Linear Unidimensional Scaling in the L_2-Norm: Basic Optimization Methods Using MATLAB

L. J. Hubert
University of Illinois

P. Arabie
Rutgers University

J. J. Meulman
Leiden University

The research reported here has been partially supported by the National Science Foundation through Grant No. SES-981407 (to Hubert) and by the Netherlands Organization for Scientific Research (NWO) through Grant No. 575-67-053 for the 'PIONEER' project 'Subject Oriented Multivariate Analysis' (to Meulman).

Authors' Addresses: L. J. Hubert, Department of Psychology, University of Illinois, 603 East Daniel Street, Champaign, Illinois 61820, USA; P. Arabie, Faculty of Management, Rutgers University, 180 University Avenue, Newark, NJ 07102-1895, USA; J. J. Meulman, Department of Education, Data Theory Group, Leiden University, PO Box 9555, 2300 RB Leiden, The Netherlands.

Abstract: A comparison is made among four different optimization strategies for the linear unidimensional scaling task in the L_2-norm: (1) dynamic programming; (2) an iterative quadratic assignment improvement heuristic; (3) the Guttman update strategy as modified by Pliner's technique of smoothing; (4) a nonlinear programming reformulation by Lau, Leung, and Tse. The methods are all implemented through (freely downloadable) MATLAB m-files; their use is illustrated by a common data set carried throughout. For the computationally intensive dynamic programming formulation that can guarantee a globally optimal solution, several possible computational improvements are discussed and evaluated using (a) a transformation of a given m-function with the MATLAB Compiler into C code and compiling the latter; (b) rewriting an m-function and a mandatory MATLAB gateway directly in Fortran and compiling into a MATLAB callable file; (c) comparisons of the acceleration of raw m-files implemented under the most recent release of MATLAB Version 6.5 (and compared to the absence of such acceleration under the previous MATLAB Version 6.1). Finally, and in contrast to the combinatorial optimization task of identifying a best unidimensional scaling for a given proximity matrix, an approach is given for the confirmatory fitting of a given unidimensional scaling based only on a fixed object ordering, and to nonmetric unidimensional scaling that incorporates an additional optimal monotonic transformation of the proximities.

Keywords: Combinatorial Data Analysis; Linear Unidimensional Scaling; MATLAB; Nonlinear Programming; Dynamic Programming; Quadratic Assignment.

1 Introduction

The task of linear unidimensional scaling (LUS) can be characterized as a specific data analysis problem: given a set of n objects, $S = \{O_1, ..., O_n\}$, and an $n \times n$ symmetric proximity matrix $\mathbf{P} = \{p_{ij}\}$, arrange the objects along a single dimension such that the induced $n(n-1)/2$ interpoint distances between the objects reflect the proximities in $\mathbf{P}$. The term "proximity" refers to any symmetric numerical measure of relationship between each object pair ($p_{ij} = p_{ji}$ for $1 \leq i, j \leq n$) and for which all self-proximities are considered irrelevant and set equal to zero ($p_{ii} = 0$ for $1 \leq i \leq n$). As a technical convenience, proximities are assumed nonnegative and are given a dissimilarity interpretation, so that large proximities refer to dissimilar objects.

As a starting point to be developed exclusively in this paper, we consider the most common formalization of measuring how close the interpoint distances are to the given proximities by the sum of squared discrepancies. Specifically, we wish to find the n coordinates, $x_1, x_2, \ldots, x_n$, such that the least-squares (or L_2) criterion

$$\sum_{i<j} (p_{ij} - |x_j - x_i|)^2 \tag{1}$$

is minimized. Although there is some arbitrariness in the selection of this measure of goodness-of-fit for metric scaling, the choice is traditional and has been discussed in some detail in the literature by Guttman (1968), Defays (1978), de Leeuw and Heiser (1977), and Hubert and Arabie (1986), among others. In the various sections that follow, several functions are presented within a MATLAB environment for this L_2 task based on a number of different optimization strategies: dynamic programming; the iterative use of a quadratic assignment improvement heuristic; Pliner's technique of smoothing as implemented within a Guttman update method; and a nonlinear programming reformulation by Lau, Leung, and Tse (1998).

Besides making available the basic MATLAB m-functions for carrying out these various approaches to the unidimensional scaling task, several important computational improvements are also discussed and compared for the computationally intensive dynamic programming formulation that can guarantee a globally optimal solution. These involve either transforming a given m-function with the MATLAB Compiler into C code that can in turn be submitted to a C/C++ compiler, or alternatively, rewriting an m-function and the mandatory MATLAB gateway directly in Fortran and then compiling into a MATLAB callable *.dll file (within a windows environment). In some cases studied, the computational improvements are very dramatic when the use of an external Fortran coded *.dll is compared either to one generated through C by use of the MATLAB Compiler, or as might be more expected, to the original interpreted m-function directly within a MATLAB environment.[1]

[1]In the presentation throughout the paper, reference is made continually to the m-files available in an Appendix A. This appendix (in pdf format) called lus_appendix.pdf is available at the ftp site: (ftp://www.psych.uiuc.edu/pub/cda). The m-files themselves

In addition to developing the combinatorial optimization task of actually identifying a best unidimensional scaling, Section 4 introduces two additional problems within the LUS context: (a) the confirmatory fitting of a unidimensional scale (through coordinate estimation) based on a fixed (and given) object ordering; (b) the extension to nonmetric unidimensional scaling incorporating an additional optimal monotonic transformation of the proximities. Both of these optimization tasks are formulated through the L_2 norm and carried out through applications of what is called the Dykstra-Kaczmarz method of solving linear (in)equality constrained least-squares tasks. The latter strategy is reviewed briefly in a short Appendix B to this paper.

2 LUS in the L_2 Norm

As a reformulation of the L_2 unidimensional scaling task that will prove very convenient as a point of departure in our development of computational routines, the optimization suggested by (1) can be subdivided into two separate problems to be solved simultaneously: find a set of n numbers, $x_1 \leq x_2 \leq \cdots \leq x_n$, *and* a permutation on the first n integers, $\rho(\cdot) \equiv \rho$, for which

$$\sum_{i<j} (p_{\rho(i)\rho(j)} - (x_j - x_i))^2 \tag{2}$$

is minimized. Thus, a set of locations (coordinates) is defined along a continuum as represented in ascending order by the sequence $x_1, x_2, \ldots, x_n$; the n objects are allocated to these locations by the permutation ρ, so object $O_{\rho(i)}$ is placed at location i. Without loss of generality we will impose the one additional constraint that $\sum_i x_i = 0$, i.e., any set of values, $x_1, x_2, \ldots, x_n$, can be replaced by $x_1 - \bar{x}, x_2 - \bar{x}, \ldots, x_n - \bar{x}$, where $\bar{x} = (1/n)\sum_i x_i$, without altering the value of (1) or (2). Formally, if ρ^* and $x_1^* \leq x_2^* \leq \cdots \leq x_n^*$ define a global minimum of (2), and Ω denotes the set of all permutations of the first n integers, then

$$\sum_{i<j} (p_{\rho^*(i)\rho^*(j)} - (x_j^* - x_i^*))^2 =$$

$$\min[\sum_{i<j} (p_{\rho(i)\rho(j)} - (x_j - x_i))^2 \mid \rho \in \Omega;\ x_1 \leq \cdots \leq x_n;\ \sum_i x_i = 0].$$

(plus all other files mentioned throughout the paper) are in the 'zipped' file, lus_files.zip, at this same address.

The measure of loss in (2) can be reduced algebraically:

$$\sum_{i<j} p_{ij}^2 + n(\sum_i x_i^2 - 2\sum_i x_i t_i^{(\rho)}), \tag{3}$$

subject to the constraints that $x_1 \leq \cdots \leq x_n$ and $\sum_i x_i = 0$, and letting

$$t_i^{(\rho)} = (u_i^{(\rho)} - v_i^{(\rho)})/n,$$

where

$$u_i^{(\rho)} = \sum_{j=1}^{i-1} p_{\rho(i)\rho(j)} \text{ for } i \geq 2;$$

$$v_i^{(\rho)} = \sum_{j=i+1}^{n} p_{\rho(i)\rho(j)} \text{ for } i < n,$$

and

$$u_1^{(\rho)} = v_n^{(\rho)} = 0.$$

In words, $u_i^{(\rho)}$ is the sum of the entries within row $\rho(i)$ of $\{p_{\rho(i)\rho(j)}\}$ from the extreme left up to the main diagonal; $v_i^{(\rho)}$ is the sum from the main diagonal to the extreme right. Or, we might rewrite (3) as

$$\sum_{i<j} p_{ij}^2 + n\left(\sum_i (x_i - t_i^{(\rho)})^2 - \sum_i (t_i^{(\rho)})^2\right). \tag{4}$$

In (4), the two terms $\sum_i (x_i - t_i^{(\rho)})^2$ and $\sum_i (t_i^{(\rho)})^2$ control the size of the discrepancy index since $\sum_{i<j} p_{ij}^2$ is constant for any given data matrix. Thus, to minimize the original index in (2), we should simultaneously minimize $\sum_i (x_i - t_i^{(\rho)})^2$ and maximize $\sum_i (t_i^{(\rho)})^2$. If the equivalent form of (3) is considered, our concern would be in minimizing $\sum_i x_i^2$ and maximizing $\sum_i x_i t_i^{(\rho)}$.

As noted first by Defays (1978), the minimization of (4) can be carried out directly by the maximization of the single term, $\sum_i (t_i^{(\rho)})^2$ (under the mild regularity condition that all off-diagonal proximities in $\mathbf{P}$ are positive and not merely nonnegative). Explicitly, if ρ^* is a permutation that maximizes $\sum_i (t_i^{(\rho)})^2$, then we can let $x_i = t_i^{(\rho^*)}$, which eliminates the term $\sum_i (x_i - t_i^{(\rho^*)})^2$ from (4). In short, because the order induced by $t_1^{(\rho^*)}, \ldots, t_n^{(\rho^*)}$ is consistent with the constraint $x_1 \leq x_2 \leq \cdots \leq x_n$, the minimization of (4) reduces to the maximization of the single term $\sum_i (t_i^{(\rho)})^2$, with the coordinate estimation completed as an automatic byproduct.

Table 1: The number.dat data file from Shepard, Kilpatric, and Cunningham (1975)

.000	.421	.584	.709	.684	.804	.788	.909	.821	.850
.421	.000	.284	.346	.646	.588	.758	.630	.791	.625
.584	.284	.000	.354	.059	.671	.421	.796	.367	.808
.709	.346	.354	.000	.413	.429	.300	.592	.804	.263
.684	.646	.059	.413	.000	.409	.388	.742	.246	.683
.804	.588	.671	.429	.409	.000	.396	.400	.671	.592
.788	.758	.421	.300	.388	.396	.000	.417	.350	.296
.909	.630	.796	.592	.742	.400	.417	.000	.400	.459
.821	.791	.367	.804	.246	.671	.350	.400	.000	.392
.850	.625	.808	.263	.683	.592	.296	.459	.392	.000

It is convenient to have a small numerical example available as we discuss the various optimization strategies in the unidimensional scaling context. To this end we list a data file in Table 1, called '`number.dat`', that contains a dissimilarity matrix taken from Shepard, Kilpatric, and Cunningham (1975). The stimulus domain is the first ten single-digits {0,1,2, . . . , 9} considered as abstract concepts; the 10×10 proximity matrix (with an i^{th} row or column corresponding to the $i - 1$ digit) was constructed by averaging dissimilarity ratings for distinct pairs of those integers over a number of subjects and conditions. Given the various analyses of this proximity matrix that have appeared in the literature (e.g., see Hubert, Arabie, and Meulman, 2001), the data reflect two types of very regular patterning based on absolute digit magnitude and the structural characteristics of the digits (e.g., the powers of 2 or of 3, the salience of the two additive/multiplicative identities [0/1], oddness/evenness). These data will be relied on to provide concrete numerical illustrations of the various MATLAB functions we introduce, and will be loaded as a proximity matrix (and importantly, as one that is symmetric and has zero values along the main diagonal) in the MATLAB environment by the command `'load number.dat'`. As we will see in the various applications, the dominant single unidimensional scale found for these data is most consistent with digit magnitude.

3 L_2 Norm Optimization Methods

The subsections below emphasize four distinct optimization strategies for LUS in the L_2 norm. We begin in Section 3.1 with the discussion of a dynamic programming strategy for the maximization of $\sum_i (t_i^{(\rho)})^2$ proposed by Hubert and Arabie (1986) that will produce globally optimal solutions for proximity matrices of sizes up to, say, the low twenties (within a MATLAB environment). The maximization of $\sum_i (t_i^{(\rho)})^2$ over all permutations is a prototypical combinatorial optimization task, and unfortunately, representative of the class of so-called NP-hard problems (e.g., see Garey and Johnson, 1979); thus, any procedure yielding verifiably globally optimal solutions will be severely limited by the size of the matrices that could be realistically processed. In providing MATLAB functions that carry out the dynamic programming optimization, mechanisms will also be discussed and evaluated for possibly speeding up the optimization process by the use of the MATLAB C/C++ Compiler and/or external Fortran subroutines and their gateways to allow externally generated functions to be callable from MATLAB. Section 3.2 shows how another well-known combinatorial optimization task, called quadratic assignment, can be used iteratively for LUS in the L_2 norm. Based on the reformulation in (3), we concentrate on maximizing $\sum_i x_i t_i^{(\rho)}$, with iterative re-estimation of the coordinates $x_1, \ldots, x_n$. Various function implementations within MATLAB are given both for the basic quadratic assignment task as well as for how it is used for LUS. Section 3.3 presents gradient-based update strategies for LUS based on Guttman's original update strategy as modified by a smoothing technique due to Pliner (1996); and finally, Section 3.4 implements a (reformulated) nonlinear programming heuristic from Lau, Leung, and Tse (1998) that relies on the MATLAB Optimization Toolbox (and, in particular, its function `fmincon.m`) for the MATLAB implementation that we give.

3.1 Dynamic Programming

To maximize $\sum_i (t_i^{(\rho)})^2$ over all permutations, we construct a function, $\mathcal{F}(\cdot)$, by recursion for all possible subsets of the first n integers, $\{1, 2, \ldots, n\}$:

a) $\mathcal{F}(\oslash) = 0$, where $\oslash$ is the empty set;

b) $\mathcal{F}(R') = \max[\mathcal{F}(R) + d(R, i)]$, where R' and R are subsets of size $k+1$ and k, respectively; the maximum is taken over all subsets R and indices

i such that $R' = R \cup \{i\}$; and $d(R, i)$ is the incremental value that would be added to the criterion *if* the objects in R *had* formed the first k values assigned by the optimal permutation and i had been the next assignment made, i.e., $\rho(k+1) = i$. Explicitly,

$$d(R, i) = [(1/n)\{\sum_{j \in R} p_{ij} - \sum_{j(\neq i) \notin R} p_{ij}\}]^2;$$

c) the optimal value of the criterion, $\mathcal{F}(\{1, 2, \ldots, n\})$, is obtained for $R = \{1, 2, \ldots, n\}$ and the optimal permutation, ρ^*, identified by working backwards through the recursion to identify the sequence of successive subsets of decreasing size that led to the value attained for $\mathcal{F}(\{1, 2, \ldots, n\})$.

This type of dynamic programming strategy is a very general one and can be used for any criterion for which the incremental value in identifying the index to be assigned to $\rho(k+1)$ does not depend on the particular *order* of the assigned values in the set $\{\rho(1), \ldots, \rho(k)\}$. The reader might refer to Hubert, Arabie, and Meulman (2001) for many more applications of dynamic programming in the combinatorial data analysis context.

In the three subsections that immediately follow, comparisons will be given between the execution times for four different implementations of the same dynamic programming strategy: the use of a raw m-file both under MATLAB Version 6.1 and Version 6.5, with the latter release now optimized for the execution of "loops"; an m-function transformed to C code by the MATLAB C/C++ Compiler that is in turn compiled to produce a callable *.dll file within MATLAB; the use of a Fortran produced *.dll that is called upon to do the computationally intensive recursion within a MATLAB m-file.

3.1.1 The MATLAB function uniscaldp.m

The MATLAB function m-file, `uniscaldp.m`, given in Section A.1 of Appendix A, carries out a unidimensional scaling of a symmetric proximity matrix (with a zero main diagonal and a dissimilarity interpretation) using the dynamic programming recursion just described. The usage syntax has the form

```
[coord permut cumobfun diff] = uniscaldp(prox)
```

where `PROX` is the input proximity matrix; `COORD` is the set of coordinates of the optimal unidimensional scaling in ascending order; `PERMUT` is the order of the objects in the optimal permutation (say, ρ^*); `CUMOBFUN` gives the cumulative values of the objective function for successive placements of the objects in the optimal permutation: $\sum_{i=1}^{k}(t_i^{(\rho^*)})^2$ for $k = 1, \ldots, n$; `DIFF` is the value of the least-squares loss function for the optimal coordinates and object permutation.

A recording of a MATLAB session using the `number.dat` data file follows (note that a semicolon is placed after the invocation of the m-function to initially suppress the output; transposes (') are then used on the output vectors to conserve space by only using row vectors in the listed output; also, to conserve space, blank lines are always deleted in any output given throughout the paper). The three crucial outputs are:

(1) the optimal coordinates given as a response to listing `coord'`:

-.6570, -4247, -.2608, -.1492, -.0566, .0842, .1988, .3258, .4050, .5345

(2) the optimal permutation given as a response to listing `permut'`:

1 2 3 5 4 6 7 9 10 8

(3) the minimum value of the least-squares loss function in (1) given as a response to listing `diff`:

1.959

```
>> load number.dat
>> [coord permut cumobfun diff] = uniscaldp(number);
>> coord'
ans =
  Columns 1 through 6
   -0.6570   -0.4247   -0.2608   -0.1492   -0.0566    0.0842
  Columns 7 through 10
    0.1988    0.3258    0.4050    0.5345
>> permut'
ans =
     1     2     3     5     4     6     7     9    10     8
>> cumobfun'
ans =
  Columns 1 through 6
   43.1649   61.2019   68.0036   70.2296   70.5500   71.2590
  Columns 7 through 10
   75.2111   85.8257  102.2282  130.7972
>> diff
diff =
    1.9599
```

The second column of Table 2 provides some time comparisons (in seconds) for the use of `uniscaldp.m` over randomly constructed proximity matrices of size $n \times n$ for $n = 10$ to 25. The matrices were randomly generated using the utility program `ransymat.m` given in Section A.8 of Appendix A (the usage syntax of `ransymat.m` can be seen from its header comments). The computer on which these times were obtained is a laptop with a 1.7GHz Pentium processor and 1.0 GB of RAM; for matrices larger than size 25×25, an "insufficient memory" message was consistently obtained so the largest matrix size possible in Table 2 is 25 for the available RAM. The execution times (obtained using the `tic toc` command pair in MATLAB) range from .05 seconds for $n = 10$ to 3450.4 seconds for $n = 25$. As can be seen from the timings given, there is a fairly regular proportional increase in execution time of about 2.1 for each unit increase in n.

3.1.2 The MATLAB C/C++ Compiler

One of the separate add-on components that can be obtained with MATLAB is a C/C++ Compiler that when applied to an m-file, such as `uniscaldp.m`, produces C or C++ code. The latter can itself then be compiled by a separate C/C++ compiler to produce (in a windows environment) a *.dll file that can be called within MATLAB just like a *.m function. As an example, we first renamed a version of `uniscaldp.m` to `uniscaldpc.m` and then applied in MATLAB 6.5 the C/C++ Compiler Version 3.0 with all the possible optimization options selected; the resulting code was compiled with the built-in C compiler (called `lcc`) to produce `uniscaldpc.dll`

We give the timings for the use of `uniscaldpc.dll` in the fourth column of Table 2. As n increases by 1, the execution times increase by a little more than twice, just as for `uniscaldp.m`; but overall (and rather incredibly, we might add) the compiled `uniscaldpc.dll` executes at about 7 to 8 times *slower* than the interpreted file `uniscaldp.m`. The times are given for running `uniscaldp.m` under MATLAB 6.1 in the third column of Table 1 (but with the same computer). Here, the execution times are a little less than four times slower than for the compiled C code. So, it is pretty obvious that the accelerations provided in the new release of MATLAB (in 6.5) are working as promised, with a decrease in execution rate generally between 20 and 30 times for our recursive-intensive m-function. At least for speed of execution considerations, these results more-or-less obviate the value of ever using the MATLAB C/C++ Compiler when MATLAB version 6.5 is available.

3.1.3 External Fortran subroutines

One strategy for decreasing the execution time of an m-file would replace part of the code that may be slow (usually nonvectorizable "for" loops, for example) by a call to a *.dll that is produced from a Fortran subroutine implementing the code in its own language. As an example, the m-function listed in Appendix A.2, `uniscaldpf.m`, is a parallel of `uniscaldp.m` except that the actual recursion constructing the two crucial vectors (and from which the optimal solution is eventually identified by working backwards) is replaced by a call to `uscalfor.dll`. This latter file (called a Fortran MEX-file in MATLAB) was produced using the MATLAB Application Program Interface (API) through the Compaq Digital Visual Fortran (6.6) Compiler and with the central computational Fortran subroutine `uscalfor.for` in Appendix Section A.2.1 and the second necessary gateway Fortran subroutine `uscalforgw.for` in Appendix Section A.2.2. These two subroutines are compiled together to produce `uscalfor.dll` that is then called in `uniscaldpf.m` to do "the heavy lifting".

In comparison to the other times listed in Table 2, the speedup provided by using the routine `uniscaldpf.m` is rather spectacular. There is generally the same type of proportional increase in time that is slightly greater than twice for each unit change in n. In comparison to the basic m-function, `uniscaldp.m`, running under MATLAB 6.5, there is a 20/25 increase in execution speed for `uniscaldpf.m`.

3.2 Iterative Quadratic Assignment

Because of the manner in which the discrepancy index for the unidimensional scaling task can be rephrased as in (3) and (4), the two optimization subproblems to be solved simultaneously of identifying an optimal permutation and a set of coordinates can be separated:

(a) assuming that an ordering of the objects is known (and denoted, say, as ρ^0 for the moment), find those values $x_1^0 \leq \cdots \leq x_n^0$ to minimize $\sum_i (x_i^0 - t_i^{(\rho^0)})^2$. If the permutation ρ^0 produces a *monotonic* form for the matrix $\{p_{\rho^0(i)\rho^0(j)}\}$ in the sense that $t_1^{(\rho^0)} \leq t_2^{(\rho^0)} \leq \cdots \leq t_n^{(\rho^0)}$, the coordinate estimation is immediate by letting $x_i^0 = t_i^{(\rho^0)}$, in which case $\sum_i (x_i^0 - t_i^{(\rho^0)})^2$ is zero.

(b) assuming that the locations $x_1^0 \leq \cdots \leq x_n^0$ are known, find the permu-

Table 2: Time comparisons in seconds for the various implementations of unidimensional scaling through dynamic programming. The three entries marked by an asterisk are estimated values.

matrix size	uniscaldp.m (Version 6.5)	uniscaldp.m (Version 6.1)	uniscaldpc.dll	uniscaldpf.m
10	.05	.85	.28	.00
11	.11	1.86	.54	.00
12	.23	4.14	1.14	.00
13	.46	9.29	2.50	.02
14	.89	20.85	5.49	.04
15	1.81	46.50	12.43	.08
16	3.85	103.65	27.65	.19
17	8.23	228.84	61.30	.39
18	17.59	503.88	135.84	.84
19	37.62	1107.3	300.06	1.99
20	80.10	2473.2	648.26	3.87
21	170.60	5267.8	1421.1	8.15
22	362.19	11420.	3097.7	18.30
23	767.93	24845.	6772.0	40.40
24	1625.3	54052.*	14985.	85.99
25	3450.4	117595.*	33158.*	185.93

tation ρ^0 to maximize $\sum_i x_i t_i^{(\rho^0)}$. We note from the work of Hubert and Arabie (1986, p. 189) that any such permutation which even only locally maximizes $\sum_i x_i t_i^{(\rho^0)}$, in the sense that no adjacently placed pair of objects in ρ^0 could be interchanged to increase the index, will produce a monotonic form for the non-negative matrix $\{p_{\rho^0(i)\rho^0(j)}\}$. Also, the task of finding the permutation ρ^0 to maximize $\sum_i x_i t_i^{(\rho^0)}$ is actually a quadratic assignment (QA) task which has been discussed extensively in the literature of operations research, e.g., see Francis and White (1974), Lawler (1975), Hubert and Schultz (1976), among others. As usually defined, a QA problem involves two $n \times n$ matrices $\mathbf{A} = \{a_{ij}\}$ and $\mathbf{B} = \{b_{ij}\}$, and we seek a permutation ρ to maximize

$$\Gamma(\rho) = \sum_{i,j} a_{\rho(i)\rho(j)} b_{ij}. \tag{5}$$

If we define $b_{ij} = |x_i - x_j|$ and let $a_{ij} = p_{ij}$, then

$$\Gamma(\rho) = \sum_{i,j} p_{\rho(i)\rho(j)} |x_i - x_j| = 2n \sum_i x_i t_i^{(\rho)},$$

and thus, the permutation that maximizes $\Gamma(\rho)$ also maximizes $\sum x_i t_i^{(\rho)}$.

The QA optimization task as formulated through (5) has an enormous literature attached to it, and the reader is referred to Pardalos and Wolkowicz (1994) for an up-to-date and comprehensive review. For current purposes and as provided in three general m-functions of the next section (`pairwiseqa.m`, `rotateqa.m`, and `insertqa.m`), one might consider the optimization of (5) through simple object interchange/rearrangement heuristics. Based on given matrices **A** and **B**, and beginning with some permutation (possibly chosen at random), local interchanges/rearrangements of a particular type are implemented until no improvement in the index can be made. By repeatedly initializing such a process randomly, a distribution over a set of local optima can be achieved. At least within the context of some common data analysis applications, such a distribution may be highly relevant diagnostically for explaining whatever structure might be inherent in the matrix **A**.

In a subsequent subsection below, we introduce the main m-function (`uniscalqa.m`) for unidimensional scaling based on these earlier QA optimization strategies. In effect, we begin with an equally-spaced set of fixed coordinates with their interpoint distances defining the **B** matrix of the general QA index in (5) and a random object permutation; a locally-optimal permutation is then identified through a collection of local interchanges/rearrangements; the coordinates are re-estimated based on this identified permutation, and the whole process repeated until no change can be made in either the identified permutation or coordinate collection.

3.2.1 The QA interchange/rearrangement heuristics

The three m-functions of Appendix A.9 that carry out general QA interchange/rearrangement heuristics all have the same general usage syntax (note the use of three dots to denote a statement continuation in MATLAB):

```
[outperm rawindex allperms index] = pairwiseqa(prox,targ,inperm)

[outperm rawindex allperms index] = ...
                        rotateqa(prox,targ,inperm,kblock)

[outperm rawindex allperms index] = ...
                        insertqa(prox,targ,inperm,kblock)
```

`pairwiseqa.m` carries out an iterative QA maximization task using the pairwise interchanges of objects in the current permutation defining the row and column order of the data matrix. All possible such interchanges are generated and considered in turn, and whenever an increase in the cross-product index would result from a particular interchange, it is made immediately. The process continues until the current permutation cannot be improved upon by any such pairwise object interchange; this final locally optimal permutation is `OUTPERM`. The input beginning permutation is `INPERM` (a permutation of the first n integers); `PROX` is the $n \times n$ input proximity matrix and `TARG` is the $n \times n$ input target matrix (which are respective analogues of the matrices $\mathbf{A}$ and $\mathbf{B}$ of (5)); the final `OUTPERM` row and column permutation of `PROX` has the cross-product index `RAWINDEX` with respect to `TARG`. The cell array `ALLPERMS` contains `INDEX` entries corresponding to all the permutations identified in the optimization, from `ALLPERMS{1} = INPERM` to `ALLPERMS{INDEX} = OUTPERM`. (Note that within a MATLAB environment, entries of a cell array must be accessed through the curly braces, { }.) `rotateqa.m` carries out a similar iterative QA maximization task but now uses the rotation (or inversion) of from 2 to `KBLOCK` (which is less than or equal to $n-1$) consecutive objects in the current permutation defining the row and column order of the data matrix. `insertqa.m` relies on the (re-)insertion of from 1 to `KBLOCK` consecutive objects somewhere in the permutation defining the current row and column order of the data matrix.

3.2.2 The MATLAB function uniscalqa.m

The MATLAB function m-file in Section A.3 of Appendix A, `uniscalqa.m`, carries out a unidimensional scaling of a symmetric dissimilarity matrix (with a zero main diagonal) using an iterative quadratic assignment strategy. We begin with an equally-spaced target, a (random) starting permutation, and use a sequential combination of the pairwise interchange/rotation/insertion heuristics; the target matrix is re-estimated based on the identified (locally optimal) permutation. The whole process is repeated until no changes can be made in the target or the identified (locally optimal) permutation. The explicit usage syntax is

```
[outperm rawindex allperms index coord diff] = ...
                   uniscalqa(prox,targ,inperm,kblock)
```

where all terms are present either in `uniscaldp.m` or in the three QA heuristic m-functions of the previous subsection. A recording of a MATLAB session using `number.dat` follows with results completely consistent with what was identified using `uniscaldp.m`. Note the application of the built-in MATLAB function `randperm(10)` to obtain a random input permutation of the first 10 digits, and the use of the utility m-function from the Appendix A.8.1, `ransymat(10)`, to generate a target matrix `targlin` based on an equally (and unit) spaced set of coordinates. Also, the optimal permutation given in response to `outperm` is the (equivalent) reversal of that given earlier using `uniscaldp.m`; thus, the optimal coordinates given in response to `coord'` are listed in the reverse order as well.

```
>> load number.dat
>> [prox10 targlin targcir] = ransymat(10);
>> inperm = randperm(10);
>> kblock = 2;
>> [outperm rawindex allperms index coord diff] = ...
uniscalqa(number,targlin,inperm,kblock);
>> outperm
```

```
outperm =
     8    10     9     7     6     4     5     3     2     1
>> coord'
ans =
  Columns 1 through 6
   -0.5345   -0.4050   -0.3258   -0.1988   -0.0842    0.0566
  Columns 7 through 10
    0.1492    0.2608    0.4247    0.6570
>> diff
diff =
    1.9599
```

3.3 Gradient-Based Optimization with Pliner's Smoothing Strategy

In Guttman's 1968 paper on multidimensional scaling, the optimization task in (1) is treated as a special case of a general iterative algorithm based on the partial derivatives of (1) with respect to the unknown locations. For one dimension, Guttman's multidimensional scaling algorithm reduces to a simple updating procedure:

$$x_i^{(t+1)} = \frac{1}{n}\sum_{j=1}^{n} p_{ij}\text{sign}(x_i^{(t)} - x_j^{(t)}), \tag{6}$$

where t is the index of iteration. As pointed out by de Leeuw and Heiser (1977), convergence of (6) is guaranteed because $x_i^{(t+1)}$ only depends on the rank order of $x_1^{(t)}, \ldots, x_n^{(t)}$, there are a finite number of different rank orders, and no rank order can be repeated with intermediate different rank orders. In fact, the stationary points of (6) are defined by all possible orderings of $\mathbf{P}$ that lead to monotonic forms. Specifically, if $x_1, \ldots, x_n$ is a stationary point of (6) and ρ is the permutation for which $x_{\rho(1)} \le x_{\rho(2)} \le \cdots \le x_{\rho(n)}$, then $\{p_{\rho(i)\rho(j)}\}$ is monotonic, i.e., $t_1^{(\rho)} \le \cdots \le t_n^{(\rho)}$, and $t_i^{(\rho)} = x_{\rho(i)}$ for $1 \le i \le n$. Conversely, if $\{p_{ij}\}$ is monotonic, then (6) converges in one step if we let the initial value of x_i be, say, i for $1 \le i \le n$.

Guttman's updating algorithm is in reality a procedure for finding monotonic forms for a proximity matrix and only very indirectly can it even be characterized as a strategy for unidimensional scaling. From a somewhat wider perspective, the general weakness of the monotonic forms for a given matrix may indicate why multidimensional scaling methods generally have such difficulties with local optima when restricted to a single dimension (e.g., see Shepard, 1974, pp. 378–379). As can be seen in the way the index of goodness-of-fit is rewritten in (4), the crucial quantity for distinguishing among different monotonic forms is $\sum_i (t_i^{(\rho)})^2$. Consideration of this latter term disappears in Guttman's update method because of the algorithm's reliance on a gradient approach; but as we will see in Section 3.3.2, there is a method for improving upon the basic update strategy enormously through a smoothing strategy.

3.3.1 The MATLAB function guttorder.m

The MATLAB m-function in Section A.4 of Appendix A, `guttorder.m`, carries out a unidimensional scaling of a symmetric proximity matrix based on the Guttman update formula in (6). The usage syntax is

```
[gcoordsort gperm] = guttorder(prox,inperm)
```

where `PROX` and `INPERM` are as before, and the output vector `GCOORDSORT` contains the coordinates ordered from the most negative to most positive; `GPERM` is the object permutation indicating where the objects are placed at the ordered coordinates in `GCOORDSORT`. One easy exercise for the reader would be to call `guttorder` with `inperm` as `randperm(10)` and `prox` as `number` and merely use the 'up arrow' key to retrieve the call to `guttorder` and rerun the routine with a new random starting permutation. One will quickly see the weakest of the update procedure in (6) in finding anything that isn't just another local optimum.

3.3.2 Pliner's smoothing strategy and the MATLAB function plinorder.m

Although the use of the basic Guttman update formula is severely prone to finding only local optima, a smoothing strategy applied to (6) seems to alleviate this problem (almost) completely. Very simply, Pliner's (1996) smoothing strategy for the sign function would replace sign(t) in (6) with

$$\begin{array}{ll} (t/\epsilon)(2 - [|t|/\epsilon]) & \text{if } |t| \leq \epsilon; \\ \text{sign } (t) & \text{if } |t| > \epsilon, \end{array}$$

for $\epsilon > 0$. Beginning with a randomly generated set of initial coordinate values and a sufficiently large value of ϵ (e.g., in the m-function `plinorder.m` introduced below, we use Pliner's suggestion of an initial value of ϵ equal to twice the maximum of the row (or column) averages of the input proximity matrix), the update in (6) (with the replacement smoother) would be applied until convergence. The parameter ϵ (given as `ep` in the m-function) is then reduced (e.g., we use `ep = ep*(100-k+1)/100` for `k = 2:100`), and beginning with the coordinates from the previous solution, the update in (6) is again applied until convergence. The process continues until ϵ has been effectively reduced to zero.

Pliner's strategy is a relatively simple modification in the use of the iterative update in (6), and although it is still a heuristic strategy in the sense that a globally optimal solution is not guaranteed, the authors' experience with it suggests that it works incredibly well. (We might also add that because of its computational simplicity and speed of execution, it may be the key to scaling huge proximity matrices.) The m-function `plinorder.m` in Section A.5 of the appendix has the usage syntax as follows:

```
[pcoordsort pperm gcoordsort gperm gdiff pdiff] = ...
                                 plinorder(prox,inperm)
```

where some of the terms are the same as in `guttorder.m` since that update method is initially repeated with the invocation of `plinorder.m`; `PCOORDSORT` and `PPERM` are analogues of `GCOORDSORT` and `GPERM` but using the smoother, and `PDIFF` and `GDIFF` are the least-squares loss function values for using the Pliner smoother and the Guttman update, respectively. The pattern illustrated by the single call of `plinorder.m` to follow is expected: the smoothing strategy identifies a globally optimal solution and the Guttman update provides one that is only locally optimal.

```
>> load number.dat
>> inperm = randperm(10);
>> [pcoordsort pperm gcoordsort gperm gdiff pdiff] = ...
plinorder(number,inperm);
>> pcoordsort'
ans =
  Columns 1 through 6
   -0.6570   -0.4247   -0.2608   -0.1492   -0.0566    0.0842
  Columns 7 through 10
```

```
    0.1988    0.3258    0.4050    0.5345
>> pperm
pperm =
     1     2     3     5     4     6     7     9    10     8
>> gcoordsort'
ans =
  Columns 1 through 6
   -0.5089   -0.3784   -0.2160   -0.1426   -0.1110    0.1316
  Columns 7 through 10
    0.1735    0.2902    0.3502    0.4114
>> gperm
gperm =
     2     6     5     9     1    10     8     4     3     7
>> gdiff
gdiff =
    5.9895
>> pdiff
pdiff =
    1.9599
```

3.4 A Nonlinear Programming Heuristic

In considering the unidimensional scaling task in (1), Lau, Leung, and Tse (1998) note the equivalence to the minimization over $x_1, x_2, \ldots, x_n$ of

$$\sum_{i<j} \min\{[p_{ij} - (x_i - x_j)]^2, [p_{ij} - (x_j - x_i)]^2\}. \tag{7}$$

Two zero/one variables can then be defined, w_{1ij} and w_{2ij}, and (7) rewritten as the mathematical program

$$\text{minimize} \sum_{i<j} \{w_{1ij}(e_{1ij})^2 + w_{2ij}(e_{2ij})^2\} \tag{8}$$

subject to

$$p_{ij} = x_i - x_j + e_{1ij};$$

$$p_{ij} = x_j - x_i + e_{2ij};$$

$$w_{1ij} + w_{2ij} = 1;$$

$$w_{1ij}, w_{2ij} \geq 0,$$

where e_{1ij} is the error if $x_i > x_j$ and e_{2ij} is the error if $x_i < x_j$. These authors observe that the binary restriction on w_{1ij} and w_{2ij} can be removed since they will automatically be forced to zero or one. In short, what initially appears as a combinatorial optimization task in (2) has now been replaced by a nonlinear programming model in (8).

3.4.1 The MATLAB function unifitl2nlp.m

The m-function `unifitl2nlp.m` given in Section A.6 carries out the optimization task specified in (8) by a call to a very general m-function from the MATLAB Optimization Toolbox, `fmincon.m`. The latter is an extremely general routine for the minimization of a constrained multivariable function, and requires in our case a separate m-function, `objfunl2.m`, that we give in Section A.6.1 to evaluate the objective function in (8). So, to use the function `unifitl2nlp.m`, the user needs to have the Optimization Toolbox installed. The usage syntax for `unifitl2nlp.m` has the form

```
[startcoord begval outcoord endval exitflag] = ...
                            unifitl2nlp(prox,inperm)
```

An input permutation `INPERM` is used to obtain a set of starting coordinates (`STARTCOORD`) that would lead to an initial least-squares loss value (`BEGVAL`). The starting coordinates are obtained from the usual $t_i^{(\rho)}$ formula of (3) irrespective of whether `INPERM` provides a monotonic form for the reordered matrix `PROX(INPERM,INPERM)` or not. The ending coordinates (`OUTCOORD`) at the end of the process leads to a final least-squares loss value (`ENDVAL`). The `EXITFLAG` variable gives the success of the optimization (greater than 0 indicates convergence; 0 implies that the maximum number of function evaluations or iterations were reached; less than 0 denotes nonconvergence).

An example of the use of `unifitl2nlp.m` is given below for two starting permutations — the identity ordering and the second one random. Given these results and others that the reader can replicate given the availability of the m-function, it appears in general that "the apple is not allowed to fall very far from the tree". The end result is very close to where one starts, which is very similar to the disappointing performance of an unmodified Guttman update strategy. The need to have such a good initial permutation to start with, pretty much defeats the use of the nonlinear programming reformulation as a search technique. Both iterative QA and Pliner's smoother, which can begin just with random permutations and usually end up with very good final permutations, would appear thus far to be the heuristic methods of choice.

```
>> load number.dat
>> inperm = [1 2 3 4 5 6 7 8 9 10];
>> [startcoord begval outcoord endval exitflag] = ...
unifitl2nlp(number,inperm);
>> outcoord'
ans =
  Columns 1 through 6
   -0.6570   -0.4247   -0.2608   -0.1392   -0.0666    0.0842
  Columns 7 through 10
    0.1988    0.3627    0.4058    0.4968
>> endval
endval =
    2.1046
>> inperm = randperm(10);
>> [startcoord begval outcoord endval exitflag] = ...
unifitl2nlp(number,inperm);
>> outcoord'
ans =
  Columns 1 through 6
    0.2094   -0.0955    0.2728    0.0100   -0.3778    0.2434
  Columns 7 through 10
    0.0312   -0.3061   -0.4842    0.4968
>> endval
endval =
    5.9856
```

4 The Implementation of Nonmetric and Confirmatory LUS

In developing linear unidimensional scaling (as well as other types of) representations for a proximity matrix, it is convenient to have a general mechanism available for solving linear (in)equality constrained least-squares tasks. The two such instances discussed in this section involve (a) the confirmatory fitting of a given object order to a proximity matrix (through an m-file called `linfit.m`), and (b) the construction of an optimal monotonic transformation of a proximity matrix in relation to a given unidimensional ordering (through an m-file called `proxmon.m`). In both of these cases, we rely on what can be called the Dykstra-Kaczmarz method. An equality constrained least-squares task may be rephrased as a linear system of equations, with the later solvable through a strategy of iterative projection as attributed to Kaczmarz (1937; see Bodewig, 1956, pp. 163–164); a more general inequality constrained least-squares task can also be approached through iterative projection as developed by Dykstra (1983). The Kaczmarz and Dykstra strategies are reviewed very briefly in Appendix B, and implemented within the two m-files, `linfit.m` and `proxmon.m`, discussed below.

4.1 The confirmatory fitting of a given order using the MATLAB function linfit.m

The MATLAB m-function in Section A.7, `linfit.m`, fits a set of coordinates to a given proximity matrix based on some given input permutation, say, $\rho^{(0)}$. Specifically, we seek $x_1 \leq x_2 \leq \cdots \leq x_n$ such that $\sum_{i<j}(p_{\rho^0(i)\rho^0(j)} - |x_j - x_i|)^2$ is minimized (and where the permutation $\rho^{(0)}$ may not even put the matrix $\{p_{\rho^0(i)\rho^0(j)}\}$ into a monotonic form). Using the syntax

```
[fit diff coord] = linfit(prox,inperm)
```

the matrix $\{|x_j - x_i|\}$ is referred to as the fitted matrix (`FIT`); `COORD` gives the ordered coordinates; and `DIFF` is the value of the least-squares criterion. The fitted matrix is found through the Dykstra-Kaczmarz method where the equality constraints defined by distances along a continuum are imposed to find the fitted matrix, i.e., if $i < j < k$, then $|x_i - x_j| + |x_j - x_k| = |x_i - x_k|$. Once found, the actual ordered coordinates are retrieved by the usual $t_i^{(\rho^0)}$ formula in (3) but computed on `FIT`.

The example below of the use of `linfit.m` fits two separate orders: the identity permutation and the one that we know is least-squares optimal. The consistency of the results can be compared to those given earlier.

```
>> load number.dat
>> inperm = [1 2 3 4 5 6 7 8 9 10];
>> [fit diff coord] = linfit(number,inperm);
>> coord'
ans =
  Columns 1 through 6
   -0.5345   -0.4050   -0.3258   -0.1988   -0.0842    0.0566
  Columns 7 through 10
    0.1492    0.2608    0.4247    0.6570
>> diff
diff =
    2.1046
>> inperm = [1 2 3 5 4 6 7 9 10 8];
>> [fit diff coord] = linfit(number,inperm);
>> coord'
ans =
  Columns 1 through 6
   -0.6570   -0.4247   -0.2608   -0.1492   -0.0566    0.0842
  Columns 7 through 10
    0.1988    0.3258    0.4050    0.5345
>> diff
diff =
    1.9599
```

4.2 The monotonic transformation of a proximity matrix using the MATLAB function proxmon.m

The MATLAB function, `proxmon.m`, given in Section A.10, provides a monotonically transformed proximity matrix that is close in a least-squares sense to a given input matrix. The syntax is

```
[monproxpermut vaf diff] = proxmon(proxpermut,fitted)
```

Here, `PROXPERMUT` is the input proximity matrix (which may have been subjected to an initial row/column permutation, hence the suffix '`PERMUT`') and `FITTED` is a given target matrix; the output matrix `MONPROXPERMUT` is closest to `FITTED` in a least-squares sense and obeys the order constraints obtained from each pair of entries in (the upper-triangular portion of) `PROXPERMUT` (and where the inequality constrained optimization is carried out using the Dykstra-Kaczmarz iterative projection strategy); `VAF` denotes 'variance-accounted-for' and indicates how much variance in `MONPROXPERMUT` can be accounted for by `FITTED`; finally `DIFF` is the value of the least-squares loss function and is (one-half) the sum of squared differences between the entries in `FITTED` and `MONPROXPERMUT`.

In the notation of the previous section when fitting a given order, `FITTED` would correspond to the matrix $\{|x_j - x_i|\}$, where $x_1 \leq x_2 \leq \cdots \leq x_n$; the input `PROXPERMUT` would be $\{p_{\rho^0(i)\rho^0(j)}\}$; `MONPROXPERMUT` would be $\{f(p_{\rho^0(i)\rho^0(j)})\}$, where the function $f(\cdot)$ satisfies the monotonicity constraints, i.e., if $p_{\rho^0(i)\rho^0(j)} < p_{\rho^0(i')\rho^0(j')}$ for $1 \leq i < j \leq n$ and $1 \leq i' < j' \leq n$, then $f(p_{\rho^0(i)\rho^0(j)}) \leq f(p_{\rho^0(i')\rho^0(j')})$. The transformed proximity matrix $\{f(p_{\rho^0(i)\rho^0(j)})\}$ minimizes the least-squares criterion (`DIFF`) of

$$\sum_{i<j}(f(p_{\rho^0(i)\rho^0(j)}) - |x_j - x_i|)^2,$$

over all functions $f(\cdot)$ that satisfy the monotonicity constraints. The `VAF` is a normalization of this loss value by the sum of squared deviations of the transformed proximities from their mean:

$$\text{VAF} = 1 - \frac{\sum_{i<j}(f(p_{\rho^0(i)\rho^0(j)}) - |x_j - x_i|)^2}{\sum_{i<j}(f(p_{\rho^0(i)\rho^0(j)}) - \bar{f})^2},$$

where $\bar{f}$ denotes the mean of the off-diagonal entries in $\{f(p_{\rho^0(i)\rho^0(j)})\}$.

4.2.1 An application incorporating proxmon.m

The script m-file listed below gives an application of `proxmon.m` using the globally optimal permutation found previously for our `number.dat` matrix. First, `linfit.m` is invoked to obtain a fitted matrix (`fit`); `proxmon.m` then generates the monotonically transformed proximity matrix (`monproxpermut`) with `vaf` = .5821 and `diff` = 1.0623. The strategy is then repeated cyclically (i.e., finding a fitted matrix based on the monotonically transformed proximity matrix, finding a new monotonically transformed matrix, and so on). To avoid degeneracy (where all matrices would converge to zeros), the sum of squares of the fitted matrix is kept the same as it was initially; convergence is based on observing a minimal change (less than 1.0e-006) in the `vaf`. As indicated in the output below, the final `vaf` is .6672 with a `diff` of .9718. (Although the globally optimal permutation found earlier for `number.dat` remains the same throughout the construction of the optimal monotonic transformation, in this particular example it would also remain optimal with the same vaf if the unidimensional scaling was repeated with `monproxpermut` now considered the input proximity matrix. Even though probably rare, other data sets might not have such an invariance, and it may be desirable to initiate an iterative routine that finds both a unidimensional scaling [i.e., an object ordering] in addition to monotonically transforming the proximity matrix.)

```
load number.dat
inperm = [8 10 9 7 6 4 5 3 2 1];
[fit diff coord] = linfit(number,inperm);
[monproxpermut vaf diff] = ...
    proxmon(number(inperm,inperm),fit);
sumfitsq = sum(sum(fit.^2));
prevvaf = 2;
while (abs(prevvaf-vaf) >= 1.0e-006)
   prevvaf = vaf;
   [fit diff coord] = linfit(monproxpermut,1:10);
   sumnewfitsq = sum(sum(fit.^2));
   fit = sqrt(sumfitsq)*(fit/sqrt(sumnewfitsq));
    [monproxpermut vaf diff] = proxmon(number(inperm,inperm), fit);
end

fit
diff
coord'
monproxpermut
vaf

fit =
  Columns 1 through 6
         0    0.0824    0.1451    0.3257    0.4123    0.5582
    0.0824         0    0.0627    0.2432    0.3298    0.4758
    0.1451    0.0627         0    0.1806    0.2672    0.4131
    0.3257    0.2432    0.1806         0    0.0866    0.2325
    0.4123    0.3298    0.2672    0.0866         0    0.1459
    0.5582    0.4758    0.4131    0.2325    0.1459         0
    0.5834    0.5010    0.4383    0.2578    0.1711    0.0252
    0.7244    0.6419    0.5793    0.3987    0.3121    0.1662
    0.8696    0.7872    0.7245    0.5440    0.4573    0.3114
    1.2231    1.1406    1.0780    0.8974    0.8108    0.6649
  Columns 7 through 10
    0.5834    0.7244    0.8696    1.2231
    0.5010    0.6419    0.7872    1.1406
    0.4383    0.5793    0.7245    1.0780
    0.2578    0.3987    0.5440    0.8974
    0.1711    0.3121    0.4573    0.8108
    0.0252    0.1662    0.3114    0.6649
         0    0.1410    0.2862    0.6397
    0.1410         0    0.1452    0.4987
    0.2862    0.1452         0    0.3535
    0.6397    0.4987    0.3535         0
diff =
    0.9718
ans =
  Columns 1 through 6
   -0.4558   -0.3795   -0.3215   -0.1544   -0.0742    0.0609
  Columns 7 through 10
    0.0842    0.2147    0.3492    0.6764
monproxpermut =
  Columns 1 through 6
         0    0.2612    0.2458    0.2612    0.2458    0.5116
    0.2612         0    0.2458    0.2458    0.4286    0.2458
    0.2458    0.2458         0    0.2458    0.5116    0.6899
    0.2612    0.2458    0.2458         0    0.2458    0.2458
    0.2458    0.4286    0.5116    0.2458         0    0.2612
    0.5116    0.2458    0.6899    0.2458    0.2612         0
    0.6080    0.5116    0.2458    0.2458    0.2458    0.2458
    0.6899    0.7264    0.2458    0.2612    0.5116    0.2458
    0.5116    0.5116    0.6899    0.6080    0.4286    0.2458
    1.2231    1.1406    1.0780    0.6899    0.7264    0.6080
  Columns 7 through 10
    0.6080    0.6899    0.5116    1.2231
    0.5116    0.7264    0.5116    1.1406
    0.2458    0.2458    0.6899    1.0780
    0.2458    0.2612    0.6080    0.6899
    0.2458    0.5116    0.4286    0.7264
    0.2458    0.2458    0.2458    0.6080
```

```
         0    0.1410    0.5116    0.6080
    0.1410         0    0.2458    0.4286
    0.5116    0.2458         0    0.2612
    0.6080    0.4286    0.2612         0
vaf =
    0.6672
```

5 Some Concluding Comments

The computational results presented in this paper suggest that for proximity matrices up to, say, 20×20, guaranteed optimal linear unidimensional scalings are easily done using just an interpreted m-file as long as one is working under Version 6.5 of MATLAB. For matrices up to 25 × 25, the Fortran enhanced routine would provide the computational strategy of choice. Beyond matrices of size 25, either iterative QA or Pliner's smoothing strategy should lead to optimal solutions (although not verifiably so) upon the use of some number of random starts. Generally, both the original Guttman (1968) update strategy or the nonlinear programming reformulation of Lau, Lam, and Tse (1998) are problematic, and should be avoided.

With respect to broader computational issues, there are four observations that might be made: (1) the optimization of "loops" incorporated in MATLAB Version 6.5 (and in contrast to Version 6.1) is performing extremely well, and may in fact allow the simple use of raw m-files for situations where before, a formal search might have been undertaken to find more computationally efficient alternatives; (2) compiling the type of C code that is constructed by the MATLAB C/C++ Compiler seems to lead to particularly inefficient routines, even as compared to just using the raw m-files under the new MATLAB Version 6.5; for questions of execution speed, the MATLAB Compiler has very questionable utility. The issue then arises as to whether it is the C language/compiler per se, or the initial code generation that leads to these particularly inefficient routines; (3) the use of Fortran subroutines (and their respective gateways) to carry out computationally intensive subtasks, can be extremely advantageous; for example, we can now construct verifiably optimal linear unidimensional scalings for fairly large problems (up to 25 × 25) within reasonable computation times.

Based on what we have learned computationally in this paper, the next logical steps would be to extend the presented MATLAB routines to multiple linear unidimensional scales, and thus, to implement city-block multidimensional scaling. These generalizations could include the movement away from the reliance on just the L_2 norm (say, to the L_1 norm as well), and to the automatic incorporation of optimal (monotonic) transformations of the given proximity matrix. Included in these extensions would be an option for an "individual differences model" in which the "group space" would consist solely of object orders along the given axes, and the "private spaces" defined by individually specified coordinates for each of the separate sources that are consistent with respect to these group-space orders. These extensions are now being prepared for eventual publication.

References

[1] Bodewig, E. (1956). *Matrix calculus.* Amsterdam: North-Holland.

[2] Defays, D. (1978). A short note on a method of seriation. *British Journal of Mathematical and Statistical Psychology*, *3*, 49–53.

[3] de Leeuw, J., & Heiser, W. (1977). Convergence of correction-matrix algorithms for multidimensional scaling. In J. C. Lingoes, E. E. Roskam, & I. Borg (Eds.), *Geometric representations of relational data* (pp. 735–752). Ann Arbor, MI: Mathesis Press.

[4] Dykstra, R. L. (1983). An algorithm for restricted least squares regression. *Journal of the American Statistical Association*, *78*, 837–842.

[5] Francis, R. L., & White, J. A. (1974). *Facility layout and location: An analytical approach.* Englewood Cliffs, NJ: Prentice-Hall.

[6] Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability.* San Francisco: W. H. Freeman.

[7] Guttman, L. (1968). A general nonmetric technique for finding the smallest coordinate space for a configuration of points. *Psychometrika*, *33*, 469–506.

[8] Hubert, L. J., & Arabie, P. (1986). Unidimensional scaling and combinatorial optimization. In J. de Leeuw, W. Heiser, J. Meulman, & F. Critchely (Eds.), *Multidimensional data analysis* (pp. 181–196). Leiden, The Netherlands: DSWO Press.

[9] Hubert, L. J., Arabie, P., & Meulman, J. (2001). *Combinatorial data analysis: Optimization by dynamic programming.* Philadelphia: SIAM.

[10] Hubert, L. J., & Schultz, J. W. (1976). Quadratic assignment as a general data analysis strategy. *British Journal of Mathematical and Statistical Psychology*, *29*, 190–241.

[11] Kaczmarz, S. (1937). Angenäherte Auflösung von Systemen linearer Gleichungen. *Bulletin of the Polish Academy of Sciences*, *A35*, 355–357.

[12] Lawler, E. L. (1975). The quadratic assignment problem: A brief review. In R. Roy (Ed.), *Combinatorial programming: Methods and applications* (pp. 351–360). Dordrecht, The Netherlands: Reidel.

[13] Lau, Kin-nam, Leung, Pui Lam, & Tse, Ka-kit (1998). A nonlinear programming approach to metric unidimensional scaling. *Journal of Classification*, *15*, 3–14.

[14] Pardalos, P. M., & Wolkowicz, H. (Eds.). (1994). *Quadratic assignment and related problems.* DIMACS Series on Discrete Mathematics and Theoretical Computer Science. Providence, RI: American Mathematical Society.

[15] Pliner, V. (1996). Metric unidimensional scaling and global optimization. *Journal of Classification*, *13*, 3–18.

[16] Shepard, R. N. (1974). Representation of structure in similarity data: Problems and prospects. *Psychometrika*, *39*, 373–421.

[17] Shepard, R. N., Kilpatric, D. W., & Cunningham, J. P. (1975). The internal representation of numbers. *Cognitive Psychology*, *7*, 82–138.

Appendix B: The Dykstra-Kaczmarz Method for Solving Linear (In)equality Constrained Least-Squares Tasks

Kaczmarz's method can be characterized as follows:

Given $\mathbf{A} = \{a_{ij}\}$ of order $m \times n$, $\mathbf{x}' = \{x_1, \ldots, x_n\}$, $\mathbf{b}' = \{b_1, \ldots, b_m\}$, and assuming the linear system $\mathbf{Ax} = \mathbf{b}$ is consistent, define the set $C_i = \{\mathbf{x} \mid a_{ij}x_j = b_i\}$, for $1 \leq i \leq m$. The projection of any $n \times 1$ vector $\mathbf{y}$ onto C_i is simply $\mathbf{y} - (\mathbf{a}_i'\mathbf{y} - b_i)\mathbf{a}_i(\mathbf{a}_i'\mathbf{a}_i)^{-1}$, where $\mathbf{a}_i' = \{a_{i1}, \ldots, a_{in}\}$. Beginning with a vector $\mathbf{x}_0$, and successively projecting $\mathbf{x}_0$ onto C_1, and that result onto C_2, and so on, and cyclically and repeatedly reconsidering projections onto the sets $C_1, \ldots, C_m$, leads at convergence to a vector $\mathbf{x}_0^*$ that is closest to $\mathbf{x}_0$ (in vector 2-norm, so $\sum_{i=1}^n (x_{0i} - x_{0i}^*)^2$ is minimized) and $\mathbf{Ax}_0^* = \mathbf{b}$. In short, Kaczmarz's method provides an iterative way to solve least-squares tasks subject to equality restrictions.

Dykstra's method can be characterized as follows:

Given $\mathbf{A} = \{a_{ij}\}$ of order $m \times n$, $\mathbf{x}_0' = \{x_{01}, \ldots, x_{0n}\}$, $\mathbf{b}' = \{b_1, \ldots, b_m\}$, and $\mathbf{w}' = \{w_1, \ldots, w_n\}$, where $w_j > 0$ for all j, find $\mathbf{x}_0^*$ such that $\mathbf{a}_i'\mathbf{x}_0^* \leq b_i$ for $1 \leq i \leq m$ and $\sum_{i=1}^n w_i(x_{0i} - x_{0i}^*)^2$ is minimized. Again, (re)define the (closed convex) sets $C_i = \{\mathbf{x} \mid a_{ij}x_j \leq b_i\}$ and when a vector $\mathbf{y} \notin C_i$, its projection onto C_i (in the metric defined by the weight vector $\mathbf{w}$) is $\mathbf{y} - (\mathbf{a}_i'\mathbf{y} - b_i)\mathbf{a}_i\mathbf{W}^{-1}(\mathbf{a}_i'\mathbf{W}^{-1}\mathbf{a}_i)^{-1}$, where $\mathbf{W}^{-1} = \text{diag}\{w_1^{-1}, \ldots, w_n^{-1}\}$. We again initialize the process with the vector $\mathbf{x}_0$ and each set $C_1, \ldots, C_m$ is considered in turn. If the vector being carried forward to this point when C_i is (re)considered does not satisfy the constraint defining C_i, a projection onto C_i occurs. The sets $C_1, \ldots, C_m$ are cyclically and repeatedly considered but with one difference from the operation of Kaczmarz's method — each time a constraint set C_i is revisited, any changes from the previous time C_i was reached are first "added back". This last process ensures convergence to an optimal solution $\mathbf{x}_0^*$ (see Dykstra, 1983). Thus, Dykstra's method generalizes the equality restrictions that can be handled by Kaczmarz's strategy to the use of inequality constraints.