Algorithm 744: A Stochastic Algorithm for Global Optimization with Constraints

F. MICHAEL RABINOWITZ
Memorial University of Newfoundland

A stochastic algorithm is presented for finding the global optimum of a function of n variables subject to general constraints. The algorithm is intended for moderate values of n, but it can accommodate objective and constraint functions that are discontinuous and can take advantage of parallel processors. The performance of this algorithm is compared to that of the Nelder–Mead Simplex algorithm and a Simulated Annealing algorithm on a variety of nonlinear functions. In addition, one-, two-, four-, and eight-processor versions of the algorithm are compared using 64 of the nonlinear problems with constraints collected by Hock and Schittkowski [1981]. In general, the algorithm is more robust than the Simplex algorithm, but computationally more expensive. The algorithm appears to be as robust as the Simulated Annealing algorithm, but computationally cheaper. Issues discussed include algorithm speed and robustness, applicability to both computer and mathematical models, and parallel efficiency.

Categories and Subject Descriptors: G.1.6 [**Numerical Analysis**]: Optimization—*nonlinear programming*; G.3 [**Mathematics of Computing**]: Probability and Statistics—*probabilistic algorithms (including Monte Carlo)*; G.4 [**Mathematics of Computing**]: Mathematical Software—*certification and testing*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Constrained optimization, global optimization, stochastic optimization, test functions

1. INTRODUCTION

In this article a stochastic algorithm (the Torus algorithm) for finding the global minimum of a function subject to general constraints is described and then compared to the Nelder–Mead Simplex and Simulated Annealing algorithms on the Rosenbrock and parabolic minimization problems. Further evaluation is provided using 64 of the nonlinear programming test examples collected by Hock and Schittkowski [1981]. The global optimization problem of concern is

$$\underset{x}{\text{minimize}}\, f(x) \qquad \text{subject to} \qquad l \le c(x) \le u,$$

Preparation of this article was supported by Grant OGP0002017 from the Natural Sciences and Engineering Research Council of Canada.
Author's address: Department of Psychology, Memorial University of Newfoundland, St. John's, Newfoundland, Canada A1B 3X9.

where x is an n-vector of bounded variables ($x_1, x_2, \ldots, x_n$; $a_i \le x_i \le b_i$), $f(x)$ is a scalar objective function or computer model of k variables ($k \le n$), $c(x)$ is an m-vector of constraint functions of j-variables ($j \le n$), and l and u are m-vectors specifying finite bounds on $c(x)$.

A large number of optimization algorithms have been developed to solve this problem. Some algorithms are only applicable to mathematical functions having first and second derivatives, while others only require that the functions return a value. The latter algorithms can also be used to estimate variables or parameters in computer models. Most algorithms can solve such problems if the function is unimodal. Multimodal functions prove much more difficult. In the multimodal case, many algorithms exit at the first minimum found. For most problems, an exhaustive or grid search of the variable space to locate a global minimum is impractical, as it would involve too large a number of function evaluations. Therefore, some form of limited search, involving random and/or systematic procedures, is used (see Luenberger [1986] and Press et al. [1988] for discussions and descriptions of many optimization procedures).

Kirkpatrick et al. [1983] introduced and operationalized an optimization procedure they referred to as *simulated annealing*. The idea came from statistical mechanics and was based on the observation that solutions when cooled slowly will develop orderly stable molecular structures as solids. The idea can be applied to combinatorial optimization problems in the following way: First, a new point is randomly sampled. If it generates a new minimum value, it is always accepted. If not, it is accepted when a random number between 0 and 1 is less than a probability defined by a mathematical function, usually the Boltzmann equation. Early in the iterative process, the mathematical function generates values near 1, and most points are accepted. By adjusting a parameter in the probability function, usually referred to as temperature, the probability function generates smaller values across successive iterations (cooling), and eventually, only points that produce better solutions are accepted. The procedure generates solutions for combinatorial problems, such as the traveling salesman problem, that are presently mathematically intractable. The success of the method appears to be a consequence of exploring a large number of combinations, and of the small probability of being trapped by local minima until late in the iterative process (see Pannetier [1990] for a recent review).

Corana et al. [1987] extended simulated annealing ideas from combinatorial problems to continuous functions. They demonstrated that their procedure was robust, but very inefficient, in a series of comparisons involving Nelder and Mead's [1965] Simplex algorithm, Masri and Bekey's [1980] Adaptive Random Search algorithm, and their Simulated Annealing algorithm. In general, the Adaptive Random Search algorithm performed poorly in these comparisons and will not be discussed further in this paper.

On their series of tests involving variants of the Rosenbrock function, Corana et al. [1987] found that the Simulated Annealing algorithm solved all 17 problems, while the Nelder–Mead Simplex algorithm solved 15 of the 17 problems. However, the number of function evaluations varied from 161 to

1,869 using the Simplex algorithm, but from 488,001 to 1,328,001 using the Simulated Annealing algorithm. On most Rosenbrock problems, the Simulated Annealing algorithm required from 500 to 1,000 times as many function calls as the Simplex algorithm. Corana et al. constructed a parabolic minimization function for a second series of tests. In general, none of the 26 variants used was solved by the Simplex algorithm, whereas all were either accurately or approximately solved by the Simulated Annealing algorithm, using from 656,000 to 1,665,000 function evaluations. Based on these findings, it appeared that it would be useful to develop a global optimization algorithm that was as robust as but more efficient than the Simulated Annealing algorithm.

In particular, the development of the Torus algorithm was motivated by several ideas: (1) to improve the efficiency of simulated annealing by rejecting points that do not lead to a better solution; (2) to handle both linear and nonlinear constraints; (3) to use only function evaluations, so that the procedure would also be applicable to optimization in computer models and discontinuous mathematical functions; and (4) to implement the procedure in parallel so that moderate-size multidimensional problems could be solved in real time on appropriate computers. Since the algorithm is combinatorial, the practical problem size is about 20 variables with contemporary computer technology.

The Torus algorithm is more flexible than most other optimization procedures. As specified above, the algorithm can be run in parallel, and its success does not depend on the availability of derivatives; nor does it require that the function be continuous. If the algorithm is run several times on the same data set, then multiple solutions, if they exist, will often be revealed. This permits the user to select the solution that is most convenient to implement. Furthermore, the algorithm can handle any mixture of integer and continuous variables. The latter property means that the algorithm can be used in situations in which only integer variables make sense (e.g., a cognitive simulation in which depth of search of a game-board position is a variable).

2. METHOD

The Torus algorithm is designed to solve the global minimization problem defined in the Introduction. Neither $f(x)$, the function to be minimized, nor $c(x)$, the m-vector of constraint functions, needs to be continuous, but both must be computationally bounded. The variables of the n-vector x used to define $f(x)$ and $c(x)$ must range in finite intervals such that $a_1 \leq x_1 \leq b_1, \ldots, a_n \leq x_n \leq b_n$.

In the Torus algorithm, the constraints are handled by penalty functions, with the penalties added to the value returned by $f(x)$. Inequalities are treated as hard constraints by associating very large penalty values with violations. Equalities are treated as soft constraints by associating a monotonic penalty function, which increases with the absolute size of the discrepancy, with each violation.

Table I. Parameters in the Stochastic Minimization Algorithm

Parameter name	Default value	Description
$*$ *bump* $*$	$\frac{1}{2}$	Used to displace a variable from the best-fitting value on the last iteration by a constant proportion of the range.
$*$ *counter*1 $*$	4	The number of parallel processors simulated.
$*$ *counter*2 $*$	40	Maximum number of trial blocks.
$*$ *exit* $*$	10^{-6}	Exit criterion.
$*$ *scalar*1 $*$	1	Weighting factor for the number of single-variable iterations per trial.
$*$ *scalar*2 $*$	1	Weighting factor for the number of multiple-variable iterations per trial.
$*$ *shrink-hit* $*$	$\frac{3}{2}$	Rate the torus collapses after a hit.
$*$ *shrink-trial* $*$	$\frac{3}{2}$	Rate the torus collapses after a trial block.
$*$ *torus* $*$	4000	Used to partially determine the size of the hole in the torus.

Three computer functions constitute the core of the algorithm. In the controlling function, an n-dimensional torus moves in n-space and monotonically shrinks in size over trials isolating the region containing the global minimum. This shrinkage is gradual, mimicking slow cooling in an annealing algorithm. In the multi- and single-dimension functions, which are repeatedly called from the controlling function, new points are randomly sampled around the currently best-fitting point. The permitted range of these sampled points shrinks logarithmically over iterations (mimicking rapid cooling).

The algorithm involves a number of parameters that may be altered to obtain optimal behavior with particular problems. These parameters appear in Table I, along with the default values that are satisfactory for the majority of problems tested. The $*$ *counter*1 $*$ and $*$ *scalar*2 $*$ parameters have the greatest impact on the outcome, and both should be increased for difficult problems.

The descriptions in Table I provide a hint as to how the algorithm is constructed. The controlling function initializes and changes parameters, controls the logic, and calls the other two functions, which do most of the work. The multidimension function randomly generates points in an n-space torus that shrinks logarithmically around the currently best-fitting point across iterations. The single-dimension function works similarly to the multidimension function except that only one variable is randomly altered on each iteration. This function is used in a serial fashion, until each of the n variables is independently adjusted and a new local minimum is isolated. Since these local minima are fairly stable, moves from this point to a new best-fitting point almost always involve changing the values of two or more variables. The user controls the number of function calls of $f(x)$ made on each pass in the single- and multidimension functions by adjusting the $*$ *scalar*1 $*$ and $*$ *scalar*2 $*$ parameters, respectively.

The $*counter1*$ parameter is used to simulate parallel processing. For example, if $*counter1* = 4$, then each call of both the multi- and single-dimension functions would actually be four sequentially implemented calls with the best of the four values returned used for further computation. In contrast, if four parallel processors were available, then each would be assigned the same job and called once, every time the program invoked either the multi- or single-dimension function. Torn and Zilinskas [1989] referred to this type of parallelism as Monte Carlo parallelism. In the discussion that follows, it is assumed that $*counter1* = 1$ in order to simplify the description.

2.1 The Controlling Function

Pseudocode for this function appears in Table II. Seven parameters are initialized (see Table III), and then the multidimension function is called to calculate the first minimum. A loop, consisting of a set of conditional definitions and calls to the single- and multidimension functions, is then established. The loop is exited following a call to the single-dimension function if the optimality criterion is reached ($0 < \textit{last-minimum} - \textit{last-score} < *\textit{exit}*$), if there are too many successive iterations without finding a new *last-minimum*, or if the number of trials equals the value of $*counter2*$. In the definition of the optimality criterion, *last-minimum* refers to the smallest value returned by all prior calls of the single-dimension function, while *last-score* refers to the value returned by the current call of the single-dimension function. The set of variable values that generated *last-minimum* is referred to as *best-set*.

In the loop, the *within-trial* and *within-trial-count* parameters are both markers for positions within a trial. A complete trial consists of an initial call to the single-dimension function, followed by repeated sequential calls of the multi- and single-dimension functions. The *count* parameter is an index of the number of complete trials. The *next-variable* parameter is used to designate the variable whose value is changed (bumped, i.e., either incremented or decremented) before each call to the multidimension function. A *bump-set* is used to pass all of the variable values to the multidimension function. This is accomplished by setting all of the variable values in *bump-set* identical to those in *best-set* with the exception of the changed bumped-variable value. The bumping serves two purposes: It creates an instability so that new local minima are found, eventually leading to the location of the global minimum; and it increases the likelihood that the variable space is adequately searched.

The *flag-direction* parameter controls the order in which the variables are fitted in the single-dimension function. For this purpose, the variables were treated as members of a closed set, which were rotated in either a forward or backward order, with the bumped variable readjusted last. For example, in a four-variable problem, if $\textit{variable}_2$ was the bumped variable and $\textit{flag-direction} = 0$, then the variables would be fitted in the order 3, 4, 1, 2. Alternatively, if $\textit{flag-direction} = 1$, then the variables would be fitted in the order 1, 4, 3, 2. The change of direction in the single-dimension fitting process is a

Table II. Pseudocode for the Controlling Function

```
(A) Initialize parameters (see Table III).
(B) Pass start-value to the multidimension function, and get the first last-minimum and the
    first best-set.
(C) Loop
(1) Set the within-trial parameter (starting value = 1).
    Purpose: To index developments within each trial.
    (a) If a new minimum is found and within-trial > 1, then no change.
    (b) Else, if within-trial = 3, then 1.
    (c) Otherwise, within-trial + 1.

(2) Set the within-trial-count parameter (starting value = 0).
    Purpose: To index developments within each trial.
    (a) If within-trial > 1, then within-trial-count + 1.
    (b) Otherwise, 0.

(3) Set the count parameter (starting value = 0).
    Purpose: To count the number of completed trials.
    (a) If a new trial (i.e., within-trial = 1), then count + 1.
    (b) Otherwise, count.

(4) Set the next-variable parameter (starting value = 0).
    Purpose: To index the next variable to be bumped.
    (a) The remainder of count/number-of-variables.

(5) Set the flag-direction parameter (starting value = 0).
    Purpose: To control the direction in which the variables are evaluated when the single-
    dimension function is called.
    (a) If flag-direction = 0, then 1.
    (b) Otherwise, 0.

(6) Set the cut parameter (starting value = cutoff).
    Purpose: To set the inside diameter of the torus.
    (a) If a new minimum is found and within-trial-count > 1, then for each variable the larger
        of cut/ * shrink-hit * and the minimum supplied by the user.
    (b) Else, if a new trial, then for each variable the larger of cut/ * shrink-trial * and the
        minimum supplied by the user.
    (c) Otherwise, cut.

(7) Set the new-range parameter (starting value = range).
    Purpose: To set the outside diameter of the torus.
    (a) If a new minimum is found and within-trial-count > 1, then for each variable the larger
        of new-range/ * shrink-hit * and the minimum-outside-of-the-torus.
    (b) Else, if a new trial, then for each variable the larger of new-range/ * shrink-trial * and
        the minimum-outside-of-the-torus.
    (c) Otherwise, new-range.

(8) Set the down-up parameter (starting value = 0).
    Purpose: To set the direction of changing the value of the variable indexed by next-variable
    (i.e., the bumped variable).
    (a) If a new minimum is found, within-trial-count > 1, and the value of the bumped
        variable is smaller than it was in the prior minimum, then 0.
```

Table II—*Continued*

(b) Else, if a new minimum is found, *within-trial-count* > 1, and the value of the bumped variable is greater than it was in the prior minimum, then 1.

(c) Else, if *within-trial* = 2, then 1.

(d) Else, if *down-up* = 1, then 0.

(e) Otherwise, 1.

(9) Set the *bump* parameter (starting value = 0).

Purpose: To calculate the magnitude by which the value of the bumped variable is changed.

(a) If *down-up* = 1, then the *new-range* value of *next-variable* times * *bump* *.

(b) Otherwise, the *new-range* value of *next-variable* times * *bump* * times −1.

(10) Set the *bump-set* parameter (starting value = first *best-set*).

Purpose: To calculate the variable values to be passed to the single- and multidimension functions.

(a) If a new trial, then *best-set*.

(b) Else, if the *best-set* value of the variable indexed by *next-variable* plus *bump* falls between the permissible upper and lower bounds, then *best-set* with the bumped variable altered.

(c) Else, if the *best-set* value of the variable indexed by *next-variable* minus *bump* falls between the permissible upper and lower bounds, then *best-set* with the bumped variable altered.

(d) Otherwise, *best-set*.

(11) Set the *last-minimum* parameter (starting value = first *last-minimum*).

Purpose: To save best *minimum-value*.

(a) If *flagz* = 0, then *last-score* (new minimum found).

(b) Otherwise, *last-minimum*.

(12) Set the *best-set* parameter (starting value = first *best-set*).

Purpose: To save the variable values associated with the *last-minimum*.

(a) If a new minimum found, then *last-set*.

(b) Otherwise, *best-set*.

(13) Set the *last-score* and *last-set* parameters (starting values returned by a call to the single-dimension function).

Purpose: To search for global minimum.

(a) If a new trial, pass *bump-set* to the single-dimension function.

(b) Otherwise, pass *bump-set* to the multidimension function. The set returned by the multidimension function then is passed to the single-dimension function. The values returned by the single-dimension function are the new values of *last-score* and *last-set*.

(14) Set the *flagz* parameter (starting value = 0 if new *last-minimum* obtained by the first call to the single-dimension function; otherwise, starting value = 1).

Purpose: A logical flag indicating whether or not a new minimum was obtained.

(a) If *last-score* < *last-minimum*, then 0.

(b) Otherwise, *Flagz* + 1.

(15) Exit test: Return to the beginning of the loop unless one of the following criteria is met:

(a) *Count* = * *counter*2 * (too many complete trials).

(b) *Flagz* = 36 (too many successive failures).

(c) 0 < *last-minimum* − *last-score* < * *exit* * (success).

Table III. The Seven Parameters Initialized in the Controlling Function

Parameter name	Definition
Range	For each variable, the upper - lower bound supplied by the user.
Cutoff-alt	For each variable, the larger of the *range/* * *torus* * and the minimum meaningful value supplied by the user.
Minimum-Outside-of-the-Torus	If the variable is an integer, then twice the minimum specified by the user. If the variable is a single float, then 32 times the minimum specified. If the variable is a double float, then 64 times the minimum specified.
Start-Value	For each variable, either the average of the upper and lower bounds or the starting value supplied by the user.
Number-of-Variables	Number of variables.
Single-dimension-Iterations	Round (10 * *scalar*1 *).
Multiple-dimension-Iterations	Round (10 * *scalar*2 * times *number-of-variables* squared).

simple way to minimize the interdependencies of the variables and leads to fairly stable local minima.

The *cut* and *new-range* parameters define the inside and outside diameters of the torus for each of the variables, and are adjusted following each call to the single-dimension function. Thus, these parameters constrain the range of the search. An additional use of *new-range* is to determine the value by which *next-variable* is bumped (*new-range* value of *next-variable* times * *bump* *). Since *new-range* changes each time the single-dimension function is called, the *bump* values shrink dynamically across iterations. The *down-up* parameter determines whether the bumped-variable will be incremented or decremented on the next call of the multidimension function, increasing the adequacy of the search.

Note that following each call to the single-dimension function, exit tests are performed, and if they are not satisfied, the *within-trial*, *within-trial-count*, *count*, *next-variable*, *flag-direction*, *cut*, *new-range*, *down-up*, and *bump* parameters are all updated. If a new minimum is found, *best-set* is updated. Preceding each call to the multidimension function, a new *bump-set* is created by either incrementing or decrementing the *best-set* value of *next-variable*. As noted earlier, the remaining variables in *best-set* are unaltered in *bump-set*.

When the *within-trial* parameter = 1, the single-dimension function is called. *Within-trial* is then incremented, and *bump-set* is calculated by incrementing the *best-set* value of *next-variable*. If the bumped value is outside the upper bound supplied by the user, *bump* is subtracted rather than added to the value of *next-variable*. The *bump-set* is then passed to the multidimension function. The set returned, in turn, is passed to the single-dimension function. If a new minimum is returned by the single-dimension function, *best-set* is updated, and *down-up* is altered to reflect the direction of change of the value of *next-variable*. If the value increased, then *down-up* is left at 1; if the value decreased, *down-up* is set to 0. A new *bump-set* is then calculated, and the process is repeated. Alternatively, if a new minimum is not returned by the single-dimension function, *within-trial* is set to 3, and *down-up* is changed from either 1 to 0 or 0 to 1. A new *bump-set* is then calculated, and the process continues to repeat as long as new minima are returned by the single-dimension function. Following the first failure to return a new minimum, the *trial-count* parameter is set to 1, and a new trial begins.

2.2 The Multidimension Function

Pseudocode for this function appears in Table IV. The number of function calls of $f(x)$ is equal to the *multiple-dimension-iteration* parameter. The logarithm of that parameter is also calculated, *log-constant*. The multidimension function consists of a loop in which the *iteration* parameter is set to 1 and augmented each time $f(x)$ is calculated. *Iterate-delta* is computed as 1 minus the ratio of the logarithm of *iteration* and *log-constant*. Prior to each function call, a *delta* score is independently calculated for each variable by an amount equal to *iterate-delta* multiplied by a random number between -1 and 1 multiplied by the *new-range* value of that variable. The absolute value of the *delta* for each variable must be greater than or equal to the *cutoff* value associated with that variable. The new value of each variable must fall between the lower and upper bounds of the variable supplied by the user. If a new minimum is obtained, it is stored, and its associated variable values become the center of a new torus; otherwise, the old minimum is retained.

2.3 The Single-Dimension Function

Pseudocode for the single-dimension function appears in Table V. This function is similar to the multidimension function except that the value of only one variable is changed prior to each function call of $f(x)$. The number of function calls associated with each variable is equal to the *single-dimension-iteration* parameter. As explained earlier, the variable that is fit first is determined by the *next-variable* and *flag-direction* parameters appearing in Table II. After the iterations associated with one variable are completed, the next variable follows until the entire set of variables is exhausted.

3. RESULTS

Three sets of results are presented. The first two sets involve comparisons with the findings reported by Corana et al. [1987] (two- and four-dimension Rosenbrock problems; two-, four-, and ten-dimension Parabolic Multiminima problems), while the third set involves 64 nonlinear problems collected by Hock and Schittkowski [1981].

3.1 The Rosenbrock Function

The Rosenbrock function in n-dimensions is defined as

$$f_n(x) = \sum_{k=1}^{n-1} 100(x_{k+1} - x_k^2)^2 + (1 - x_k)^2.$$

Table IV. Pseudocode for the Multidimension Function

```
(A) Set log-constant = ln(multiple-dimension-iterations); see Table III for the definition of
    multiple-dimension-iterations.
    Purpose: To bound the iterate-delta between 0 and 1.
(B) Loop
(1) Set the iterate parameter (starting value = 1).
    Purpose: To count the number of iterations.
    (a) If a new variable-set is not found (i.e., variable-flag = 1), then iterate.
    (b) Otherwise, iterate + 1.
(2) Set the iterate-delta parameter (starting value = 1).
    Purpose: To weight the values of the variables.
    (a) 1 − ln(iterate)/log-constant.
(3) Set the delta parameter (starting values = 0 for all variables).
    Purpose: To calculate a set of bounded stochastic values by which the values in variable-set
    can be modified.
    (a) Set $d_i$ = iterate-delta × new-range value of $variable_i$ × a random number between −1
        and 1. (Note that new-range is defined in Table II and that a different random number
        is sampled for each variable.)
    (b) If the absolute value of $d_i$ < cut value of $variable_i$, then if $d_i < 0$ set $d_i$ = −cut value
        of $variable_i$; otherwise, set $d_i$ = cut value of $variable_i$ (minimum change for a variable;
        the cut parameter is defined in Table II).
    (c) Otherwise, $d_i$.
(4) Set the variable-set parameter (starting value = bump-set; see Table II for a definition.)
    Purpose: To construct a set of variables to pass to the function that is to be minimized.
    (a) If $d_i < 0$ and if $v_i + d_i > l_i$ (where $d_i$, $l_i$, and $v_i$ are the delta, lower bound, and
        value-set values associated with $variable_i$), then $v_i + d_i$; otherwise, $v_i$.
    (b) Otherwise, if $d_i > 0$ and if $v_i + d_i < u_i$ (where $d_i$, $u_i$, and $v_i$ are the delta, upper bound,
        and value-set values associated with $variable_i$), then $v_i + d_i$; otherwise, $v_i$.
(5) Set the variable-flag parameter (starting value = 0).
    Purpose: To flag if the values in variable-set and value-set are identical for all variables.
    (a) If the values in variable-set and value-set are identical, then 1.
    (b) Otherwise, 0.
(6) Set the value parameter (starting value = value returned by the function to be minimized
    when passed the bump-set).
    Purpose: To compute a value for each function call.
    (a) Value returned by the function when passed the variable-set.
(7) Set the value-flag parameter (starting value = 0).
    Purpose: To flag if a new minimum obtained.
    (a) If value < best-value, then 1.
    (b) Otherwise, 0.
(8) Set the best-value parameter (starting value = value).
    Purpose: To store the minimum value obtained across iterations.
    (a) If value-flag = 1, then value.
    (b) Otherwise, best-value.
```

Table IV—*Continued*

```
(9) Set value-set parameter (starting value = bump-set).
    Purpose: To store the values of the variables that generate best-value.
    (a) If value-flag = 1, then variable-set.
    (b) Otherwise, value-set.
(10) Exit test: Return to the beginning of the loop unless the following criterion is met:
    (a) Multiple-dimension-iterations = iterate.
```

Table V. Pseudocode for the Single-Dimension Function

```
(A) Outer Loop
(1) Set the count1 parameter (starting value = 1).
    Purpose: Used to calculate count2 and as the exit criterion for the Outer Loop.
    (a) Count1 + 1.
(2) Set the count2 parameter (starting value = if flag-direction is 0, then the modulus (next-variable + 1)/number-of-variables; otherwise, modulus (next-variable − 1)/number-of-variables; see Table II for the definitions of flag-direction and next-variable).
    Purpose: To select the variable to be changed in the Inner Loop.
    (a) If flag-direction = 0, then modulus (next-variable + count1)/number-of-variables.
    (b) Otherwise, modulus (next-variable − count1)/number-of-variables.
(3) Set the result parameter (starting value = call Inner Loop passing count2 and bump-set; see Table II for a definition of bump-set).
    Purpose: To compute new minimum and associated set of variables.
    (a) Call the Inner Loop passing count2 and the set of variables stored in result.
(4) Exit test: Return to the beginning of the Outer Loop unless the following criterion is met:
    (a) Count1 = number-of-variables.
(B) Inner Loop
    (Note: The Inner Loop is identical to the multidimension function outlined in Table IV except that log-constant is set to ln(single-dimension-iterations), and only the variable designated by count2 is changed on each pass. Single-dimension-iterations is defined in Table III.)
```

Using the Rosenbrock function with $n = 2, 3, \ldots, 10$, it was determined that when the Torus algorithm is used the number of function calls to obtain a solution increases approximately as the square of the number of variables. For this reason, *multiple-dimension-iteration* was set to a function of the number of variables squared (see Table III).

Corana et al. [1987] performed a series of two- and four-variable tests using the Rosenbrock function. The admissible domains of the variables were $-2000 < x1,\ x2 < 2000$, and $-200 < x1, x2, x3, x4 < 200$. The results Corana et al. obtained using the Nelder–Mead Simplex and Simulated Annealing algorithms appear in Table VI, as do the results obtained using the Torus algorithm with both one and four processors. As can be seen in the table, the Simplex algorithm failed to solve two of the four-variable problems, but converged in fewer iterations than did the other algorithms.

The four-processor Torus algorithm was run using all default values, except that $* exit *$ was set to 1.0E-8 to produce median final function values in the range of the other algorithms. Default values were also used in the one-processor Torus algorithm, except that $* scalar2 *$ was set to 4. This adjustment forced the same total number of iterations for each call of the multidimension function for the one- and four-processor versions (i.e., the single processor used four times as many iterations per call as did each of the four processors). Since, among those tested, the Rosenbrock equation is one of the most difficult problems for the Torus algorithm to solve, the default value of $* scalar2 *$ is not great enough to always generate the solution (see Table VI). However, over 80 and 90 percent of the runs generated the correct solution in the one- and four-processor versions, respectively. In comparison to the Simplex algorithm, the Torus algorithm is computationally expensive. However, the Simulated Annealing algorithm used more than 30 times as many iterations as did the one-processor Torus and more than 100 times as many iterations as did the four-processor Torus algorithm.

3.2 The Parabolic Multiminima Function

Corana et al. [1987] constructed a family of test functions that are simple to compute, that contain a high number of minima, and that are defined in a limited domain. As noted earlier, the Nelder–Mead Simplex algorithm fails to solve these problems. Corana et al. described the function as follows:

Let D_f be the domain of definition of the function q_n in n-space:

$$D_f \equiv \{x \in \mathbb{R}^n: -a_1 \le x_1 \le a_1, \dots, -a_n \le x_n \le a_n;\ a \in \mathbb{R}^n_+\}.$$

D_f is a rectangular subdomain of R^n, centered around the origin and whose width along each coordinate direction is determined by the corresponding component of the vector **a**.

Let D_m be the family of open, disjoint, rectangular subdomains of R^n contained in D_f and defined as

$$\begin{aligned} d_{k_1,\dots,k_n} \equiv \Big\{ & x \in D_f : k_1 s_1 - t_1 < x_1 < k_1 s_1 + t_1 \\ & , \dots, k_n s_n - t_n < x_n < k_n s_n + t_n; \\ & k_1, \dots, k_n s \in \mathbb{R}^n_+;\ t_i < \frac{s_i}{2},\ i = 1, \dots, n \Big\}; \\ D_m \equiv & \bigsqcup_{k_1,\dots,k_n \in \mathbb{Z}} d_{k_1,\dots,k_n} - d_{0,0,\dots,0}. \end{aligned}$$

D_m is the open subset of D_f where the function q_n present local minima. The vector **s** controls the grid steps along each axis, the grid points being the centers of these subdomains, while the vector **t** controls the size of these subdomains. The condition $t_i < s_i/2$ ensures that the subdomains are disjoint.

Let D_r be the closed subdomain of D_f complementary to D_m:

$$D_r \equiv D_f - D_m.$$

Table VI. Rosenbrock Output from Three Optimization Algorithms[a]

Starting point	Nelder–Mead Simplex: Number of function evaluations	Nelder–Mead Simplex: Final function value	Simulated Annealing: Number of function evaluations	Simulated Annealing: Final function value	Torus, one processor: Number of function evaluations[b]	Torus, one processor: Final function value[c]	Torus, four processors: Number of function evaluations[b]	Torus, four processors: Final function value[c]
Two dimensions								
1001, 1001	993	4 9E-10	500,001	1.8E-10	11,440 (2)	1.2E-11	3,530 (0)	9.5E-11
1001, −999	276	7.4E-10	508,001	2.6E-9	11,620 (3)	3.0E-10	3,430 (0)	2.1E-10
−999, −999	730	2.7E-10	524,001	1.2E-9	11,840 (1)	2.0E-10	3,420 (0)	3.7E-10
−999, 1,001	907	9.2E-10	484,001	4.2E-8	11,820 (1)	9.4E-11	3,180 (1)	5.8E-10
1443, 1	907	5.4E-11	492,001	1.5E-8	11,650 (3)	2.0E-10	3,330 (1)	1.0E-10
1, 1443	924	2.2E-10	512,001	1.6E-9	11,460 (2)	3.6E-10	3,600 (0)	5.9E-11
1.2, 1	161	2.4E-10	488,001	2.0E-8	11,280 (0)	2.3E-9	3,140 (1)	8.7E-8
Four dimensions								
101, 101, 101, 101	1,869	3.70	1,288,001	5.0E-7	36,610 (2)	2.8E-9	9,540 (1)	3.8E-10
101, 101, 101, −99	784	5.5E-17	1,328,001	1.8E-7	37,150 (2)	3.9E-9	8,980 (1)	2.4E-10
101, 101, −99, −99	973	9.8E-18	1,264,001	5.9E-7	38,570 (2)	3.4E-9	9,680 (1)	6.6E-10
101, −99, −99, −99	1,079	3.4E-17	1,296,001	7.4E-8	37,860 (1)	5.9E-10	9,880 (0)	1.4E-10
−99, −99, −99, −99	859	8.3E-18	1,304,001	3.3E-7	37,270 (2)	1.7E-9	9,560 (1)	1.8E-10
−99, 101, −99, 101	967	1.2E-17	1,280,001	2.8E-7	40,400 (2)	6.8E-8	9,190 (2)	5.7E-10
101, −99, 101, −99	870	6.0E-18	1,272,001	2.3E-7	35,190 (2)	3.1E-9	10,070 (0)	4.4E-10
201, 0, 0, 0	1,419	3.70	1,288,001	7.5E-7	35,200 (3)	5.6E-8	9,340 (0)	2.4E-10
1, 201, 1, 1	1,077	9.5E-18	1,304,001	4.6E-7	39,850 (2)	8 3E-10	9,540 (0)	1.9E-10
1, 1, 1, 201	1,265	3.9E-17	1,272,001	5.2E-7	37,910 (2)	8.9E-9	9,170 (4)	2.5E-9

[a]Data for the Nelder–Mead Simplex and Simulated Annealing algorithms are from Corana et al. [1987].

[b]Median number of iterations per processor for 11 passes. The numbers in brackets indicate the number of passes that failed to find the minima, $f(x) > 1.0E\text{-}4$.

[c]Median value of 11 passes

The test function $q_n(\mathbf{x})$ of n real variables is defined as

$$q_n(\mathbf{x}): D_f \to \mathbb{R}_+,$$

$$q_n(\mathbf{x}) \equiv \sum_{i=1}^{n} d_i x_i^2, \qquad \mathbf{x} \in D_r, \quad \mathbf{d} \in \mathbb{R}_+^n,$$

$$q_n(\mathbf{x}) \equiv c_r \sum_{i=1}^{n} d_i z_i^2, \qquad \mathbf{x} \in d_{k_1,\dots,k_n}, \quad (k_1,\dots,k_n) \neq 0,$$

where

$$z_i = \left(\begin{array}{ccc} k_i s_i + t_i & \text{if} & k_i < 0 \\ 0 & \text{if} & k_i = 0 \\ k_i s_i - t_i & \text{if} & k_i > 0 \end{array} \right)$$

The components of the vector $\mathbf{d}$ determine the steepness of the paraboloid along the axes, while the coefficient c_r controls the depth of local minima relative to the function values along the boundaries of the regions $d_{k_1,\dots,k_n}$. It is worth noting that the region $d_{0,\dots,0}$ belongs to the subdomain D_r, where q_n is not constant around the origin, which is the unique global minimum of q_n [Corana et al. 1987, pp. 71–72].

In order to evaluate the performance of the algorithms, Corana et al. [1987] set the parameters of the test function to the following values:

$$\begin{array}{ll} a_i = 10^4, & i = 1,\dots,n; \\ s_i = 0.2, & i = 1,\dots,n; \\ t_i = 0.05, & i = 1,\dots,n; \\ \mathbf{d} = (1, 1000), & n = 2; \\ \mathbf{d} = (1, 1000, 10, 100), & n = 4; \\ \mathbf{d} = (1, 1000, 10, 100, 1, 10, 100, 1000, 1, 10), & n = 10; \\ c_r = 0.15. & \end{array}$$

The number of grid points associated with each dimension is $2a_i/s_i$. Given the values of a_i and s_i selected by Corana et al. [1987], the total number of local minima of the test function $q_n(\mathbf{x})$ is $10^{5n} - 1$.

The four-processor Torus algorithm was run using all default values, except that $*$ *exit* $*$ was set to 1.0E-8 to produce median final function values in the range of the Simulated Annealing algorithm. Default values were also used in the one-processor Torus algorithm, except that $*$ *scalar2* $*$ was set to 4. Unlike the Rosenbrock function, the Parabolic Multiminima function is fairly easily solved by the Torus algorithm and is more dependent on the number of iterations per processor than on the number of processors involved. As can be seen in Table VII, the Simulated Annealing algorithm is considerably slower and less robust than the Torus algorithm. It is especially interesting that on the ten-dimension problem the Simulated Annealing algorithm is trapped by the local minimum immediately adjacent to the global minimum. The one-processor version of the Torus algorithm converged on all runs of all Parabolic Multiminima problems tested, while the four-processor version

Table VII Parabolic Multiminima Function Output from Two Optimization Algorithms[a]

Starting point	Simulated annealing		Torus, one processor		Torus, four processors	
	Number of function evaluations	Final function value	Number of function evaluations[b]	Final function value[c]	Number of function evaluations[b]	Final function value[c]
Two dimensions (T_0 = 1.0E8, ϵ = 1.0E-4, s = 0.2, t = 0.05)						
1000, 888	684,000	2.5E-8	11,810	4.1E-8	3,500	1.9E-8
−999, 1001	680,000	3.2E-9	11,300	2.4E-8	3,430	5.0E-8
−999, −889	708,000	4.0E-9	12,870	7.3E-8	3,550	1.7E-8
1001, −998	696,000	1.2E-9	12,520	1.1E-7	3,520	3.7E-8
1441, 3	708,000	3.2E-9	13,380	4.6E-9	3,470	3.0E-8
−10, −1410	680,000	4.0E-8	12,130	3.2E-8	3,500	2.0E-8
−1100, 850	696,000	1.2E-8	12,170	1.0E-8	3,250	1.0E-7
850, −1100	656,000	4.2E-10	11,790	6.3E-8	3,540	1.6E-8
Four dimensions (T_0 = 1.0E8, ϵ = 1.0E-4, s = 0.2, t = 0.05)						
−999, −999, −9999, −1000	1,440,000	2.6E-7	43,780	2.5E-7	11,320	2.2E-7
999, 1000, 1001, −998	1,160,000	3.4E-3	47,760	1.7E-7	13,060	1.4E-7
1000, −1000, 10000, −10000	1,464,000	8.7E-8	44,430	2.2E-7	10,940	4.0E-7
−999, −999, −998, −1000	1,440,000	2.0E-7	45,120	2.1E-7	11,320	8.0E-7
1000, 999, 999, 998	1,424,000	3.4E-7	45,750	2.3E-7	12,210	2.8E-7
1000, −1000, −9999, 9999	1,416,000	2.5E-7	43,780	3.7E-7	11,620	2.0E-7
1000, −1000, 998, 1000	1,176,000	3.4E-3	43,160	4.5E-7	11,960	1.9E-7
0, 0, 1, 2001	1,408,000	4.0E-7	44,470	2.2E-7	10,980	4.1E-7
1998, 3, 10, −13	1,408,000	4.6E-7	47,110	1.1E-7	12,550	3.2E-7
1234, −1234, 560, −334	1,432,000	3.4E-7	41,160	6.0E-7	10,770	2.7E-7
Ten dimensions (T_0 = 1.0E9, ϵ = 1.0E-4, s = 0.1, t = 0.04)						
1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000	1,638,000	5.4E-4	276,930	1.5E-6	62,790[d]	2.1E-6
−1000, 1000, −1000, 1000, −1000, 1000, −1000, 1000, −1000, 1000	1,638,000	5.4E-4	256,830	1.1E-6	60,790	2.5E-6
−999, −999, −999, −999, −999, −999, −999, −999, −999, −999	1,638,000	5.4E-4	240,820	2.7E-6	59,760	2.1E-6
999, 999, 999, 999, 999, −999, −999, −999, −999, −999	1,548,000	5.4E-4	264,920	2.0E-6	60,790	1.9E-6
−999, 1000, −999, 1000, −999, 1000, −999, 1000, −999, 1000	1,593,000	5.4E-4	260,880	2.0E-6	56,730[d]	2.5E-6
3000, 4, 20, 40, 120, −3, −6, 0, 0, 100	1,638,000	5.4E-4	244,830	2.3E-6	59,780[d]	2.3E-6
1000, −999, 1000, −999, 1000, −999, 1000, −999, 1000, −999	1,665,000	5.4E-4	252,880	1.1E-6	62,790	1.5E-6
1000, −999, 1000, −999, 1000, −999, 1000, −999, 1000, −999	1,611,000	5.4E-4	272,940	2.6E-6	60,790	2.0E-6

[a]Data for the Simulated Annealing algorithm are from Corana et al. [1987]

[b]Median number of iterations per processor for 11 passes.

[c]Median value of 11 passes

[d]One of the 11 passes failed to converge, $q_n(x) > 1.0\text{E-}4$

failed to converge on only 3 of the 88 runs on the ten-dimension problems. To demonstrate that the Parabolic Multiminima function is primarily dependent on the number of iterations per processor, as compared to the number of processors, the $*$ *scalar*2 $*$ parameter was set to 1, and the ten-dimension problems were refit with the one-processor version of the Torus algorithm. The algorithm converged to the correct solution on 87 of the 88 runs, but took about 10 percent more iterations per processor than did the four-processor Torus algorithm. Considering Corana et al.'s [1987] arguments regarding the difficulty of the Parabolic Multiminima function, the increase in both speed and robustness associated with the Torus algorithm, compared to the Simulated Annealing algorithm, is significant.

3.3 Hock and Schittkowski, 64 Problems

Hock and Schittkowski [1981] prepared a monograph containing 118 nonlinear programming problems with continuously differentiable functions. They evaluated the ability of six different nonstochastic algorithms to solve these test problems. In order to evaluate the abilities of the Nelder–Mead Simplex algorithm and one-, two-, four-, and eight-processor versions of the Torus algorithm to solve problems of this type, Hock and Schittkowski's [1981] first 60 problems were used as part of a test set. Since Problem 58 was omitted in their monograph, Problem 61 was included. Problems 65, 83, 85, and 117 completed the test set. They were included because either none of the algorithms described by Hock and Schittkowski solved the problem or the problem involved fitting functions to data. The results appear in Table VIII.

In order both to demonstrate the power of the Torus algorithm and to provide a conservative estimate of savings associated with adding processors, default values were used with all of the parameters. In particular, $*$ *scalar*2 $*$ was set to 1, independent of the number of processors used. Thus, the one-processor Torus algorithm was seriously underpowered, while the eight-processor model was somewhat overpowered. Means, rather than medians, are provided in this table because they are relevant to the analyses of variances reported in the last two columns. These analyses involved comparing the number of iterations and the final function values provided by one-, two-, four-, and eight-processor versions of the Torus algorithm. Inspection of the F-values in these two columns reveals a substantial number of significant differences reflecting that as the number of processors increased the number of iterations per processor tended to decrease, as did the error and variability associated with the final function value.

The function values reported in Table VIII are considered to be failures if they deviate from the correct function values by more than 0.01. Using this criterion, the Nelder–Mead Simplex algorithm failed on 34 of the 64 problems, while the one-, two-, four-, and eight-processor Torus algorithms failed on 18, 10, 9, and 9 of the 64 problems, respectively. The six nonstochastic algorithms tested by Hock and Schittkowski [1981] failed 10, 11, 11, 13, 15, and 13 times. In comparison to the other algorithms, the performance of the Torus algorithm was satisfactory, surprisingly so in the case of the one-

Table VIII Output from the Nelder–Mead Simplex and Torus Algorithms for Selected Problems in Hock and Schittkowski [1981][a]

		Simplex		Torus, one processor		Torus, two processors		Torus, four processors		Torus, eight processors			
Problem	Correct function value	Number of function evaluations	Final function value	Number of function evaluations	Final function value	Number of function evaluations	Final function value	Number of function evaluations	Final function value	Number of function evaluations	Final function value	Iteration F-test $df = 3, 40$	Value F-test $df = 3, 40$
1	0	92	1.2E-6	1,686	7 7E-6	1,801	2 2E-6	1,415	2 7E-6	1,380	2 4E-6	3.32*	1.07
2	0.0504	89	0.0504	2,078	0.1715	2,077	0 0505	1,959	0 0504	1,488	0 0508	2.53	3.57*
3	0	177	0.001	231	0 0038	136	0 0014	182	0.0018	85	0.0018	2.59	1.48
4	2 6667	177	2.6667	4,153	2.6667	3,064	2 6667	3,498	2 6667	2,285	2.6667	7.83**	0.30
5	−1 9132	56	−1 9132	725	−1.9132	555	−1 9132	444	−1 9132	303	−1 9132	13.38**	0.36
6	0	536	5.6E-7	4,232	0.0031	2,275	3 3E-6	1,953	4.0E-6	1,815	1 70E-6	13.26**	1.02
7	−1.7321	368	−1.7323	1,268	−1.7320	1,366	−1 7320	1,077	−1.7322	845	−1.7322	5.57**	0 78
8	−1	184	−0 9999	3,775	−0 9998	3,659	−0.9999	3,960	−0.9999	4,045	−1	0.64	8.90**
9	−0 5	94	−0 5	2,701	−0.4997	2,191	−0.5	2,050	−0 5	2,038	−0 5	4.22	1.29
10	−1	143	−0 9999	1,968	−1	2,078	−1	2,472	−1	2,440	−1	0.61	—[b]
11	−8 4985	179	−8.4985	4,593	−8.4984	3,965	−8.4983	4,081	−8.4984	4,304	−8 4985	1.52	1.42
12	−30	155	−29 9999	4,240	−30	4,185	−30	3,929	−30	3,094	−30	3 71*	1 13
13	1	131	2 9588	3,728	1 0025	2,544	1	2,785	1	2,712	1	4.88**	0.99
14	1 3934	145	1 3919	3,480	1 3919	3,993	1.3919	3,524	1 3919	3,141	1 3919	2.56	8 18**
15	306.5	571	306.5	3,791	345 70	4,051	340.80	4,147	326.10	4,029	326 09	0.42	1 58
16	0.25	122	0.25	1,373	0 25	1,335	0.25	1,268	0.25	1,345	0 25	0.18	0.48
17	1	148	1	2,834	1	2,735	1	2,722	1	2,684	1	0 07	0.30
18	5	142	7 1720	3,851	5.0001	3,219	5	2,597	5	2,430	5	4.55**	3 21*
19	−6,961.8138	375	−6,961.8139	4,590	−6,142.83	4,545	−6,960 12	4,525	−6,960 45	3,895	−6,960.68	9.69**	1 00
20	38.1987	257	58 5	4,238	39.8378	4,143	39.8372	3,940	39.4730	3,775	39 4725	0 80	0 58
21	−99 96	133	−99 9599	1,375	−99 96	1,253	−99 96	1,285	−99.96	971	−99 96	5 31	1.02
22	1	208	1	1,391	1 0021	1,459	1.0019	825	1.0019	843	1.0011	2 68	0 47
23	2	205	2	4,426	2 0004	4,066	2 0002	4,355	2.0002	4,409	2 0001	1.81	11 85**
24	−1	147	−0.7482	1,325	−0 9944	850	−0 9976	920	−0 9991	934	−0.9982	0 86	1 71
25	0	—[c]	—	1,571	6 7E-5	1,594	6 2E-6	1,292	7 5E-5	1,054	4 1E-5	3 11*	0.76
26	0	425	4 6E-6	6,035	9 1E-4	6,090	0 0061	3,641	1 2E-4	2,962	1.3E-6	3 50*	0.93
27	0.04	114	55 7100	2,250	0 0450	2,098	0 0401	1,857	0.04	1,746	0.04	1.73	1 00
28	0	147	1 2E-6	4,497	0.0332	3,780	2 3E-5	3,427	9.4E-6	3,033	9 1E-5	5 64**	2.92*
29	−22 6274	191	−18 1019	9,208	−22 6272	8,575	−22.6274	8,177	−22.6274	7,673	−22.6274	1 55	4.26*
30	1	190	1	1,905	1 0001	1,573	1.0001	1,361	1	1,259	1	8 59**	1.43
31	6	209	6.2256	3,581	6.0040	2,325	6 0007	2,268	6.0021	3,094	6.0013	4.15*	1.28
32	1	377	1 2227	7,604	1	7,759	1	7,034	1	7,670	1	0.33	0 84
33	−4 5858	220	−4 1766	6,705	−4 5857	6,988	−4 5857	7,093	−4 5858	6,685	−4.5857	0 25	7.39*
34	−0 8340	249	−0 0671	9,155	−0.8253	7,411	−0.8317	7,015	−0.8340	7,432	−0.8340	2 66	0.84
35	0 1111	182	0 5294	7,061	0 1111	6,713	0.1111	5,016	0.1111	5,066	0.1111	4.42**	1.99
36	−3,300	443	−3,207 9050	8,955	−3,299 0568	7,593	−3,299.9907	8,279	−3,299.9945	7,535	−3,299 9966	1 85	1.99
37	−3,456	363	−3,071.9999	12,772	−3,454 4396	11,044	−3,455 9871	8,923	−3,455 9996	8,519	−3,455 997	7 34**	1.28

38	0	607	7.5E-7	6,988	0.0574	5,624	8.3E-6	5,259	2.7E-6	5,088	3.5E-5	2 79	1.03
39	−1	152	−0.3849	4,726	−1	4,515	−0.9999	3,618	−0.9995	3,900	−0.9999	2 84*	1 17
40	−0.25	297	−0.2475	12,893	−0.2128	14,079	−0.2224	12,240	−0.2480	12,832	−0.2484	1 01	1.28
41	1.9259	157	2	15,002	1 9260	10,191	1.9260	10,280	1.9259	6,739	1.9259	9 00**	4.86**
42	13 8579	218	16 0936	4,158	13.8586	4,465	13.8586	3,327	13.8594	3,492	13 8587	2.75	0.65
43	−44	326	−41.7083	16,276	−43.9985	14,420	−43.9999	14,064	−43.9999	12,975	−43.9999	4.07	2.72
44	−15	573	−5.0417	10,914	−14.8177	11,682	−14.9996	11,867	−14.9997	11,755	−14.9998	0 38	1.00
45	1	389	1	14,935	1	14,161	1	12,733	1	11,370	1.0001	0.94	0.65
46	0	579	0.1014	21,705	4.5E-4	19,609	3.5E-4	18,999	7.4E-5	19,227	4.8E-5	0.65	1.70
47	0	845	2.2448	20,334	0.7372	19,276	0.3329	20,516	0.2800	18,399	0.1110	0 66	5.14**
48	0	466	4.7E-4	20,950	5 3E-5	18,875	5.2E-5	17,528	1.4E-4	17,209	3.0E-5	1 17	0.92
49	0	396	0.1502	22,063	0.0015	21,924	0.0012	18,731	3.6E-4	16,552	0.0010	2.92*	0.75
50	0	369	0.0658	22,072	1 5E-4	21,185	1.3E-4	18,984	1.0E-4	19,601	7.6E-6	1.50	5.26**
51	0	486	1 4E-5	19,146	1.1E-4	20,510	6 8E-5	18,205	6.0E-5	16,928	3.7E-5	1.41	4.66*
52	5.3266	454	5.5000	22,801	5.3282	20,916	5.3271	19,347	5.3269	20,213	5.3268	0 91	13.56**
53	4.0930	710	5.5001	25,974	4.0959	23,525	4.0936	22,393	4.0934	18,028	4.0934	4 89**	2.17
54	−0 9081	—[c]	—	12,527	−0 6025	10,863	−0 4811	10,615	−0.5373	10,206	−0.6668	3.02	0.60
55	6.3333	386	7.0000	29,925	6.6294	23,984	6.6554	22,801	6.6475	20,888	6.6681	1.34	0.31
56	−3.4560	—[c]	—	38,068	−2.3430	40,166	−3.1887	37,197	−3 2653	32,193	−3.3780	1 38	16.30**
57	0 0285	86	0.0306	1,730	0 0285	1,324	0.0285	1,404	0.0285	1,249	0.0285	1.65	1.46
59	−7 8042	67	−6.7491	2,434	−7.3240	3,299	−7.8028	2,863	−7.8028	2,616	−7.8028	1.39	8.33*
60	0.0326	155	0.1635	8,435	0.0326	7,160	0 0326	6,909	0.0326	8,002	0.0326	2 24	2.36
61	−143.6461	161	−131.1619	7,940	−143.6460	7,751	−143.6461	8,349	−143.6461	8,031	−143.6461	0 40	5.64**
65	0 9535	260	0.9535	7,325	0 9535	6,277	0.9535	4,829	0.9536	5,660	0.9535	4.31**	0.78
83	−30,665.5388	825	−30,057.0107	22,499	−30,641.8739	20,831	−30,665 5106	18,925	−30,655.5185	18,493	−30,655.5188	3.25*	1.00
86	−32.3487	641	−21.2775	26,846	−32.3155	22,257	−32.3409	19,369	−32.3451	19,913	−32.3462	4.94**	5.86**
117	32.3487	3362	79.6700	258,885	52.5106	250,026	42.2376	312,115	37.8829	302,933	37 1571	0 67	12.01*

[a]The data reported in the Torus columns are means of 11 runs.

[b]No *F*-test is reported because the algorithm failed to leave the starting point on 8, 8, 5, and 2 of the 11 runs involving the 1-, 2-, 4-, and 8-processor Torus algorithms, respectively.

[c]The Nelder–Mead Simplex algorithm failed to converge to a value.

$^{*}p < 0.05$

$^{**}p < 0.01$

processor version since it was underpowered. Furthermore, even on the problems in which the average of the 11 runs did not fall within 0.01 of the solution, either exact or approximate solutions were returned on one or more of the runs by the Torus algorithm. Although the Torus algorithm does not always find exact solutions, running the algorithm more than once on a particular problem will nearly always generate a useful approximation to the solution.

As a consequence of the stochastic nature of the Torus algorithm, multiple solutions to problems can be obtained when the algorithm is run repeatedly. Problems 8, 9, 29, and 56 from Hock and Schittkowski [1981] have multiple solutions. Recall that 11 runs were made with each problem. The four-processor version of the Torus algorithm returned all 4 possible solutions with Problem 8, 7 of the 13 possible solutions ($-100 < x_1, x_2 < 100$) with Problem 9, 3 of the 4 possible solutions with Problem 29, and 7 approximate solutions of 8 possible solutions with Problem 56.

4. DISCUSSION

The data in Tables VI and VIII reflect that the Nelder–Mead Simplex algorithm is less robust (i.e., solves fewer problems) but is more efficient (i.e., uses fewer function calls) than the Torus algorithm. If the Simplex algorithm is providing correct solutions to a class of optimization problems, one would be ill advised to replace it with the Torus algorithm. However, its use in conjunction with the Torus algorithm would seem well advised for most problems. Because of its much greater robustness, the Torus algorithm should be used initially and as an occasional check to establish that the Simplex algorithm, or some other efficient but less robust algorithm, is returning global minima.

The trade-off between efficiency and robustness does not occur when the Torus and Simulated Annealing algorithms are compared. As can be seen in Tables VI and VII, the Torus algorithm is much faster than the Simulated Annealing algorithm in all comparisons. The Torus algorithm is also more robust in solving the Parabolic Multiminima Function. Furthermore, if the value of the $*scalar2*$ parameter is increased from the value of 1 used in Table VI, the error rate of the Torus algorithm in fitting the Rosenbrock equation approaches 0, while the number of function calls remains several magnitudes below that for the Simulated Annealing algorithm.

Torn and Zilinskas [1989] discussed several metrics to evaluate parallelism. The speedup (time to solve for one processor/time to solve for m processors) is an adequate metric for present purposes. Inspection of Tables VI–VIII reveals that speedup depends on the problem considered and the adjustment of the parameters that control the behavior of the algorithm. If only the one- and four-processor Torus algorithms are considered, the speedup ranges from a high of 4.40 in Table VI, a Rosenbrock four-dimension problem, to a low of 0.92 in Table VIII, Problem 44. In most of the examples, but not all, there are gains in speedup, efficiency, and robustness associated with increasing the number of processors while using the Torus algorithm.

The Torus algorithm is relatively robust and efficient compared to other Monte Carlo algorithms. It would seem to be a procedure of choice if multiple solutions are of interest, if functions are not continuous, and if parameters represent integers, rather than continuous variables. The Torus algorithm should also be considered if more efficient algorithms do not generate satisfactory solutions, if it is necessary to establish boundary conditions for each variable so that the function of interest is computable, and if it is desirable to validate a more efficient procedure.

ACKNOWLEDGMENT

Special thanks are due to Avery Earle, Richard Charron, Malcolm Grant, Mark Howe, and an anonymous reviewer for their help and encouragement. Requests for reprints should be addressed to the author at the Department of Psychology, Memorial University of Newfoundland, St. John's, Newfoundland, Canada A1B 3X9.

REFERENCES

CORANA, A., MARCHESI, M., MARTIN, C., AND RIDELLA, S. 1987. Minimizing multimodal function of continuous variables with the "simulated annealing" algorithm. *ACM Trans. Math. Softw. 13*, 262–280.

HOCK, W. AND SCHITTKOWSKI, K. 1981. Test examples for nonlinear programming codes. *Lect. Notes Econ. Math. Syst. 187*, 1–177.

KIRKPATRICK, S., GELATT, C. D., AND VECCHI, M. P. 1983. Optimization by simulated annealing. *Science 220*, 671–679.

LUENBERGER, D. G. 1986. *Linear and Nonlinear Programming*. Addison-Wesley, Reading, Mass.

MASRI, S. F. AND BEKEY, G. A. 1980. A global optimization algorithm using adaptive random search. *Appl. Math. Comput. 7*, 353–375.

NELDER, J. A. AND MEAD, R. 1965. A simplex method for function minimization. *Comput. J. 7*, 308–313.

PANNETIER, J. 1990. Simulated annealing: An introductory review. *Inst. Phys. Conf. Ser. 107*, 23–44.

PRESS, W. H., FLANNERY, B. P., TEUKOLSKY, S. A., AND VETTERLING, W. T. 1988. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York.

TORN, A. AND ZILINSKAS, A. 1989. Global optimization. In *Lecture Notes in Computer Science*, vol. 350, G. Goos and J. Hartmanis, Eds. Springer-Verlag, New York, 1–255.

Received February 1993; revised April and May 1994; accepted June 1994