle et al. (c)	11	(b)	(b)	(b)	339	302	4728	1846	(b)	(b)
Boender et al.	12	567	624	755	235	462	398	235	(b)	(b)
Snyman–Fatti	13	845	799	920	365	517	474	(b)	178	(b)
Kostrowicki–Piela	14	12000	12000	12000	200	200	120	(b)	120	(b)
Yao	9	(b)	(b)	(b)	(b)	(b)	(b)	(b)	1132	<6000
Perttunen (d)	15	516	371	250	264	(b)	82	97	54	197
Perttunen–Stuckman (d)	16	109	109	109	140	175	113	109	96	(a)
DIRECT, error <1%		103	97	97	83	213	101	63	113	2883
DIRECT, error <0.01%		155	145	145	199	571	191	195	285	2967

(a) Method converged to a local minimum.
(b) Method not applied.
(c) Average evaluations when converged. For H6, converged only 70% of time.
(d) Convergence defined as obtaining <0.01 percent error.

standardized computation time. All these comparisons use the strict definition of convergence for DIRECT (<0.01% error).

The results of these comparisons show DIRECT to be very competitive with existing algorithms. In terms of function evaluations required for convergence, DIRECT wins in 50% or more of the comparisons against every competing algorithm except Perttunen and Perttunen–Stuckman. In terms of computation time, DIRECT wins in over 50% of the comparisons against every algorithm except Snyman–Fatti and Kostrowicki–Piela.

Many of the close competitors to DIRECT have features that make them less attractive in practical settings. For example, the Perttunen–Stuckman method is known to get fairly close to the optimum quickly but to take much more time to get as close as 0.01% error. To obtain results

Table 3. Normalized computation in various methods compared to DIRECT.

		Test functions								
Method	Reference	S5	S7	S10	H3	H6	GP	BR	C6	SHU
Bremmerman	8	(a)	(a)	(a)	(a)	(a)	(a)	1.00	(b)	(b)
Mod Bremmerman	8	(a)	(a)	(a)	(a)	3.00	0.70	0.50	(b)	(b)
Zilinskas	8	(a)	(a)	(a)	175.00	(b)	(b)	80.00	(b)	(b)
Gomulka–Branin	8	9.00	8.50	9.50	(b)	(b)	(b)	(b)	(b)	(b)
Törn	8	10.00	13.00	15.00	8.00	16.00	4.00	4.00	(b)	(b)
Gomulka–Törn	8	17.00	15.00	20.00	(b)	(b)	(b)	(b)	(b)	(b)
Gomulka–V.M.	8	19.00	23.00	23.00	17.00	48.00	2.00	3.00	(b)	(b)
Price	8	14.00	20.00	20.00	8.00	46.00	3.00	4.00	(b)	(b)
Fagiuoli	8	7.00	9.00	13.00	5.00	100.00	0.70	5.00	(b)	(b)
De Biase–Frontini	8	23.00	20.00	30.00	16.00	21.00	15.00	14.00	(b)	(b)
Mockus	8	(c)	(c)	(c)	(c)	(c)	(c)	(c)	(b)	(b)
Belisle et al.	11	(b)	(b)	(b)	0.88	0.86	9.80	3.40	(b)	(b)
Boender et al.	12	3.50	4.50	7.00	1.70	4.30	1.50	1.00	(b)	(b)
Snyman–Fatti	13	1.10	1.30	2.00	0.60	1.30	0.20	(b)	0.10	(b)
Kostrowicki–Piela	14	15.00	19.00	26.00	0.30	0.50	0.04	(b)	0.05	(b)
Yao	9	(b)	(b)	(b)	(b)	(b)	(b)	(b)	(c)	(c)
Perttunen (d)	15	9259.20	4769.10	2272.00	434.30	(b)	10.11	13.37	4.80	39.06
Perttunen–Stuckman (d)	16	20.59	20.54	20.61	18.32	34.32	16.36	16.39	15.77	(a)
DIRECT, error <1%		0.32	0.33	0.37	0.29	0.70	0.29	0.19	0.28	22.95
DIRECT, error <0.01%		0.68	0.69	0.75	0.87	2.24	0.67	0.70	0.90	23.50

(a) Method converged to a local minimum.
(b) Method not applied.
(c) Computation time not reported.
(d) Convergence defined as obtaining <0.01 percent error.

comparable to DIRECT, we therefore ran the Perttunen–Stuckman method for 100 function evaluations and, if necessary, used a local optimizer to fine-tune the solution to 0.01% error. The local optimizer was the IMSL subroutine DBCONF, a quasi-Newton method using numerical derivatives; the number of function evaluations in Table 2 includes those required to compute these derivatives. While using 100 global evaluations worked well on these test problems, the appropriate number of global evaluations is likely to be problem dependent. As Table 2 shows, a limit of 100 global evaluations was inadequate for the two-dimensional Shubert function. The Shubert function can be successfully optimized using 200 global evaluations but, if 200 evaluations had been used for all the test problems, DIRECT would have won in 6 out of 9 comparisons.

Among the other close competitors, Perttunen's method is extremely CPU intensive (in terms of CPU time, DIRECT beats Perttunen's method on every test function). The multistart algorithms of Boender et al. and Snyman and Fatti require the objective function to be differentiable and may require multiple runs. The diffusion equation method of Kostrowicki and Piela requies, for efficiency, the ability to obtain a closed-form formula

Table 4. Summary comparisons of DIRECT vs. competing algorithms.

	Function evaluations		CPU time
Algorithm	Reference	Win–loss	Win–loss
Bremmerman	8	7–0	7–0
Mod Bremmerman	8	5–2	6–1
Zilinskas	8	5–0	5–0
Gomulka–Branin	8	3–0	3–0
Törn	8	7–0	7–0
Gomulka–Törn	8	3–0	3–0
Gomulka–V.M.	8	7–0	7–0
Price	8	7–0	7–0
Fagiuoli	8	6–1	7–0
De Biase–Frontini	8	7–0	7–0
Mockus	8	6–1	(a)
Belisle et al.	11	3–1	3–1
Boender et al.	12	6–1	7–0
Snyman–Fatti	13	5–2	3–4
Kostrowicki–Piela	14	4–3	3–4
Yao	9	2–0	(a)
Perttunen	15	4–4	8–0
Perttunen–Stuckman	16	1–7	8–0

(a) Computation times not reported.

Table 5. Function evaluations required for convergence.

	Test functions								
ε	S5	S7	S10	H3	H6	GP	BR	C6	SHU
0.01	3749	3741	3741	3817	>10000	191	787	521	1623
0.001	155	145	145	533	985	191	259	285	1887
0.0001 (a)	155	145	145	199	571	191	195	285	2967
0.00001	155	145	145	199	571	191	195	285	3959
0.000001	155	145	145	199	571	191	195	285	4899
0.0000001	155	145	145	199	571	191	195	285	5747

(a) This is the value of ε used in comparisons of DIRECT to other algorithms.

for a particular integral (this was not possible for S5, S7, and S10, which accounts for the high number of function evaluations on those functions).

Table 5 explores the sensitivity of DIRECT to the parameter ε. For each of the test functions, we report the number of function evaluations until convergence for values of ε between 10^{-2} and 10^{-7}. Convergence was defined as achieving less than 0.01% error. Given this definition of convergence, it is most natural to set ε equal to 0.0001. This tells DIRECT to ignore rectangles whose lower bound (using the rate-of-change constant $\tilde{K}$ that makes them potentially optimal) suggests that further search in the rectangle will not improve upon the current best solution by 0.01%. Smaller values of ε make the algorithm more local and tend to increase the number of function evaluations. But, as the table shows, ε can be made several orders of magnitude smaller than its natural value without drastically affecting the results (test function SHU is an exception).

It is risky, however, to increase ε above the value implied by the definition of convergence. Even if the algorithm finds the basin of convergence of the global optimum, large values of epsilon can prevent DIRECT from refining its solution to the desired accuracy. The search becomes very global and lengthy, as indicated by the line in Table 5 for $\varepsilon = 0.01$. In general, we expect that setting ε equal to the desired solution accuracy will yield good results.

7. Conclusions

For an algorithm to be truly global, some effort must be allocated to what we have called global search—search done primarily to ensure that potentially good parts of the space are not overlooked. On the other hand,

to achieve efficiency, some effort must also be placed on local search—search done in the area of the current best solution(s). Most existing algorithms strike a balance between local and global search using one of two approaches. The first is to start with a large emphasis on global search and then shift the emphasis towards local search as the algorithm proceeds. This is the approach followed by simulated annealing (Ref. 11) and the diffusion equation method (Ref. 14). The second approach is to combine a local optimization technique with some other procedure that gives a global aspect to the search. This is the approach adopted by multistart and tunneling algorithms.

DIRECT introduces a third approach to balancing global and local search: do a little of both on every iteration (recall that an iteration in DIRECT consists of several function evaluations). As we have seen, this is accomplished by selecting all those rectangles that would have the lowest lower bound for some rate-of-change constant: small constants select rectangles good for local search, while large constants select those good for global search.

An advantage of this third approach is that it leads to an algorithm with few parameters. In contrast, those algorithms which shift from global to local search usually have parameters that specify the rate at which this shift is accomplished (e.g., the temperature schedule in simulated annealing). Similarly, methods which combine local optimizers with global procedures often have several parameters which control how the global procedures operate. For example, the multistart method of Boender et al. requires one to specify how many random points to evaluate and what fraction of these should be followed up with the local optimizer. Sometimes algorithms are sensitive to such parameters and a good deal of experimentation is required before a satisfactory result is obtained. As we have seen, DIRECT has only one parameter and appears to be fairly insensitive to how it is set.

In summary, practical use of Lipschitzian algorithms has long been impeded by the need to specify a Lipschitz constant. DIRECT eliminates this requirement by carrying out simultaneous searches with all possible Lipschitz constants. The algorithm can operate in high-dimensional spaces because it uses an especially easy-to-manage partition of the space into hyperrectangles whose center points are sampled. The algorithm requires no derivatives and, because it is deterministic, no multiple runs. Parameter fine-tuning is minimized because there is only one parameter which is easy to set. Results for standard test functions suggests that, for problems up to six dimensions, DIRECT is an extremely effective global optimizer that requires relatively few function evaluations.

References

1. STUCKMAN, B., CARE, M., and STUCKMAN, P., *System Optimization Using Experimental Evaluation of Design Performance*, Engineering Optimization, Vol. 16, pp. 275–289, 1990.
2. SHUBERT, B., *A Sequential Method Seeking the Global Maximum of a Function*, SIAM Journal on Numerical Analysis, Vol. 9, pp. 379–388, 1972.
3. GALPERIN, E., *The Cubic Algorithm*, Journal of Mathematical Analysis and Applications, Vol. 112, pp. 635–640, 1985.
4. PINTER, J., *Globally Convergent Methods for n-Dimensional Multiextremal Optimization*, Optimization, Vol. 17, pp. 187–202, 1986.
5. HORST, R., and TUY, H., *On the Convergence of Global Methods in Multiextremal Optimization*, Journal of Optimization Theory and Applications, Vol. 54, pp. 253–271, 1987.
6. MLADINEO, R., *An Algorithm for Finding the Global Maximum of a Multimodal, Multivariate Function*, Mathematical Programming, Vol. 34, pp. 188–200, 1986.
7. PREPARATA, F., and SHAMOS, M., *Computational Geometry: An Introduction*, Springer-Verlag, New York, New York, 1985.
8. DIXON, L., and SZEGO, G., *The Global Optimization Problem: An Introduction*, Toward Global Optimization 2, Edited by L. Dixon and G. Szego, North-Holland, New York, New York, pp. 1–15, 1978.
9. YAO, Y., *Dynamic Tunneling Algorithm for Global Optimization*, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 19, pp. 1222–1230, 1989.
10. STUCKMAN, B., and EASON, E., *A Comparison of Bayesian Sampling Global Optimization Techniques*, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 22, pp. 1024–1032, 1992.
11. BELISLE, C., ROMEIJN, H., and SMITH, R., *Hide-and-Seek: A Simulated Annealing Algorithm for Global Optimization*, Technical Report 90-25, Department of Industrial and Operations Engineering, University of Michigan, 1990.
12. BOENDER, C., et al., *A Stochastic Method for Global Optimization*, Mathematical Programming, Vol. 22, pp. 125–140, 1982.
13. SNYMAN, J., and FATTI, L., *A Multistart Global Minimization Algorithm with Dynamic Search Trajectories*, Journal of Optimization Theory and Applications, Vol. 54, pp. 121–141, 1987.
14. KOSTROWICKI, J., and PIELA, L., *Diffusion Equation Method of Global Minimization: Performance on Standard Test Functions*, Journal of Optimization Theory and Applications, Vol. 69, pp. 269–284, 1991.
15. PERTTUNEN, C., *Global Optimization Using Nonparametric Statistics*, University of Louisville, PhD Thesis, 1990.
16. PERTTUNEN, C., and STUCKMAN, B., *The Rank Transformation Applied to a Multiunivariate Method of Global Optimization*, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 20, pp. 1216–1220, 1990.