The 3-2-3, Stratified Split and Nested Interval Line Search Algorithms

Fred Glover
OptTek Systems, Inc.
Revised July 5, 2010

Abstract

This note describes three line search algorithms for multi-modal function optimization that incorporate a combination of binary search and direct search, called the *3-2-3*, the *Stratified Split and Nested Interval Algorithms*. The methods are designed to be used as subroutines of a more general global optimization procedure over a bounded search space, as a way to perform effective line searches without the limiting assumption of unimodality.

1. Introduction

The literature on optimization contains a wide variety of line search algorithms for finding a minimum of a nonlinear function f(y) on a line (or a line segment in the bounded variable case), under the assumption that f(y) is unimodal over the domain of interest (see, e.g., Himmelblau, 1972; Knuth, 1997; Lasdon, 2002; Mangasarian, 1969; Murty, 1988; Nocedal and Wright, 1999). In this note we focus on line search algorithms for the setting where f(y) is a multi-modal function, to provide methods that can be used as subroutines for more general global optimization methods over bounded spaces.

The problem we address may be expressed as that of minimizing f(y) on a line segment denoted by LS(y',y"), which passes through points y' and y", where

$$LS(y',y'') = \{y = y(\theta):\ y(\theta) = y' + (y'' - y')\theta \text{ for } \theta_{min} \le \theta \le \theta_{max}\}. \qquad (1.1)$$

This representation is motivated by an application involving Direct Search methods in global optimization, using a design where y' and y" are strategically or randomly generated to lie within a particular region of the solution space. The values θ_{min} and θ_{max} are computed so that LS(y',y") lies within a bounded convex region defining a set of feasible solutions either for an original problem of interest or for a mathematical relaxation of such a problem. Hence the points y' and y" are not necessarily endpoints of LS(y',y").

We begin by adopting an approach commonly used in nonlinear line search procedures, which consists of identifying a succession of points $y(\theta_o)$, $y(\theta_1)$, …, $y(\theta_s)$ on LS(y',y") such that

$$\theta_o < \theta_1 < \ldots < \theta_s, \text{ where } \theta_o = \theta_{min}, \ \theta_s = \theta_{max} \text{ and } s \geq 2. \qquad (1.2)$$

A straightforward way to generate the θ_h values is to subdivide the interval $[\theta_{min}, \theta_{max}]$ into s equal subintervals so that

$$\theta_h = \theta_{h-1} + \Delta \ (= \theta_o + h\Delta) \text{ for } h = 1, \ldots, s, \text{ where } \Delta = (\theta_{max} - \theta_{min})/s \quad . \qquad (1.3)$$

We have represented the division of the line segment as above to match with notation used in certain methods for global function optimization. Other ways of subdividing the line segment besides (1.3) can be used if prior knowledge about the form of f(y) is available. In certain contexts, for example, it is useful to partition the range of θ values between θ_{max} and θ_{min} into a collection of subintervals, each of which is then subdivided in the manner of (1.3) but for different values of s.

We are interested in sequences of points on the subdivided line defined by reference to the θ values that have either the form of a pair $(y(\theta_{h-1}), y(\theta_h))$ for $1 \leq h \leq s$ or a triple $(y(\theta_{h-1}), y(\theta_h), y(\theta_{h+1}))$ for $1 \leq h \leq s - 1$. (Hence the points $y(\theta_o)$ and $y(\theta_s)$ can be endpoints of intervals defined by such pairs and triples.)

The algorithms at the focus of this note begin with a sequence of parameter values θ, and undertake to find a solution on the line segment (1.1) that is better than any of the solutions $y(\theta_h)$ identified by the initial division of the line segment. Our methods are related to binary search methods (such as interval bisection) and also to direct search methods (such as the Golden Section method), but in contrast to these approaches require neither that f(y) be differentiable nor unimodal.

Some of the methods of this note make use of a sorting algorithm to identify some number of best (smallest f(y) value) points from collections that begin with the line segment LS(y',y"). For this purpose, it is possible to use a standard method such as quicksort or quickselect (see, e.g., Martínez and Roura, 2001 or Knuth, 1997), but if the number of points to be selected is relatively small, for example 32 or fewer, then unpublished experiments (Tseng and Glover, 1998) have shown that a combination of simple insertion and binary "hardwired sort" algorithms can work as well or better. These simple and easily coded sorting approaches are given in Appendix 1 (with slight improvements) for the interested reader.

The rest of this note is organized as follows. In Section 2 we introduce an approach called the 3-2-3 Algorithm that operates by selecting subintervals defined by reference to triples. An abbreviated version of this approach provides a configuration of points on LS(y',y") that can also

be used to launch the other line search algorithms proposed, producing successive values of θ_h for subdividing the interval $[\theta_{min}, \theta_{max}]$ that are not necessarily equally spaced. Section 3 then describes an Advanced 3-2-3 Algorithm which is designed to perform a more elaborate search by investing additional computational effort. Section 4 considers subintervals defined by reference to pairs, and introduces a Stratified Split Algorithm that refines ideas of the Advanced 3-2-3 Algorithm to handle all selected pairs simultaneously, while adaptively varying the scope and computational effort of the search. Section 5 then presents a Nested Interval Method which utilizes a different way of organizing the search which is highly convenient to implement. Finally, Section 6 briefly summarizes conclusions and potential applications of the algorithms.

2. The 3-2-3 Line Search Method

The 3-2-3 procedure starts from an initial construction similar to that used by Golden Section methods for unimodal function optimization on a line, consisting of triples $y(\theta_{h-1})$, $y(\theta_h)$, $y(\theta_{h+1})$ but allowing more than one starting interval and more complex ways for operating on the intervals considered.

This approach can be particularly appropriate when the distances between successive points $y(\theta_h)$ are relatively small, or when the goal is to refine a coarse search of the line by focusing more thoroughly on the region around the point $y(\theta_q)$.

We select a current instance of such a triple $y(\theta_{q-1})$, $y(\theta_q)$, $y(\theta_{q+1})$ by requiring that it satisfy

$$f(y(\theta_q)) \leq f(y(\theta_{q-1}), f(y(\theta_{q+1})). \qquad (2.1)$$

A triple satisfying (2.1) will be called a *fertile triple*, and the point $y(\theta_q)$ is called the *anchor point* of the triple.

The 3-2-3 Algorithm is organized to operate on each fertile triple for which $f(y(\theta_q))$ does not exceed the globally minimum $f(y(\theta))$ value over the points $y(\theta_1)$, …, $y(\theta_s))$ by more than a relatively small amount. The treatment of a fertile triple $y(\theta_{q-1})$, $y(\theta_q)$, $y(\theta_{q+1})$ consists of "splitting" each of the two intervals $[\theta_{h-1}, \theta_h]$ for $h = q$ and $h = q + 1$ by defining $\theta = .5(\theta_{h-1} + \theta_h)$, thus producing a point $y(\theta)$ for each of these two intervals that lies halfway between $y(\theta_{h-1})$ and $y(\theta_h)$, i.e., $y(\theta) = .5(y(\theta_{h-1}) + y(\theta_h))$. Because $y(\theta)$ can be represented in this latter form, we also refer to the operation of splitting the interval $[\theta_{h-1}, \theta_h]$ as that splitting the associated "vector interval" $[y(\theta_{h-1}), y(\theta_h)]$.

Define an *anchor point sequence* to be a maximal sequence of (one or more) successively adjacent points whose members all qualify as the anchor point $y(\theta_q)$ by (2.1). We are concerned

with such sequences due to the following fact. If two adjacent points $y(\theta_{h-1})$ and $y(\theta_h)$ of an anchor point sequence are both selected in the role of an anchor point $y(\theta_q)$, thus producing the two successive fertile triples $y(\theta_{h-2})$, $y(\theta_{h-1})$, $y(\theta_h)$ and $y(\theta_{h-1})$, $y(\theta_h)$ $y(\theta_{h+1})$, then this will result in splitting the interval $[y(\theta_{h-1}), y(\theta_h)]$ twice. To avoid this situation, we employ the following rule.

The Odd Skip Rule: From each anchor point sequence, create a collection of fertile triples by selecting the first and then every other (odd) point in the sequence to take the role of an anchor point $y(\theta_q)$.

The Odd Skip Rule by itself is insufficient to accomplish the splitting of all intervals $[y(\theta_{h-1}), y(\theta_h)]$ associated with fertile triples, because if the sequence contains an even number of points, then the last point $y(\theta_{q'})$ of the sequence does not get selected as an anchor point $y(\theta_q)$, and the interval $[y(\theta_{q'}), y(\theta_{q'+1})]$ is omitted from consideration. We call such an interval a *residual interval*, and handle it by employing a special design, described later, for splitting it separately.

The reference to residual intervals also has an additional important function, by making it possible for $y(\theta_o)$ and $y(\theta_s)$ (i.e., $y(\theta_{min})$ and $y(\theta_{max})$) to take the role of the anchor point $y(\theta_q)$ of a fertile triple. In particular, it is possible that the first two points $y(\theta_o)$ and $y(\theta_1)$ yield $f(y(\theta_o)) < f(y(\theta_1))$ or that the last two points $y(\theta_{s-1})$ and $y(\theta_s)$ yield $f(y(\theta_s)) < f(y(\theta_{s-1}))$, which suggests that $y(\theta_o)$ and $y(\theta_s)$ should be considered relevant when undertaking the process of splitting intervals, yet $y(\theta_o)$ and $y(\theta_s)$ in these cases do not belong to fertile sequences. (This would be a particularly grave oversight if $f(y(\theta_o))$ or $f(y(\theta_s))$ gave the smallest f(y) value of all points on the line segment.)

We handle this situation by considering $[y(\theta_o),y(\theta_1)]$ to be a residual interval if $f(y(\theta_o)) < f(y(\theta_1))$ and considering $[y(\theta_{s-1}),y(\theta_s)]$ to be a residual interval if $f(y(\theta_s)) < f(y(\theta_{s-1}))$. It is convenient to relax the "<" requirement of these classifications, under certain circumstances noted later, by instead requiring $f(y(\theta_o)) \leq f(y(\theta_1))$ and $f(y(\theta_s)) \leq f(y(\theta_{s-1}))$ in the respective cases. These latter inequalities correspond to the two "halves" of the fertile triple inequalities $f(y(\theta_q)) \leq f(y(\theta_{q+1}))$ and $f(y(\theta_q)) \leq f(y(\theta_{q-1})$ for the situation where q = 0 and q = s. (Hence, we consider (2.1) to hold by default for the case q = 0 if $f(y(\theta_q)) \leq f(y(\theta_{q+1}))$ and for the case q = s if $f(y(\theta_q)) \leq f(y(\theta_{q-1}))$.) Specifically, then, $y(\theta_o)$ and $y(\theta_s)$ are defined as anchor points if they qualify as belonging to these special residual intervals.

2.1 The Abbreviated 3-2-3 Line Search Algorithm

We begin by describing an abbreviated version of the 3-2-3 Algorithm which is a one-pass algorithm for creating an initial subdivision of the line segment L(y'y") that differs from the "equally spaced" subdivision produced by (1.3). The procedure can be used as a stand-alone

starting procedure to give a beginning division of LS(y',y") for any of the algorithms of this paper.

The basic idea underlying the Abbreviated Algorithm, which is also exploited in other versions of the 3-2-3 algorithm, is as follows. For a given fertile triple $y(\theta_{q-1})$, $y(\theta_q)$, $y(\theta_{q+1})$, let $y^a = y(\theta_{q-1})$, $y^b = y(\theta_q)$, $y^c = y(\theta_{q+1})$. Because $f(y^b) \leq f(y^a)$, $f(y^c)$, the sequence of points y^a, y^b, y^c may be viewed as a "descent configuration" (particularly if $f(y^b) < f(y^a)$ or $f(y^b)) < f(y^c))$ which suggests the potential existence of a point y^{a1} lying between y^a and y^b, or a point y^{b1} lying between y^b and y^c, such that the new point continues the descent to yield a still better objective value; i.e., $f(y^{a1}) < f(y^b)$ or $f(y^{b1}) < f(y^b)$. We make an "unbiased estimate" about where such an intermediate point y^{a1} or y^{b1} may lie by bisecting the intervals $[y^a, y^b]$ and $[y^b, y^c]$; i.e., setting $y^{a1} = .5(y^a + y^b)$ and $y^{b1} = .5(y^b + y^c)$. This bisection rule, which is used throughout the paper, can be replaced by a rule that forms convex combinations of the endpoints of $[y^a, y^b]$ and $[y^b, y^c]$ by assigning larger weights to points that have better (smaller) f(y) values.

The one-pass Abbreviated Algorithm selects an initial value Δ_o as the basic step size for subdividing the line segment LS(y',y"), for example by choosing a beginning value s_o and setting

$$\Delta_o = (\theta_{max} - \theta_{min})/s_o$$

When the Abbreviated Algorithm is used as a starting point for other algorithms, s_o can be selected to be somewhat larger (and Δ_o, as a result, somewhat smaller) than otherwise might be done. We observe that the value s_o in any event will be smaller than the value s of (1.3) because the algorithm splits some of the intervals in the process of subdividing the line segment, hence effectively using a Δ value in these instances equal to $.5\Delta_o$ or $.25\Delta_o$, etc.

We use the notation θ^a, θ^b and θ^c to denote the values of θ that identify the points $y^a = y(\theta^a)$, $y^b = y(\theta^b)$ and $y^c = y(\theta^c)$. The superscripts a, b and c are used as "markers" (symbols) to differentiate among the items referenced, while the subscripts h of θ_h are numbers linked to specific positions on the line segment LS(y',y"). By convention, then, we stipulate that $y^x = y(\theta^x)$ for x = a, b, c.

The Abbreviated Algorithm examines successive points on LS(y',y") by assigning each in turn the role of the "middle point" y^b in the sequence y^a, y^b, y^c. Then we employ the customary inequalities to check whether this middle point may qualify as an anchor point. More precisely, the points y^a, y^b, y^c are related to each other by setting $\theta^b = \theta^a + \Delta_o$ and $\theta^c = \theta^b + \Delta_o$ regardless of whether an intermediate point is generated between y^a and y^b or between y^b and y^c. This is a conservative approach to splitting intervals that allows points y^b to qualify as anchor points, and hence permits more intermediate points to be generated than might otherwise be the case. (The value h itself is incremented by the algorithm so that θ_h is the θ value for the last point generated, which may or may not correspond to the value θ^b, depending on whether an interval is split when θ^b is examined.)

We handle the treatment of residual intervals automatically by means of a logical indicator PreviousSplit that is set to *true* or *false* according to whether the interval immediately preceding the one presently examined has already been split. To handle the special residual intervals that contain the first point $y(\theta_{min})$ or the last point $y(\theta_{max})$, we organize the method so that the initial step only checks $f(y^b) \le f(y^c)$ and the final step only checks $f(y^b) \le f(y^a)$, instead of checking $f(y^b) \le f(y^a), f(y^c)$.

We give a simple "Depth-1" version of the Abbreviated Algorithm here, which only splits intervals once (without splitting other intervals inside of them), to make the principles of the method clear. More elaborate "Depth-2" and "Depth-3" versions that are generally more effective are given in Appendix 2. (The Appendices of this paper provide more detailed expositions than customary in order to remove potential ambiguities in going from more general ideas to specific implementations, and to make sure that essential features are not misinterpreted.)

Depth-1 Abbreviated 3-2-3 Algorithm

Choose Δ_o, set h = 0 and $\theta^b = \theta_o = \theta_{min}$ and $\theta^c = \theta^b + \Delta_o$.
(Initial Step)
If $f(y^b) \le f(y^c)$ then
    (Split $[y^b,y^c]$)
    $\theta_1 = \theta_o + .5\Delta_o$
    h := 1
    PreviousSplit = *true*
Else
    PreviousSplit = *false*
Endif
(Initialize θ^a, θ^b and θ^c for remaining iterations)
$\theta^a = \theta_o$
$\theta^b = \theta^a + \Delta_o$
$\theta^c = \theta^b + \Delta_o$
For h_o = 1 to $s_o - 1$
    If $f(y^b) \le f(y^a), f(y^c)$ then
        (Split the two intervals $[y^a,y^b]$ and $[y^b,y^c]$, but exclude $[y^a,y^b]$ if previously split)
        If PreviousSplit = *true* then
            (Split $[y^b,y^c]$)
            $\theta_{h+1} = \theta^b$
            $\theta_{h+2} = \theta^b + .5\Delta_o$
            h := h + 2

Else

(Split both $[y^a,y^b]$) and $[y^b,y^c]$)

$\theta_{h+1} = \theta^a + .5\Delta_o$

$\theta_{h+2} = \theta^b$

$\theta_{h+3} = \theta^b + .5\Delta_o$

h := h + 3

Endif

PreviousSplit = *true*

Else

(Don't split the intervals)

$\theta_{h+1} = \theta^b$

h :=h + 1

PreviousSplit = *false*

Endif

(Update θ^a, θ^b and θ^c for next iteration)

$\theta^a = \theta^b$

$\theta^b = \theta^c$

$\theta^c = \theta^b + \Delta_o$

Endfor

(Final Step)

If $f(y^b) \leq f(y^a)$ and PreviousSplit = *false* then

$\theta_{h+1} = \theta^a + .5\Delta_o$

$\theta_{h+2} = \theta^b$

s = h + 2

Else

$\theta_{h+1} = \theta^b$

s = h + 1

Endif

Throughout the execution of the previous algorithm, the value $f(y(\theta_h))$ for h = 0 to s is examined to determine if $y(\theta_h)$ qualifies as f* or one of the β best solutions, or as one of the β best anchor points when the Abbreviated Algorithm is used as the starting point for the Basic or Advanced 3-2-3 Algorithm .

2.2 The Basic 3-2-3 Line Search Algorithm

The Basic 3-2-3 Algorithm starts by segregating the β best anchor points found in the process of applying the initial Abbreviated Algorithm by differentiating between those that are anchor points selected by the Odd Skip Rule and those that are anchor points associated with residual

intervals. Let Triple denote the collection of fertile triples determined by the former anchor points and let Residual denote the collection of residual intervals determined by the latter anchor points; i.e.,

Triple = $\{(y^a, y^b, y^c)$: for selected fertile triples, with anchor point $y^b\}$ and
Residual = $\{(y^a,y^c)$: for selected residual intervals with anchor point $y^a\}$.

The elements of Triple are identified by $y^a = y(\theta_{q-1})$, $y^b = y(\theta_q)$, $y^c = y(\theta_{q+1})$ for the selected anchor points y^b, while the elements of Residual are identified by $y^a = y(\theta_{q'})$, $y^c = y(\theta_{q'+1})$ where $y(\theta_{q'})$ is the last point of an anchor point sequence containing an even number of points, or where $y(\theta_{q'})$ is the point $y(\theta_o)$ and $y(\theta_o)$ qualifies as an anchor point of a residual sequence. In addition, to handle the situation where $y(\theta_s)$ may be the anchor point of a residual sequence, we allow for $y^a = y(\theta_s)$ and $y^c = y(\theta_{s-1})$. It is understood that the residual intervals for $y(\theta_o)$ and $y(\theta_s)$ are not chosen to belong to Residual if these intervals already belong to one of the fertile triples in Triple.

We operate on the pairs (y^a,y^c) in Residual by means of supporting method called the 2-1-2 Algorithm. This method successively splits a chosen pair (y^a,y^c) to produce a point $y^b = .5(y^a + y^c)$, and immediately transfers to the basic 3-2-3 Algorithm as soon as y^b qualifies as an anchor point of the triple y^a, y^b, y^c (i.e., $f(y^b) \le f(y^a)$, $f(y^c)$). Until this happens, the splitting operation yields $f(y^b) > f(y^a)$ since y^a is the anchor point (hence $f(y^a) \le f(y^c)$), and we redefine y^c to be the resulting y^b, so that y^a again qualifies as the anchor point for the new pair (y^a,y^c).

Both the 2-1-2 Algorithm and the 3-2-3 Algorithm use an iteration counter, Iter, and an iteration limit, LastIter, initialized by setting Iter = 0 and setting LastIter to a selected value. (In most circumstances, LastIter can be relatively small, e.g., from 4 to 7.)

The following methods are applied by successively selecting elements from Residual and Triple, and then executing the methods as indicated. The elements may be selected in the order based on the f(y) values of their anchor points, if desired.

2-1-2 Line Search Algorithm
(Start with and maintain $f(y^a) \le f(y^c)$.)
While Iter ≤ LastIter
    Iter := Iter + 1
    $y^b = .5(y^a + y^c)$.
    If $f(y^b) \le f(y^a)$ then
        ((y^a,y^b,y^c) has the proper form for the 3-2-3 Algorithm)
        If $f(y^b) < f(y^*)$ set $y^* = y^b$.
        Terminate and Execute the 3-2-3 Algorithm

Else
($f(y^a) < f(y^b)$)
Designate (y^a,y^b) to be the new (y^a,y^c)
Endif
Endwhile

In the situation above where $f(y^a) < f(y^b)$, if $f(y^a) = f(y^c)$ then (y^c,y^b) could equally be chosen as the new (y^a,y^c).

Basic 3-2-3 Line Search Algorithm
(Start with and maintain $f(y^b) \leq f(y^a), f(y^c)$.)
While Iter ≤ LastIter
Iter := Iter + 1
$y^{a1} = .5(y^a + y^b)$.
$y^{b1} = .5(y^b + y^c)$.
If $f(y^b) \leq f(y^{a1}), f(y^{b1})$ then
Designate (y^{a1},y^b,y^{b1}) to be the new (y^a,y^b,y^c).
(The old y^a and old y^c are discarded.)
Elseif $f(y^{a1}) \leq f(y^{b1})$ then
$(f(y^{a1}) < f(y^b), f(y^a))$
Designate (y^a,y^{a1},y^b) to be the new (y^a,y^b,y^c)
(y^{b1} and the old y^c are discarded)
If $f(y^b) < f(y^*)$ set $y^* = y^b$.
Else
$(f(y^{b1}) < f(y^b), f(y^c))$
Designate (y^b,y^{b1},y^c) to be the new (y^a,y^b,y^c)
(The old y^a and y^{a1} are discarded.)
If $f(y^b) < f(y^*)$ set $y^* = y^b$.
Endif
Endwhile

The "2-1-2" Algorithm gets its name from the fact that each iteration starts with 2 points, generates 1 new point, and then discards one of the starting points to end with 2 points (one being the new point).

Similarly, the main algorithm is called the "3-2-3" Algorithm because each iteration starts with 3 points, generates 2 new points, and then discards two to end with 3 points (including at least one of the new points). Viewed from the perspective of intervals, the algorithm might be called the "2-4-2" Algorithm, because each iteration starts with the 2 adjacent intervals $[y^a,y^b]$, $[y^b,y^c]$,

expands them to produce the 4 adjacent subintervals $[y^a,y^{a1}]$, $[y^{a1},y^b]$, $[y^b,y^{b1}]$, $[y^{b1},y^c]$, and then finally shrinks the latter collection to again obtain 2 adjacent intervals.

We illustrate the 2-1-2 and the 3-2-3 Procedures in Diagrams 1 and 2, respectively, using a 2-dimensional representation.

Diagram 1A shows the starting configuration for the 2-1-2 Procedure, where f(xa) < f(xc). A line has been drawn connecting the points $f(y^a)$ and $f(y^c)$ to clarify their relationship, but the line has no role in the procedure itself. Remaining components of Diagram 1 include reference not only to y^a and y^c, but also to the point y^b and its function value $f(y^b)$.

Diagram 1B illustrates the case where $f(y^b) \leq f(y^a)$. The lines successively joining $f(y^a)$, $f(y^b)$, and $f(y^c)$ are accentuated to indicate that this configuration qualifies as a 3-2-3 configuration, and hence the 2-1-2 Procedure terminates at this point and the 3-2-3 Procedure begins.

Diagrams 1C and 1D illustrate two versions of the same case, where $f(y^a) < f(y^b)$. In both instances, the line joining $f(y^a)$ and $f(y^b)$ is accentuated, to indicate that the resulting configuration qualifies as a 2-1-2 configuration. Hence the 2-1-2 Procedure continues by reference to the accentuated portion of the diagram, and y^b becomes the new y^c. (A third instance of the case for $f(y^a) < f(y^b)$ is also possible, where in addition $f(y^b) > f(y^c)$. The treatment is the same as illustrated in Diagrams 1C and 1D.)

Diagram 2A shows the starting configuration for the 3-2-3 Procedure, where $f(y^b) \leq f(y^a)$ and $f(y^b) \leq f(y^c)$. Again the points identifying the function values are connected by a broken line for purposes of illustration. Remaining components of Diagram 2 additionally include the points y^{a1} and y^{b1}, together with their function values $f(y^{a1})$ and $f(y^{b1})$.

Diagram 2B illustrates the situation in which $f(y^b). \leq f(y^{a1})$ and $f(y^b). \leq f(y^{b1})$. In this case y^b retains its identity as the point having a smallest f(y) value, and the sequence y^{a1}, y^b, y^{b1} qualifies as a 3-2-3 configuration as indicated by accentuating the lines successively joining $f(y^{a1})$, $f(y^b)$, and $f(y^{b1})$. The next iteration of the 3-2-3 Procedure therefore resumes with the current y^{a1} becoming the new y^a and the current y^{b1} becoming the new y^c.

Diagrams 2C and 2D illustrate two versions of the case where the condition of Diagram 2B is not satisfied (hence $f(y^{a1}) < f(y^b)$ or $f(y^{b1}) < f(y^b)$), and in addition $f(y^{a1}) \leq f(y^{b1})$. Now y^{a1} qualifies to become the new y^b, and in both of the Diagrams 2C and 2D we have accentuated the broken line joining $f(y^a)$, $f(y^{a1})$ and $f(y^b)$, thus identifying the configuration that qualifies as a 3-2-3 configuration for the next iteration.

DIAGRAM 1

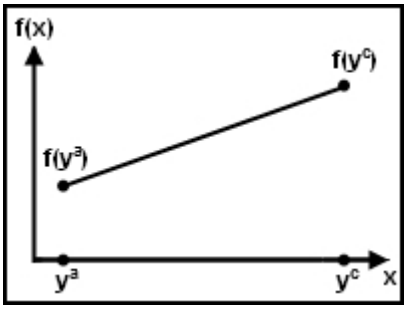

Diagram 1A

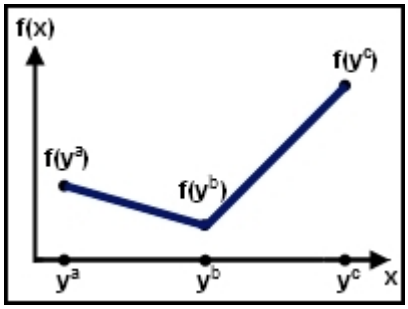

Diagram 1B

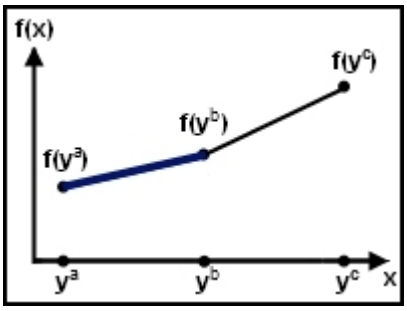

Diagram 1C

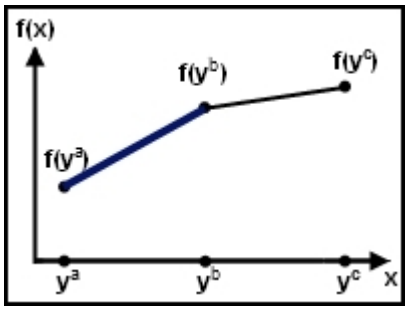

Diagram 1D

DIAGRAM 2

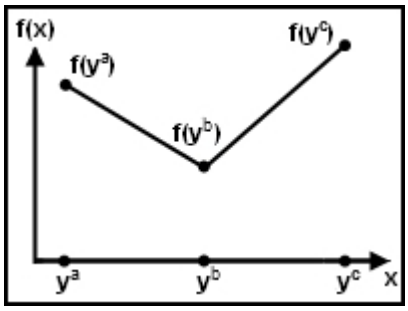

Diagram 2A

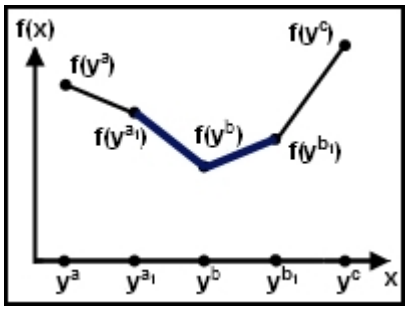

Diagram 2B

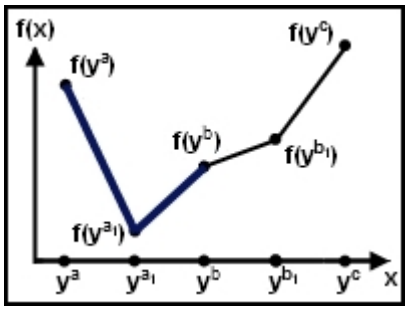

Diagram 2C

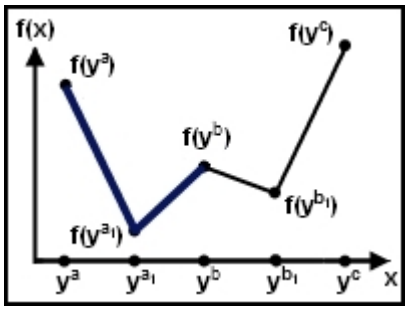

Diagram 2D

There remains the case where the conditions of both Diagram 2B and of Diagram 2C (and 2D) are all not satisfied, and hence we have $f(y^{b1}) < f(y^b)$ and $f(y^{b1}) < f(y^{a1})$. This situation is the same as the one illustrated in Diagrams 2C and 2D, with the roles of y^{b1} and y^{a1} interchanged, and hence we have not included an additional diagram to illustrate it.

The preceding diagrams disclose that this "basic" form of the 3-2-3 procedure neglects to examine potentially fruitful regions that might be explored more thoroughly – as in the case of Diagram 2D, where the triple (y^b, y^{b1}, y^c) might reasonably be considered as a candidate to become the next (y^a, y^b, y^c). The more advanced version of the 3-2-3 method that follows is designed to handle such considerations.

3. An Advanced 3-2-3 Algorithm

A more complete version of the 3-2-3 Algorithm results by allowing the method to avoid discarding members of the subintervals $[y^a,y^{a1}]$, $[y^{a1},y^b]$, $[y^b,y^{b1}]$, $[y^{b1},y^c]$ on steps where the first two and last two subintervals both appear attractive to take the role of the new $[y^a,y^b]$ and $[y^b,y^c]$ (i.e., where $f(y^{a1})$ and $f(y^{b1})$ are both relatively small). The two new intervals $[y^a,y^b]$, $[y^b,y^c]$ are then both submitted to the 3-2-3 Algorithm. Likewise, in a case where $f(y^a)$ and $f(y^b)$ are both relatively small, but $f(y^{a1})$ is larger, we retain $[y^a, y^{a1}]$ as a subinterval to be submitted to the 2-1-2 Algorithm, together with retaining the two intervals $[y^{a1},y^b]$ and $[y^b,y^{b1}]$ (as would normally be done) to submit to the 3-2-3 Algorithm. Similarly the subinterval $[y^{b1},y^c]$ is retained if $f(y^b)$ and $f(y^c)$ are relatively small but $f(y^{b1})$ is larger.

To describe this Advanced 3-2-3 Algorithm we employ the Basic 3-2-3 Algorithm a subroutine. It is convenient in the following to refer to a pair of points (y^a, y^c) satisfying the conditions of the 2-1-2 Algorithm (deriving from a beginning residual interval) as a 2-Structure, and refer to a 3-tuple of points (y^a,y^b,y^c) satisfying the conditions of the Basic 3-2-3 Algorithm (deriving from a beginning fertile triple) as a 3-Structure. The Advanced 3-2-3 Algorithm, like the Basic Algorithm, will generate an intermediate 5-tuple of points $(y^a, y^{a1},y^b,y^{b1},y^c)$, but under certain conditions will not discard various portions to shrink it back to a single 3-Structure. Instead, as intimated above, the Advanced Algorithm will sometimes generate two separate 3-Structures from the 3-tuples (y^a, y^{a1},y^b) and (y^b,y^{b1},y^c), and in other instances will generate a single 3-Structure together with possibly one or two 2-Structures.

The structures produced by the Advanced Algorithm fall into different classes. A Class A structure is a 3-Structure (y^a,y^b,y^c) for which $f(y^b) = f^*$, where f^* denotes the minimum f(y) value so far generated by the algorithm. Such a structure will automatically generate a descendant that is another Class A structure by the rules of the Basic 3-2-3 Algorithm, since this algorithm yields a 3-Structure that either improves $f(y^b)$ or leaves it unchanged (where by definition y^b is the best

point of the structure). A Class A structure can also produce a 3-Structure as a descendant that instead belongs to Class B when the two 3-tuples (y^a, y^{a1},y^b) and (y^b,y^{b1},y^c) yield values f(y^{a1}) and f(y^{b1}) that match or improve the current value f(y^b). If one of these 3-tuples produces a new f* value and the other does not, then the second becomes a 3-Structure that receives a Class B status. A Class A structure can also produce a 2-Structure belonging to Class B if the 2-Structure qualifies as "promising" according to criteria identified by the Advanced Algorithm.

A Class B structure may be determined to lack promise if the new y^b point generated does not improve upon the current best point of the structure. In this case the Class B structure is simply discarded without producing a descendant. However, if the Class B structure produces a new 3-Structure that yields a new point improving on its current best, the resulting 3 Structure survives to become a descendant. If in addition this new structure yields f(y^b) = f* (after updating f*, if appropriate) then it becomes a Class A structure. Otherwise it transitions to the next lower Class (after Class B) and becomes a Class C structure. Finally, when the algorithm processes a Class C structure, the outcome is required to produce a new y^b that yields f(y^b) = f* (hence yielding a Class A structure). Otherwise the Class C structure has no descendants.

Only a Class A structure can give rise to more than one descendant (and one or both may be a new Class A structure). A descendant that has a Class B structure will automatically have a Class B status. To handle the fact that the Advanced Algorithm can generate more than one descendant to be processed, the method uses a queue operated by a first-in first-out protocol. This protocol assures that the 2-Structures and 3-Structures will be processed according to the order in which they are generated.

The iteration counter, Iter, is only incremented when a Class A structure is produced as the descendant of the structure currently examined, to approximately match the way this counter is incremented in the Basic Algorithm. However, LastIter should be given a larger value for the Advanced Algorithm, since it will generate more Class A structures than the Basic Algorithm.

The Advanced Algorithm begins from an initial 3 Structure. If such a structure is not initially available, the 2-1-2 Algorithm is applied in order to produce it (assuming this can be done within LastIter iterations). The method then proceeds as follows, setting Begin = *true* as a signal to permit the first 3-Structure to be treated as an improved structure, making it a potential source of new 2-structures. We give the algorithm in a highly detailed form, as in the case of the Basic 3-2-3 method, in order to convey the content of ideas that are not fully visible from a high level description.

Advanced 3-2-3 Algorithm

Designate the starting 3-Structure (y^a,y^b,y^c) to be a Class A Structure, set $y^* = y^b$, $f^* = f(y^b)$, and place (y^a,y^b,y^c) as the first element on the queue.
Begin = *true*
While Iter ≤ LastIter
    Remove a structure from the start of the queue.
    If the structure is a 2-Structure (y^a,y^c) (where $f(y^a) \le f(y^c)$) then
        (the structure is automatically a Class B structure)
        $y^b = .5(y^a + y^c)$.
        If $f(y^b) < f(y^a)$ then
            (an improving 3-Structure is generated)
            If $f(y^b) \le f^*$ then
                Designate the 3-Structure (y^a,y^b,y^c) to be a Class A Structure
                Iter := Iter + 1
                If $f(y^b) < f^*$ then
                    $y^* = y^b$
                    $f^* = f(y^b)$
                Endif
            Else
                Designate the 3-Structure (y^a,y^b,y^c) to be a Class C Structure
            Endif
            Add the 3-Structure (y^a,y^b,y^c) to the end of the queue.
        Endif
        (Above, no descendant is produced to be added to the queue unless it is a
        3-Structure that improves on the current 2-Structure examined.)
.   Else
        (The structure is a 3-Structure)
        $y^{a1} = .5(y^a + y^b)$.
        $y^{b1} = .5(y^b + y^c)$.
        $f^o = \text{Min}(f(y^{a1}),f(y^b),f(y^{b1}))$
        If $f^o \le f^*$ then
            (at least one descendant of the 3-Structure will be a Class A Structure)
            Iter := Iter + 1
            If $f(y^{a1}) \le f(y^b)$ and $f(y^{b1}) \le f(y^b)$ then
                (create two new 3-Structures)
                $f^* = \text{Min}(f(y^{a1}),f(y^{b1}))$
                If $f(y^{a1}) > f^*$ then
                    Assign (y^a,y^{a1},y^b) a Class B status

Assign (y^b,y^{b1},y^c) a Class A status
$y^* = y^{b1}$
Elseif $f(y^{b1}) > f^*$ then
Assign (y^a,y^{a1},y^b) a Class A status
Assign (y^b,y^{b1},y^c) a Class B status
$y^* = y^{a1}$
Else
Assign (y^a,y^{a1},y^b) a Class A status
Assign (y^b,y^{b1},y^c) a Class A status
$y^* = y^{a1}$
Endif
Add both of the 3 structures (y^a,y^{a1},y^b) and (y^b,y^{b1},y^c) to the end of the queue, adding a Class A structure before a Class B structure
Elseif $f(y^{a1}) \leq f(y^b)$ then
Assign (y^a,y^{a1},y^b) a Class A status
Add (y^a,y^{a1},y^b) to the start of the queue
If $f(y^{a1}) < f^*$ then
$f^* = f(y^{a1})$
$y^* = y^{a1}$
Endif
Assign the 2-Structure (y^{b1},y^b) a Class B status
Add (y^{b1},y^b) to the start of the queue
Elseif $f(y^{b1}) \leq f(y^b)$ then
Assign (y^b,y^{b1},y^c) a Class A status
Add (y^b,y^{b1},y^c) to the start of the queue
If $f(y^{b1}) < f^*$ then
$f^* = f(y^{b1})$
$y^* = y^{b1}$
Endif
Assign the 2-Structure (y^{a1},y^b) a Class B status
Add (y^{a1},y^b) to the start of the queue
Else
(The basic unimproved but refined (y^{a1},y^b,y^{b1}) 3-Structure is obtained)
Assign the 3- Structure (y^{a1},y^b,y^{b1}) a Class A status
Add (y^{a1},y^b,y^{b1}) to the start of the queue
If Begin = *true* then
(This is the beginning 3-structure, hence is treated as an improved structure)
If $f(y^{a1}) < f(y^a)$ then
Assign the 2-Structure (y^{a1},y^a) a Class B status

Add (y^{a1},y^{a}) to the start of the queue
Endif
If $f(y^{b1}) < f(y^{c})$ then
Assign the 2-Structure (y^{b1},y^{c}) a Class B status
Add (y^{b1},y^{c}) to the start of the queue
Endif
Begin = *false*
Endif
Endif
Else
(No descendant of the current 3-Structure (y^{a},y^{b},y^{c}) is a Class A Structure.
If (y^{a},y^{b},y^{c}) has a Class C status it will be dropped, with no descendants.
Otherwise a single descendant will be generated as in the Basic 3-2-3
Algorithm. If (y^{a},y^{b},y^{c}) itself was originally assigned a Class A status, then its
descendant will receive a Class B status, and otherwise a Class C status.)
If (y^{a},y^{b},y^{c}) has a Class A or Class B status then
(Apply the Basic 3-2-3 algorithm)
$y^{a1} = .5(y^{a} + y^{b})$.
$y^{b1} = .5(y^{b} + y^{c})$.
If $f(y^{b}) \leq f(y^{a1}), f(y^{b1})$ then
Add the 3-Structure (y^{a1},y^{b},y^{b1}) to the end of the queue
Assign (y^{a1},y^{b},y^{b1}) a Class B status if (y^{a},y^{b},y^{c}) has a Class A
status and assign (y^{a1},y^{b},y^{b1}) a Class C status otherwise
Elseif $f(y^{a1}) \leq f(y^{b1})$ then
Add the 3-Structure (y^{a},y^{a1},y^{b}) to the end of the queue
Assign (y^{a},y^{a1},y^{b}) a Class B status if (y^{a},y^{b},y^{c}) has a Class A
status and assign (y^{a},y^{a1},y^{b}) a Class C status otherwise
Elseif $f(y^{b1}) \leq f(y^{a1})$ then
Add the 3-Structure (y^{b},y^{b1},y^{c}) to the end of the queue
Assign (y^{b},y^{b1},y^{c}) a Class B status if (y^{a},y^{b},y^{c}) has a Class A
status and assign (y^{b},y^{b1},y^{c}) a Class C status otherwise
Endif
Endif
Endif
Endif
Endwhile

In the next section we identify an algorithm that embodies some of the key ideas of the Advanced 3-2-3 Algorithm, but that is less complex in its design and yet highly flexible to execute.

4. Line Search by Stratified Splitting

Since a 3-Structure is composed of 2 adjacent pairs, the 3-2-3 Algorithm can alternatively be viewed as a set of rules for operating on such pairs, subject to implicitly taking account of the relationship between them. Building on this observation, we propose a line search that operates entirely on pairs (y^a,y^c). The method is based on a design for assigning the outcomes to certain levels, or *strata*, when a pair (y^a,y^c) is split to create descendants (y^a,y^b) and (y^b,y^c).

Before discussing this stratification process, we first describe the starting procedure for selecting the pairs (y^a,y^c) submitted to the algorithm.

Selecting the Initial Pairs (y^a,y^c)

It is convenient in the following to use shorthand notation by defining $y^h = y(\theta_h)$ and $f^h = f(y^h)$. Accompanying this, we let $H = \{0, 1, \ldots, s\}$ and $Y = \{y^h \ h \in H\}$ (identifying Y as the set of all points generated from the line segment (1.1) either by an initial equal spacing or by applying the Abbreviated 3-2-3 Algorithm). For two adjacent pairs (y^{h-1},y^h) and (y^h,y^{h+1}) determined from a selected point y^h of Y, we say that (y^{h-1},y^h) is *defined* if $h > 0$ and similarly say that (y^h,y^{h+1}) is *defined* if $h < s$. An initial set P of pairs, starting with P empty, is created from a subset of such adjacent pairs, as follows.

Sketch of the Initial Pair Generation Method.

1. Choose a value $\beta \geq 1$ as a basis for selecting some number of the points y^h from Y. (E.g., $\beta = |Y|/10$, rounded to an integer).
2. Identify a subset H_o of H, and a corresponding subset $Y_o = \{y^h: \ h \in H_o\}$ of Y, so that the elements y^h in Y_o yield the β best (smallest) f^h values over $h \in H$.
3. For each $y^h \in Y_o$, add the defined instances of the adjacent pairs (y^{h-1},y^h) and (y^h,y^{h+1}) to P.

When the Abbreviated 3-2-3 Algorithm is used to generate an initial Y, and Step 1 above uses a rule such as $\beta = |Y|/10$, the identification of β best points must be postponed until after the Abbreviated Algorithm is completed, since the number of elements in Y will not be known in advance.

To avoid adding pairs to P that duplicate others already added, we provide the following specific algorithm to carry out the steps sketched above. As indicated, we begin by selecting β and identifying $Y_o = \{y^h: \ h \in H_o\}$ containing the β best points y^h from Y. A generalization of this

approach is used again in the Nested Interval Algorithm of Section 5, where it is applied to successively different sets in the roles of Y and Y_o.

Pair Generation

$P = \varnothing$
$h' = \text{Min}(h \in H_o)$
Add the defined instances of the pairs $(y^{h'-1},y^{h'})$ and $(y^{h'},y^{h'+1})$ to P.
$h'' = h' + 1$
$h^{\#} = \text{Min}(h \in H_o: h \geq h'')$
While $h^{\#}$ exists
    $h = h^{\#}$
    Add (y^h,y^{h+1}) to P (if defined)
    If $h > h'$ add (y^{h-1},y^h) to P
    $h'' = h + 1$
    $h^{\#} = \text{Min}(h \in H_o: h \geq h'')$
EndWhile

The number of pairs in P takes a minimum value equal to $|H_o| + 1$ when H_o consists of $|H_o|$ consecutive indexes of H (and the element h = 0 or h = s belongs to H_o), and takes a maximum value equal to $2|H_o|$ when none of the indexes in H_o are consecutive. Since $|H_o| = \beta$, it follows that 1 to β of the points y belonging to pairs in P come from $Y - Y_o$. A useful feature of the Stratified Split Algorithm is its ability to work with all current pairs simultaneously, starting from the initial P, while maintaining a set of decision rules that account for the status of these pairs as a basis for determining the preferred step to take at each juncture.

The Stratification Process

The descendants (y^a,y^b) and (y^b,y^c) of (y^a,y^c), based on defining $y^b = .5(y^a + y^c)$, are stratified by classifying them as either *improving*, *promising* or *marginal*, with sub-classifications of *weakly promising* and *weakly marginal*, according to the four cases. We use a notation similar to that of $f^h = f(y^h)$, by defining $f^a = f(y^a)$, $f^b = f(y^b)$ and $f^c = f(y^c)$. However, in this case the superscripts a, b and c of y^a, y^b and y^c are simply to distinguish these points from each other, and do not refer to specific indexes such as $h \in H$. We continue to observe the convention $f^a \leq f^c$.

(C1) If $f^b < f^a$ (hence also $f^b < f^c$), then (y^a,y^b) and (y^b,y^c) are *improving*.
(C2) If $f^a \leq f^b \leq f^c$, then (y^a,y^b) is *promising*
    (a) If $f^a = f^b = f^c$, then (y^b,y^c) is *promising*.
    (b) Otherwise, (y^b,y^c) is *marginal* (a lower status than *promising*).
(C3) If $f^a < f^c < f^b$, then (y^a,y^b) is *weakly promising* and (y^b,y^c) is *weakly marginal*.
(C4) If $f^a = f^c < f^b$, then (y^a,y^b) and (y^b,y^c) are both *weakly marginal*.

The strata produced by these cases will be indexed using the notation Stratum = 1, 2, …, MaxStratum. MaxStratum will typically take the value 3, but can range from 2 to 5 as subsequently explained. (As will be seen, the strata defined by reference to (C1) through (C4) are loosely related to the Classes A, B and C of the Advanced 3-2-3 Algorithm, particularly when MaxStratum = 3.)

To begin all pairs of the initial set P are assigned to Stratum 1. Throughout the process of modifying P, discarding some pairs and adding others during the splitting process, we refer to a pair $\rho \in P$ in the conventional form $\rho = (y^a, y^c)$, and define $F(\rho) = f^a$ (hence $F(\rho) = \text{Min}(f^a, f^c)$). The best (smallest) currently known value f* of f(y) can also be expressed as $f^* = \text{Min}(F(\rho): \rho \in P)$.

The notation Stratum(ρ) denotes the stratum that a particular pair ρ belongs to. (Hence Stratum(ρ) = 1 for pairs in the initial P)[1] When a pair $\rho = (y^a, y^c)$ is split, the cases (C1) – (C4) are used to determine the strata to which the new pairs (y^a, y^b) and (y^b, y^c) are assigned. By convention, Stratum(ρ) > MaxStratum implies that ρ is to be discarded, and is no longer considered for future examination.

To state the rules for determining strata membership when $\rho = (y^a, y^c)$ is split, let ρ_o denote either of the two descendants (y^a, y^b) or (y^b, y^c) of ρ, and let δ denote a small positive threshold value to determine when $F(\rho_o)$ lies "close to" the best value f*, as established by the relationship

$$F(\rho_o) \leq f^* + \delta \qquad (4.1)$$

Finally, let S = Stratum(ρ), to identify the stratum of the parent $\rho = (y^a, y^c)$ of a descendant ρ_o under consideration. Then the rules to identify the stratum to which ρ_o belongs are as follows.

(R1) If ρ_o is improving, then
- (a) If $F(\rho_o)$ satisfies (4.1) then $\text{Stratum}(\rho_o) = 1$.
- (b) Otherwise, ρ_o is assigned to the same stratum as ρ; i.e., $\text{Stratum}(\rho_o) = S$.

(R2) If ρ_o is not improving, then ρ_o is assigned to a stratum S_o inferior to that of ρ ($S_o > S$).
- (a) If ρ_o is promising, $\text{Stratum}(\rho_o) = S + \text{Inc\_f*}$, where Inc_f* = 1 if f* changed on the preceding pass, and Inc_f* = 0 otherwise. (Inc_f* = 0 when pass = 1.)

---

[1] If we allow more variation in the F(ρ) values of initially selected pairs, then those pairs in the beginning P with larger F(ρ) values may be assigned Stratum(ρ) = 2.

(b) If ρ_o is marginal, Stratum(ρ_o) = S + 2.
(c) If ρ_o is weakly marginal, Stratum(ρ_o) = S + 3.

Note that when MaxStratum = 3, then ρ_o will be discarded in one of the following cases: S = 3 under the conditions of (R2)(a); S ≥ 2 under the conditions of (R2)(b); or S ≥ 1 under the conditions of (R2)(c). This last case implies that all weakly marginal descendants ρ_o will be discarded (since S ≥ 1 always holds) unless MaxStratum is chosen larger than 3. Such a larger choice for MaxStratum implies that fewer descendants will be discarded, and hence the algorithm will perform a more thorough search at the expense of greater computational effort.

By default, we treat the weakly promising classification the same as a promising classification for the purpose of applying the preceding rules. Under this assumption, MaxStratum will be given a value of at most 4. Considerations for choosing other values for MaxStratum are discussed in Appendix 3.

To decide whether an improving pair ρ_o should be assigned to Stratum 1 by rule (R1)(a), we postpone applying the criterion of (4.1) until all improving pairs are identified, since the value of f* may change during this identification process. The final f* that results is then used to scan the improving pairs, which are saved on an Improving List, so that these pairs may be assigned to their appropriate strata. Each entry on the Improving List is a triple $((y^a,y^b),(y^b,y^c),S)$, since by (C1) the two pairs (y^a,y^b) and (y^b,y^c) are both improving simultaneously, and we store S = Stratum(ρ) for the parent ρ of (y^a,y^b) and (y^b,y^c), in order to provide the necessary information to execute the rule (R1)(b). (We also implicitly record the value $F(\rho_o) = f^b$, which is the same for $\rho_o = (y^a,y^b)$ and $\rho_o = (y^b,y^c)$, to avoid recalculating this value when the threshold condition (4.1) is checked.)

Finally, we add a special screening provision as a counterpart to (4.1) to weed out and eliminate new pairs ρ_o that are insufficiently attractive relative to f*, thus saving computational effort by reducing the number of descendants that will be evaluated on future passes of the method. To accomplish this, we let *pass* denote the current number of passes (major iterations) of the method, and introduce a threshold $\delta(pass)$ and eliminate any pair ρ_o that fails to satisfy

$$F(\rho_o) \leq f^* + \delta(pass) \qquad (4.2)$$

The value of δ(pass) is chosen relatively large for the first two passes, to avoid eliminating early descendants that may be "slow starters," giving them a chance to produce their own descendants that may ultimately yield attractive $F(\rho_o)$ values. (The stratification of cases that underlies rules (R1) and (R2) provides its own screening even if δ(pass) is large. Note that δ(pass) is entirely independent of δ and is initialized separately.) After the first two passes, δ(pass) may be

progressively decreased so that (4.2) will exclude a greater proportion of descendants and hence yield fewer pairs to examine on future iterations.

We now state the Stratified Split algorithm as follows, continuing to maintain the convention that the pair (y^a,y^c) is ordered so that $f^a \le f^c$. The value LastPass will normally be given a moderately small value, e.g., in the range from 4 to 7. (Larger values may be used in cases where an exceedingly fine grain search is desired.)

Stratified Split Line Search Algorithm

Initialization:

Assign each initial pair to Stratum 1.
Create a Scan List equal to the initial P.
Begin with the Next Scan List empty.
Begin with the Improving List empty.
Set pass = 1 and Inc_f* = NextInc_f* = 0.

While pass ≤ LastPass

While the Scan List is not empty

Remove a pair $\rho = (y^a,y^c)$ from the Scan List.
S = Stratum(ρ)
$y^b = .5(y^a + y^c)$ (creating the two potential descendants (y^a,y^b) and (y^b,y^c))
If $f^b < f^a$ then

((y^a,y^b) and (y^b,y^c) are both improving)
Add the triple $((y^a,y^b),(y^b,y^c),S)$ to the Improving List.
Also add (y^a,y^b) and (y^b,y^c) to the Next Scan List
If $f^b < f^*$ then

$y^* = y^b$
$f^* = f^b$
NextInc_f* = 1

Endif

Else

(Neither (y^a,y^b) nor (y^b,y^c) is improving)
If $f^a \le f^b \le f^c$, then

$((y^a,y^b)$ is *promising*)
Stratum(y^a,y^b) = S + Inc_f*
If $f^a = f^c$ then

$((y^b,y^c)$ is *promising*)
Stratum(y^b,y^c) = S + Inc_f*

Else

$((y^b,y^c)$ is *marginal*)
Stratum(y^b,y^c) = S + 2

Endif
Elseif $f^a < f^b < f^c$, then
((y^a,y^b) is *weakly promising* and (y^b,y^c) is *weakly marginal*)
Stratum(y^a,y^b) = S + Inc_f*
Stratum(y^b,y^c) = S + 3
Else (If $f^a = f^c < f^b$ then)
((y^a,y^b) and (y^b,y^c) are both *weakly marginal*)
Stratum(y^a,y^b) = S + 3
Stratum = (y^b,y^c) = S + 3
Endif
Add $\rho_o = (y^a,y^b)$ to the Next Scan List if Stratum$(\rho_o) \leq$ MaxStratum and
$F(\rho_o) \leq f^* + \delta$(pass)
Add $\rho_o = (y^b,y^c)$ to the Next Scan List if Stratum$(\rho_o) \leq$ MaxStratum and
$F(\rho_o) \leq f^* + \delta$(pass)
Endif
Endwhile
(The Scan List is now Empty.)
While the Improving List is not empty
Remove an element $((y^a,y^b),(y^b,y^c),S)$ from the Improving List.
If $F(\rho_o) \leq f^* + \delta$ then
Stratum(y^a,y^b) = Stratum(y^b,y^c) = 1
Else
Stratum(y^a,y^b) = Stratum(y^b,y^c) = S
Endif
EndWhile
pass := pass + 1
If pass ≤ LastPass then
Interchange the names of the Next Scan List and the Scan List
(equivalently, place the contents of the Next Scan List in the Scan List, leaving the Next Scan List empty.)
Inc_f* = NextInc_f*
NextInc_f* = 0
Endif
EndWhile

By the structure of the foregoing algorithm, the Next Scan List will always have at least one element on each pass. Relevant data structures for implementing the algorithm are described in Appendix 3.

5. The Nested Interval Algorithm

The Nested Interval Algorithm departs from the two preceding algorithms by dispensing with rules to differentiate among different classes of solutions. Only very simple data structures are needed to provide an effective implementation.

The method treats the beginning collection of points $y^o, y^1, \ldots, y^s$ as a discrete interval $[y^o, y^1, \ldots, y^s]$ which is progressively modified by generating new intervals to lie within it. The generic structure that launches each iteration is a current *domain* D that consists of a collection of discrete intervals which share no points in common. .

Apart from the value LastIter that limits the total number of iterations, the algorithm has just a single decision parameter β. (To be more precise, the rule for subdividing the original line segment to produce $[y^o, y^1, \ldots, y^s]$ introduces an additional parameter, which is either the value s or the value s_o of the Abbreviated Algorithm.) Within the Nested Interval Algorithm, β refers to the number of best points drawn from the current D (independent of the intervals containing these points). We denote the subset of D containing these selected points by D(β).

For convenience in the present context, we refer to a solution y^h by representing it as y(h), and accordingly represent the solutions in the current partitioned domain D by y(h), h = 0, …,s. The value s as well as the identity of the points y(h) in D will vary from iteration to iteration..

Denote the intervals into which D is partitioned by D_i = [y(h1(i)), …., y(h2(i))], for , i = 1,…, d, and hence h1(i) and h2(i) are respectively the first and last h indexes of the points y(h) of D_i (yielding h1(1) = 0 and h2(d) = s). Identify the interval D_i for i = i(h) that contains a given point y(h) in D. Then the two points y(h-1) and y(h+1) are called "interval adjacent" (or IntAdjacent) to y(h) in D if y(h-1) and y(h+1) also belong to D_i, hence we exclude y(h-1) from being IntAdjacent to y(h) if h = h1(i), and exclude y(h+1) from being IntAdjacent to y(h) if h = h2(i). By extension, for any subset S of D, we define IntAdjacent(S) = {y ∈ D: y is IntAdjacent to some y' ∈ S}. We are particularly interested in the set InAdjacent(D(β)) and in the union of this set with the points of D(β), which we denote by $D_o(\beta)$ = D(β) ∪IntAdjacent(D(β)). The set $D_o(\beta)$ is foundation for building the new intervals for the next iteration of the Nested Interval Algorithm. We refer to the maximal intervals of IntAdjacent points within D_o as *source intervals*. The completed intervals that produce the next domain D are given by expanding each source interval by splitting the subinterval between each pair of its IntAdjacent points, thus adding these midpoints to the source interval to yield the completed discrete interval for the new D.

To illustrate these ideas, suppose the first interval within the domain D consists of the points y(0) to y(9), and the points of D(β) that lie within this first interval consist of y(2), y(4), y(5) and y(9).

We provide a visual schematic to support our illustration, where "best" points (belonging to D(β)) are denoted by "B" and other points are denoted by "•", as follows.

```
[•    •    B    •    B    B    •    •    •    B]
 0    1    2    3    4    5    6    7    8    9
```

The points of $D_o(\beta) = D(\beta) \cup \text{IntAdjacent}(D(\beta))$ are shown next by placing the symbol # above them. We also have placed brackets ([ ]) around the points that are the maximal intervals of the IntAdjacent points within D_o, thus identifying the source intervals for the next domain.

```
          #    #    #    #    #    #         #    #
•    [•   B    •    B    B   •]   •    [•   B]
0    1    2    3    4    5    6    7    8    9
```

To produce the next domain, we expand each source interval of D shown above by splitting each of its successive pairs y(h-1), y(h) to produce the point .5(y(h-1) + y(h)), and insert this new point between its parents within the source interval. For the current illustration we denote each of these new points by "n", and show the completed intervals for the next domain by

```
          #    #    #    #    #    #         #    #
•    [•  n  B  n  •  n  B  n  B  n  •]   •    [•  n  B]
0    1    2    3    4    5    6    7    8    9
```

Points of the original D that do not lie in these completed intervals are dropped, and the points y(h) that remain are re-indexed to become y(0), …, y(s) so that the new domain D may be represented in exactly the same form as the previous one. Likewise, we will again denote the intervals into which D is partitioned by D_i, i = 1, …, d (where s and d, as well as the intervals D_i, are new). The indexing for the points of the two newly created intervals of the present illustration is as shown below.

```
          #    #    #    #    #    #         #    #
•    [•  n  B  n  •  n  B  n  B  n  •]   •    [•  n  B]
     0  1  2  3  4  5  6  7  8  9  10        11 12 13
```

We can readily identify bounds on the size NumSource which we define to be the number of points in the source intervals that are used to generate the new domain produced by the foregoing operations (hence the number of points represented by # above). Each point y(h) in D(β) (denoted by "B" above) will in general be accompanied by at most 2 IntAdjacent points to produce a source interval in D. Consequently, the total number of points contained in all the source intervals is at most 3β, yielding NumSource ≤ 3β. The minimum number is just β, arising in the case where D(β) lies in a single interval of D and includes all the points of this interval, hence bounding NumSource by $\beta \leq \text{NumSource} \leq 3\beta$.

It is also of interest to identify bounds on the value NumAdd which we define to be the number of new points added to the source intervals to produce the completed intervals for the new D (the points denoted by "n" above). We note that NumAdd equals the number of pairs (y(h-1), y(h)) and (y(h), y(h+1)) that are produced from the IntAdjacent points in the source intervals. The number of such pairs is at most 2β, disclosing that NumAdd is likewise at most 2β. The smallest number of such pairs (and hence new points) is β.

Putting these results together, we can identify bounds on the size |D'| of the new domain D' produced by the algorithm. In particular, |D'| = NumSource + NumAdd, and hence we conclude $2\beta \leq |D'| \leq 5\beta$. (In the preceding example, the 4 points of D(β) produced source intervals containing a total of 8 points, which resulted in adding 6 new points to contribute a total of 14 points to the new D, a number that lies midway between the bounds of 2x4 = 8 and 5x4 = 20.points. The value of β in this example may of course be somewhat larger than 4, since we are only considering the instances of D(β) that lie in the first interval.)

By reference to these ideas, we now state the Nested Interval Algorithm using the following notation and conventions. For each h = 0, …, s, let EndInterval(h) = *true* if y(h) is the last element (end) of some interval D_i (hence h = h2(i) for some i) and EndInterval(h) = *false* otherwise. (The use of the EndInterval(h) array is more convenient for stating the algorithm than explicitly accessing the values h1(i) and h2(i).)

Accompanying this we use a "logical indicator" StartInterval which is assigned the value StartInterval = *true* each time the next y(h) to be examined starts a new interval, and StartInterval = *false* otherwise. (Hence we perform the initialization StartInterval = *true* before examining the first point y(0) of the beginning interval.)

To identify the "best elements" of D contained in D(β), let Best(h) = *true* if $y(h) \in D(\beta)$ and Best(h) = *false* otherwise. We denote two IntAdjacent points y(h-1) and y(h) of D to be *D(β)-Adjacent* if y(h-1) and y(h) belong to $D_o(\beta) = D(\beta) \cup \text{IntAdjacent}(D(\beta))$. It follows that y(h-1) and y(h) are D(β)-Adjacent if "Best(h-1) = *true* or Best(h) = *true*" excluding the situation where StartInterval = *true* (to appropriately avoid referring to the pair (y(h-1), y(h)) when y(h) is the first element of an interval). Consequently, we can identify all elements of $D_o(\beta)$ by identifying the pairs for which the condition "Best(h-1) = *true* or Best(h) = *true*" holds in conjunction with StartInterval = *false*.

The new solutions to be generated for the next domain D will be denoted by $y_o(h_o)$, $h_o = 1, \ldots, s_o$. These solutions will include the solutions of $D_o(\beta) = D(\beta) \cup \text{IntAdjacent}(D(\beta))$ as well as the new solutions generated from the source intervals of D. Finally, we use a logical indicator PreMember, which is set to *true* whenever the current y(h) satisfies "Best(h-1) = *true* or Best(h)

= *true*" and is set to *false* otherwise. Then when examining the next y(h) (i.e., y(h+1)), if PreMember = *true* and if this current y(h) (likewise) does not satisfy "Best(h-1) = *true* or Best(h) = *true*", we know that the associated next y(h-1) (the previous y(h)) is a point that will be dropped from the domain D to produce the next domain. With these ingredients, we now give the details of the Nested Interval Algorithm.

Nested Interval Algorithm

The initial domain D contains the single interval [y(0), y(1), … y(s)].
EndInterval(h) = *false* for h = 0, …, s – 1, and EndInterval(s) = *true*.
y* = arg min(f(y): y ∈ D) and f* = f(y*).
For Iter = 1 to LastIter
    Choose β for the current iteration and identify the set D(β). For each h = 0, …, s. set
        Best(h) = *true* if y(h) ∈ D(β) and Best(h) = *false* otherwise.
    StartInterval = *true*
    h_o = - 1
    (Next generate the new domain D in the form $y_o(h_o)$, h_o = 0, …, s_o)
    For h = 1, …, s
        If StartInterval = *true* then
            StartInterval = *false*
            PreMember = *false*
        Elseif Best(h-1) = *true* or Best(h) = *true* then
            (y(h-1) and y(h) qualify as members of $D_o(\beta)$ recorded as the new $y_o(h_o+1)$ and $y_o(h_o+3)$, and we insert their midpoint as the new $y_o(h_o+2)$)
            $y_o(h_o+1)$ = y(h-1)
            $y_o(h_o+2)$ = y = .5(y(h-1) + y(h))
            $y_o(h_o+3)$ = y(h)
            If f(y) < f* then
                y* = y
                f* = f(y*)
            Endif
            $EndInterval_o(h_o+1)$ = *false*
            $EndInterval_o(h_o+2)$ = *false*
            $EndInterval_o(h_o+3)$ = *false*
            $h_o = h_o + 3$
            PreMember = *true*
        Else
            If PreMember = *false* then
                Drop y(h-1)
            Endif

PreMember = *false*
Endif
If EndInterval(h) = true then
StartInterval = *true*
$EndInterval_o(h_o)$ = *true*
Endif
Endfor (h = 1, …, s)
$s = h_o$
$EndInterval(h) = EndInterval_o(h)$, h = 1 to s
$y(h) = y_o(h)$, h = 0 to s
Endfor (Iter = 1 to LastIter)

The preceding method can use a straightforward data structure to avoid the expense of transferring solutions back and forth between vectors y(h) and $y_o(h_o)$ and at the same time avoid expanding the array space required for recording new solutions. Let IndexPool denote a set of currently available indexes h for storing solutions y(h). The size of IndexPool can be given by |IndexPool| = 2β* + Max(s*+1,5β*), where s* is the original s value for the beginning set of points y(h), h = 0 to s, and β* is the largest β value selected by the algorithm. (The method will start by removing the first s*+ 1 indexes from IndexPool to build the first domain D = [y(0), …, y(s*)].)

An index q is removed from IndexPool each time a new point y(q) is generated to be added to D (involving at most 2β* points) and an index h is added back to IndexPool each time a point y(h) is dropped from D. By this means, each solution created will be put in a location y(q) that does not change as long as the solution remains in D.

To exploit IndexPool, we use an array ID(h) (h = 1 to s) that names the index q = ID(h) such that the point y(h) of the preceding algorithmic description actually represents the solution y(q). A parallel array, $ID_o(h_o)$, h_o = 1, …, s_o keeps track of the "new ID(h)" array that will be used on the next iteration.

The two instructions $y_o(h_o+1) = y(h-1)$ and $y_o(h_o+3) = y(h+1)$ of the algorithm thus become $ID_o(h_o+1) = ID(h-1)$ and $ID_o(h_o+3) = ID(h)$, while the instruction $y_o(h_o+2) = .y$ is handled by identifying y = .5(y(q') + y(q")) where q' = ID(h-1) and q" = ID(h). Then a new index q is selected from IndexPool and we set y(q) = y, followed by $ID_o(h_o + 2) = q$.

The instruction "Drop y(h-1)" becomes "Add q' to the IndexPool" for q' = ID(h-1). The value Best(h) has the interpretation "Best(h) = *true*" if and only if the point y(q) for q = ID(h) is a member of the set D(β). Likewise, the value EndInterval(h) has the interpretation EndInterval(h) = *true* if and only if y(q) is the last solution of an interval. Nothing needs to be done to maintain

this interpretation of EndInterval(h). The algorithm as stated will assign EndInterval(h) the correct value at each iteration. Finally, the last update of the algorithm, "$y(h) = y_o(h)$, h = 0 to s," is replaced by "$ID(h) = ID_o(h)$, h = 0 to s."

Given the bounds on the size of the new domain D' given by $2\beta \leq |D'| \leq 5\beta$, and more particularly the bounds on the number of new points that are added given by $\beta \leq NumAdd \leq 2\beta$, the computation required by the Nested Interval Algorithm at each iteration can clearly be controlled by means of the parameter β. Total computation can be controlled by additionally selecting the value s* that determines the size of the first domain D, and by selecting the value of LastIter that determines the number of iterations. (The value LastIter can be indirectly determined by imposing a limit on the total number of points generated, though in any event LastIter will usually be selected to be at most 7.)

Several elements of the algorithm can be viewed as a simplification of analogous ideas and processes of the Stratified Split Algorithm, suggesting that the computation of the Stratified Split Algorithm can likewise be susceptible to effective control (by the procedures recommended in the Appendix 3). The increased simplicity of the Nested Interval Algorithm, however, invites computational experimentation with strategies for manipulating its primary parameter β in conjunction with choosing the initial s* value.

6. Concluding Remarks

The 3-2-3, Stratified Split and Nested Interval algorithms have a natural use as subroutines for metaheuristic methods. A prominent application of this type arises in the setting of population-based metaheuristics where the methods can be used to find good local minima over various line segments in order to yield candidates for new population members. For example, the scatter search method is particularly suited to benefit from such a process, because it characteristically uses linear combinations to generate new solutions, hence automatically producing line segments that can be submitted to global line searches. In turn, the best solutions obtained by these line search algorithms provide a source of potential new population members.

The 3-2-3, Stratified Split and Nested Interval algorithms can also be applied in the situation where f(y) is an approximation to another function whose evaluations are expensive to calculate or imperfectly known, as occurs in the setting of simulation optimization. In general, the algorithms can be used within a variety of more advanced solution processes for global minimization over a bounded search space.

Yet another application of these line search algorithms is that of finding global optima for multi-dimensional functions. This can be accomplished by successively optimizing over different

coordinates of the y vector and using re-starting, an approach successfully used for continuous nonlinear optimization in Gardeux et al. (2009). More recently, a simplified instance of the 3-2-3 algorithm has been embedded in such a strategy in Gardeux et al. (2010) with highly promising results. A related approach for the fixed-mix problem in finance is employed in Glover, Mulvey and Hoyland (1996) which simultaneously changes the values of two variables at a time. In this latter case the variables are first scaled and an increase is one variable is matched by a decrease in another, thus effectively inducing a unidimensional change within a transformed coordinate system. The efficacy of this strategy in the context of the fixed-mix problem motivates a more dedicated examination of transformed coordinate systems in the context of multimodal line search, where such systems may be used to supplement approaches that restrict attention to manipulating variables in the original coordinate system. Issues involved in creating useful transformed coordinate systems are discussed in Appendix 4.

Each of the line search algorithms of this paper has its own particular strengths, and the settings where each proves most effective will likely vary. For ease of implementation, the "Basic" version of the 3-2-3 Algorithm is the simplest of the methods, though the solutions it generates will undoubtedly be dominated by those produced by the Advanced 3-2-3 Algorithm. The logic that underlies the Advanced 3-2-3 Algorithm is harder to penetrate, however, since a good portion of its rationale is embedded in the instructions of the algorithm itself.

The Stratified Split Algorithm has more visible foundations than the Advanced 3-2-3 Algorithm, and its principles may give a useful perspective to facilitate an understanding of the 3-2-3 Algorithm. From a computational standpoint, the increased flexibility of the Stratified Split Algorithm will likely translate into greater robustness, with a consequent reduced variability in solution quality and computation time. Offsetting this, the implementation of the Stratified Split Algorithm is more subtle than that of the Advanced 3-2-3 Algorithm, though again in compensation the data structures and supporting considerations described in the Appendix may be expected to remove potential obstacles that may otherwise be posed by this subtlety.

Finally, the Nested Interval Algorithm employs a somewhat simpler strategic foundation than the other methods, though in certain respects the method exhibits some underlying resemblances to the Stratified Split approach. The Nested Interval Algorithm shares the useful feature of working with all pairs simultaneously, and its organization may facilitate the discovery of good choice strategies, as in rules for manipulating the value of its primary parameter β (e.g., by a schedule for decreasing β as the number of iterations increases). In general, the simplicity of the Nested Interval Algorithm suggests that it may appropriately serve as a standard for measuring the computational performance of the other algorithms.

Acknowledgments

Thanks go to Akbar Karimi, who has provided valuable proofreading help and suggestions, and to Vincent Gardeux, who produced the figures used in Diagrams 1 and 2.

References

V. Gardeux, R. Chelouah, R. Siarry and F. Glover (2009) "Unidimensional Search for Solving Continuous High-Dimensional Optimization Problems," *Intelligent Systems Design and Applications. ISDA '09 Ninth International Conference*, pp, 1096-1011.

V. Gardeux, R. Chelouah, R. Siarry and F. Glover (2010) "EM323 : A Line Search based algorithm for solving high-dimensional continuous non-linear optimization problems," working paper, EISTI Institute, Cergy-Pontoise, France.

Glover, F., J. Mulvey and K. Hoyland (1996) "Solving Dynamic Stochastic Control Problems in Finance Using Tabu Search with Variable Scaling," in *Meta-Heuristics: Theory & Applications*, I.H. Osman & J.P. Kelly, eds., pp. 429-448, Kluwer Academic Publishers.

Himmelblau, D. (1972) *Applied Nonlinear Programming*, McGraw Hill.

Knuth, D. (1997) *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley.

Lasdon, L. (2002) *Optimization Theory for Large Systems*, Dover Publications, Inc.

Mangasarian, O. (1969) *Nonlinear Programming*, McGraw Hill.

Martínez, C. and S. Roura (2001) "Optimal sampling strategies in quicksort and quickselect*,"* *SIAM Journal of Computing* 31(3), pp. 683-705.

Murty, K. (1988) *Linear Complementarity, Linear and Nonlinear Programming*, Helderman Verlag.

Nocedal, J. and S.J. Wright (1999). *Numerical Optimization.* Springer

Tseng, F. and F. Glover (1998) "A Study on Algorithms for Selecting r Best Elements from an Array," Research Report, University of Colorado, Boulder.

Appendix 1: Hardwired Threshold Sorting

The goal is to identify β "best" (smallest) values v(i) and their associated indexes i from a collection given by v(i), i = 1, …, n. We assume that β is relatively small, e.g., ≤ 32.

The hardwired threshold sorting procedures we describe will both identify an appropriate set of β indexes that give the smallest v(i) values and will also order these indexes by means of an auxiliary indexing i(k) so that $v(i(1)) \le v(i(2)) \le \ldots \le v(i(\beta))$. The resulting process can of course also be used to identify a set of indexes giving the n – β largest v(i) values (since this set will consist of those not included in the set for the β smallest v(i) values), but the indexes giving the n – β largest v(i) values will not be in sequenced order. (This implies that we can find a set of

indexes for the β smallest v(i) values when β is relatively large, e.g., at least n – 32. For this we replace the v(i) values by v'(i) = - v(i) and replace β by β' = n – β. The result will find an index set for the β smallest v(i) values, though not in sorted order.)

Hardwired threshold sorting is based on a binary sort procedure combined with an insertion sort procedure (where the latter is used to complete the sorting algorithm when β is not close to a power of 2). These methods are designed so that the variable indexes i(1), i(2), …, i(β) are treated as "hardwired constants" i1, i2, …, iβ. Likewise, each of the levels of the sorting algorithms are also hardwired, instead of being given a variable representation (i.e., using an iterative framework). The resulting computer code has more instructions than a more customary type of code, but the procedure executes somewhat more rapidly. Empirical tests show that when these hardwired codes are given the form indicated below, they can be appreciably faster than the more customarily used sorting codes such as quicksort and heapsort, which are reputed to be the most effective alternatives. In addition, the simplicity of the hardwired binary sort and insertion sort algorithms makes the task of embedding them in software relatively easy. The proper form of these codes will become apparent by means of a few basic illustrations, which also show some alternatives.

To save space in the following descriptions, we use the notation p → q (where p > q) as shorthand for the hardwired version of the sequence of assignments i(p) = i(p-1); i(p-1) = i(p-2), … ,i(q+1) = q. Hence, 8 → 4 is shorthand for the sequence of assignments i8 = i7, i7 = i6, i6 = i5, i5 = i4..

1.1 Hardwired Binary Sorting

We first depict the form of the hardwired binary sorting procedure. It is important in the following that none of the "≤" inequalities be replaced by strict "<" inequalities, though it is permissible to replace any of the "<" inequalities by "≤" inequalities (causing the method to run slightly slower in some situations). Specifically, "<" is only permissible for the branch statement "If v(i) < Threshold then", and for the branch statements that are used to identify a single element out of 2 remaining posibilities. The value of Threshold in the code is always the value v(iβ), as set at the conclusion of each major branch sequence.

Hardwired Binary Sort for β = 8.

```
(Initialization) Set i1 = … = i8 = n + 1 (a dummy value) and Threshold = v(n+1) = ∞.
 For i = 1 to n
        If v(i) < Threshold then
                (i1, i2, …, i8 remain to be checked to identify the precise location of v(i))
                If v(i) ≤ v(i4) then
                        (i1,i2,i3,i4 remain to be checked)
                        8 → 4
                        If v(i) ≤ v(i2) then
                                (i1,i2 remain)
                                4 → 2
```

```
                    If v(i) < v(i1) then
                        (i1 uniquely remains)
                        2 → 1
                        i1 = i
                    Else
                        (i2 uniquely remains)
                        i2 = i
                    Endif
                Elseif v(i) < v(i3) then
                    (i3, i4 remained, but now i3 uniquely remains)
                    4 → 3
                    i3 = i
                Else
                    (i4 uniquely remains)
                    i4 = i
                Endif
            Elseif v(i) ≤ v6 then
                (i5, i6, i7, i8 remained, but now i5,i6 remain)
                8 → 6
                If v(i) < v(i5) then
                    (5 uniquely remains)
                    6 → 5
                    i5 = i
                Else
                    (i6 uniquely remains)
                    i6 = i
                Endif
            Elseif v(i) < v(i7) then
                (i7,i8 remained but now i7 uniquely remains)
                    8 → 7
                    i7 = i
                Else
                    (i8 remains)
                    i8 = i
                Endif
            Endif
            Threshold = v(i8)
        Endif
Endfor
```

If it is desired only to find the best 6 v(i) values, the foregoing code can be used by initializing i1 = i2 = n + 2, and v(n+2) = – ∞. Then i1 and i2 will retain their initially assigned value n + 2, while v(i3) to v(i8) will identify the 6 best v(i) values. The code can of course be made faster in this circumstance by dropping the branch "If v(i) ≤ v(i2) then" and replacing it by the "Elseif" branch that follows, changing "Elseif" to "If" (thus yielding "If v(i) < v(i3) then"). For a β value

as small as 6, however, it can be preferable to use the Hardwired Insertion Sort shown in Section 1.2.

To see how to extend the preceding code to larger β values, it suffices to consider the hardwired binary sort for β = 16. This code contains the instructions for the β = 8 case within it. The "first tier" of instructions for the β = 16 code are shown below.

Hardwired Binary Sort for β = 16 (Partial)

```
(Initialization) Set i1 = i2 = … = i16 = n + 1 and Threshold = v(n+1) = ∞.
For i = 1 to n
        If v(i) < Threshold then
                (i1, i2, …, i16 remain to be checked to determine the precise location of
                v(i))
                If v(i) ≤ v(i8) then
                        (i1,.., i8 remain to be checked)
                        16 → 8
                        If v(i) ≤ v(i4) then
                                (i1,i2, i3, i4 remain)
                                8 → 4
                                 If v(i) ≤ v(i2) then
                                        (i1,i2 remain)
                                        4 → 2
                                        If v(i) < v(i1) then
                                                (i1 uniquely remains)
                                                2 → 1
                                                i1 = i
                                        Else
                                                (i2 uniquely remains)
                                                i2 = i
                                        Endif
                                Elseif v(i) < v(i3) then
                                        (i3, i4 remained, but now i3 uniquely remains)
                                        4 → 3
                                        i3 = i
                                Else
                                        (i4 uniquely remains)
                                        i4 = i
                                Endif
```

The pattern for remaining instructions is evident.

An alternative code is possible for binary sorting that is likely to be faster than the preceding code for some compilers (unless the v(i) values are arranged so that larger values appear first). We illustrate this alternative code for β = 8. The structure of the code permits a more compact representation by putting associated instructions on the same line, separated by a semicolon. (It

may be noted that this alternative code uses a few more instructions to re-sequence the indexes, but does not entail more computation.)

Alternative Hardwired Binary Sort for β = 8

```
(Initialization) Set i1 = i2 = … = i8 = n + 1 and Threshold = v(n+1) = ∞.
 For i = 1 to n
        If v(i) < Threshold then
                (i1, i2, …, i8 remain to be checked to determine the precise location of
                v(i))
                If v(i) > v(i4) then
                        (i5 to i8 remain to be checked)
                        If v(i) > v(i6) then
                                (i7, i8 remain)
                                 If v(i) > v(i7) then
                                        (i8 uniquely remains)
                                        i8 = i
                                Else
                                        (i7 uniquely remains)
                                        8 → 7;  i7 = i
                                Endif
                        Elseif v(i) > v(i5) then
                                (i5, i6 remained, but now i6 uniquely remains)
                                8 → 6;  i6 = i
                        Else
                                (i5 uniquely remains)
                                8 → 5;  i5 = i
                        Endif
                Elseif v(i) > v(i2) then
                        (i1 to i4 remained, but now i3,i4 remain)
                        8 → 4
                        If v(i) > v(i3) then
                                (i4 uniquely remains)
                                i4 = i
                        Else
                                (i3 uniquely remains)
                                4 → 3;  i3 = i
                        Endif
                Elseif v(i) > v(i1) then
                        (i1, i2 remained but now i2 uniquely remains)
                                8 → 2;  i2 = i
                        Else
                                (i1 remains)
                                8 → 1;  i1 = i
                        Endif
                Endif
```

```
            Threshold = v(i8)
        Endif
Endfor
```

1.2 Hardwired Insertion Sorting

We consider the form of the Hardwired Insertion Sort (sometimes also called the Bubble Sort), which we illustrate for $\beta = 6$.

Hardwired Insertion Sort for $\beta = 6$.

```
(Initialization) Set i1 = i2 = … = i6 = n + 1 and Threshold = v(n+1) = ∞.
 For i = 1 to n
        If v(i) < Threshold then
                (i1, i2, …, i6 remain to be checked to determine the precise location of
                v(i))
                If v(i) > v(i5) then
                        i6 = i
                Elseif v(i) > v(i4) then
                        6 → 5;  i5 = i
                Elseif v(i) > v(i3) then
                        6 → 4;  i4 = i
                Elseif v(i) > v(i2) then
                        6 → 3;  i3 = i
                Elseif v(i) > v(i1) then
                        6 → 2;  i2 = i
                Else
                        6 → 1; i1 = i
                Endif
                Threshold = v(i6)
        Endif
Endfor
```

The number of instructions used to re-assigning index values can be reduced in the preceding code by organizing the portion within the loop as follows:

```
        If v(i) < Threshold then
                (i1, i2, …, i6 remain to be checked to determine the precise location of
                v(i))
                If v(i) > v(i5) then
                        i6 = i
                Else
                        6 → 5;
                        If v(i) > v(i4) then
                                i5 = i
```

Else
5 → 4
(Etc.)

This nested structure is probably not as fast to execute as the earlier structure using the successive "Elseif" statements.

1.3 Combined Hardwired Insertion and Binary Sorting Methods

The Hardwired Insertion Sort can be positioned before the Hardwired Binary Sort to provide a fast method for values of β that are not powers of 2. For example, if β = 11 then a Hardwired Insertion Sort for v(i11), v(i10) and v(i9) can be joined with a Hardwired Binary Sort for β = 8 by means of the following design.

```
If v(i) < Threshold then
        (i1, i2, …, i11 remain to be checked )
        If v(i) > v(i10) then
                I11 = i
        Elseif v(i) > v(i9) then
                11 → 10;  i10 = i
        Elseif v(i) > v(i8) then
                11 → 9;  i9 = i
        Else
                11 → 9
                Insert the "If … then" block for the Hardwired Binary Sort for β = 8.
        Endif
        Threshold = v(i11)
Endif
```

The Alternative Hardwired Binary Sort can also be inserted in this procedure.

Appendix 2: Depth-2 and Depth-3 Versions of the Abbreviated 3-2-3 Line Search Algorithm

The Depth-2 and Depth-3 versions of the Abbreviated Algorithm split additional intervals during the pass of subdividing the line segment LS(y',y"). In the Depth-2 version, whenever a given split produces a point y(θ) whose value f(y(θ)) is better than that of both of its adjacent points on the line segment, then each of the two subintervals between y and its adjacent points are further split. To determine when the condition holds for carrying out this additional splitting, it is only necessary to check whether $f(y(\theta)) \leq f(y^b)$, since the original split only occurs if $f(y^b) \leq f(y^a)$, $f(y^c)$. The Depth-3 version goes further by seeing whether the intervals produced by additional splitting qualify to be split as well. Parenthetical comments within the algorithms provide additional explanation of the steps performed. As before, the values $f(y(\theta_h))$ for h = 0 to s are examined as the θ_h values are generated to determine whether $y(\theta_h)$ qualifies as f* or as one of the β best solutions. The choice between implementing the Depth-2 or the Depth-3 Algorithm depends on the empirically determined relationship between the ideal size of s_o for a given depth, and the quality of solutions produced for the computational effort expended. The greater complexity of the Depth-3 Algorithm does not necessarily translate into a significant difference in the amount of computation time consumed.

Depth-2 Abbreviated 3-2-3 Algorithm

Choose Δ_o, set h = 0 and $\theta^b = \theta_o = \theta_{min}$ and $\theta^c = \theta^b + \Delta_o$.
(Initial Step)
If $f(y^b) \leq f(y^c)$ then
    (Split $[y^b,y^c]$)
    $\theta = \theta^b + .5\Delta_o$
    If $f(y(\theta)) \leq f(y^b)$ then
        (Additionally split $[y^b,y(\theta)]$ and $[y(\theta),y^c]$)
        $\theta_1 = \theta^b + .25\Delta_o$
        $\theta_2 = \theta$
        $\theta_3 = \theta + .25\Delta_o$
        h = 3
    Else
        $\theta_1 = \theta$
        h := 1
    Endif
    PreviousSplit = *true*
Else
    PreviousSplit = *false*

Endif
$\theta^a = \theta_o$
$\theta^b = \theta^a + \Delta_o$
$\theta^c = \theta^b + \Delta_o$
For $h_o = 1$ to $s_o - 1$
    If $f(y^b) \le f(y^a), f(y^c)$ then
        (Split $[y^a,y^b]$ and $[y^b,y^c]$, but exclude $[y^a,y^b]$ if previously split)
        If PreviousSplit = *true* then
            (Split $[y^b,y^c]$)
            $\theta_{h+1} = \theta^b$
            $\theta = \theta^b + .5\Delta_o$
            If $f(y(\theta)) \le f(y^b)$ then
                (Additionally split $[y^b,y(\theta)]$ and $[y(\theta),y^c]$)
                $\theta_{h+2} = \theta^b + .25\Delta_o$
                $\theta_{h+3} = \theta$
                $\theta_{h+4} = \theta + .25\Delta_o$
                h = h+ 4
            Else
                $\theta_{h+2} = \theta$
                h := h + 2
            Endif
        Else
            (First split $[y^a,y^b]$)
            $\theta = \theta^a + .5\Delta_o$
            If $f(y(\theta)) \le f(y^b)$ then
                (Additionally split $[y^a,y(\theta)]$ and $[y(\theta),y^b]$)
                $\theta_{h+1} = \theta^a + .25\Delta_o$
                $\theta_{h+2} = \theta$
                $\theta_{h+3} = \theta + .25\Delta_o$
                h := h+ 3
            Else
                $\theta_{h+1} = \theta$
                h := h + 1
            Endif
            (Next split $[y^b,y^c]$)
            $\theta_{h+1} = \theta^b$
            $\theta_h = \theta^b + .5\Delta_o$
            If $f(y(\theta)) \le f(y^b)$ then
                (Additionally split $[y^b,y(\theta)]$ and $[y(\theta),y^c]$)
                $\theta_{h+2} = \theta^b + .25\Delta_o$

$\theta_{h+3} = \theta$

$\theta_{h+4} = \theta + .25\Delta_o$

h = h+ 4

Else

$\theta_{h+2} = \theta$

h := h + 2

Endif

Endif

PreviousSplit = *true*

Else

(Don't split the intervals)

$\theta_{h+1} = \theta^b$

h :=h + 1

PreviousSplit = *false*

Endif

$\theta^a = \theta^b$

$\theta^b = \theta^c$

$\theta^c = \theta^b + \Delta_o$

Endfor

(Final Step)

If $f(y^b) \leq f(y^a)$ and PreviousSplit = *false* then

$\theta_{h+1} = \theta^a + .5\Delta_o$

(Split $[y^a,y^b]$)

$\theta = \theta^a + .5\Delta_o$

If $f(y(\theta)) \leq f(y^b)$ then

(Additionally split $[y^a,y(\theta)]$ and $[y(\theta),y^b]$)

$\theta_{h+1} = \theta^a + .25\Delta_o$

$\theta_{h+2} = \theta$

$\theta_{h+3} = \theta + .25\Delta_o$

$\theta_{h+4} = \theta^b$ ($= \theta_{max}$)

s = h+ 4

Else

$\theta_{h+1} = \theta$

$\theta_{h+2} = \theta^b$ ($= \theta_{max}$)

s = h + 2

Endif

Else

$\theta_{h+1} = \theta^b$

s = h + 1

Endif

The Depth-3 Algorithm begins similarly, but employs a slightly different organization in order to efficiently determine the merit of splitting intervals at a greater depth.

Depth-3 Abbreviated 3-2-3 Algorithm

Choose Δ_o, set h = 0 and $\theta^b = \theta_o = \theta_{min}$ and $\theta^c = \theta^b + \Delta_o$.
(Initial Step)
If $f(y^b) \leq f(y^c)$ then
    (Split $[y^b,y^c]$)
    $\theta = \theta^b + .5\Delta_o$
    If $f(y(\theta)) \leq f(y^b)$ then
        (Additionally split $[y^b,y(\theta)]$ and $[y(\theta),y^c]$)
        $\theta' = \theta^b + .25\Delta_o$
        $\theta'' = \theta + .25\Delta_o$
        If $f(y(\theta'))$, $f(y(\theta'')) \leq f(y(\theta))$ then
            (Doubly split $[y^b,y(\theta)]$ and $[y(\theta),y^c]$)
            $\theta_1 = \theta^b + .125\Delta_o$
            $\theta_2 = \theta'$
            $\theta_3 = \theta' + .125\Delta_o$
            $\theta_4 = \theta$
            $\theta_5 = \theta + .125\Delta_o$
            $\theta_6 = \theta''$
            $\theta_7 = \theta'' + .125\Delta_o$
            h = 7
        Elseif $f(y(\theta')) \leq f(y(\theta))$ then
            (Doubly split $[y^b,y(\theta)]$ and split $[y(\theta),y^c]$)
            $\theta_1 = \theta^b + .125\Delta_o$
            $\theta_2 = \theta'$
            $\theta_3 = \theta' + .125\Delta_o$
            $\theta_4 = \theta$
            $\theta_5 = \theta + .25\Delta_o$
            h = 5
        Elseif $f(y(\theta'')) \leq f(y(\theta))$ then
            (Split $[y^b,y(\theta)]$ and doubly split $[y(\theta),y^c]$)
            $\theta_1 = \theta'$
            $\theta_2 = \theta$
            $\theta_3 = \theta + .125\Delta_o$
            $\theta_4 = \theta''$
            $\theta_5 = \theta'' + .125\Delta_o$

```
                    h = 5
                Else
                    (Only split [y^b,y(θ)] and [y(θ),y^c])
                    θ_1 = θ'
                    θ_2 = θ
                    θ_3 = θ"
                    h = 3
                Endif
            Else
                θ_1 = θ
                h := 1
            Endif
            PreviousSplit = true
Else
            PreviousSplit = false
Endif
θ^a = θ_o
θ^b = θ^a + Δ_o
θ^c = θ^b + Δ_o
For h_o = 1 to s_o – 1
            If f(y^b) ≤ f(y^a), f(y^c) then
                (Split [y^a,y^b] and [y^b,y^c], but exclude [y^a,y^b] if previously split )
                If PreviousSplit = true then
                    (Split [y^b,y^c])
                    θ_{h+1} = θ^b
                    θ = θ^b + .5Δ_o
                    If f(y(θ)) ≤ f(y^b) then
                        (Additionally split [y^b,y(θ)] and [y(θ),y^c])
                        θ' = θ^b + .25Δ_o
                        θ" = θ + .25Δ_o
                        If  f(y(θ')), f(y(θ")) ≤ f(y(θ))  then
                            (Doubly split [y^b,y(θ)] and [y(θ),y^c])
                            θ_{h+2} = θ^b + .125Δ_o
                            θ_{h+3} = θ'
                            θ_{h+4} = θ' + .125Δ_o
                            θ_{h+5} = θ
                            θ_{h+6} = θ + .125Δ_o
                            θ_{h+7} = θ"
                            θ_{h+8} = θ" + .125Δ_o
                            h = h+8
```

Elseif $f(y(\theta')) \leq f(y(\theta))$ then

(Doubly split $[y^b, y(\theta)]$ and split $[y(\theta), y^c]$)

$\theta_{h+2} = \theta^b + .125\Delta_o$

$\theta_{h+3} = \theta'$

$\theta_{h+4} = \theta' + .125\Delta_o$

$\theta_{h+5} = \theta$

$\theta_{h+6} = \theta + .25\Delta_o$

$h = h + 6$

Elseif $f(y(\theta'')) \leq f(y(\theta))$ then

(Split $[y^b, y(\theta)]$ and doubly split $[y(\theta), y^c]$)

$\theta_{h+2} = \theta'$

$\theta_{h+3} = \theta$

$\theta_{h+4} = \theta + .125\Delta_o$

$\theta_{h+5} = \theta''$

$\theta_{h+6} = \theta'' + .125\Delta_o$

$h = h+6$

Else

(Only split $[y^b, y(\theta)]$ and $[y(\theta), y^c]$)

$\theta_{h+2} = \theta'$

$\theta_{h+3} = \theta$

$\theta_{h+4} = \theta''$

$h = h + 4$

Endif

Else

$\theta_{h+2} = \theta$

$h := h + 2$

Endif

Else

(First split $[y^a, y^b]$)

$\theta = \theta^a + .5\Delta_o$

If $f(y(\theta)) \leq f(y^b)$ then

(Additionally split $[y^a, y(\theta)]$ and $[y(\theta), y^b]$)

$\theta' = \theta^a + .25\Delta_o$

$\theta'' = \theta + .25\Delta_o$

If $f(y(\theta'))$, $f(y(\theta'')) \leq f(y(\theta))$ then

(Doubly split $[y^a, y(\theta)]$ and $[y(\theta), y^b]$)

$\theta_{h+1} = \theta^a + .125\Delta_o$

$\theta_{h+2} = \theta'$

$\theta_{h+3} = \theta' + .125\Delta_o$

$\theta_{h+4} = \theta$

$\theta_{h+5} = \theta + .125\Delta_o$
$\theta_{h+6} = \theta''$
$\theta_{h+7} = \theta'' + .125\Delta_o$
$h = h + 7$

Elseif $f(y(\theta')) \leq f(y(\theta))$ then

(Doubly split $[y^a, y(\theta)]$ and split $[y(\theta), y^b]$)
$\theta_{h+1} = \theta^a + .125\Delta_o$
$\theta_{h+2} = \theta'$
$\theta_{h+3} = \theta' + .125\Delta_o$
$\theta_{h+4} = \theta$
$\theta_{h+5} = \theta + .25\Delta_o$
$h = h + 5$

Elseif $f(y(\theta'')) \leq f(y(\theta))$ then

(Split $[y^a, y(\theta)]$ and doubly split $[y(\theta), y^b]$)
$\theta_{h+1} = \theta'$
$\theta_{h+2} = \theta$
$\theta_{h+3} = \theta + .125\Delta_o$
$\theta_{h+4} = \theta''$
$\theta_{h+5} = \theta'' + .125\Delta_o$
$h = h + 5$

Else

(Only split $[y^a, y(\theta)]$ and $[y(\theta), y^b]$)
$\theta_{h+1} = \theta'$
$\theta_{h+2} = \theta$
$\theta_{h+3} = \theta''$
$h = h + 3$

Endif

Else

$\theta_{h+1} = \theta$
$h := h + 1$

Endif

(Next split $[y^b, y^c]$)
$\theta_{h+1} = \theta^b$
$\theta = \theta^b + .5\Delta_o$

If $f(y(\theta)) \leq f(y^b)$ then

(Additionally split $[y^b, y(\theta)]$ and $[y(\theta), y^c]$)
$\theta' = \theta^b + .25\Delta_o$
$\theta'' = \theta + .25\Delta_o$

If $f(y(\theta'))$, $f(y(\theta'')) \leq f(y(\theta))$ then

(Doubly split $[y^b, y(\theta)]$ and $[y(\theta), y^c]$)

$\theta_{h+2} = \theta^b + .125\Delta_o$
$\theta_{h+3} = \theta'$
$\theta_{h+4} = \theta' + .125\Delta_o$
$\theta_{h+5} = \theta$
$\theta_{h+6} = \theta + .125\Delta_o$
$\theta_{h+7} = \theta''$
$\theta_{h+8} = \theta'' + .125\Delta_o$
h = h + 8

Elseif $f(y(\theta')) \leq f(y(\theta))$ then

(Doubly split $[y^b, y(\theta)]$ and split $[y(\theta), y^c]$)
$\theta_{h+2} = \theta^b + .125\Delta_o$
$\theta_{h+3} = \theta'$
$\theta_{h+4} = \theta' + .125\Delta_o$
$\theta_{h+5} = \theta$
$\theta_{h+6} = \theta + .25\Delta_o$
h = h + 6

Elseif $f(y(\theta'')) \leq f(y(\theta))$ then

(Split $[y^b, y(\theta)]$ and doubly split $[y(\theta), y^c]$)
$\theta_{h+2} = \theta'$
$\theta_{h+3} = \theta$
$\theta_{h+4} = \theta + .125\Delta_o$
$\theta_{h+5} = \theta''$
$\theta_{h+6} = \theta'' + .125\Delta_o$
h = h + 6

Else

(Only split $[y^b, y(\theta)]$ and $[y(\theta), y^c]$)
$\theta_{h+2} = \theta'$
$\theta_{h+3} = \theta$
$\theta_{h+4} = \theta''$
h = h + 4

Endif

Else

$\theta_{h+2} = \theta$
h := h + 2

Endif

Endif

PreviousSplit = *true*

Else

(Don't split the intervals)
$\theta_{h+1} = \theta^b$

h :=h + 1

PreviousSplit = *false*

Endif

$\theta^a = \theta^b$

$\theta^b = \theta^c$

$\theta^c = \theta^b + \Delta_o$

Endfor

(Final Step)

If $f(y^b) \leq f(y^a)$ and PreviousSplit = *false* then

$\theta_{h+1} = \theta^a + .5\Delta_o$

(Split $[y^a, y^b]$)

$\theta = \theta^a + .5\Delta_o$

If $f(y(\theta)) \leq f(y^b)$ then

(Additionally split $[y^a, y(\theta)]$ and $[y(\theta), y^b]$)

$\theta' = \theta^a + .25\Delta_o$

$\theta'' = \theta + .25\Delta_o$

If $f(y(\theta'))$, $f(y(\theta'')) \leq f(y(\theta))$ then

(Doubly split $[y^a, y(\theta)]$ and $[y(\theta), y^b]$)

$\theta_{h+1} = \theta^a + .125\Delta_o$

$\theta_{h+2} = \theta'$

$\theta_{h+3} = \theta' + .125\Delta_o$

$\theta_{h+4} = \theta$

$\theta_{h+5} = \theta + .125\Delta_o$

$\theta_{h+6} = \theta''$

$\theta_{h+7} = \theta'' + .125\Delta_o$

$\theta_{h+8} = \theta^b$

s = h + 8

Elseif $f(y(\theta')) \leq f(y(\theta))$ then

(Doubly split $[y^a, y(\theta)]$ and split $[y(\theta), y^b]$)

$\theta_{h+1} = \theta^a + .125\Delta_o$

$\theta_{h+2} = \theta'$

$\theta_{h+3} = \theta' + .125\Delta_o$

$\theta_{h+4} = \theta$

$\theta_{h+5} = \theta + .25\Delta_o$

$\theta_{h+6} = \theta^b$

s = h+ 6

Elseif $f(y(\theta'')) \leq f(y(\theta))$ then

(Split $[y^a, y(\theta)]$ and doubly split $[y(\theta), y^b]$)

$\theta_{h+1} = \theta'$

$\theta_{h+2} = \theta$

$\theta_{h+3} = \theta + .125\Delta_o$
$\theta_{h+4} = \theta$”
$\theta_{h+5} = \theta$” $+ .125\Delta_o$
$\theta_{h+6} = \theta^b$
s = h+ 6

Else

(Only split $[y^a, y(\theta)]$ and $[y(\theta), y^b]$)
$\theta_{h+1} = \theta$’
$\theta_{h+2} = \theta$
$\theta_{h+3} = \theta$”
$\theta_{h+4} = \theta^b$
s = h+ 4

Else

$\theta_{h+1} = \theta$
$\theta_{h+2} = \theta^b$
s = h + 2

Endif

Else

$\theta_{h+1} = \theta^b$
s = h + 1

Endif

Throughout the execution of the previous algorithms, the value $f(y(\theta_h))$ for h = 0 to s is examined to determine if $y(\theta_h)$ qualifies as f* or one of the β best solutions.

Appendix 3: Implementation Considerations and Data Structures for the Stratified Split Algorithm

1. Implementation Aspects

We first discuss implementation aspects, listed by category, and then discuss relevant data structures.

Determining the δ(pass) values.

As discussed in Section 4, δ(pass) should receive a large value for pass = 1 and 2. For pass ≥ 3, smaller δ(pass) values can usefully be determined in conjunction with imposing an upper bound UB on the number of pairs to be examined. For example, δ(pass) could be given the smallest value that would allow UB pairs to avoid being eliminated. Once UB pairs have succeeded in satisfying $F(\rho_o) \leq f^* + \delta(\text{pass})$, then the screening can stop, and all further pairs can be discarded on the current pass.

This rule can be applied without assigning δ(pass) a "smallest acceptable value" for admitting UB pairs, and can also be implemented by an aggressive determination of δ(pass) that permits fewer than UB pairs to survive the screening. Once such a rule is applied on a given pass, the resulting value of δ(pass) can be used as a starting point for determining the value of δ(pass+1), since appropriate values on successive passes will probably be fairly similar. (For example, a first guess for δ(pass+1) can be δ(pass) + (f*(pass) – f*(pass-1)), where f*(pass) denotes the value of f* on the specified pass.)

Selecting MaxStratum.

As previously noted, MaxStratum is chosen by default to be either 3 or 4 (and may typically be set to 3). However, MaxStratum can be made as large as 5 for a still more thorough (and expensive) search if the weakly promising moves are placed in their own special category. This can be done by expanding (R2)(a) to specify $\text{Stratum}(\rho_o) = S + 2$ when ρ_o is weakly promising, accompanied by increasing the amount added to S by 1 in (R2)(b) and (c). In reverse, to have the algorithm perform a less thorough but a faster search, we can retain the preceding rule structure but stipulate that MaxStratum = 2, and then all marginal and weakly marginal pairs will always be discarded by (R2)(b) and (R2)(c).

The computation required by the algorithm can also be reduced by replacing the value MaxStratum for the first few passes by a smaller value ReducedMax (e.g., ReducedMax = MaxStratum – 1), and adding a pair ρ_o to the Next Scan List only if Stratum(ρ_o) ≤ ReducedMax. Otherwise, if ReducedMax < Stratum(ρ_o) ≤ MaxStratum, and ρ_o satisfies (4.2) the pair is placed on a Reserve List. Finally, when the time comes to eliminate reference to Reduced Max (making it the same as MaxStratum), the contents of the Reserve List can be added to those of the Next Scan List. (The criterion (4.2) is not reapplied at this point, even though f* may have changed, in order to give descendants of elements on the Reserve List the same chance to be generated that they would have had if the pairs on the Reserve List had been added to the Next Scan List earlier.)

2. Useful Data Structures

It is useful to record and access solution pairs (y^a,y^c) on the Scan List by using an array ScanList(i), i = 1 to LastScan to identify the locations where the information about these pairs reside.

Specifically, for each i = 1 to LastScan, the "pair index" p = ScanList(i) is used to give information about the solutions y^a and y^c by means of two "solution location indexes" s1 and s2, where s1 = SolLocation(p,1) and s2 = SolLocation(p,2).

These solution location indexes identify the component solutions of (y^a,y^c) by storing y^a and y^b as the 2-dimensional n vectors y(s1,j) and y(s2,j) where y_j^a = y(s1,j) and y_j^c = y(s2,j) for j = 1 to n. Accompanying this, we store the function value f(y) for y = y(s1,j) and y = y(s1,j), respectively, as f(s1) and f(s2). Hence, f(s1) ≤ f(s2).by the convention $f^a \le f^c$.

Viewed from the standpoint of solutions rather than pairs on the Scan List, a given solution y has a solution location index s so that y is stored as y(s,j) for j = 1 to n. The same solution y may be an endpoint of more than one pair, i.e., y may be either y^a or y^c in more than one pair (y^a,y^c). The solution y itself will be recorded only once, but its solution location index s will be accessed by all (1 or 2) indexes i such that ScanList(i) gives the pair location index p = ScanList(i), and where s is accessed as s = s1 = SolLocation(p,1) or as s = s2 = SolLocation(p,2).

As the previous discussion indicates, a given solution y with solution location index s is stored as y(s,j), and the function value f(y) is stored as f(s).

For each solution y = y(s,j), we also keep track of the pairs that contain y = y(s,j) as an endpoint. (i.e., the pairs that y lies "in") by means of the records InPair(s,1) and InPair(s,2) (noting that y can lie in either one or two pairs). Then p = InPair(s,1) and p = InPair(s,2) name the (up to) two

pair location indexes such that s = SolLocation(p,1) or s = SolLocation(p,2). (It doesn't matter which of the positions 1 or 2 of SolLocation identifies the solution location index s. All that matters is the value of the pair location index p.) In case y = y(s,j) lies in just a single pair, then by convention we let InPair(s,2) = -1 (since no pair location index can be -1). We identify how this InPair array is used later.

Each pair (y^a,y^c) has a Stratum value Stratum(y^a,y^c). If the pair is identified by the pair location index p (= ScanList(i) for some i) then Stratum(y^a,y^c) is recorded as Stratum(p). (In other words, the pair location index p accesses all the relevant information about the pair.)

The fact that solutions and pairs can be added and deleted means that we want to put new solutions and new pairs in locations previously occupied by old ones, in order to keep the memory space from growing. For this purpose we use arrays AvailablePairIndex(h), h = 1 to MaxAvailPair and AvailableSolIndex(h), h = 1 to MaxAvailSol, to store the indexes p = AvailablePairIndex(h) and s = AvailableSolIndex(h) for pair location indexes and solution location indexes that are available to store new pairs and solutions.

It should be noted that the indexes i = 1 to LastScan of the array ScanList(i) are arbitrary. It doesn't matter which i names the pair location index p that identifies a particular pair by p = ScanList(i). The operation of examining the pairs on the Scan List corresponds to examining the indexes i from 1 to LastScan on the ScanList(i) array, and accessing the associated pair location index p = ScanList(i). The process of shrinking the Scan List by removing elements from it is implicitly achieved simply by looking at the indexes i from 1 to LastScan, since upon examining i = LastScan there are no more elements of the Scan List to consider.

When a pair with the pair location index p* is deleted, we perform the operation

MaxAvailPair = MaxAvailPair + 1 (O1)
AvailablePairIndex(MaxAvailPair) = p*

Likewise, when a solution y is deleted, where y is accessed as y(s*,j), we perform the operation

MaxAvailSol = MaxAvailSol + 1 (O2)
AvailableSolIndex(MaxAvailSol) = s*

Note that deleting a pair (using (O1)) may or may not result in deleting a solution that is one of its endpoints. The endpoint solutions for the pair stored in p* are given by s1* = SolLocation(p*,1) and s2* = SolLocation(p*,2). This is where we make use of the InPair array. For s = s1 and s = s2, we check whether InPair(s,2) = -1. If so, then the endpoint solution stored as y(s,j) lies in only one pair (which is the one being deleted, and hence we know by implication

that InPair(s,2) = p*). Consequently, we know that the solution stored as y(s,j) must be deleted too, and we can denote the index for this solution by s* to perform the operation (O2).

If InPair(s,2) > 0, however, then we check to see whether InPair(s,2) = p*. If so, we set InPair(s,2) = - 1. If not, we know InPair(s,1) = p*, and hence we first set InPair(s,1) = InPair(s,2), followed by setting InPair(s,2) = s*.

Executing the Splitting Operation.

We now consider the operation of examining and splitting a particular pair (y^a,y^c) to create the two new pairs (y^a,y^b) and (y^b,y^c). We will repeat some of the foregoing observations for clarity. The examination of (y^a,y^c) corresponds to examining a current index i* and accessing the pair location index p* = ScanList(i*). Then p* gives the information that identifies the component solutions y^a and y^c of (y^a,y^c). Specifically, for s1* = SolLocation(p*,1) and s2* = SolLocation(p*,2), if f(s1*) ≤ f(s2*) then y^a corresponds to y(s1*,j) and y^c corresponds to y(s2*,j), and otherwise y^a corresponds to y(s2*,j) and y^c corresponds to y(s1*,j). (It makes no difference which solution is treated as y^a or y^c if f(s1*) = f(s2*).)

From the recorded solutions y(s1*,j) and y(s2*,j) we form y^b by setting

y_j^b = .5(y(s1*,j) + y(s2*,j)) for j = 1 to n. (O3)
At the same time we identify $f^b = f(y^b)$.

The operation of generating (y^a,y^b) and (y^b,y^c) by splitting (y^a,y^c) automatically results in deleting (y^a,y^c). However, if one of the pairs (y^a,y^b) and (y^b,y^c) survives to be added to the Next Scan List, then we will replace (y^a,y^c) by putting one of these surviving pairs in the location previously allotted to (y^a,y^c).

As in (O3), we denote the solutions that identify the current y^a and y^c respectively as y(s1*,j) and y(s2*,j) (for s1* = SolLocation(p*,1) and s2* = SolLocation(p*,2)).

There are four cases to consider.

Case 1.(y^a,y^b) survives to be added to the Next Scan List but .(y^b,y^c) does not.

We first check whether the current solution y^c = y(s2*,j) will no longer be an endpoint of a pair solution pair as a result of dropping the pair (y^a,y^c) and failing to add (y^b,y^c).

If InPair(s2*,2) = -1, we know that the solution y(s2*,j) lies in the single pair whose index p is given by p = InPair(s2*,1) (and hence by implication, p = p*). Hence we perform the following operation.

```
If InPair(s2*,2) = -1 then                                          (O4)
        (drop the solution y(s2*,j))
        MaxAvailSol = MaxAvailSol + 1
        AvailableSolIndex(MaxAvailSol) = s2*
Elseif p* = InPair(s2*,2) then
        (InPair(s2*,1) is appropriate and doesn't need to be changed)
        InPair(s2*,2) = -1
Else
        InPair(s2*,1) = InPair(s2*,2)
        InPair(s2*,2) = -1
Endif
```

Next we add the new solution y^b to the record of current solutions. For this we obtain a location s* for y^b from the AvailableSolIndex array, and then record y^b as the solution y(s*,j), by the operation

```
s* = AvailableSolIndex(MaxAvailSol)                                 (O5)
MaxAvailSol = MaxAvailSol – 1
y(s*,j) = y_j^b for j = 1 to n
f(s*) = f^b
```

In fact, the vector y(s*,j) in (O5) can be created directly from the formula for generating y_j^b in (O3) without going through the step of generating and storing intermediate y_j^b values for j = 1 to n. In the special case where y(s2*,j) was dropped in (O4) the new s* is just the same as the old s2*. (It isn't worth the trouble to try to check in advance for this, since the update of AvailableSolIndex requires a trivial amount of computation.)

Having thus stored y^b as the solution y(s*,j), we now check to find out which of f^a and f^b is smaller, in order to determine which of y^a and y^b will be stored as the new y^a and which will be stored as the new y^c. The appropriate operation is as follows, where p* now changes from referring to the current pair (y^a,y^c) (which is dropped) to refer to the new pair (y^a,y^c) created from the current (y^a,y^b).

```
If f(s1*) ≤ f(s*) then                                              (O6)
        (The current y^a = y(s1*,j) becomes the new y^a, and y(s*,j) becomes the new y^c.).
        (SolLocation(p*,1) = s1* is already true and doesn't need to be changed)
```

SolLocation(p*,2) = s*
Else
(y^b = y(s*,j) becomes the new y^a. Record the new s1* and s2* values.)
SolLocation(p*,1) = s*
SolLocation(p*,2) = s1*
Endif
InPair(s*,1) = p* and InPair(s*,2) = - 1.
(the – 1 is because p* is the only pair containing s*, since (y^b,y^c) is not added.).
LastNextScan = LastNextScan + 1
NextScan(LastNextScan) = p* (thus adding the pair p*, which now identifies the new (y^a,y^c), to the Next Scan List)

Case 2.(y^b,y^c) survives to be added to the Next Scan List but (y^a,y^b) does not.

We first check whether the current solution y^a = y(s1*,j) will no longer be an endpoint of a pair solution pair as a result of dropping the pair (y^a,y^c) and failing to add (y^a,y^b).

If InPair(s2*,2) = -1, we know that the solution y(s2*,j) lies in the single pair whose index p is given by p = InPair(s2*,1) (and hence by implication, p = p*). Hence we perform the following operation, which is essentially the same as (O4), except that s2* is replaced with s1*

If InPair(s1*,2) = -1 then (O4a)
(drop the solution y(s1*,j))
MaxAvailSol = MaxAvailSol + 1
AvailableSolIndex(MaxAvailSol) = s1*
Elseif p* = InPair(s1*,2) then
(InPair(s1*,1) is appropriate and doesn't need to be changed)
InPair(s1*,2) = -1
Else
InPair(s1*,1) = InPair(s1*,2)
InPair(s1*,2) = -1
Endif

Next we add the new solution y^b to the record of current solutions. For this we perform operation (O5) as in Case 1.

We now check to find out which of f^b and f^c is smaller, in order to determine which of y^b and y^c will be stored as the new y^a and which will be stored as the new y^c. The appropriate operation is as follows, where p* now changes from referring to the current pair (y^a,y^c) (which is dropped) to

refer to the new pair (y^a,y^c) created from the current (y^b,y^c). This first part of this operation differs slightly from (O6).

If $f(s2^*) \leq f(s^*)$ then (O6a)
    (The current $y^c = y(s2^*,j)$ becomes the new y^a, and $y(s^*,j)$ becomes the new y^c.).
    SolLocation(p*,1) = s2*
    SolLocation(p*,2) = s*
Else
    ($y^b = y(s^*,j)$ becomes the new y^a. Record the new s1* and s2* values.)
    SolLocation(p*,1) = s*
    SolLocation(p*,2) = s2*
Endif
InPair(s*,1) = p* and InPair(s*,2) = - 1.
  (the – 1 is because p* is the only pair containing s*, since (y^b,y^c) is not added.).
LastNextScan = LastNextScan + 1
NextScan(LastNextScan) = p* (thus adding the pair p*, which now identifies the new (y^a,y^c), to the Next Scan List)

Case3. Both (y^a,y^b) and (y^b,y^c) survive to be added to the Next Scan List.

We don't need to check whether the current solutions $y^a = y(s1^*,j)$ and $y^c = y(s2^*,j)$ will no longer be endpoints of a solution pair.

We add the new solution y^b to the record of current solutions by performing operation (O5) as in Case 1 and Case 2.

We will use p* as the pair location index to store the current (y^a,y^b) as a new (y^a,y^c) pair, and consequently we must also generate another new pair location index p# in order to store the current (y^b,y^c) as a second new (y^a,y^c) pair. This is accomplished by the operation

p# = AvailablePairIndex(MaxAvailPair)
MaxAvailPair = MaxAvailPair – 1.

We now account for adding the pair (y^a,y^b). As in Case 1 we check to find out which of f^a and f^b is smaller, in order to determine which of y^a and y^b will be stored as the new y^a and which will be stored as the new y^c. The appropriate operation is as follows, where p* now changes from referring to the current pair (y^a,y^c) (which is dropped) to refer to the new pair (y^a,y^c) created from the current (y^a,y^b). The operation is almost the same as operation (O6) of Case 1, except that we

account for the pair index p# for the pair (y^b,y^c), since this pair will contain y^b (as will the pair indexed by p*).

If f(s1*) ≤ f(s*) then (O6b)
    (The current y^a = y(s1*,j) becomes the new y^a, and y(s*,j) becomes the new y^c.).
    (SolLocation(p*,1) = s1* is already true and doesn't need to be changed)
    SolLocation(p*,2) = s*
Else
    (y^b = y(s*,j) becomes the new y^a. Record the new s1* and s2* values.)
    SolLocation(p*,1) = s*
    SolLocation(p*,2) = s1*
Endif
InPair(s*,1) = p* and InPair(s*,2) = p#.
LastNextScan = LastNextScan + 1
NextScan(LastNextScan) = p* (thus adding the pair p*, which now identifies the new (y^a,y^c), to the Next Scan List)

Finally, as in Case 2, we find out which of f^b and f^c is smaller, in order to determine which of y^b and y^c will be stored as the new y^a and which will be stored as the new y^c. The appropriate operation is almost the same as (O6a) of Case 2, except that p# is now used to refer to the new pair (y^a,y^c) created from the current (y^b,y^c), while p* is only used to identify the value for InPair(s*,2).

If f(s2*) ≤ f(s*) then (O6c)
    (The current y^c = y(s2*,j) becomes the new y^a, and y(s*,j) becomes the new y^c.).
    SolLocation(p#,1) = s2*
    SolLocation(p#,2) = s*
Else
    (y^b = y(s*,j) becomes the new y^a. Record the new s1* and s2* values.)
    SolLocation(p#,1) = s*
    SolLocation(p#,2) = s2*
Endif
InPair(s*,1) = p# and InPair(s*,2) = p*.
LastNextScan = LastNextScan + 1
NextScan(LastNextScan) = p# (thus adding the pair p#, which now identifies the new (y^a,y^c), to the Next Scan List)

Case4. Neither (y^a,y^b) nor (y^b,y^c) survive to be added to the Next Scan List.

Here we must update the InPair lists InPair(s1*,__) and InPair(s2*,__) for the solutions y^a (accessed by s1*) and y^c (accessed by s2*),

If InPair(s1*,2) = -1 (InPair(s2*,2) = -1), then the solution y(s1*,j) (respectively, y(s2*,j)) lies solely in the single pair indexed by p*, and hence this solution will be discarded. Otherwise, we simply remove p* from the InPair record for the solution. This is done simply by executing both of the operations (O4) and (O4a).

Nothing else needs to be done for Case 4, since no new solution is added and no new pair is created. The deletion of the old pair indexed by p* is automatic, since p* is not added to the Next Scan List.

Once the last pair index i = LastScan is examined (to operate on the pair p* = ScanList(LastScan)), the Next Scan List replaces the Scan List by setting

ScanList(i) = NextScanList(i) for i = 1 to NextLastScan
LastScan = NextLastScan
NextLastScan = 0

Appendix 4: Transformed Coordinate Systems

We examine the situation where a line search algorithm is adapted to multi-dimensional function optimization by modifying variables one-at-a-time, hence effectively using unidimensional line searches along the coordinate axes. Our premise is that such an approach can be enhanced by a strategy of using more than one coordinate system. Consequently, it becomes of interest to consider the appropriate form of such alternative systems.

Geometric Motivation

We adopt the viewpoint that an alternative coordinate system, which we represent by a vector $z = (z_1, z_2, \ldots, z_n)$, will prove more valuable if it is significantly different from the original system, which we represent by the vector $y = (y_1, \ldots, y_n)$. The supposition is that a system that exhibits marked differences from the original will make it possible to produce appreciably altered search trajectories and to gain access to regions that were bypassed when employing the original system.

A geometric analysis is helpful to get a picture of how a transformed system may be made to differ significantly from the original,. We start by examining the two-dimensional case.

A coordinate system (z_1, z_2) in two dimensions that contrasts with the (y_1, y_2) system can be visualized to arise as follows. Imagine the z_1 and z_2 axes begin by coinciding with the y_1 and y_2 axes, but then are rotated until further rotation fails to increase the difference between the two systems. Evidently a 45 degree rotation creates the maximum difference.

The same effect can be achieved by creating a hypercube centered at (x_1, x_2) = (0, 0), whose vertices are given by (a) (1, 1), (-1, -1) and (b) (-1, 1), (1, -1). We have grouped the vertices in pairs of points that are diagonally opposite each other. Then we stipulate that the axes for the (z_1, z_2) system lie along the diagonals of this hypercube. The corresponding geometric construction consists of picking any point of (a) and any point of (b) as a foundation for defining z in terms of y. (All such choices, consisting of one point from (a) and one from (b), will cause the z axes to go through the same set of vertices.) Hence, for example, we may select the point (z_1, z_2) = (1, 0) to correspond to the point (y_1, y_2) = (1, 1), and the point (z_1, z_2) = (0,1) to correspond to the point (y_1, y_2) = (- 1, 1). This is algebraically equivalent to defining $z_1 = (y_1 + y_2)/2$ and $z_2 = (- y_1 + y_2)/2$. Hence the transformation we seek is given by the matrix equation z = Ay, where A is given by writing

$$2A = \begin{matrix} 1 & 1 \\ -1 & 1 \end{matrix}$$

(We have identified 2A instead of A as a convenience to avoid the use of fractions.) The inverse B of this transformation, yielding y = Bz, is given by

$$B = \begin{matrix} 1 & -1 \\ 1 & 1 \end{matrix}$$

This inverse can then be used in a line search that operates along the z coordinate dimension by referring to the equation y = Bz. (For example, the transformation discloses that changing z_1 by Δ corresponds to changing both y_1 and y_2 by Δ, whereas changing z_2 by Δ corresponds to changing y_2 by Δ changing y_1 by $-\Delta$.)

Generalization

The situation for three dimensions is a bit more complex. We envelope the point (y_1, y_2, y_3) = 0 in a hypercube whose diagonally opposed vertices are given by the pairs (a) (1, 1, 1), (-1, -1, -1), (b) (-1, 1, 1), (1, - 1, -1), (c) (-1, -1, 1), (1, 1, -1), (d) (1, -1, 1), (-1, 1, 1). Only three of these pairs are required to provide the basis for a complete coordinate system. We arbitrarily choose (a), (b)

and (c), and select the first of the two points within each (which gives a representation most closely analogous to our two-dimensional case). Then the z coordinate system can be expressed by the transformation z = Ay where

$$2A = \begin{matrix} 1 & 1 & 1 \\ -1 & 1 & 1 \\ -1 & -1 & 1 \end{matrix}$$

Again, 2A is shown in place of A to avoid listing fractions. (Geometrically it is appealing to instead identify the indicated matrix as 3A, which would provide a correspondence between the two sets of points given by z = (1 0 0), (0 1 0), (0 0 1) and y = (1 1 1), (-1 1 1), (-1 -1 1). Numerically, however, for computing the inverse it is preferable to use 2A here instead of 3A.) The inverse of A which gives the transformation y = Bz is given by

$$B = \begin{matrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 1 & 0 & 1. \end{matrix}$$

The general form of A and B can be inferred by showing the five dimensional case, where the equation z = Ay results by defining

$$2A = \begin{matrix} 1 & 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 & 1 \\ -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 \end{matrix}$$

and the inverse of A that yields the transformation y = Bz is given by

$$B = \begin{matrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \\ 1 & 0 & 0 & 0 & 1. \end{matrix}$$

The fact that A and B are inverses allows us to interchange their roles. Specifically, we reverse the preceding operations by starting out with z = By, and then recover y by the inverse transformation y = Az. Then the latter equation can be used to identify how a change of Δ in a variable z_j creates a change in each component of y.

Experimentation may be performed to address the question of whether one of the two transformations $z = Ay$ and $z = By$ is more useful than the other. The first alternative, which yields $y = Bz$, clearly entails less computation (since each change in a variable z_j affects only two components of the y vector).

Additional transformations can be derived from this same model, each creating a somewhat different relationship between y and z, by permuting the components of the y vector in the original equation $z = Ay$ or $z = By$. The more "radical" the permutation (i.e., the greater the difference between the permuted y and the original y, as measured for example by a Hamming distance measure), the more the resulting transformation will be altered. For example, a transformation that completely reverses the ordering of the components of y will yield a significant change in the relationship between z and y.

A further question to be settled by experimentation concerns the relative merits of a search based on permuting components of y versus one based on changing the roles of A and B. For large problems, the advantage of only changing two components of y for each change in a component of z will likely make the system based on $y = Bz$ preferable, and simply permuting components of y in order to obtain different transformations to drive the search.