Dynamic Data Structures for a Direct Search Algorithm

JIAN HE

Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, USA

LAYNE T. WATSON

Departments of Computer Science and Mathematics, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, USA

NAREN RAMAKRISHNAN

Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, USA

CLIFFORD A. SHAFFER

Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, USA

ALEX VERSTAK

Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, USA

JING JIANG

Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, USA

KYUNG BAE

Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, USA

WILLIAM H. TRANTER

Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061, USA

Abstract. The DIRECT (DIviding RECTangles) algorithm of Jones, Perttunen, and Stuckman (1993), a variant of Lipschitzian methods for bound constrained global optimization, has proved effective even in higher dimensions. However, the performance of a DIRECT implementation in real applications depends on the characteristics of the objective function, the problem dimension, and the desired solution accuracy. Implementations with static data structures often fail in practice, since it is difficult to predict memory resource requirements in advance. This is especially critical in multidisciplinary engineering design applications, where the DIRECT optimization is just one small component of a much larger computation, and any component failure aborts the entire design process. To make the DIRECT global optimization algorithm efficient and robust on large-scale, multidisciplinary engineering problems, a set of dynamic data structures is proposed here to balance the memory requirements with execution time, while simultaneously adapting to arbitrary problem size. The focus of this paper is on design issues of the dynamic data structures, and related memory management strategies. Numerical computing techniques and modifications of Jones' original DIRECT algorithm in terms of stopping rules and box selection rules are also explored. Performance studies are done for synthetic test problems with multiple local optima. Results for application to a site-specific system simulator for wireless communications systems (S^4W) are also presented to demonstrate the effectiveness of the proposed dynamic data structures for an implementation of DIRECT.

Keywords: Global optimization, DIRECT algorithm, direct search, dynamic data structures

1. Introduction

The DIRECT (DIviding Rectangles) algorithm by Jones et al. [8] was proposed as an effective approach to solve global optimization problems (GOP) subject to simple constraints. The general problem statement is [11]

$$\min_{x \in D} f_0(x) \tag{1.1}$$

$$D = \{x \in D_0 \mid f_j(x) \leq 0,\ j = 1, \ldots, J\},$$

where $D_0 = \{x \in E^n \mid \ell \leq x \leq u\}$ is a simple box constraint set. The objective function and constraints f_j, $j = 0, \ldots, J$, must be Lipschitz-continuous on D_0, satisfying

$$|f_j(x_1) - f_j(x_2)| \leq L_j \|x_1 - x_2\|, \quad \forall x_1, x_2 \in D_0. \tag{1.2}$$

This assumption means that the rates-of-change of the objective function f_0 and constraints $f_1, \ldots, f_J$ are bounded.

Traditionally, this class of problems was solved by the Lipschitz optimization method, which had been considered as a practical and deterministic approach to many science and engineering problems for several decades. Unlike some other methods (e.g., concave minimization), the Lipschitz global optimization method requires only a few parameters. This is the major reason why it is an ideal system model for "black box" or "oracle" systems, which can only generate corresponding function values for a given collection of arguments, but can not provide any more analytical information on the system [11]. Furthermore, the convergence of Lipschitz-based global optimization algorithms can be easily proved by assuming the knowledge of a Lipschitz constant [8]. However, as a coin has two faces, this assumption of a Lipschitz constant carries disadvantages. First of all, the Lipschitz constant of a particular function is usually unknown or hard to estimate in practice. Although an overestimated Lipschitz constant is still valid for the application of Lipschitz global optimization (LGOP) methods, it results in slow convergence and complicates computation in higher dimensions. These practical problems motivated Jones et al. [8] to develop a new Lipschitz-based optimization algorithm—DIRECT—that is guaranteed to converge to the global optimum without the knowledge of the Lipschitz constant.

Jones et al. [8] named the new algorithm after one of its key steps—dividing rectangles. DIRECT is a pattern search method, which is categorized as a direct search technique by Lewis et al. [10]. Generally speaking, "pattern search methods are characterized by a series of exploratory moves that consider the behavior of the objective function at a pattern of points," [10], which are chosen as the centers of rectangles in the DIRECT algorithm. This center-sampling strategy reduces the computational complexity, especially for higher dimensional problems, and hence outperforms some other earlier attempts at improving LGOP methods, such as Shubert's or Piyavskii's methods (a detailed comparison of DIRECT and Shubert's method in one dimension can be found in [8]). Moreover, DIRECT adopts a strategy of balancing local and global search by selecting potentially optimal rectangles to be further explored. This strategy gives rise to fast convergence with reasonably broad space coverage.

So far, DIRECT has been used with fair success for modern large-scale, multidisciplinary engineering problems [1]. Nevertheless, it does have limitations as pointed out by Jones [9]. Some applicability concerns include: (1) the space-partitioning strategy in practice limits the algorithm to low-dimensional problems (≤ 20), although Baker et al. [1] have solved realistic 29-dimensional problems, and (2) the stopping criterion—a limit on function evaluations is not convincing. The difficulty of implementing space partitioning in high dimensions lies in the efficiency of maintaining partitioning information. To address this efficiency issue, this paper proposes a data structure to store such information in a way that balances efficient access with memory requirements. Moreover, alternate choices for the stopping criterion are offered, which provide more freedom for a wide variety of applications.

Unpredictable memory demand is a practical problem due to different characteristics of the objective functions, problem dimensions, and desired solution accuracy. Many implementations of DIRECT rely on allocating a large static two-dimensional array to store the current state of the space partitioning. This can lead to failure of the code if the array is insufficient to hold the necessary information due to exceeding one or the other of the dimensions. To overcome this problem, some implementations will reallocate the array to be larger if necessary. Even with this modification there remains a significant amount of overhead in both execution time and space required. The problem is that a few columns of the array will require an unusually large amount of space. Thus, some form of dynamic data structure is required for at least these relatively few columns. To reduce the execution overhead and adapt to varying memory requirements, a set of dynamic data structures are proposed here. They are extensible and flexible in dealing with information generated by the space partitioning process in high dimensions. The dynamic memory implementation proposed here is implemented for a single processor, but it should provide considerable flexibility for future parallelization of the DIRECT algorithm.

The paper is organized as follows. Section 2 begins with an overview of the DIRECT algorithm followed by the proposed modifications. Section 3 details the design aspects of the dynamic data structures and related memory management strategies. Important implementation considerations involved in numerical computing and computational geometry are also discussed. In Section 4, numerical results and performance analyses for four sample objective functions are presented. Section 5 addresses issues related to an optimization problem for wireless communications, and presents experimental results for S^4W (a Site-Specific System Simulator for Wireless communication). Finally, Section 6 summarizes some key contributions of this implementation and considers several future tasks such as a dynamic memory MPI-based parallel version.

2. DIRECT algorithm overview and modifications

DIRECT evolved from the one-dimensional Piyavskii-Shubert algorithm and was further extended from one dimension to multiple dimensions by adopting a center-sampling strategy. Its corresponding 1-D description contrasted with Piyavskii-Shubert's algorithm can be found in [8]. Here, only the multidimensional DIRECT algorithm, which is of more interest for large-scale applications, is described. Also, constraints other than bound constraints are not considered here. Thus henceforth assume $D = D_0$.

DIRECT's behavior in multiple dimensions can be viewed as taking steps in potentially optimal directions within the entire design space. The potentially optimal directions are determined through evaluating the objective function at center points of the subdivided boxes. The multivariate DIRECT algorithm can be described by the following six steps [8].

Given an objective function f and the design space $D = D_0$:

Step 1. Normalize the design space D to be the unit hypercube. Sample the center point c_i of this hypercube and evaluate $f(c_i)$. Initialize $f_{\min} = f(c_i)$, evaluation counter $m = 1$, and iteration counter $t = 0$.

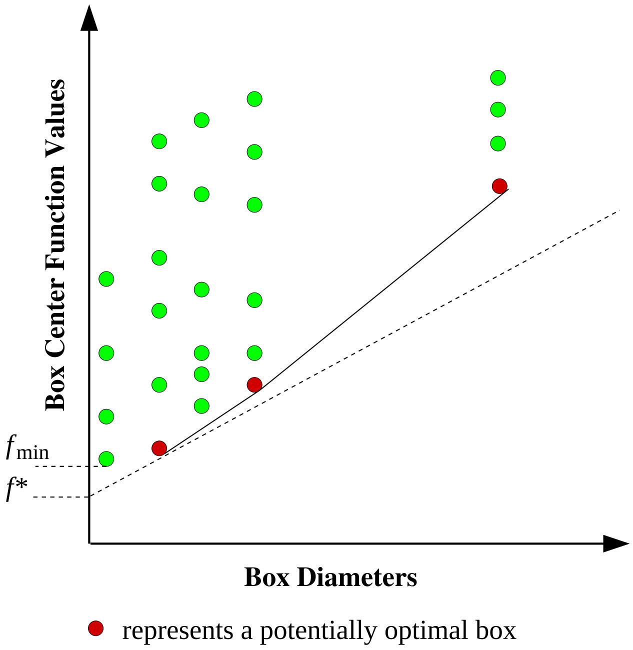

Figure 2.1. Illustration of potentially optimal boxes on convex hull with ϵ test from [9]. Note that $f^* = f_{\min} - \epsilon|f_{\min}|$. Potentially optimal boxes are on the lower-right convex hull.

Step 2. Identify the set S of potentially optimal boxes.
Step 3. Select any box $j \in S$.
Step 4. Divide the box j as follows:
- **(1)** Identify the set I of dimensions with the maximum side length. Let δ equal one-third of this maximum side length.
- **(2)** Sample the function at the points $c \pm \delta e_i$ for all $i \in I$, where c is the center of the box and e_i is the ith unit vector.
- **(3)** Divide the box j containing c into thirds along the dimensions in I, starting with the dimension with the lowest value of $w_i = \min\{f(c + \delta e_i), f(c - \delta e_i)\}$, and continuing to the dimension with the highest w_i. Update $f_{\min}$ and m.

Step 5. Set $S = S - \{j\}$. If $S \neq \emptyset$ go to Step 3.
Step 6. Set $t = t + 1$. If iteration limit or evaluation limit has been reached, stop. Otherwise, go to Step 2.

[8] provides a good step-by-step example visualizing how DIRECT accomplishes the task of locating a global optimum. Steps 2 to 6 form a processing loop controlled by two stopping criteria—limits on iterations and function evaluations. Starting from the center of the initial hypercube, DIRECT makes exploratory moves across the design space by probing the potentially optimal subspaces. “Potentially optimal” is an important concept defined next [8].

Definition 2.1. Suppose that the unit hypercube has been partitioned into m (hyper) boxes. Let c_i denote the center point of the ith box, and let d_i denote the distance from the center point to the vertices. Let $\epsilon > 0$ be a positive constant. A box j is said to be *potentially optimal* if there exists some $\tilde{K} > 0$ such that for all $i = 1, \ldots, m$,

$$f(c_j) - \tilde{K} d_j \leq f(c_i) - \tilde{K} d_i, \tag{2.1}$$

$$f(c_j) - \tilde{K} d_j \leq f_{\min} - \epsilon |f_{\min}|. \tag{2.2}$$

Figure 2.1 represents the set of boxes as points in a plane. The first inequality (2.1) screens out the boxes that are not on the lower right of the convex hull of the plotted points, as shown in Figure 2.1. Note that $\tilde{K}$ plays the role of the (unknown) Lipschitz constant. The second inequality (2.2) prevents the search from becoming too local and ensures that a nontrivial improvement will (potentially) be found based on the current best solution. In Figure 2.1, $f_{\min}$ is the current best solution, but its associated box is screened out of the potentially optimal box set due to the second inequality (2.2).

As a comparatively young method, DIRECT is being enhanced with novel ideas and concepts. Jones has made a couple of modifications to the original DIRECT in a recent paper [9]. In Step 4, the modified version only trisects in one dimension with the longest side length instead of in all identified dimensions in set I as above. The dimension to choose depends upon a tie breaking mechanism. Baker [2] proposes an "aggressive DIRECT", which discards the convex hull idea of identifying potentially optimal boxes. Instead, it subdivides all the boxes with the smallest objective function values for different box sizes. The change results in more subdivision tasks generated at every iteration, which helps to balance the workload in a parallel computing environment. Gablonsky et al. [6] studied the behavior of DIRECT in low dimensions and developed an alternative version for biasing the search more toward local improvement by forcing $\epsilon = 0$.

The implementation of DIRECT considered here is mostly based on the original version. Some modifications with respect to the stopping rules and box selection rules are proposed here to offer more choices for different types of intended applications. Two new stopping criteria are (1) minimum diameter (terminate when the best potentially optimal box's diameter is less than this minimum diameter) and (2) objective function convergence tolerance (exit when the objective function does not decrease sufficiently between iterations). The minimum diameter of a hyperbox represents the degree of space partition, and therefore is a reasonable criterion for applications requiring only some depth of design space exploration, such as conceptual aircraft design [14]. The objective function convergence tolerance was inspired by some experimental observations in the later stages of running the DIRECT algorithm, when the objective function convergence tolerance test avoids wasting a great number of expensive function evaluations in pursuit of very small improvements. In terms of box selection rules, two modifications are proposed. First, an optional "aggressive switch" is proposed to turn on/off convex hull processing as first used in [2]. Secondly, ϵ is taken as zero by default, but also can be assigned a value on input tailored to the application. Comparisons of DIRECT performance with the "aggressive" switch on/off, and with ϵ tuning will be presented in Section 4.

A final observation here is that Jones' original description of DIRECT used the word "rectangle" rather than the more commonly accepted terms "box" or "hyperbox." In the following, the step of identifying potentially optimal boxes is often referred to as *convex hull processing*.

3. Implementation

Recall that the motivation for this implementation is to handle efficiently the unpredictable amount of storage and information required by the space partition. The main problem to be solved is how to store the large collection of boxes, typically viewed as a set of separate columns making up the points shown in Figure 2.1. The key operations are to find the element in a column with least value, to remove this least-valued element, and to add new elements to a column. Thus each column can be viewed abstractly as a priority queue.

Typical implementations for DIRECT simply allocate a large two-dimensional array to store the boxes as organized in Figure 2.1. Each column of the array corresponds to the set of boxes with a given diameter. This approach has the advantage of being simple, and matching well with the memory access patterns that work efficiently in parallel implementations. However, the actual performance for this implementation is poor for two reasons. First, there can be a large number of distinct box diameters at various times during the execution of the algorithm. This translates to a potential (but changing) need for many columns. Second, specific columns can get unusually large numbers of boxes at various times, translating into a potential (but changing) need for many rows. These behaviors are both transient and unpredictable. Thus, a dynamic data structure is needed.

In practice, only a few of the columns become large at any given time. The large memory requirements of the computations involved (of which the box processing is only a small part) argue against careless use of dynamic memory allocation, since, for example, a list implementation that spreads the contents of a column widely through virtual memory will result in poor use of the memory cache.

The proposed implementation is a simple modification to the columns to provide flexibility in their length. Initially a two-dimensional array of fairly large size (depending on the dimension of the problem) is allocated in the usual way. Depending on the size and nature of the problem, this array might hold all boxes in the partitioning. Certainly, for most columns all elements in the column will remain in the array. However, the array is dynamic in that it can grow in either of two ways. First, if the array provides insufficient columns, new blocks of columns will be allocated as needed. Second, should a given column outgrow the space available in the array, a new chunk of space is allocated to that column.

Within a column, the points are maintained in sorted order, removing the top (lowest) value as needed, and adding new values when needed. As necessary, a chunk of additional space is added or removed from the column. One alternative implementation would be to store the column points in a standard heap data structure rather than in sorted order, but the sorted list and heap implementations have not been compared.

Functionally, the Fortran 90 derived data type dynamic structures can be classified into two groups: box structures and linked list structures. The box structures (`BoxMatrix`, `BoxLink`, and `HyperBox`) are responsible for holding boxes. The linked lists (`setInd`, `setDia`, and `setFcol`) are built out of linked vectors (`real_vector` and `int_vector`), and manage the allocated memory for the box structures. Their use is illustrated by Figures 3.1 and 3.3.

The row dimension of the initial `BoxMatrix` is

$$n_r = \begin{cases} \max\{10, 2n\}, & \text{if } n \le 10; \\ 17 + \lceil \log_2 n \rceil, & \text{otherwise;} \end{cases}$$

and the column dimension is $n_c = 35n$, where n is the problem dimension. These formulas are based on empirical observations of box sequence lengths and the number of distinct diameters extant during runs of many different test problems with n from 2 to 50. An attempt was made to balance memory utilization within the initial `BoxMatrix` with the need to minimize the number of new `BoxLink`s and `BoxMatrix`s allocated. This balance is extremely problem dependent, but typically the above formulas result in all but a few very long box sequences fitting in the initial `BoxMatrix`, and only occasionally are additional `BoxMatrix`s required, depending on the problem and stopping criteria.

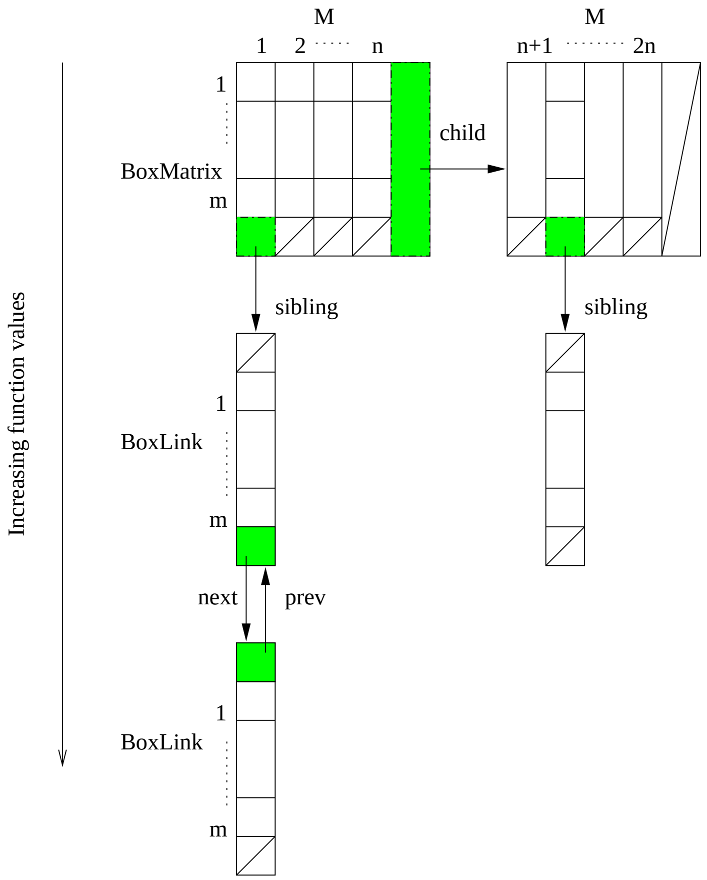

Figure 3.1. Box structures comprised of `HyperBox`es.

3.1. Box structures

Figure 3.1 shows the two dimensional chain structure of the box structures group. It consists of three derived data types: `BoxMatrix`, `BoxLink`, and `HyperBox`. `HyperBox` is the basic unit for constructing `BoxMatrix` and `BoxLink`. It contains all the necessary information about a hyperbox, namely, the objective function value at the box center, the coordinates of the center point, the side lengths in all dimensions, and the box size (diameter squared). Without further organizing the information listed above, some well-known methods for finding the convex hull can be applied. In [8], Graham's scan method is recommended because it is one of the most efficient algorithms, finding the convex hull of a set of m arbitrary points in time $O(m \log_2 m)$. In the present implementation, a different approach is taken to shrink the initial set with m points to a much smaller set of vertices exclusively around the low edge of the convex hull as depicted in Figure 3.2.

As already described, all hyperboxes of a given diameter are sorted according to the center points' function values. The actual sorted list is made up of a column from a `BoxMatrix`, perhaps followed by some number of `BoxLink`s as shown in Figure 3.1. When a column in the initial `BoxMatrix` named `M` is full, a `BoxLink` is allocated and connected at the end of the column as a **`sibling`** link, which holds a one-dimensional array of `HyperBox`es with the same number of `HyperBox`es as a column in `M`. A `BoxLink` is extended in the same fashion when it becomes full. All boxes of the same size find their places in this box sequence, consisting of a column of `M` followed by an unlimited number of box links. Figure 3.1 illustrates the use of these box structures during execution of the DIRECT algorithm, when column one in `M` of the first `BoxMatrix` has become full,

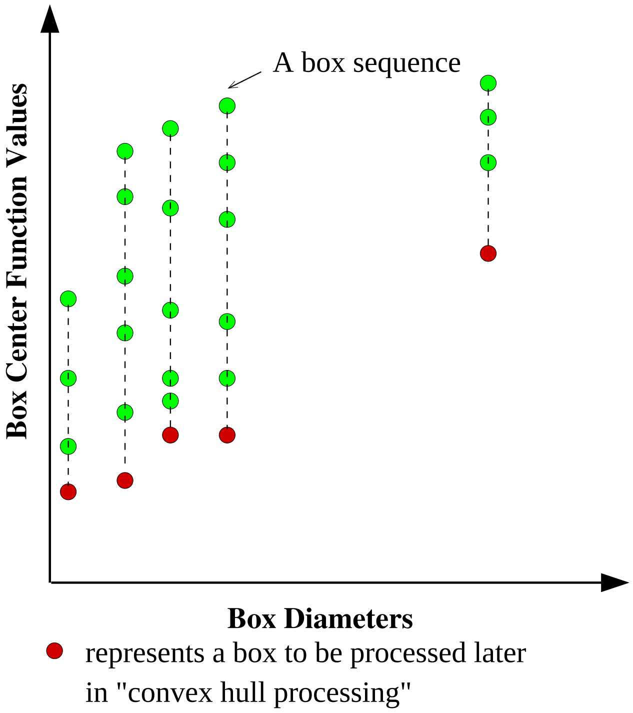

Figure 3.2. Scatter plot pattern.

thereafter having been linked with two more `BoxLinks`, which are associated with each other by referencing their `next` and `prev` pointers. Notice that `M` has the same number of hyperboxes in a column as a `BoxLink` does, which unifies the procedures for box insertion both in `M` and `BoxLink`. Inserting a new box into a box sequence requires three steps. First, locate the segment of the sequence that the box's function value falls within, either the column in `M` or one of the `sibling` box links. Second, apply a binary search to the function values in the located segment to find the appropriate position at which to insert the new box. Third, shift the remaining elements in the column down by one position, possibly causing an additional `BoxLink` to be allocated.

While caching performance encourages maintaining adjacent elements of a column in adjacent memory locations, the same is not true of adjacent columns of the array. Further, during processing it may happen that a given column becomes empty (that is, all boxes of a given diameter may be split) and another column may need to be created (as boxes of new diameters are created by the splitting process). Because it would be costly to sort box sequences with respect to box sizes by rearranging the columns of `M`, columns are not kept sorted by box size. However, it is necessary to find the column (if any) that stores the boxes of a given size. A linked list structure (described in more detail in the next section) is used to maintain the box sizes in logical decreasing order. Physically, the columns in `M` are treated as independent cells that can be popped up for any boxes with a new size. In some sense, `M` acts as a memory pool of recyclable cells. When cells are used up, a new `BoxMatrix` is allocated and connected as the `child` link at the end of the chain of `BoxMatrices`, so that the memory pool can be filled up again using new cells from `M` in the newly allocated `BoxMatrix`. As an instance, Figure 3.1 shows a chain of two `BoxMatrices`. In this specific example, a `BoxMatrix` allocates `M` with m rows and n columns of `Hyperboxes`. The column indices of the second `BoxMatrix` begin with $n+1$ to be distinguished from indices in the first `BoxMatrix`. Cell recycling is handled by the linked list data structures, discussed below.

With all the hyperboxes linked logically in the scatter plot pattern as in Figure 3.2, Jarvis's march (or gift wrapping) method is applied starting from the box sequence with the biggest size, and eventually identifies all the potentially optimal boxes to be further subdivided for the next iteration. Pseudo code for finding the convex hull follows. Let the first box in column j have radius d_j and center value f_j. (Recall that box sequences are indexed by *decreasing* box diameters).

$i :=$ index of first (largest diameter) box column
$k :=$ index of second box column
while i has not reached the column with $f_{\min}$ **do**
begin
  $\bar{s} := -\infty$
  while k has not reached the column with $f_{\min}$ **do**
  begin
    $s := \dfrac{f_i - f_k}{d_i - d_k}$
    if $s > \bar{s}$ **then**
      $\bar{s} := s$
      $t := k$
    end if
    move k to the next box column index
  end
  screen out the columns between i and t
  if $\epsilon \neq 0$ **then**
    if $\bar{s} < \dfrac{f_i - (f_{\min} - \epsilon|f_{\min}|)}{d_i}$ **then**
      screen out the columns from t through the column with $f_{\min}$
      break
    end if
  end if
  move i to t and move k to the column index next to i
end

3.2 Linked list structures

The linked list data structures play an important role in maintaining the logical scatter plot pattern and recycling memory cells. They are doubly linked lists constructed with two derived data types. `setInd` and `setFcol` are of the type `int_vector`, which contains a one-dimensional array of integer elements and two pointers—`next` and `prev`—for tracing back and forth. `setDia` differs only in containing real elements defined in `real_vector`. Each linked list starts out with only one link initialized corresponding to the first `BoxMatrix`. The number of elements in the one-dimensional array is equal to the number of columns n in `M` of a `BoxMatrix`. Except for the first column used by the normalized hyperbox at Step 1 of the DIRECT algorithm, the other column indices are pushed into `setFcol` for later usage. When a new `BoxMatrix` is added at the end of the existing `BoxMatrix` chain, each of the three linked lists is also expanded with a newly allocated link for

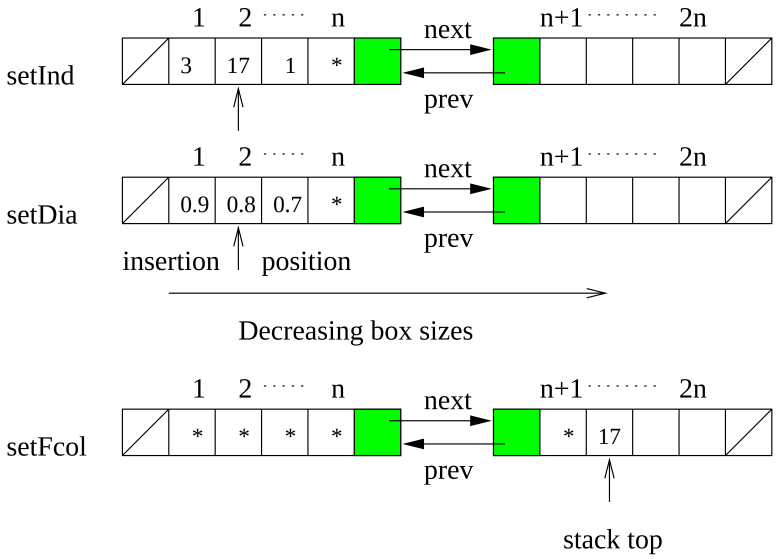

Figure 3.3. Linked list structures. Insertion of a new box size (0.8) has four steps: (a) request a free column index (17) from the stack top of `setFcol` by popping the stack, (b) locate the insertion position (2) in `setDia` and insert the new diameter (`setDia` is shown after the insertion), (c) add the column index to `setInd` at the insertion position (shown), and (d) add the box at the beginning of the requested column (17) in `M`.

manipulating the new `BoxMatrix`. For example, in Figure 3.3, each linked list data structure has two links corresponding to the two `BoxMatrices` in Figure 3.1.

From the viewpoint of memory management, these three linked lists collaborate with each other recycling the memory cells allocated for `BoxMatrix` structures. Every time a new box size is produced from box subdividing, the box with this size requests a free column index from the (stack) top of `setFcol`. Similar to locating the position at which to insert a new box into a box sequence (illustrated in pseudo code in Section 3.1), an appropriate position will be found for this new box size in `setDia`, which is kept in descending order of box sizes. Finally, the requested column index is added in `setInd` at the corresponding position. The process is reversed when a box size no longer exists after the last box with this size has been subdivided. As a result, the released column index is pushed back to the stack—`setFcol`. Figure 3.3 illustrates insertion of a new size.

For faster execution, sorting is not involved in the strategy for maintaining a logical scatter plot pattern of hyperboxes. Instead, binary search is used in locating the insertion positions in sorted sequences, in both the cases of inserting boxes and box sizes. Some shifting operations are needed for inserting/deleting boxes in a particular column of boxes in `M` and its box links, if any, while shifting boxes among columns is avoided by keeping column indices sorted (by decreasing box sizes) in `setInd`.

The last important implementation issue is related to floating point comparisons involved in box insertion. For portability, the module REAL_PRECISION from HOMPACK90 ([15]) is used to define 64-bit real arithmetic. All equality tests between two real values are done in the following manner: given two real values r_1 and r_2, r_1 and r_2 are considered equal if they satisfy

$$\frac{|r_1 - r_2|}{|r_2|} \leq 4nu, \tag{3.1}$$

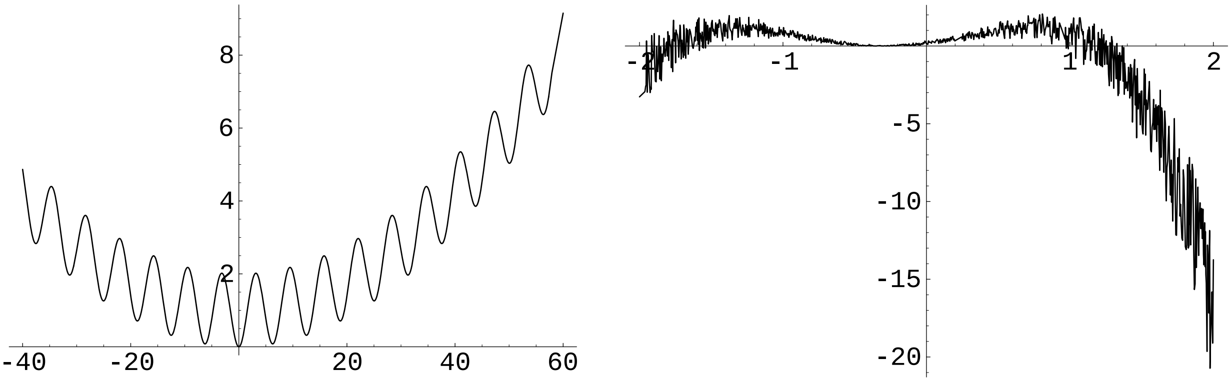

Figure 4.1. One-dimensional Griewank function with parameter $d = 500$ (left), and one-dimensional noisy quartic function (right).

where $4nu$ is the estimated round-off error based on the problem dimension n. This is very important when comparing sizes of boxes in high dimensions after a number of iterations, since round off error will make mathematically equal diameters slightly different. The same principle is followed when comparing objective function values for inserting a box to a box sequence.

4. Test cases and performance studies

The DIRECT algorithm as described here has been applied to several standard test functions. Among them, the Griewank function and quartic function [3] are chosen here to study the behavior of the DIRECT algorithm and evaluate the performance of this implementation.

The n-dimensional Griewank function [3]

$$f(x) = 1 + \sum_{i=1}^{n} \frac{{x_i}^2}{d} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right), \tag{4.1}$$

where $d > 0$ is a constant to adjust the noise, has a unique global minimum at $x = 0$, and numerous local minima (see Figure 4.1). The larger the value of d, the deeper the minima values are.

The second test function is an n-dimensional quartic function with a random noise variable defined by [3]

$$f(x) = \sum_{i=1}^{n} [2.2(x_i + e_i)^2 - (x_i + e_i)^4], \tag{4.2}$$

where e_i is a uniformly distributed random variable in the range $[0.2, 0.4]$. Such a random function tests the algorithm's ability to locate the global optimum in the presence of noise. Figure 4.1 shows a one-dimensional plot of one instance of the quartic function. The quartic function is considered in the box $[-2, 2]^n$, $n \geq 2$; the global minimum occurs at a vertex of this box.

With respect to the proposed modifications of the DIRECT algorithm, three groups of experiments were conducted.

I. ϵ test

The ϵ test was designed to explore the sensitivity of DIRECT to the parameter ϵ [8]. Eight different ϵ values have been tested for evaluating the performance of DIRECT as shown in Table 4.1. For each test function, the stopping rule of a limit on the number of evaluations was set to ensure comparability between test cases in terms of the amount of work. 2000 and 300 evaluations were used for the Griewank function and the quartic function, respectively. Due to characteristics of the two functions, different performance measures were chosen. Table 4.1 shows that $|f_{\min}|$ is used for the Griewank function, which has its unique global minimum $\tilde{f} = 0$ at $\tilde{x} = 0$. Obviously, the closer $f_{\min}$ gets to $\tilde{f}$, the better the DIRECT algorithm performs. As for the quartic function, the random noise variable e_i makes it hard to find the true global optimum near the boundary. However, the global minimum falls around the vector $\tilde{x} = (2, \ldots, 2)$ at the boundary. Therefore, an alternative measure for convergence is taken as

$$\delta_{\tilde{x}} = \frac{||x_{\min} - \tilde{x}||}{||\tilde{x}||}, \tag{4.3}$$

where $x_{\min}$ is the computed optimal vector.

From the experimental results shown in Table 4.1, the DIRECT algorithm's behavior is different for the two test functions. For the Griewank function, a smaller ϵ gives a closer $|f_{\min}|$, while a larger ϵ seems to work better for the quartic function in terms of the smallest $\delta_{\tilde{x}}$. [6] conducted similar experiments and observed that the choice of the ϵ value depends on the characteristics of objective functions, such as the dimension of the problem n and the number of local and global minima.

Table 4.1. ϵ test results for stopping rule of a limit on the number of function evaluations (2000 for the Griewank function and 300 for the quartic function) for $n = 2$.

	Griewank Function		**Quartic Function**	
ϵ value	**evaluations**	$\lvert f_{\min}\rvert$	**evaluations**	$\delta_{\tilde{x}}$
0.01	2007	1.75E-006	301	7.38E-03
0.001	2013	1.75E-006	303	9.20E-03
0.0001	2001	2.16E-008	301	4.95E-02
0.00001	2031	2.16E-008	305	1.23E-02
0.000001	2043	2.66E-010	301	1.64E-02
0.0000001	2027	2.66E-010	305	2.77E-02
0.00000001	2023	3.29E-012	303	2.76E-02
0.0	2071	0.00	307	2.05E-02

II. "Aggressive switch" test

This test was intended for observing the effect of the "aggressive switch", which was first implemented in [2] to adapt to a parallel computing environment. Basically, the "aggressive" switch determines whether DIRECT performs the convex hull processing or not. With the switch on, it bypasses the procedure of finding the boxes on the lower right convex hull. Instead, it subdivides all boxes with the lowest function values in box sequences. Figure 4.2 compares the natural logarithms of the number of evaluations with the switch on and off for both functions as the problem dimension N increases from 2 to 28. The stopping rule is the limit on the number of iterations. As the problem dimension N grows, the number of evaluations is increasing with the switch either on (dotted) or off (solid). With the aggressive switch on, many more evaluation tasks are generated in each iteration. In a serial computing environment, aggressive switch off is preferred in order

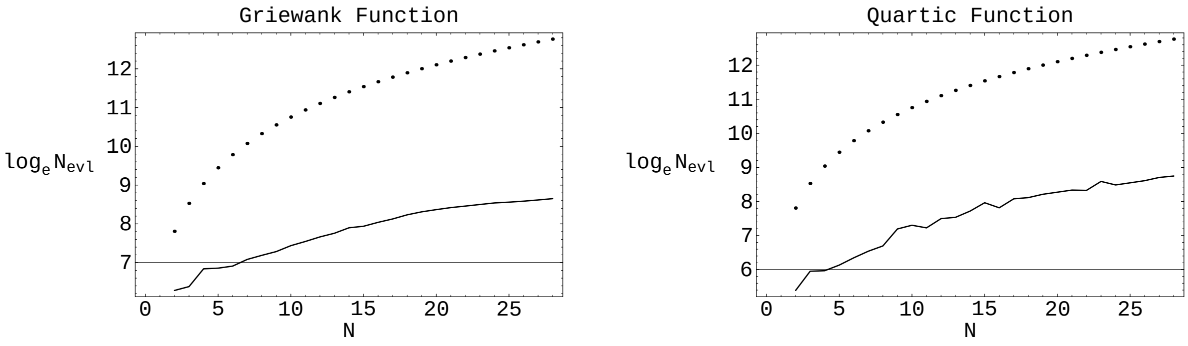

Figure 4.2. Dimension N vs. $\log_e(N_{evl})$ with aggressive switch off (solid) and on (dotted). N_{evl} is the number of function evaluations.

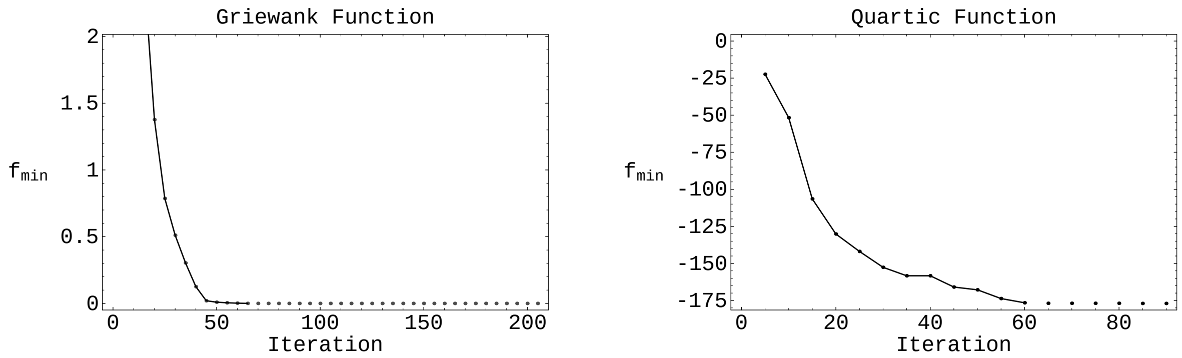

Figure 4.3. Change in computed $f_{\min}$ as DIRECT progresses for the Griewank function and the quartic function with objective function tolerance $= 0$ (dotted) and 0.0001 (solid), for $n = 20$, $\epsilon = 0$.

to reduce the workload of space partitioning. However, the switch is desired to be on to balance the workload for massively parallel multiprocessors. In that context, the switch on also speeds up locating the global optimum. Detailed experimental results and analyses can be found in [2].

III. Performance tests

Efficiency is one of the critical performance issues that the present implementation tends to emphasize. It involves several aspects, including the speed in locating the global minimum, the storage required, and the algorithm performance in the presence of noise.

Figure 4.3 shows the history of $f_{\min}$ for the 20-dimensional Griewank function and the quartic function. Both of them stop when the box holding the current $f_{\min}$ has reached the allowed minimum diameter, which is estimated to be at the round off level within the bounded design space. The similar trend, sharply decreasing at the beginning and leveling off at the end, motivates the implementation of the new stopping rule—objective function convergence tolerance—

$$\tau_f = \frac{\tilde{f}_{\min} - f_{\min}}{1.0 + \tilde{f}_{\min}}, \tag{4.4}$$

where $\tilde{f}_{\min}$ represents the previous computed minimum. The algorithm stops when τ_f becomes less than a user specified value. It avoids wasting function evaluations for small improvements,

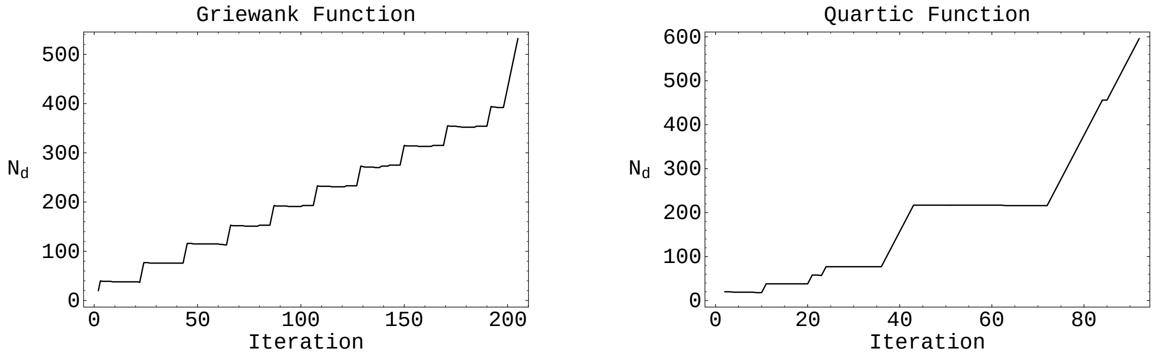

Figure 4.4. Change in number of distinct diameters N_d as DIRECT progresses for $n = 20$, $\epsilon = 0$.

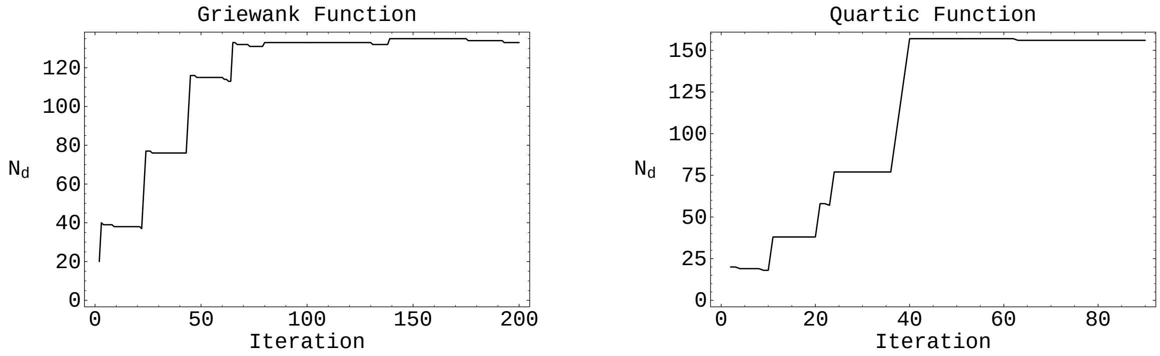

Figure 4.5. Change in number of distinct diameters N_d as DIRECT progresses for $n = 20$, $\epsilon = 0.0001$.

which are plotted as dotted tails in Figure 4.3. The definition of objective function convergence tolerance (4.4) differs from the percent error in [8], which is based on the knowledge of the true global optimum of the objective function, while τ_f measures the convergence with the current best estimate of the optimum. This is a reasonable stopping criterion for large-scale engineering design problems.

The required storage is directly related to two factors—the number of distinct diameters N_d and the length of box sequences L_b, which determine the memory occupied by the box structures `BoxMatrices` and `BoxLinks`. An interesting observation here is that ϵ plays a role in reducing the number of distinct diameters N_d. Figures 4.4 and 4.5 show the change in N_d with $\epsilon = 0$ and $\epsilon = 0.0001$. The case with $\epsilon = 0$ produces more distinct diameters since it always subdivides the box with $f_{\min}$, which also is the smallest one among all boxes on the lower right convex hull. In contrast, $\epsilon = 0.0001$ skips the leftmost part of the lower right convex hull as illustrated in Figure 2.1, thereby reducing the chances of generating new distinct diameters.

The changes in the maximum and average lengths of box sequences were tracked as DIRECT progressed. Figure 4.6 shows that the maximum box sequence length increases dramatically compared with the average one. Only a few box sequences are very long. This is the reason for using `BoxLinks` to extend the box sequences instead of only allocating `BoxMatrices` with a great number of rows, which would waste memory for short box sequences.

The extent to which the allocated memory is used depends on the problem, ϵ, and the stopping criteria. Figures 4.7 and 4.8 show the allocated and used memory, and how the relationship varies.

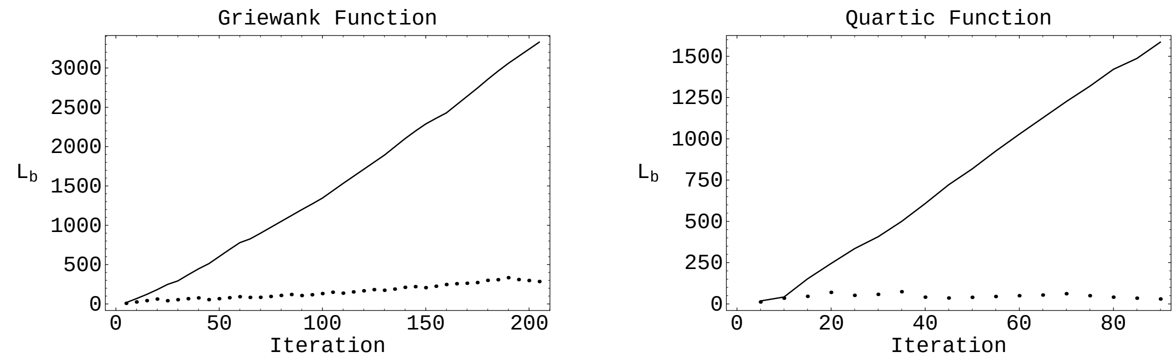

Figure 4.6. History of maximum (solid) and average (dotted) box sequence lengths for $n = 20$, $\epsilon = 0$.

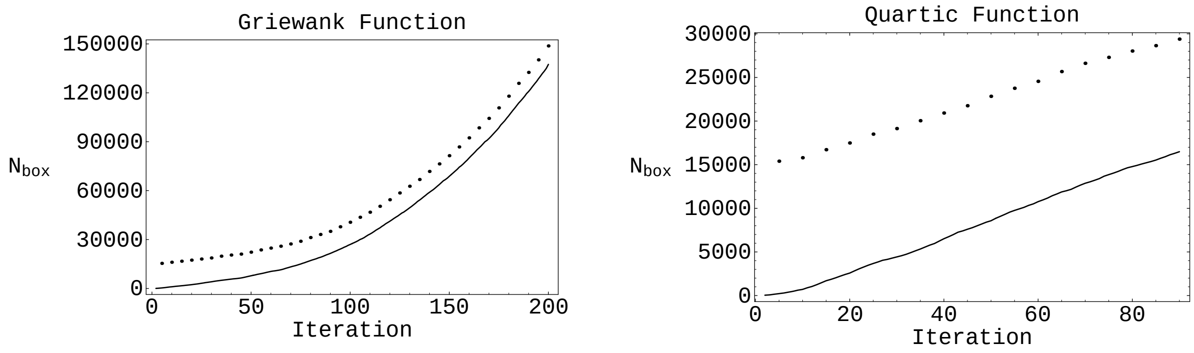

Figure 4.7. The number N_{box} of hyperboxes allocated (dotted) compared to the number of hyperboxes actually used (solid), as the iteration progresses, for $n = 20$, $\epsilon = 0$.

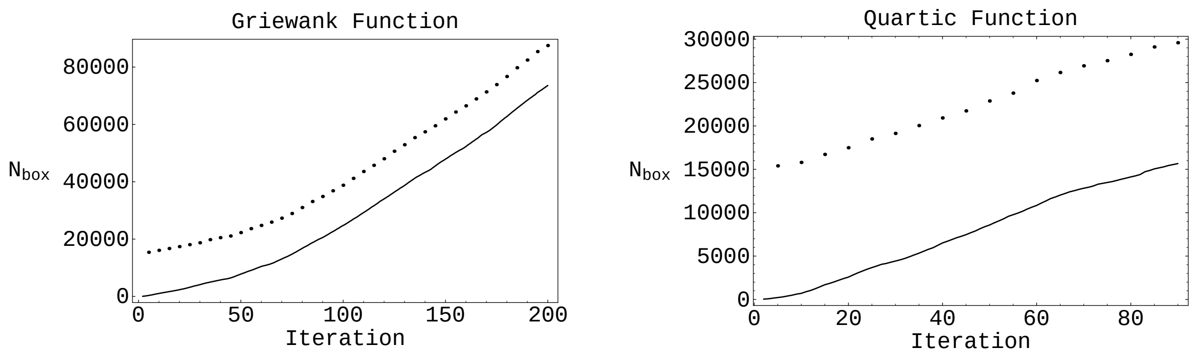

Figure 4.8. The number N_{box} of hyperboxes allocated (dotted) compared to the number of hyperboxes actually used (solid), as the iteration progresses, for $n = 20$, $\epsilon = 0.0001$.

For small numbers of iterations, much of the allocated memory can remain unused, but for large numbers of iterations (1000s), almost all the allocated memory can be used.

The last experiment tests the performance of DIRECT in the presence of different noise levels. In the sense of [3], the Griewank function is a quadratic function with noise added by including a cosine function. The noise level can be controlled by the parameter d. In Figure 4.9, the global minimum $\tilde{f} = 0$ of the Griewank function can be located within 2000 evaluations until d becomes

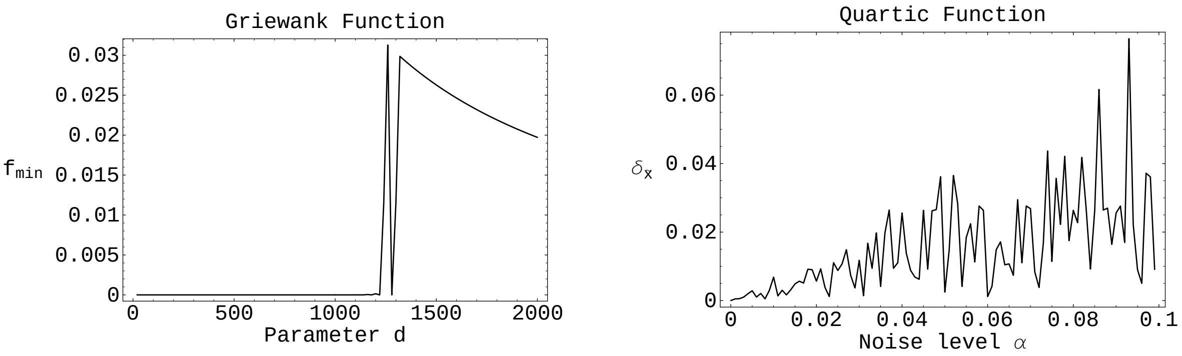

Figure 4.9. Test results for varying parameter values (parameter d for Griewank function and noise level α for the quartic function ($e_i \in [0.3-\alpha, 0.3+\alpha]$)) for stopping rule of a limit on number of the function evaluations (2000 for the Griewank function and 300 for the quartic function) with $n=2$, $\epsilon=0$.

greater than 1200. More function evaluations would be needed for higher noise levels. The $\delta_{\tilde{x}}$ of the quartic function has increasing fluctuations as the noise level α increases, reflecting the impossibility of locating the minimum with small signal to noise ratios.

5. S^4W optimization design

S^4W(Site-Specific System Simulator for Wireless system design) is an integrated wireless communication system design software tool being developed jointly by the Mobile & Portable Radio Research Group (MPRG) and the problem solving environment (PSE) research group at Virginia Polytechnic Institute & State University. It proposes a built-in optimization loop to maximize the efficiency of channel models and surrogate functions to reduce the cost of propagation model simulations. An example S^4W system model consisting of a propagation model, a channel model, and an optimizer is illustrated in [13]. There are a wide variety of optimization problems inherent in this project. As a typical “black box” engineering design problem, the application of DIRECT to base station placement is presented here.

Generally, base station placement optimization is aimed at covering a geographical area of interest (indoor and outdoor environments are modeled differently), to a specified minimum power level (threshold) at a minimum cost [5]. The system considered here is an indoor system, located on the fourth floor of a building on campus. In such a building, performance of wireless networks (e.g., wireless LAN) is affected by signal attenuation, reflection, interference, and multipath propagation. Given transceiver frequency/power, a threshold power level, a signal-to-noise ratio, and the number of base stations, the optimization variables are the coordinates of the base stations. Consider the placement of a single base station, with three variables x, y, and z. Based on the chosen design variables, an appropriate objective function is needed to determine the best system configuration. In [5], “coverage” (ratio of the number of grid points with received power above the threshold to the total number of grid points) is taken as the objective function for the Nelder-Mead simplex method, another type of multidirectional search method [12]. However, this ratio is discontinuous, so the objective function here is constructed as the (Lipschitz continuous) function

$$f(x,y,z) = \sum_{i=1}^{M} \left(T - P_i(x,y,z)\right)_+^2, \tag{5.1}$$

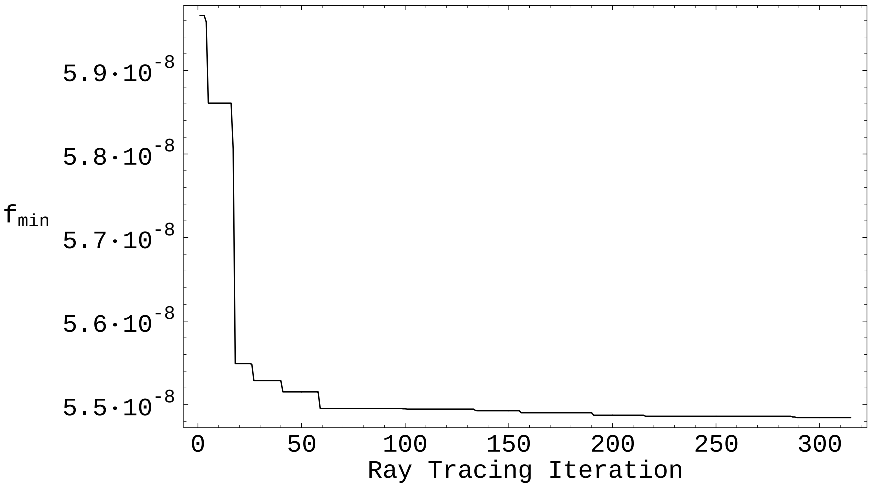

Figure 5.1 Change in evaluated $f_{\min}$ by the ray tracing simulation for the function (5.1). Power level threshold $= -55$dB.

where T is the given power level threshold, P_i is the power received at the ith receiver with the single base station located at (x, y, z), and M is the total number of receivers. A lower value of f indicates a better "coverage". The continuity of $P_i(x, y, z)$ depends on the wave (electromagnetic radiation) propagation model.

The function evaluation was done using a MPI-based parallel ray tracer running on an 80 node Athlon 650 Beowulf cluster of workstations [13]. The environment is approximated by triangular facets, multipath reflections are modeled but not transmission through solid objects or diffraction around sharp edges, a very efficient octree data structure is used, and rays are uniformly distributed in the solid angle covering the system domain. An omnidirectional antenna pattern is assumed to produce the power delay profile (PDP) used in calculating the received power at each point on a grid laying at an arbitrary height in the specified environment. The information exchanged between the ray tracer and the optimizer consists of the coordinates of the base station and the objective function value. Figure 5.1 shows the simulation results of 315 objective function evaluations. $f_{\min}$ has the same fast drop at the beginning as the ones shown in Figure 4.3. It takes about 60 evaluations to reduce the objective function by 7.88% (from 5.96E-08 to 5.49E-08), and another 255 evaluations to improve it by only 0.18% (from 5.49E-08 to 5.48E-08). This supports the earlier claim that objective function convergence tolerance (4.4) is a reasonable stopping criterion for the DIRECT algorithm in practice.

6. Conclusions and future work

The main contribution of the present implementation of DIRECT is the design of the dynamic data structures. They not only address efficiently the problem of unpredictable memory requirements in large-scale engineering optimization, but also simplify key steps of the DIRECT algorithm for identifying potentially optimal boxes. In addition, the proposed modifications in stopping criteria and box selection rules have shown great value in adapting DIRECT to varying types of objective functions and design goals. Future work includes a MPI-based parallel version using the dynamic data structures (or suitable variants of them) proposed here. Both a master-slave version (for moderately parallel system or low dimensional problems) and a distributed control version (for massively parallel systems and higher dimensional ($n > 30$) problems) are likely to find practical applications. Incorporating nonlinear constraints has been attempted by several authors (Jones [9], Torczon), but the issue is by no means satisfactorily resolved.

Acknowledgment

This work was supported in part by NASA Grant NAG-2-1180, NSF Grant DMI-9979711, and NSF Grant EIA-9974956.

References

1. C. A. Baker, L. T. Watson, B. Grossman, R. T. Haftka, and W. H. Mason, "Parallel global aircraft configuration design space exploration", in Proc. High Performance Computing Symposium 2000, A. Tentner (Ed.), Soc. for Computer Simulation Internat, San Diego, CA, 2000, pp. 101–106.
2. C. A. Baker, "Parallel global aircraft configuration design space exploration", Technical Report MAD 2000-06-28, Virginia Polytechnic Institute and State University, Blacksburg, VA, 2000.
3. S. E. Cox, R. T. Haftka, C. Baker, B. Grossman, W. H. Mason, L. T. Watson, "Global multidisciplinary optimization of a high speed civil transport", in Proc. Aerospace Numerical Simulation Symposium '99, Tokyo, Japan, June, 1999, pp. 23–28.
4. S. Cox, R.T. Haftka, C. Baker, B. Grossman, W. Mason and L. T. Watson, "A comparison of optimization methods for the design of a high speed civil transport", Journal of Global Optimization, 2001. To appear.
5. S. J. Fortune, D. M. Gay, B. W. Kernighan, O. Landron, R. A. Valenzuela, and M. H. Wright (AT&T Bell Laboratories), "WISE design of indoor wireless systems: practical computation and optimization", IEEE Computational Science & Engineering, vol. 2, no. 1, pp. 58–68, Spring, 1995.
6. J. M. Gablonsky and C. T. Kelley, "A locally-biased form of the DIRECT algorithm", Technical Report CRSC-TR00-31, Center for Research in Scientific Computation, North Carolina State University, Raleigh, NC, 2001.
7. J. M. Gablonsky, "An implementation of the DIRECT algorithm", Technical Report CRSC-TR98-29, Center for Research in Scientific Computation, North Carolina State University, Raleigh, NC, 1998.
8. D. R. Jones, C. D. Perttunen, and B. E. Stuckman, "Lipschitzian optimization without the Lipschitz constant", Journal of Optimization Theory and Applications, vol. 79, no. 1, pp. 157–181, 1993.
9. D. R. Jones, "The DIRECT global optimization algorithm", Encyclopedia of Optimization, Kluwer Academic Publishers:Boston. To appear.
10. R. M. Lewis, V. Torczon, and M. W. Trosset, "Direct search methods: then and now", Journal of Computational and Applied Mathematics, vol. 124, pp. 191–207, 2000.
11. J. D. Pinter, *Global Optimization In Action*, Kluwer Academic Publishers:Boston, 1996.
12. V. Torczon, "On the convergence of the multidirectional search algorithm", SIAM Journal on Optimization, vol. 1, no. 1, pp. 123–145, 1991.
13. A. Verstak, M. Vass, N. Ramakrishnan, C. Shaffer, L. T. Watson, K. K. Bae, J. Jiang, W. H. Tranter, and T. S. Rappaport, "Lightweight data management for compositional modeling in problem solving environments", in Proc. High Performance Computing Symposium 2001, A. Tentner (ed.), Soc. for Modeling and Simulation Internat., San Diego, CA, 2001, pp. 148–153.
14. L. T. Watson and C. A. Baker, "A fully-distributed parallel global search algorithm", Engineering Computations, vol. 18, no. 1/2, pp. 155–169, 2001.
15. L. T. Watson, M. Sosonkina, R. C. Melville, A. P. Morgan, and H. F. Walker, "Algorithm 777: HOMPACK90: A suite of FORTRAN 90 codes for globally convergent homotopy algorithms", ACM Transactions on Mathematical Software, vol. 23, pp. 514–549, 1997.