Unidimensional Scaling: A (Nascent) Toolbox for MATLAB

From the toolbox construction team (listed in alphabetical order):

Lawrence Hubert; Hans-Friedreich Köhn; Douglas Steinley

M-files are available from:

`http://cda.psych.uiuc.edu/unidimensionalscaling_mfiles`

Version: January 26, 2007

Contents

List of Tables

List of Figures

1 Introduction

A broad definition of unidimensional scaling can be given as the search for an arrangement of n objects from a set, say $S = \{O_1, \ldots, O_n\}$, along a single dimension (for linear unidimensional scaling (LUS)), or around a closed circular structure (for circular unidimensional scaling (CUS)), such that the induced $n(n-1)/2$ interpoint distances between the objects reflect the given proximity information. These latter proximities are assumed to be the data available to guide the search and in the form of an $n \times n$ symmetric matrix $\mathbf{P} = \{p_{ij}\}$, where p_{ij} ($= p_{ji} \geq 0$, and $p_{ii} = 0$) is a dissimilarity measure for the objects O_i and O_j in which larger values indicate more dissimilar objects. We begin with presenting such an illustrative data set that can be carried along throughout this chapter for our numerical illustrations. Later sections introduce the basic LUS and CUS tasks and provide a variety of useful extensions and generalizations of each. In all instances, the MATLAB computational environment is relied on to effect our analyses, using the Statistical Toolbox, for example, to carry out some of the common (multi)dimensional scaling methods, and our own open-source MATLAB M-files (freely available as a nascent Toolbox from a web site listed later) whenever the extensions go beyond what is currently available commercially, and/or if the commercial methods fail to provide adequate analysis strategies.

1.1 A Proximity Matrix for Illustrating Unidimensional Scaling: Agreement Among Supreme Court Justices

On Saturday, July 2, 2005, the lead headline in *The New York Times* read as follows: 'O'Connor to Retire, Touching Off Battle Over Court." Opening the story attached to the headline, Richard W. Stevenson wrote, 'Justice Sandra Day O'Connor, the first woman to serve on the United States Supreme Court and a critical swing vote on abortion and a host of other divisive social issues, announced Friday that she is retiring, setting up a tumultuous fight over her successor." Our interests are in the data set also provided by the *Times* that day, quantifying the (dis)agreement among the Supreme Court justices during the decade they had been together. We give this in Table 1 in the form of the percentage of non-unanimous cases in which the justices *dis*agree, from the 1994/95 term through 2003/04 (known as the Rehnquist Court). The dissimilarity matrix (in which larger entries reflect less similar justices) is listed in the same row and column order as the *Times* data set, with the justices obviously ordered from ‘liberal” to ‘conservative”:

1: John Paul Stevens (St)
2: Stephen G. Breyer (Br)
3: Ruth Bader Ginsberg (Gi)
4: David Souter (So)
5: Sandra Day O’Connor (Oc)
6: Anthony M. Kennedy (Ke)
7: William H. Rehnquist (Re)
8: Antonin Scalia (Sc)
9: Clarence Thomas (Th)

We use the Supreme Court data matrix of Table 1 for various illustrations of unidimensional scaling in the sections to follow. It will be loaded into a MATLAB environment with the command ‘`load supreme_agree.dat`’. The `supreme_agree.dat` file is in simple `ascii` form with verbatim contents as follows:

```
.00  .38  .34  .37  .67  .64  .75  .86  .85
.38  .00  .28  .29  .45  .53  .57  .75  .76
.34  .28  .00  .22  .53  .51  .57  .72  .74
.37  .29  .22  .00  .45  .50  .56  .69  .71
.67  .45  .53  .45  .00  .33  .29  .46  .46
.64  .53  .51  .50  .33  .00  .23  .42  .41
.75  .57  .57  .56  .29  .23  .00  .34  .32
.86  .75  .72  .69  .46  .42  .34  .00  .21
.85  .76  .74  .71  .46  .41  .32  .21  .00
```

	St	Br	Gi	So	Oc	Ke	Re	Sc	Th
1 St	.00	.38	.34	.37	.67	.64	.75	.86	.85
2 Br	.38	.00	.28	.29	.45	.53	.57	.75	.76
3 Gi	.34	.28	.00	.22	.53	.51	.57	.72	.74
4 So	.37	.29	.22	.00	.45	.50	.56	.69	.71
5 Oc	.67	.45	.53	.45	.00	.33	.29	.46	.46
6 Ke	.64	.53	.51	.50	.33	.00	.23	.42	.41
7 Re	.75	.57	.57	.56	.29	.23	.00	.34	.32
8 Sc	.86	.75	.72	.69	.46	.42	.34	.00	.21
9 Th	.85	.76	.74	.71	.46	.41	.32	.21	.00

Table 1: Dissimilarities Among Nine Supreme Court Justices.

2 The Basics of Linear Unidimensional Scaling (LUS)

The LUS task can be characterized as arranging the objects in S along a single dimension such that the induced $n(n-1)/2$ interpoint distances between the objects reflect the proximities in $\mathbf{P}$. The most common formalization of this task is through a least-squares criterion and finding n coordinates, $x_1, x_2, \ldots, x_n$, so

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

is minimized. In turn, this optimization suggested by (1) can be rephrased as 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.

The minimization of (2) 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), where

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

for

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

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

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. If ρ^* denotes the permutation that maximizes $\sum_i (t_i^{(\rho)})^2$, then we can let $x_i = t_i^{(\rho^*)}$, with the order induced by $t_1^{(\rho^*)}, \ldots, t_n^{(\rho^*)}$ being consistent with the constraint, $x_1 \leq x_2 \leq \cdots \leq x_n$. In short, the minimization of (2) reduces to the combinatorial optimization of the single term $\sum_i (t_i^{(\rho)})^2$, and where the coordinate estimation is completed as an automatic byproduct.

2.1 Iterative Quadratic Assignment

Because the measure of loss in (2) can be reduced algebraically to

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

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

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

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 permutation ρ^0 to maximize $\sum_i x_i t_i^{(\rho^0)}$. 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 nonnegative 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, discussed extensively in the literature of operations research. 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 $\sum x_i t_i^{(\rho)}$.

2.2 An M-file for Performing LUS Through Iterative QA

To carry out the unidimensional scaling of proximity matrix, we will rely on the M-file, `uniscalqa.m`, downloadable (as are all the other M-files we mention throughout this manual) as open-source code from

`http://cda.psych.uiuc.edu/unidimensionalscaling_mfiles`.

We give the output from a MATLAB session below using the data of Table 1. We note that these Supreme Court data were first written into a text file called `supreme_agree.dat` and placed into the MATLAB workspace with the `load` command. Also, from the help file written as part of the output for `uniscalqa.m` (and which is also given in the Appendix), the proximity input matrix is called `supreme_agree`; we use an equally-spaced target matrix `targlin(9)` (available from the same site that `uniscalqa.m` was obtained); the built-in MATLAB random permutation generator, `randperm(9)`, is invoked for a starting permutation.

```
>> load supreme_agree.dat
>> supreme_agree

supreme_agree =

     0    0.3800    0.3400    0.3700    0.6700    0.6400    0.7500    0.8600    0.8500
0.3800         0    0.2800    0.2900    0.4500    0.5300    0.5700    0.7500    0.7600
0.3400    0.2800         0    0.2200    0.5300    0.5100    0.5700    0.7200    0.7400
0.3700    0.2900    0.2200         0    0.4500    0.5000    0.5600    0.6900    0.7100
0.6700    0.4500    0.5300    0.4500         0    0.3300    0.2900    0.4600    0.4600
0.6400    0.5300    0.5100    0.5000    0.3300         0    0.2300    0.4200    0.4100
```

```
0.7500    0.5700    0.5700    0.5600    0.2900    0.2300         0    0.3400    0.3200
0.8600    0.7500    0.7200    0.6900    0.4600    0.4200    0.3400         0    0.2100
0.8500    0.7600    0.7400    0.7100    0.4600    0.4100    0.3200    0.2100         0

>> help uniscalqa.m

   UNISCALQA carries out a unidimensional scaling of a symmetric
   proximity matrix using iterative quadratic assignment.

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

   PROX is the input proximity matrix (with a zero main diagonal
   and a dissimilarity interpretation);
   TARG is the input target matrix (usually with a zero main
   diagonal and a dissimilarity interpretation representing
   equally spaced locations along a continuum);
   INPERM is the input beginning permutation (a permutation of the
   first $n$ integers). OUTPERM is the final permutation of PROX
   with the cross-product index RAWINDEX
   with respect to TARG redefined as
   $ = \{abs(coord(i) - coord(j))\}$;
   ALLPERMS is a cell array containing INDEX entries corresponding
   to all the permutations identified in the optimization from
   ALLPERMS{1} = INPERM to ALLPERMS{INDEX} = OUTPERM.
   The insertion and rotation routines use from 1 to KBLOCK
   (which is less than or equal to $n-1$) consecutive objects in
   the permutation defining the row and column order of the data
   matrix.  COORD is the set of coordinates of the unidimensional
   scaling in ascending order;
   DIFF is the value of the least-squares loss function for the
   coordinates and object permutation.



>> [outperm, rawindex, allperms, index, coord, diff] = ...
    uniscalqa(supreme_agree, targlin(9), randperm(9), 1);

>> outperm

outperm =

     1     2     3     4     5     6     7     8     9

>> coord

coord =

   -0.5400
   -0.3611
   -0.2967
   -0.2256
    0.0622
    0.1611
    0.2567
    0.4478
    0.4956

>> diff

diff =

    0.4691
```

As might be expected given the *Times* presentation of Table 1 using the order from ‘liberal” to ‘conservative”, the obtained unidimensional scaling was for the identity permutation in `outperm` with the (ordered) coordinates given in `coord` with a least-squares loss of .4691 (in `diff`).

2.3 A Useful Utility for the QA Task Generally

For the QA problem in (5), the attempt to find ρ to maximize $\Gamma(\rho)$, reorganizes the (proximity) matrix as $\mathbf{A}_\rho = \{a_{\rho(i)\rho(j)}\}$, which hopefully shows the same pattern, more or less, as (the fixed target) $\mathbf{B}$; equivalently, we maximize the usual Pearson product-moment correlation between the off-diagonal entries in $\mathbf{B}$ and $\mathbf{A}_\rho$. Another way of rephrasing this search when $\mathbf{B}$ is given by the equally-spaced target matrix, $\{|i-j|\}$, is to say that we seek a permutation ρ that provides a structure ‘close’ as possible to what is called an anti-Robinson (AR) form for $\mathbf{A}_\rho$, i.e., the degree to which the entries in $\mathbf{A}_\rho$, moving away from the main diagonal in either direction never decrease (and usually increase); this is exactly the pattern exhibited by the equally-spaced target matrix $\mathbf{B} = \{|i-j|\}$.

The type of heuristic optimization strategy we use for the QA task in `order.m` implements simple object interchange/rearrangement operations. Based on given matrices $\mathbf{A}$ and $\mathbf{B}$, and beginning with some permutation (possibly chosen at random), local interchanges and 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. Three different classes of local operations are used in the M-file, `order.m`: (i) the pairwise interchanges of objects in the current permutation defining the row and column order of the data matrix $\mathbf{A}$. 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. The procedure then proceeds to (ii): the local operations considered are all reinsertions of from 1 to `kblock` (which is less than n and set by the user) consecutive objects somewhere in the permutation defining the current row and column order of the data matrix. When no further improvement can be made, we move to (iii): the local operations are now all possible rotations (or inversions) of from 2 to `kblock` consecutive objects in the current row/column order of the data matrix. (We suggest a use of `kblock` equal to 3 as a reasonable compromise between the extensiveness of local search, speed of execution, and quality of solution.) The three collections of local changes are revisited (in order) until no alteration is possible in the final permutation obtained.

The use of `order.m` is illustrated in the verbatim recording below on the `supreme_agree` data. There are `index` permutations stored in the MATLAB cell-array `allperms`, from the first randomly generated one in `allperms{1}`, to the found local optimum in `allperms{index}`. (These have been suppressed in the output.) Notice that retrieving entries in a cell-array requires the use of curly braces, `{,}`. The M-file, `targlin.m`, provides the equally-spaced target matrix as an input. Starting with a random permutation and the `supreme_agree` data matrix, the identity permutation is found (in fact, it would be the sole local optimum identified upon repeated starts using random permutations).

```
>> load supreme_agree.dat

>> [outperm,rawindex,allperms,index] = order(supreme_agree,targlin(9),randperm(9),3)

outperm =

     1     2     3     4     5     6     7     8     9


rawindex =

  145.1200


index =

    19
```

3 Confirmatory and Nonmetric 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 these cases, we rely on what can be called the Dykstra-Kaczmarz method for solving linear inequality constrained least-squares problems.

3.1 The Confirmatory Fitting of a Given Order

The M-function, `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 construct 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 but computed on `fit`. In the example below of the use of `linfit.m`, the identity permutation obtained from the use of `uniscalqa.m` is used as the input permutation.

```
>> load supreme_agree.dat
>> [fit,diff,coord] = linfit(supreme_agree,[1 2 3 4 5 6 7 8 9])

fit =

         0    0.1789    0.2433    0.3144    0.6022    0.7011    0.7967    0.9878    1.0356
    0.1789         0    0.0644    0.1356    0.4233    0.5222    0.6178    0.8089    0.8567
    0.2433    0.0644         0    0.0711    0.3589    0.4578    0.5533    0.7444    0.7922
    0.3144    0.1356    0.0711         0    0.2878    0.3867    0.4822    0.6733    0.7211
    0.6022    0.4233    0.3589    0.2878         0    0.0989    0.1944    0.3856    0.4333
    0.7011    0.5222    0.4578    0.3867    0.0989         0    0.0956    0.2867    0.3344
    0.7967    0.6178    0.5533    0.4822    0.1944    0.0956         0    0.1911    0.2389
    0.9878    0.8089    0.7444    0.6733    0.3856    0.2867    0.1911         0    0.0478
    1.0356    0.8567    0.7922    0.7211    0.4333    0.3344    0.2389    0.0478         0
```

```
diff =

    0.4691


coord =

   -0.5400
   -0.3611
   -0.2967
   -0.2256
    0.0622
    0.1611
    0.2567
    0.4478
    0.4956
```

3.2 The Monotonic Transformation of a Proximity Matrix

The function, `proxmon.m`, provides a monotonically transformed proximity matrix that is closest 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 the sum of squared differences between the entries in `fitted` and `monproxpermut` (actually, `diff` is one-half of such a sum because the loss function is over $i < j$).

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 $\{f(p_{\rho^0(i)\rho^0(j)})\}$.

3.2.1 An Application Incorporating proxmon.m

The script M-file listed below gives an application of `proxmon.m` using the identity permutation for our `supreme_agree` matrix. First, `linfit.m` is invoked to obtain a fitted matrix (`fit`); `proxmon.m` then generates the monotonically transformed proximity matrix (`monprox`) with `vaf` = .9869 and `diff` = .0349. The strategy is then repeated one-hundred times (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 normalized. As indicated in the output below, the final `vaf` is .9934 with a `diff` of .0190. (Although the permutation found earlier for `supreme_agree` 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 were repeated with `monprox` 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.)

```
>> type uniscale_monotone_test

load supreme_agree.dat
```

```
inperm = [1 2 3 4 5 6 7 8 9];

proxpermut = supreme_agree(inperm,inperm);

[fit,diff,coord] = linfit(proxpermut,1:9)

[monprox,vaf,diff] = proxmon(proxpermut,fit)

 sumfitsq = sum(sum(fit.^2));

for i = 1:100

    [fit,diff,coord] = linfit(monprox,1:9);

    sumnewfitsq = sum(sum(fit.^2));

    fit = sqrt(sumfitsq)*(fit/sumnewfitsq);

    [monprox,vaf,diff] = proxmon(proxpermut,fit);

end

fit

coord

vaf

diff

monprox

supreme_agree


>> uniscale_monotone_test

fit =

         0    0.1789    0.2433    0.3144    0.6022    0.7011    0.7967    0.9878    1.0356
    0.1789         0    0.0644    0.1356    0.4233    0.5222    0.6178    0.8089    0.8567
    0.2433    0.0644         0    0.0711    0.3589    0.4578    0.5533    0.7444    0.7922
    0.3144    0.1356    0.0711         0    0.2878    0.3867    0.4822    0.6733    0.7211
    0.6022    0.4233    0.3589    0.2878         0    0.0989    0.1944    0.3856    0.4333
    0.7011    0.5222    0.4578    0.3867    0.0989         0    0.0956    0.2867    0.3344
    0.7967    0.6178    0.5533    0.4822    0.1944    0.0956         0    0.1911    0.2389
    0.9878    0.8089    0.7444    0.6733    0.3856    0.2867    0.1911         0    0.0478
    1.0356    0.8567    0.7922    0.7211    0.4333    0.3344    0.2389    0.0478         0


diff =

    0.4691


coord =

   -0.5400
   -0.3611
   -0.2967
   -0.2256
    0.0622
    0.1611
    0.2567
    0.4478
    0.4956


monprox =

         0    0.2467    0.2433    0.2467    0.6517    0.6517    0.7967    1.0117    1.0117
    0.2467         0    0.0800    0.1356    0.4044    0.5022    0.6178    0.8089    0.8567
    0.2433    0.0800         0    0.0711    0.4092    0.4092    0.5533    0.7444    0.7922
    0.2467    0.1356    0.0711         0    0.3030    0.4092    0.5022    0.6733    0.7211
    0.6517    0.4044    0.4092    0.3030         0    0.1774    0.1774    0.4044    0.4092
    0.6517    0.5022    0.4092    0.4092    0.1774         0    0.0800    0.3030    0.3030
    0.7967    0.6178    0.5533    0.5022    0.1774    0.0800         0    0.1911    0.1774
    1.0117    0.8089    0.7444    0.6733    0.4044    0.3030    0.1911         0    0.0478
    1.0117    0.8567    0.7922    0.7211    0.4092    0.3030    0.1774    0.0478         0


vaf =

    0.9869


diff =

    0.0349


fit =

         0    0.2234    0.2463    0.2643    0.6338    0.7235    0.8021    1.0067    1.0067
    0.2234         0    0.0228    0.0409    0.4103    0.5001    0.5787    0.7832    0.7832
    0.2463    0.0228         0    0.0180    0.3875    0.4773    0.5559    0.7604    0.7604
    0.2643    0.0409    0.0180         0    0.3695    0.4592    0.5378    0.7424    0.7424
    0.6338    0.4103    0.3875    0.3695         0    0.0898    0.1684    0.3729    0.3729
    0.7235    0.5001    0.4773    0.4592    0.0898         0    0.0786    0.2831    0.2831
    0.8021    0.5787    0.5559    0.5378    0.1684    0.0786         0    0.2046    0.2046
    1.0067    0.7832    0.7604    0.7424    0.3729    0.2831    0.2046         0         0
    1.0067    0.7832    0.7604    0.7424    0.3729    0.2831    0.2046         0         0


coord =

   -0.1226
   -0.0724
   -0.0672
   -0.0632
    0.0199
    0.0401
    0.0578
    0.1038
    0.1038


vaf =

    0.9934


diff =

    0.0190


monprox =
         0    0.2447    0.2447    0.2447    0.6787    0.6787    0.7927    1.0067    1.0067
    0.2447         0    0.0474    0.0474    0.3854    0.5001    0.5787    0.7832    0.7927
    0.2447    0.0474         0    0.0180    0.4414    0.4414    0.5559    0.7604    0.7604
    0.2447    0.0474    0.0180         0    0.3695    0.4414    0.5378    0.7424    0.7424
    0.6787    0.3854    0.4414    0.3695         0    0.1542    0.1542    0.3854    0.3854
    0.6787    0.5001    0.4414    0.4414    0.1542         0    0.0474    0.2831    0.2831
    0.7927    0.5787    0.5559    0.5378    0.1542    0.0474         0    0.2046    0.1542
    1.0067    0.7832    0.7604    0.7424    0.3854    0.2831    0.2046         0         0
    1.0067    0.7927    0.7604    0.7424    0.3854    0.2831    0.1542         0         0


supreme_agree =

         0    0.3800    0.3400    0.3700    0.6700    0.6400    0.7500    0.8600    0.8500
    0.3800         0    0.2800    0.2900    0.4500    0.5300    0.5700    0.7500    0.7600
    0.3400    0.2800         0    0.2200    0.5300    0.5100    0.5700    0.7200    0.7400
    0.3700    0.2900    0.2200         0    0.4500    0.5000    0.5600    0.6900    0.7100
    0.6700    0.4500    0.5300    0.4500         0    0.3300    0.2900    0.4600    0.4600
    0.6400    0.5300    0.5100    0.5000    0.3300         0    0.2300    0.4200    0.4100
    0.7500    0.5700    0.5700    0.5600    0.2900    0.2300         0    0.3400    0.3200
    0.8600    0.7500    0.7200    0.6900    0.4600    0.4200    0.3400         0    0.2100
    0.8500    0.7600    0.7400    0.7100    0.4600    0.4100    0.3200    0.2100         0

>> monproxvec = squareform(monprox)

monproxvec =

  Columns 1 through 12

    0.2447    0.2447    0.2447    0.6787    0.6787    0.7927    1.0067    1.0067    0.0474    0.0474    0.3854    0.5001

  Columns 13 through 24

    0.5787    0.7832    0.7927    0.0180    0.4414    0.4414    0.5559    0.7604    0.7604    0.3695    0.4414    0.5378

  Columns 25 through 36

    0.7424    0.7424    0.1542    0.1542    0.3854    0.3854    0.0474    0.2831    0.2831    0.2046    0.1542         0

>> supreme_agreevec = squareform(supreme_agree)

supreme_agreevec =

  Columns 1 through 12

    0.3800    0.3400    0.3700    0.6700    0.6400    0.7500    0.8600    0.8500    0.2800    0.2900    0.4500    0.5300

  Columns 13 through 24

    0.5700    0.7500    0.7600    0.2200    0.5300    0.5100    0.5700    0.7200    0.7400    0.4500    0.5000    0.5600

  Columns 25 through 36

    0.6900    0.7100    0.3300    0.2900    0.4600    0.4600    0.2300    0.4200    0.4100    0.3400    0.3200    0.2100

>> fitvec = squareform(fit)

fitvec =

  Columns 1 through 12

    0.2234    0.2463    0.2643    0.6338    0.7235    0.8021    1.0067    1.0067    0.0228    0.0409    0.4103    0.5001
  Columns 13 through 24

    0.5787    0.7832    0.7832    0.0180    0.3875    0.4773    0.5559    0.7604    0.7604    0.3695    0.4592    0.5378

  Columns 25 through 36

    0.7424    0.7424    0.0898    0.1684    0.3729    0.3729    0.0786    0.2831    0.2831    0.2046    0.2046         0

>> plot(supreme_agreevec,monproxvec,'.k')
>> plot(fitvec,monproxvec,'.k')
```

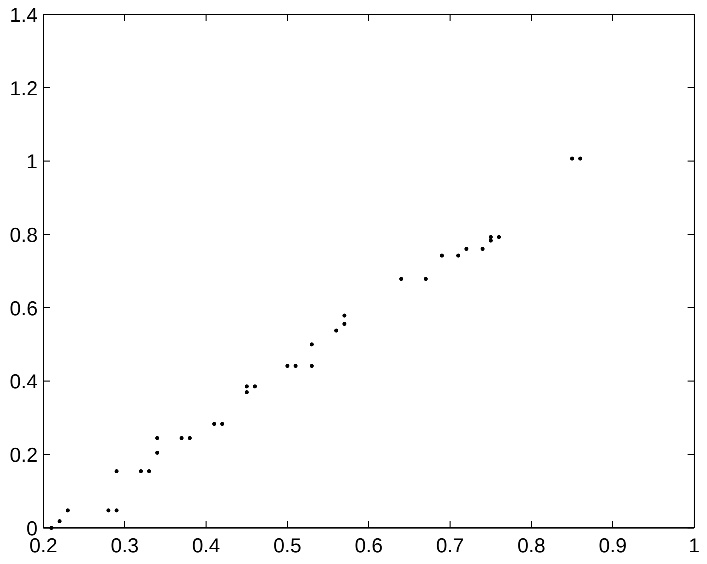


Figure 1: Plot of the Monotonically Transformed Proximities (y-axis) Against the Original Supreme Court Proximities (x-axis).

There are several items to point out about the example just given. First, it was actually run by invoking a script M-file, `uniscale_monotone_test`, the contents of which can be seen by issuing the simple command,

'`type uniscale_monotone_test`'.

The commands are performed by typing the script file name in the command window. We also note that a rather arbitrary number of iterations of the fitting process were carried out (i.e., one-hundred). An alternative strategy would have been to exit upon a minimal change in, say, the VAF value.

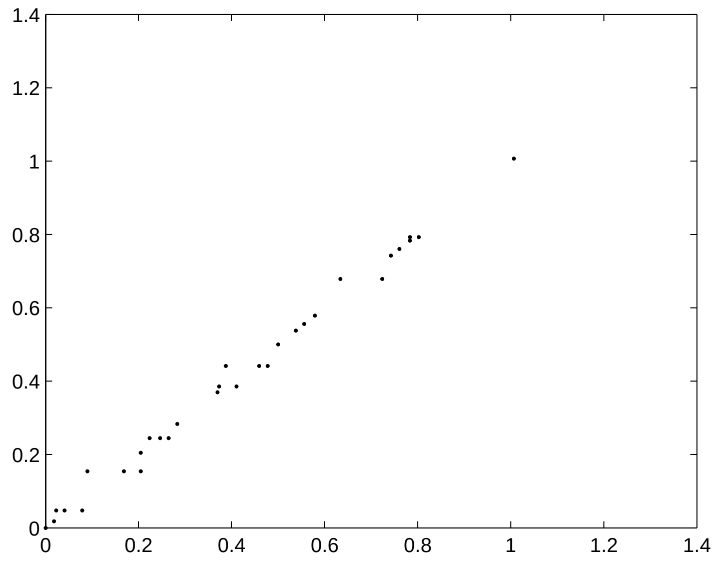


Figure 2: Plot of the Monotonically Transformed Supreme Court Proximities (y-axis) Against the Fitted Values (x-axis).

Second, we show how to plot the entries in the various matrices (`monprox`, `supreme_agree`, and `fit`) by first changing them to vector form through the MATLAB M-function, `squareform.m`. Figures 1 and 2 then show the plot of `monprox` against `supreme_agree` and `fit`. Various editings of these plots could now be done in MATLAB to produce axis labels, legends, and so on.

3.3 Using the MATLAB Statistical Toolbox M-file for Metric and Nonmetric (Multi)dimensional scaling

There is an M-file, `mdscale.m`, within the MATLAB Statistical Toolbox that performs various types of metric and nonmetric multidimensional scaling analyses. When the dimensionality is set at '1', and the loss criterion is set to '`metricstress`', the LUS task in (1) is being solved but with a different type of optimization strategy based on gradients. The criterion reported is `stress`, defined by the square-root of our `diff` divided by the sum-of-squares for the original proximities. As can be seen from the MATLAB session below, the gradient-based method has a very difficult time in finding the best solution defined by the identity permutation, and only two out of one-hundred random starts produced it. (We have suppressed most of the output; the best solution out of the one-hundred is reported automatically, and is identical [given the same coordinates] to that obtained with a single random start of `uniscalqa.m`). It is true generally that gradient-based methods have an extremely hard time avoiding purely local optima when used in one dimension. A reliance on `uniscalqa.m` is a much better option for approaching the LUS task.

```
>> load supreme_agree.dat

>> opts = statset('Display','final','Maxiter',1000);

>> [coord,stress] = mdscale(supreme_agree,1,'Criterion','metricstress','Start','random','Replicates',100,'Options',opts)

6 iterations, Final stress criterion = 0.213021

4 iterations, Final stress criterion = 0.511745

4 iterations, Final stress criterion = 0.213232

4 iterations, Final stress criterion = 0.222266

5 iterations, Final stress criterion = 0.373536

2 iterations, Final stress criterion = 0.607790

2 iterations, Final stress criterion = 0.579076

2 iterations, Final stress criterion = 0.567206

3 iterations, Final stress criterion = 0.459125

2 iterations, Final stress criterion = 0.602695

*remaining starts deleted*


coord =

   -0.5400
   -0.3611
   -0.2967
   -0.2256
    0.0622
    0.1611
    0.2567
    0.4478
    0.4956


stress =

    0.2125
```

3.4 A Convenient Utility for Plotting a LUS Representation

To actually plot a LUS representation, we provide an M-file, `linearplot.m`, with usage syntax

`[linearlength] = linearplot(coord,inperm)`

Here, `linearlength` is the total length of representation from the smallest coordinate to the largest; `coord` is the ordered set of coordinates that get labeled with the values in `inperm`. As can be seen from the output below and Figure 3, the LUS representation separates the far left, {Stevens:1}, from the moderate liberals, {Breyer:2, Ginsberg:3, Souter:4}; and the far right, {Scalia:8, Thomas:9}, from the moderate right, {O'Connor:5, Kennedy:6, Rehnquist:7}.

```
>> [fit,diff,coord] = linfit(supreme_agree,[1 2 3 4 5 6 7 8 9])

fit =

         0    0.1789    0.2433    0.3144    0.6022    0.7011    0.7967    0.9878    1.0356
    0.1789         0    0.0644    0.1356    0.4233    0.5222    0.6178    0.8089    0.8567
    0.2433    0.0644         0    0.0711    0.3589    0.4578    0.5533    0.7444    0.7922
    0.3144    0.1356    0.0711         0    0.2878    0.3867    0.4822    0.6733    0.7211
    0.6022    0.4233    0.3589    0.2878         0    0.0989    0.1944    0.3856    0.4333
    0.7011    0.5222    0.4578    0.3867    0.0989         0    0.0956    0.2867    0.3344
    0.7967    0.6178    0.5533    0.4822    0.1944    0.0956         0    0.1911    0.2389
    0.9878    0.8089    0.7444    0.6733    0.3856    0.2867    0.1911         0    0.0478
    1.0356    0.8567    0.7922    0.7211    0.4333    0.3344    0.2389    0.0478         0


diff =

    0.4691


coord =

   -0.5400
   -0.3611
   -0.2967
   -0.2256
    0.0622
    0.1611
    0.2567
    0.4478
    0.4956

>> [linearlength] = linearplot(coord,[1 2 3 4 5 6 7 8 9])

linearlength =

    1.0356
```

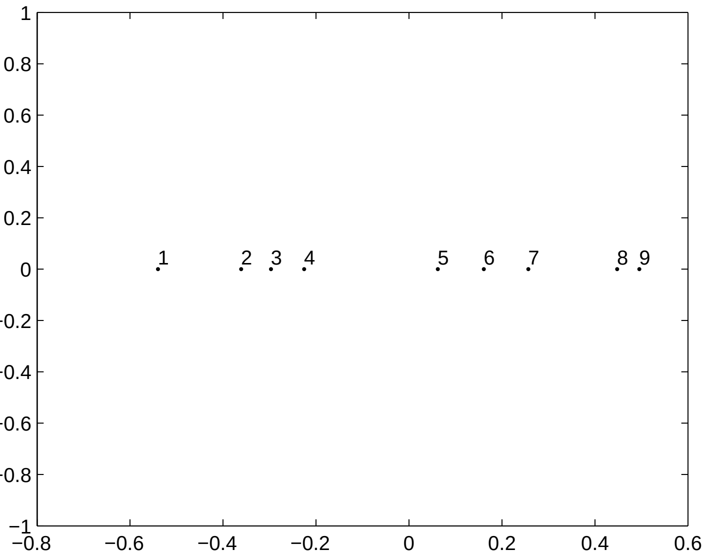


Figure 3: The LUS Representation Using linearplot.m with the Coordinates Obtained from linfit.m on the supreme_agree Proximities.

4 Incorporating an Additive Constant in LUS

A generalization to the basic LUS task that incorporates an additional additive constant will prove extremely convenient when extensions to multiple unidimensional scales are proposed. In this section we emphasize a single LUS structure through the more general least-squares loss function of the form

$$\sum_{i<j} (p_{ij} - \{|x_j - x_i| - c\})^2, \tag{6}$$

where c is some constant to be estimated along with the coordinates $x_1, \ldots, x_n$. Much later, the restriction to fitting only a single unidimensional structure to a symmetric proximity matrix is removed; the latter will rely heavily on a computational approach that includes the augmentation by an estimated additive constant and a procedure of successive residualization of the original proximity matrix. For example, the fitting of two LUS structures to a proximity matrix $\{p_{ij}\}$ could be rephrased as the minimization of a loss function generalizing (6) to the form

$$\sum_{i<j}(p_{ij} - [|x_{j1} - x_{i1}| - c_1] - [|x_{j2} - x_{i2}| - c_2])^2. \tag{7}$$

The attempt to minimize (7) could proceed with the fitting of a single LUS structure to $\{p_{ij}\}$, $[|x_{j1} - x_{i1}| - c_1]$, and once obtained, fitting a second LUS structure, $[|x_{j2} - x_{i2}| - c_2]$, to the residual matrix, $\{p_{ij} - [|x_{j1} - x_{i1}| - c_1]\}$. The process would then cycle by repetitively fitting the residuals from the second linear structure by the first, and the residuals from the first linear structure by the second, until the sequence converges. In any case, obvious extensions would also exist for the inclusion of more than two LUS structures.

The explicit inclusion of two constants, c_1 and c_2, in (7) rather than adding these two together and including a single additive constant, c, deserves some additional introductory explanation. As would be the case in fitting a single LUS structure using the loss function in (6), two interpretations exist for the role of the additive constant, c. We could consider $\{|x_j - x_i|\}$ to be fitted to the translated proximities $\{p_{ij} + c\}$, or alternatively, $\{|x_j - x_i| - c\}$ to be fitted to the original proximities $\{p_{ij}\}$, where the constant c becomes part of the actual model. Although these two interpretations do not lead to any algorithmic differences in how we would proceed with minimizing the loss function in (6), a consistent use of the second interpretation suggests that we frame extensions to the use of multiple LUS structures as we did in (7), where it is explicit that the constants c_1 and c_2 are part of the actual models to be fitted to the (untransformed) proximities $\{p_{ij}\}$. Once c_1 and c_2 are obtained, they could be summed as $c = c_1 + c_2$, and an interpretation made that we have attempted to fit a transformed set of proximities $\{p_{ij} + c\}$ by the sum $\{|x_{j1} - x_{i1}| + |x_{j2} - x_{i2}|\}$ (and in this latter case, a more usual terminology would be one of a two-dimensional scaling (MDS) based on the city-block distance function). However, such a further interpretation is unnecessary and could lead to at least some small terminological confusion in further extensions that we might wish to pursue. For instance, if some type of (optimal nonlinear) transformation, say $f(\cdot)$, of the proximities is also sought (e.g., a monotonic function of some form as we did in Section 3.2), in addition to fitting multiple LUS structures, and where p_{ij} in (7) is replaced by $f(p_{ij})$, and $f(\cdot)$ is to be constructed, the first interpretation would require the use of a 'doubly transformed' set of proximities $\{f(p_{ij}) + c\}$ to be fitted by the sum $\{|x_{j1} - x_{i1}| + |x_{j2} - x_{i2}|\}$. In general, it seems best to avoid the need to incorporate the notion of a double transformation in this context, and instead merely consider the constants c_1 and c_2 to be part of the models being fitted to a transformed set of proximities $f(p_{ij})$.

4.1 The Incorporation of an Additive Constant in LUS Through the M-file linfitac.m

We present and illustrate an M-function, `linfitac.m`, that fits a given single unidimensional scale (by providing the coordinates $x_1, \ldots, x_n$) and the additive constant (c) for some fixed input object ordering along the continuum defined by a permutation $\rho^{(0)}$. This approach directly parallels the M-function given earlier as `linfit.m`, but now with an included additive constant estimation. The usage syntax of

```
[fit,vaf,coord,addcon] = linfitac(prox,inperm)
```

is similar to that of `linfit.m` except for the inclusion (as output) of the additive constant `addcon`, and the replacement of the least-squares criterion of `diff` by the variance-accounted-for (`vaf`) given by the general formula

$$\mathrm{VAF} = 1 - \frac{\sum_{i<j}(p_{\rho^{(0)}(i)\rho^{(0)}(j)} + c - |x_j - x_i|)^2}{\sum_{i<j}(p_{ij} - \bar{p})^2},$$

where $\bar{p}$ is the mean of the proximity values under consideration.

To illustrate the invariance of `vaf` to the use of linear transformations of the proximity matrix (although `coord` and `addcon` obviously will change depending on the transformation used), the identity permutation was fitted using two different matrices: the original proximity matrix for `supreme_agree`, and one standardized to mean zero and variance one. The latter matrix is obtained with the utility `proxstd.m`, with usage explained in its M-file header comments given in the Appendix. Note that for the two proximity matrices employed, the VAF values are exactly the same (.9796) but the coordinates and additive constants differ; a listing of the standardized proximity matrix is given in the output to show explicitly how negative proximities pose no problem for the fitting process that allows the incorporation of additive constants within the fitted model.

```
>> load supreme_agree.dat
>> inperm = [1 2 3 4 5 6 7 8 9];
>> [fit,vaf,coord,addcon] = linfitac(supreme_agree,inperm)

fit =

         0    0.1304    0.1464    0.1691    0.4085    0.4589    0.5060    0.6483    0.6483
    0.1304         0    0.0160    0.0387    0.2780    0.3285    0.3756    0.5179    0.5179
    0.1464    0.0160         0    0.0227    0.2620    0.3124    0.3596    0.5019    0.5019
    0.1691    0.0387    0.0227         0    0.2393    0.2898    0.3369    0.4792    0.4792
    0.4085    0.2780    0.2620    0.2393         0    0.0504    0.0976    0.2399    0.2399
    0.4589    0.3285    0.3124    0.2898    0.0504         0    0.0471    0.1894    0.1894
    0.5060    0.3756    0.3596    0.3369    0.0976    0.0471         0    0.1423    0.1423
    0.6483    0.5179    0.5019    0.4792    0.2399    0.1894    0.1423         0         0
    0.6483    0.5179    0.5019    0.4792    0.2399    0.1894    0.1423         0         0


vaf =

    0.9796


coord =

   -0.3462
   -0.2158
   -0.1998
   -0.1771
    0.0622
    0.1127
    0.1598
    0.3021
    0.3021


addcon =

   -0.2180

>> supreme_agree_stan = proxstd(supreme_agree,0.0)

supreme_agree_stan =

         0   -0.6726   -0.8887   -0.7266    0.8948    0.7326    1.3271    1.9216    1.8676
   -0.6726         0   -1.2130   -1.1590   -0.2942    0.1381    0.3543    1.3271    1.3812
   -0.8887   -1.2130         0   -1.5373    0.1381    0.0300    0.3543    1.1650    1.2731
   -0.7266   -1.1590   -1.5373         0   -0.2942   -0.0240    0.3003    1.0028    1.1109
    0.8948   -0.2942    0.1381   -0.2942         0   -0.9428   -1.1590   -0.2402   -0.2402
    0.7326    0.1381    0.0300   -0.0240   -0.9428         0   -1.4832   -0.4564   -0.5104
    1.3271    0.3543    0.3543    0.3003   -1.1590   -1.4832         0   -0.8887   -0.9968
    1.9216    1.3271    1.1650    1.0028   -0.2402   -0.4564   -0.8887         0   -1.5913
    1.8676    1.3812    1.2731    1.1109   -0.2402   -0.5104   -0.9968   -1.5913         0

>> [fit,vaf,coord,addcon] = linfitac(supreme_agree_stan,inperm)

fit =

         0    0.7050    0.7914    0.9139    2.2073    2.4799    2.7345    3.5037    3.5037
    0.7050         0    0.0864    0.2089    1.5024    1.7750    2.0295    2.7987    2.7987
    0.7914    0.0864         0    0.1225    1.4159    1.6885    1.9431    2.7123    2.7123
    0.9139    0.2089    0.1225         0    1.2935    1.5661    1.8206    2.5898    2.5898
    2.2073    1.5024    1.4159    1.2935         0    0.2726    0.5272    1.2964    1.2964
    2.4799    1.7750    1.6885    1.5661    0.2726         0    0.2546    1.0238    1.0238
    2.7345    2.0295    1.9431    1.8206    0.5272    0.2546         0    0.7692    0.7692
    3.5037    2.7987    2.7123    2.5898    1.2964    1.0238    0.7692         0         0
    3.5037    2.7987    2.7123    2.5898    1.2964    1.0238    0.7692         0         0


vaf =

    0.9796


coord =

   -1.8710
   -1.1661
   -1.0796
   -0.9572
    0.3363
    0.6089
    0.8635
    1.6327
    1.6327


addcon =

    1.5480
```

5 Circular Unidimensional Scaling (CUS)

Circular unidimensional scaling (CUS) has the objective of placing n objects around a closed continuum such that the reconstructed distance between each pair of objects, defined by the minimum length over the two possible paths that join the objects, reflects the given proximities as well as possible. Explicitly, and in analogy with the loss function for linear unidimensional scaling (LUS), we wish to find a set of coordinates, $x_1, \ldots, x_n$, plus an $(n+1)^{st}$ value, $x_0 \geq |x_j - x_i|$ for all $1 \leq i \neq j \leq n$, minimizing

$$\sum_{i<j} (p_{ij} + c - \min\{|x_j - x_i|, x_0 - |x_j - x_i|\})^2, \tag{8}$$

or equivalently,

$$\sum_{i<j} (p_{ij} - [\min\{|x_j - x_i|, x_0 - |x_j - x_i|\} - c])^2, \tag{9}$$

where c is again some constant to be estimated. The value x_0 represents the total length of the closed continuum, and the expression, $\min\{|x_j - x_i|, x_0 -$

$|x_j - x_i|\}$, gives the minimum length over the two possible paths joining objects O_i and O_j.

Because the `supreme_agree` proximity matrix is so well-represented by LUS, we use a different data set here for illustration of CUS, given in the form of a rather well-known proximity matrix in Table 2 (and called '`morse_digits`'). The later is a 10×10 proximity matrix for the ten Morse Code symbols that represent the first ten digits: (0: $- - - - -$; 1: $\bullet - - - -$; 2: $\bullet \bullet - - -$; 3: $\bullet \bullet \bullet - -$; 4: $\bullet \bullet \bullet \bullet -$; 5: $\bullet \bullet \bullet \bullet \bullet$; 6: $- \bullet \bullet \bullet \bullet$; 7: $- - \bullet \bullet \bullet$; 8: $- - - \bullet \bullet$; 9: $- - - - \bullet$). (Note that the labeling of objects in the output is from 1 to 10; thus, a translation back to the actual numbers corresponding to the Morse Code symbols requires a subtraction of one.) The entries in Table 2 have a dissimilarity interpretation and are defined for each object pair by 2.0 minus the sum of the two proportions for a group of subjects used by Rothkopf in the 1950's, representing 'same" judgments to the two symbols when given in the two possible presentation orders of the signals. Based on previous multidimensional scalings of the complete data set involving all of the Morse code symbols and in which the data of Table 2 are embedded, it might be expected that the symbols for the digits would form a clear linear unidimensional structure that would be interpretable according to a regular progression in the number of dots to dashes. It turns out, as discussed in greater detail below, that a circular model is probably more consistent with the patterning of the proximities in Table 2 than are representations based on linear unidimensional scalings.

5.1 The Circular Unidimensional Scaling Utilities

The two circular unidimensional scaling utilities that implement the mechanics of fitting the CUS model, parallel the LUS utilities of `linfit.m` and `linfitac.m`. The M-files, `cirfit.m` and `cirfitac.m`, carry out confirmatory fittings of a given order (assumed to be an object ordering around a closed unidimensional structure), and have syntax:

```
[fit, diff] = cirfit(prox,inperm)
```

```
[fit,vaf,addcon] = cirfitac(prox,inperm)
```

Table 2: A Proximity Matrix, morse_digits, for the Ten Morse Code Symbols Representing the First Ten Digits.

0.00	.75	1.69	1.87	1.76	1.77	1.59	1.26	.86	.95
.75	0.00	.82	1.54	1.85	1.72	1.51	1.50	1.45	1.63
1.69	.82	0.00	1.25	1.47	1.33	1.66	1.57	1.83	1.81
1.87	1.54	1.25	0.00	.89	1.32	1.53	1.74	1.85	1.86
1.76	1.85	1.47	.89	0.00	1.41	1.64	1.81	1.90	1.90
1.77	1.72	1.33	1.32	1.41	0.00	.70	1.56	1.84	1.64
1.59	1.51	1.66	1.53	1.64	.70	0.00	.70	1.38	1.70
1.26	1.50	1.57	1.74	1.81	1.56	.70	0.00	.83	1.22
.86	1.45	1.83	1.85	1.90	1.84	1.38	.83	0.00	.41
.95	1.63	1.81	1.86	1.90	1.64	1.70	1.22	.41	0.00

where `inperm` is the given order; `fit` is an $n \times n$ matrix fitted to the matrix `prox(inperm,inperm)` with a least-squares value `diff`. The syntax for the routine, `cirfitac.m`, is the same except for the inclusion of an additive constant, `addcon`, and the use of `vaf` rather than `diff`.

In brief, then, the type of matrix being fitted to the proximity matrix has the form

$$\{\min(\mid x_{\rho(j)} - x_{\rho(i)} \mid,\ x_0 - \mid x_{\rho(j)} - x_{\rho(i)} \mid)\ - c\},$$

where c is an estimated additive constant (assumed equal to zero in `cirfit.m`), $x_{\rho(1)} \leq x_{\rho(2)} \leq \cdots \leq x_{\rho(n)} \leq x_0$, and the last coordinate, x_0, is the circumference of the circular structure. We can obtain these latter coordinates from the adjacent spacings in the output matrix `fit`. As an example, we applied `cirfit.m` to the `morse_digits` proximity matrix with an assumed identity input permutation; the spacings around the circular structure between the placements for objects 1 and 2 is .5337; 2 and 3: .7534; 3 and 4: .6174; 4 and 5: .1840; 5 and 6: .5747; 6 and 7: .5167; 7 and 8: .3920; 8 and 9: .5467; 9 and 10: .1090; and back around between 10 and 1: .5594 (the sum of all these adjacent spacings is 4.787 and is the circumference (x_0) of the circular structure). For `cirfitac.m` the additive constant was estimated as -.8031 with a `vaf` of .7051; here, the spacings around the circular structure between the placements for objects 1 and 2 is .2928; 2 and 3: .4322; 3 and 4: .2962; 4 and 5: .0234; 5 and 6: .3338; 6 and 7: .2758; 7 and 8: .2314; 8 and 9: .2800;

9 and 10: .0000; and back around between 10 and 1: .2124 (here, x_0 has a value of 2.378).

```
>> load morse_digits.dat
>> morse_digits

morse_digits =

         0    0.7500    1.6900    1.8700    1.7600    1.7700    1.5900    1.2600    0.8600    0.9500
    0.7500         0    0.8200    1.5400    1.8500    1.7200    1.5100    1.5000    1.4500    1.6300
    1.6900    0.8200         0    1.2500    1.4700    1.3300    1.6600    1.5700    1.8300    1.8100
    1.8700    1.5400    1.2500         0    0.8900    1.3200    1.5300    1.7400    1.8500    1.8600
    1.7600    1.8500    1.4700    0.8900         0    1.4100    1.6400    1.8100    1.9000    1.9000
    1.7700    1.7200    1.3300    1.3200    1.4100         0    0.7000    1.5600    1.8400    1.6400
    1.5900    1.5100    1.6600    1.5300    1.6400    0.7000         0    0.7000    1.3800    1.7000
    1.2600    1.5000    1.5700    1.7400    1.8100    1.5600    0.7000         0    0.8300    1.2200
    0.8600    1.4500    1.8300    1.8500    1.9000    1.8400    1.3800    0.8300         0    0.4100
    0.9500    1.6300    1.8100    1.8600    1.9000    1.6400    1.7000    1.2200    0.4100         0

>> [fit,diff] = cirfit(morse_digits,1:10)

fit =

         0    0.5337    1.2871    1.9044    2.0884    2.1237    1.6071    1.2151    0.6684    0.5594
    0.5337         0    0.7534    1.3707    1.5547    2.1294    2.1407    1.7487    1.2021    1.0931
    1.2871    0.7534         0    0.6174    0.8014    1.3761    1.8927    2.2847    1.9554    1.8464
    1.9044    1.3707    0.6174         0    0.1840    0.7587    1.2754    1.6674    2.2141    2.3231
    2.0884    1.5547    0.8014    0.1840         0    0.5747    1.0914    1.4834    2.0301    2.1391
    2.1237    2.1294    1.3761    0.7587    0.5747         0    0.5167    0.9087    1.4554    1.5644
    1.6071    2.1407    1.8927    1.2754    1.0914    0.5167         0    0.3920    0.9387    1.0477
    1.2151    1.7487    2.2847    1.6674    1.4834    0.9087    0.3920         0    0.5467    0.6557
    0.6684    1.2021    1.9554    2.2141    2.0301    1.4554    0.9387    0.5467         0    0.1090
    0.5594    1.0931    1.8464    2.3231    2.1391    1.5644    1.0477    0.6557    0.1090         0


diff =

    7.3898

>> [fit,vaf,addcon] = cirfitac(morse_digits,1:10)

fit =

         0    0.2928    0.7250    1.0212    1.0446    0.9996    0.7238    0.4924    0.2124    0.2124
    0.2928         0    0.4322    0.7284    0.7518    1.0856    1.0166    0.7852    0.5052    0.5052
    0.7250    0.4322         0    0.2962    0.3196    0.6534    0.9292    1.1606    0.9374    0.9374
    1.0212    0.7284    0.2962         0    0.0234    0.3572    0.6330    0.8644    1.1444    1.1444
    1.0446    0.7518    0.3196    0.0234         0    0.3338    0.6096    0.8410    1.1210    1.1210
    0.9996    1.0856    0.6534    0.3572    0.3338         0    0.2758    0.5072    0.7872    0.7872
    0.7238    1.0166    0.9292    0.6330    0.6096    0.2758         0    0.2314    0.5114    0.5114
    0.4924    0.7852    1.1606    0.8644    0.8410    0.5072    0.2314         0    0.2800    0.2800
    0.2124    0.5052    0.9374    1.1444    1.1210    0.7872    0.5114    0.2800         0    0.0000
    0.2124    0.5052    0.9374    1.1444    1.1210    0.7872    0.5114    0.2800    0.0000         0


vaf =

    0.7051


addcon =

   -0.8031
```

5.1.1 The M-function unicirac.m

The function M-file, `unicirac.m`, carries out a circular unidimensional scaling of a symmetric dissimilarity matrix (with the estimation of an additive constant) using an iterative quadratic assignment strategy (and thus, can be viewed an analogue of `uniscalqa.m` for the LUS task). We begin with an equally-spaced circular target constructed using the M-file `targcir.m` (that could be invoked with the command `targcir(10)`), a (random) starting permutation, and then 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

```
[find,vaf,outperm,addcon] = unicirac(prox,inperm,kblock)
```

where the various terms should now be familiar. The given starting permutation, `inperm`, is of the first n integers (assumed to be around the circle); `find` is the least-squares optimal matrix (with variance-accounted-for of `vaf`) to `prox` having the appropriate circular form for the row and column object ordering given by the final permutation, `outperm`. The spacings between the objects are given by the entries immediately above the main diagonal in `find` (and the extreme $(1,n)$ entry in `find`). The block size in the use the iterative quadratic assignment routine is `kblock`; the additive constant for the model is given by `addcon`.

The problem of local optima is much more severe in CUS than in LUS. Given the heuristic identification of inflection points (i.e., the clock- or counterclockwise change of direction for the calculation of distances between object pairs), the relevant spacings can vary somewhat depending on the 'equivalent' orderings identified around a circular structure. The example given below was identified as the best achievable (and for some multiple number of times) over 100 random starting permutations for `inperm`; with its `vaf` of 71.90%, it is apparently the best attainable. Given the (equivalent to the) identity permutation identified for `outperm`, the substantive interpretation for this representation is fairly clear — we have a nicely interpretable ordering of the Morse code symbols around a circular structure involving a regular replacement of dashes by dots moving clockwise until the symbol containing all dots is reached, and then a subsequent replacement of the dots by dashes until the initial symbol containing all dashes is reached.

```
>> [find,vaf,outperm,addcon] = unicirac(morse_digits,randperm(10),2)

find =

         0    0.0247    0.3620    0.6413    0.9605    1.1581    1.1581    1.0358    0.7396    0.3883
    0.0247         0    0.3373    0.6165    0.9358    1.1334    1.1334    1.0606    0.7643    0.4131
    0.3620    0.3373         0    0.2793    0.5985    0.7961    0.7961    1.0148    1.1016    0.7503
    0.6413    0.6165    0.2793         0    0.3193    0.5169    0.5169    0.7355    1.0318    1.0296
    0.9605    0.9358    0.5985    0.3193         0    0.1976    0.1976    0.4163    0.7125    1.0638
    1.1581    1.1334    0.7961    0.5169    0.1976         0    0.0000    0.2187    0.5149    0.8662
    1.1581    1.1334    0.7961    0.5169    0.1976    0.0000         0    0.2187    0.5149    0.8662
    1.0358    1.0606    1.0148    0.7355    0.4163    0.2187    0.2187         0    0.2963    0.6475
    0.7396    0.7643    1.1016    1.0318    0.7125    0.5149    0.5149    0.2963         0    0.3513
    0.3883    0.4131    0.7503    1.0296    1.0638    0.8662    0.8662    0.6475    0.3513         0


vaf =

    0.7190

outperm =

     4     5     6     7     8     9    10     1     2     3


addcon =

   -0.7964
```

The plotting function circularplot.m

To assist in the visualization of the results from a circular unidimensional scaling, the M-function called `circularplot.m`, provides the coordinates of a scaling around a circular structure plus a plot of the (labeled) objects around the circle. The usage syntax is

```
[circum,radius,coord,degrees,cumdegrees] = ...
  circularplot(circ,inperm)
```

The coordinates are derived from the $n \times n$ interpoint distance matrix (around a circle) given by `circ`; the positions are labeled by the order of objects given in `inperm`. The output consists of a plot, the circumference of the circle (`circum`) and radius (`radius`); the coordinates of the plot positions (`coord`), and the degrees and cumulative degrees induced between the plotted positions (in `degrees` and `cumdegrees`). The positions around the circle are numbered from 1 (at the ‘noon’ position) to n, moving clockwise around the circular structure.

As an example, Figure 4 provides an application of `circularplot.m` to the just given example of `unicirac.m`. The text output also appears below:

```
>> [circum,radius,coord,degrees,cumdegrees] = circularplot(find,outperm)

circum =

    2.4126


radius =

    0.3840


coord =

         0    0.3840
    0.0247    0.3832
    0.3107    0.2256
    0.3821   -0.0380
    0.2293   -0.3080
    0.0481   -0.3810
    0.0481   -0.3810
   -0.1649   -0.3468
   -0.3600   -0.1336
   -0.3254    0.2038


degrees =

    0.0644
    0.8783
    0.7273
    0.8315
    0.5146
    0.0000
    0.5695
    0.7716
    0.9148
    1.0113


cumdegrees =

    0.0644
    0.9428
    1.6700
    2.5015
    3.0161
    3.0161
    3.5856
    4.3571
    5.2719
    6.2832
```

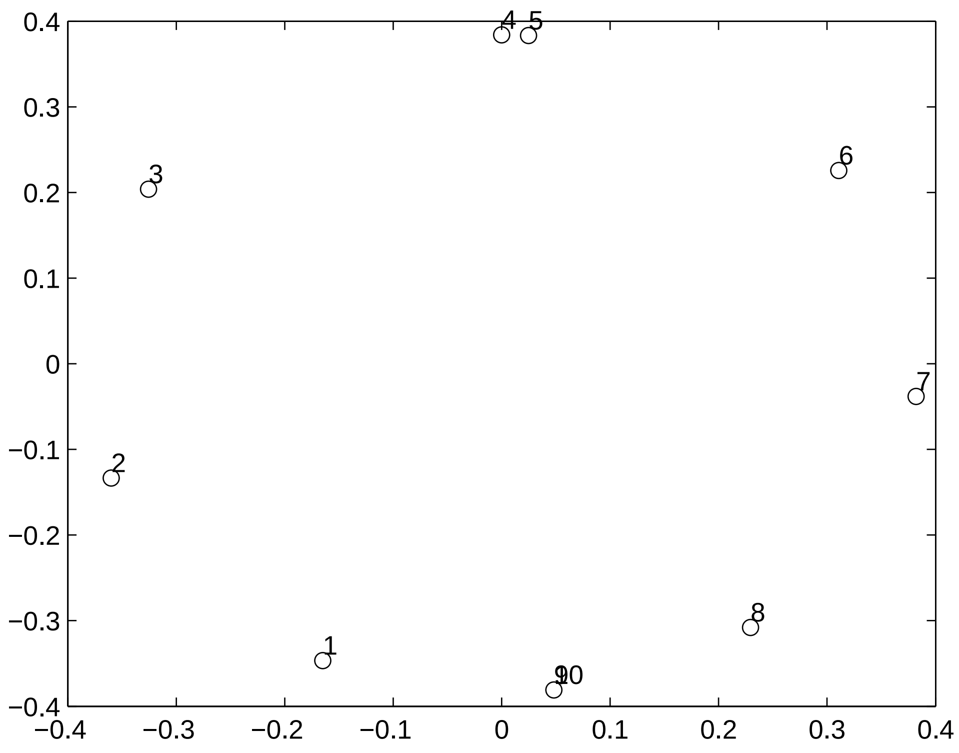


Figure 4: Two-dimensional Circular Plot for the morse_digits Data Obtained Using circularplot.m.

5.1.2 Using unicirac.m on the Supreme Court Proximity Matrix

To illustrate the use of CUS when a data set is well-represented by LUS, as is `supreme_agree`, a verbatim output is provided below. This shows the common occurrence of a very large spacing constructed between the first and last justices, and one that is much larger than the others. The CUS model tries, more-or-less, to mimic a LUS model as best it can.

```
>> [find,vaf,outperm,addcon] = unicirac(supreme_agree,randperm(9),3)

find =

         0    0.0837    0.6183    0.6183    0.3822    0.3374    0.2893    0.0421    0.0421
    0.0837         0    0.5346    0.5346    0.4658    0.4211    0.3730    0.1258    0.1258
    0.6183    0.5346         0   -0.0000    0.1349    0.1797    0.2278    0.4749    0.4749
    0.6183    0.5346   -0.0000         0    0.1349    0.1797    0.2278    0.4749    0.4749
    0.3822    0.4658    0.1349    0.1349         0    0.0448    0.0929    0.3400    0.3400
    0.3374    0.4211    0.1797    0.1797    0.0448         0    0.0481    0.2953    0.2953
    0.2893    0.3730    0.2278    0.2278    0.0929    0.0481         0    0.2472    0.2472
    0.0421    0.1258    0.4749    0.4749    0.3400    0.2953    0.2472         0    0.0000
    0.0421    0.1258    0.4749    0.4749    0.3400    0.2953    0.2472    0.0000         0


vaf =

    0.9434


outperm =

     2     1     9     8     7     6     5     3     4


addcon =

   -0.2286
```

5.1.3 Using uniscalqa.m on the Morse Code Proximity Matrix

To now illustrate the use of a LUS model when the data are nicely interpretable with a CUS structure, as is `morse_digits` (displayed in Figure 4), we show the application on the latter data matrix of both `uniscalqa.m` and `linfitac.m`. The `vaf` for the constructed LUS structure is 58.53% compared with 70.51% for the CUS result given earlier. The output permutation of [10 9 1 8 2 7 6 3 4 5], however, now simply represents a projection of the symbols on a vertical axis imposed on the circular plot of Figure 4. In other words, we lose the nice circular interpretation of the data – a not uncommon result for this type of proximity matrix.

```
>> load morse_digits.dat
>> [outperm,rawindex,allperms,index,coord,diff] = uniscalqa(morse_digits,targlin(10),randperm(10),2)

outperm =

    10     9     1     8     2     7     6     3     4     5


rawindex =

  174.3040


index =

    14


coord =

   -1.3120
   -1.1530
   -0.8880
   -0.5570
   -0.2110
    0.1350
    0.5170
    0.7990
    1.2070
    1.4630


diff =

   14.3915

>> [fit,vaf,coord,addcon] = linfitac(morse_digits,outperm)

fit =

         0         0    0.0511    0.1865    0.3370    0.4874    0.6738    0.7602    0.9726    1.0330
         0         0    0.0511    0.1865    0.3370    0.4874    0.6738    0.7602    0.9726    1.0330
    0.0511    0.0511         0    0.1354    0.2858    0.4362    0.6227    0.7091    0.9215    0.9819
    0.1865    0.1865    0.1354         0    0.1504    0.3008    0.4872    0.5737    0.7861    0.8465
    0.3370    0.3370    0.2858    0.1504         0    0.1504    0.3368    0.4232    0.6357    0.6961
    0.4874    0.4874    0.4362    0.3008    0.1504         0    0.1864    0.2728    0.4852    0.5457
    0.6738    0.6738    0.6227    0.4872    0.3368    0.1864         0    0.0864    0.2988    0.3592
    0.7602    0.7602    0.7091    0.5737    0.4232    0.2728    0.0864         0    0.2124    0.2728
    0.9726    0.9726    0.9215    0.7861    0.6357    0.4852    0.2988    0.2124         0    0.0604
    1.0330    1.0330    0.9819    0.8465    0.6961    0.5457    0.3592    0.2728    0.0604         0


vaf =

    0.5853


coord =

   -0.4502
   -0.4502
   -0.3990
   -0.2636
   -0.1132
    0.0372
    0.2236
    0.3100
    0.5225
    0.5829


addcon =

   -0.9779
```

6 LUS for Two-Mode (Rectangular) Proximity Data

The proximity data considered thus far for obtaining some type of structure, such as a LUS or CUS, have been assumed to be on one intact set of objects, $S = \{O_1, \ldots, O_n\}$, and complete in the sense that proximity values are present between all object pairs. Suppose now that the available proximity data are two-mode, and between two distinct object sets, $S_A = \{O_{1A}, \ldots, O_{n_a A}\}$ and $S_B = \{O_{1B}, \ldots, O_{n_b B}\}$, containing n_a and n_b objects, respectively, given by an $n_a \times n_b$ proximity matrix $\mathbf{Q} = \{q_{rs}\}$. Again, we assume that the entries in $\mathbf{Q}$ are keyed as dissimilarities, and a joint structural representation is desired for the combined set $S_A \cup S_B$. We might caution at the outset of the need to have legitimate proximities to make the analyses to follow very worthwhile or interpretable. There are many numerical elicitation schemes where subjects (e.g., raters) are asked to respond to some set of objects (e.g., items). If the elicitation is for, say, preference, then proximity may be a good interpretation for the numerical values. If, on the other hand, the numerical value is merely a rating given on some more-or-less objective criterion where only errors of observation induce the variability from rater to rater, then probably not.

To have an example of a two-mode data set that might be used in our illustrations, we extracted a 5×4 section from our `supreme_agree` proximity matrix. The five rows correspond to the judges, {St,Gi,Oc,Re,Th}; the four columns to {Br,So,Ke,Sc}; the corresponding file, `supreme_agree5x4.dat`, has contents:

```
    0.3000     0.3700     0.6400     0.8600
    0.2800     0.2200     0.5100     0.7200
    0.4500     0.4500     0.3300     0.4600
    0.5700     0.5600     0.2300     0.3400
    0.7600     0.7100     0.4100     0.2100
```

Because of the way the joint set of row and columns objects is numbered, the five rows are labeled from 1 to 5 and the four columns from 6 to 9. Thus, the correspondence between the justices and numbers differs from earlier applications: 1:St; 2:Gi; 3:Oc; 4:Re; 5:Th; 6:Br; 7:So; 8:Ke; 9:Sc

6.1 Reordering Two-Mode Proximity Matrices

Given an $n_a \times n_b$ two-mode proximity matrix, $\mathbf{Q}$, defined between the two distinct sets, S_A and S_B, it may be desirable to reorder separately the rows and columns of $\mathbf{Q}$ to display some type of pattern that may be present in its entries, or to obtain some joint permutation of the n $(= n_a + n_b)$ row and column objects to effect some further type of simplified representation. These kinds of reordering tasks will be approached with a variant of the quadratic assignment heuristics of the earlier LUS discussion applied to a square, $(n_a + n_b) \times (n_a + n_b)$, proximity matrix, $\mathbf{P}^{(tm)}$, in which a two-mode matrix $\mathbf{Q}_{(dev)}$ and its transpose (where $\mathbf{Q}_{(dev)}$ is constructed from $\mathbf{Q}$ by deviating its entries from the mean proximity), form the upper-right- and lower-left-hand portions, respectively, with zeros placed elsewhere. (This use of zero in the presence of deviated proximities, appears a reasonable choice generally in identifying good reorderings of $\mathbf{P}^{(tm)}$. Without this type of deviation strategy, there would typically be no 'mixing' of the row and column objects in the permutations that we would identify for the combined [row and column] object set.) Thus, for $\mathbf{0}$ denoting (an appropriately dimensioned) matrix of all zeros,

$$\mathbf{P}^{(tm)} = \left[\begin{array}{cc} \mathbf{0}_{n_a \times n_a} & \mathbf{Q}_{(dev) n_a \times n_b} \\ \mathbf{Q}'_{(dev) n_b \times n_a} & \mathbf{0}_{n_b \times n_b} \end{array} \right],$$

is the (square) $n \times n$ proximity matrix subjected to a simultaneous row and column reordering, which in turn will induce separate row and column reorderings for the original two-mode proximity matrix $\mathbf{Q}$.

The M-file, `ordertm.m`, implements a quadratic assignment reordering heuristic on the derived matrix $\mathbf{P}^{(tm)}$, with usage

```
[outperm,rawindex,allperms,index,squareprox] = ...
 ordertm(proxtm,targ,inperm,kblock)
```

where the two-mode proximity matrix `proxtm` (with its entries deviated from the mean proximity within the use of the M-file) forms the upper-right-

and lower-left-hand portions of a defined square ($n \times n$) proximity matrix (`squareprox`) with a dissimilarity interpretation, and with zeros placed elsewhere ($n =$ number of rows + number of columns of `proxtm` $= n_a + n_b$); three separate local operations are used to permute the rows and columns of the square proximity matrix to maximize the cross-product index with respect to a square target matrix `targ`: (a) pairwise interchanges of objects in the permutation defining the row and column order of the square proximity matrix; (b) the insertion of from 1 to `kblock` (which is less than or equal to $n - 1$) consecutive objects in the permutation defining the row and column order of the data matrix; (c) the rotation of from 2 to `kblock` (which is less than or equal to $n - 1$) consecutive objects in the permutation defining the row and column order of the data matrix. The beginning input permutation (a permutation of the first n integers) is `inperm`; `proxtm` is the two-mode $n_a \times n_b$ input proximity matrix; `targ` is the $n \times n$ input target matrix. The final permutation of `squareprox` is `outperm`, having the cross-product index `rawindex` with respect to `targ`; `allperms` is a cell array containing `index` entries corresponding to all the permutations identified in the optimization from `allperms{1} = inperm` to `allperms{index} = outperm`.

In the example to follow, `ordertm.m`, is used on the `supreme_agree5x4` dissimilarity matrix. The square equally-spaced target matrix is obtained from the LUS utility, `targlin.m`. The (reordered) matrix, `squareprox` (using the permutation, `outperm`), shows clearly the unidimensional pattern for a two-mode data matrix that will be explicitly fitted in the next section of this chapter. The order of the justices is as expected in the new coding scheme, except for the minor inversion of Th:5 and Sc:9 — St:1 $\succ$ Br:6 $\succ$ Gi:2 $\succ$ So:7 $\succ$ Oc:3 $\succ$ Ke:8 $\succ$ Re:4 $\succ$ Th:5 $\succ$ Sc:9

```
>> load supreme_agree5x4.dat
>> supreme_agree5x4

supreme_agree5x4 =

    0.3000    0.3700    0.6400    0.8600
    0.2800    0.2200    0.5100    0.7200
    0.4500    0.4500    0.3300    0.4600
    0.5700    0.5600    0.2300    0.3400
    0.7600    0.7100    0.4100    0.2100

>> [outperm,rawindex,allperms,index,squareprox] = ordertm(supreme_agree5x4,targlin(9),randperm(9),3)

outperm =
     1     6     2     7     3     8     4     5     9


rawindex =

   14.1420



index =

    17

>> squareprox(outperm,outperm)

ans =

         0   -0.1690         0   -0.0990         0    0.1710         0         0    0.3910
   -0.1690         0   -0.1890         0   -0.0190         0    0.1010    0.2910         0
         0   -0.1890         0   -0.2490         0    0.0410         0         0    0.2510
   -0.0990         0   -0.2490         0   -0.0190         0    0.0910    0.2410         0
         0   -0.0190         0   -0.0190         0   -0.1390         0         0   -0.0090
    0.1710         0    0.0410         0   -0.1390         0   -0.2390   -0.0590         0
         0    0.1010         0    0.0910         0   -0.2390         0         0   -0.1290
         0    0.2910         0    0.2410         0   -0.0590         0         0   -0.2590
    0.3910         0    0.2510         0   -0.0090         0   -0.1290   -0.2590         0
```

6.2 Fitting a Two-Mode Unidimensional Scale

It is possible to fit unidimensional scales to two-mode proximity data based on a given permutation of the combined row and column object set. Specifically, if $\rho(\cdot)$ denotes some given permutation of the first n integers (where the first n_a integers denote row objects labeled $1, 2, \ldots, n_a$, and the remaining n_b integers denote column objects, labeled $n_a + 1, n_a + 2, \ldots, n_a + n_b$ $(= n)$), we seek a set of coordinates, $x_1 \leq x_2 \leq \cdots \leq x_n$, such that using the reordered square proximity matrix, $\mathbf{P}^{(tm)}_{\rho_0} = \{p^{(tm)}_{\rho_0(i)\rho_0(j)}\}$, the least-squares criterion

$$\sum_{i,j=1}^{n} w_{\rho_0(i)\rho_0(j)}(p^{(tm)}_{\rho_0(i)\rho_0(j)} - |x_j - x_i|)^2,$$

is minimized, where $w_{\rho_0(i)\rho_0(j)} = 0$ if $\rho_0(i)$ and $\rho_0(j)$ are both row or both column objects, and $= 1$ otherwise. The entries in the matrix fitted to $\mathbf{P}^{(tm)}_{\rho_0}$ are based on the absolute coordinate differences (and which correspond to nonzero values of the weight function $w_{\rho_0(i)\rho_0(j)}$), and thus satisfy certain linear inequality constraints generated from how the row and column objects are intermixed by the given permutation $\rho_0(\cdot)$. To give a schematic representation of how these constraints are generated, suppose r_1 and r_2 (c_1 and c_2) denote two arbitrary row (column) objects, and suppose the following 2×2 matrix represents what is to be fitted to the four proximity values present between r_1, r_2 and c_1, c_2:

	c_1	c_2
r_1	a	b
r_2	c	d

Depending on how these four objects are ordered (and intermixed) by the permutation $\rho_0(\cdot)$, certain constraints must be satisfied by the entries a, b, c, and d. The representative constraints are given schematically below according to the types of intermixing that might be present:

(a) $r_1 \prec r_2 \prec c_1 \prec c_2$ implies $a + d = b + c$;
(b) $r_1 \prec c_1 \prec r_2 \prec c_2$ implies $a + c + d = b$;
(c) $r_1 \prec c_1 \prec c_2 \prec r_2$ implies $a + c = b + d$;
(d) $r_1 \prec r_2 \prec c_1$ implies $c \leq a$;
(e) $r_1 \prec c_1 \prec c_2$ implies $a \leq b$.

The confirmatory unidimensional scaling of a two-mode proximity matrix (based on iterative projection using a given permutation of the row and column objects) is carried out with the M-file, `linfittm`, with usage

```
[fit,diff,rowperm,colperm,coord] = linfittm(proxtm,inperm)
```

Here, `proxtm` is the two-mode proximity matrix, and `inperm` is the given ordering of the row and column objects pooled together; `fit` is an $n_a \times n_b$ matrix of absolute coordinate differences fitted to `proxtm(rowperm,colperm)`, with `diff` being the (least-squares criterion) sum of squared discrepancies between `fit` and `proxtm(rowperm,colperm)`; `rowperm` and `colperm` are the row and column object orderings derived from `inperm`. The $(n_a + n_b) = n$ coordinates (ordered with the smallest such coordinate value set at zero) are given in `coord`. The example given below uses the permutation obtained from `ordertm.m` on the data matrix `supreme_agree5x4`.

```
>> inperm = [1 6 2 7 3 8 4 5 9]

inperm =

     1     6     2     7     3     8     4     5     9
```

```
>> [fit,diff,rowperm,colperm,coord] = linfittm(supreme_agree5x4,inperm)

fit =

    0.1635    0.2895    0.6835    1.0335
    0.0865    0.0395    0.4335    0.7835
    0.4065    0.2805    0.1135    0.4635
    0.6340    0.5080    0.1140    0.2360
    0.7965    0.6705    0.2765    0.0735


diff =

    0.2849


rowperm =

     1
     2
     3
     4
     5


colperm =

     1
     2
     3
     4


coord =

         0
    0.1635
    0.2500
    0.2895
    0.5700
    0.6835
    0.7975
    0.9600
    1.0335
```

In complete analogy with the LUS discussion (where the M-file, `linfitac.m`, generalizes `linfit.m` by fitting an additive constant along with the absolute coordinate differences), the more general unidimensional scaling model can be fitted with an additive constant using the M-file, `linfittmac.m`. Specifically, we now seek a set of coordinates, $x_1 \leq x_2 \leq \cdots \leq x_n$, and an additive constant c, such that using the reordered square proximity matrix, $\mathbf{P}^{(tm)}_{\rho_0} = \{p^{(tm)}_{\rho_0(i)\rho_0(j)}\}$, the least-squares criterion

$$\sum_{i,j=1}^{n} w_{\rho_0(i)\rho_0(j)}(p^{(tm)}_{\rho_0(i)\rho_0(j)} + c - |x_j - x_i|)^2,$$

is minimized, where again $w_{\rho_0(i)\rho_0(j)} = 0$ if $\rho_0(i)$ and $\rho_0(j)$ are both row or both column objects, and $= 1$ otherwise. The M-file usage is

```
[fit,vaf,rowperm,colperm,addcon,coord] =  ...
      linfittmac(proxtm,inperm)
```

and does a confirmatory two-mode fitting of a given unidimensional ordering of the row and column objects of a two-mode proximity matrix, `proxtm`, using the Dykstra-Kaczmarz iterative projection least-squares method. In comparison, the M-file `linfittmac.m` differs from `linfittm.m` by including the estimation of an additive constant, and thus allowing `vaf` to be legitimately given as the goodness-of-fit index (as opposed to just `diff` as we did in `linfittm.m`). Again, `inperm` is the given ordering of the row and column objects together; `fit` is an n_a (number of rows) by n_b (number of columns) matrix of absolute coordinate differences fitted to `proxtm(rowperm,colperm)`; `rowperm` and `colperm` are the row and column object orderings derived from `inperm`. The estimated additive constant, `addcon`, can be interpreted as being added to `proxtm` (or alternatively, subtracted from the fitted matrix `fit`).

The same exemplar permutation is used below (as for `linfittm.m`); following the MATLAB output that now includes the additive constant of $-.2132$ and the `vaf` of .9911, the two unidimensional scalings (in their coordinate forms) are provided in tabular form with an explicit indication of what is a row object (R) and what is a column object (C).

```
>> [fit,vaf,rowperm,colperm,addcon,coord] = linfittmac(supreme_agree5x4,[1 6 2 7 3 8 4 5 9])

fit =

    0.0974    0.1405    0.4469    0.6325
    0.0431         0    0.3064    0.4920
    0.2594    0.2163    0.0901    0.2757
    0.3803    0.3372    0.0309    0.1548
    0.5351    0.4920    0.1856         0


vaf =

    0.9911


rowperm =

     1
     2
```

Table 3: The Two Unidimensional Scalings of the supreme_agree5x4 Data Matrix.

justice	number	R or C	no constant	with constant
St	1	R	.0000	.0000
Br	6	C	.1635	.0974
Gi	2	R	.2500	.1405
So	7	C	.2895	.1405
Oc	3	R	.5700	.3568
Ke	8	C	.6835	.4469
Re	4	R	.7975	.4777
Th	5	R	.9600	.6325
Sc	9	C	1.0335	.6325

```
     3
     4
     5


colperm =

     1
     2
     3
     4


addcon =

   -0.2132


coord =

         0
    0.0974
    0.1405
    0.1405
    0.3568
    0.4469
    0.4777
    0.6325
    0.6325
```

7 Order-Constrained Partition Construction

The classification task considered in the present section is one of constructing an (optimal) ordered partition for a set of n objects, $S = \{O_1, \ldots, O_n\}$, defined by a collection of M mutually exclusive and exhaustive subsets of S, denoted $S_1, S_2, \ldots, S_M$, for which an order is imposed on the placement of the classes, $S_1 \prec S_2 \prec \cdots \prec S_M$, and also a prior order is present for the objects within classes. Again, the data available to guide this search are assumed to be in the form of an $n \times n$ symmetric proximity matrix $\mathbf{P} = \{p_{ij}\}$. In general, the identification of an optimal ordered partition for S will be carried out by the maximization of an index of merit intended to measure how well a given ordered partition reflects the data in $\mathbf{P}$. The initial constraining object order will be constructed with the (heuristic) unidimensional scaling routine, `uniscalqa.m`, discussed in Section 2.2.

A merit measure can be developed directly based on a coordinate representation for each of the M ordered classes, $S_1 \prec S_2 \prec \cdots \prec S_M$, that generalizes the use of the single term $\sum_i (t_i^{(\rho)})^2$ for a unidimensional scaling discussed in Section 2. Here, M coordinates, $x_1 \leq \cdots \leq x_M$, are to be identified so that the residual sum-of-squares

$$\sum_{k \leq k'} \sum_{i_k \in S_k,\ j_{k'} \in S_{k'}} (p_{i_k j_{k'}} - \mid x_{k'} - x_k \mid)^2,$$

is minimized (the notation $p_{i_k j_{k'}}$ indicates those proximities in $\mathbf{P}$ defined between objects with subscripts $i_k \in S_k$ and $j_{k'} \in S_{k'}$). Define each of the sets, $\Omega_1, \ldots, \Omega_M$, by the n subsets of S that contain the first i objects, $\{O_1, \ldots, O_i\}$, for $1 \leq i \leq n$; a transformation of an entity in Ω_{k-1} (say, A_{k-1}) to one in Ω_k (say, A_k) is possible if $A_{k-1} \subset A_k$.

A direct extension of the argument that led to optimal coordinate representation for single objects would require the maximization of

$$\sum_{k=1}^{M} (\frac{1}{n_k})(G(A_k - A_{k-1}))^2, \qquad (10)$$

where $G(A_k - A_{k-1}) =$

$$\sum_{k' \in A_k - A_{k-1}} \sum_{i' \in A_{k-1}} p_{k'i'} - \sum_{k' \in A_k - A_{k-1}} \sum_{i' \in S - A_k} p_{k'i'},$$

and n_k denotes the number of objects in $A_k - A_{k-1}$. If an optimal ordered partition that maximizes (10) is denoted by $S_1^* \prec \cdots \prec S_M^*$, the optimal coordinates for each of the M classes can be given as

$$x_k^* = (\frac{1}{nn_k})G(S_k^*), \tag{11}$$

where $x_1^* \leq \cdots \leq x_M^*$, and $\sum_k n_k x_k^* = 0$. The residual sum-of-squares has the form

$$\sum_{i<j} p_{ij}^2 - (\frac{1}{n}) \sum_k (\frac{1}{n_k})(G(S_k^*))^2. \tag{12}$$

7.1 The Dynamic Programming Implementation

Given the proximity matrix, $\mathbf{P}$, suppose we have a constraining object order, assumed without loss of generality, for now, to be the identity order, and used to label the rows and columns of $\mathbf{P}$. An order constrained clustering consists of finding a set of M classes, $S_1, \ldots, S_k, \ldots, S_M$, having n_k objects in S_k and where the objects in S_k are consecutive:

$$\{O_{n_1+\cdots+n_{k-1}+1}, O_{n_1+\cdots+n_{k-1}+2}, \ldots, O_{n_1+\cdots+n_{k-1}+n_k}\}.$$

A recursive dynamic programming strategy can be used to solve this task that we implement in the M-file, `orderpartitionfnd.m`. In the session recorded below, we give the help information for this M-file (as well as in the Appendix) and run it on the `supreme_agree` data with a constraining identity permutation on the objects obtained from the previous unidimensional scaling. Generally, the class membership into from 1 to n ordered classes is given by the two $n \times n$ matrices of `membership` and `permmember` with rows corresponding to the number of ordered classes constructed and columns to the objects. The identity permutation also labels the columns of `membership`; the constraining object order labels the columns of `permmember` (in this example, these two permutations happen to be the same). The two vectors of `objectives` and `residsumsq` contain, respectively, the values maximized in (10) and the corresponding residual sums-of-squares from (12). We will continue with the interpretation after the verbatim output from the session is provided.

```
>> load supreme_agree.dat
>> help orderpartitionfnd.m
```

```
  ORDERPARTITIONFND uses dynamic programming to
  construct a linearly constrained cluster analysis that
  consists of a collection of partitions with from 1 to
  n ordered classes.

  syntax: [membership,objectives,permmember,clusmeasure,...
     cluscoord,residsumsq] = orderpartitionfnd(prox,lincon)

  PROX is the input proximity matrix (with a zero main diagonal
  and a dissimilarity interpretation); LINCON is the given
  constraining linear order (a permutation of the integers from
  1 to n).
  MEMBERSHIP is the n x n matrix indicating cluster membership,
  where rows correspond to the number of ordered clusters,
  and the columns are in the identity permutation input order
  used for PROX. PERMMEMBER uses LINCON to reorder the columns
  of MEMBERSHIP.
  OBJECTIVES is the vector of merit values maximized in the
  construction of the ordered partitions; RESIDSUMSQ is the
  vector of residual sum of squares obtained for the ordered
  partition construction.  CLUSMEASURE is the n x n matrix
  (upper-triangular) containing the cluster measures for contiguous
  object sets; the appropriate values in CLUSMEASURE are added
  to obtain the values optimized in OBJECTIVES; CLUSCOORD is also
  an n x n (upper-triangular) matrix but now containing the coordinates
  that would be would be used for all the (ordered)
  objects within a class.



>> [membership,objectives,permmember,clusmeasure,...
     cluscoord,residsumsq] = orderpartitionfnd(supreme_agree,[1 2 3 4 5 6 7 8 9])

membership =

     1     1     1     1     1     1     1     1     1
     2     2     2     2     1     1     1     1     1
     3     3     3     3     2     2     2     1     1
     4     3     3     3     2     2     2     1     1
     5     4     4     4     3     2     2     1     1
     6     5     5     4     3     2     2     1     1
     7     6     6     5     4     3     2     1     1
     8     7     6     5     4     3     2     1     1
     9     8     7     6     5     4     3     2     1


objectives =

         0
   73.8432
   83.2849
   86.9479
   88.1095
   88.6861
   89.0559
   89.2241
   89.3166


permmember =

     1     1     1     1     1     1     1     1     1
     2     2     2     2     1     1     1     1     1
     3     3     3     3     2     2     2     1     1
     4     3     3     3     2     2     2     1     1
     5     4     4     4     3     2     2     1     1
     6     5     5     4     3     2     2     1     1
     7     6     6     5     4     3     2     1     1
     8     7     6     5     4     3     2     1     1
     9     8     7     6     5     4     3     2     1


clusmeasure =

   23.6196   32.8860   38.7361   41.0240   30.0125   19.4400   10.2972    2.4865         0
         0   10.5625   17.5232   21.0675   13.6530    7.0567    2.1961    0.0229    2.9524
         0         0    7.1289   11.0450    5.7132    1.8090    0.0289    2.2204    9.3960
         0         0         0    4.1209    1.0804    0.0001    1.3110    7.9885   19.3681
         0         0         0         0    0.3136    2.0201    6.2208   17.4306   32.8192
         0         0         0         0         0    2.1025    7.0688   20.2280   37.5156
         0         0         0         0         0         0    5.3361   20.0978   38.8800
         0         0         0         0         0         0         0   16.2409   36.0401
         0         0         0         0         0         0         0         0   19.8916


cluscoord =

   -0.5400   -0.4506   -0.3993   -0.3558   -0.2722   -0.2000   -0.1348   -0.0619         0
         0   -0.3611   -0.3289   -0.2944   -0.2053   -0.1320   -0.0672    0.0063    0.0675
         0         0   -0.2967   -0.2611   -0.1533   -0.0747   -0.0084    0.0676    0.1287
         0         0         0   -0.2256   -0.0817   -0.0007    0.0636    0.1404    0.1996
         0         0         0         0    0.0622    0.1117    0.1600    0.2319    0.2847
         0         0         0         0         0    0.1611    0.2089    0.2885    0.3403
         0         0         0         0         0         0    0.2567    0.3522    0.4000
         0         0         0         0         0         0         0    0.4478    0.4717
         0         0         0         0         0         0         0         0    0.4956


residsumsq =

   10.3932
    2.1884
    1.1393
    0.7323
    0.6033
    0.5392
    0.4981
    0.4794
    0.4691
```

To interpret this example further, it appears that five ordered classes may be a good ‘stopping point’ for the clustering process — moving to four gives a noticeable drop in the achievable objective function value. The fifth row of `membership` is the vector `5 4 4 4 3 2 2 1 1`, and thus the justice partitioning of {{St},{Br, Gi, So},{Oc},{Ke,Re},{Sc,Th}}, clearly placing O’Connor in a separate ‘swing’ class. The `objectives` value for this partition is 88.1095, and can be reconstructed from the values in `clusmeasure` that delineate the extent of the various classes, i.e., the values in this matrix at positions (1,1), (2,4), (5,5), (6,7), (8,9): 23.6196 + 21.0675 + .3136 + 7.0688 + 36.0401 =

88.1096 ($\approx$ 88.1095 to rounding). The coordinates for the classes are given in these same positions in the matrix `cluscoord`: $-.5400$; $-.2944$; .0622; .2089; .4717. Weighting these by the class sizes of 1, 3, 1, 2, 2, respectively, and then summing, gives the value (to rounding) of 0.0 (i.e., the constraint $\sum_k n_k x_k^* = 0$ is satisfied). Finally, the residual sum-of-squares of .6033 is reconstructible from (12) as 10.3932 – (1/9)88.1096, where 10.3932 is $\sum_{i<j} p_{ij}^2$, and is always given as the first entry in `residsumsq` corresponding to only one class that must be placed at a coordinate value of 0.0 (because of the constraint $\sum_k n_k x_k^* = 0$).

7.2 Two Utility Functions For Coordinate Estimation

When constructing ordered partitions through optimizing the measure in (10), the coordinates were generated as a byproduct through the closed-form expression in (11). For some extensions we contemplate, and particularly to multiple dimensions and the imposition of ordered partitions on each, it would be useful to have a fitting mechanism that would not depend on the presence of nonnegative proximities. To this end we provide two utility M-files, `linfit_tied.m` and `linfitac_tied.m`, that for a given ordered partition and underlying constraining object order, will fit the M coordinates, $x_1 \leq \cdots \leq x_M$ (and an additional additive constant c in the case of `linfitac_tied.m`) by minimizing

$$\sum_{k \leq k'} \sum_{i_k \in S_k,\ j_{k'} \in S_{k'}} (p_{i_k j_{k'}} - (\mid x_{k'} - x_k \mid - \, c \,))^2.$$

The MATLAB session below includes the help comments for each M-file and fits the five-class ordered partition found earlier. For both M-files, the `supreme_agree` proximity matrix is used, along with a constraining order in `inperm` (here, the identity), and the pattern of tied coordinates imposed in order along the continuum (given as the fifth row of `permmember`, `[5 4 4 4 2 2 1 1]`).

```
>> load supreme_agree.dat

>> help linfit_tied.m

  LINFIT_TIED does a confirmatory fitting of a given
  unidimensional order using Dykstra's
```

```
   (Kaczmarz's) iterative projection least-squares method. This
   includes the possible imposition of tied coordinates.

   syntax: [fit, diff, coord] = linfit_tied(prox,inperm,tiedcoord)

   INPERM is the given order;
   FIT is an $n \times n$ matrix that is fitted to
   PROX(INPERM,INPERM) with least-squares value DIFF;
   COORD gives the ordered coordinates whose absolute
   differences could be used to reconstruct FIT; TIEDCOORD
   is the tied pattern of coordinates imposed (in order)
   along the continuum (using the integers from 1 up to n
   to indicate the tied positions).



>> [fit,diff,coord] = linfit_tied(supreme_agree,[1 2 3 4 5 6 7 8 9],[5 4 4 4 3 2 2 1 1])

fit =

         0    0.2456    0.2456    0.2456    0.6022    0.7489    0.7489    1.0117    1.0117
    0.2456         0         0         0    0.3567    0.5033    0.5033    0.7661    0.7661
    0.2456         0         0         0    0.3567    0.5033    0.5033    0.7661    0.7661
    0.2456         0         0         0    0.3567    0.5033    0.5033    0.7661    0.7661
    0.6022    0.3567    0.3567    0.3567         0    0.1467    0.1467    0.4094    0.4094
    0.7489    0.5033    0.5033    0.5033    0.1467         0         0    0.2628    0.2628
    0.7489    0.5033    0.5033    0.5033    0.1467         0         0    0.2628    0.2628
    1.0117    0.7661    0.7661    0.7661    0.4094    0.2628    0.2628         0         0
    1.0117    0.7661    0.7661    0.7661    0.4094    0.2628    0.2628         0         0


diff =

    0.6032


coord =

   -0.5400
   -0.2944
   -0.2944
   -0.2944
    0.0622
    0.2089
    0.2089
    0.4717
    0.4717

>> help linfitac_tied.m

   LINFITAC_TIED does a confirmatory fitting of a given unidimensional order
   using the Dykstra--Kaczmarz iterative projection
   least-squares method, but differing from linfit_tied.m in
   including the estimation of an additive constant.  This also allows
   the possible imposition of tied coordinates.

   syntax: [fit, vaf, coord, addcon] = linfitac_tied(prox,inperm,tiedcoord)

   INPERM is the given order;
   FIT is an $n \times n$ matrix that is fitted to
   PROX(INPERM,INPERM) with variance-accounted-for VAF;
   COORD gives the ordered coordinates whose absolute differences
   could be used to reconstruct FIT; ADDCON is the estimated
   additive constant that can be interpreted as being added to PROX.
   TIEDCOORD is the tied pattern of coordinates imposed (in order)
   along the continuum (using the integers from 1 up to n
   to indicate the tied positions).


>> [fit,vaf,coord,addcon] = linfitac_tied(supreme_agree,[1 2 3 4 5 6 7 8 9],[5 4 4 4 3 2 2 1 1])

fit =

         0    0.1435    0.1435    0.1435    0.3981    0.4682    0.4682    0.6289    0.6289
    0.1435         0         0         0    0.2546    0.3247    0.3247    0.4854    0.4854
    0.1435         0         0         0    0.2546    0.3247    0.3247    0.4854    0.4854
    0.1435         0         0         0    0.2546    0.3247    0.3247    0.4854    0.4854
    0.3981    0.2546    0.2546    0.2546         0    0.0701    0.0701    0.2308    0.2308
    0.4682    0.3247    0.3247    0.3247    0.0701         0         0    0.1607    0.1607
    0.4682    0.3247    0.3247    0.3247    0.0701         0         0    0.1607    0.1607
    0.6289    0.4854    0.4854    0.4854    0.2308    0.1607    0.1607         0         0
    0.6289    0.4854    0.4854    0.4854    0.2308    0.1607    0.1607         0         0


vaf =

    0.9671


coord =

   -0.3359
   -0.1924
   -0.1924
   -0.1924
    0.0622
    0.1323
    0.1323
    0.2930
    0.2930


addcon =

   -0.2297
```

As can be seen in the preceding output, the coordinates given earlier for the five-class ordered partition can be retrieved using `linfit_tied.m` along with the least-squares loss value, `diff`, of 0.6032 (this is within a .0001 rounding error of the previously given five-class residual sum-of-squares of 0.6033). In incorporating the estimation of an additive constant, c, as part of the model that is fitted with `linfitac_tied`, a legitimate variance-accounted-for (VAF) measure can be given and used in place of the unnormalized least-squares loss value (usually denoted as `vaf`). Here, we would define the VAF measure as

$$VAF = 1 - \frac{\sum_{k \le k'} \sum_{i_k \in S_k,\ j_{k'} \in S_{k'}} (p_{i_k j_{k'}} - (\mid x_{k'} - x_k \mid - c\ ))^2}{\sum_{i<j} (p_{ij} - \bar{p})^2}, \qquad (13)$$

where $\bar{p}$ is the mean of the off-diagonal proximities in $\mathbf{P}$. The argument for the legitimacy of a VAF measure follows the same logic as being able to use a VAF measure only in a multiple regression that includes an additive constant (and therefore, the least-squares structure is not forced to go through the origin).

The normalized VAF measure may help in deciding when an unacceptable drop is present when going from one ordered partition to another. Based on running the MATLAB script given below, we can generate the following table:

Number of Ordered Classes	Variance-Accounted-For
9	.9796
8	.9796
7	.9786
6	.9708
5	.9671
4	.9446
3	.8513
2	.6759
1	.0000

As can be seen, the five-class partition has a very high VAF of .9671, and there is a somewhat precipitous drop of over 2% in going to one fewer; also, in going the complete way from nine classes to five, we have a drop of only slightly larger than 1%. So, based on this reasoning, the five-class ordered partition might be considered the 'stopping place' of choice.

```
>> load supreme_agree.dat

>> identityperm = [1 2 3 4 5 6 7 8 9];

>> [membership,objectives,permmember,clusmeasure,...
cluscoord,residsumsq] = orderpartitionfnd(supreme_agree,identityperm);

>> for i = 1:9

tiedcoord = permmember(10-i,:);

 [fit,vaf,coord,addcon] = linfitac_tied(supreme_agree,identityperm,tiedcoord);

 fits{i} = fit;

 vafs{i} = vaf;
```

```
coords{i} = coord; addcons{i} = addcon;

 end

>> for i = 1:9

 vaf = vafs{i}

end

vaf =

    0.9796

... (output deleted)

vaf =

    0.6759


vaf =

     0
```

7.3 Extensions to Generalized Ultrametrics

As we construct a collection of T ordered partitions of S into anywhere from 1 to n classes, and where each class within a partition defines a consecutive set of objects with respect to some fixed ordering of the n objects, denote the T partitions as $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_T$. Here, $\mathcal{P}_1$ is a partition containing n classes, $\mathcal{P}_T$ includes only a single class, and $\mathcal{P}_{t-1}$ has more classes than $\mathcal{P}_t$ for $t \geq 2$. Based on $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_{T-1}$, if a corresponding collection of $n \times n$ 0/1 dissimilarity matrices $\mathbf{P}_1, \ldots, \mathbf{P}_{T-1}$ is constructed, where a 0 in $\mathbf{P}_t$ indicates an object pair defined within a class in $\mathcal{P}_t$, and 1 otherwise, then for any collection of nonnegative weights $\alpha_1, \ldots, \alpha_{T-1}$, the dissimilarity matrix, say, $\mathbf{P}_\alpha = \{p_{ij}^{(\alpha)}\} \equiv \sum_{t=1}^{T-1} \alpha_t \mathbf{P}_t$, defines a metric on the objects (based on the observation that sums of metrics are metric [but with the possible extension that allows some dissimilarities to be zero for nonidentical objects]). (We don't consider the partition $\mathcal{P}_T$ defined by one class since the corresponding $\mathbf{P}_T$ would be identically zero and thus provides no contribution to the defining sum of weights.) Depending on the constraints placed on $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_{T-1}$, more restrictive forms for the metric defined by $\mathbf{P}_\alpha$ ensue; and specific to the restrictions made, it may be possible to retrieve $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_{T-1}$ and $\alpha_1, \ldots, \alpha_{T-1}$ given only $\mathbf{P}_\alpha$, provide convenient graphical representations for the collection $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_{T-1}$, or somehow to approach the task of constructing $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_{T-1}$ and $\alpha_1, \ldots, \alpha_{T-1}$ from some given proximity matrix $\mathbf{P}$ so that $\mathbf{P}_\alpha$ approximates $\mathbf{P}$ in some explicitly defined sense.

The obvious prime exemplar for this type of structure just discussed would be when $\mathcal{P}_t$ is formed from $\mathcal{P}_{t-1}$ by uniting two or more classes in the latter. The entries in $\mathbf{P}_\alpha$ then satisfy the ultrametric inequality ($p_{ij}^{(\alpha)} \leq \max\{p_{ik}^{(\alpha)}, p_{jk}^{(\alpha)}\}$ for all $O_i, O_j, O_k \in S$), the partition hierarchy and the weights are retrievable given only $\mathbf{P}_\alpha$, and a representation of the hierarchical clustering can be given in the form of what is usually called a dendrogram. In a more general context where $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_{T-1}$ are merely restricted to be ordered partitions, each defined by classes contiguous with respect to some given ordering for the objects in S, the entries in the matrix $\mathbf{P}_\alpha$ satisfy (at the least) the anti-Robinson condition (i.e., if $O_i \prec O_j \prec O_k$, then $p_{ik}^{(\alpha)} \geq \max\{p_{ij}^{(\alpha)}, p_{jk}^{(\alpha)}\}$, and can be constructed by sums of subsets of a collection of nonnegative weights $\alpha_1, \ldots, \alpha_{T-1}$, just as in the more restrictive ultrametric context. Thus, although the same number of weights may be needed to construct $\mathbf{P}_\alpha$ as for an ultrametric, the structures definable through ordered partitions restricted only by the class contiguity constraint are broader than those possible through the concept of an ultrametric.

To illustrate the fitting process for a collection of ordered partitions, we use the membership matrix obtained from `orderpartitionfnd.m` as an input argument to `orderpartitionfit.m`, where the latter provides a least-squares approximation to a proximity matrix based on a given collection of partitions with ordered classes. Note that we must remove the first row of `membership` that is output from `orderpartitionfnd.m` before it is used as an input argument for `orderpartitionfit.m`. This removes the one-class ordered partition that adds nothing to the fitting but actually causes difficulty in our nonnegative least-squares routine. The latter M-file is based on the Wollan and Dykstra (1987) Fortran subroutine code that we have rewritten and included as an M-file called `dykstra.m` (this is called by `orderpartitionfit.m`).

The MATLAB session recorded below includes the help information for `orderpartitonfit.m` and should be relatively self-explanatory. Because the ordered partitions here happen to be hierarchically nested, the resulting fitted matrix given is an ultrametric (with `vaf` of .7339), and built up from the partition weights given in `weights`.

```
>> load supreme_agree.dat
>> [membership,objectives,permmember,clusmeasure,...
cluscoord,residsumsq] = orderpartitionfnd(supreme_agree,[1 2 3 4 5 6 7 8 9]);
>> membership = membership(2:9,:);
>> membership

membership =

     2     2     2     2     1     1     1     1     1
     3     3     3     3     2     2     2     1     1
     4     3     3     3     2     2     2     1     1
     5     4     4     4     3     2     2     1     1
     6     5     5     4     3     2     2     1     1
     7     6     6     5     4     3     2     1     1
     8     7     6     5     4     3     2     1     1
     9     8     7     6     5     4     3     2     1

>> help orderpartitionfit.m

  ORDERPARTITIONFIT provides a least-squares approximation to a proximity
  matrix based on a given collection of partitions with ordered classes.

  syntax: [fit,weights,vaf] = orderpartitionfit(prox,lincon,membership)

  PROX is the n x n input proximity matrix (with a zero main diagonal
  and a dissimilarity interpretation); LINCON is the given constraining
  linear order (a permutation of the integers from 1 to n).
  MEMBERSHIP is the m x n matrix indicating cluster membership, where
  each row corresponds to a specific ordered partition (there are
  m partitions in general);
  the columns are in the identity permutation input order used for PROX.
  FIT is an n x n matrix fitted to PROX (through least-squares) constructed
  from the nonnegative weights given in the m x 1 WEIGHTS vectors
  corresponding to each of the ordered partitions.  VAF is the variance-
  accounted-for in the proximity matrix PROX by the fitted matrix FIT.



>> [fit,weights,vaf] = orderpartitionfit(supreme_agree,[1 2 3 4 5 6 7 8 9],membership)

fit =

         0    0.3633    0.3633    0.3633    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633         0    0.2550    0.2550    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2550         0    0.2550    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2550    0.2550         0    0.6405    0.6405    0.6405    0.6405    0.6405
    0.6405    0.6405    0.6405    0.6405         0    0.3100    0.3100    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100         0    0.2550    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100    0.2550         0    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.4017    0.4017    0.4017         0    0.2100
    0.6405    0.6405    0.6405    0.6405    0.4017    0.4017    0.4017    0.2100         0


weights =

    0.2388
    0.0383
    0.0533
    0.0550
         0
         0
    0.0450
    0.2100


vaf =

    0.7339
```

8 Some Possible LUS and CUS Generalizations

8.1 Additive Representation Through Multiple Structures

The use of multiple structures to represent additively a given proximity matrix, whether they come from a LUS or CUS model, proceeds directly through successive residualization and iteration. We restrict ourselves to the fitting of two such structures but the same process would apply for any such number. Initially, a first matrix is fitted to a given proximity matrix and a first residual matrix obtained; a second structure is then fitted to these first residuals, producing a second residual matrix. Iterating, the second fitted matrix is now subtracted from the original proximity matrix and a first (re)fitted matrix obtained; this first (re)fitted matrix in turn is subtracted from the original proximity matrix and a new second matrix (re)fitted. This process continues until the `vaf` for the sum of both fitted matrices no longer changes substantially.

The M-files, `biscalqa.m` and `biscaltmac.m` fit (additively) two LUS structures in the least-squares sense for, respectively, one and two-mode proximity matrices; `bicirac.m` fits two CUS models. The explicit usages are

```
[outpermone,outpermtwo,coordone,coordtwo,fitone,fittwo,addconone,addcontwo,vaf] = ...
    biscalqa(prox,targone,targtwo,inpermone,inpermtwo,kblock,nopt)

[find,vaf,targone,targtwo,outpermone,outpermtwo,rowpermone,colpermone,rowpermtwo, ...
colpermtwo,addconone,addcontwo,coordone,coordtwo,axes] = ...
    biscaltmac(proxtm,inpermone,inpermtwo,kblock,nopt)

[find,vaf,targone,targtwo,outpermone,outpermtwo,addconone,addcontwo] = ...
    bicirac(prox,inperm,kblock)
```

where (in `biscalqa.m`) `prox` is the input proximity matrix (with a zero main diagonal and a dissimilarity interpretation); `targone` is the input target matrix for the first dimension (usually with a zero main diagonal and a dissimilarity interpretation representing equally-spaced locations along a continuum); targtwo is the input target matrix for the second dimension; inpermone is the input beginning permutation for the first dimension (a permutation of the first n integers); inpermtwo is the input beginning permutation for the second dimension; the insertion and rotation routines use from 1 to kblock (which is less than or equal to $n-1$) consecutive objects in the permutation defining the row and column orders of the data matrix. The switch variable, nopt, controls the confirmatory or exploratory fitting of the unidimensional scales; a value of nopt = 0 will fit in a confirmatory manner the two scales indicated by inpermone and inpermtwo; a value of nopt = 1 uses iterative QA to locate the better permutations to fit; outpermone is the final object permutation for the first dimension; outpermtwo is the final object permutation for the second dimension; coordone is the set of first dimension coordinates in ascending order; coordtwo is the set of second dimension coordinates in ascending order; addconone is the additive constant for the first dimension model; addcontwo is the additive constant for the second dimension model; vaf is the variance-accounted-for in prox by the bidimensional scaling.

In biscaltmac.m, proxtm is the input two-mode proximity matrix with a dissimilarity interpretation; find is the least-squares optimal matrix (with variance-accounted-for of vaf) to proxtm and is the sum of the two matrices, targone and targtwo, based on the two row and column object orderings given by the ending permutations, outpermone and outpermtwo, and in turn, rowpermone and rowpermtwo, and colpermone and colpermtwo. The $n \times 2$ matrix axes gives the plotting coordinates for the combined row and column object set. For bicirac, inperm is the single starting permutation for both circular structures.

Because of a later Toolbox now being constructed, we will not give an explicit illustration here of using two fitted structures to represent a proximity matrix. Also, the primary data set we have been using, supreme_agree, is not a particularly good example for multiple structures because only one such device is really needed to explain everything present in the data. More suitable proximity matrices would probably themselves be obtained by a mixture or aggregation of other proximity matrices, reflecting somewhat different underlying structures; hopefully, these could be 'teased apart' in an analysis using multiple additive structures.

8.2 Individual Differences

One aspect of the given M-files introduced in earlier sections but not emphasized, is their possible use in the confirmatory context of fitting individual differences. Explicitly, we begin with a collection of, say, N proximity matrices, $\mathbf{P}_1, \ldots, \mathbf{P}_N$, obtained from N separate sources, and through some weighting and averaging process, construct a single aggregate proximity matrix, $\mathbf{P}_A$. On the basis of $\mathbf{P}_A$, suppose a LUS or CUS structure is constructed; we label the latter the 'common space' consistent with what is usually done in the (weighted) Euclidean model in multidimensional scaling. Each of the N proximity matrices can then be used in a confirmatory fitting of a LUS (with, say, `linfitac.m`) or a CUS (with, say, `cirfitac.m`). A very general 'subject/private space' is generated for each source, and where the coordinates are unique to that source, subject only to the order constraints of the group space. In effect, we would be carrying out an individual differences analysis by using a 'deviation from the mean' philosophy. A group structure is first identified in an exploratory manner from an aggregate proximity matrix; the separate matrices that went into the aggregate are then fit in a confirmatory way, one-by-one. There does not seem to be any particular a priori advantage in trying to carry out this process 'all at once'; to the contrary, the simplicity of the deviation approach and its immediate generalizability to a variety of possible structural representations, holds out the hope of greater substantive interpretability.

8.3 Incorporating Transformations of the Proximities

In the use of either a one- or two-mode proximity matrix, the data were assumed 'as is', and without any preliminary transformation. It was noted that some analyses leading to negative values might be more pleasingly interpretable if an additive constant could be fitted along with the LUS or CUS structures. In other words, the structures fit to proximity matrices then have an invariance with respect to linear transformations of the proximities. A more general transformation will be discussed briefly in a later section where a centroid (metric), fit as part of the whole representational structure, has the effect of double-centering (i.e., making the rows and columns sum to zero). Considering the input proximity matrix deviated from the centroid, zero sums are present within rows or columns. The analysis methods could iterate between fitting a LUS or CUS structure and a centroid, attempting to squeeze out every last bit of VAF. Maybe a more direct strategy (and one that would most likely not affect substantive interpretations materially) would be to initially double-center (either a one- or two-mode matrix), and then treat the later to the analyses we wish to carry out, without again revisiting the double-centering operation during the iterative process.

A more serious consideration of proximity transformation would involve monotonic functions of the type familiar in nonmetric multidimensional scaling. We provide two utilities, `proxmon.m` and `proxmontm.m`, that will allow the user a chance to experiment with these more general transformations for both one- and two-mode proximity matrices (as we did briefly in Section 3.2). The usage is similar for both M-files in providing a monotonically transformed proximity matrix that is closest in a least-squares sense to a given (usually the structurally fitted) matrix:

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

[monproxpermuttm,vaf,diff] = proxmontm(proxpermuttm,fittedtm)
```

Here, `proxpermut` (`proxpermuttm`) is the input proximity matrix (which may have been subjected to an initial row/column permutation, hence the suffix `permut`), and `fitted` (`fittedtm`) is a given target matrix (typically the representational matrix such as the identified ultrametric); the output matrix, `monproxpermut` (`monproxpermuttm`), is closest to `fitted` (`fittedtm`) in a least-squares sense and obeys the order constraints obtained from each pair of entries in (the upper-triangular portion of) `proxpermut` or `proxpermuttm`. As usual, `vaf` denotes 'variance-accounted-for' but here indicates how much variance in `monproxpermut` (`monproxpermuttm`) can be accounted for by `fitted` (`fittedtm`); finally, `diff` is the value of the least-squares loss function and is one-half the squared differences between the entries in `fitted` (`fittedtm`) and `monproxpermut` (`monproxpermuttm`).

8.4 Finding and Fitting Best LUS Structures in the Presence of Missing Proximities

The various M-files discussed thus far have required proximity matrices to be complete in the sense of having all entries present. This was true even for the two-mode case where the between-set proximities are assumed available although all within-set proximities were not. Two different M-files are mentioned here (analogues of `order.m` and `linfitac.m`) allowing some of the proximities in a symmetric matrix to be absent. The missing proximities are identified in an input matrix, `proxmiss`, having the same size as the input proximity matrix, `prox`, but otherwise the syntaxes are the same as earlier:

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

[fit,vaf,addcon] = linfitac_missing(prox,inperm,proxmiss)
```

The `proxmiss` matrix guides the search and fitting process so the missing data are ignored whenever they should be considered in some kind of comparison. Typically, there will be enough other data available that this really doesn't pose any difficulty.

As an illustration of the M-files just introduced, Table 4 provides data on the ten supreme court justices present at some point during the 2005/6 term, and the percentage of times justices disagreed in non-unanimous decisions during the year. (These data were in the *New York Times* on July 2, 2006, as part of a 'first-page, above-the-fold' article bylined by Linda Greenhouse entitled 'Roberts Is at Court's Helm, But He Isn't Yet in Control'.) There is a single missing value in the table between O'Connor (Oc) and Alito (Al) because they shared a common seat for the term until Alito's confirmation by Congress. Roberts (Ro) served the full year as Chief Justice so no missing data entries involve him. As can be seen in the verbatim output to follow, an empirically obtained ordering (presumably from 'left' to 'right') using `order_missing.m` is

	St	So	Br	Gi	Oc	Ke	Ro	Sc	Al	Th
1 St	.00	.28	.32	.31	.43	.62	.74	.70	.87	.76
2 So	.28	.00	.17	.36	.14	.50	.61	.64	.64	.75
3 Br	.32	.17	.00	.36	.29	.57	.56	.59	.65	.70
4 Gi	.31	.36	.36	.00	.43	.47	.52	.61	.59	.72
5 Oc	.43	.14	.29	.43	.00	.43	.33	.29	*	.43
6 Ke	.62	.50	.57	.47	.43	.00	.29	.35	.13	.41
7 Ro	.74	.61	.56	.52	.33	.29	.00	.12	.09	.18
8 Sc	.70	.64	.59	.61	.29	.35	.12	.00	.22	.16
9 Al	.87	.64	.65	.59	*	.13	.09	.22	.00	.17
10 Th	.76	.75	.70	.72	.43	.41	.18	.16	.17	.00

Table 4: Dissimilarities Among Ten Supreme Court Justices for the 2005/6 Term. The Missing Entry Between O'Connor and Alito is Represented With an Asterisk.

$$1\text{:St} \succ 4\text{:Gi} \succ 3\text{:Br} \succ 2\text{:So} \succ 5\text{:Oc} \succ 6\text{:Ke} \succ 7\text{:Ro} \succ 8\text{:Sc} \succ 9\text{:Al} \succ 10\text{:Th}$$

suggesting rather strongly that Kennedy will most likely now occupy the middle position (although possibly shifted somewhat to the right) once O'Connor is removed from the court's deliberations. The best-fitting LUS structure obtained with `linfitac_missing.m` has VAF of 86.78%, and is given in Table 5 plotted with `linearplot`. Notice that because of the missing values in `fit`, the coordinates were entered 'by hand' in the vector `coord` before plotted with `linearplot`.

```
>> load supreme_agree_2005_6.dat
>> load supreme_agree_2005_6_missing.dat
>> supreme_agree_2005_6

supreme_agree_2005_6 =

         0    0.2800    0.3200    0.3100    0.4300    0.6200    0.7400    0.7000    0.8700    0.7600
    0.2800         0    0.1700    0.3600    0.1400    0.5000    0.6100    0.6400    0.6400    0.7500
    0.3200    0.1700         0    0.3600    0.2900    0.5700    0.5600    0.5900    0.6500    0.7000
    0.3100    0.3600    0.3600         0    0.4300    0.4700    0.5200    0.6100    0.5900    0.7200
    0.4300    0.1400    0.2900    0.4300         0    0.4300    0.3300    0.2900         0    0.4300
    0.6200    0.5000    0.5700    0.4700    0.4300         0    0.2900    0.3500    0.1300    0.4100
    0.7400    0.6100    0.5600    0.5200    0.3300    0.2900         0    0.1200    0.0900    0.1800
    0.7000    0.6400    0.5900    0.6100    0.2900    0.3500    0.1200         0    0.2200    0.1600
    0.8700    0.6400    0.6500    0.5900         0    0.1300    0.0900    0.2200         0    0.1700
    0.7600    0.7500    0.7000    0.7200    0.4300    0.4100    0.1800    0.1600    0.1700         0

>> supreme_agree_2005_6_missing

supreme_agree_2005_6_missing =

     0     1     1     1     1     1     1     1     1     1
     1     0     1     1     1     1     1     1     1     1
     1     1     0     1     1     1     1     1     1     1
     1     1     1     0     1     1     1     1     1     1
     1     1     1     1     0     1     1     1     0     1
     1     1     1     1     1     0     1     1     1     1
     1     1     1     1     1     1     0     1     1     1
     1     1     1     1     1     1     1     0     1     1
     1     1     1     1     0     1     1     1     0     1
     1     1     1     1     1     1     1     1     1     0

>> [outperm,rawindex,allperms,index] = ...
order_missing(supreme_agree_2005_6,targlin(10),randperm(10),3,supreme_agree_2005_6_missing);
>> outperm

outperm =

     1     4     3     2     5     6     7     8     9    10

>> [fit, vaf, addcon] = linfitac_missing(supreme_agree_2005_6,outperm,supreme_agree_2005_6_missing)

fit =

         0    0.0967    0.1553    0.1620    0.3146    0.4873    0.5783    0.5783    0.5983    0.6490
    0.0967         0    0.0587    0.0653    0.2179    0.3906    0.4816    0.4816    0.5017    0.5523
    0.1553    0.0587         0    0.0067    0.1593    0.3320    0.4230    0.4230    0.4430    0.4936
    0.1620    0.0653    0.0067         0    0.1526    0.3253    0.4163    0.4163    0.4363    0.4870
    0.3146    0.2179    0.1593    0.1526         0    0.1727    0.2637    0.2637   -0.1567    0.3343
    0.4873    0.3906    0.3320    0.3253    0.1727         0    0.0910    0.0910    0.1110    0.1616
    0.5783    0.4816    0.4230    0.4163    0.2637    0.0910         0         0    0.0200    0.0707
    0.5783    0.4816    0.4230    0.4163    0.2637    0.0910         0         0    0.0200    0.0707
    0.5983    0.5017    0.4430    0.4363   -0.1567    0.1110    0.0200    0.0200         0    0.0506
    0.6490    0.5523    0.4936    0.4870    0.3343    0.1616    0.0707    0.0707    0.0506         0


vaf =

    0.8678

addcon =

   -0.1567

>> coord = [.0000,.0967,.1553,.1620,.3146,.4873,.5783,.5783,.5983,.6490]

coord =

         0    0.0967    0.1553    0.1620    0.3146    0.4873    0.5783    0.5783    0.5983    0.6490

>> inperm = [1 4 3 2 5 6 7 8 9 10]

inperm =

     1     4     3     2     5     6     7     8     9    10

>> [linearlength] = linearplot(coord,inperm)

linearlength =

    0.6490
```

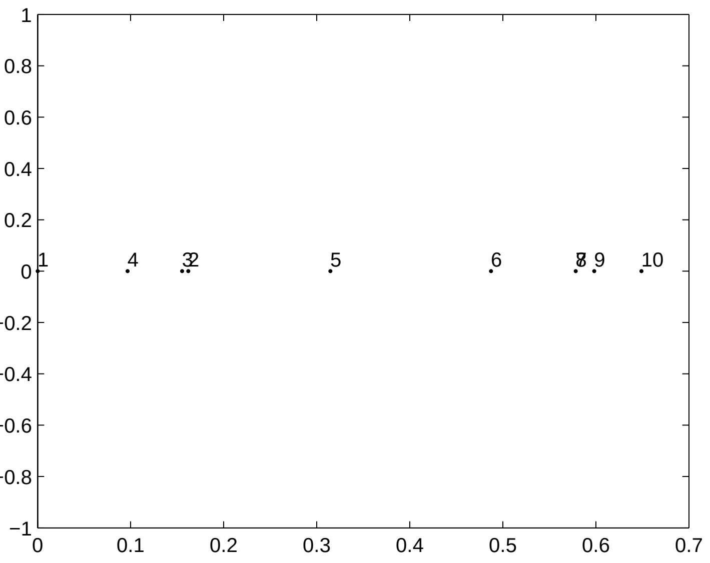


Figure 5: The LUS Representation for the supreme_agree_2005_6 Proximities Using linearplot.m with the Coordinates Constructed from linfitac_missing.m.

8.5 Obtaining Good Object Orders Through a Dynamic Programming Strategy

We have relied on the QA optimization formulation (as in `uniscalqa.m`) to obtain a basic LUS representation (or when necessary, a constraining order). Usually, this usage is sufficient to generate a very good object ordering, especially when the routine is initiated a number of times randomly and the best local optimum chosen. In those instances in which one may wish to explore further the adequacy of a particular ordering in terms of the best achievable (possibly when the proximity matrix is rather large), the M-file, `class_scaledp.m`, is made available. Here, it is possible to form given classes of the object set S to be sequenced (or possibly, to delete some of the objects from consideration altogether), and use a dynamic programming (DP) strategy guaranteeing global optimality for the constructed ordering of the classes. The optimization criterion is the same as in Section 2 (i.e., $\sum_i (t_i^{\rho})^2$), but now the index is taken over the number of object classes formed. Given the limitations on storage demanded by the implemented DP recursion, the method is limited to, say, twenty or fewer object classes.

The syntax for this optimization strategy is

`[permut,cumobfun] = class_scaledp(prox,numbclass,membclass)`

Here, `prox` (as usual) is the $n \times n$ input proximity matrix with a dissimilarity interpretation; `numbclass` is the number of object classes to be sequenced; `membclass` is an $n \times 1$ vector containing the input class membership and includes all the integers from 1 to `numbclass`, with zeros when objects are to be deleted from consideration. The output vectors are `permut` (the order of the classes in the optimal permutation), and `cumobfun` (the cumulative values of the objective function for the successive placements of the objects in the optimal permutation).

In the example below on `supreme_agree`, the number of classes is chosen to be nine, the same as the number of objects; the classes are numbered 1 to 9 just like the original objects, so the `membclass` vector is merely `[1 2 3 4 5 6 7 8 9]`. What can be inferred from the identity permutation being found for `permut` is that we have been using the globally optimum result throughout.

```
>> load supreme_agree.dat
>> [permut,cumobfun] = class_scaledp(supreme_agree,9,[1 2 3 4 5 6 7 8 9])

permut =

     1
     2
     3
     4
     5
     6
     7
     8
     9


cumobfun =

   23.6196
   34.1821
   41.3110
   45.4319
   45.7455
   47.8480
   53.1841
   69.4250
   89.3166
```

8.6 Extending LUS and CUS Representations Through Additively Imposed Centroid Matrices

In the companion Toolbox on Cluster Analysis mentioned earlier, the notion of a matrix representing an additive tree was introduced and characterized by a certain four-point condition that its entries must satisfy. Alternatively, it was noted that any such matrix could be represented (in many ways) as a sum of two matrices, say $\mathbf{U} = \{u_{ij}\}$ and $\mathbf{C} = \{c_{ij}\}$, where $\mathbf{U}$ is an ultrametric matrix (and whose entries satisfy a certain three-point condition), and $c_{ij} = g_i + g_j$ for $1 \leq i \neq j \leq n$, and $c_{ii} = 0$ for $1 \leq i \leq n$, based on some set of values, $g_1, \ldots, g_n$. We will call $\mathbf{C}$ a centroid metric, for convenience (and will continue to do so even though some entries in $\mathbf{C}$ may be negative because of possible negative values for $g_1, \ldots, g_n$).

Computationally, one can construct a best-fitting additive tree matrix by using the sum of an ultrametric and centroid metric, and (through residualization) carry out an iterative fitting strategy using the two structures. As noted earlier, this would try to squeeze out every last bit of VAF we could. The same type of approach could be implemented with a replacement of the ultrametric structure by one based on LUS (or CUS). Whether all of this iterative fitting is really worth it from a substantive interpretation perspective, is questionable. A simpler alternative would be to merely fit best centroid metrics to either the given one- or two-mode proximity matrix; residualize the matrix from the centroid structure; and then treat the residual matrix to whatever representation device one would wish. The syntax for the two centroid fitting M-files is as follows (both implement closed-form expressions for the least-squares solutions):

```
[fit,vaf,lengths] = centfit(prox)

[fit,vaf,lengths] = centfittm(proxtm)
```

In both cases, `fit` is the least-squares approximation matrix with VAF given by `vaf`. For `centfit.m`, the n values defining the centroid metric are given in `lengths`; in `centfittm.m`, the row values are followed by the column values for the defining centroid metric.

As examples in the output below on `supreme_agree` and `supreme_agree5x4`, the residual matrix from the best-fitting centroid is subjected to a LUS. One can still see in the results most of the previously given interpretations. We might note that in the process of residualization, the matrices so produced sum to zero within each row or column, so we have effectively double-centered the matrices by the residualization process.

```
>> load supreme_agree.dat
>> load supreme_agree5x4.dat
>> [fit,vaf,lengths] = centfit(supreme_agree)

fit =

         0    0.6186    0.6043    0.5871    0.5657    0.5557    0.5643    0.6814    0.6829
    0.6186         0    0.4829    0.4657    0.4443    0.4343    0.4429    0.5600    0.5614
    0.6043    0.4829         0    0.4514    0.4300    0.4200    0.4286    0.5457    0.5471
    0.5871    0.4657    0.4514         0    0.4129    0.4029    0.4114    0.5286    0.5300
    0.5657    0.4443    0.4300    0.4129         0    0.3814    0.3900    0.5071    0.5086
    0.5557    0.4343    0.4200    0.4029    0.3814         0    0.3800    0.4971    0.4986
    0.5643    0.4429    0.4286    0.4114    0.3900    0.3800         0    0.5057    0.5071
    0.6814    0.5600    0.5457    0.5286    0.5071    0.4971    0.5057         0    0.6243
    0.6829    0.5614    0.5471    0.5300    0.5086    0.4986    0.5071    0.6243         0


vaf =

    0.1908


lengths =

    0.3700    0.2486    0.2343    0.2171    0.1957    0.1857    0.1943    0.3114    0.3129

>> residual_supreme_agree = supreme_agree - fit

residual_supreme_agree =

         0   -0.2386   -0.2643   -0.2171    0.1043    0.0843    0.1857    0.1786    0.1671
   -0.2386         0   -0.2029   -0.1757    0.0057    0.0957    0.1271    0.1900    0.1986
   -0.2643   -0.2029         0   -0.2314    0.1000    0.0900    0.1414    0.1743    0.1929
   -0.2171   -0.1757   -0.2314         0    0.0371    0.0971    0.1486    0.1614    0.1800
    0.1043    0.0057    0.1000    0.0371         0   -0.0514   -0.1000   -0.0471   -0.0486
    0.0843    0.0957    0.0900    0.0971   -0.0514         0   -0.1500   -0.0771   -0.0886
    0.1857    0.1271    0.1414    0.1486   -0.1000   -0.1500         0   -0.1657   -0.1871
    0.1786    0.1900    0.1743    0.1614   -0.0471   -0.0771   -0.1657         0   -0.4143
    0.1671    0.1986    0.1929    0.1800   -0.0486   -0.0886   -0.1871   -0.4143         0

>> [fit,vaf,lengths] = centfittm(supreme_agree5x4)

fit =

    0.5455    0.5355    0.4975    0.5915
    0.4355    0.4255    0.3875    0.4815
    0.4255    0.4155    0.3775    0.4715
    0.4280    0.4180    0.3800    0.4740
    0.5255    0.5155    0.4775    0.5715


vaf =
    0.1090


lengths =

    0.3080
    0.1980
    0.1880
    0.1905
    0.2880
    0.2375
    0.2275
    0.1895
    0.2835

>> residual_supreme_agree5x4 = supreme_agree5x4 - fit

residual_supreme_agree5x4 =

   -0.2455   -0.1655    0.1425    0.2685
   -0.1555   -0.2055    0.1225    0.2385
    0.0245    0.0345   -0.0475   -0.0115
    0.1420    0.1420   -0.1500   -0.1340
    0.2345    0.1945   -0.0675   -0.3615

>> [outperm,rawindex,allperms,index] = order(residual_supreme_agree,targlin(9),randperm(9),3);
>> outperm

outperm =

     9     8     7     6     5     2     4     3     1

>> [fit,vaf,coord,addcon] = linfitac(residual_supreme_agree,outperm)

fit =

         0         0    0.0420    0.0999    0.1647    0.4064    0.4185    0.4226    0.4226
         0         0    0.0420    0.0999    0.1647    0.4064    0.4185    0.4226    0.4226
    0.0420    0.0420         0    0.0579    0.1227    0.3643    0.3765    0.3806    0.3806
    0.0999    0.0999    0.0579         0    0.0648    0.3065    0.3186    0.3227    0.3227
    0.1647    0.1647    0.1227    0.0648         0    0.2417    0.2538    0.2579    0.2579
    0.4064    0.4064    0.3643    0.3065    0.2417         0    0.0121    0.0162    0.0162
    0.4185    0.4185    0.3765    0.3186    0.2538    0.0121         0    0.0041    0.0041
    0.4226    0.4226    0.3806    0.3227    0.2579    0.0162    0.0041         0         0
    0.4226    0.4226    0.3806    0.3227    0.2579    0.0162    0.0041         0         0


vaf =

    0.9288


coord =

   -0.2196
   -0.2196
   -0.1776
   -0.1198
   -0.0549
    0.1867
    0.1989
    0.2030
    0.2030
```

```
addcon =

    0.2232


>> [outperm,rawindex,allperms,index,squareprox] = ordertm(residual_supreme_agree5x4,targlin(9),randperm(9),3);
>> outperm
outperm =

     6     1     7     2     3     8     4     5     9

>> [fit,vaf,rowperm,colperm,addcon,coord] = linfittmac(residual_supreme_agree5x4,outperm)

fit =

         0    0.0000    0.3363    0.4452
    0.0000         0    0.3363    0.4452
    0.2249    0.2249    0.1114    0.2204
    0.3671    0.3671    0.0308    0.0782
    0.4452    0.4452    0.1089         0


vaf =

    0.9394


rowperm =

     1
     2
     3
     4
     5


colperm =

     1
     2
     3
     4


addcon =

    0.2094


coord =

         0
         0
    0.0000
    0.0000
    0.2249
    0.3363
    0.3671
    0.4452
    0.4452
```

The two M-files, `cent_linearfit.m` and `cent_linearfnd.m`, illustrated below are of the ‘squeezing as much VAF as possible’ variety for the sum of a centroid and a LUS model for a symmetric proximity matrix. The M-files differ in that `cent_linearfnd.m` finds a best order to use for the LUS component; `cent_linearfit.m` allows one to be imposed. The syntax for the two files are:

```
[find,vaf,outperm,targone,targtwo,lengthsone,coordtwo, ...
  addcontwo] = cent_linearfnd(prox,inperm)

[find,vaf,outperm,targone,targtwo,lengthsone,coordtwo, ...
  addcontwo] = cent_linearfit(prox,inperm)
```

Here, `prox` is obviously the input dissimilarity matrix; `inperm` is the given constraining order in `cent_linearfit.m`, and the beginning input order (possibly random) for `cent_linearfnd.m`. For output, `find` is the found least-squares approximation of `prox` with VAF of `vaf`; the found or given constraining order is `outperm`; `targtwo` is the linear unidimensional scaling component of the decomposition defined by the coordinates in `coordtwo` with additive constant `addcontwo`.

```
>> [find,vaf,outperm,targone,targtwo,lengthsone,coordtwo,addcontwo] = cent_linearfnd(supreme_agree,randperm(9))

find =

         0    0.2109    0.3292    0.4051    0.4736    0.7295    0.7064    0.8659    0.7302
    0.2109         0    0.3307    0.4065    0.4750    0.7309    0.7078    0.8673    0.7317
    0.3292    0.3307         0    0.2483    0.3168    0.5727    0.5496    0.7091    0.5734
    0.4051    0.4065    0.2483         0    0.2745    0.5303    0.5072    0.6668    0.5311
    0.4736    0.4750    0.3168    0.2745         0    0.4965    0.4734    0.6329    0.4972
    0.7295    0.7309    0.5727    0.5303    0.4965         0    0.2555    0.4150    0.2794
    0.7064    0.7078    0.5496    0.5072    0.4734    0.2555         0    0.3628    0.2271
    0.8659    0.8673    0.7091    0.6668    0.6329    0.4150    0.3628         0    0.3398
    0.7302    0.7317    0.5734    0.5311    0.4972    0.2794    0.2271    0.3398         0


vaf =

    0.9856


outperm =

     8     9     7     6     5     2     4     1     3


targone =

         0    0.4720    0.4520    0.4688    0.4861    0.5051    0.4674    0.6035    0.4619
    0.4720         0    0.4535    0.4702    0.4875    0.5066    0.4689    0.6050    0.4633
    0.4520    0.4535         0    0.4503    0.4676    0.4866    0.4489    0.5850    0.4434
    0.4688    0.4702    0.4503         0    0.4844    0.5034    0.4657    0.6018    0.4601
    0.4861    0.4875    0.4676    0.4844         0    0.5207    0.4830    0.6191    0.4775
    0.5051    0.5066    0.4866    0.5034    0.5207         0    0.5020    0.6381    0.4965
    0.4674    0.4689    0.4489    0.4657    0.4830    0.5020         0    0.6004    0.4588
    0.6035    0.6050    0.5850    0.6018    0.6191    0.6381    0.6004         0    0.5949
    0.4619    0.4633    0.4434    0.4601    0.4775    0.4965    0.4588    0.5949         0


targtwo =

         0         0    0.1383    0.1974    0.2486    0.4855    0.5000    0.5235    0.5295
         0         0    0.1383    0.1974    0.2486    0.4855    0.5000    0.5235    0.5295
    0.1383    0.1383         0    0.0591    0.1103    0.3472    0.3617    0.3852    0.3912
    0.1974    0.1974    0.0591         0    0.0512    0.2881    0.3026    0.3261    0.3321
    0.2486    0.2486    0.1103    0.0512         0    0.2369    0.2514    0.2749    0.2809
    0.4855    0.4855    0.3472    0.2881    0.2369         0    0.0146    0.0380    0.0440
    0.5000    0.5000    0.3617    0.3026    0.2514    0.0146         0    0.0234    0.0294
    0.5235    0.5235    0.3852    0.3261    0.2749    0.0380    0.0234         0    0.0060
    0.5295    0.5295    0.3912    0.3321    0.2809    0.0440    0.0294    0.0060         0


lengthsone =

    0.2353    0.2367    0.2168    0.2335    0.2508    0.2699    0.2322    0.3683    0.2266


coordtwo =

   -0.2914
   -0.2914
   -0.1531
   -0.0940
   -0.0428
    0.1940
    0.2086
    0.2321
    0.2380


addcontwo =

    0.2611

>> [find,vaf,outperm,targone,targtwo,lengthsone,coordtwo,addcontwo] = ...
   cent_linearfit(supreme_agree,[1 2 3 4 5 6 7 8 9])

find =

         0    0.3798    0.3707    0.3793    0.6305    0.6644    0.7067    0.8635    0.8650
    0.3798         0    0.2492    0.2578    0.5090    0.5429    0.5852    0.7420    0.7435
    0.3707    0.2492         0    0.2427    0.4939    0.5278    0.5701    0.7269    0.7284
    0.3793    0.2578    0.2427         0    0.4665    0.5003    0.5427    0.6995    0.7009
    0.6305    0.5090    0.4939    0.4665         0    0.2745    0.3168    0.4736    0.4750
    0.6644    0.5429    0.5278    0.5003    0.2745         0    0.2483    0.4051    0.4065
    0.7067    0.5852    0.5701    0.5427    0.3168    0.2483         0    0.3292    0.3307
    0.8635    0.7420    0.7269    0.6995    0.4736    0.4051    0.3292         0    0.2109
    0.8650    0.7435    0.7284    0.7009    0.4750    0.4065    0.3307    0.2109         0


vaf =

    0.9841
```

```
outperm =

     1     2     3     4     5     6     7     8     9


targone =

         0    0.6241    0.6120    0.6026    0.6153    0.5980    0.5812    0.5997    0.6012
    0.6241         0    0.5035    0.4941    0.5068    0.4895    0.4727    0.4913    0.4927
    0.6120    0.5035         0    0.4821    0.4947    0.4774    0.4606    0.4792    0.4806
    0.6026    0.4941    0.4821         0    0.4853    0.4680    0.4512    0.4698    0.4712
    0.6153    0.5068    0.4947    0.4853         0    0.4806    0.4639    0.4824    0.4838
    0.5980    0.4895    0.4774    0.4680    0.4806         0    0.4466    0.4651    0.4665
    0.5812    0.4727    0.4606    0.4512    0.4639    0.4466         0    0.4483    0.4498
    0.5997    0.4913    0.4792    0.4698    0.4824    0.4651    0.4483         0    0.4683
    0.6012    0.4927    0.4806    0.4712    0.4838    0.4665    0.4498    0.4683         0


targtwo =

         0    0.0130    0.0160    0.0341    0.2726    0.3238    0.3829    0.5212    0.5212
    0.0130         0    0.0030    0.0211    0.2596    0.3108    0.3699    0.5082    0.5082
    0.0160    0.0030         0    0.0180    0.2566    0.3078    0.3669    0.5051    0.5051
    0.0341    0.0211    0.0180         0    0.2385    0.2897    0.3488    0.4871    0.4871
    0.2726    0.2596    0.2566    0.2385         0    0.0512    0.1103    0.2486    0.2486
    0.3238    0.3108    0.3078    0.2897    0.0512         0    0.0591    0.1974    0.1974
    0.3829    0.3699    0.3669    0.3488    0.1103    0.0591         0    0.1383    0.1383
    0.5212    0.5082    0.5051    0.4871    0.2486    0.1974    0.1383         0         0
    0.5212    0.5082    0.5051    0.4871    0.2486    0.1974    0.1383         0         0


lengthsone =

    0.3663    0.2578    0.2457    0.2363    0.2490    0.2317    0.2149    0.2334    0.2349


coordtwo =

   -0.2316
   -0.2186
   -0.2156
   -0.1976
    0.0410
    0.0922
    0.1513
    0.2895
    0.2895


addcontwo =

    0.2574
```

8.7 Fitting the LUS Model Through Partitions Consistent With a Given Object Order

To show there may be several ways to approach a particular (least-squares) fitting task, a general M-file is available, `partitionfit.m`, that provides a least-squares approximation to a proximity matrix based on a given collection of partitions. In the syntax

```
[fitted,vaf,weights,end_condition] = partitionfit(prox,member)
```

the input dissimilarity matrix is `prox`; `member` is the $m \times n$ matrix indicating cluster membership, where each row corresponds to a specific partition (there are m partitions in general); the columns of `member` are in the same input order used for `prox`. For output, `fitted` is an $n \times n$ matrix approximating `prox` (through least-squares) constructed from the nonnegative `weights` vector corresponding to the partitions. The VAF value, `vaf`, is for the proximity matrix, `prox`, compared to `fitted`. The `end_condition` flag should be zero for a normal termination.

As an example below, the least-squares fitting of the identity permutation on `supreme_agree` with `linfitac.m` is replicated with `partitionfit.m`. The central matrix is `member`, where the first eight rows correspond to the eight separations between the justices along the line (in the jargon of graph theory, we have eight 'cuts' of a graph, each defined by two disjoint [and exhaustive] subsets, and characterized by a 0/1 dissimilarity matrix with 1's indicating objects present across the two separate subsets). The last row of `member` is the disjoint partition representing an additive constant, and producing a single 0/1 dissimilarity matrix with all 1's in the off-diagonal positions. Thus, to move from one object to another along the continuum, the various 'gaps' must be traversed separating the two objects. To construct the approximation, the weights attached to the gaps are summed to produce the complete path; an additional additive constant is then imposed. Because we are using nonnegative least-squares to obtain that weights and the additive constant, an obtained zero value for the additive constant (i.e., we have an estimation at the boundary) suggests the need to augment the original proximities by a positive value before `partitionfit.m` is used.

```
>> load supreme_agree.dat
>> [fit,vaf,coord,addcon] = linfitac(supreme_agree,1:9)

fit =

         0    0.1304    0.1464    0.1691    0.4085    0.4589    0.5060    0.6483    0.6483
    0.1304         0    0.0160    0.0387    0.2780    0.3285    0.3756    0.5179    0.5179
    0.1464    0.0160         0    0.0227    0.2620    0.3124    0.3596    0.5019    0.5019
    0.1691    0.0387    0.0227         0    0.2393    0.2898    0.3369    0.4792    0.4792
    0.4085    0.2780    0.2620    0.2393         0    0.0504    0.0976    0.2399    0.2399
    0.4589    0.3285    0.3124    0.2898    0.0504         0    0.0471    0.1894    0.1894
    0.5060    0.3756    0.3596    0.3369    0.0976    0.0471         0    0.1423    0.1423
    0.6483    0.5179    0.5019    0.4792    0.2399    0.1894    0.1423         0         0
    0.6483    0.5179    0.5019    0.4792    0.2399    0.1894    0.1423         0         0


vaf =

    0.9796


coord =

   -0.3462
   -0.2158
   -0.1998
   -0.1771
    0.0622
    0.1127
    0.1598
    0.3021
    0.3021


addcon =

   -0.2180

>> member = [1 9 9 9 9 9 9 9 9;1 1 9 9 9 9 9 9 9;1 1 1 9 9 9 9 9 9;1 1 1 1 9 9 9 9 9;1 1 1 1 1 9 9 9 9;
1 1 1 1 1 1 9 9 9;1 1 1 1 1 1 1 9 9;1 1 1 1 1 1 1 1 9;1 2 3 4 5 6 7 8 9]

member =

     1     9     9     9     9     9     9     9     9
     1     1     9     9     9     9     9     9     9
     1     1     1     9     9     9     9     9     9
     1     1     1     1     9     9     9     9     9
     1     1     1     1     1     9     9     9     9
     1     1     1     1     1     1     9     9     9
     1     1     1     1     1     1     1     9     9
     1     1     1     1     1     1     1     1     9
     1     2     3     4     5     6     7     8     9

>> [fitted,vaf,weights,end_condition] = partitionfit(supreme_agree,member)

fitted =

         0    0.3485    0.3645    0.3871    0.6264    0.6769    0.7240    0.8663    0.8663
    0.3485         0    0.2340    0.2567    0.4960    0.5464    0.5936    0.7359    0.7359
    0.3645    0.2340         0    0.2407    0.4800    0.5305    0.5776    0.7199    0.7199
    0.3871    0.2567    0.2407         0    0.4574    0.5078    0.5549    0.6972    0.6972
    0.6264    0.4960    0.4800    0.4574         0    0.2685    0.3156    0.4579    0.4579
    0.6769    0.5464    0.5305    0.5078    0.2685         0    0.2651    0.4075    0.4075
    0.7240    0.5936    0.5776    0.5549    0.3156    0.2651         0    0.3603    0.3603
    0.8663    0.7359    0.7199    0.6972    0.4579    0.4075    0.3603         0    0.2180
    0.8663    0.7359    0.7199    0.6972    0.4579    0.4075    0.3603    0.2180         0


vaf =

    0.9796
```

```
weights =

    0.1304
    0.0160
    0.0227
    0.2393
    0.0504
    0.0471
    0.1423
         0
    0.2180


end_condition =

     0
```

9 Comparing Categorical (Ultrametric) and Continuous (LUS) Representations for a Proximity Matrix

One of the basic tasks of data analysis for proximity matrices lies in the choice of representation, and in particular, whether it should be 'continuous', as reflected in LUS, or 'categorical', as in the construction of a best-fitting ultrametric. These latter discrete or categorical models are the main topic of a companion Cluster Analysis Toolbox, and the reader is referred to this source for specifics. Here, we demonstrate the use of imposing a constraining object order on the analysis performed that is either given (in `cat_vs_con_orderfit.m`), or is found (in `cat_vs_con_orderfnd.m`). In either case, a best-fitting anti-Robinson (AR) matrix is first identified based on the constraining order (either given or found), recalling that an AR matrix is characterized by its entries never decreasing (and usually increasing) as we move away from the main diagonal within a row or a column. Treating this latter AR matrix as if it were the input proximity matrix, both a best-fitting LUS and ultrametric structure is then identified, respecting the given or found constraining order. We note, in particular, that the AR constraints imposed are weaker than those for a LUS or ultrametric model, and the AR defining inequalities are actually implicit in those for the stricter representations. This implies that one can proceed, without loss of any generality, to obtain a LUS or ultrametric structures from the best-fitting AR matrix treated as the input proximity matrix. The 'successive averaging' necessary for a least-squares AR matrix is also part of and is needed to generate best LUS or ultrametric approximations.

Generally, LUS and ultrametric structures can themselves be put into AR forms. So, in this sense, both continuous and discrete representations are part of a broader representational device that places an upper-bound on how well a given proximity matrix can be represented by either a continuous or discrete structure. For example, in its use below on the `supreme_agree` data matrix, `cat_vs_con_orderfnd.m` finds the identity permutation as the constraining permutation and gives a VAF of 99.55% for the best-fitting AR matrix. The LUS model VAF of 97.96% is trivially less than that for the AR form; in contrast, the ultrametric VAF of 73.69% is quite a drop. Given these comparisons, one might argue that a continuous representation does much better than a categorical one, at least for this particular proximity matrix.

The syntax for the two M-files is very similar:

```
[findultra,vafultra,vafarob,arobprox,fitlinear,vaflinear,...
    coord,addcon] = cat_vs_con_orderfit(prox,inperm,conperm)

[findultra,vafultra,conperm,vafarob,arobprox,fitlinear,...
    vaflinear,coord,addcon] = cat_vs_con_orderfnd(prox,inperm)
```

As usual, `prox` is the input dissimilarity matrix and `inperm` is a starting permutation for how the ultrametric constraints are searched for; in `cat_vs_con_orderfnd.m`, `inperm` also initializes the search for a constraining order. The permutation, `conperm`, in `cat_vs_con_orderfit.m` is the given constraining order. As output, `findultra` is the best ultrametric found with VAF of `vafultra`; `arobprox` is the best AR form identified with a VAF of `vafarob`; `fitlinear` is the best LUS model with VAF of `vaflinear` with `coord` containing the coordinates and `addcon` the additive constant. For `cat_vs_con_orderfnd.m`, the identified constraining order, `conperm`, is also given as an output vector.

```
>> load supreme_agree.dat
>> [findultra,vafultra,conperm,vafarob,arobprox,fitlinear,vaflinear, ...
coord,addcon] = cat_vs_con_orderfnd(supreme_agree,randperm(9))
```

```
findultra =

         0    0.3633    0.3633    0.3633    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633         0    0.2850    0.2850    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2850         0    0.2200    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2850    0.2200         0    0.6405    0.6405    0.6405    0.6405    0.6405
    0.6405    0.6405    0.6405    0.6405         0    0.3100    0.3100    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100         0    0.2300    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100    0.2300         0    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.4017    0.4017    0.4017         0    0.2100
    0.6405    0.6405    0.6405    0.6405    0.4017    0.4017    0.4017    0.2100         0


vafultra =

    0.7369


conperm =

     1     2     3     4     5     6     7     8     9


vafarob =

    0.9955


arobprox =

         0    0.3600    0.3600    0.3700    0.6550    0.6550    0.7500    0.8550    0.8550
    0.3600         0    0.2800    0.2900    0.4900    0.5300    0.5700    0.7500    0.7600
    0.3600    0.2800         0    0.2200    0.4900    0.5100    0.5700    0.7200    0.7400
    0.3700    0.2900    0.2200         0    0.4500    0.5000    0.5600    0.6900    0.7100
    0.6550    0.4900    0.4900    0.4500         0    0.3100    0.3100    0.4600    0.4600
    0.6550    0.5300    0.5100    0.5000    0.3100         0    0.2300    0.4150    0.4150
    0.7500    0.5700    0.5700    0.5600    0.3100    0.2300         0    0.3300    0.3300
    0.8550    0.7500    0.7200    0.6900    0.4600    0.4150    0.3300         0    0.2100
    0.8550    0.7600    0.7400    0.7100    0.4600    0.4150    0.3300    0.2100         0


fitlinear =

         0    0.1304    0.1464    0.1691    0.4085    0.4589    0.5060    0.6483    0.6483
    0.1304         0    0.0160    0.0387    0.2780    0.3285    0.3756    0.5179    0.5179
    0.1464    0.0160         0    0.0227    0.2620    0.3124    0.3596    0.5019    0.5019
    0.1691    0.0387    0.0227         0    0.2393    0.2898    0.3369    0.4792    0.4792
    0.4085    0.2780    0.2620    0.2393         0    0.0504    0.0976    0.2399    0.2399
    0.4589    0.3285    0.3124    0.2898    0.0504         0    0.0471    0.1894    0.1894
    0.5060    0.3756    0.3596    0.3369    0.0976    0.0471         0    0.1423    0.1423
    0.6483    0.5179    0.5019    0.4792    0.2399    0.1894    0.1423         0         0
    0.6483    0.5179    0.5019    0.4792    0.2399    0.1894    0.1423         0         0


vaflinear =

    0.9796


coord =

   -0.3462
   -0.2158
   -0.1998
   -0.1771
    0.0622
    0.1127
    0.1598
    0.3021
    0.3021


addcon =

   -0.2180
```

There is one somewhat unresolved issue as to whether the LUS and ultrametric representations incorporate the same number of 'weights', because otherwise, direct comparison of VAF values may be considered problematic. We argue that, indeed, the number of weights are the same; there are $n-1$ distinct values in an ultrametric matrix and $n-1$ separations along a line in a LUS model. The one additional additive constant for LUS that is needed to insure invariance to linear transformations of the proximities, should not count against the representation. The ultrametric model automatically has such invariance, and should not be given an inherent advantage just because of this.

9.1 Comparing Equally-Spaced Versus Unrestricted Representations for a Proximity Matrix

The fitting strategies offered by `linfitac.m` and `cirfitac.m` (as well as `ultrafit.m` from the companion Cluster Analysis Toolbox), all allow unequal spacings to generate the least-squares approximations. This, in effect, requires multiple weights to be constructed. At times, it may be of interest to see how a much simpler model might fair, based only on one 'free weight'. In LUS, we would have equal spacings along a line; for CUS, there would be equal spacings around a circular structure; and for an ultrametric, only multiples of the integer-valued levels (typically, n minus the number of classes in a partition) at which new subsets are formed. In the examples below of `eqspace_linfitac.m`, `eqspace_cirfitac.m`, and `eqspace_ultrafit.m`, the addition of the prefix '`eq`' shows the `vaf`, `fit`, or `addcon` for the equally-spaced alternatives. (All of the latter, we might add, are based on simple regression, rather than on any iterative fitting strategy.) The usual non-equally-spaced alternatives are also given for comparison. We found the high

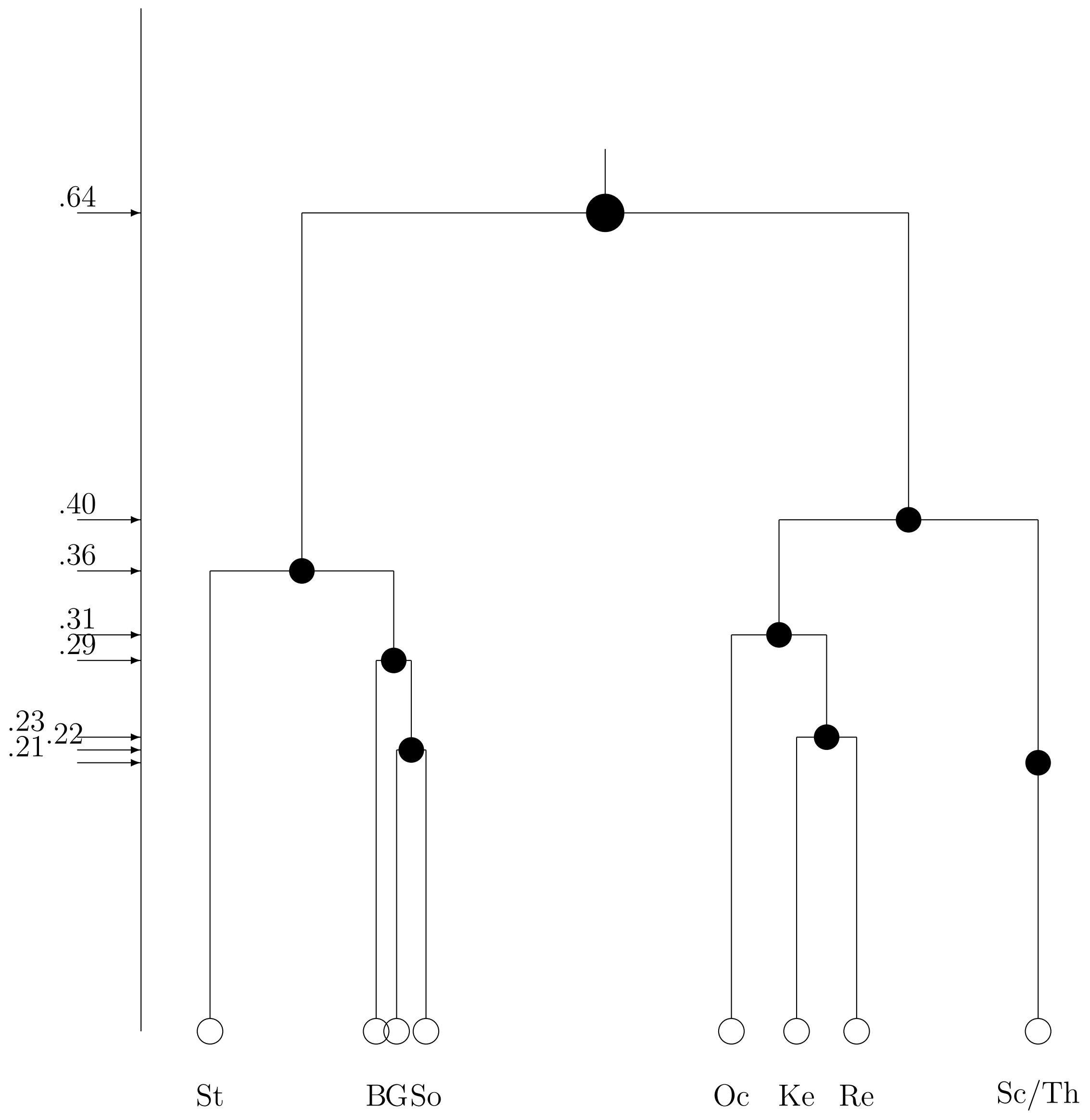
VAF of 83.83% interesting for the equally-spaced LUS model; it is remarkable that only one weight is necessary to generate such a value.

```
>> load supreme_agree.dat
>> [fit, vaf, coord, addcon, eqfit, eqvaf, eqaddcon] = ...
           eqspace_linfitac(supreme_agree,1:9)

fit =

         0    0.1304    0.1464    0.1691    0.4085    0.4589    0.5060    0.6483    0.6483
    0.1304         0    0.0160    0.0387    0.2780    0.3285    0.3756    0.5179    0.5179
    0.1464    0.0160         0    0.0227    0.2620    0.3124    0.3596    0.5019    0.5019
    0.1691    0.0387    0.0227         0    0.2393    0.2898    0.3369    0.4792    0.4792
    0.4085    0.2780    0.2620    0.2393         0    0.0504    0.0976    0.2399    0.2399
    0.4589    0.3285    0.3124    0.2898    0.0504         0    0.0471    0.1894    0.1894
    0.5060    0.3756    0.3596    0.3369    0.0976    0.0471         0    0.1423    0.1423
    0.6483    0.5179    0.5019    0.4792    0.2399    0.1894    0.1423         0         0
    0.6483    0.5179    0.5019    0.4792    0.2399    0.1894    0.1423         0         0


vaf =

    0.9796


coord =

   -0.3462
   -0.2158
   -0.1998
   -0.1771
    0.0622
    0.1127
    0.1598
    0.3021
    0.3021


addcon =

   -0.2180


eqfit =

         0    0.0859    0.1718    0.2577    0.3436    0.4295    0.5154    0.6013    0.6872
    0.0859         0    0.0859    0.1718    0.2577    0.3436    0.4295    0.5154    0.6013
    0.1718    0.0859         0    0.0859    0.1718    0.2577    0.3436    0.4295    0.5154
    0.2577    0.1718    0.0859         0    0.0859    0.1718    0.2577    0.3436    0.4295
    0.3436    0.2577    0.1718    0.0859         0    0.0859    0.1718    0.2577    0.3436
    0.4295    0.3436    0.2577    0.1718    0.0859         0    0.0859    0.1718    0.2577
    0.5154    0.4295    0.3436    0.2577    0.1718    0.0859         0    0.0859    0.1718
    0.6013    0.5154    0.4295    0.3436    0.2577    0.1718    0.0859         0    0.0859
    0.6872    0.6013    0.5154    0.4295    0.3436    0.2577    0.1718    0.0859         0


eqvaf =

    0.8383


eqaddcon =
   -0.2181

>> load morse_digits.dat
>> [fit, vaf, addcon, eqfit, eqvaf, eqaddcon] = ...
            eqspace_cirfitac(morse_digits,[4 5 6 7 8 9 10 1 2 3])

fit =

         0    0.0247    0.3620    0.6413    0.9605    1.1581    1.1581    1.0358    0.7396    0.3883
    0.0247         0    0.3373    0.6165    0.9358    1.1334    1.1334    1.0606    0.7643    0.4131
    0.3620    0.3373         0    0.2793    0.5985    0.7961    0.7961    1.0148    1.1016    0.7503
    0.6413    0.6165    0.2793         0    0.3193    0.5169    0.5169    0.7355    1.0318    1.0296
    0.9605    0.9358    0.5985    0.3193         0    0.1976    0.1976    0.4163    0.7125    1.0638
    1.1581    1.1334    0.7961    0.5169    0.1976         0    0.0000    0.2187    0.5149    0.8662
    1.1581    1.1334    0.7961    0.5169    0.1976    0.0000         0    0.2187    0.5149    0.8662
    1.0358    1.0606    1.0148    0.7355    0.4163    0.2187    0.2187         0    0.2963    0.6475
    0.7396    0.7643    1.1016    1.0318    0.7125    0.5149    0.5149    0.2963         0    0.3513
    0.3883    0.4131    0.7503    1.0296    1.0638    0.8662    0.8662    0.6475    0.3513         0


vaf =

    0.7190


addcon =

   -0.7964


eqfit =

         0    0.2208    0.4416    0.6624    0.8833    1.1041    0.8833    0.6624    0.4416    0.2208
    0.2208         0    0.2208    0.4416    0.6624    0.8833    1.1041    0.8833    0.6624    0.4416
    0.4416    0.2208         0    0.2208    0.4416    0.6624    0.8833    1.1041    0.8833    0.6624
    0.6624    0.4416    0.2208         0    0.2208    0.4416    0.6624    0.8833    1.1041    0.8833
    0.8833    0.6624    0.4416    0.2208         0    0.2208    0.4416    0.6624    0.8833    1.1041
    1.1041    0.8833    0.6624    0.4416    0.2208         0    0.2208    0.4416    0.6624    0.8833
    0.8833    1.1041    0.8833    0.6624    0.4416    0.2208         0    0.2208    0.4416    0.6624
    0.6624    0.8833    1.1041    0.8833    0.6624    0.4416    0.2208         0    0.2208    0.4416
    0.4416    0.6624    0.8833    1.1041    0.8833    0.6624    0.4416    0.2208         0    0.2208
    0.2208    0.4416    0.6624    0.8833    1.1041    0.8833    0.6624    0.4416    0.2208         0


eqvaf =

    0.5518


eqaddcon =

   -0.8371

>> load sc_completelink_integertarget.dat
>> sc_completelink_integertarget

sc_completelink_integertarget =

     0     6     6     6     8     8     8     8     8
     6     0     4     4     8     8     8     8     8
     6     4     0     2     8     8     8     8     8
     6     4     2     0     8     8     8     8     8
     8     8     8     8     0     5     5     7     7
     8     8     8     8     5     0     3     7     7
     8     8     8     8     5     3     0     7     7
     8     8     8     8     7     7     7     0     1
     8     8     8     8     7     7     7     1     0

>> [fit,vaf,eqfit,eqvaf] = eqspace_ultrafit(supreme_agree,sc_completelink_integertarget)

fit =

         0    0.3633    0.3633    0.3633    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633         0    0.2850    0.2850    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2850         0    0.2200    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2850    0.2200         0    0.6405    0.6405    0.6405    0.6405    0.6405
    0.6405    0.6405    0.6405    0.6405         0    0.3100    0.3100    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100         0    0.2300    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100    0.2300         0    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.4017    0.4017    0.4017         0    0.2100
    0.6405    0.6405    0.6405    0.6405    0.4017    0.4017    0.4017    0.2100         0


vaf =

    0.7369


eqfit =

         0    0.4601    0.4601    0.4601    0.6135    0.6135    0.6135    0.6135    0.6135
    0.4601         0    0.3067    0.3067    0.6135    0.6135    0.6135    0.6135    0.6135
    0.4601    0.3067         0    0.1534    0.6135    0.6135    0.6135    0.6135    0.6135
    0.4601    0.3067    0.1534         0    0.6135    0.6135    0.6135    0.6135    0.6135
    0.6135    0.6135    0.6135    0.6135         0    0.3834    0.3834    0.5368    0.5368
    0.6135    0.6135    0.6135    0.6135    0.3834         0    0.2300    0.5368    0.5368
    0.6135    0.6135    0.6135    0.6135    0.3834    0.2300         0    0.5368    0.5368
    0.6135    0.6135    0.6135    0.6135    0.5368    0.5368    0.5368         0    0.0767
    0.6135    0.6135    0.6135    0.6135    0.5368    0.5368    0.5368    0.0767         0


eqvaf =

    0.5927
```

9.2 Representing an Order-Constrained LUS and an Ultrametric on the Same Graph

Whenever the same constraining object order is used to generate both a LUS and ultrametric structure, it is possible to represent them jointly within the same graphical display. The usual dendrogram showing when the new groups form in the hierarchical clustering is given for the ultrametric, but the latter also has its terminal nodes separated according to the coordinates constructed for the LUS. An example is given in Figure 6 using the `supreme_agree` data and the earlier analyses given with `cat_vs_con_orderfnd`. We believe this provides a very nice combined representation for both the discrete and continuous models found for a proximity matrix.

Figure 6: A Joint Order-Constrained LUS and Ultrametric Representation for the supreme_agree Proximity Matrix



10 Some Bibliographic Comments

There are a number of book-length presentations of (multi)dimensional scaling methods available (encompassing differing collections of subtopics within the field). We list several of the better ones to consult in the reference section to follow, and note these here in chronological order: Kruskal & Wish (1978); Shiffman, Reynolds, & Young (1981); Everitt & Rabe-Hesketh (1997); Carroll & Arabie (1998); Cox & Cox (2001); Borg & Groenen (2005); Lattin, Carroll, & Green (2003); Hand (2004). The items that would be closest to the approaches taken here with MATLAB and the emphasis on least-squares, would be the monograph by Hubert, Arabie, and Meulman (2006), and the reviews by Hubert, Arabie, & Meulman (1997; 2001; 2003); Hubert & Steinley (2005); Steinley & Hubert (in review).

References

[1] Borg, I., & Gronenen, P. J. F. (2005). *Modern multidimensional scaling* (2nd Ed.). New York: Springer.

[2] Carroll, J. D., & Arabie, P. (1998). Multidimensional scaling. In M. H. Birnbaum (Ed.), *Handbook of perception and cognition, Vol. 3* (pp. 179–250). San Diego: Academic Press.

[3] Cox, T. F., & Cox, M. A. A. (2001). *Multidimensional scaling* (2nd Ed.). Boca Raton, FL: Chapman and Hall/CRC.

[4] Everitt, B. S., & Rabe-Hesketh, S. (1997). *The analysis of proximity data.* New York: Wiley.

[5] Hand, D. J. (2004). *Measurement theory and practice.* New York: Oxford University Press.

[6] Hubert, L. J., Arabie, P., & Meulman, J. J. (1997). Linear and circular unidimensional scaling for symmetric proximity matrices. *British Journal of Mathematical and Statistical Psychology*, *50*, 253–284.

[7] Hubert, L., Arabie, P., & Meulman, J. (2001). *Combinatorial data analysis: Optimization by dynamic programming.* SIAM Monographs on Discrete Mathematics and Applications. Philadelphia: SIAM.

[8] Hubert, L. J., Arabie, P., & Meulman, J. J. (2002). Linear unidimensional scaling in the L_2-norm: Basic optimization methods using MATLAB. *Journal of Classification*, *19*, 303–328.

[9] Hubert, L., Arabie, P., & Meulman, J. (2006). *The structural representation of proximity matrices with MATLAB.* ASA-SIAM Series on Statistics and Applied Probability. Philadelphia: SIAM.

[10] Hubert, L., & Steinley, D. (2005). Agreement among Supreme Court justices: Categorical vs. continuous representation. *SIAM News*, *38(8)*, 4–7.

[11] Lattin, J., Carroll, J. D., & Green, P. E. (2003). *Analyzing multivariate data.* Pacific Grove, CA: Brooks/Cole.

[12] Steinley, D., & Hubert, L. (in review). The construction of order-constrained partitions with MATLAB: Unidimensional precedents.

[13] Kruskal, J. B., & Wish, M. (1978). *Multidimensional scaling.* Newbury Park, CA: Sage.

[14] Schiffman, S. S., Reynolds, M. L., & Young, F. W. (1981). *Introduction to multidimensional scaling.* New York: Academic Press.

[15] Wollan, P. C., & Dykstra, R. L. (1987). Minimizing linear inequality constrained Mahalanobis distances. *Applied Statistics*, *36*, 234–240.

A Header Comments for the M-files Mentioned in the Text or Used Internally by Other M-files; Given in Alphabetical Order

arobfit.m

```
% AROBFIT fits an anti-Robinson matrix using iterative projection to
% a symmetric proximity matrix in the $L_{2}$-norm.
%
% syntax: [fit, vaf] = arobfit(prox, inperm)
%
% PROX is the input proximity matrix ($n \times n$ with a zero main
% diagonal and a dissimilarity interpretation);
% INPERM is a given permutation of the first $n$ integers;
% FIT is the least-squares optimal matrix (with variance-
% accounted-for of VAF) to PROX having an anti-Robinson form for
% the row and column object ordering given by INPERM.
```

arobfnd.m

```
% AROBFND finds and fits an anti-Robinson
% matrix using iterative projection to
% a symmetric proximity matrix in the $L_{2}$-norm based on a
% permutation identified through the use of iterative quadratic
% assignment.
%
% syntax: [find, vaf, outperm] = arobfnd(prox, inperm, kblock)
%
% PROX is the input proximity matrix ($n \times n$ with a zero main
% diagonal and a dissimilarity interpretation);
% INPERM is a given starting permutation of the first $n$ integers;
% FIND is the least-squares optimal matrix (with
% variance-accounted-for of VAF) to PROX having an anti-Robinson
% form for the row and column object ordering given by the ending
% permutation OUTPERM. KBLOCK defines the block size in the use of the
% iterative quadratic assignment routine.
```

bicirac.m

```
% BICIRAC finds and fits the sum of two circular
% unidimensional scales using iterative projection to
% a symmetric proximity matrix in the $L_{2}$-norm based on
% permutations identified through the use
% of iterative quadratic assignment.
%
% syntax: [find,vaf,targone,targtwo,outpermone,outpermtwo, ...
%     addconone,addcontwo] = bicirac(prox,inperm,kblock)
%
% PROX is the input proximity matrix ($n \times n$ with a zero
% main diagonal and a dissimilarity interpretation);
% INPERM is a given starting permutation of the first $n$ integers;
% FIND is the least-squares optimal matrix (with variance-
% accounted-for of VAF) to PROX and is the sum of the two
% circular anti-Robinson matrices;
% TARGONE and TARGTWO are based on the two row and column
% object orderings given by the ending permutations OUTPERMONE
% and OUTPERMTWO. KBLOCK defines the block size in the use of the
% iterative quadratic assignment routine and ADDCONONE and ADDCONTWO
% are the two additive constants for the two model components.
```

biscalqa.m

```
%  BISCALQA carries out a bidimensional scaling of a symmetric
%  proximity matrix using iterative quadratic assignment.
%
%  syntax: [outpermone,outpermtwo,coordone,coordtwo,fitone,fittwo,...
%    addconone,addcontwo,vaf] = ...
%     biscalqa(prox,targone,targtwo,inpermone,inpermtwo,kblock,nopt)
%
%  PROX is the input proximity matrix (with a zero main diagonal
%  and a dissimilarity interpretation);
%  TARGONE is the input target matrix for the first dimension
%  (usually with a zero main diagonal and a dissimilarity
%  interpretation representing equally spaced locations along
%  a continuum); TARGTWO is the input target
%  matrix for the second dimension;
%  INPERMONE is the input beginning permutation for the first
%  dimension (a permutation of the first $n$ integers);
%  INPERMTWO is the input beginning
%  permutation for the second dimension;
%  the insertion and rotation routines use from 1 to KBLOCK
%  (which is less than or equal to $n-1$) consecutive objects in
%  the permutation defining the row and column orders of the data
%  matrix. NOPT controls the confirmatory or exploratory fitting
%  of the unidimensional scales; a value of NOPT = 0 will fit in a
%  confirmatory manner the two scales
%  indicated by INPERMONE and INPERMTWO;
%  a value of NOPT = 1 uses iterative QA
%  to locate the better permutations to fit;
%  OUTPERMONE is the final object permutation for the
%  first dimension; OUTPERMTWO is the final object permutation
%  for the second dimension;
%  COORDONE is the set of first dimension coordinates
%  in ascending order; COORDTWO is the set of second dimension
%  coordinates in ascending order;
%  ADDCONONE is the additive constant for the first
%  dimensional model; ADDCONTWO is the additive constant for
%  the second dimensional model;
%  VAF is the variance-accounted-for in PROX by
%  the bidimensional scaling.
```

biscaltmac.m

```
% BISCALTMAC finds and fits the sum of two linear
% unidimensional scales using iterative projection to
% a two-mode proximity matrix in the $L_{2}$-norm based on
% permutations identified through the use of iterative quadratic
% assignment.
%
% syntax: [find,vaf,targone,targtwo,outpermone,outpermtwo, ...
%        rowpermone,colpermone,rowpermtwo,colpermtwo,addconone,...
%        addcontwo,coordone,coordtwo,axes] = ...
%    biscaltmac(proxtm,inpermone,inpermtwo,kblock,nopt)
%
% PROXTM is the input two-mode proximity matrix ($nrow \times ncol$
% with a dissimilarity interpretation);
% FIND is the least-squares optimal matrix (with variance-accounted-
% for of VAF) to PROXTM and is the sum of the two matrices
% TARGONE and TARGTWO based on the two row and column
% object orderings given by the ending permutations OUTPERMONE
% and OUTPERMTWO, and in turn ROWPERMONE and ROWPERMTWO and
% COLPERMONE and COLPERMTWO. KBLOCK defines the block size
% in the use of the iterative quadratic assignment routine and
% ADDCONONE and ADDCONTWO are
% the two additive constants for the two model components;
% The $n$ coordinates
% are in COORDONE and COORDTWO.  The input permutations are INPERMONE
% and INPERMTWO.  The $n \times 2$ matrix AXES gives the
% plotting coordinates for the
% combined row and column object set.
% NOPT controls the confirmatory or
% exploratory fitting of the unidimensional
% scales; a value of NOPT = 0 will
% fit in a confirmatory manner the two scales
% indicated by INPERMONE and INPERMTWO;
% a value of NOPT = 1 uses iterative QA
% to locate the better permutations to fit.
```

cat_vs_con_orderfit.m

```
% CAT_VS_CON_ORDERFIT uses a constraining order to fit a best
% ultrametric, anti-Robinson form, and linear unidimensional scale; all
% three of these representations conform to this order.
%
% syntax: [findultra,vafultra,vafarob,arobprox,fitlinear,vaflinear,...
%    coord,addcon] = cat_vs_con_orderfit(prox,inperm,conperm)
%
% PROX is the input dissimilarity matrix and INPERM is
% a starting permutation for how the ultrametric constraints are searched.
% The permutation CONPERM is a given constraining order.
% As output, FINDULTRA is the best ultrametric found with VAF of
% VAFULTRA; AROBPROX is the best AR form identified with VAF of
% VAFAROB; FITLINEAR is the best LUS model with VAF of VAFLINEAR with
% COORD constraining the coordinates and ADDCON the additive constant.
```

cat_vs_con_orderfnd.m

```
% CAT_VS_CON_ORDERFND finds a constraining order to fit a best
% ultrametric, anti-Robinson form, and linear unidimensional scale; all
% three of these representations conform to this order.
%
% syntax: [findultra,vafultra,conperm,vafarob,arobprox,fitlinear,vaflinear,...
%    coord,addcon] = cat_vs_con_orderfnd(prox,inperm)
%
% PROX is the input dissimilarity matrix and INPERM is
% a starting permutation for how the ultrametric constraints are searched.
% The permutation CONPERM is a given order found and used to
% constrain the various representations. FINDULTRA is the best ultrametric
% found with VAF of VAFULTRA; AROBPROX is the best AR form identified
% with VAF of VAFAROB; FITLINEAR is the best LUS model with VAF of VAFLINEAR
% with COORD constraining the coordinates and ADDCON the additive constant.
```

cent_linearfit

```
% CENT_LINEARFIT fits a structure to a proximity matrix by first fitting
% a centroid metric and secondly a linear unidimensional scale
% to the residual matrix where the latter is constrained by a given object order.
%
% syntax: [find,vaf,outperm,targone,targtwo,lengthsone,coordtwo,addcontwo] = ...
%      cent_linearfit(prox,inperm)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation); INPERM is the given
% input constraining order (permutation) which is also given
% as the output vector OUTPERM;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROX. TARGTWO is the linear unidimensional scaling
% component of the decomposition defined by the coordinates in COORDTWO
% with additive constant ADDCONTWO;
% TARGONE is the centroid metric component defined by the
% lengths in LENGTHSONE.
```

cent_linearfnd

```
% CENT_LINEARFND finds fits a structure to a proximity matrix by first fitting
% a centroid metric and secondly a linear unidimensional scale
% to the residual matrix.
%
% syntax: [find,vaf,outperm,targone,targtwo,lengthsone,coordtwo,addcontwo] = ...
%      cent_linearfnd(prox,inperm)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation); INPERM is the given
% input beginnining order (permutation); the found output vector is OUTPERM;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROX. TARGTWO is the linear unidimensional scaling
% component of the decomposition defined by the coordinates in COORDTWO
% with addtive constant ADDCONTWO;
% TARGONE is the centroid metric component defined by the
% lengths in LENGTHSONE.
```

centfit.m

```
% CENTFIT finds the least-squares fitted centroid metric (FIT) to
% PROX, the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation).
```

```
%
% syntax: [fit,vaf,lengths] = centfit(prox)
%
% The $n$ values that serve to define the approximating sums,
% $g_{i} + g_{j}$, are given in the vector LENGTHS of size $n \times 1$.
```

centfittm.m

```
% CENTFITTM finds the least-squares fitted two-mode centroid metric
% (FIT) to PROXTM, the two-mode rectangular input proximity matrix
% (with a dissimilarity interpretation).
%
% syntax: [fit,vaf,lengths] = centfittm(proxtm)
%
% The $n$ values (where $n$ = number of rows + number of columns)
% serve to define the approximating sums,
% $u_{i} + v_{j}$, where the $u_{i}$ are for the rows and the $v_{j}$
% are for the columns; these are given in  the vector LENGTHS of size
% $n \times 1$, with row values first followed by the column values.
```

circularplot.m

```
% CIRCULARPLOT plots the object set using the coordinates
% around a circular structure derived from the $n \times n$
% interpoint distance matrix around a circle given by CIRC.
% The positions are labeled by the order of objects
% given in INPERM.
%
% syntax: [circum,radius,coord,degrees,cumdegrees] = ...
%   circularplot(circ,inperm)
%
% The output consists of a plot, the circumference of the
% circle (CIRCUM) and radius (RADIUS); the coordinates of
% the plot positions (COORD), and the degrees and cumulative
% degrees induced between the plot positions
% (in DEGREES and CUMDEGREES).
% The positions around the circle are numbered from 1
% (at the "noon" position) to $n$, moving
% clockwise around the circular structure.
```

cirfit.m

```
%  CIRFIT does a confirmatory fitting of a given order
% (assumed to reflect a circular ordering around a closed
% unidimensional structure) using Dykstra's
% (Kaczmarz's) iterative projection least-squares method.
%
% syntax: [fit, diff] = cirfit(prox,inperm)
%
% INPERM is the given order; FIT is an $n \times n$ matrix that
% is fitted to PROX(INPERM,INPERM) with least-squares value DIFF.
```

cirfitac.m

```
% CIRFITAC does a confirmatory fitting (including
% the estimation of an additive constant) for a given order
% (assumed to reflect a circular ordering around a closed
% unidimensional structure) using the Dykstra--Kaczmarz
% iterative projection least-squares method.
%
% syntax: [fit, vaf, addcon] = cirfitac(prox,inperm)
%
% INPERM is the given order; FIT is an $n \times n$ matrix that
% is fitted to PROX(INPERM,INPERM) with variance-accounted-for of
% VAF; ADDCON is the estimated additive constant.
```

class_scaledp.m

```
% CLASS_SCALEDP carries out a unidimensional seriation or
% scaling of a set of object classes defined for a symmetric proximity
% matrix using dynamic programming.
%
% syntax: [permut,cumobfun] = class_scaledp(prox,numbclass,membclass)
%
% PROX is the ($n \times n$) input proximity matrix (with a zero
% main diagonal and a dissimilarity interpretation);
% NUMBCLASS (= $n_{c}$) is the number of object classes to be
% sequenced;
% MEMBCLASS is an $n \times 1$ vector containing the input class
% membership and includes all the integers from 1 to NUMBCLASS and
% zeros when objects are to be deleted from consideration;
% PERMUT is the order of the classes in the optimal permutation (say,
% $\rho^{*}$);
% CUMOBFUN gives the cumulative values of the objective function for
% the successive placements of the objects in the optimal permutation:
% $\sum_{i=1}^{k} (t_{i}^{(\rho^{*})})^{2}$ for $k = 1, \ldots, n_{c}$.
%
```

```
% Initializations:  The vectors VALSTORE and IDXSTORE store the
% results of the recursion for the $(2^n_{c})-1$ nonempty subsets of the
% set of classes. The integer positions in these vectors correspond to
% subsets whose binary number equivalents are equal to those integer positions.
```

eqspace_cirfitac.m

```
% EQSPACE_CIRFITAC does a confirmatory fitting (including
% the estimation of an additive constant) for a given order
% (assumed to reflect a circular ordering around a closed
% unidimensional structure) using the Dykstra--Kaczmarz
% iterative projection least-squares method. Also, an equally-spaced
% confirmatory fitting alternative is carried out.
%
% syntax: [fit, vaf, addcon, eqfit, eqvaf, eqaddcon] = ...
%             eqspace_cirfitac(prox,inperm)
%
% INPERM is the given order; FIT is an $n \times n$ matrix that
% is fitted to PROX(INPERM,INPERM) with variance-accounted-for of
% VAF; ADDCON is the estimated additive constant. The equally-spaced
% output alternatives are prefixed with an EQ.
```

eqspace_linfitac.m

```
% EQSPACE_LINFITAC does a confirmatory fitting of a given unidimensional order
% using the Dykstra--Kaczmarz iterative projection
% least-squares method, but differing from linfit.m in
% including the estimation of an additive constant.  Also
% an equally-spaced confirmatory fitting alternative is carried out.
%
% syntax: [fit, vaf, coord, addcon, eqfit, eqvaf, eqaddcon] = ...
%             eqspace_linfitac(prox,inperm)
%
% INPERM is the given order;
% FIT is an $n \times n$ matrix that is fitted to
% PROX(INPERM,INPERM) with variance-accounted-for VAF;
% COORD gives the ordered coordinates whose absolute differences
% could be used to reconstruct FIT; ADDCON is the estimated
% additive constant that can be interpreted as being added to PROX.
% The equally-spaced output alternatives are prefixed with an EQ.
```

eqspace_ultrafit.m

```
% EQSPACE_ULTRAFIT fits a given ultrametric using iterative projection to
% a symmetric proximity matrix in the $L_{2}$-norm.  Also, an
% equally-spaced confirmatory fitting alternative is carried out using the
% entries in TARG (assumed to be integer-valued reflecting the level at
% which the clusters are formed).
%
% syntax: [fit,vaf,eqfit,eqvaf,eqaddcon] = eqspace_ultrafit(prox,targ)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% TARG is an ultrametric matrix of the same size as PROX;
% FIT is the least-squares optimal matrix (with
% variance-accounted-for of VAF) to PROX satisfying the ultrametric
% constraints implicit in TARG.  The equally-spaced output alternatives are
% prefixed with an EQ.
```

linearplot.m

```
% LINEARPLOT plots the object set using the ordered coordinates in COORD
% and labels the positions by the order of the objects given in INPERM.
%
% syntax: [linearlength] = linearplot(coord,inperm)
%
% The output value LINEARLENGTH is the sum of the interpoint distances from
% COORD.
```

linfit.m

```
%  LINFIT does a confirmatory fitting of a given
%  unidimensional order using Dykstra's
%  (Kaczmarz's) iterative projection least-squares method.
%
%  syntax: [fit, diff, coord] = linfit(prox,inperm)
%
%  INPERM is the given order;
%  FIT is an $n \times n$ matrix that is fitted to
%  PROX(INPERM,INPERM) with least-squares value DIFF;
%  COORD gives the ordered coordinates whose absolute
%  differences could be used to reconstruct FIT.
```

linfit_tied.m

```
% LINFIT_TIED does a confirmatory fitting of a given
% unidimensional order using Dykstra's
% (Kaczmarz's) iterative projection least-squares method. This
% includes the possible imposition of tied coordinates.
%
% syntax: [fit, diff, coord] = linfit_tied(prox,inperm,tiedcoord)
%
% INPERM is the given order;
% FIT is an $n \times n$ matrix that is fitted to
% PROX(INPERM,INPERM) with least-squares value DIFF;
% COORD gives the ordered coordinates whose absolute
% differences could be used to reconstruct FIT; TIEDCOORD
% is the tied pattern of coordinates imposed (in order)
% along the continuum (using the integers from 1 up to n
% to indicate the tied positions).
```

linfitac.m

```
% LINFITAC does a confirmatory fitting of a given unidimensional order
% using the Dykstra--Kaczmarz iterative projection
% least-squares method, but differing from linfit.m in
% including the estimation of an additive constant.
%
% syntax: [fit, vaf, coord, addcon] = linfitac(prox,inperm)
%
% INPERM is the given order;
% FIT is an $n \times n$ matrix that is fitted to
% PROX(INPERM,INPERM) with variance-accounted-for VAF;
% COORD gives the ordered coordinates whose absolute differences
% could be used to reconstruct FIT; ADDCON is the estimated
% additive constant that can be interpreted as being added to PROX.
```

linfitac_missing.m

```
% LINFITAC_MISSING does a confirmatory fitting of a given unidimensional order
% using the Dykstra--Kaczmarz iterative projection
% least-squares method, but differing from linfit.m in
% including the estimation of an additive constant;also, missing entries
% in the input proximity matrix PROX are given values of zero.
%
% syntax: [fit, vaf, addcon] = ...
%               linfitac_missing(prox,inperm,proxmiss)
```

```
%
% INPERM is the given order;
% FIT is an $n \times n$ matrix that is fitted to
% PROX(INPERM,INPERM) with variance-accounted-for VAF;
% ADDCON is the estimated additive constant that can be interpreted
% as being added to PROX.  PROXMISS is the same size as PROX (with main
% diagonal entries all zero); an off-diagonal entry of 1.0 denotes an
% entry in PROX that is present and 0.0 if it is absent.
```

linfitac_tied.m

```
% LINFITAC_TIED does a confirmatory fitting of a given unidimensional order
% using the Dykstra--Kaczmarz iterative projection
% least-squares method, but differing from linfit_tied.m in
% including the estimation of an additive constant.  This also allows
% the possible imposition of tied coordinates.
%
% syntax: [fit, vaf, coord, addcon] = linfitac_tied(prox,inperm,tiedcoord)
%
% INPERM is the given order;
% FIT is an $n \times n$ matrix that is fitted to
% PROX(INPERM,INPERM) with variance-accounted-for VAF;
% COORD gives the ordered coordinates whose absolute differences
% could be used to reconstruct FIT; ADDCON is the estimated
% additive constant that can be interpreted as being added to PROX.
% TIEDCOORD is the tied pattern of coordinates imposed (in order)
% along the continuum (using the integers from 1 up to n
% to indicate the tied positions).
```

linfittm.m

```
% LINFITTM does a confirmatory two-mode fitting of a given
% unidimensional ordering of the row and column objects of
% a two-mode proximity matrix PROXTM using Dykstra's (Kaczmarz's)
% iterative projection least-squares method.
%
% syntax: [fit,diff,rowperm,colperm,coord] = linfittm(proxtm,inperm)
%
% INPERM is the given ordering of the row and column objects
% together; FIT is an nrow (number of rows) by ncol (number
% of columns) matrix of absolute coordinate differences that
% is fitted to PROXTM(ROWPERM,COLPERM) with DIFF being the
% (least-squares criterion) sum of squared discrepancies
% between FIT and PROXTM(ROWPERM,COLMEAN);
%  ROWPERM and COLPERM are the row and column object orderings
%  derived from INPERM.  The nrow + ncol coordinates
%  (ordered with the smallest
%  set at a value of zero) are given in COORD.
```

linfittmac.m

```
%  LINFITTMAC does a confirmatory two-mode fitting of a given
%  unidimensional ordering of the row and column objects of
%  a two-mode proximity matrix PROXTM using Dykstra's (Kaczmarz's)
%  iterative projection least-squares method;
%  it differs from linfittm.m  by including the estimation of an
%  additive constant.
%
%  syntax: [fit,vaf,rowperm,colperm,addcon,coord] = ...
%    linfittmac(proxtm,inperm)
%
%  INPERM is the given ordering of the row and column objects
%  together; FIT is an nrow (number of rows) by ncol (number
%  of columns) matrix  of absolute coordinate differences that
%  is fitted to PROXTM(ROWPERM,COLPERM) with VAF being the
%  variance-accounted-for.  ROWPERM and COLPERM are the row and
%  column object orderings derived from INPERM.  ADDCON is the
%  estimated additive constant that can be interpreted as being
%  added to PROXTM (or, alternatively, subtracted
%  from the fitted matrix FIT).  The nrow + ncol coordinates
%  (ordered with the smallest
%  set at a value of zero) are given in COORD.
```

order.m

```
% ORDER carries out an iterative Quadratic Assignment maximization
% task using a given square ($n x n$) proximity matrix PROX (with
% a zero main diagonal and a dissimilarity interpretation).
%
% syntax: [outperm,rawindex,allperms,index] = ...
%   order(prox,targ,inperm,kblock)
%
% Three separate local operations are used to permute
% the rows and columns of the proximity matrix to maximize the
% cross-product index with respect to a given square target matrix
% TARG: pairwise interchanges of objects in the permutation defining
% the row and column order of the square proximity matrix;
% the insertion of from 1 to KBLOCK
% (which is less than or equal to $n-1$) consecutive objects in
% the permutation defining the row and column order of the data
% matrix; the rotation of from 2 to KBLOCK
% (which is less than or equal to $n-1$) consecutive objects in
% the permutation defining the row and column order of the data
% matrix. INPERM is the input beginning permutation (a permutation
% of the first $n$ integers).
% OUTPERM is the final permutation of PROX with the
% cross-product index RAWINDEX
% with respect to TARG. ALLPERMS is a cell array containing INDEX
% entries corresponding to all the
% permutations identified in the optimization from ALLPERMS{1} =
% INPERM to ALLPERMS{INDEX} = OUTPERM.
```

order_missing.m

```
% ORDER_MISSING carries out an iterative Quadratic Assignment maximization
% task using a given square ($n x n$) proximity matrix PROX (with
% a zero main diagonal and a dissimilarity interpretation; missing entries
% PROX are given values of zero).
%
% syntax: [outperm,rawindex,allperms,index] = ...
%   order_missing(prox,targ,inperm,kblock,proxmiss)
%
% Three separate local operations are used to permute
% the rows and columns of the proximity matrix to maximize the
% cross-product index with respect to a given square target matrix
% TARG: pairwise interchanges of objects in the permutation defining
% the row and column order of the square proximity matrix;
% the insertion of from 1 to KBLOCK
% (which is less than or equal to $n-1$) consecutive objects in
% the permutation defining the row and column order of the data
% matrix; the rotation of from 2 to KBLOCK
% (which is less than or equal to $n-1$) consecutive objects in
% the permutation defining the row and column order of the data
% matrix. INPERM is the input beginning permutation (a permutation
% of the first $n$ integers).  PROXMISS is the same size as PROX (with
% main diagonal entries all zero); an off-diagonal entry of 1.0 denotes an
% entry in PROX that is present and 0.0 if it is absent.
% OUTPERM is the final permutation of PROX with the
% cross-product index RAWINDEX
% with respect to TARG. ALLPERMS is a cell array containing INDEX
% entries corresponding to all the
% permutations identified in the optimization from ALLPERMS{1} =
% INPERM to ALLPERMS{INDEX} = OUTPERM.
```

orderpartitionfit.m

```
% ORDERPARTITIONFIT provides a least-squares approximation to a proximity
% matrix based on a given collection of partitions with ordered classes.
%
% syntax: [fit,weights,vaf] = orderpartitionfit(prox,lincon,membership)
%
% PROX is the n x n input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation); LINCON is the given constraining
% linear order (a permutation of the integers from 1 to n).
% MEMBERSHIP is the m x n matrix indicating cluster membership, where
% each row corresponds to a specific ordered partition (there are
% m partitions in general);
% the columns are in the identity permutation input order used for PROX.
% FIT is an n x n matrix fitted to PROX (through least-squares) constructed
% from the nonnegative weights given in the m x 1 WEIGHTS vectors
% corresponding to each of the ordered partitions.  VAF is the variance-
% accounted-for in the proximity matrix PROX by the fitted matrix FIT.
```

orderpartitionfnd.m

```
% ORDERPARTITIONFND uses dynamic programming to
% construct a linearly constrained cluster analysis that
% consists of a collection of partitions with from 1 to
% n ordered classes.
%
% syntax: [membership,objectives,permmember,clusmeasure,...
%    cluscoord,residsumsq] = orderpartitionfnd(prox,lincon)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation); LINCON is the given
% constraining linear order (a permutation of the integers from
% 1 to n).
% MEMBERSHIP is the n x n matrix indicating cluster membership,
% where rows correspond to the number of ordered clusters,
% and the columns are in the identity permutation input order
% used for PROX. PERMMEMBER uses LINCON to reorder the columns
% of MEMBERSHIP.
% OBJECTIVES is the vector of merit values maximized in the
% construction of the ordered partitions; RESIDSUMSQ is the
% vector of residual sum of squares obtained for the ordered
% partition construction.  CLUSMEASURE is the n x n matrix
% (upper-triangular) containing the cluster measures for contiguous
% object sets; the appropriate values in CLUSMEASURE are added
% to obtain the values optimized in OBJECTIVES; CLUSCOORD is also
% an n x n (upper-triangular) matrix but now containing the coordinates
% that would be would be used for all the (ordered)
% objects within a class.
```

ordertm.m

```
% ORDERTM carries out an iterative
% quadratic assignment maximization task using the
% two-mode proximity matrix PROXTM
% (with entries deviated from the mean proximity)
% in the upper-right- and lower-left-hand portions of
% a defined square ($n x n$) proximity matrix
% (called SQUAREPROX with a dissimilarity interpretation)
% with zeros placed elsewhere ($n$ = number of rows +
% number of columns of PROXTM = nrow + ncol).
%
% syntax: [outperm, rawindex, allperms, index, squareprox] = ...
%   ordertm(proxtm, targ, inperm, kblock)
%
% Three separate local operations are used to permute
% the rows and columns of the square
% proximity matrix to maximize the cross-product
% index with respect to a square target matrix TARG:
% pairwise interchanges of objects in the
% permutation defining the row and column
% order of the square proximity matrix; the insertion of from 1 to
% KBLOCK (which is less than or equal to $n-1$) consecutive objects
% in the permutation defining the row and column order of the
% data matrix; the rotation of from 2 to KBLOCK (which is less than
% or equal to $n-1$) consecutive objects in
% the permutation defining the row and column order of the data
% matrix. INPERM is the input beginning permutation (a permutation
% of the first $n$ integers).
% PROXTM is the two-mode $nrow x ncol$ input proximity matrix.
% TARG is the $n x n$ input target matrix.
% OUTPERM is the final permutation of SQUAREPROX with the
% cross-product index RAWINDEX
% with respect to TARG. ALLPERMS is a cell array containing INDEX
% entries corresponding to all the
% permutations identified in the optimization from ALLPERMS{1}
% = INPERM to ALLPERMS{INDEX} = OUTPERM.
```

partitionfit.m

```
% PARTITIONFIT provides a least-squares approximation to a proximity
% matrix based on a given collection of partitions.
%
% syntax: [fitted,vaf,weights,end_condition] = partitionfit(prox,member)
%
% PROX is the n x n input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation); MEMBER is the m x n matrix
% indicating cluster membership, where each row corresponds to a specific
% partition (there are m partitions in general); the columns of MEMBER
% are in the same input order used for PROX.
% FITTED is an n x n matrix fitted to PROX (through least-squares)
% constructed from the nonnegative weights given in the m x 1 WEIGHTS
% vector corresponding to each of the partitions.  VAF is the variance-
% accounted-for in the proximity matrix PROX by the fitted matrix FITTED.
% END_CONDITION should be zero for a normal termination of the optimization
% process.
```

proxmon.m

```
%  PROXMON produces a monotonically transformed proximity matrix
%  (MONPROXPERMUT) from the order constraints obtained from each
%  pair of entries in the input proximity matrix PROXPERMUT
%  (symmetric with a zero main diagonal and a dissimilarity
%  interpretation).
%
%  syntax: [monproxpermut, vaf, diff] = proxmon(proxpermut, fitted)
%
%  MONPROXPERMUT is close to the
%  $n \times n$ matrix FITTED in the least-squares sense;
%  the variance accounted for (VAF) is how
%  much variance in MONPROXPERMUT can be accounted for by
%  FITTED; DIFF is the value of the least-squares criterion.
```

proxmontm.m

```
%  PROXMONTM produces a monotonically transformed
%  two-mode proximity matrix (MONPROXPERMUTTM)
%  from the order constraints obtained
%  from each pair of entries in the input two-mode
%  proximity matrix PROXPERMUTTM (with a dissimilarity
%  interpretation).
%
```

```
% syntax: [monproxpermuttm, vaf, diff] = ...
%      proxmontm(proxpermuttm, fittedtm)
%
% MONPROXPERMUTTM is close to the $nrow \times ncol$
% matrix FITTEDTM in the least-squares sense;
% The variance accounted for (VAF) is how much variance
% in MONPROXPERMUTTM can be accounted for by FITTEDTM;
% DIFF is the value of the least-squares criterion.
```

proxstd.m

```
% PROXSTD produces a standardized proximity matrix (STANPROX)
% from the input $n \times n$ proximity matrix
% (PROX) with zero main diagonal and a dissimilarity
% interpretation.
%
% syntax: [stanprox, stanproxmult] = proxstd(prox,mean)
%
% STANPROX entries have unit variance (standard deviation of one)
% with a mean of MEAN given as an input number;
% STANPROXMULT (upper-triangular) entries have a sum of
% squares equal to $n(n-1)/2$.
```

targcir.m

```
% TARGCIR produces a symmetric proximity matrix of size
% $n \times n$, containing distances
% between equally and unit-spaced positions
% around a circle: targcircular(i,j) = min(abs(i-j),n-abs(i-j)).
%
% syntax: [targcircular] = targcir(n)
```

targlin.m

```
% TARGLIN produces a symmetric proximity matrix of size
% $n \times n$, containing distances
% between equally and unit-spaced positions
% along a line: targlinear(i,j) = abs(i-j).
%
% syntax: [targlinear] = targlin(n)
```

ultrafit.m

```
% ULTRAFIT fits a given ultrametric using iterative projection to
% a symmetric proximity matrix in the $L_{2}$-norm.
%
% syntax: [fit,vaf] = ultrafit(prox,targ)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% TARG is an ultrametric matrix of the same size as PROX;
% FIT is the least-squares optimal matrix (with
% variance-accounted-for of VAF) to PROX satisfying the ultrametric
% constraints implicit in TARG.
```

unicirac.m

```
% UNICIRAC finds and fits a circular
% unidimensional scale using iterative projection to
% a symmetric proximity matrix in the $L_{2}$-norm based on a
% permutation identified through the use of iterative
% quadratic assignment.
%
% syntax: [find, vaf, outperm, addcon] = unicirac(prox, inperm, kblock)
%
% PROX is the input proximity matrix ($n \times n$ with a
% zero main diagonal and a dissimilarity interpretation);
% INPERM is a given starting permutation (assumed to be around the
% circle) of the first $n$ integers;
% FIND is the least-squares optimal matrix (with
% variance-accounted-for of VAF) to PROX having a circular
% anti-Robinson form for the row and column
% object ordering given by the ending permutation OUTPERM.
% The spacings among the objects are given by the diagonal entries
% in FIND (and the extreme (1,n) entry in FIND). KBLOCK
% defines the block size in the use of the iterative quadratic
% assignment routine. The additive constant for the model is
% given by ADDCON.
```

uniscalqa.m

```
%  UNISCALQA carries out a unidimensional scaling of a symmetric
%  proximity matrix using iterative quadratic assignment.
%
%  syntax: [outperm, rawindex, allperms, index, coord, diff] = ...
%   uniscalqa(prox, targ, inperm, kblock)
%
%  PROX is the input proximity matrix (with a zero main diagonal
%  and a dissimilarity interpretation);
%  TARG is the input target matrix (usually with a zero main
%  diagonal and a dissimilarity interpretation representing
%  equally spaced locations along a continuum);
%  INPERM is the input beginning permutation (a permutation of the
%  first $n$ integers). OUTPERM is the final permutation of PROX
%  with the cross-product index RAWINDEX
%  with respect to TARG redefined as
%  $ = \{abs(coord(i) - coord(j))\}$;
%  ALLPERMS is a cell array containing INDEX entries corresponding
%  to all the permutations identified in the optimization from
%  ALLPERMS{1} = INPERM to ALLPERMS{INDEX} = OUTPERM.
%  The insertion and rotation routines use from 1 to KBLOCK
%  (which is less than or equal to $n-1$) consecutive objects in
%  the permutation defining the row and column order of the data
%  matrix.  COORD is the set of coordinates of the unidimensional
%  scaling in ascending order;
%  DIFF is the value of the least-squares loss function for the
%  coordinates and object permutation.
```