GLOBAL OPTIMIZATION OF TRANSMITTER PLACEMENT FOR INDOOR WIRELESS COMMUNICATION SYSTEMS

by

Jian He

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

APPROVED:

Layne T. Watson
Chair of Advisory Committee

Calvin J. Ribbens

Eunice Santos

August 22, 2002
Blacksburg, Virginia

Key words: Bit error rate, DIRECT algorithm, direct search, dynamic data structures, global optimization, power coverage, transmitter placement.

Copyright 2002, Jian He

GLOBAL OPTIMIZATION OF TRANSMITTER PLACEMENT FOR INDOOR WIRELESS COMMUNICATION SYSTEMS

by

Jian He

(ABSTRACT)

The DIRECT (DIviding RECTangles) algorithm [23], a variant of Lipschitzian methods for bound constrained global optimization, has been applied to the optimal transmitter placement for indoor wireless systems. Power coverage and BER (bit error rate) are considered as two criteria for optimizing locations of a specified number of transmitters across the feasible region of the design space. The performance of a DIRECT implementation in such 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 because of unpredictable memory requirements. This is especially critical in S^4W (Site-Specific System Simulator for Wireless communication systems), where the DIRECT optimization is just one small component connected to a parallel 3D propagation ray tracing modeler running on a 200-node Beowulf cluster of Linux workstations, and surrogate functions for a WCDMA (wideband code division multiple access) simulator are also used to estimate the channel performance. Any component failure of this large computation would abort the entire design process. To make the DIRECT global optimization algorithm efficient and robust, 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 is on design issues of the dynamic data structures, related memory management strategies, and application issues of the DIRECT algorithm to the transmitter placement optimization for wireless communication systems. Results for two indoor systems are presented to demonstrate the effectiveness of the present work.

ACKNOWLEDGEMENTS

I would like to express my appreciation for the support and guidance that I received from my advisor Dr. Layne T. Watson in the past two years. Also, I want to thank the other two members of my advisory committee, Dr. Calvin J. Ribbens and Dr. Eunice Santos, for their sincere support and advice.

Furthermore, I would like to thank everyone in the S^4W research group. Especially, I thank Alex Verstak and Chris A. Anderson for their work on ray tracing modeling and site measurements (documented in Chapter 5.1). Moreover, I want to give my thanks to Kyung Bae, Jing Jiang, and Cheryl A. Stinson for their work on WCDMA (wideband code division multiple access) simulations and the corresponding surrogate MARS (multivariate adaptive regression splines) modeling (documented in Chapter 5.2). I would also thank Dr. Naren Ramakrishnan, Dr. Clifford A. Shaffer, Dr. William H. Tranter, and Dr. Theodore S. Rappaport for their effort in organizing research meetings to help students collaborate with each other. In this project, many practical optimization applications were provided for exploring this global search algorithm—DIRECT. Finally, I would like to thank Dr. Srinidhi Varadarajan for his administration on the 200 node AMD Athlon Beowulf cluster at the Laboratory for Advanced Scientific Computing and Applications (LASCA). Without such computing power, this work would not be possible.

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

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1: INTRODUCTION

Optimal transmitter placement provides high spectral efficiency and system capacity while reducing network costs, which are the key criteria for wireless network planning [5]. As the complexity and popularity of modern wireless networks increases, automatic transmitter placement provides cost savings when compared to the traditional human process of site planning. Automatic design tools are being developed to offer efficient and optimal planning solutions. Besides [11], [20], and [28], S^4W (Site-Specific System Simulator for Wireless system design) is among the few known wireless system tools for in-building network design. It is 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. An optimization loop in S^4W is proposed to maximize the efficiency of simulated channel models and surrogate functions are proposed to reduce the cost of simulations. Transmitter placement optimization is one specific problem that can be solved by S^4W. An example of an S^4W model consisting of a propagation model, a wireless system model, and an optimizer is given in [34]. The underlying optimization algorithm is known as DIRECT (DIviding RECTangles), a direct search algorithm proposed by Jones et al. [23] as an effective approach to solve global optimization problems subject to simple constraints. The present thesis work includes research work published in [1], [18], [19], and [33].

1.1 Application Background

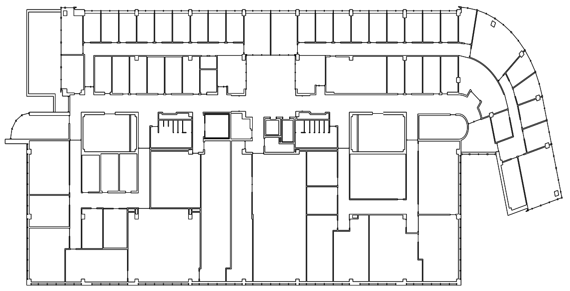


Figure 1.1. Durham Hall, fourth floor.

In general, transmitter placement optimization is aimed at ensuring an acceptable level of wireless system performance within a geographical area of interest (Figure 1.1 shows an indoor environment for the present study) at a minimum cost. [11] considers the major performance factor to be the power coverage, defined as the ratio of the number of receiver locations with received power above an assumed threshold to the total number of receiver locations. This nonsmooth function leads to the rank based methods used by [11]. In [20] and [31], the objective function is based on several weighted factors, such as covered area, interference area, and mean signal path loss. [6] proposes a QoS (Quality of Service)-based penalty function resulting in an unconstrained optimization problem. In the present work, two performance metrics form objective functions for optimal transmitter placement. The metrics are continuous penalty functions defined in terms of power levels (i.e., power coverage) and bit error rates at given receiver locations within the covered region. Both objective functions are devised to minimize the average shortfall of the estimated performance metric with respect to the corresponding threshold. 3D ray tracing is used as a deterministic propagation model to estimate power coverage levels and impulse responses within the region of interest for transmitter locations sampled by the optimization algorithm [29][30]. Surrogates for the Monte Carlo WCDMA simulation are used to estimate the BERs (bit error rates) for the second optimization criterion. Both the surrogates and the WCDMA simulation utilize the impulse responses estimated by the ray tracing model. Since 3D ray tracing and WCDMA simulation are computationally expensive, MPI-based parallel implementations are used in the present work.

1.2 Algorithm Introduction

DIRECT was named after one of its key steps—dividing rectangles. It is a pattern search method that is categorized as a direct search technique by Lewis et al. [24]. 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" [24], 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. 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. These features have motivated its successful application in modern large-scale multidisciplinary engineering problems [2], [4], and [35].

Nevertheless, DIRECT does have limitations as pointed out by Jones [22]. Some applicability concerns include: (1) the space-partitioning strategy in practice limits the algorithm to low-dimensional problems (≤ 20), although Baker et al. [3] 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, the present work 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 (e.g., [2], [8], and [15]) 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.

1.3 Organization

Chapters are organized as follows. Chapter 2 begins with an overview of the DIRECT algorithm followed by the proposed modifications. Chapter 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 Chapter 4, numerical results and performance analyses for four sample objective functions are presented. Chapter 5 addresses issues related to the transmitter placement optimization problem for indoor wireless systems, involving the parallel 3D ray tracing modeling, the parallel WCDMA simulation, the surrogate fitting, and the objective formulation. Optimization results for optimizing transmitter locations in terms of both power coverage and bit error rate are presented and analyzed at the end of Chapter 5. Finally, Chapter 6 summarizes some key contributions of the present work and suggests directions for future research.

Chapter 2: DIRECT ALGORITHM

DIRECT is aimed at solving global optimization problems (GOP) subject to simple bounds. The general problem statement is [26]

$$\min_{x \in D} f_0(x) \tag{2.1}$$

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

where $D_0 = \left\{x \in E^n \mid \ell \leq x \leq u\right\}$ 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

$$|f_j(x_1) - f_j(x_2)| \leq L_j \|x_1 - x_2\|, \quad \forall x_1, x_2 \in D_0. \tag{2.2}$$

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 [26]. Furthermore, the convergence of Lipschitz-based global optimization algorithms can be easily proved by assuming the knowledge of a Lipschitz constant [23]. 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. [23] 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.

2.1 Overview

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 [23]. 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 [23].

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 $t = 0$.

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.

[23] 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 [23].

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 $i = 1, \ldots, m$,

$$f(c_j) - \tilde{K} d_j \leq f(c_i) - \tilde{K} d_i, \tag{2.3}$$

$$f(c_j) - \tilde{K} d_j \leq f_{\min} - \epsilon |f_{\min}|. \tag{2.4}$$

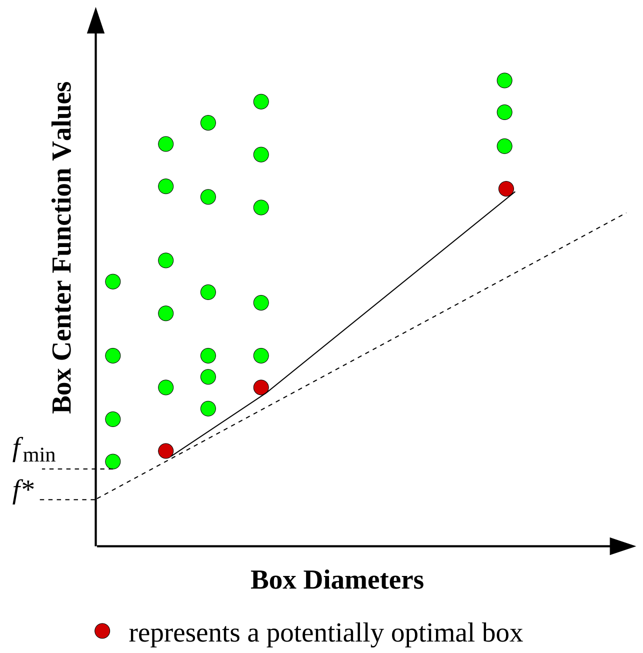

Box Center Function Values
$f_{\min}$
f^*
Box Diameters
● represents a potentially optimal box

Figure 2.1. Illustration of potentially optimal boxes on convex hull with ϵ test from [22]. Note that $f^* = f_{\min} - \epsilon|f_{\min}|$. Potentially optimal boxes are on the lower-right convex hull.

Figure 2.1 represents the set of boxes as points in a plane. The first inequality (2.3) 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 (2.4) 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 (2.4). This is illustrated by the dotted line in Figure 2.1.

2.2 Modifications

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 [22]. In Step 4 of 2.1, 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 (e.g., random selection or priority by age). 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. [16] 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 [35]. 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 Chapter 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.*

Chapter 3: DYNAMIC 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 can be 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. Within a column, shifting operations on boxes in time $O(m)$ are required to keep their function values sorted. A more efficient approach is to implement each column with a heap data structure, which is a typical priority queue that replaces shifting operations with sift down operations in time $O(\log_2 m)$.

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

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

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.

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 [23], 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

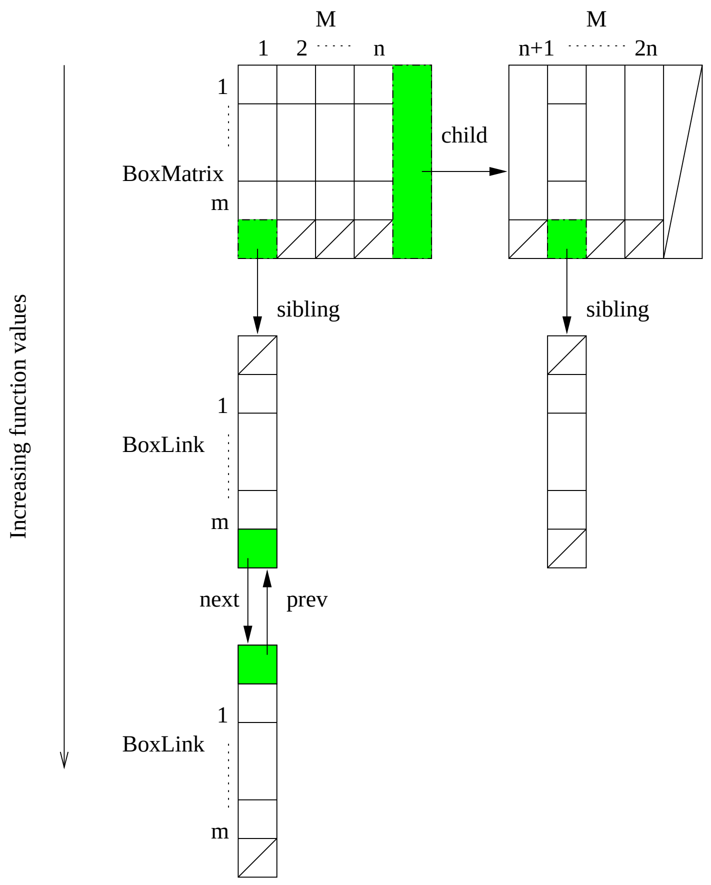


Figure 3.1. Box structures comprised of `HyperBox`es.

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, 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

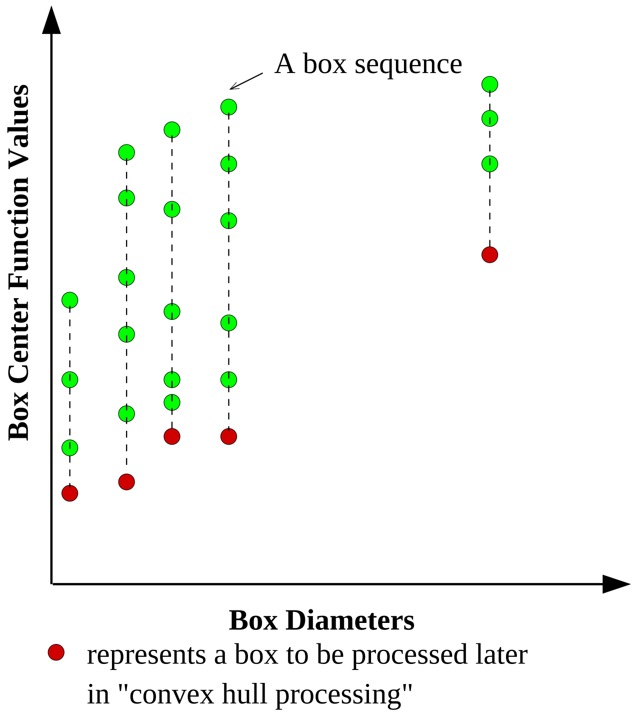


Figure 3.2. Scatter plot pattern.

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**
        $s := \frac{f_i - f_k}{d_i - d_k}$
        **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** $\bar{s} < \frac{f_i - (f_{\min} - \epsilon|f_{\min}|)}{d_i}$ **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 manipulating the

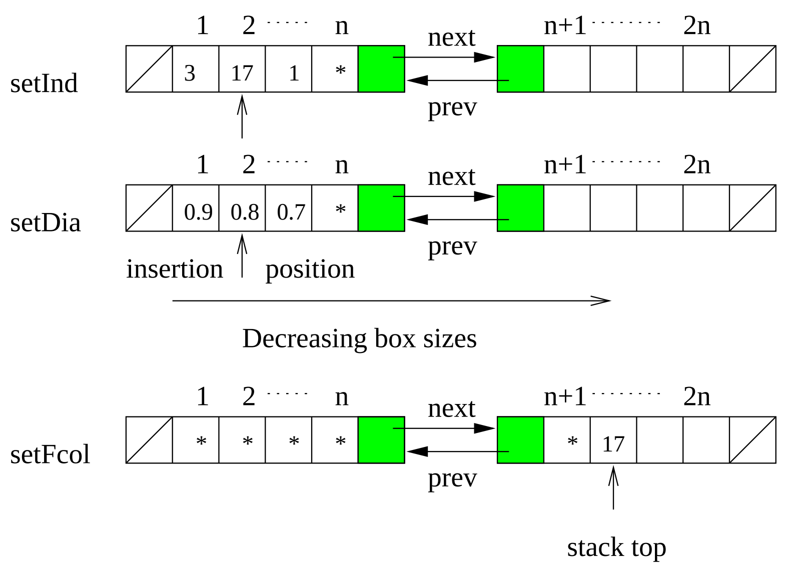


Figure 3.3. Linked list structures. Insertion of a new box size (0.8) has four steps: (a) request a free column index (17) from the stack top of `setFcol` by popping the stack, (b) locate the insertion position (2) in `setDia` and insert the new diameter (`setDia` is shown after the insertion), (c) add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

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 box size sequences. Some heap sifting 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 ([36]) 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

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

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.

Chapter 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 [7] are chosen here to study the behavior of the DIRECT algorithm and evaluate the performance of this implementation.

The n-dimensional Griewank function [7]

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

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 numerical results here are for an initial box $[-40, 60]^n$ and $d = 500$.

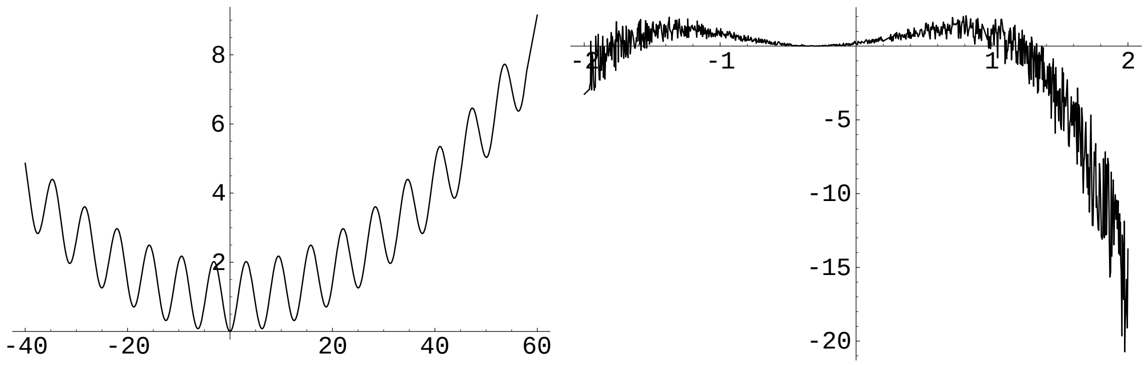


Figure 4.1. One-dimensional Griewank function with parameter $d = 500$ (left), and one-dimensional noisy quartic function (right).

The second test function is an n-dimensional quartic function with a random noise variable defined by [7]

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

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, four groups of experiments were conducted.

4.1. ϵ Test

The ϵ test was designed to explore the sensitivity of DIRECT to the parameter ϵ [23]. 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 $\delta_{\tilde{x}}$. [16] 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**	$|f_{\min}|$	**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

4.2. "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 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].

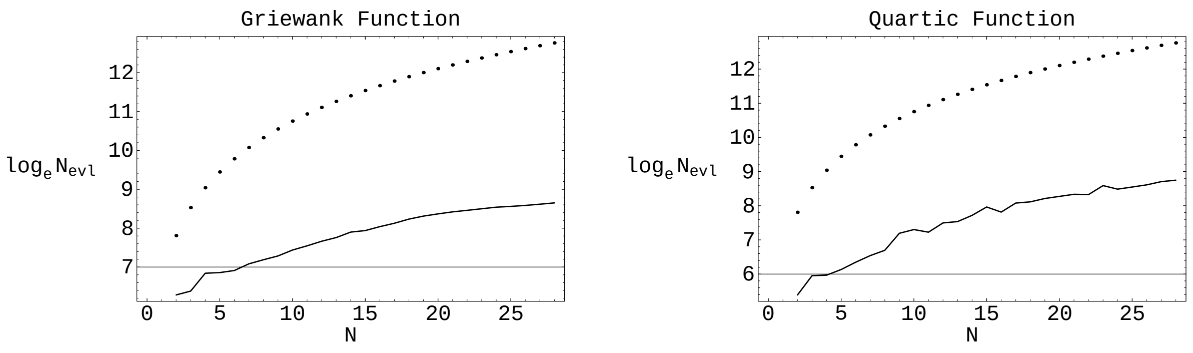


Figure 4.2. Dimension N vs. $\log_e(N_{evl})$ with aggressive switch off (solid) and on (dotted). N_{evl} is the number of function evaluations.

4.3. 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, 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 [23], 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. Note that the stopping criterion (4.4) results in premature termination if $\tau_f \approx 0$ early in the iterations. Such a failure is easily recognized, though, by the size of the final box containing the minimizing point.

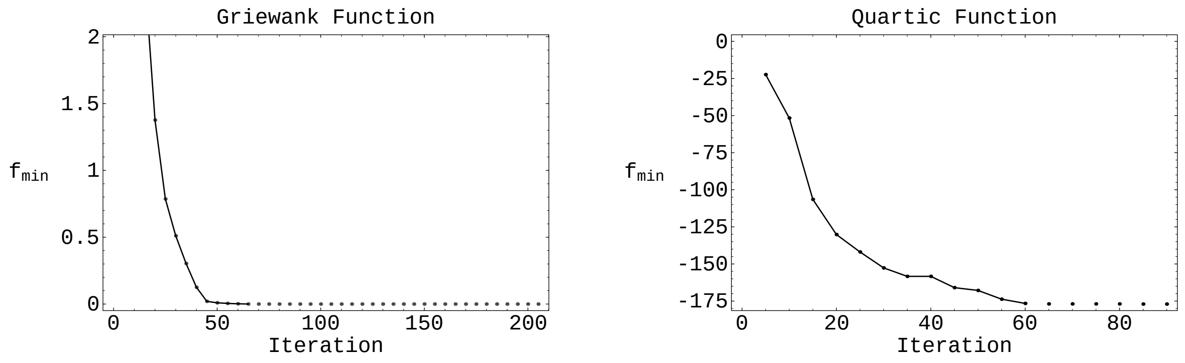


Figure 4.3. Change in computed $f_{\min}$ as DIRECT progresses for the Griewank function and the quartic function with objective function tolerance = 0 (dotted) and 0.0001 (solid), for $n = 20$, $\epsilon = 0$.

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

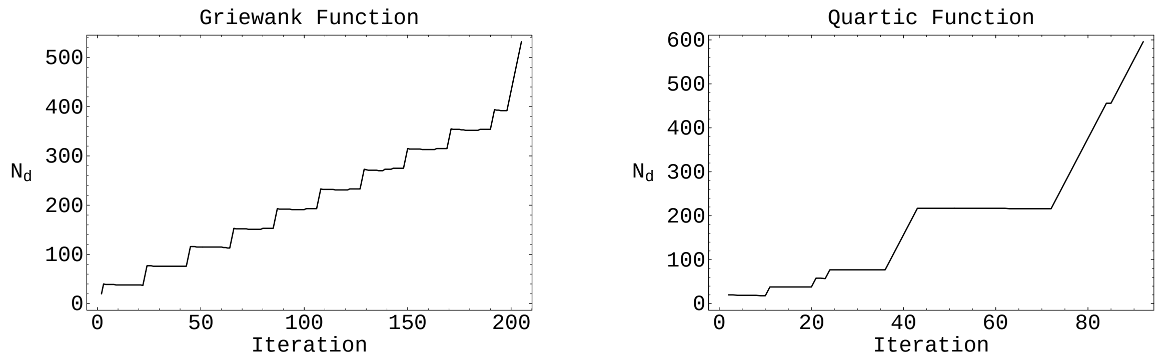


Figure 4.4. Change in number of distinct diameters N_d as DIRECT progresses for $n = 20$, $\epsilon = 0$.

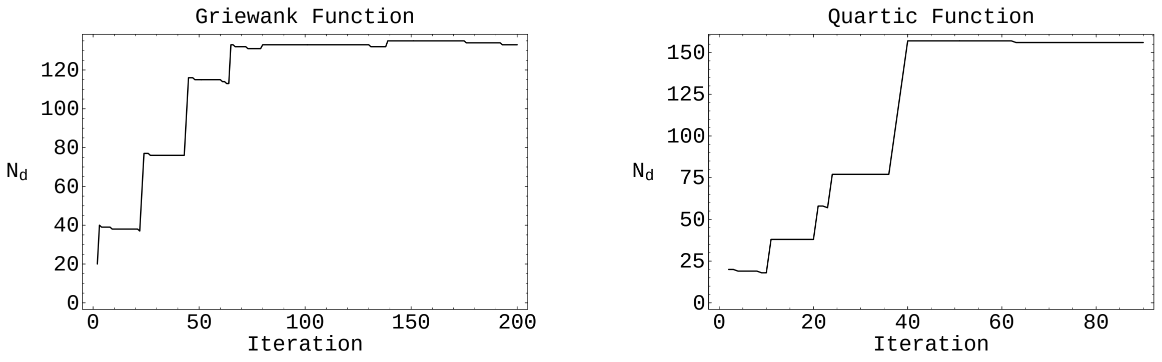


Figure 4.5. Change in number of distinct diameters N_d as DIRECT progresses for $n = 20$, $\epsilon = 0.0001$.

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. 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 next experiment tests the performance of DIRECT in the presence of different noise levels. In the sense of [7], 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 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

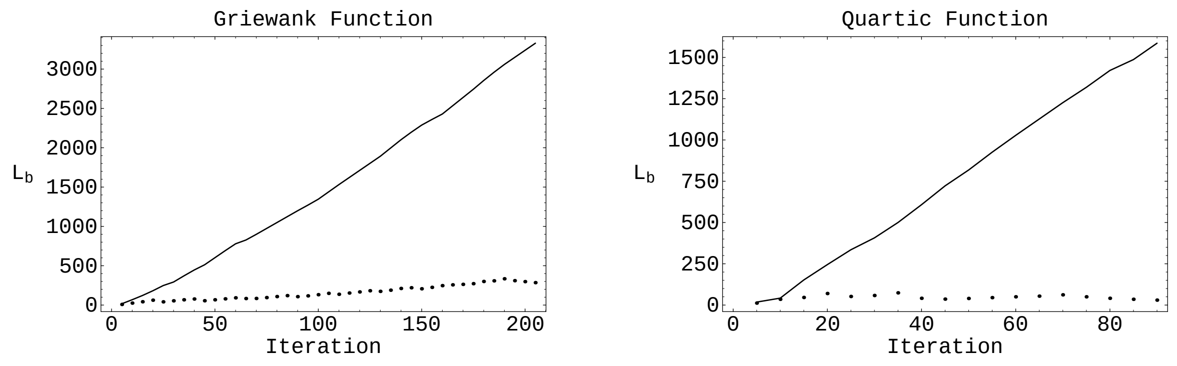


Figure 4.6. History of maximum (solid) and average (dotted) box sequence lengths for $n = 20$, $\epsilon = 0$.

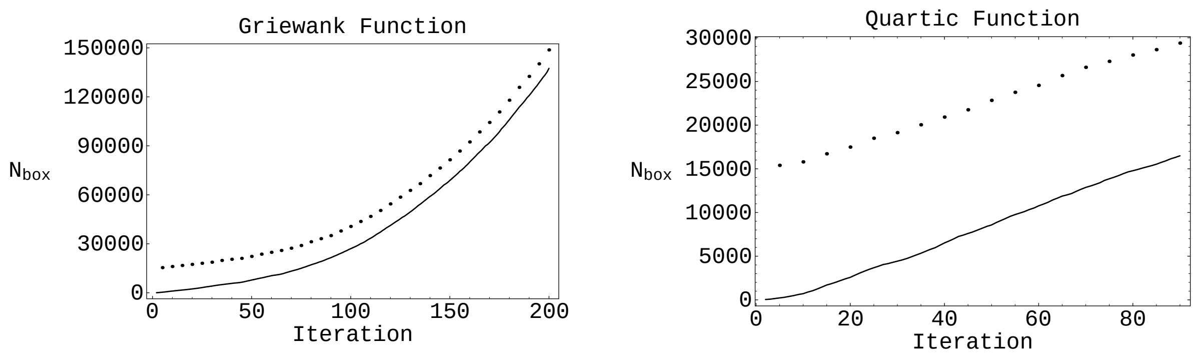


Figure 4.7. The number N_{box} of hyperboxes allocated (dotted) compared to the number of hyperboxes actually used (solid), as the iteration progresses, for $n = 20$, $\epsilon = 0$.

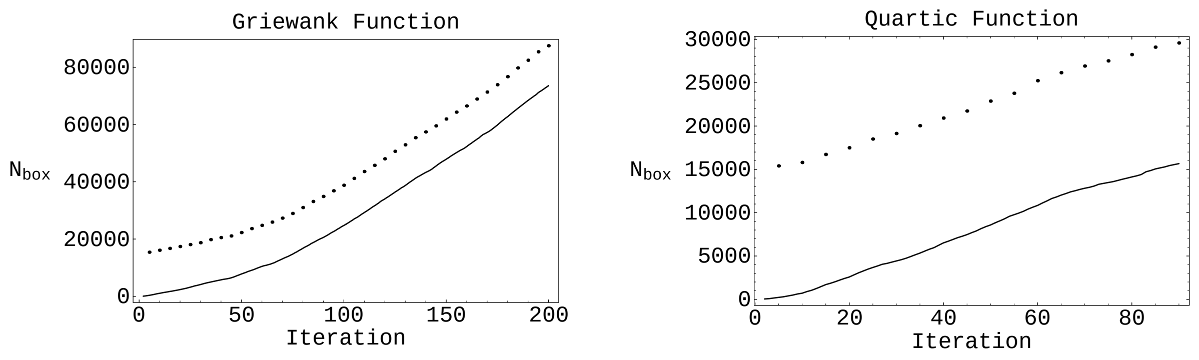
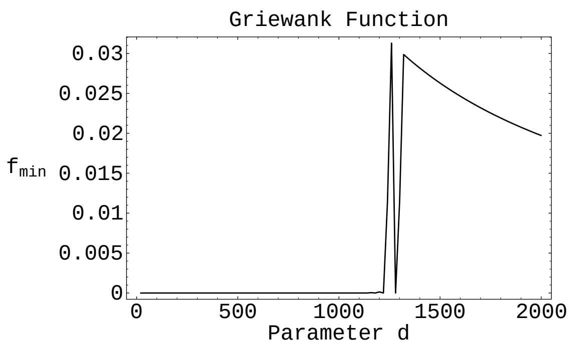


Figure 4.8. The number N_{box} of hyperboxes allocated (dotted) compared to the number of hyperboxes actually used (solid), as the iteration progresses, for $n = 20$, $\epsilon = 0.0001$.

the noise level α increases, reflecting the impossibility of locating the minimum with small



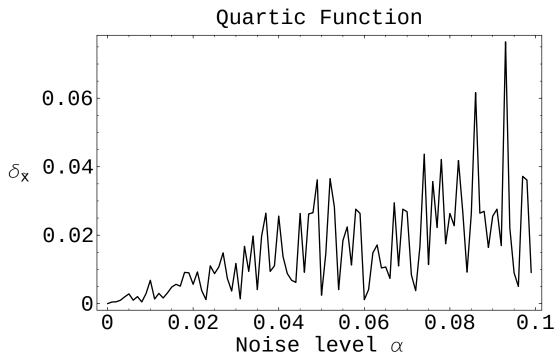


Figure 4.9. Test results for varying parameter values (parameter d for Griewank function and noise level α for the quartic function ($e_i \in [0.3 - \alpha, 0.3 + \alpha]$)) for stopping rule of a limit on number of the function evaluations (2000 for the Griewank function and 300 for the quartic function)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

Table 4.2. Comparison of static and dynamic implementations with `BoxMatrix` column dimension $n_c = 2n$, problem dimension n, L iterations, $\epsilon = 0$.

			Baker [2]		Gablonsky [15]		dynamic structures	
Problem	n	L	time	memory	time	memory	time	memory
Griewank	2	50	172	10264	34	2224	85	1040
Griewank	5	50	199	11504	34	2352	73	1024
Griewank	10	50	310	15424	51	2648	110	1616
Griewank	15	50	639	18280	88	3232	192	2744
Griewank	20	50	*	*	170	4464	397	6080
Griewank	50	70	*	*	*	*	6161	82664
Quartic	2	50	108	10240	26	2176	25	520
Quartic	5	50	151	11488	31	2240	27	528
Quartic	10	50	441	15432	36	2472	58	1160
Quartic	15	50	1260	18336	54	2992	125	2176
Quartic	20	50	*	*	82	3872	240	4560
Quartic	50	90	*	*	*	*	6572	86656

signal to noise ratios.

4.4. Comparison with Static Allocation Programs

Table 4.2 compares two static data structure implementations of the DIRECT algorithm, [2] and [15], with the dynamic data structure implementation proposed here. The test problems are the same two used throughout this section. Execution time is reported in milliseconds, and the memory usage reported is the maximum working set size in pages (1 page = 512 bytes). This number precisely reflects the virtual memory required by the program during execution. Not surprisingly, static implementations can execute much faster, until paging of the large static structures dominates the time. As Table 4.2 shows, the difference in memory requirements can be substantial. Of course, if a DIRECT code is being used inside a larger scientific computation, there is no contest in terms of robustness. The dynamic code described here will always return with something useful, whereas a statically allocated code will simply fail when it exhausts its memory allocation. The results in Table 4.2 used `BoxMatrix` column dimension $n_c = 2n$, which produced better results than the earlier mentioned value of $n_c = 35n$ derived from a large ensemble of experiments. An asterisk in the table indicates that the code failed with an execution exception.

Chapter 5: S^{4}W DESIGN OPTIMIZATION

This chapter briefly describes the other three major components for solving the optimal transmitter placement problem for indoor wireless communication systems. They are propagation modeling (5.1), channel modeling (5.2), and the objective function formulation (5.3). The optimization experiments with the present DIRECT implementation are presented in 5.4.

5.1 Ray Tracing Propagation Model

Received impulse responses are approximated with a 3D ray tracing propagation model that is based on geometrical optics. Electromagnetic waves are modeled as rays that are traced through reflections and transmissions through the walls. Beams [10] are shot from geodesic domes drawn around transmitters. Each beam is a triangular pyramid formed by the point location of the transmitter and one of the triangles on the surface of the dome. Essentially, the spherical wavefront is triangulated and the 3D sphere is split into pyramidal beams. Following the argument in [30], all such beams are disjoint and have nearly the same shape and angular separation. Only the central ray of each beam is traced to identify reflection locations. However, the whole beam is used for ray-receiver intersection tests. Once an intersection with a receiver location is detected, a ray will be traced back from the receiver to the transmitter through the sequence of reflections and transmissions (penetrations) encountered by the beam. The illustration of this process in 2D is given in Figure 5.1. Figure 5.2 depicts a fast intersection test of a beam with a grid of receiver locations. Neither diffraction nor scattering are modeled for computational complexity reasons, although these phenomena play an important role in propagation [27]. Octree space partitioning [17] and image parallelism with dynamic scheduling [12] are used to reduce simulation run time.

Although material parameters and incidence angles affect losses in a wireless channel, a constant 6 dB reflection loss (same as in [29]) and a constant 4.6 dB transmission (penetration) loss (the loss for plaster board in [9]) are assumed. The power contribution of each ray, in dBW, is calculated according to the model developed in [30]:

$$P_j = P(d_0) - 20\log_{10}(d/\lambda) - nL_r - mL_t, \tag{5.1}$$

where P_j is the power of the j-th ray, d is the total distance traveled by the ray, $P(d_0)$ is the transmitter power at a reference distance d_0 from the transmitter, n and m are the numbers of reflections and transmissions, $L_r = 6$ dB and $L_t = 4.6$ dB are reflection and transmission losses, and λ is the wavelength.

The ray tracer has been validated and calibrated with a series of measurements in the corridor of the fourth floor of Durham Hall, Virginia Tech. An ultrawideband sliding

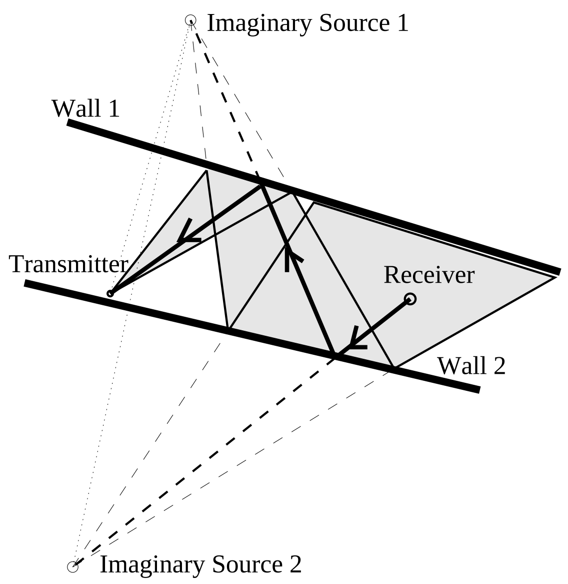


Figure 5.1. 2D beam tracing: a beam (shadowed region) is traced from the transmitter location to the receiver location through two reflections, and then a ray (bold line) is traced back.

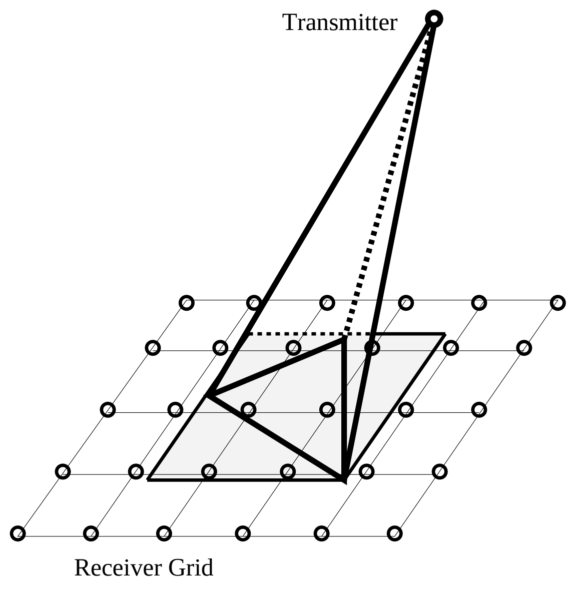


Figure 5.2. Beam intersection with a receiver grid: only the locations inside of the bounding box of the projection of the beam onto the grid (shadowed region) are tested for intersection with the beam pyramid.

correlator channel sounder [27] operating at 2.5 GHz and outfitted with omnidirectional antennas was used to record impulse responses at six separate locations. The sliding correlator utilized an 11-bit, 400 MHz pseudo-noise spreading code for a time domain multipath resolution of 2.5 nanoseconds and a dynamic range of 30 dB. Simulated power delay profiles were post-processed and compared to the measured ones location by location.

Comparing ray tracer output with a physical channel requires accounting for antennas and resampling the signal to match the sampling rate of the measurement system. The same conversion sequence was used for both validation against measurements and interfacing with the WCDMA simulation. The received E-field envelope of ray j (in V/m) that arrived at time t_j is $E_j = \sqrt{\eta 10^{0.1 P_j}}$, where P_j is the output of the ray tracer (in dBW) and $\eta = 120\pi\ \Omega$ is the impedance of free space [27]. To account for antenna directivity, an omnidirectional antenna pattern must be applied to all E_js. The electric field that would be registered at time t_j by a hypothetical measurement system with infinite bandwidth resolution is

$$E'_j = E_j G_t G_r \cos\Theta_t \cos\Theta_r, \tag{5.2}$$

where Θ_t and Θ_r are ray transmission and reception elevation angles relative to the horizon, and G_t and G_r are maximum transmitter and receiver antenna gains, respectively. Further, the discrete impulse response must be convolved with a Gaussian filter and sampled at uniform time intervals of width δ. The measurement system output samples with $\delta = 1$ ns while the WCDMA simulation used chip width $\delta \approx 260$ ns. The measured electric field E_k^m of bin k centered at time $k\delta$ is

$$E_k^m = C \sum_{j=1}^{Q} E'_j e^{i\phi_j} \int_{t_j - k\delta - \delta/2}^{t_j - k\delta + \delta/2} e^{-\tau^2/(2\sigma^2)} \mathrm{d}\tau, \tag{5.3}$$

where Q is the number of rays, σ is the half-width of the Gaussian pulse (1.25 ns for measurements), and C is a scale factor that fits this generic equation to a particular system. Since most of the energy in the Gaussian pulse should fall into one time interval of width δ, assume that

$$C \int_{-\delta/2}^{\delta/2} e^{-\tau^2/(2\sigma^2)}\, d\tau = 1. \tag{5.4}$$

The complex factor $e^{i\phi_j}$ accounts for ray interference. Phase angles ϕ_j were determined from transmitter wavelength λ, total ray path length d_j, and number of reflections n (a 180 degree phase shift per reflection was assumed). Another interpretation of (5.3) is that every time bin registers a weighted average of the energies of all predicted rays, where the weight decreases exponentially as the time difference of the ray and the bin increases. Finally, $P_k^m = |E_k^m|^2/\eta$ gives the measured power of bin k, in watts.

Figure 5.3 shows measurements and predictions for one location with relatively strong multipath. As can be seen from the graph, the predictions are within 3–5 dB of the measurements, which is similar to the results achieved by earlier research [30]. The difference can be explained by device positioning errors (devices were positioned with ±3 cm precision,

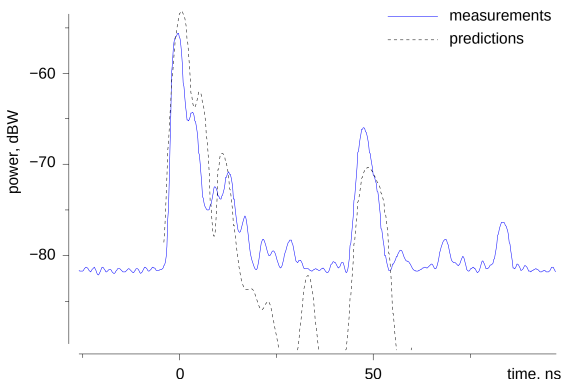


Figure 5.3. Measurement vs. prediction of channel impulse response.

which is crude given that the wavelength was 12 cm) and imprecise modeling of reflections. Additionally, small multipath components were missed by the ray tracer. These components are probably due to scattering and diffraction, which were not simulated. Geodesic tessellation frequency was 700 (9.8×10^6 beams) for calibration because the simulation results for frequencies above 700 were indistinguishable.

5.2 WCDMA Simulation and Surrogates

The ray tracing propagation model predicts a measured impulse response P_1^m, P_2^m, . . ., P_n^m of a wireless channel (see 5.1). This propagation model does not directly predict the performance of any particular wireless system that operates in this channel. A meaningful performance metric is the bit error rate (BER) defined as the ratio of the number of incorrectly received bits to the total number of bits sent. The power level P_1^m at the receiver location maps directly to the BER of a narrowband system designed for $n = 1$. However, estimating the BER of a wideband system (designed for $n > 1$) in a mobile wireless environment usually involves analytically non-tractable problems [14]. This work uses simple least squares and multivariate adaptive regression splines (MARS [13]) to fit the results of a Monte Carlo simulation of a WCDMA system. The WCDMA simulation models channel variation due to changes in the environment as a random process [21]. Notice that channel variation due to receiver movement is modeled in both the ray tracing and the WCDMA simulations, but other kinds of variation are modeled only in the WCDMA simulation. This section outlines the WCDMA simulation and describes the surrogate functions used for optimization.

Figure 5.4 briefly describes the computational steps of the WCDMA simulator. The source module of the transmitter generates information data to be sent through a wireless channel. The generated information is processed with a series of digital signal processing algorithms to reduce the potential channel errors. The wireless channel is modeled as a linear time varying process in the present work. The channel is characterized by the impulse response predicted by the ray tracer. Before being sent to the receiver, the channel output

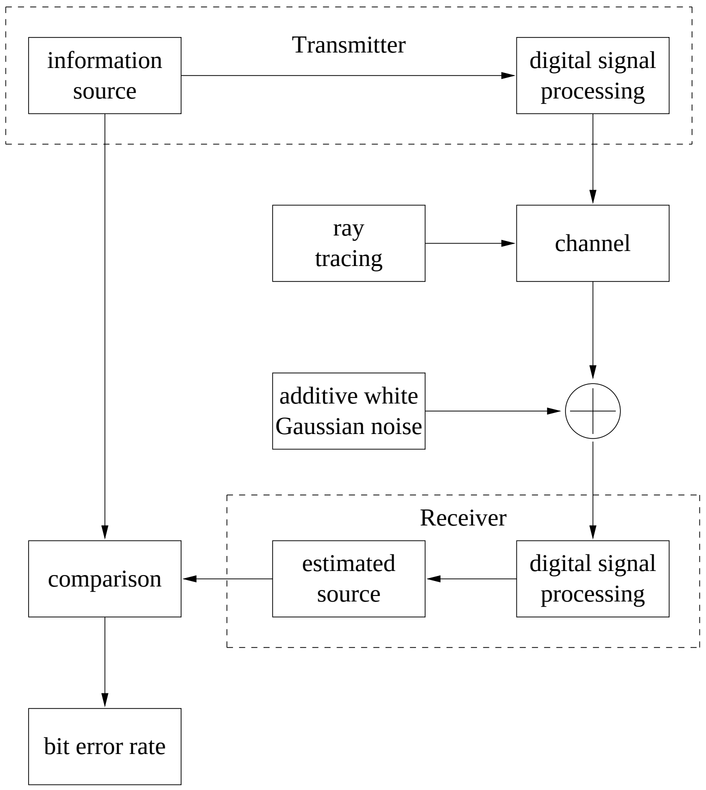


Figure 5.4. Block diagram of the WCDMA simulator.

is combined with Gaussian noise at the receiver front end. Similarly, the received distorted signal is processed with a series of digital signal processing algorithms by the receiver, which thereafter estimates the information bits to be compared with the original information bits for the BER.

The WCDMA simulation is computationally intensive since a satisfactory BER value ranges from 10^{-3} to 10^{-6}. The parallelized WCDMA simulator significantly speeds up the simulation process, but its run time is still far from practical for optimization problems. The BER depends on small-scale propagation effects that exhibit large variation with respect to receiver location. Practical coverage optimization problems involve wavelengths of less than a foot and areas of thousands of square feet. Four samples per wavelength should be taken to obtain meaningful aggregate results. Therefore, the BER results of the WCDMA simulation were approximated by simple models.

Consider a distribution of impulse responses in the environment shown in Figure 1.1, as measured by the receiver with the carrier frequency 900 HMz, the standard chip width $\delta \approx 260$ ns, and a dynamic range (a ratio of the peak power to the noise level) of 12 dB. Empirically, 49% of the impulse responses have only one multipath component ($n = 1$), 42% have two multipath components where the first one is dominant ($n = 2$, $P_1^m \geq P_2^m$), 7% have two multipath components where the second one is dominant ($n = 2$, $P_1^m < P_2^m$), and the remaining 2% have three multipath components ($n = 3$). It turns out that simple models can approximate the BERs at the majority of the receiver locations. This work considers the first two cases that account for 91% of the data.

Given a measured impulse response P_1^m, P_2^m, …, P_n^m, define the relative strength of the first multipath component

$$p_1 = P_1^m / \sum_{1 \le i \le n} P_i^m \tag{5.5}$$

and the signal-to-noise ratio (SNR)

$$S = \max_{1 \le i \le n} \{10 \log_{10}(P_i^m / N_0)\} \tag{5.6}$$

(in dB), where N_0 is the noise power level (in watts).

The BER b_1 of a WCDMA system in the first case ($n = 1$, $p_1 = 1$) was approximated by

$$\log_e(b_1) = -0.251S - 2.258, \tag{5.7}$$

obtained by a linear least squares fit of the simulated BERs for $S = 0$, 2, …, 30 in steps of 2 dB (16 points). In other words, the BER of a WCDMA system with a single path is a simple monotonically decreasing function of the SNR. This observation justifies the use of power levels to predict system performance when there is no multipath. However, using the strongest multipath component to predict the BER does not work when $n > 1$.

The second case ($n = 2$, $p_1 \ge 0.5$) was approximated using MARS models. The MARS models provided a more accurate fit to the data in comparison with the previously used linear least squares fit ([33]), reducing both the relative and absolute error. The MARS fit is a sum of products of univariate functions in the form

$$f(x) = a_0 + \sum_{n=1}^{M} a_n \prod_{k=1}^{K_n} B_{kn}\big(x_{v(k,n)}\big). \tag{5.8}$$

In this model, the multivariate spline basis functions are denoted by B, and their associated coefficients by a. This expansion of spline basis functions determines the number of basis functions, M, as well as product degree and knot locations (number of splits that gave rise to B_n is denoted by K_n) automatically from the data. In this model, the covariates are represented by x, where $v_{(k,n)}$ label the predictor variables. MARS models were developed for three different coding choices: no channel coding, rate 1/3 coding, and rate 1/2 coding. The latter two cases both use FECC (forward error correction code) to improve the BER performance. A soft decision Viterbi algorithm is used in decoding the convolutional FECC, because it produces a smaller BER than the hard decision Viterbi algorithm. The difference between these two cases lies in the convolutional code rate. Rate 1/3 coding provides a better error correction mechanism than the rate 1/2 coding.

The data used to build the no channel coding model consisted of 63 points from a Cartesian product of $S = 0, 1, \ldots, 20$ and $p_1 = 0.9, 0.7, 0.5$, and 48 points from a Cartesian product of $S = 0, 1, \ldots, 15$ and $p_1 = 0.9, 0.7, 0.5$ for the channel coding models. Plots of the fitted models reveal that the BER approaches zero as the SNR increases and that stronger multipath significantly improves performance for a fixed SNR. The latter needs some explanation because multipath is often thought of as an impairment that degrades the system performance. In this work, the SNR is defined in terms of the strongest component of the impulse response. When the SNRs of two channels that meet the criteria for this case are the same, the channel with a stronger second component contains more total power than the channel with a weaker second component. In this case, the benefits of more power outweigh the disadvantages of multipath.

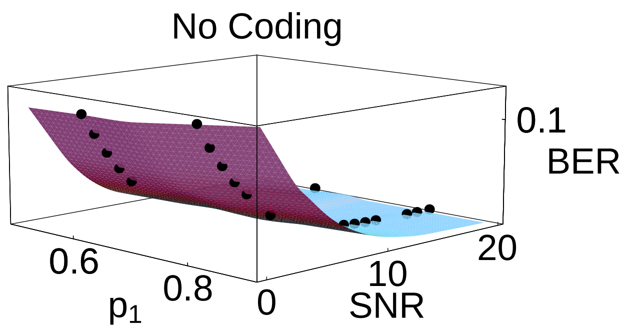


Figure 5.5. The MARS surface plot of no channel coding model.

Both surrogate models were validated with the simulated BER results. In the first case, the approximate values had an average relative error of 9.7% (0.9% minimum, 19.4% maximum) for the simulation output at $S = 1, 3, \ldots, 29$. In the second case, the approximate values had an average relative error of 12.8% and average absolute error of 0.0006 for the no channel coding model, 14.1% average relative error and 0.0005 average absolute error for the rate 1/3 coding model, and 19.9% average relative error and 0.0012 average absolute error for the rate 1/2 coding model. The validation sets for the second case consisted of 42 points for a Cartesian product of $S = 1, 2, \ldots, 20$ and $p_1 = 0.8, 0.6$ for the no channel coding model, and 32 points for a Cartesian product of $S = 1, 2, \ldots, 15$ and $p_1 = 0.8, 0.6$ for the channel coding models.

Finally, observe that the models for the two cases are not asymptotically matched. The simulated WCDMA receiver had two rake fingers, one of which was turned on or off depending on whether or not the second multipath component met the relative power threshold. Discontinuity can pose problems for the DIRECT optimization algorithm, which assumes Lipschitz continuity.

To summarize, this work considers two surrogate models for the BER of a WCDMA system. The first model was obtained using a least squares fit of the logarithm of the BER to a combination of channel characteristics. The second model was obtained using a MARS fit of the BER to a combination of channel characteristics. Empirically, both models cover 91% of the data with about 9.7% average relative error for the least squares model and 15.6% average relative error and 0.0008 average absolute error for the MARS models. The large average relative error is misleading since the larger errors occur where the BER is nearly zero, and the absolute error there is very small. The quality of the MARS approximation is apparent in Figure 5.5, which shows the MARS spline surface for the no channel coding model, the points used to construct it ($p_1 = 0.9$, 0.7, 0.5), and the points used to validate the approximation ($p_1 = 0.8$, 0.6). However, no confident claims can be made because the distribution of the fitted data is unknown. In particular, these models do not apply for $n > 2$. While the spline surrogate is quite good within the range of the fitted data, the surrogate function needs to be modified to give reasonable values for $S < 0$. The latter is crucial for using surrogates to solve the optimization problem with the DIRECT algorithm described in Chapter 2.

5.3 Objective Formulation

As [23] proved, the DIRECT algorithm is guaranteed to converge globally if the objective function is Lipschitz continuous. However, the original definitions for both performance criteria—power coverage and BER (bit error rate) do not satisfy this condition. Similar to the power coverage criterion introduced in Chapter 1, BER is the ratio of bits that have errors relative to the total number of bits received in a transmission. Reformulation is required to eliminate the discontinuity. The following two subsections describe the single transmitter case and the multiple transmitters case, respectively.

A. Single Transmitter Case

Consider the placement of a single transmitter in a coverage-limited indoor environment, in which outage is a result of excessively low local mean wanted signal power and fading of the wanted signal [6]. The ray tracing technique serves as a deterministic way to calculate the local mean signal power propagating from the transmitter to each receiver on the reception grid. To optimize the location of the transmitter, the decision variables are the x and y coordinates (transmitter height $z = z_0$ is fixed, which is a reasonable assumption in indoor environments). The objective function for the power coverage is the average shortfall of the received power $P_i(x, y, z_0)$ from the threshold T_p:

$$f(x, y, z_0) = \frac{1}{m} \sum_{i=1}^{m} (T_p - P_i(x, y, z_0))_+, \tag{5.9}$$

where T_p is the given power threshold (in dBm), $P_i(x, y, z_0)$ is the power received at the ith receiver (in dBm) with a single transmitter located at (x, y, z_0), and m is the total number of receivers. The penalty $(T_p - P_i(x, y, z_0))_+$ implies that receivers with power above the threshold do not contribute to f, while receivers with power below the threshold contribute the difference of the power and the threshold. The goal of the optimization is to minimize f. The best possible value for f is zero, which corresponds to perfect coverage.

Similarly, the objective function for BER optimization is the average shortfall of the continuous surrogate BERs $b_i(x, y, z_0)$ from the specified threshold T_b:

$$f(x, y, z_0) = \frac{1}{m} \sum_{i=1}^{m} (b_i(x, y, z_0) - T_b)_+, \tag{5.10}$$

where $b_i(x, y, z_0)$ is the surrogate estimate of BER at the ith receiver from a single transmitter located at (x, y, z_0). The penalty here is for a high bit error rate.

B. Multiple Transmitter Case

To extend the problem to the placement of multiple transmitters, an assumption was made to validate the objective function reformulation. Transmitters are assumed to operate at sufficiently different frequencies so that receivers can pick up the strongest signal. In other words, the indoor environment is only considered as a coverage-limited one, which is different from the interference-limited environment, where the outage is a result of co-channel signals dominating or interfering with the wanted signals [6]. Such an environment presents more complexities and challenges for implementing the WCDMA channel model. Chapter 6 will discuss this issue in the context of future work.

As in [11], the design variables are the transmitter coordinates

$$X = (x_1, y_1, z_1, x_2, y_2, z_2, \ldots, x_n, y_n, z_n),$$

where all $z_j = z_0$ are assumed to be fixed, the same as in the single transmitter case. Permuted coordinates will occur during optimization, since the DIRECT algorithm treats the function as a black box and has no knowledge of any symmetry relationships. When system settings are a permutation of an earlier setting, clearly the objective should not be reevaluated by using expensive ray tracing. A solution is to simply sort coordinates on each dimension, and buffer current pairs of sorted coordinates and the corresponding function values. When the permuted coordinates are detected by the optimizer, the function value will be taken directly from the buffer instead of calling the ray tracer to reevaluate the function.

With the stated assumption and the design variables above, the single transmitter location problem is a special case of n transmitter problem with $n = 1$. The goal is to

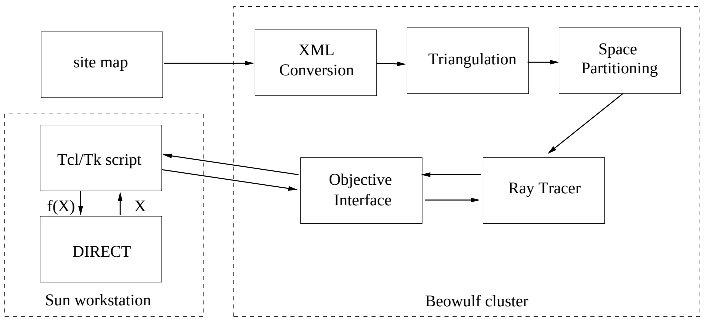


Figure 5.6. Problem solving environment for transmitter placement optimization.

minimize the average shortfall (power coverage or bit error rate) of the n transmitters over m receiver locations. Let transmitter (k, i), located at (x_k, y_k, z_0), $1 \le k \le n$, generate the highest peak power level $P_{ki}(x_k, y_k, z_0) \ge P_{ji}(x_j, y_j, z_0)$, $1 \le j \le n$, at the receiver location i, $1 \le i \le m$. The objective function is the average shortfall of the estimated performance metric from the given threshold T, given by

$$f(X) = \begin{cases} \frac{1}{m}\sum_{i=1}^{m}(T - p_{ki})_+, & \text{coverage,} \\ \frac{1}{m}\sum_{i=1}^{m}(p_{ki} - T)_+, & \text{BER,} \end{cases} \tag{5.11}$$

where p_{ki} is the performance metric of transmitter (k, i) evaluated at the ith receiver location. For power coverage optimization, p_{ki} is $P_{ki}(x_k, y_k, z_0)$ and $(T - p_{ki})_+$ is the penalty for a low power level. For BER optimization, p_{ki} is BER_{ki} and $(p_{ki} - T)_+$ is the penalty for a high bit error rate.

5.4 Optimization Results

Optimization was done inside a problem solving environment (PSE) as shown in Figure 5.6. Ray tracing was performed on a 200-node Athlon 650 Beowulf cluster of Linux workstations. The DIRECT optimizer and the user interface ran on a Sun workstation outside the cluster. Tcl/Tk scripts glued the pieces together and provided a graphical user interface. Similar to [25], users could select regions for transmitter placement (to be optimized) and regions to be covered.

Runs for optimizing transmitter placement were executed with respect to the two performance criteria—coverage and BER. The ray tracer's tessellation frequency was 100 for coverage and 300 for BER. The former was sufficient to match the peak powers against measurements, while the latter was required to match the whole impulse responses. Two different indoor environments were chosen to demonstrate the effectiveness of the optimization. The first environment is located on the fourth floor of Durham Hall at Virginia Tech.

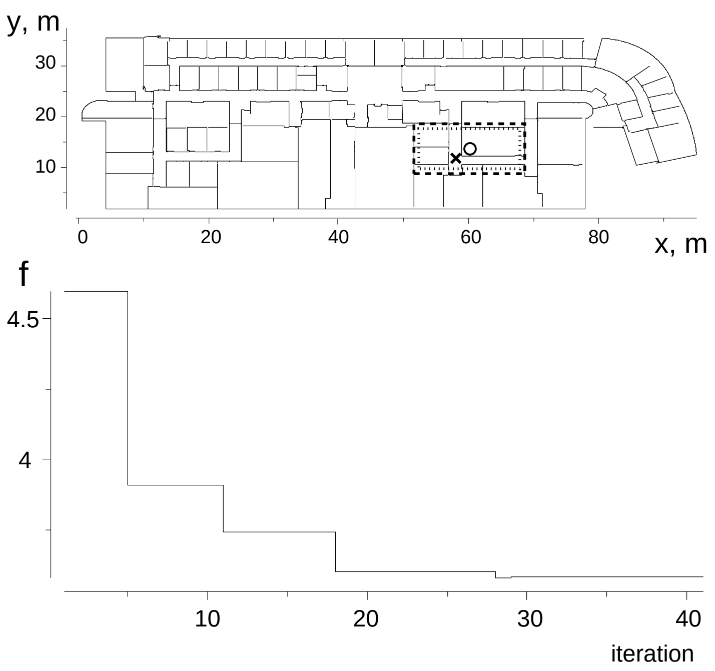


Figure 5.7. Power coverage optimization results for one transmitter. Bounds on transmitter placement are drawn with dotted lines and the initial (final) position is marked with a circle (cross) in the center.

It was the first case study for the global optimization technique. Simulations have been verified with measurement data (see 5.1). The simulation verification with measurement on the second environment—the second floor of Whittemore Hall—is in progress. The same environment had been used in both raytracing simulations and measurements in [30], which considers signal diffractions in the propagation model so that it can match well the measured and predicted propagation in a variety of indoor environments. (The propagation model code of [30], while having better physics than the present ray tracing code, is orders of magnitude slower because of inefficient data structures.)

A. Durham Hall, fourth floor

The results of optimizing a single transmitter location in the case of coverage are shown in Figure 5.7. It took 41 evaluations (3 minutes, 45 seconds) to reduce the objective function by 22.2% (from 4.60 dB to 3.58 dB) demonstrating the fast convergence of the DIRECT algorithm.

Figure 5.8 illustrates power coverage optimization of the locations of three transmitters to cover eighteen rooms and a corridor bounded by the box in the upper-left corner. 93 function evaluations reduced the objective from 2.77 dB to 2.51 dB, or by 9.4%, in 38 minutes on 40 machines. Figure 5.9 depicts BER optimization of the locations of two transmitters to cover half of the former region. In Figure 5.9 (a), 56 iterations reduced

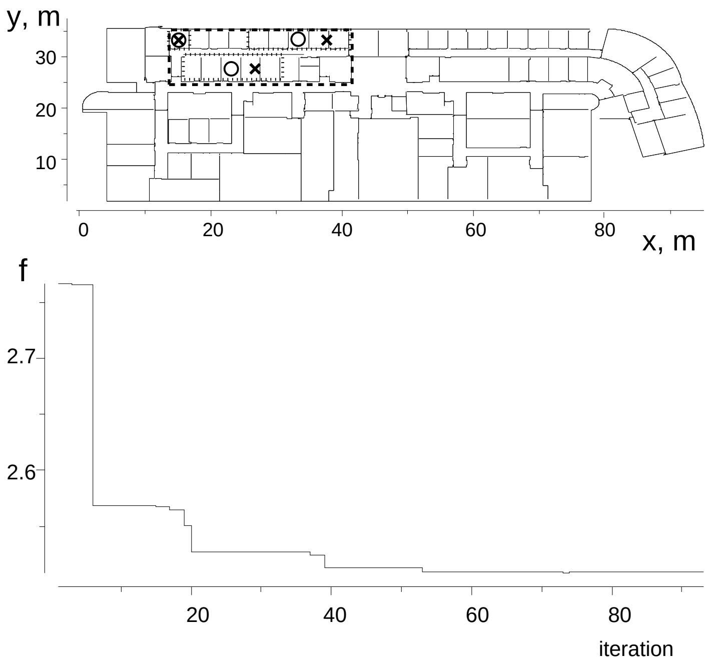


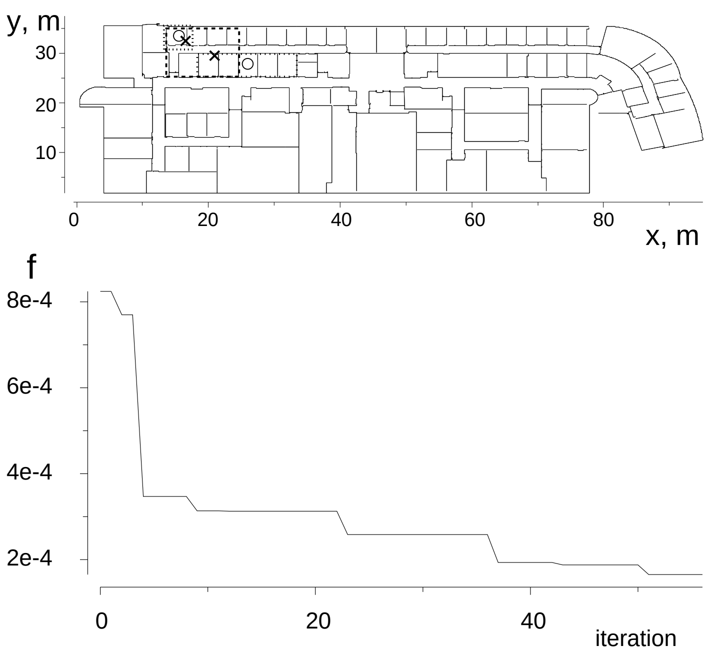
Figure 5.8. Power coverage optimization results for three transmitters. Bounds on transmitter placement are drawn with dotted lines and their initial (final) positions are marked with circles (crosses). The dashed line delimits the region to be covered.

the objective function from 8.24e-4 to 1.65e-4 in 3 hours and 26 minutes on 40 machines. The BER threshold was 10^{-3}, so this improvement corresponds to a 79.9% reduction in the average BER. From Figure 5.9 (b), it's observed that the percentage of the receivers with satisfied BER is growing as the objective function decreases. In both cases, the optimization loop stops with the minimum diameter required by the problem. System performance was significantly improved by DIRECT with a reasonable number of evaluations.

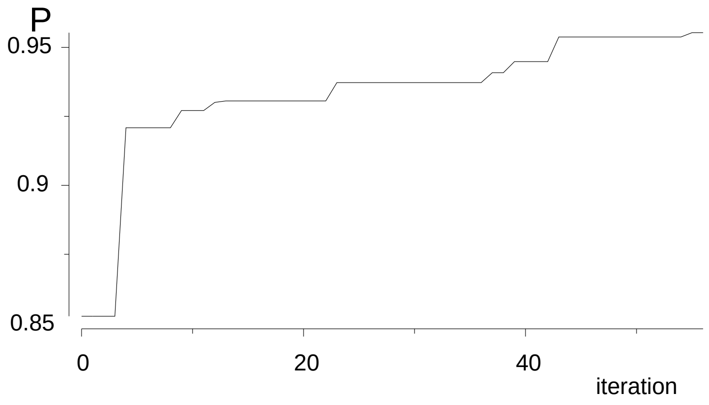
Figure 5.10 demonstrates the effectiveness of the new stopping criterion—objective function convergence tolerance. This figure shows the power coverage optimization results for two transmitters in the region delimited by dashed lines. The initial locations are marked as circles. Bounds on transmitter placement are drawn with dotted lines. Two simulations were done with different objective function convergence tolerances—0 and 0.001. In the former case, it took 52 iterations to reach the final locations (marked as crosses). In the latter case, the final locations (marked as triangles) were found after 27 iterations. Using a nonzero objective function convergence tolerance saved 25 expensive ray tracing iterations.

B. Whittemore Hall, second floor

Similar numerical experiments were conducted for the second environment to compare the optimization results and performance in terms of power coverage and BER. Figure 5.11



(a) Objective function.



(b) Percentage of receivers with satisfied BER values.

Figure 5.9. BER optimization results for two transmitters. The region to be covered is half of that in Figure 5.8

and Figure 5.12 show the results for optimizing the placement for a single transmitter and three transmitters respectively.

To optimize the single transmitter location, the minimum diameter stopping criterion was used in both power coverage and BER optimizations. It took 6 more evaluations for the BER optimization to finish than the power coverage optimization. Since BER simulation is affected by numerous system and channel parameters such as signal-to-noise ratio, data rate, modulation type, etc. [14], it is very sensitive to parameter changes caused by changing transmitter locations. The objective function for BER exhibits more complexity (both

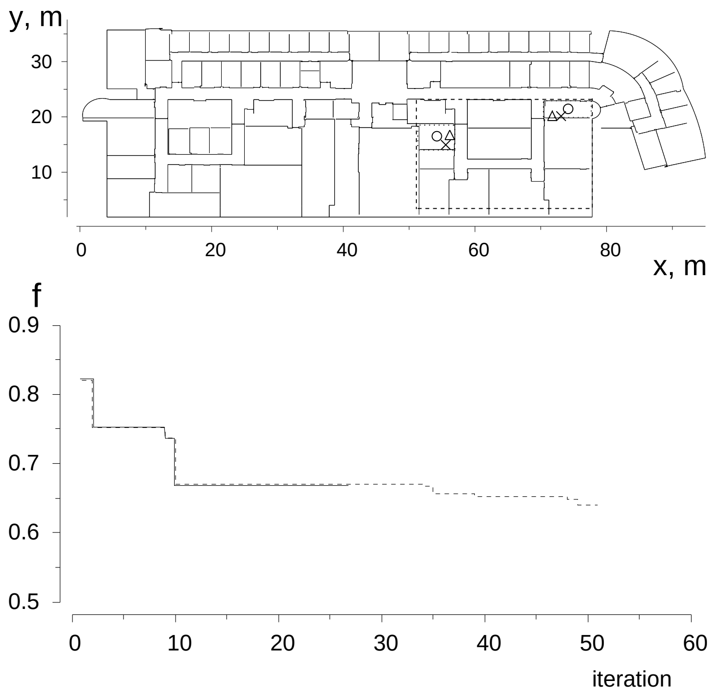
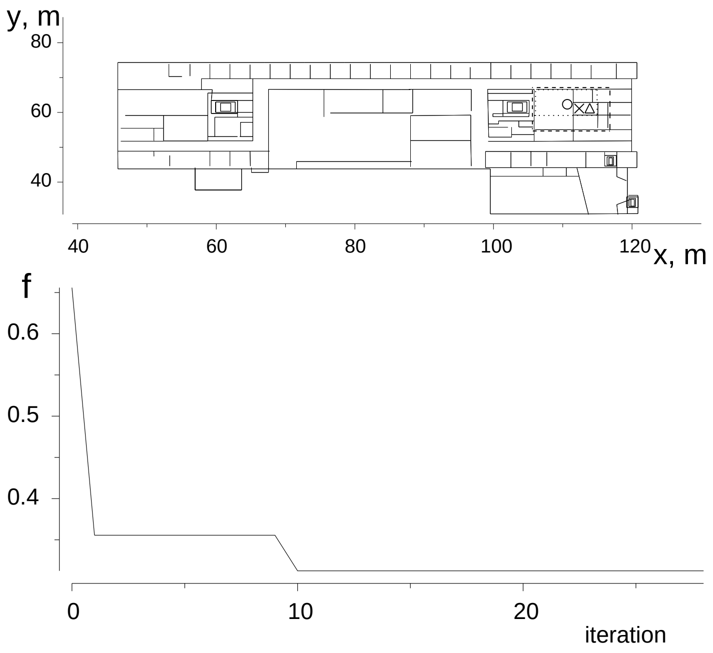


Figure 5.10. Power coverage optimization results for two transmitters with objective function convergence tolerance = 0 (dashed) and 0.001 (solid).

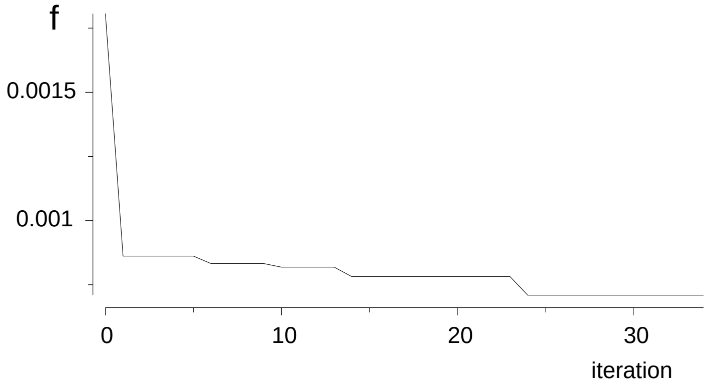
multipath components are involved when there are two resolvable paths.) than the one for power coverage (only the dominant multipath component is considered), therefore it takes more evaluations to approach the global optimum. It's also observed in Figure 5.11 that the final locations are different. Generally, BER optimization results are preferred, since BER is considered a better performance criterion in the design of mobile communication systems as pointed out in Chapter 5.2.

In the case of optimizing three transmitter locations, the stopping criterion was the maximum number of evaluations. Both BER and coverage optimization stopped at the 54th iteration. The exact same final transmitter locations were reached at the 51st iteration. Interestingly, the final locations are exactly the same (marked as crosses at the top of Figure 5.12). This indicates a reasonable connection between these two different performance metrics— power coverage and BER.

Table 5.1 compares the cost and improvement for these four optimization experiments. The computational cost of ray tracing iterations is the metric. Improvement is defined as the ratio of reduction in the function value to the initial function value. In both cases, the BER optimization achieved a better improvement than the power coverage optimization with almost the same cost. For the single transmitter, the objective function value of the BER optimization was reduced by 60.7% while the power coverage optimization improved only by 37.7%. In the case of three transmitters, the objective function was reduced by



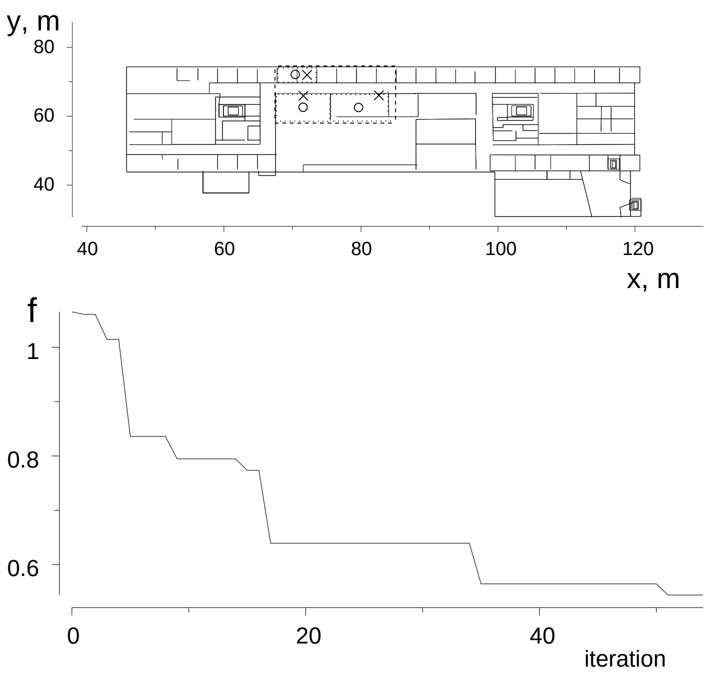
(a) Power coverage



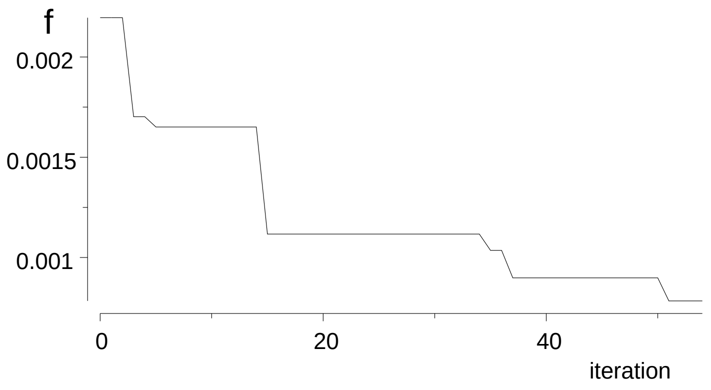
(b) BER

Figure 5.11. Power coverage and BER optimization results for a single transmitter. Bounds on transmitter placement are drawn with dotted lines and the initial position is marked with a circle. Final position found by the power coverage (BER) optimization is marked with a triangle (cross). The dashed line delimits the region to be covered.

48.9% for the power coverage optimization and by 64.2% for the BER optimization. From this comparison, the DIRECT algorithm works very cost-effectively for BER optimization problems. On the other hand, the center-sampling strategy of DIRECT benefits the power coverage optimizations by starting at the centers of bounded areas, so that the well-distributed initial locations only need a little adjustment. This can also explain why the power coverage optimization gave less improvement.



(a) Power coverage



(b) BER

Figure 5.12. Power coverage and BER optimization results for three transmitters. Bounds on transmitter placement are drawn with dotted lines and their initial (final) positions are marked with circles (crosses). The dashed line delimits the region to be covered.

Table 5.1. Cost (number of ray tracing iterations) and improvement (relative function value reduction) comparison for power coverage and BER optimization experiments. (Second environment.)

	cost	improvement
Power coverage	28	37.7%
BER	34	60.7%

(a) single transmitter

	cost	improvement
Power coverage	54	48.9%
BER	54	64.2%

(b) three transmitters

Chapter 6: CONCLUSIONS AND FUTURE WORK

DIRECT has demonstrated its effectiveness in solving the global optimal transmitter placement problem in wireless communication systems. One of the major contributions of the present work is the design of the dynamic data structures for the DIRECT algorithm. 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.

Several extensions to the present work are envisioned. First, different ways of combining the deterministic propagation model with the stochastic wireless system model will require a parallel implementation of the DIRECT algorithm. Some ongoing research topics include a MPI-based parallel version using the dynamic data structures (or suitable variants of them) proposed here. Both a master-slave version (for a 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 in wireless communication systems and other MDO (multidisciplinary design optimization) problems. Second, incorporating nonlinear constraints has been attempted by several authors (Jones [22], Torczon), but the issue is by no means satisfactorily resolved. Finally, the surrogate functions for the BER can be extended to channels with relatively strong multipath and interference. Moreover, wireless systems with data quality bit error rates (10^{-6}) can be considered.

REFERENCES

[1] K. K. Bae, J. Jiang, W. H. Tranter, C. R. Anderson, T. S. Rappaport, J. He, A. Verstak, L. T. Watson, N. Ramakrishnan, and C. A. Shaffer, "WCDMA STTD performance analysis with transmitter location optimization in indoor systems using ray tracing techniques", in *IEEE 2002 Radio and Wireless Conference (RAWCON 2002), Boston, MA, 2002*, to appear.

[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] C. A. Baker, L. T. Watson, B. Grossman, R. T. Haftka, and W. H. Mason, "Parallel global aircraft configuration design space exploration", in *High Performance Computing Symposium 2000, A. Tentner (Ed.), Soc. for Computer Simulation Internat*, San Diego, CA, 2000, pp. 101–106.

[4] M. C. Bartholomew-Biggs, S. C. Parkhurst, and S. P. Wilson, "Global optimization approaches to an aircraft routing problem", *Engrg. Comput.*, to appear.

[5] K. S. Butterworth, K. W. Sowerby, and A. G. Williamson, "Base station placement for in-building mobile communication systems to yield high capacity and efficiency", *IEEE Transactions on Communications*, vol. 48, pp. 658–669, 2000.

[6] K. W. Cheung and R. D. Murch, "Optimizing Indoor Base Station Locations in Coverage- and Interference- Limited Indoor Environments", in *IEE Proceedings-Communications*, vol. 143(6), 1998, pp. 445–450.

[7] 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 *Aerospace Numerical Simulation Symposium '99*, Tokyo, Japan, June, 1999, pp. 23–28.

[8] 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*, 21(4), 2001, pp. 415-432.

[9] G. D. Durgin, T. S. Rappaport, and Hao Xu, "Measurements and models for radio path loss and penetration loss in and around homes and trees at 5.85 GHz", *IEEE Transactions on Communications*, vol. 46(11), pp. 1484–1496, 1998.

[10] S. J. Fortune, "A beam-tracing algorithm for prediction of indoor radio propagation", in *WACG: 1st Workshop on Applied Computational Geometry: Towards Geometric Engineering, Lecture Notes in Computer Science*, vol. 1148, pp. 157–166, 1996.

[11] S. J. Fortune, D. M. Gay, B. W. Kernighan, O. Landron, R. A. Valenzuela, and M. H. Wright, "WISE design of indoor wireless systems: practical computation and optimization", *IEEE Computational Science & Engineering*, vol. 2(1), pp. 58–68, Spring, 1995.

[12] B. Freisleben, D. Hartmann and T. Kielmann, "Parallel raytracing: a case study on partitioning and scheduling on workstation clusters", in *Proc. Thirtieth International Conference on System Sciences, Hawaii*, vol. 1, pp. 596–605, 1997.

[13] J. H. Friedman, "Multivariate adaptive regression splines", *Annals of Statistics*, vol. 19(1), pp. 1–67, 1991.

[14] Victor Fung, Theodore S. Rappaport, and Berthold Thoma, "Bit error simulation for $\frac{\pi}{4}$ DQPSK mobile radio communications using two-ray and measurement-based impulse response models", *IEEE Journal on Selected Areas in Communications*, vol. 11, No. 3, April, 1993.

[15] 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.

[16] 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.

[17] A. Glassner, "Space subdivision for fast ray tracing", *IEEE Computer Graphics and Applications*, vol. 4(10), pp. 15–22, October, 1984.

[18] J. He, A. Verstak, L. T. Watson, T. S. Rappaport, C. R. Anderson, N. Ramakrishnan, C. A. Shaffer, W. H. Tranter, K. Bae, and J. Jiang, "Global optimization of transmitter placement in wireless communication systems", in *Proc. High Performance Computing Symposium 2002*, A. Tentner (ed.), Soc. for Modeling and Simulation International, San Diego, CA, pp. 328–333, 2002.

[19] J. He, L. T. Watson, N. Ramakrishnan, C. A. Shaffer, A. Verstak, J. Jiang, K. Bae, and W. H. Tranter, "Dynamic data structures for a direct search algorithm", *Computational Optimization and Applications, 2002*, to appear.

[20] X. Huang, U. Behr, and W. Wiesbeck, “Automatic base station placement and dimensioning for mobile network planning”, in *Proc. of Vehicular Technology Conference*, IEEE VTS Fall VTC 2000. 52nd, vol. 4, pp. 1544–1549, 2000.

[21] M. C. Jeruchim, Philip Balaban, and K. S. Shanmugan, *Simulation of Communication Systems*, Plenum Press, New York, 1992.

[22] D. R. Jones, “The DIRECT global optimization algorithm”, *in Encyclopedia of Optimization, vol. 1*, Kluwer Academic Publishers, Boston, 2001, pp. 431–440.

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

[24] 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.

[25] M. A. Panjwani, A. L. Abbott, and T. S. Rappaport, “Interactive computation of coverage regions for wireless communication in multifloored indoor environments”, *IEEE Journal on Selected Areas in Communications*, vol. 14(3), pp. 420–430, 1996.

[26] J. D. Pinter, *Global Optimization In Action*, Kluwer Academic Publishers, Boston, 1996.

[27] T. S. Rappaport, *Wireless Communications: Principles and Practice*, Prentice Hall, New Jersey, 1996.

[28] T. S. Rappaport and R. R. Skidmore, Wireless Valley Communications, Inc., *Method and System for Automated Optimization of Antenna Positioning in 3-D*, US Patent 6,317,599, November, 2001.

[29] K. R. Schaubach, N. J. Davis IV, and T. Rappaport, “A ray tracing method for predicting path loss and delay spread in microcellular environments”, in *Proc. IEEE Vehicular Technology Conference*, vol. 2, pp. 932–935, 1992.

[30] S. Y. Seidel and T. S. Rappaport, “Site-specific propagation prediction for wireless in-building personal communication system design”, *IEEE Transactions on Vehicular Technology*, vol. 43(4), pp. 879–891, 1994.

[31] H. D. Sherali, C. M. Pendynala, and T. S. Rappaport, “Optimal location of transmitters for micro-cellular radio communication system design”, *IEEE Journal on Selected Areas in Communications*, vol. 14(4), pp. 662–673, 1996.

[32] V. Torczon, “On the convergence of the multidirectional search algorithm”, *SIAM Journal on Optimization*, vol. 1, no. 1, pp. 123–145, 1991.

[33] A. Verstak, J. He, L. T. Watson, N. Ramakrishnan, C. A. Shaffer, T. S. Rappaport, C. R. Anderson, K. Bae, J. Jiang, and W. H. Tranter, “S^4W: globally optimized design of wireless communication systems”, in *16th Internat. Parallel & Distributed Processing Symp.*, CD-ROM, IEEE Computer Soc., Los Alamitos, CA, 2002, 8 pages.

[34] 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, pp. 148–153, 2001.

[35] L. T. Watson and C. A. Baker, “A fully-distributed parallel global search algorithm”, *Engineering Computations*, vol. 18(1/2), pp. 155–169, 2001.

[36] 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.

Appendix A: Fortran 90 MODULE REAL_PRECISION

The Fortran 90 module REAL_PRECISION (from HOMPACK90 [36]) that defines 64-bit real arithmetic discussed in Chapter 3.2 is listed here.

```
MODULE REAL_PRECISION  ! From HOMPACK90.
  ! This is for 64-bit arithmetic.
  INTEGER, PARAMETER:: R8=SELECTED_REAL_KIND(13)
END MODULE REAL_PRECISION
```

Appendix B: Fortran 90 MODULE VTDIRECT_GLOBAL

The Fortran 90 module VTDIRECT_GLOBAL that defines the dynamic data structures discussed in Chapter 3 is listed here.

```
MODULE VTDIRECT_GLOBAL  ! Defines data types, parameters, and
  USE REAL_PRECISION    ! module procedures used by VTDIRECT.
  IMPLICIT NONE
!
!HyperBox: Defines an n-dimensional box.
!  val  - Function value at the box center.
!  c    - The center point coordinates.
!  side - Box side lengths for all dimensions.
!  diam - Box diameter squared.
!
TYPE HyperBox
  REAL(KIND = R8) :: val
  REAL(KIND = R8), DIMENSION(:), POINTER :: c
  REAL(KIND = R8), DIMENSION(:), POINTER :: side
  REAL(KIND = R8) :: diam
END TYPE HyperBox
!
!BoxLink: Contains 1-D array of hyperboxes, linked to each column of
!         BoxMatrix when needed.
!  Line - 1-D array of boxes.
!  ind  - Index of last box in array 'Line'.
!  next - The pointer to the next BoxLink.
!  prev - The pointer to the previous BoxLink.
!
TYPE BoxLink
  TYPE(HyperBox), DIMENSION(:), POINTER :: Line
  INTEGER :: ind
  TYPE(BoxLink), POINTER :: next
  TYPE(BoxLink), POINTER :: prev
END TYPE BoxLink
!
!P_BoxLink: Contains a pointer to a BoxLink for a column in BoxMatrix.
!  p - Pointer to a BoxLink.
!
TYPE P_BoxLink
  TYPE(BoxLink), POINTER :: p
END TYPE P_BoxLink
!
!BoxLine: Contains 1-D array of newly sampled hyperboxes.
!  Line - 1-D array of boxes.
!  ind  - Index of last box in array 'Line'.
!  dir  - Directions in which box centers are sampled.
!
TYPE BoxLine
  TYPE(HyperBox), DIMENSION(:), POINTER :: Line
  INTEGER :: ind
  INTEGER, DIMENSION(:), POINTER :: dir
END TYPE BoxLine
!
!BoxMatrix: Contains 2-D array of hyperboxes.
```

```
!  M      - 2-D array of boxes.
!  ind    - An array holding the number of boxes in all the columns in 'M'.
!  child  - The pointer to the next BoxMatrix with the smaller diameters.
!  sibling - The pointer array for all columns, pointing to the next
!           BoxLinks with the same diameters.
!  id     - Identifier of this box matrix among all box matrices.
!
TYPE BoxMatrix
  TYPE(HyperBox), DIMENSION(:,:), POINTER :: M
  INTEGER, DIMENSION(:), POINTER :: ind
  TYPE(BoxMatrix), POINTER :: child
  TYPE(P_BoxLink), DIMENSION(:), POINTER :: sibling
  INTEGER :: id
END TYPE BoxMatrix
!
!int_vector : A list holding integer values.
!  dim      - The index of the last element in the list.
!  elements - The integer array.
!  flags    - The integer array holding the status for 'elements'.
!             Bit 0: status of convex hull processing.
!             Bit 1: update status of a box column.
!  next     - The pointer to the next integer vector list.
!  prev     - The pointer to the previous integer vector list.
!  id       - Identifier of this integer vector list among all lists.
!
TYPE int_vector
  INTEGER :: dim
  INTEGER, DIMENSION(:), POINTER :: elements
  INTEGER, DIMENSION(:), POINTER :: flags
  TYPE(int_vector), POINTER :: next
  TYPE(int_vector), POINTER :: prev
  INTEGER :: id
END TYPE int_vector
!
!real_vector: A list holding real values.
!  dim      - The index of the last element in the list.
!  elements - The real array.
!  next     - The pointer to the next real vector list.
!  prev     - The pointer to the previous real vector list.
!  id       - Identifier of this real vector list among all lists.
!
TYPE real_vector
  INTEGER :: dim
  REAL(KIND = R8), DIMENSION(:), POINTER :: elements
  TYPE(real_vector), POINTER :: next
  TYPE(real_vector), POINTER :: prev
  INTEGER :: id
END TYPE real_vector
!
!ValList: a list for sorting the wi for all dimensions i corresponding
!  to the maximum side length. wi = min f(c+delta*ei), f(c-delta*ei),
!  the minimum of objective function values at the center point c +
!  delta*ei and the center point c - delta*ei, where delta is one-third of
!  this maximum side length and ei is the ith standard basis vector.
!  dim - The index of the last element in the list.
!  val - An array holding the minimum function values.
!  dir - An array holding the sampling directions corresponding to the
!       function values in array 'val'.
!
TYPE ValList
  INTEGER :: dim
  REAL(KIND = R8), DIMENSION(:), POINTER :: val
  INTEGER, DIMENSION(:), POINTER :: dir
END TYPE
! Parameters.
! Argument input error.
INTEGER, PARAMETER :: INPUT_ERROR = 10
! Allocation failure error.
INTEGER, PARAMETER :: ALLOC_ERROR = 20
! Stop rule 1: maximum iterations.
INTEGER, PARAMETER :: STOP_RULE1 = 0
! Stop rule 2: maximum evaluations.
INTEGER, PARAMETER :: STOP_RULE2 = 1
! Stop rule 3: minimum diameter.
INTEGER, PARAMETER :: STOP_RULE3 = 2
! Stop rule 4: minimum relative change in objective function.
INTEGER, PARAMETER :: STOP_RULE4 = 3
! 'flags' bits for 'setInd' of type 'int_vector'.
INTEGER, PARAMETER :: CONVEX_BIT = 0
! Bit 0: if set, the first box on the corresponding column is in the convex
!       hull box set.
INTEGER, PARAMETER :: UPDATE_BIT = 1
! Bit 1: if set, the box column has been updated in the previous
!       iteration.
! Interfaces.
INTERFACE ASSIGNMENT (=)
  MODULE PROCEDURE AssgBox
END INTERFACE
INTERFACE insNode
  MODULE PROCEDURE insNodeI
  MODULE PROCEDURE insNodeR
END INTERFACE insNode
INTERFACE rmNode
  MODULE PROCEDURE rmNodeI
  MODULE PROCEDURE rmNodeR
END INTERFACE rmNode
CONTAINS
SUBROUTINE AssgBox(x, y)
IMPLICIT NONE
! Copies the contents of box 'y' to box 'x'.
!
! On input:
! y - Box with type 'HyperBox'.
!
! On output:
! x - Box with type 'HyperBox' having contents of box 'y'.
!
TYPE(HyperBox), INTENT(IN) :: y
TYPE(HyperBox), INTENT(INOUT) :: x
x%val = y%val
x%diam = y%diam
x%c = y%c
```

```
x%side = y%side
RETURN
END SUBROUTINE AssgBox
SUBROUTINE insNodeR(n, pt, index, Set)
IMPLICIT NONE
! Inserts a real number 'pt' at the indexed position 'index' of 'Set'.
!
! On input:
! n     - The maximum length of the real array in each node of 'Set'.
! pt    - The real number to be inserted to 'Set'.
! index - The position at which to insert 'pt' in a node of 'Set'.
! Set   - A linked list of type(real_vector) nodes.
!
! On output:
! Set   - 'Set' has an added real number and modified 'dim' component.
!
INTEGER, INTENT(IN) :: n
REAL(KIND = R8), INTENT(IN) :: pt
INTEGER, INTENT(IN) :: index
TYPE(real_vector), INTENT(INOUT), TARGET :: Set
! Local variables.
TYPE(real_vector), POINTER :: p_set ! Pointer to a node of 'Set'.
! Insert 'pt' into 'Set'.
IF (Set%dim < n ) THEN
  ! The head node is not full. There are no other nodes.
  ! Update 'dim'.
  Set%dim = Set%dim + 1
  IF (index == Set%dim) THEN
    ! The desired position is at end, so insert 'pt' at end.
    Set%elements(Set%dim) = pt
  ELSE
    ! The desired position is not at end, so shift elements before
    ! insertion.
    Set%elements(index+1:Set%dim) = Set%elements(index:Set%dim-1)
    Set%elements(index) = pt
  END IF
ELSE
  ! The head node is full. Check other nodes.
  p_set => Set%next
  ! To shift elements, find the last node which is not full.
  DO WHILE(p_set%dim == n)
    p_set => p_set%next
  END DO
  ! Found the last node 'p_set' which is not full.
  ! Update 'dim' of 'p_set'. Shift element(s) inside this node, if any.
  p_set%dim = p_set%dim + 1
  ! Loop shifting until reaching the node 'Set' at which to insert 'pt'.
  DO WHILE(.NOT. ASSOCIATED(p_set, Set))
    ! Shift element(s) inside this node,  if any.
    p_set%elements(2:p_set%dim) = p_set%elements(1:p_set%dim-1)
    ! Shift the last element from the previous node to this one.
    p_set%elements(1) = p_set%prev%elements(n)
    ! Finished shifting this node. Go to the previous node.
    p_set => p_set%prev
  END DO
  ! Reached the original node 'Set'.
  IF (index == Set%dim) THEN
    ! The desired position is at end, so insert 'pt' at end.
    Set%elements(Set%dim) = pt
  ELSE
    ! The desired position is not at end, so shift elements before
    ! insertion.
    Set%elements(index+1:Set%dim) = Set%elements(index:Set%dim-1)
    Set%elements(index) = pt
  END IF
END IF
RETURN
END SUBROUTINE insNodeR
SUBROUTINE insNodeI(n, pt, index, Set)
IMPLICIT NONE
! Inserts an integer number 'pt' at the indexed position 'index' of 'Set'.
!
! On input:
! n     - The maximum length of the integer array in each node of 'Set'.
! pt    - The integer number to be inserted to 'Set'.
! index - The position at which to insert 'pt'.
! Set   - A linked list of type(int_vector) nodes.
!
! On output:
! Set   - 'Set' has an added integer number and modified 'dim' component.
INTEGER, INTENT(IN) :: n
INTEGER, INTENT(IN) :: pt
INTEGER, INTENT(IN) :: index
TYPE(int_vector), INTENT(INOUT), TARGET :: Set
! Local variables.
TYPE(int_vector), POINTER :: p_set ! Pointer to a node of 'Set'.
! Insert 'pt' into 'Set'.
IF (Set%dim < n ) THEN
  ! The head node is not full. There are no other nodes.
  Set%dim = Set%dim + 1
  IF (index == Set%dim) THEN
    ! The desired position is at end, so insert 'pt' at end.
    Set%elements(Set%dim) = pt
    ! Clear the 'flags'.
    Set%flags(Set%dim) = 0
ELSE
    ! The desired position is not at end, so shift elements before
    ! insertion.
    Set%elements(index+1:Set%dim) = Set%elements(index:Set%dim-1)
    ! Shift the 'flags'.
    Set%flags(index+1:Set%dim) = Set%flags(index:Set%dim-1)
    ! Insert 'pt'.
    Set%elements(index) = pt
    ! Clear the 'flags'.
    Set%flags(index) = 0
  END IF
ELSE
  ! The head node is full. There must be other nodes.
  p_set => Set%next
  ! To shift elements, find the last node which is not full.
  DO WHILE(p_set%dim == n)
    p_set => p_set%next
  END DO
  ! Found the last node 'p_set' which is not full.
  ! Update 'dim' of 'p_set'. Shift element(s), if any.
  p_set%dim = p_set%dim + 1
  ! Loop shifting until reaching the original node 'Set'.
  DO WHILE(.NOT. ASSOCIATED(p_set, Set))
    ! Shift element(s) inside this node, if any.
    p_set%elements(2:p_set%dim) = p_set%elements(1:p_set%dim-1)
    ! Shift the last element from the previous node to this one.
    p_set%elements(1) = p_set%prev%elements(n)
    ! Shift the 'flags'.
    p_set%flags(2:p_set%dim) = p_set%flags(1:p_set%dim-1)
    p_set%flags(1) = p_set%prev%flags(n)
    ! Finished shifting this node. Go to the previous node.
    p_set => p_set%prev
  END DO
  ! Reached the original node 'Set'.
  IF (index == Set%dim) THEN
    ! The desired position is at end, so insert 'pt' at end.
    Set%elements(Set%dim) = pt
    ! Clear the 'flags'.
    Set%flags(Set%dim) = 0
  ELSE
    ! The desired position is not at end, so shift elements before
    ! insertion.
    Set%elements(index+1:Set%dim) = Set%elements(index:Set%dim-1)
    ! Shift the 'flags'.
    Set%flags(index+1:Set%dim) = Set%flags(index:Set%dim-1)
    ! Insert 'pt'.
    Set%elements(index) = pt
    ! Clear the 'flags'.
    Set%flags(index) = 0
  END IF
END IF
RETURN
END SUBROUTINE insNodeI
SUBROUTINE rmNodeI(n, offset, index, Set)
IMPLICIT NONE
! Removes an integer entry at position 'index' from the integer array
! in the node at 'offset' links from the beginning of the linked list
! 'Set'.
!
! On input:
! n      - The maximum length of the integer array in each node of 'Set'.
! offset - The offset of the desired node from the first node of 'Set'.
! index  - The position at which to delete an integer from the integer
!          array in the node.
! Set    - A linked list of type(int_vector) nodes.
!
! On output:
! Set    - The desired node of 'Set' has the indexed integer entry removed
!          and the 'dim' component modified.
!
INTEGER, INTENT(IN) :: n
INTEGER, INTENT(IN) :: offset
INTEGER, INTENT(IN) :: index
```

```
TYPE(int_vector), INTENT(INOUT), TARGET :: Set
! Local variables.
INTEGER :: i  ! Loop counter.
TYPE(int_vector), POINTER :: p_set  ! Pointer to a node of 'Set'.
! Find the desired node.
p_set => Set
DO i = 1, offset
  p_set => p_set%next
END DO
IF (index < p_set%dim) THEN
  ! It's not the last entry in 'p_set', so shift elements.
  p_set%elements(index:p_set%dim-1) = p_set%elements(index+1:p_set%dim)
  ! Shift the 'flags'.
  p_set%flags(index:p_set%dim-1) = p_set%flags(index+1:p_set%dim)
END IF
IF (p_set%dim < n) THEN
  ! There are not other elements in next node, so remove the indexed
  ! entry directly from 'Set' by updating 'dim'.
  p_set%dim = p_set%dim - 1
ELSE
  ! There might be nodes in which to shift elements.
  ! Check if any element(s) in next node to shift.
  IF (ASSOCIATED(p_set%next)) THEN
    p_set => p_set%next
    DO
      IF (p_set%dim > 0) THEN
        ! There are elements to shift.
        ! Shift one element from p_next into its previous node.
        p_set%prev%elements(n) = p_set%elements(1)
        ! Shift elements inside p_next.
        p_set%elements(1:p_set%dim-1) = p_set%elements(2:p_set%dim)
        ! Shift the 'flags'.
        p_set%prev%flags(n) = p_set%flags(1)
        p_set%flags(1:p_set%dim-1) = p_set%flags(2:p_set%dim)
      ELSE
        ! There are no elements to shift. Update 'dim' of previous node.
        p_set%prev%dim = p_set%prev%dim - 1
        EXIT
      END IF
      ! Move on to the next node, if any. If there are no more nodes, update
      ! 'dim'.
      IF (ASSOCIATED(p_set%next)) THEN
        p_set => p_set%next
      ELSE
        p_set%dim = p_set%dim - 1
        EXIT
      END IF
    END DO
  ELSE
    ! There are no more nodes. Update 'dim' of 'p_set'.
    p_set%dim = p_set%dim - 1
  END IF
END IF
RETURN
END SUBROUTINE rmNodeI
SUBROUTINE rmNodeR(n, offset, index, Set)
```

```
IMPLICIT NONE
! Removes a real entry at position 'index' from the real array
! in the node at 'offset' links from the beginning of the linked list
! 'Set'.
!
! On input:
! n      - The maximum length of the real array in each node of 'Set'.
! offset - The offset of the desired node from the first node of 'Set'.
! index  - The position at which to delete a real entry from the real
!          array in the node.
! Set    - A linked list of type(real_vector) nodes.
!
! On output:
! Set    - The desired node of 'Set' has the indexed real entry removed
!          and the 'dim' component modified.
!
INTEGER, INTENT(IN) :: n
INTEGER, INTENT(IN) :: offset
INTEGER, INTENT(IN) :: index
TYPE(real_vector), INTENT(INOUT), TARGET :: Set
! Local variables.
INTEGER :: i  ! Loop counter.
TYPE(real_vector), POINTER :: p_set  ! Pointer to a node of 'Set'.
! Find the desired node.
p_set => Set
DO i = 1, offset
  p_set => p_set%next
END DO
IF (index < p_set%dim)THEN
  ! It's not the last entry in 'p_set', so shift elements.
  p_set%elements(index:p_set%dim-1) = p_set%elements(index+1:p_set%dim)
END IF
IF (p_set%dim < n) THEN
  ! There are not other elements in next node, so remove the indexed
  ! entry directly from 'Set' by updating 'dim'.
  p_set%dim = p_set%dim - 1
ELSE
  ! There might be nodes in which to shift elements.
  ! Check if any element(s) in next node to shift.
  IF (ASSOCIATED(p_set%next)) THEN
    p_set => p_set%next
    DO
      IF (p_set%dim > 0) THEN
        ! There are elements to shift.
        ! Shift one element from p_next into its previous node.
        p_set%prev%elements(n) = p_set%elements(1)
        ! Shift elements inside p_next.
        p_set%elements(1:p_set%dim-1) = p_set%elements(2:p_set%dim)
      ELSE
        ! There are no elements to shift. Update 'dim' of previous node.
        p_set%prev%dim = p_set%prev%dim - 1
        EXIT
      END IF
      ! Move on to the next node if any. If there are no more nodes, update
      ! 'dim'.
      IF (ASSOCIATED(p_set%next)) THEN
```

```
        p_set => p_set%next
      ELSE
        p_set%dim = p_set%dim - 1
        EXIT
      END IF
    END DO
  ELSE
    ! There are no  more nodes. Update 'dim' of 'p_set'.
    p_set%dim = p_set%dim - 1
  END IF
END IF
RETURN
END SUBROUTINE rmNodeR
END MODULE VTDIRECT_GLOBAL
```

Appendix C: Fortran 90 MODULE VTDIRect_MOD

The Fortran 90 module VTDIRect_MOD that declares the subroutine VTDIRect, the present implementation of the DIRECT algorithm discussed in Chapter 2 is listed here.

```
MODULE VTDIRect_MOD
USE VTDIRECT_GLOBAL
CONTAINS
SUBROUTINE VTDIRect(N, L, U, X, FMIN, STATUS, OBJ_FUNC, SWITCH, &
                MAX_ITER, MAX_EVL, MIN_DIA, OBJ_CONV, EPS)
!
! This is an implementation of the DIRECT global unconstrained
! optimization algorithm described in:
!
!   D.R. Jones, C.D. Perttunen, and B.E. Stuckman, Lipschitzian
!   optimization without the Lipschitz constant, Journal of Optimization
!   Theory and Application, Vol. 79, No. 1, 1993, pp. 157-181.
!
! The algorithm to minimize f(x) inside the box L <= x <= U is as follows:
!
!   1. Normalize the search space to be the unit hypercube. Let c_1 be
!      the center point of this hypercube and evaluate f(c_1).
!   2. Identify the set S of potentially optimal rectangles.
!   3. For all rectangles j in S:
!      3a. Identify the set I of dimensions with the maximum side length.
!          Let delta equal one-third of this maximum side length.
!      3b. Sample the function at the points c +- delta * e_i for all i
!          in I, where c is the center of the rectangle and e_i is the ith
!          unit vector.
!      3c. Divide the rectangle containing c into thirds along the
!          dimensions in I, starting with the dimension with the lowest
!          value of f(c +- delta * e_i) and continuing to the dimension
!          with the highest f(c +- delta * e_i).
!   4. Repeat 2.-3. until stopping criterion is met.
!
!
! On input:
!
! N is the dimension of L, U, and X.
!
! L(1:N) is a real array giving lower bounds on X.
!
! U(1:N) is a real array giving upper bounds on X.
!
! OBJ_FUNC is the name of the real function procedure defining the
!   objective function f(x) to be minimized. OBJ_FUNC(C,IFLAG) returns
!   the value f(C) with IFLAG=0, or IFLAG .NE. 0 if f(C) is not defined.
!   OBJ_FUNC is precisely defined in the INTERFACE block below.
!
! Optional arguments:
!
! SWITCH =
!   1   select potentially optimal boxes on the convex hull of the
!       (box diameter, function value) points (default).
!   0   select as potentially optimal the box of each diameter with the
!       smallest function value. This is an aggressive selection
!       procedure which generates many more boxes to subdivide.
!
! MAX_ITER is the maximum number of iterations (repetitions of Steps 2-3)
!   allowed; defines stopping rule 1. If MAX_ITER is present but <= 0
!   on input, there is no iteration limit and the number of iterations
!   executed is returned in MAX_ITER.
!
! MAX_EVL is the maximum number of function evaluations allowed; defines
!   stopping rule 2. If MAX_EVL is present but <= 0 on input, there is no
!   limit on the number of function evaluations, which is returned in
!   MAX_EVL.
!
! MIN_DIA is the minimum box diameter allowed; defines stopping rule 3.
!   If MIN_DIA is present but <= 0 on input, a minimum diameter below
!   the roundoff level is not permitted, and the box diameter of the
!   box containing the smallest function value FMIN is returned in MIN_DIA.
!
! OBJ_CONV is the smallest acceptable relative improvement in the minimum
!   objective function value 'FMIN' between iterations; defines
!   stopping rule 4. OBJ_CONV must be positive and greater than the round
!   off level.  If absent, it is taken as zero.
!
! EPS is the tolerance defining the minimum acceptable potential
!   improvement in a potentially optimal box.  Larger EPS values eliminate
!   more boxes from consideration as potentially optimal, and bias the
!   search toward exploration.  EPS must be positive and greater than the
!   round off level.  If absent, it is taken as zero.  EPS > 0 is
!   incompatible with SWITCH = 0.
!
!
! On output:
!
! X(1:N) is a real vector containing the sampled box center with the
!   minimum objective function value FMIN.
!
! FMIN is the minimum function value.
!
! STATUS is a return status flag. The units decimal digit specifies
!   which stopping rule was satisfied on a successful return. The tens
!   decimal digit indicates a successful return, or an error condition with
!   the cause of the error condition reported in the units digit.
!
! Tens digit =
!  0 Normal return.
!   Units digit =
!    1  Stopping rule 1 (iteration limit) satisfied.
!    2  Stopping rule 2 (function evaluation limit) satisfied.
!    3  Stopping rule 3 (minimum diameter reached) satisfied. The
!       minimum diameter corresponds to the box for which X and
!       FMIN are returned.
!    4  Stopping rule 4 (relative change in 'FMIN') satisfied.
!  1 Input data error.
!   Units digit =
!    0  N < 2.
!    1  Assumed shape array L, U, or X does not have size N.
!     2  Some lower bound is >= the corresponding upper bound.
!     3  MIN_DIA, OBJ_CONV, or EPS is below the roundoff level.
!     4  None of MAX_EVL, MAX_ITER, MIN_DIA, and OBJ_CONV are specified;
!        there is no stopping rule.
!     5  Invalid SWITCH value.
!     6  SWITCH = 0 and EPS > 0 are incompatible.
!  2 Memory allocation failure.
!    Units digit =
!     0  BoxMatrix type allocation.
!     1  BoxLink type allocation.
!     2  int_vector or real_vector type allocation.
!     3  HyperBox type allocation.
!
! MAX_ITER (if present) contains the number of iterations.
!
! MAX_EVL (if present) contains the number of function evaluations.
!
! MIN_DIA (if present) contains the diameter of the box associated with
!        X and FMIN.
!
!
IMPLICIT NONE
INTEGER, INTENT(IN) :: N
REAL(KIND = R8), DIMENSION(:), INTENT(IN) :: L
REAL(KIND = R8), DIMENSION(:), INTENT(IN) :: U
REAL(KIND = R8), DIMENSION(:), INTENT(OUT) :: X
REAL(KIND = R8), INTENT(OUT) :: FMIN
INTEGER, INTENT(OUT) :: STATUS
INTERFACE
  FUNCTION OBJ_FUNC(C, IFLAG) RESULT(F)
    USE REAL_PRECISION, ONLY : R8
    REAL(KIND = R8), DIMENSION(:), INTENT(IN) :: C
    INTEGER, INTENT(OUT) :: IFLAG
    REAL(KIND = R8) :: F
  END FUNCTION OBJ_FUNC
END INTERFACE
INTEGER, INTENT(IN), OPTIONAL :: SWITCH
INTEGER, INTENT(INOUT), OPTIONAL :: MAX_ITER
INTEGER, INTENT(INOUT), OPTIONAL :: MAX_EVL
REAL(KIND = R8), INTENT(INOUT), OPTIONAL :: MIN_DIA
REAL(KIND = R8), INTENT(IN), OPTIONAL :: OBJ_CONV
REAL(KIND = R8), INTENT(IN), OPTIONAL :: EPS
! Local variables.
INTEGER :: alloc_err    ! Allocation error status.
INTEGER :: b_id        ! Box matrix identifier.
INTEGER :: col         ! Local column index.
INTEGER :: col_w, row_w  ! Factors defining reasonable memory space for
                     ! each box matrix allocation.
INTEGER :: convex   ! Switch for processing convex hull boxes.
INTEGER :: eval_c   ! Function evaluation counter.
INTEGER :: i, j     ! Loop counters.
INTEGER :: iflag    ! Error flag for subroutine calls.
INTEGER :: i_start  ! Records the start index for searching in a node of
                 ! 'setInd'.
INTEGER :: stop_rule  ! Bits 0, 1, 2 being set correspond to stopping rules
                   ! 1 (iteration limit), 2 (function evaluation limit),
                    ! 3 (minimum box diameter) respectively.
INTEGER :: t  ! Loop counter for main loop.
LOGICAL :: do_it ! Sign to process first box in each column of BoxMatrix.
TYPE(BoxLine) :: setB    ! Set of newly sampled boxes.
TYPE(BoxMatrix), POINTER :: m_head  ! The first box matrix.
TYPE(BoxMatrix), POINTER :: p_b ! Pointer to box matrix.
TYPE(Hyperbox), POINTER :: p_box ! Box for the removed parent box to divide.
TYPE(Hyperbox), POINTER :: tempbox ! Box for swapping heap elements.
TYPE(int_vector), POINTER :: p_start  ! Records the start node for searching
                                  ! the column with CONVEX_BIT set in
                                  ! 'setInd'.
TYPE(int_vector), POINTER :: p_setInd ! Pointer to a node of 'setInd'.
TYPE(int_vector), POINTER :: setFcol ! A linked list. Each node holds free
   ! column indices in BoxMatrices.
TYPE(int_vector) :: setI  ! Set I of dimensions with the maximum side length.
TYPE(int_vector), POINTER :: setInd ! A linked list. Each node holds column
   ! indices corresponding to different squared diameters in 'setDia'.
TYPE(real_vector), POINTER :: setDia ! A linked list. Each node holds
   ! current different squared diameters from largest to smallest.
TYPE(ValList) :: setW    ! Function values for newly sampled center points.
REAL(KIND = R8) :: dia ! Diameter squared associated with 'FMIN'.
REAL(KIND = R8) :: dia_limit ! Minimum diameter permitted.
REAL(KIND = R8) :: EPS4N  ! Tolerance for equality tests.
REAL(KIND = R8) :: epsl   ! Epsilon test for potentially optimal boxes.
REAL(KIND = R8) :: fmin_old  ! FMIN backup.
REAL(KIND = R8), DIMENSION(N) :: unit_x ! X normalized to unit hypercube.
! Sanity check of input arguments.
STATUS = 0
IF (N < 2) THEN
  STATUS = INPUT_ERROR
  RETURN
END IF
IF ((SIZE(X) /= N) .OR. (SIZE(L) /= N) .OR. (SIZE(U) /= N)) THEN
  STATUS = INPUT_ERROR + 1
  RETURN
END IF
IF (ANY(L >= U)) THEN
  STATUS = INPUT_ERROR + 2
  RETURN
END IF
! Parse optional arguments.
IF (PRESENT(SWITCH)) THEN
  IF ((SWITCH < 0) .OR. (SWITCH > 1)) THEN
    STATUS = INPUT_ERROR + 5
    RETURN
  END IF
  IF ((SWITCH == 0) .AND. PRESENT(EPS)) THEN
    STATUS = INPUT_ERROR + 6
    RETURN
  END IF
  ! User specified.
  convex = SWITCH
ELSE
  ! Default: processing boxes only on convex hull.
  convex = 1
END IF
```

```
stop_rule = 0
! When MAX_ITER <=0, the number of iterations will be returned on exit.
IF (PRESENT(MAX_ITER)) THEN
  IF (MAX_ITER > 0) THEN
    ! Set bit 0 of stop_rule.
    stop_rule = IBSET(stop_rule, STOP_RULE1)
  END IF
END IF
! When MAX_EVL <=0, the number of evaluations will be returned on exit.
IF (PRESENT(MAX_EVL)) THEN
  IF (MAX_EVL > 0) THEN
    ! Set bit 1 of stop_rule.
    stop_rule = IBSET(stop_rule, STOP_RULE2)
  END IF
END IF
! Find the maximum side of the feasible box L <= X <= U.
! Even if user doesn't specify 'MIN_DIA', a diameter smaller than
! MAX(U(i) - L(i))*EPSILON(1.0_R8)*N is not permitted to occur.
! When MIN_DIA <=0, the diameter associated with X and FMIN will be
! returned on exit.
dia_limit =  MAXVAL(U - L)*EPSILON(1.0_R8)*N
IF (PRESENT(MIN_DIA)) THEN
  IF (MIN_DIA > 0) THEN
    IF (MIN_DIA < dia_limit) THEN
      STATUS = INPUT_ERROR + 3
      RETURN
    ELSE
      ! Set bit 2 of stop_rule.
      stop_rule = IBSET(stop_rule, STOP_RULE3)
    END IF
  END IF
END IF
! When OBJ_CONV is present a minimum relative change in the minimum
! objective function value will be enforced.
IF (PRESENT(OBJ_CONV)) THEN
  IF ((OBJ_CONV < EPSILON(1.0_R8)*REAL(N,KIND=R8)) .OR. &
     (OBJ_CONV >= 1.0_R8)) THEN
    STATUS = INPUT_ERROR + 3
    RETURN
  ELSE
    ! Set bit 3 of stop_rule.
    stop_rule = IBSET(stop_rule, STOP_RULE4)
  END IF
END IF
! When EPS is present a test involving EPS is used to define potentially
! optimal boxes.  The absence of this test is equivalent to EPS=0.
IF (PRESENT(EPS)) THEN
  IF (EPS <= EPSILON(1.0_R8)) THEN
    STATUS = INPUT_ERROR + 3
    RETURN
  ELSE
    epsl = EPS
  END IF
ELSE
  epsl = 0.0_R8
END IF
```

```
! Check if stop_rule has at least at 1 bit set. Otherwise no stopping rule
! has been given.
IF (stop_rule == 0) THEN
  STATUS = INPUT_ERROR + 4
  RETURN
END IF
! End of argument sanity checks.
! Assign row_w and col_w in terms of N.
IF (N <= 10) THEN
  row_w = MAX(10, 2*N)
ELSE
  row_w = 17 + CEILING(LOG(REAL(N))/LOG(2.0))
END IF
col_w = 35*N
! Tolerance for REAL number equality tests.
EPS4N = REAL(4*N, KIND=R8)*EPSILON(1.0_R8)
! Allocate 'setI', 'setB' and 'setW'.
ALLOCATE(setI%elements(N), STAT = alloc_err)
IF (alloc_err /= 0) THEN ; STATUS=ALLOC_ERROR + 2 ; RETURN ; END IF
ALLOCATE(setI%flags(N), STAT = alloc_err)
IF (alloc_err /= 0) THEN ; STATUS=ALLOC_ERROR + 2 ; RETURN ; END IF
setI%dim = 0
ALLOCATE(setB%Line(2*N), STAT = alloc_err)
IF (alloc_err /= 0) THEN ; STATUS=ALLOC_ERROR + 1 ; RETURN ; END IF
ALLOCATE(setB%dir(2*N),STAT = alloc_err)
IF (alloc_err /= 0) THEN ; STATUS=ALLOC_ERROR + 1 ; RETURN ; END IF
DO i = 1, 2*N
  ALLOCATE(setB%Line(i)%c(N), STAT = alloc_err)
  IF (alloc_err /= 0) THEN ; STATUS=ALLOC_ERROR + 1 ; RETURN ; END IF
  ALLOCATE(setB%Line(i)%side(N), STAT = alloc_err)
  IF (alloc_err /= 0) THEN ; STATUS=ALLOC_ERROR + 1 ; RETURN ; END IF
END DO
setB%ind = 0
ALLOCATE(setW%val(N),STAT = alloc_err)
IF (alloc_err /= 0) THEN ; STATUS=ALLOC_ERROR + 2 ; RETURN ; END IF
ALLOCATE(setW%dir(N),STAT = alloc_err)
IF (alloc_err /= 0) THEN ; STATUS=ALLOC_ERROR + 2 ; RETURN ; END IF
setW%dim = 0
! Allocate 'setDia', 'setInd', and 'setFcol' for the first box matrix.
ALLOCATE(setDia)
ALLOCATE(setDia%elements(col_w),STAT = alloc_err)
IF (alloc_err /= 0) THEN ; STATUS=ALLOC_ERROR + 2 ; RETURN ; END IF
NULLIFY(setDia%next)
NULLIFY(setDia%prev)
setDia%id = 1
setDia%dim = 0
ALLOCATE(setInd)
ALLOCATE(setInd%elements(col_w),STAT = alloc_err)
IF (alloc_err /= 0) THEN ; STATUS=ALLOC_ERROR + 2 ; RETURN ; END IF
ALLOCATE(setInd%flags(col_w),STAT = alloc_err)
IF (alloc_err /= 0) THEN ; STATUS=ALLOC_ERROR + 2 ; RETURN ; END IF
setInd%flags(:) = 0
NULLIFY(setInd%next)
NULLIFY(setInd%prev)
setInd%id = 1
setInd%dim = 0
```

```
ALLOCATE(setFcol)
ALLOCATE(setFcol%elements(col_w),STAT = alloc_err)
IF (alloc_err /= 0) THEN ; STATUS=ALLOC_ERROR + 2 ; RETURN ; END IF
ALLOCATE(setFcol%flags(col_w),STAT = alloc_err)
IF (alloc_err /= 0) THEN ; STATUS=ALLOC_ERROR + 2 ; RETURN ; END IF
NULLIFY(setFcol%next)
NULLIFY(setFcol%prev)
setFcol%id = 1
setFcol%dim = 0
! Allocate p_box.
ALLOCATE(p_box)
ALLOCATE(p_box%c(N),STAT = alloc_err)
IF (alloc_err /= 0) THEN ; STATUS = ALLOC_ERROR + 3 ; RETURN ; END IF
ALLOCATE(p_box%side(N), STAT = alloc_err)
IF (alloc_err /= 0) THEN ; STATUS = ALLOC_ERROR + 3 ; RETURN ; END IF
! Allocate tempbox.
ALLOCATE(tempbox)
ALLOCATE(tempbox%c(N),STAT = alloc_err)
IF (alloc_err /= 0) THEN ; STATUS = ALLOC_ERROR + 3 ; RETURN ; END IF
ALLOCATE(tempbox%side(N), STAT = alloc_err)
IF (alloc_err /= 0) THEN ; STATUS = ALLOC_ERROR + 3 ; RETURN ; END IF
! Step 1: Normalization of the search space and initialization of first
!        hyperbox.
ALLOCATE(m_head, STAT = alloc_err)
IF (alloc_err /= 0) THEN ; STATUS = ALLOC_ERROR ; RETURN ; END IF
iflag = 0
CALL init(m_head, iflag)
! Check the returned iflag.
IF (iflag == 1) THEN ; STATUS = ALLOC_ERROR ; RETURN ; END IF
! Initialize 'setDia', 'setInd' and 'setFcol'.
setDia%dim = 1
setDia%elements(1) = m_head%M(1,1)%diam
! Set the first box as being on convex hull by setting  the CONVEX_BIT
! of the 'flags'.
setInd%dim = 1
setInd%elements(1) = 1
setInd%flags(1) = IBSET(setInd%flags(1), CONVEX_BIT)
! Starting from the last column, push free columns to 'setFcol'.
DO i = 1, col_w-1
  setFcol%elements(i) = col_w - (i-1)
END DO
setFcol%dim = col_w - 1
! Initialization for main loop.
t = 1
eval_c = 1
MAIN_LOOP: DO
  !Step 2: Identify the set of potentially optimal boxes.
  !       They are the first boxes of all columns with CONVEX_BIT set
  !       in 'flags' in 'setInd'.
  !
  ! Initialize 'i_start' and 'p_start' in order to search such
  ! columns in 'setInd'. The first boxes on columns with CONVEX_BIT set
  ! in 'flags' are potentially optimal.
  i_start = 1
  p_start => setInd
  ! Loop processing any boxes in the columns with CONVEX_BIT set in 'flags'
  ! in 'setInd'.
INNER:  DO
    do_it = .FALSE.
    ! Find such a box column in linked list 'setInd' starting
    ! from position 'i_start' in the node 'p_start'. If found,
    ! 'do_it' will be set TRUE and index 'i' and node 'p_setInd'
    ! will be returned.
    p_setInd => findcol(i_start, p_start, i, do_it)
    IF (do_it) THEN
      ! Step 3:
      ! Step 3a: Obtain the 'setI' of dimensions with the maximum
      !         side length for the first box on column
      !         'p_setInd%elements(i)', where 'i' is the index
      !         in 'setInd' for the column holding the hyperbox to
      !         subdivide.
      CALL findsetI(m_head, p_setInd%elements(i), setI)
      ! Step 3b: Sample new center points at c + delta * e_i and
      !         c - delta * e_i for all dimensions in 'setI', where
      !         c is the center of the parent hyperbox being processed,
      !         and e_i is the ith unit vector. Evaluate the objective
      !         function at new center points and keep track of current
      !         global minimum 'FMIN' and its associated 'unit_x'.
      CALL sampleP(p_setInd%elements(i), setI, m_head, setB)
      ! If the optional argument OBJ_CONV is present, save 'FMIN' to be
      ! compared with the updated 'FMIN' in subroutine sampleF.
      IF (PRESENT(OBJ_CONV)) fmin_old = FMIN
      CALL sampleF(setB, eval_c)
      IF (PRESENT(OBJ_CONV)) THEN
        IF (fmin_old /= FMIN) THEN
          ! 'FMIN' has been updated.
          IF (fmin_old - FMIN < (1.0_R8+ABS(fmin_old))*OBJ_CONV) THEN
            STATUS = 4
            EXIT MAIN_LOOP
          END IF
        END IF
      END IF
      ! Step 3c: Divide the hyperbox containing c into thirds along the
      !         dimensions in 'setI', starting with the dimension with
      !         the lowest function value of f(c +- delta * e_i) and
      !         continuing to the dimension with the highest function
      !         value f(c +- delta * e_i).
      CALL divide(i, p_setInd%id, m_head, setB, setDia, &
                 setInd, setFcol, p_box, setW, setI, iflag)
      IF (iflag /= 0) THEN ; STATUS=ALLOC_ERROR+iflag-1 ; RETURN ; END IF
    ELSE
      ! There are no more columns of boxes to divide for this iteration.
      EXIT
    END IF
  END DO INNER
  ! Update iteration counter.
  t = t + 1
  ! Check stop rules:
  ! Stop rule 1: maximum iterations.
  IF (BTEST(stop_rule, 0)) THEN
    IF (t > MAX_ITER) THEN
      STATUS = 1
      EXIT MAIN_LOOP
    END IF
  END IF
  ! Stop rule 2: maximum evaluations.
  IF (BTEST(stop_rule, 1)) THEN
    IF (eval_c >= MAX_EVL ) THEN
      STATUS = 2
      EXIT MAIN_LOOP
    END IF
  END IF
  ! Stop rule 3: minimum diameter.
  IF (BTEST(stop_rule, 2)) THEN
    IF (sqrt(dia) <= MIN_DIA) THEN
      STATUS = 3
      EXIT MAIN_LOOP
    END IF
  ELSE
    ! Check if minimum diameter has been reached.
    IF (sqrt(dia) <= dia_limit) THEN
      STATUS = 3
      EXIT MAIN_LOOP
    END IF
  END IF
  ! Preprocess for identifying potentially optimal hyperboxes of Step
  ! 3a for the next iteration. Find and process the hyperboxes which
  ! are on the convex hull if 'convex' == 1; otherwise, process the first
  ! box of each column.
  IF (convex == 1) THEN
    ! Processing only the boxes on the convex hull of (box diameter,
    ! function value) points. Set the CONVEX_BIT in 'flags', starting
    ! from the first one in 'setInd', until reaching the column with 'FMIN'.
    p_setInd => setInd
    OUTER: DO
      DO i = 1, p_setInd%dim
        p_setInd%flags(i) = IBSET(p_setInd%flags(i), CONVEX_BIT)
        ! Check if the column has reached the one with 'FMIN'.
        b_id = (p_setInd%elements(i)-1)/col_w + 1
        col = MOD(p_setInd%elements(i)-1, col_w) + 1
        p_b => m_head
        DO j = 1, b_id -1
          p_b => p_b%child
        END DO
        IF (ALL(unit_x == p_b%M(1,col)%c)) EXIT OUTER
      END DO
      IF (ASSOCIATED(p_setInd%next)) THEN
        p_setInd => p_setInd%next
      ELSE
        EXIT OUTER
      END IF
    END DO OUTER
    ! Remove the columns not on the convex hull of potentially optimal
    ! curve. 'p_setInd' and 'i' point to the box with 'FMIN'. Pass them to
    ! convex for terminating the loop of identifying boxes on convex hull
    ! when epsl/=0.
    CALL findconvex(m_head, epsl, p_setInd, i, setInd)
  ELSE
```

```
    ! Processing all columns of boxes because convex == 0.
    ! Set CONVEX_BIT of all current columns' 'flags' for processing.
    p_setInd => setInd
    DO WHILE(ASSOCIATED(p_setInd))
      p_setInd%flags(1:p_setInd%dim) = &
        IBSET(p_setInd%flags(1:p_setInd%dim), CONVEX_BIT)
      p_setInd => p_setInd%next
    END DO
  END IF
END DO MAIN_LOOP
! Preparation for return to the caller.
! Scale 'unit_x' back to 'X' in original coordinates.
X = L + unit_x*(U - L)
! Return current diameter of the box with 'FMIN'.
IF (PRESENT(MIN_DIA)) MIN_DIA = sqrt(dia)
! Return the total iterations and evaluations.
IF (PRESENT(MAX_ITER)) MAX_ITER = t - 1
IF (PRESENT(MAX_EVL)) MAX_EVL = eval_c
! Deallocate all the data structures explicitly allocated, including
! box matrices, box links and setI, setB, setW, setInd, setFcol, setDia,
! and p_box.
CALL cleanup()
RETURN
CONTAINS
SUBROUTINE binaryS(p_rset, diam, pos, found)
IMPLICIT NONE
! Using a binary search, matches the squared diameter 'diam' with an
! element in the node 'p_rset' of 'setDia', and returns the position
! 'pos' if a match is found. If there is no match, returns the right
! 'pos' at which to insert 'diam'. When 'pos' is returned as 0, 'diam'
! should be inserted after all others. If 'pos' is not 0, 'diam' could be
! inserted at the position 'pos' of 'p_rset' depending on 'found'. See
! insMat().
! On input:
! p_rset - A pointer to one of the nodes in 'setDia'.
! diam   - The diameter squared to match against.
!
! On output:
! pos   - The position in 'p_rset' for a match or insertion.
! found - Status indicating whether 'diam' is found in 'p_rest' or not.
!
TYPE(real_vector), INTENT(IN) :: p_rset
REAL(KIND = R8), INTENT(IN) :: diam
INTEGER, INTENT(OUT) :: pos
LOGICAL, INTENT(OUT) :: found
! Local variables.
INTEGER :: low, mid, up
! Initialization for searching.
found = .FALSE.
! Initialize limits outside bounds of array for binary search.
low = 0
up = p_rset%dim + 1
IF (p_rset%dim > 0) THEN
  ! Check with the first and the last.
  IF (p_rset%elements(1) <= diam) THEN
  ! 'diam' is the biggest.
  IF (ABS(p_rset%elements(1) - diam)/diam <= EPS4N) THEN
    ! 'diam' is the same as the first one.
    found = .TRUE.
  END IF
  pos = 1
  RETURN
ELSE
  IF (p_rset%elements(up-1) >= diam) THEN
    IF (ABS(p_rset%elements(up-1) - diam)/p_rset%elements(up-1) &
       <= EPS4N) THEN
      ! 'diam' is the smallest one. Same as the last one in 'p_rset'.
      found = .TRUE.
      pos = up-1
    ELSE
      ! 'diam' is smaller than all in 'p_rset'. Set 'pos' 0 to insert
      ! 'diam' after all others.
      found = .FALSE.
      pos = 0
    END IF
    RETURN
  ELSE
    ! 'diam' falls in between the biggest and the smallest. Apply binary
    ! search.
    DO WHILE((low + 1) < up)
      mid = (low + up) / 2
      IF (ABS(diam - p_rset%elements(mid))/          &
        MAX(diam, p_rset%elements(mid)) <= EPS4N) THEN
        ! 'diam' found.
        up = mid
        EXIT
      END IF
      IF (diam < p_rset%elements(mid))THEN
        low = mid
      ELSE
        up = mid
      END IF
    END DO
    ! Check if it's found.
    mid = up
    IF (ABS(diam - p_rset%elements(mid))/MAX(diam, p_rset%elements(mid)) &
       <= EPS4N) THEN
      ! Found it, so assign 'mid' to 'pos' in order to insert the
      ! associated box in the same column as 'mid'.
      found = .TRUE.
      pos = mid
    ELSE
      found = .FALSE.
      IF (diam > p_rset%elements(mid) )THEN
        ! 'diam' is bigger than the one at 'mid'. Set 'pos' to be 'mid'
        ! in order to insert 'diam' before 'mid'.
        pos = mid
      ELSE
        ! 'diam' is smaller than the one at 'mid'. Set 'pos' to be one
        ! after 'mid' in order to insert 'diam' after 'mid'.
        pos = mid + 1
      END IF
      END IF
    END IF
  END IF
ELSE
  ! 'p_rset' is empty. Set 'pos'=0 to insert 'diam' at the end of 'p_rset'.
  found = .FALSE.
  pos = 0
END IF
RETURN
END SUBROUTINE binaryS
SUBROUTINE checkblinks(col, b, status)
IMPLICIT NONE
! Checks if this column needs a new box link. Creates one if needed.
!
! On input:
! col - The local column index.
! b   - The current link of box matrices.
!
! On output:
! b      - 'b' has the newly added box link for 'col' if needed.
! status - Return status.
!          0   Successful.
!          1   Allocation error.
!
INTEGER, INTENT(IN) :: col
TYPE(BoxMatrix), INTENT(INOUT) :: b
INTEGER, INTENT(OUT) :: status
! Local variables.
INTEGER :: iflag
TYPE(BoxLink), POINTER :: newBoxLink, p_link, p_prev
! Set normal status.
status = 0
! Find the last box link.
NULLIFY(p_prev)
p_link => b%sibling(col)%p
DO WHILE(ASSOCIATED(p_link))
  ! Keep going until next link is null. Meanwhile, save the pointer of
  ! the previous link in order to trace back.
  p_prev => p_link
  p_link => p_link%next
END DO
! 'p_prev' could point to the last box link if any.
! Check if this box link(or just column part inside of M) of this column is
! full.
IF (ASSOCIATED(p_prev)) THEN
  ! 'p_prev' points to the last box link.
  IF (p_prev%ind == row_w) THEN
    ! It's full. Need a new box link. Allocate a new one.
    ALLOCATE(newBoxLink, STAT = alloc_err)
    IF (alloc_err /= 0) THEN; status = 1; RETURN; END IF
    CALL initLink(newBoxLink, iflag)
    IF (iflag /= 0) THEN; status = 1; RETURN; END IF
    ! Link the new box link to the last one as the next link.
    p_prev%next => newBoxLink
    ! Link the last box link to the new one as the previous link.
    newBoxLink%prev => p_prev
  END IF
ELSE
  ! No box links exist.
  IF (b%ind(col) == row_w)THEN
    ! 'M' part of column is full. Need a box link.
    ! Make a new box link linked to it. Allocate a new one.
    ALLOCATE(newBoxLink, STAT = alloc_err)
    IF (alloc_err /= 0) THEN; status = 1; RETURN; END IF
    CALL initLink(newBoxLink, iflag)
    IF (iflag /= 0) THEN; status = 1; RETURN; END IF
    ! This is the first box link that does not have previous link.
    NULLIFY(newBoxLink%prev)
    ! Link it as 'sibling' of this column.
    b%sibling(col)%p => newBoxLink
  END IF
END IF
RETURN
END SUBROUTINE checkblinks
SUBROUTINE cleanup()
! Cleans up all data structures allocated by VTDIRect.
!
IMPLICIT NONE
! Local variables.
INTEGER :: i, j  ! Loop counters.
TYPE(BoxMatrix), POINTER :: p_b, p_save
TYPE(BoxLink), POINTER :: p_l
TYPE(int_vector), POINTER :: p_seti
TYPE(real_vector), POINTER :: p_setr
! Deallocate box links and box matrices starting from the first box matrix.
! First deallocate all box links associated with each box matrix, and
! finally deallocate the box matrix.
p_b => m_head
! Check all columns with box links which will be deallocated one by one
! starting from the last box link.
DO WHILE(ASSOCIATED(p_b))
  ! Check all the columns in 'p_b'.
  DO i = 1, col_w
    IF (p_b%ind(i) > row_w) THEN
      ! There must be box link(s). Chase to the last one and start
      ! deallocating them one by one.
      p_l => p_b%sibling(i)%p
      DO WHILE(ASSOCIATED(p_l%next))
        p_l => p_l%next
      END DO
      ! Found the last box link 'p_l'. Trace back and deallocate all links.
      DO WHILE(ASSOCIATED(p_l))
        IF (ASSOCIATED(p_l%prev)) THEN
          ! Its previous link is still a box link.
          p_l => p_l%prev
        ELSE
          ! There is no box link before it. This is the first box link of
          ! this column.
          DO j = 1, row_w
            DEALLOCATE(p_l%Line(j)%c)
            DEALLOCATE(p_l%Line(j)%side)
          END DO
          DEALLOCATE(p_l%Line)
          DEALLOCATE(p_l)
          EXIT
        END IF
        DO j = 1, row_w
          DEALLOCATE(p_l%next%Line(j)%c)
          DEALLOCATE(p_l%next%Line(j)%side)
        END DO
        DEALLOCATE(p_l%next%Line)
        DEALLOCATE(p_l%next)
      END DO
    END IF
  END DO
  ! Save the pointer of this box matrix for deallocation.
  p_save => p_b
  ! Before it's deallocated, move to the next box matrix.
  p_b => p_b%child
  ! Deallocate this box matrix with all box links cleaned up.
  DEALLOCATE(p_save%ind)
  DEALLOCATE(p_save%sibling)
  DO i = 1, row_w
    DO j = 1, col_w
      DEALLOCATE(p_save%M(i,j)%c)
      DEALLOCATE(p_save%M(i,j)%side)
    END DO
  END DO
  DEALLOCATE(p_save%M)
  DEALLOCATE(p_save)
END DO
! Deallocate 'setI', 'setB' and 'setW'.
DEALLOCATE(setI%elements)
DEALLOCATE(setI%flags)
DO i = 1, 2*N
  DEALLOCATE(setB%Line(i)%c)
  DEALLOCATE(setB%Line(i)%side)
END DO
DEALLOCATE(setB%Line)
DEALLOCATE(setB%dir)
DEALLOCATE(setW%val)
DEALLOCATE(setW%dir)
! Deallocate nodes of 'setDia', 'setInd' and 'setFcol' starting from
! the last node.
p_setr => setDia
DO WHILE(ASSOCIATED(p_setr%next))
  p_setr => p_setr%next
END DO
! Found the last link pointed to by 'p_setr' of 'setDia', so deallocate
! links one by one until reaching the head node which has a null 'prev'.
DO
  DEALLOCATE(p_setr%elements)
  IF (ASSOCIATED(p_setr%prev)) THEN
    p_setr => p_setr%prev
  ELSE
    DEALLOCATE(p_setr)
    EXIT
  END IF
  DEALLOCATE(p_setr%next)
END DO
p_seti => setInd
DO WHILE(ASSOCIATED(p_seti%next))
  p_seti => p_seti%next
END DO
! Found the last link pointed to by 'p_seti' of 'setInd', so deallocate
! links one by one until reaching the head node which has a null 'prev'.
DO
  DEALLOCATE(p_seti%flags)
  DEALLOCATE(p_seti%elements)
  IF (ASSOCIATED(p_seti%prev)) THEN
    p_seti => p_seti%prev
  ELSE
    DEALLOCATE(p_seti)
    EXIT
  END IF
  DEALLOCATE(p_seti%next)
END DO
p_seti => setFcol
DO WHILE(ASSOCIATED(p_seti%next))
  p_seti => p_seti%next
END DO
! Found the last link pointed to by 'p_seti' of 'setFcol', so deallocate
! links one by one until reaching the head node that has a null 'prev'.
DO
  DEALLOCATE(p_seti%elements)
  DEALLOCATE(p_seti%flags)
  IF (ASSOCIATED(p_seti%prev)) THEN
    p_seti => p_seti%prev
  ELSE
    DEALLOCATE(p_seti)
    EXIT
  END IF
  DEALLOCATE(p_seti%next)
END DO
! Deallocate p_box.
DEALLOCATE(p_box%c)
DEALLOCATE(p_box%side)
DEALLOCATE(p_box)
! Deallocate tempbox
DEALLOCATE(tempbox%c)
DEALLOCATE(tempbox%side)
DEALLOCATE(tempbox)
RETURN
END SUBROUTINE cleanup
SUBROUTINE divide(parent_i, id, b, setB, setDia, &
                  setInd, setFcol, p_box, setW, setI, iflag)
IMPLICIT NONE
! Divides the first box on a column of one of box matrices 'b', starting from
! the dimension with minimum w to the one with maximum w, where w is
! minf(c+delta), f(c-delta).
!
! On input:
! parent_i - The index in one of nodes of 'setInd' and 'setDia' for the column
!          holding the parent box to divide. Each element in 'setInd' has
!          the same index as the one in 'setDia'.
! id      - The identifier of the node of type 'setInd'.
! b       - The head link of box matrices.
! setB     - A set of 'HyperBox' type structures, each with newly sampled
!          center point coordinates and the corresponding function value.
!          After dividing, it contains complete boxes with associated side
!          lengths and the squared diameters.
! setDia   - A linked list of current different squared diameters of box
!          matrices. It's sorted from the biggest to the smallest.
! setInd   - A linked list of column indices corresponding to the different
!          squared diameters in 'setDia'.
! setFcol  - A linked list of free columns in box matrices.
! p_box    - A 'HyperBox' type structure to hold removed parent box to
!          subdivide.
! setW     - A set of type 'ValList' used to sort wi's, where wi is defined as
!          minf(c+delta*ei), f(c-delta*ei), the minimum of function values
!          at the two newly sampled points.
!
! On output:
! b      - 'b' has the parent box removed and contains the newly formed boxes
!         after dividing the parent box.
! setB    - Cleared set of type 'BoxLine'. All newly formed boxes have been
!         inserted to 'b'.
! setDia  - Updated linked list 'setDia' with new squared diameters of boxes,
!         if any.
! setInd  - Updated linked list 'setInd' with new column indices corresponding
!         to newly added squared diameters in 'setDia'.
! setFcol - Updated linked list 'setFcol' with current free columns in 'b'.
! p_box   - A 'HyperBox' structure holding removed parent box to subdivide.
! setW    - 'setW' becomes empty after dividing.
! setI    - A set of dimensions with the order of dimensions for dividing. It
!         is cleared after dividing.
! iflag   - status to return.
!          0    Normal return.
!          1    Allocation failures.
!
INTEGER, INTENT(IN) :: parent_i
INTEGER, INTENT(IN) :: id
TYPE(BoxMatrix), INTENT(INOUT),TARGET :: b
TYPE(BoxLine), INTENT(INOUT) :: setB
TYPE(real_vector),INTENT(INOUT) :: setDia
TYPE(int_vector), INTENT(INOUT), TARGET :: setInd
TYPE(int_vector), INTENT(INOUT) :: setFcol
TYPE(HyperBox), INTENT(INOUT) :: p_box
TYPE(ValList), INTENT(INOUT) :: setW
TYPE(int_vector), INTENT(OUT) :: setI
INTEGER, INTENT(OUT) :: iflag
! Local variables.
INTEGER :: b_id, b_j, i, j, k, status, temp_dir
INTEGER, DIMENSION(2*n) :: sortInd
TYPE(BoxLink), POINTER :: p_l, p_prev
TYPE(BoxMatrix), POINTER :: p_b
TYPE(int_vector), POINTER :: p_i, p_setInd
REAL(KIND = R8) :: temp
! Initialize 'iflag' for a normal return.
iflag = 0
```

```
! Find the desired node of 'setInd'.
p_setInd => setInd
DO i = 1, id -1
  p_setInd => p_setInd%next
END DO
! Clear the CONVEX_BIT of 'flags' as being processed.
p_setInd%flags(parent_i) = IBCLR(p_setInd%flags(parent_i), CONVEX_BIT)
IF (p_setInd%elements(parent_i) <= col_w) THEN
  ! This column is in the head link of box matrices.
  p_b => b
  b_j = p_setInd%elements(parent_i)
ELSE
  ! Find the box matrix that contains this column.
  b_id = (p_setInd%elements(parent_i)-1)/col_w + 1
  b_j = MOD(p_setInd%elements(parent_i)-1, col_w) + 1
  p_b => b
  DO i = 1, b_id-1
    p_b => p_b%child
  END DO
END IF
! Fill out 'setW'.
DO i = 1, setB%ind, 2
  ! Add minimum 'val' of a pair of newly sampled center points
  ! into 'setW'.
  setW%val((i+1)/2) = MIN(setB%Line(i)%val, setB%Line(i+1)%val)
  setW%dir((i+1)/2) = setB%dir(i)
END  DO
setW%dim = setB%ind/2
! Find the order of dimensions for further dividing by insertion
! sorting wi's in 'setW'.
DO i = 2, setW%dim
  DO j = i, 2, -1
    IF (setW%val(j) < setW%val(j-1)) THEN
      ! Element j is smaller than element j-1, so swap them. Also,
      ! the associated directions are swapped.
      temp = setW%val(j)
      k = setW%dir(j)
      setW%val(j) = setW%val(j-1)
      setW%dir(j) = setW%dir(j-1)
      setW%val(j-1) = temp
      setW%dir(j-1) = k
    ELSE
      EXIT
    END IF
  END DO
END DO
! Sort the indices of boxes in setB according to the dividing order in
! 'setW%dir'. Record the sorted indices in 'sortInd'.
DO i = 1, setW%dim
  DO j = 1, setB%ind, 2
    IF (setB%dir(j) == setW%dir(i)) THEN
      sortInd(2*i-1) = j
      sortInd(2*i) = j + 1
    END IF
  END DO
END DO
```

```
! 'setW%dir' contains the order of dimensions to divide the parent box.
! Loop dividing on all dimensions in 'setW%dir' by setting up the new
! side lengths as 1/3 of parent box side lengths for each newly
! sampled box center.
DO i = 1, setW%dim
  temp = p_b%M(1,b_j)%side(setW%dir(i))/3.0_R8
  DO j = i, setW%dim
   setB%Line(sortInd(2*j-1))%side(setW%dir(i)) = temp
   setB%Line(sortInd(2*j))%side(setW%dir(i)) = temp
  END DO
  ! Modify the parent's side lengths.
  p_b%M(1,b_j)%side(setW%dir(i))= temp
END DO
! Clear 'setW' for next time.
setW%dim = 0
! Remove the parent box from box matrix 'p_b'.
p_box = p_b%M(1,b_j)
! Move the last box to the first position.
IF (p_b%ind(b_j) <= row_w) THEN
  ! There are no  box links.
  p_b%M(1,b_j) = p_b%M(p_b%ind(b_j),b_j)
ELSE
  ! There are box links. Chase to the last box link.
  p_l => p_b%sibling(b_j)%p
  DO i = 1, (p_b%ind(b_j)-1)/row_w - 1
    p_l => p_l%next
  END DO
  p_b%M(1, b_j) = p_l%Line(p_l%ind)
  p_l%ind = p_l%ind - 1
END IF
p_b%ind(b_j) = p_b%ind(b_j) - 1
! Heapify this column 'b_j' of box matrix 'p_b', if it has been updated
! in the previous iteration.
IF (BTEST(p_setInd%flags(parent_i),UPDATE_BIT)) THEN
  CALL heapify(p_b, b_j)
  ! Clear the UPDATE_BIT of 'flags'.
  p_setInd%flags(parent_i) = IBCLR(p_setInd%flags(parent_i),UPDATE_BIT)
ELSE
  ! Only siftdown operation is needed, because this column 'b_j' has not
  ! been changed.
  CALL siftdown(p_b, b_j, 1)
END IF
! Update 'setDia', 'setInd' and 'setFcol' if this column is empty.
! Find which node 'setInd' is associated with by checking 'setInd%id'.
IF (p_b%ind(b_j) == 0 )THEN
  ! This column is empty. Remove this diameter squared from a corresponding
  ! node of 'setDia'.
  CALL rmNode(col_w, p_setInd%id-1, parent_i, setDia)
  ! Push the released column back to top of 'setFcol'.
  IF (setFcol%dim < col_w) THEN
    ! The head node of 'setFcol' is not full.
    CALL insNode(col_w, p_setInd%elements(parent_i), &
                 setFcol%dim+1, setFcol)
  ELSE
    ! The head node is full. There must be at least one more node
    ! for 'setFcol'. Find the last non-full node of 'setFcol' to
```

```
      ! insert the released column.
      p_i => setFcol%next
      DO
        IF (p_i%dim < col_w) THEN
          ! Found it.
          CALL insNode(col_w, p_setInd%elements(parent_i), &
                       p_i%dim+1, p_i)
          EXIT
        END IF
        ! Go to the next node.
        p_i=> p_i%next
      END DO
    END IF
    ! Remove the column index from a corresponding node of 'setInd'.
    CALL rmNode(col_w, 0, parent_i, p_setInd)
  END IF
  ! Modify the diameter squared for the parent box temporarily saved in
  ! 'p_box'.
  p_box%diam = DOT_PRODUCT(p_box%side, p_box%side)
  ! Update 'dia' associated with 'FMIN' which has coordinates in 'unit_x'.
  IF (ALL(unit_x == p_box%c)) dia = p_box%diam
  ! Compute squared diameters for all new boxes in 'setB'.
  DO i = 1, setB%ind
    setB%Line(i)%diam = DOT_PRODUCT(setB%Line(i)%side, setB%Line(i)%side)
    ! Update 'dia' if needed.
    IF (ALL(unit_x == setB%Line(i)%c)) dia = setB%Line(i)%diam
  END DO
  ! Add all new boxes in 'setB' and 'p_box' to 'b' according to different
  ! squared diameters and different function values.
  DO i = 1, setB%ind
    CALL insMat(setB%Line(i), b, setDia, setInd, setFcol, status)
    IF (status /=0) THEN ; iflag = status; RETURN ; END IF
  END DO
  CALL insMat(p_box, b, setDia, setInd, setFcol, status)
  IF (status /=0) THEN ; iflag = status ; RETURN ; END IF
  ! Clear 'setB' and 'setI' for calling divide() next time.
  setB%ind = 0
  setI%dim = 0
  RETURN
  END SUBROUTINE divide
  FUNCTION findcol(i_start, p_start, index, do_it) RESULT(p_setInd)
  IMPLICIT NONE
  ! Finds the rightmost column (setInd%elements(index)), in the plot of
  ! (box diameter, function value) points, with CONVEX_BIT of 'flags' set
  ! in linked list 'setInd', which indicates a potentially optimal box to be
  ! subdivided. When finding this column, it checks the UPDATE_BIT in 'flags'
  ! of each column. If the column has been updated in the previous iteration
  ! it heapifies the column.
  !
  ! On input:
  ! i_start - The index to start searching in node 'p_start'.
  ! p_start - The pointer to the node at which to start searching.
  !
  ! On output:
  ! index    - The found index in node 'p_setInd' of the linked list 'setInd'.
  ! do_it    - The returned sign to continue processing or not.
```

```
! i_start  - The index at which to resume searching in node 'p_start'
!           next time.
! p_start  - The pointer to the node at which to resume searching next time.
! p_setInd - The returned node which contains the next box to subdivide.
!
INTEGER, INTENT(INOUT) :: i_start
TYPE(int_vector), POINTER :: p_start
INTEGER, INTENT(OUT) :: index
LOGICAL, INTENT(OUT) :: do_it
TYPE(int_vector), POINTER :: p_setInd
! Local variables.
INTEGER :: i, start
TYPE(int_vector), POINTER :: p_set
TYPE(BoxMatrix), POINTER :: p_b
start = i_start
p_set => p_start
DO WHILE(ASSOCIATED(p_set))
  DO i = start, p_set%dim
    ! Find the first column with CONVEX_BIT set in 'flags' of 'p_set'.
    IF (BTEST(p_set%flags(i), CONVEX_BIT)) THEN
      do_it = .TRUE.
      index = i
      p_setInd => p_set
      ! Save them to i_start and p_start for resuming searching
      ! next time.
      i_start = i
      p_start => p_set
      EXIT
    ELSE
      ! If this box column had new boxes added in the previous
      ! iteration, then call heapify to order the heap elements.
      IF (BTEST(p_set%flags(i), UPDATE_BIT)) THEN
        p_b => m_head
        ! Find the box matrix 'p_b' that holds this column.
        DO j = 1, (p_set%elements(i)-1)/col_w
          p_b => p_b%child
        END DO
        ! Heapify this column.
        CALL heapify(p_b, MOD((p_set%elements(i)-1), col_w)+1)
        ! Clear the update status bit.
        p_set%flags(i) = IBCLR(p_set%flags(i), UPDATE_BIT)
      END IF
    END IF
  END DO
  ! There are no more box column with CONVEX_BIT set in 'flags' in this node.
  ! Go to the next one
  IF (.NOT.do_it) THEN
    p_set => p_set%next
    ! Reset 'start' to be 1 for all the following iterations except the
    ! first one which resumed from 'i_start'.
    start = 1
  ELSE
    EXIT
  END IF
END DO
RETURN
```

```
END FUNCTION findcol
SUBROUTINE findconvex(b, epsl, p_fmin, i_fmin, setInd)
IMPLICIT NONE
! In 'setInd', clear CONVEX_BIT of columns if the first boxes on these
! columns are not on convex hull. Bit CONVEX_BIT with value 0 indicates
! the first box on the column is not one of potentially optimal boxes.
! This is determined by comparing slopes. If epsl is 0, starting from the
! first column, find the maximum slope from the first box on that column
! to the first boxes on all other columns until reaching the box with
! 'FMIN'. Then, starting from the next column with the first box on
! convex hull, repeat the procedure until no more columns before the
! column with 'FMIN' to check. If epsl is greater than 0, the outer loop
! breaks out when the maximum slope is less than the value:
! (val - (FMIN - epsl))/diam.
!
! On input:
! b      - The head link of box matrices.
! epsl   - Epsilon test for potentially optimal boxes.
! p_fmin - Pointer to the node holding the column index of the box with
!          'FMIN'.
! i_fmin - Index of the column in the node 'p_fmin'.
! setInd - A linked list holding column indices of box matrices.
!
! On output:
! setInd - 'setInd' has the modified column indices.
!
TYPE(BoxMatrix), INTENT(IN), TARGET :: b
REAL(KIND = R8), INTENT(IN) :: epsl
TYPE(int_vector), POINTER :: p_fmin
INTEGER, INTENT(IN) :: i_fmin
TYPE(int_vector), INTENT(INOUT), TARGET :: setInd
! Local variables.
INTEGER :: b_id1, b_id2, col1, col2, i, j, k, target_i
LOGICAL :: stop_fmin
REAL(KIND = R8) :: slope, slope_max
TYPE(BoxMatrix), POINTER :: p_b1, p_b2
TYPE(int_vector), POINTER :: p_setInd1, p_setInd2, target_set
! Initialize the first node pointer.
p_setInd1 => setInd
! Initialization for outer loop which processes all columns before
! the column containing 'FMIN' in order to find a convex hull curve.
stop_fmin = .FALSE.
i = 1
k = 1
DO WHILE((.NOT.stop_fmin).AND.ASSOCIATED(p_setInd1))
  ! Initialization for inner loop, which computes the slope from the first
  ! box on the fixed column 'i' to the first boxes on all the other columns,
  ! before reaching the column containing a box with 'FMIN', to locate the
  ! target column with maximum slope. Mark off any columns in between the
  ! fixed first column and the target column.
  NULLIFY(target_set)
  slope_max = -HUGE(slope)
  p_setInd2 => p_setInd1
  ! Fix the first convex hull column as column 'i' in 'p_setInd1'.
  ! The second column used to calculate the slope has index 'k' in
  ! 'p_setInd2'.  'k' is incremented up to the column index corresponding
  ! to 'FMIN'.  Find the box matrix 'p_b1' and the local column index
  ! 'col1'.
  b_id1 =(p_setInd1%elements(i)-1)/col_w + 1
  col1 = MOD(p_setInd1%elements(i)-1, col_w) + 1
  p_b1 => b
  DO j = 1, b_id1 -1
    p_b1 => p_b1%child
  END DO
  ! Check if the first column has reached the column with 'FMIN'. If
  ! so, break out of the outer loop.
  IF (ALL(unit_x == p_b1%M(1,col1)%c)) EXIT
  k = i + 1
  DO
    IF (k > p_setInd2%dim) THEN
      ! Move to the next node as k increments beyond the maximum
      ! length for each node of 'setInd'.
      p_setInd2 => p_setInd2%next
      !Jian: IF ((.NOT.ASSOCIATED(p_setInd2)) .OR. (p_setInd2%dim == 0)) EXIT
      IF (.NOT.ASSOCIATED(p_setInd2)) THEN
EXIT
      ELSE
IF (p_setInd2%dim == 0) EXIT
      END IF
      k = 1
    END IF
    ! To compute the slope from the first box on column 'i' of 'p_setInd1' to
    ! the first box on column 'k' of 'p_setInd2', find the local column index
    ! 'col2' and the corresponding box matrix 'p_b2'.
    b_id2 = (p_setInd2%elements(k)-1)/col_w + 1
    col2 = MOD(p_setInd2%elements(k)-1, col_w) + 1
    p_b2 => b
    DO j = 1, b_id2 -1
      p_b2 => p_b2%child
    END DO
    ! Use the slope formula (f1 - f2)/(d1 - d2), where f1 and f2 are the
    ! function values at the centers of the two boxes with diameters
    ! d1 and d2.
    slope = (p_b1%M(1,col1)%val - p_b2%M(1,col2)%val) / &
            (SQRT(p_b1%M(1,col1)%diam) - SQRT(p_b2%M(1,col2)%diam))
    ! Compare the new slope with the current maximum slope. Keep track
    ! of the target column index and the target node.
    IF (slope > slope_max) THEN
      slope_max = slope
      target_i = k
      target_set => p_setInd2
      ! Check if this target column contains 'FMIN'.
      IF (ALL(unit_x == p_b2%M(1,col2)%c)) stop_fmin = .TRUE.
    END IF
    IF (ALL(unit_x == p_b2%M(1,col2)%c)) THEN
      ! The second box for computing slope has reached the column with 'FMIN'.
      ! This pass of inner loop is over. Mark off all nonconvex hull columns
      ! in between the 'target_i' of node 'target_set' and the fixed column
      ! 'i' of 'p_setInd1'.
      IF (ASSOCIATED(target_set)) THEN
        CALL markoff(i, target_i, p_setInd1, target_set)
      END IF
      ! Break out the inner loop to start next pass.
      EXIT
    END IF
    ! Move on to the next column.
    k = k + 1
  END DO
  ! Check if epsl/=0. If so, it stops if the found 'slope_max' from
  ! the first box is less than the desired accuracy of the solution.
  IF (epsl /= 0) THEN
    IF ((p_b1%M(1,col1)%val-(FMIN-(ABS(FMIN)+1)*epsl))/ &
       SQRT(p_b1%M(1,col1)%diam) > slope_max ) THEN
      ! Mark off the first boxes on the columns from the column target_i to
      ! the one with 'FMIN'.
      target_set%flags(target_i) = IBCLR(target_set%flags(target_i),&
                                   CONVEX_BIT)
      CALL markoff(target_i, i_fmin, target_set, p_fmin)
      ! Mark off the first box on the colume with 'FMIN'.
      p_fmin%flags(i_fmin) = IBCLR(p_fmin%flags(i_fmin), CONVEX_BIT)
      EXIT
    END IF
  END IF
  ! To start the next pass, the first fixed column jumps to the target column
  ! just found which is the next column on convex hull.
  i = target_i
  p_setInd1 => target_set
END DO
RETURN
END SUBROUTINE findconvex
FUNCTION findpt(b, col, i_last, index, p_last) RESULT(p_index)
IMPLICIT NONE
! Find the pointer for the box 'index' in the column 'col' of box matrix
! 'b'. If this box 'index' is in one of the box links, record the pointer
! to the box link holding this box 'index' in 'p_last' and compute the
! box position offset 'i_last'. These two records will be used to resume
! chasing the pointers for the heap elements closer to the bottom.
!
! On input:
! b      - Box matrix holding the box column 'col' with the box 'index'.
! col    - Column index.
! i_last - Box position offset used in finding the starting box position
!          from the box link 'p_last'.
! index  - Box index.
! p_last - Pointer to the last box link that has been chased so far.
!
! On output:
! i_last - Updated 'i_last'.
! p_last - Updated 'p_last'.
!
TYPE(BoxMatrix), INTENT(IN), TARGET :: b
INTEGER, INTENT(IN) :: col
INTEGER, INTENT(INOUT) :: i_last
INTEGER, INTENT(IN) :: index
TYPE(BoxLink), POINTER :: p_last
TYPE(HyperBox), POINTER :: p_index
! Local variables.
TYPE(BoxLink), POINTER :: p_l   ! Pointer to a box link.
```

```
INTEGER :: i
IF (.NOT.ASSOCIATED(p_last)) THEN
  ! 'p_last' has not been set, so start from the first box matrix 'b'.
  IF (index <= row_w) THEN
    ! The box 'index' is in 'M' array of 'b'.
    p_index => b%M(index,col)
  ELSE
    ! Chase to the box link that this box belongs to.
    p_l => b%sibling(col)%p
    DO i = 1, (index-1)/row_w -1
      p_l => p_l%next
    END DO
    ! Found the box link that holds the box 'index'.
    p_index => p_l%Line(MOD(index-1, row_w)+1)
    ! Set 'p_last' and 'i_last'.
    p_last => p_l
    i_last = ((index-1)/row_w)*row_w
  END IF
ELSE
  ! Start from 'p_last', because it is the last box link that has been
  ! processed.
  p_l => p_last
  DO i = 1, (index-i_last-1)/row_w
    p_l => p_l%next
  END DO
  ! Found the box link that holds the box 'index'.
  p_index => p_l%Line(MOD(index-1,row_w)+1)
  ! Set 'p_last' and 'i_last'.
  p_last => p_l
  i_last = ((index-1)/row_w)*row_w
END IF
RETURN
END FUNCTION findpt
SUBROUTINE findsetI(b, col, setI)
IMPLICIT NONE
! Fills out 'setI', holding dimensions with the maximum side length
! of the first box on 'col' in box matrix links 'b'.
!
! On input:
! b   - The head link of box matrices.
! col - The global column index of box matrix links.
!
! On output:
! setI - The set of dimensions with the maximum side length.
!
TYPE(BoxMatrix), INTENT(IN), TARGET :: b
INTEGER, INTENT(IN) :: col
TYPE(int_vector), INTENT(INOUT) :: setI
! Local variables.
INTEGER :: b_id, i, j
REAL(KIND = R8)  :: temp
TYPE(BoxMatrix), POINTER :: p_b
! Find the box matrix link that 'col' is associated with.
IF (col <= col_w) THEN
  p_b => b
  j = col
```

```
ELSE
  b_id = (col-1)/col_w + 1
  j = MOD(col-1, col_w) + 1
  p_b => b
  DO i = 1, b_id-1
    p_b => p_b%child
  END DO
END IF
! Search for the maximum side length.
temp = MAXVAL(p_b%M(1,j)%side(:))
! Find all the dimensions with the maximum side length.
DO i = 1, N
  IF ((ABS(p_b%M(1,j)%side(i) - temp)/temp) <= EPS4N) THEN
    ! Add it to 'setI'.
    CALL insNode(N, i, setI%dim+1, setI)
  END IF
END DO
RETURN
END SUBROUTINE findsetI
SUBROUTINE heapify(b, col)
IMPLICIT NONE
! Heapify the column 'col' (local index) of box matrix 'b'.
!
! On input:
! b   - Box matrix holding the box column 'col' to be heapified.
! col - Column index.
! On output:
! b   - Box matrix with the heapified box column 'col'.
TYPE(BoxMatrix), INTENT(INOUT), TARGET :: b
INTEGER, INTENT(IN) :: col
! Local variables.
INTEGER :: i
! Heapify the box column starting from the last non-leaf node from bottom up.
DO i = b%ind(col)/2, 1, -1
  CALL siftdown(b, col, i)
END DO
RETURN
END SUBROUTINE heapify
SUBROUTINE init(b, status)
IMPLICIT NONE
! Allocates the arrays and initializes the first center point.
! Evaluates the function value at the center point and initializes
! 'FMIN' and 'unit_x'.
!
! On output
! b      - The first box matrix to initialize.
! status - Status of return.
!          0   Successful.
!          1   Allocation error.
!
TYPE(BoxMatrix), INTENT(OUT), TARGET :: b
INTEGER, INTENT(OUT):: status
! Local variables.
INTEGER :: alloc_err, i, iflag, j
! Normal status.
status = 0
```

```
iflag = 0
! Allocate arrays.
ALLOCATE(b%M(row_w, col_w), STAT = alloc_err)
IF (alloc_err /= 0) THEN ; status = 1; RETURN; END IF
ALLOCATE(b%ind(col_w), STAT = alloc_err)
IF (alloc_err /= 0) THEN ; status = 1; RETURN; END IF
! Clear the box counter for each column.
b%ind(:) = 0
! Nullify the child link to the next box matrix.
NULLIFY(b%child)
ALLOCATE(b%sibling(col_w), STAT = alloc_err)
IF (alloc_err /= 0) THEN ; status = 1; RETURN; END IF
DO i = 1, col_w
  NULLIFY(b%sibling(i)%p)
END DO
DO i = 1, row_w
  DO j = 1, col_w
    ALLOCATE(b%M(i,j)%c(N), STAT = alloc_err)
    IF (alloc_err /= 0) THEN ; status = 1; RETURN; END IF
    ALLOCATE(b%M(i,j)%side(N), STAT = alloc_err)
    IF (alloc_err /= 0) THEN ; status = 1; RETURN; END IF
  END DO
END DO
! Initialize the center of the first unit hypercube in box matrix 'b'
! and 'unit_x' in the normalized coordinate system.
b%M(1,1)%c(:) = 0.5_R8
b%M(1,1)%side(:) = 1.0_R8
unit_x(:) = 0.5_R8
! Evaluate objective function at 'c'.
! Store the function value and initialize 'FMIN'.
iflag = 0
FMIN = OBJ_FUNC(L + b%M(1,1)%c(:) * (U - L), iflag)
! Check the iflag to deal with undefined function values.
IF (iflag /= 0) THEN
! Add a handler in future.
END IF
b%M(1,1)%val = FMIN
! Initialize the diameter squared for this box and 'dia',
! the diameter squared associated with 'FMIN'.
dia = DOT_PRODUCT(b%M(1,1)%side,b%M(1,1)%side)
b%M(1,1)%diam = dia
! Initialize the 'ind' for the first column and 'id' for this box matrix.
b%ind(1) = 1
b%id = 1
RETURN
END SUBROUTINE init
SUBROUTINE initLink(newBoxLink, iflag)
IMPLICIT NONE
! Initializes a new box link.
!
! On input:
! newBoxLink - A new box link.
!
! On output:
! newBoxLink - 'newBoxLink' with initialized structures.
! iflag      - Return status.
!             0   Normal.
!             1   Allocation failure.
!
TYPE(BoxLink), INTENT(INOUT) :: newBoxLink
INTEGER, INTENT(OUT) :: iflag
! Local variables.
INTEGER :: alloc_err, i
! Initialize 'iflag'.
iflag = 0
! Allocate 'Line' of the new BoxLink.
ALLOCATE(newBoxLink%Line(row_w),STAT = alloc_err)
IF (alloc_err /= 0) THEN; iflag = 1; RETURN; END IF
DO i = 1, row_w
  ALLOCATE(newBoxLink%Line(i)%c(N),STAT = alloc_err)
  IF (alloc_err /= 0) THEN; iflag = 1; RETURN; END IF
  ALLOCATE(newBoxLink%Line(i)%side(N),STAT = alloc_err)
  IF (alloc_err /= 0) THEN; iflag = 1; RETURN; END IF
END DO
! Nullify pointers 'next' and 'prev'.
NULLIFY(newBoxLink%next)
NULLIFY(newBoxLink%prev)
! Initialize the counter for boxes.
newBoxLink%ind = 0
RETURN
END SUBROUTINE initLink
SUBROUTINE  insBox(box, col, b, iflag)
IMPLICIT NONE
! Inserts 'box' in column 'col' of box matrices 'b'. If all positions
! in 'col' are full, makes a new box link linked to the end of
! this column.
!
! On input:
! box - The box to be inserted.
! col - The global column index at which to insert 'box'.
! b   - The head link of box matrices.
!
! On output:
! b     - 'b' has a newly added 'box'.
! iflag - Return status.
!         0   Normal.
!         1   Allocation failure.
!
TYPE(HyperBox), INTENT(IN) :: box
INTEGER, INTENT(IN) :: col
TYPE(BoxMatrix), INTENT(INOUT), TARGET :: b
INTEGER, INTENT(OUT) :: iflag
! Local variables.
INTEGER :: b_id, i, mycol, status
TYPE(BoxLink), POINTER :: p_blink
TYPE(BoxMatrix), POINTER :: p_b
! Initialize 'iflag' as a normal return.
iflag = 0
! Locate the box matrix in which to insert 'box'.
mycol = col
IF (mycol <= col_w) THEN
  p_b => b
```

```
ELSE
  b_id = (mycol-1)/col_w + 1
  mycol = MOD(mycol-1, col_w) + 1
  p_b => b
  DO i=1, b_id-1
    p_b => p_b%child
  END DO
END IF
! Insert the box at the end of column 'mycol' of box matrix 'p_b'.
IF (p_b%ind(mycol) < row_w) THEN
  ! There is no box links.
  p_b%M(p_b%ind(mycol)+1, mycol) = box
ELSE
  ! There are box links. Chase to the last box link.
  p_blink => p_b%sibling(mycol)%p
  DO i = 1, p_b%ind(mycol)/row_w - 1
    p_blink => p_blink%next
  END DO
  p_blink%ind = p_blink%ind + 1
  p_blink%Line(p_blink%ind) = box
END IF
! Update 'ind' of the column ('ind' of 'p_b' counts all the boxes in
! this column including the ones in its box links.).
p_b%ind(mycol) = p_b%ind(mycol) + 1
! Add a new box link if needed.
CALL checkblinks(mycol, p_b, status)
IF (status /=0) THEN ; iflag = 1 ; END IF
RETURN
END SUBROUTINE insBox
SUBROUTINE insMat(box, b, setDia, setInd, setFcol, status)
IMPLICIT NONE
! Retrieves all box matrices to find the place at which to insert 'box'.
! Inserts it in the column with the same squared diameter, or a new
! column if the squared diameter is new. In the same column, the smaller
! 'val', the earlier the position.
!
! On input:
! box     - The box to be inserted.
! b       - The head link of box matrices.
! setDia  - A linked list holding different squared diameters.
! setInd  - A linked list holding the column indices corresponding to
!           'setDia'.
! setFcol - A linked list holding free columns of box matrices.
!
! On output:
! b       - 'b' has the newly added 'box' and updated counters.
! setDia  - 'setDia' has a newly added squared diameter if any and
!          updated 'dim' if modified.
! setInd  - 'setInd' has a newly added column index if needed and
!          updated 'dim' if modified.
! setFcol - 'setFcol' has a column index removed if needed and updated
!          'dim' if modified.
! status  - Status of processing.
!          0   Normal.
!          1   Allocation failures of type 'BoxMatrix'.
!          2   Allocation failures of type 'BoxLink'.
!         3   Allocation failures of type 'real_vector' or 'int_vector'.
!
TYPE(HyperBox), INTENT(IN) :: box
TYPE(BoxMatrix), INTENT(INOUT), TARGET :: b
TYPE(real_vector), INTENT(INOUT), TARGET :: setDia
TYPE(int_vector), INTENT(INOUT), TARGET :: setInd
TYPE(int_vector), INTENT(INOUT), TARGET :: setFcol
INTEGER, INTENT(OUT) :: status
! Local variables.
INTEGER :: b_id, b_pos, i, iflag, j, pos
LOGICAL :: found
TYPE(BoxMatrix), POINTER :: p_b
TYPE(int_vector), POINTER :: p_set, p_setFcol
TYPE(real_vector), POINTER :: p_rset
! Initialization for msearchSet().
pos = 0
found = .FALSE.
NULLIFY(p_rset)
iflag = 0
status = 0
! Locate a node of 'setDia' into which 'diam' can be inserted.
p_rset => msearchSet(setDia, box%diam)
CALL binaryS(p_rset, box%diam, pos, found)
IF (found) THEN
  ! A match is found in 'p_rset' of 'setDia'.
  ! Find the corresponding node in 'setInd' to match 'p_rset'.
  p_set => setInd
  DO i = 1, p_rset%id-1
    p_set => p_set%next
  END DO
  ! Insert 'box' to the column indexed by 'pos' in a node of 'setInd'.
  CALL insBox(box, p_set%elements(pos), b, iflag)
 ! Mark the column indexed by 'pos' in a node of 'setInd' to be updated.
  p_set%flags(pos) = IBSET(p_set%flags(pos), UPDATE_BIT)
  IF (iflag /= 0) THEN ; status = 2 ; RETURN ; END IF
ELSE
  ! No match is found. It's a new squared diameter.
  IF (pos == 0)THEN
    ! Obtain a free column from 'setFcol' to insert 'box' after all
    ! other columns.
    IF (setFcol%dim > 0)THEN
      ! 'setFcol' is not empty, so pop a column from the top of 'setFcol'
      ! nodes.
      IF (setFcol%dim < col_w) THEN
        ! The head node is not full, therefore it must be the top node.
        i = setFcol%elements(setFcol%dim)
        ! Update 'dim'.
        setFcol%dim = setFcol%dim - 1
      ELSE
        ! There might be other nodes with element(s).
        p_setFcol => setFcol%next
        IF (ASSOCIATED(p_setFcol)) THEN
          ! Chase to the top node of 'setFcol' with element(s).
          DO WHILE(p_setFcol%dim == col_w)
            p_setFcol => p_setFcol%next
          END DO
          ! The top node could be 'p_setFcol' or its 'prev'.
          IF (p_setFcol%dim /= 0) THEN
            i = p_setFcol%elements(p_setFcol%dim)
            p_setFcol%dim = p_setFcol%dim - 1
          ELSE
            i = p_setFcol%prev%elements(p_setFcol%prev%dim)
            p_setFcol%prev%dim = p_setFcol%prev%dim - 1
          END IF
        ELSE
          ! There are no more nodes with elements. Pop a column from the
          ! head node of 'setFcol'.
          i = setFcol%elements(setFcol%dim)
          ! Update 'dim'.
          setFcol%dim = setFcol%dim -1
        END IF
      END IF
    ELSE
      ! There are no free columns, so make a new box matrix.
      CALL newMat(b, setDia, setInd, setFcol, iflag)
      IF (iflag /= 0) THEN ; status = iflag ; RETURN ; END IF
      ! Pop a column from the top of 'setFcol' for use.
      i = setFcol%elements(setFcol%dim)
      CALL rmNode(col_w, 0, setFcol%dim, setFcol)
    END IF
    ! Found the global column index 'i' at which to insert 'box'. Convert
    ! it to a local column index 'b_pos' and locate the box
    ! matrix 'p_b' at which to insert 'box'.
    IF (i <= col_w) THEN
      p_b => b
      b_pos = i
    ELSE
      b_id = (i-1)/col_w + 1
      b_pos = MOD(i-1, col_w) + 1
      p_b => b
      DO j = 1, b_id -1
        p_b => p_b%child
      END DO
    END IF
    ! Insert 'box' to the beginning of the new column 'b_pos'.
    p_b%M(1,b_pos) = box
    ! Locate the nodes in both 'setDia' and 'setInd' at which to insert
    ! the new squared diameter and the column index 'i' at the end of
    ! both linked lists ('pos' is 0).
    IF (setDia%dim < col_w)THEN
      ! There are no more nodes with elements. Assign the head node
      ! to 'p_rset'.
      p_rset => setDia
    ELSE
      ! There are other nodes to check.
      p_rset => setDia%next
      IF (ASSOCIATED(p_rset)) THEN
        ! Chase to the end of the linked list.
        DO WHILE(p_rset%dim == col_w)
          p_rset => p_rset%next
        END DO
      ELSE
```

```
      ! There are no more nodes. Assign the head node to 'p_rset'
      p_rset => setDia
    END IF
  END IF
  ! Found the node 'p_rset' of 'setDia' to insert.
  CALL insNode(col_w, box%diam, p_rset%dim+1, p_rset)
  ! Find the corresponding node in 'setInd' at which to insert the
  ! column index 'i'.
  p_set => setInd
  DO j =1, p_rset%id -1
    p_set => p_set%next
  END DO
  CALL insNode(col_w, i, p_set%dim+1, p_set)
  ! Update 'ind' of col 'b_pos' in 'p_b'.
  p_b%ind(b_pos) = 1
ELSE
  ! 'pos' is not 0. 'p_rset' points to the right node of 'setDia' to
  ! insert the new squared diameter.
  ! Obtain a free column from 'setFcol' to insert a new column before the
  ! column indexed by the returned 'pos'.
  IF (setFcol%dim > 0)THEN
    ! 'setFcol' is not empty, so pop a column from the top of 'setFcol'
    ! nodes.
    IF (setFcol%dim < col_w) THEN
    ! The head node is not full, so it must be the top.
      i = setFcol%elements(setFcol%dim)
      setFcol%dim = setFcol%dim - 1
    ELSE
      ! There might be nodes with free columns.
      p_setFcol => setFcol%next
      IF (ASSOCIATED(p_setFcol)) THEN
        ! Chase to the top of 'setFcol' links.
        DO WHILE(p_setFcol%dim == col_w)
          p_setFcol => p_setFcol%next
        END DO
        ! The top node could be 'p_setFcol' or its 'prev'.
        IF (p_setFcol%dim /= 0) THEN
          i = p_setFcol%elements(p_setFcol%dim)
          p_setFcol%dim = p_setFcol%dim - 1
        ELSE
          i = p_setFcol%prev%elements(p_setFcol%prev%dim)
          p_setFcol%prev%dim = p_setFcol%prev%dim - 1
        END IF
      ELSE
        ! There are no more nodes with elements. Pop a column from the
        ! head node of 'setFcol'.
        i = setFcol%elements(setFcol%dim)
        setFcol%dim = setFcol%dim -1
      END IF
    END IF
  ELSE
    ! There are no free columns, so make a new box matrix.
    CALL newMat(b, setDia, setInd, setFcol, iflag)
    IF (iflag /= 0) THEN ; status = iflag ; RETURN ; END IF
    ! Pop a column for use.
    i = setFcol%elements(setFcol%dim)
```

```
      CALL rmNode(col_w, 0, setFcol%dim, setFcol)
    END IF
    ! Found the global column index 'i' at which to insert 'box'.
    ! Convert it to a local column index 'b_pos' and locate the
    ! box matrix 'p_b' in which to insert 'box'.
    IF (i <= col_w) THEN
      p_b => b
      b_pos = i
    ELSE
      b_id = (i-1)/col_w + 1
      b_pos = MOD(i-1, col_w) + 1
      p_b => b
      DO j = 1, b_id -1
        p_b => p_b%child
      END DO
    END IF
    ! Add 'box' to be the first on column 'b_pos' of 'p_b'.
    p_b%M(1,b_pos) = box
    ! Insert the new squared diameter at the position 'pos' in 'p_rset'
    ! of 'setDia'.
    CALL insNode(col_w, box%diam, pos, p_rset)
    ! Insert the corresponding column index 'i' at the same position 'pos'
    ! in a node of 'setInd'.
    p_set => setInd
    DO j = 1, p_rset%id -1
      p_set => p_set%next
    END DO
    CALL insNode(col_w, i, pos, p_set)
    ! Update 'ind' of box matrix 'p_b'.
    p_b%ind(b_pos) = 1
  END IF
END IF
RETURN
END SUBROUTINE insMat
SUBROUTINE markoff(i, target_i, p_setInd1, target_set)
IMPLICIT NONE
! Marks off columns in between the column 'i' of 'p_setInd1' and column
! 'target_i' of 'target_set' by clearing the CONVEX_BIT in 'flags'.
!
! On input:
! i          - Column index of the first box for computing slope in findconvex.
! target_i   - Column index of the second box for computing slope in
!              findconvex.
! p_setInd1  - The node of 'setInd' holding the column 'i'.
! target_set - The node of 'setInd' holding the column 'target_i'.
!
! On output:
! p_setInd1  - 'p_setInd1' has changed column indices.
! target_set - 'target_set' has changed column indices.
!
INTEGER, INTENT(IN) :: i
INTEGER, INTENT(IN) :: target_i
TYPE(int_vector), INTENT(INOUT), TARGET :: p_setInd1
TYPE(int_vector), POINTER :: target_set
! Local variables.
INTEGER :: j
```

```
TYPE(int_vector), POINTER :: p_set
! Check if any columns in between.
IF (ASSOCIATED(target_set, p_setInd1)) THEN
  ! If 'target_i' is next to column 'i', return.
IF (i == target_i) RETURN
  ! If no any columns in between, return.
  IF (i+1 == target_i) RETURN
END IF
! Clear all CONVEX_BITs in 'flags' in between.
j = i
p_set => p_setInd1
DO
  j = j + 1
  IF (j > p_set%dim) THEN
    p_set => p_set%next
    IF (.NOT.ASSOCIATED(p_set).OR.(p_set%dim==0)) EXIT
    j = 1
  END IF
  ! Check if at the target node.
  IF (ASSOCIATED(target_set, p_set)) THEN
    ! If 'j' has reached 'target_i', exit.
    IF (j == target_i) EXIT
  END IF
  ! Clear the CONVEX_BIT of 'flags' for column 'j' of 'p_set'.
  p_set%flags(j) = IBCLR(p_set%flags(j), CONVEX_BIT)
END DO
END SUBROUTINE markoff
FUNCTION msearchSet(setDia, diam) RESULT(p_rset)
IMPLICIT NONE
! Finds the right node in 'setDia' in which to insert 'diam'.
!
! On input:
! setDia - A linked list holding different squared diameters.
! diam   - A diameter squared to be inserted in a node in 'setDia'.
!
! On output:
! p_rest - Pointer to the right node of 'setDia'.
!
TYPE(real_vector), INTENT(IN), TARGET :: setDia
REAL(KIND = R8), INTENT(IN) :: diam
TYPE(real_vector), POINTER :: p_rset
! Local variables.
TYPE(real_vector), POINTER :: p_setDia
! Initialize 'p_setDia'.
p_setDia => setDia
DO
  IF (p_setDia%dim > 0) THEN
    ! There are elements to be compared with 'diam'.
    IF (diam >= p_setDia%elements(1)) THEN
      ! 'diam' is the biggest. Return this node as 'p_rset'.
      p_rset => p_setDia
      EXIT
    ELSE
      IF ((diam >= p_setDia%elements(p_setDia%dim)) .OR.    &
         ((ABS(diam - p_setDia%elements(p_setDia%dim))       &
         /MAX(diam, p_setDia%elements(p_setDia%dim))) <= EPS4N) ) THEN
```

```
        ! 'diam' is within the range of elements in this node.
        p_rset => p_setDia
        EXIT
      ELSE
        ! 'diam' is smaller than the last element. Go on to the next
        ! node, if any.
        IF (ASSOCIATED(p_setDia%next)) THEN
          p_setDia => p_setDia%next
        ELSE
          ! There are no more nodes. Return this pointer.
          p_rset => p_setDia
          EXIT
        END IF
      END IF
    END IF
  ELSE
    ! It's empty. Return this pointer.
    p_rset => p_setDia
    EXIT
  END IF
END DO
RETURN
END FUNCTION msearchSet
SUBROUTINE newMat(b, setDia, setInd, setFcol, iflag)
IMPLICIT NONE
!  Makes a new box matrix and its associated linked lists for holding
!  different squared diameters, column indices, and free columns.
!  Links them to existing structures.
!
! On input:
! b      - The head link of box matrices.
! setDia  - Linked list holding different squared diameters of
!          current boxes.
! setInd  - Linked list holding column indices of box matrices.
! setFcol - Linked list holding free columns of box matrices.
!
! On output:
! b      - 'b' has a box matrix link added at the end.
! setDia  - 'setDia' has a node added at the end.
! setInd  - 'setInd' has a node added at the end.
! setFcol - 'setFcol' has a node added at the end.
! iflag   - Return status.
!          0   Normal.
!          1   Allocation failures of type 'BoxMatrix'.
!          3   Allocation failures of type 'real_vector' or 'int_vector'.
!
TYPE(BoxMatrix), INTENT(INOUT), TARGET :: b
TYPE(real_vector), INTENT(INOUT), TARGET :: setDia
TYPE(int_vector), INTENT(INOUT), TARGET :: setInd
TYPE(int_vector), INTENT(INOUT), TARGET :: setFcol
INTEGER, INTENT(OUT) :: iflag
! Local variables.
INTEGER :: alloc_err, i, j
TYPE(BoxMatrix), POINTER :: new_b, p_b
TYPE(int_vector), POINTER :: n_setFcol, n_setInd, p_setFcol, p_setInd
TYPE(real_vector), POINTER :: n_setDia, p_setDia
```

```
! Initialize iflag for normal return.
iflag = 0
! Allocate a new box matrix.
ALLOCATE(new_b, STAT=alloc_err)
IF (alloc_err /= 0) THEN; iflag = 1; RETURN; END IF
! Allocate its associated arrays.
ALLOCATE(new_b%M(row_w, col_w), STAT=alloc_err)
IF (alloc_err /= 0) THEN; iflag = 1; RETURN; END IF
ALLOCATE(new_b%ind(col_w), STAT=alloc_err)
IF (alloc_err /= 0) THEN; iflag = 1; RETURN; END IF
! Clear the box counter for each column.
new_b%ind(:) = 0
! Nullify pointers for a child link and box links.
NULLIFY(new_b%child)
ALLOCATE(new_b%sibling(col_w), STAT=alloc_err)
IF (alloc_err /= 0) THEN; iflag = 1; RETURN; END IF
DO i = 1, col_w
  NULLIFY(new_b%sibling(i)%p)
END DO
DO i = 1, row_w
  DO j = 1, col_w
    ALLOCATE(new_b%M(i,j)%c(N), STAT=alloc_err)
    IF (alloc_err /= 0) THEN; iflag = 1; RETURN; END IF
    ALLOCATE(new_b%M(i,j)%side(N), STAT=alloc_err)
    IF (alloc_err /= 0) THEN; iflag = 1; RETURN; END IF
  END DO
END DO
! Find the last box matrix to link with the new one.
p_b => b
DO WHILE(ASSOCIATED(p_b%child))
  p_b => p_b%child
END DO
! Found the last box matrix p_b. Link it to new box matrix 'b'.
p_b%child => new_b
! Set up 'id' for new_b.
new_b%id = p_b%id + 1
! Allocate new 'setDia', 'setInd' and 'setFcol' for 'new_b'.
! Find the corresponding nodes of 'setDia', 'setInd' and 'setFcol'.
p_setDia => setDia
p_setInd => setInd
p_setFcol => setFcol
DO i=1, p_b%id-1
  p_setDia => p_setDia%next
  p_setInd => p_setInd%next
  p_setFcol => p_setFcol%next
END DO
! Allocate a new node for 'setDia'. Initialize its structure.
ALLOCATE(n_setDia, STAT=alloc_err)
IF (alloc_err /= 0) THEN; iflag = 3; RETURN; END IF
ALLOCATE(n_setDia%elements(col_w), STAT=alloc_err)
IF (alloc_err /= 0) THEN; iflag = 3; RETURN; END IF
NULLIFY(n_setDia%next)
NULLIFY(n_setDia%prev)
n_setDia%id = p_setDia%id +1
n_setDia%dim = 0
! Allocate a new node for 'setInd'. Initialize its structure.
```

```
ALLOCATE(n_setInd, STAT=alloc_err)
IF (alloc_err /= 0) THEN; iflag = 3; RETURN; END IF
ALLOCATE(n_setInd%elements(col_w), STAT=alloc_err)
IF (alloc_err /= 0) THEN; iflag = 3; RETURN; END IF
ALLOCATE(n_setInd%flags(col_w), STAT=alloc_err)
IF (alloc_err /= 0) THEN; iflag = 3; RETURN; END IF
n_setInd%flags(:)= 0
NULLIFY(n_setInd%next)
NULLIFY(n_setInd%prev)
n_setInd%id = p_setInd%id + 1
n_setInd%dim = 0
! Allocate a new node for 'setFcol'. Initialize its structure.
ALLOCATE(n_setFcol, STAT=alloc_err)
IF (alloc_err /= 0) THEN; iflag = 3; RETURN; END IF
ALLOCATE(n_setFcol%elements(col_w), STAT=alloc_err)
IF (alloc_err /= 0) THEN; iflag = 3; RETURN; END IF
ALLOCATE(n_setFcol%flags(col_w), STAT=alloc_err)
IF (alloc_err /= 0) THEN; iflag = 3; RETURN; END IF
NULLIFY(n_setFcol%next)
NULLIFY(n_setFcol%prev)
n_setFcol%id = p_setFcol%id + 1
n_setFcol%dim = 0
! Link them to the end of existing sets.
p_setDia%next => n_setDia
n_setDia%prev => p_setDia
p_setInd%next => n_setInd
n_setInd%prev => p_setInd
p_setFcol%next => n_setFcol
n_setFcol%prev => p_setFcol
! Fill up 'setFcol' with new columns from the new box matrix.
! Starting from the last column, push free columns to the top of 'setFcol'.
DO i = 1, col_w
  setFcol%elements(i) = new_b%id*col_w - (i-1)
END DO
setFcol%dim = col_w
RETURN
END SUBROUTINE newMat
SUBROUTINE sampleF(setB, eval_c)
IMPLICIT NONE
! Evaluates the objective function at each newly sampled center point.
! Keeps updating 'FMIN' and 'unit_x'.
!
! On input:
! setB   - The set of newly sampled boxes with their center points'
!          coordinates.
! eval_c - The counter of evaluations.
!
! On output:
! setB   - The set of newly sampled boxes with added function values
!          at center points.
! eval_c - The updated counter of evaluations.
!
TYPE(BoxLine), INTENT(INOUT) :: setB
INTEGER, INTENT(INOUT) :: eval_c
! Local variables.
INTEGER :: i, iflag, j
```

```
! Loop evaluating all the new center points of boxes in 'setB'.
DO i = 1, setB%ind
  ! Evaluate the function in the original frame.
  iflag = 0
  setB%Line(i)%val = OBJ_FUNC(L + setB%Line(i)%c(:) * (U - L), iflag)
  ! Check 'iflag'.
  IF (iflag /= 0) THEN
    ! Add a handle to deal with undefined function value.
  END IF
  ! Update evaluation counter.
  eval_c = eval_c + 1
  IF (FMIN > setB%Line(i)%val) THEN
    ! Update 'FMIN' and 'unit_x'.
    FMIN = setB%Line(i)%val
    unit_x(:) = setB%Line(i)%c(:)
  END IF
END DO
RETURN
END SUBROUTINE sampleF
SUBROUTINE sampleP(col, setI, b, setB)
IMPLICIT NONE
! On each dimension in 'setI', samples two center points at the c+delta*ei
! and c-delta*ei. Since it has been normalized, ei equals 1. In 'setB',
! records all the new points as the centers of boxes which will be formed
! completely through subroutines sampleF and divide.
!
! On input:
! col  - The global column index of the box to subdivide.
! setI - The set of dimensions with maximum side length.
! b    - The head link of box matrices.
! setB - The empty set of type 'HyperBox' which will hold newly sampled
!       points as the centers of new boxes.
!
! On output:
! setI - The set of dimensions with maximum side length.
! b    - The head link of box matrices.
! setB - The set of boxes which contains the newly sampled center points.
!
INTEGER, INTENT(IN) :: col
TYPE(int_vector), INTENT(INOUT) :: setI
TYPE(BoxMatrix), INTENT(INOUT), TARGET :: b
TYPE(BoxLine), INTENT(INOUT) :: setB
! Local variables.
INTEGER :: b_id  ! Identifier of the associated box matrix.
INTEGER :: i, k  ! Loop counters.
INTEGER :: j     ! Local column index converted from the global one 'col'.
INTEGER :: new_i ! Index of new points in setB.
REAL (KIND=R8) :: delta ! 1/3 of the maximum side length.
TYPE (BoxMatrix), POINTER :: p_b ! Pointer to the associated box matrix.
! Find the box matrix that 'col' is associated with. Store the pointer
! to box matrix in 'p_b'. The local column index 'j' will be converted from
! 'col'.
IF (col <= col_w) THEN
  p_b => b
  j = col
ELSE
```

```
  b_id = (col-1)/col_w + 1
  j = MOD(col-1, col_w) + 1
  p_b => b
  DO i = 1, b_id-1
    p_b => p_b%child
  END DO
END IF
! Find the maximum side length by obtaining the dimension in 'setI'.
! Then, extract the maximum side length from the first box on column 'j' of
! box matrix 'p_b'. Calculate 'delta', 1/3 of the maximum side length.
delta = p_b%M(1,j)%side(setI%elements(setI%dim))/ 3
! Loop sampling two new points of all dimensions in 'setI'.
! c + delta*ei => newpt_1; c - delta*ei => newpt_2, where ei=1
! for it's a normalized space.
DO i = 1, setI%dim
  new_i = setB%ind + 1
  ! Copy the coordinates of parent box to the two new boxes
  setB%Line(new_i)%c(:) = p_b%M(1, j)%c(:)
  setB%Line(new_i + 1)%c(:) = p_b%M(1, j)%c(:)
  ! Assign changed coordinates to the two new points in 'setB'.
  setB%Line(new_i)%c(setI%elements(i)) = &
  p_b%M(1, j)%c(setI%elements(i)) + delta
  setB%Line(new_i+1)%c(setI%elements(i)) = &
  p_b%M(1, j)%c(setI%elements(i)) - delta
  ! Record the directions with changes in 'setB%dir' for further
  ! processing to find the dividing order of dimensions.
  setB%dir(new_i) = setI%elements(i)
  setB%dir(new_i + 1) = setI%elements(i)
  ! Update 'ind' of 'setB'.
  setB%ind = setB%ind + 2
  ! Initialize side lengths of new points for further dividing
  ! by copying the sides from the parent box.
  setB%Line(new_i)%side(:) = p_b%M(1,j)%side(:)
  setB%Line(new_i+1)%side(:) = p_b%M(1,j)%side(:)
END DO
! Clear 'setI'.
setI%dim = 0
RETURN
END SUBROUTINE sampleP
SUBROUTINE siftdown(b, col, index)
IMPLICIT NONE
! Siftdown heap element 'index' through the heap column 'col' in
! the box matrix 'b'.
!
! On input:
! b     - Box matrix holding the box column 'col' for siftdown.
! col   - Column index.
! index - Index of the box to be sifted down.
!
! On output:
! b     - Box matrix with the rearranged elements by siftdown.
TYPE(BoxMatrix), INTENT(INOUT), TARGET :: b
INTEGER, INTENT(IN) :: col, index
! Local variables.
INTEGER :: i, j       ! Loop counters.
INTEGER :: i_last      ! Index of the last box that has been processed
```

```
                    ! in the previous iteration.
INTEGER :: i_last_backup  ! i_last's backup used to go back to i_last,
                       ! which may be updated in the current iteration.
INTEGER :: left, right    ! Indices for the left and right children.
TYPE(BoxLink), POINTER :: p_last       ! Pointer to the last box link
                                   ! that has been processed.
TYPE(BoxLink), POINTER :: p_last_backup ! Pointer backup for 'p_last'.
TYPE(HyperBox), POINTER :: p_i         ! Pointer to the heap parent box.
TYPE(HyperBox), POINTER :: p_left, p_right  ! Pointers to the left and right
                                      ! child boxes.
NULLIFY(p_last)
! Starting siftdown operation from the box 'index'.
i = index
DO
  ! Find the indices for the left and right children.
  left = 2*i
  right = 2*i + 1
  ! If 'i' is a leaf, exit.
  IF (left > b%ind(col)) EXIT
  ! Find the pointers to the the ith box and its left child. Record the
  ! pointer 'p_last' and index 'i_last' for the currently processed box
  ! link.
  p_i => findpt(b, col, i_last, i, p_last)
  p_left => findpt(b, col, i_last, left, p_last)
  IF (left < b%ind(col)) THEN
    ! Backup the pointer 'p_last' and the index 'i_last', because they will
    ! be updated when finding the pointer to the right child. If the right
    ! child is in the correct place in the heap, restore the pointer
    ! 'p_last' and the index 'i_last'.
    p_last_backup => p_last
    i_last_backup = i_last
    p_right => findpt(b, col, i_last, right, p_last)
    IF (p_left%val > p_right%val) THEN
      p_left => p_right
      left = left + 1
    ELSE
      ! Restore 'p_last' and 'i_last' to the values before finding
      ! the pointer for the right child.
      p_last => p_last_backup
      i_last = i_last_backup
    END IF
  END IF
  ! If the boxes are in the correct order, exit.
  IF (p_i%val <= p_left%val) EXIT
  ! Swap  the boxs pointed by 'pi' and 'p_left'.
  tempbox = p_i
  p_i = p_left
  p_left = tempbox
  ! Continue siftdown operation from the left child of 'i'.
  i = left
END DO
RETURN
END SUBROUTINE siftdown
END SUBROUTINE VTDIRect
END MODULE VTDIRect_MOD
```

VITA

Jian He was born in Sichuan, P. R. China, on September 19th, 1975. She earned a bachelor's degree in Computer Engineering in July 1996 from Nanjing University of Science and Technology, Nanjing, China. After that, she worked for three years at METSTAR Meteorological Radar System Company, a joint venture set up by Lockheed Martin Corporation and China Meteorological Administration in Beijing, China. In August 2000, she started graduate study in computer science at Virginia Polytechnic Institute and State University. In October 2002, she received the Master of Science degree in computer science from Virginia Polytechnic Institute and State University. She will continue her study in pursuit of the Doctor of Philosophy degree in computer science.