A Combinatorial Data Analysis Toolbox for MATLAB

The HAM Team

October 7, 2002

Contents

Part I

The Representation of Proximity Matrices by Tree Structures — the Tree Structure Toolbox (TST)

Various methods of data representation based on graph-theoretic structures have been developed over the last several decades for explaining the pattern of information potentially present in a single (or possibly, in a collection of) numerically given proximity matri(ces), each defined between pairs of objects from a single set, or in some cases, between the objects from several distinct sets (for example, see Carroll, 1976; Carroll, Clark, & DeSarbo, 1984; Carroll & Pruzansky, 1980; De Soete, 1983, 1984a,b,c; De Soete, Carroll, & DeSarbo, 1987; De Soete, DeSarbo, Furnas, & Carroll, 1984; Hutchinson, 1989; Klauer & Carroll, 1989, 1991; Hubert & Arabie, 1995). Typically, a specific class of graph-theoretic structures is assumed capable of representing the proximity information, and the proposed method seeks a member from the class producing a reconstructed set of proximities that are as close as possible to the original. The most prominent graph-theoretic structures used are those usually referred to as ultrametrics and additive trees, and these will be the primary emphasis in this particular Toolbox as well (we do note, however, that a number of other possibilities have also been considered in the literature, e.g., more general network models as in Klauer and Carroll, 1989, 1991).

Although a variety of strategies have been proposed for locating good exemplars from whatever class of graph-theoretic structures is being considered, one approach has been to adopt a least squares criterion in which the class exemplar is identified by attempting to minimize the sum of squared discrepancies between the original proximities and their reconstructions obtained through the use of the particular structure selected by the data analyst. One common implementation of the least-squares optimization strategy has been defined by the usual least-squares criterion but augmented by some collection of penalty functions that seek to impose whatever constraints are mandated by the structural representation being sought. Then, through the use of some unconstrained optimization scheme (e.g., steepest descent, conjugate gradients), an attempt is made to find both (a) the particular constraints that should be imposed to define the specific structure from the class, and (b) the reconstructed proximities based on the structure finally identified. The resulting optimization strategy is heuristic in the sense that there is no guarantee of global optimality for the final structural representation identified even within the chosen graph-theoretic class, because the particular constraints defining the selected structure were located by a possibly reasonable but not verifiably optimal search strategy that was (implicitly) implemented in the course of the process of optimization. A second implementation of the least-squares optimization approach and the one that we will concentrate on exclusively in this particular Toolbox is based on the type of iterative projection strategy already illustrated in conjunction with linear unidimensional scaling (LUS) (see Section 1.1.5 in the LUS Toolbox on solving linear inequality constrained least-squares tasks), and developed in detail for the graph-theoretic context by Hubert and Arabie (1995). In its non-heuristic form, iterative projection allows the reconstruction of a set of proximities based on a fixed collection of constraints implied by whatever specific graph-theoretic structure has been selected for their representation. As in LUS, successive (or iterative) projections onto closed convex sets are carried out that are defined by the collection of given constraints implied by the structural representation chosen. Thus, the need for penalty terms is avoided and there is no explicit use of gradients in the attendant optimization strategy; also, it is fairly straightforward to incorporate a variety of different types of constraints that may be auxiliary to those generated from the given structural representation but none-the-less of interest to impose on the reconstruction.

As a least squares optimization strategy (in a non-heuristic form), iterative projection assumes that whatever constraint set is to be applied is completely known prior to its application. However, just as the various penalty-function and gradient-optimization techniques have been turned into heuristic search strategies for the particular structures of interest by allowing the collection of constraints to vary over the course of the optimization process, we attempt the same in this Toolbox in using iterative projection to find the better-fitting ultrametrics and additive trees for a given proximity matrix. Thus, in addition to carrying out a least squares task subject to given structural constraints, iterative projection will be considered as one possible heuristic search strategy (and an alternative to those heuristic methods that have been suggested in the literature and based exclusively on the use of some type of penalty function) for locating the actual constraints to impose in the first instance, and therefore, to identify the general form of the structural representation sought.

The various least squares optimization tasks entailing both the identification of the specific form of the structural representation to adopt and the subsequent least squares fitting itself generally fall into the class of NP-hard problems (e.g., for ultrametric and additive trees, see Day, 1987; Křivánek, 1986; Křivánek & Moravek, 1986); thus, the best we can hope for is a heuristic extension of the iterative projection strategy leading to good but not necessarily optimal final structural representations within the general class of representations desired. As is standard with a reliance on such heuristic optimization methods, the use of multiple starting points will hopefully determine a set of local optima characterizing the better solutions attainable for a given data set. The presence of local optima in the use of any heuristic and combinatorially based optimization strategy is unavoidable, given the NP-hardness of the basic optimization tasks of interest and the general inability of (partial) enumeration methods (when available) to be computationally feasible for use on even moderate-sized data sets. The number of and variation in the local optima observable for any specific situation will obviously depend on the given data, the structural representation sought, and the heuristic search strategy used. But whenever present, local optima may actually be diagnostic for the structure(s) potentially appropriate for characterizing a particular data set. Thus, their identification may even be valuable in explaining the patterning of the data and/or in noting the difficulties with adopting a specific representational form to help discern underlying structure.

Chapter 1

Ultrametrics for Symmetric Proximity Data

The task of hierarchical clustering can be characterized as a specific data analysis problem: given a set of n objects, $S = \{O_1, \ldots, O_n\}$, and an $n \times n$ symmetric proximity matrix $\mathbf{P} = \{p_{ij}\}$ (nonnegative and with a dissimilarity interpretation), find a sequence of partitions of S, denoted as $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$ satisfying the following:

(a) $\mathcal{P}_1$ is the (trivial) partition where all n objects from S are placed into n separate classes;

(b) $\mathcal{P}_n$ is the (also trivial) partition where a single subset contains all n objects;

(c) $\mathcal{P}_k$ is obtained from $\mathcal{P}_{k-1}$ by uniting some pair of classes present in $\mathcal{P}_{k-1}$;

(d) the minimum levels at which object pairs first appear together within the same class should reflect the proximities in $\mathcal{P}$. Or more formally, if we define $\mathbf{U}^0 = \{u^0_{ij}\} = \min\{k-1 \mid \text{objects } O_i \text{ and } O_j \text{ appear within the same class in the partition } \mathcal{P}_k\}$, then if the partition hierarchy is representing the given proximities well, the entries in $\mathbf{U}^0$ and $\mathbf{P}^0$ should be, for example, similarly ordered. We discuss the properties of matrices such as $\mathbf{U}^0$ in more detail below.

To give an example, we preformed a complete-link hierarchical clustering (using SYSTAT) on the `number.dat` proximity matrix used extensively in the LUS Toolbox, and obtained the following partitions of the object indices from 1 to 10 (remembering that these correspond to the digits 0 to 9):

$\mathcal{P}_1$: {{1},{2},{3},{4},{5},{6},{7},{8},{9},{10}}
$\mathcal{P}_2$: {{3,5},{1},{2},{4},{6},{7},{8},{9},{10}}
$\mathcal{P}_3$: {{3,5},{4,10},{1},{2},{6},{7},{8},{9}}
$\mathcal{P}_4$: {{3,5},{4,7,10},{1},{2},{6},{8},{9}}
$\mathcal{P}_5$: {{3,5,9},{4,7,10},{1},{2},{6},{8}}
$\mathcal{P}_6$: {{3,5,9},{4,7,10},{6,8},{1},{2}}
$\mathcal{P}_7$: {{3,5,9},{4,7,10},{6,8},{1,2}}
$\mathcal{P}_8$: {{3,5,9},{4,6,7,8,10},{1,2}}
$\mathcal{P}_9$: {{3,4,5,6,7,8,9,10},{1,2}}

$\mathcal{P}_{10}$: {{1,2,3,4,5,6,7,8,9,10}}

The matrix $\mathbf{U}^0$ was constructed and saved as a 10×10 matrix in the file `numcltarg.dat`, which will be used in an example later:

```
0 6 9 9 9 9 9 9 9 9
6 0 9 9 9 9 9 9 9 9
9 9 0 8 1 8 8 8 4 8
9 9 8 0 8 7 3 7 8 2
9 9 1 8 0 8 8 8 4 8
9 9 8 7 8 0 7 5 8 7
9 9 8 3 8 7 0 7 8 3
9 9 8 7 8 5 7 0 8 7
9 9 4 8 4 8 8 8 0 8
9 9 8 2 8 7 3 7 8 0
```

A concept routinely encountered in discussions of hierarchical clustering is that of an ultrametric, which can be characterized as any nonnegative symmetric dissimilarity matrix for the objects in S, denoted generically by $\mathbf{U} = \{u_{ij}\}$, where $u_{ij} = 0$ if and only if $i = j$, and $u_{ij} \leq \max[u_{ik}, u_{jk}]$ for all $1 \leq i, j, k \leq n$ (this last inequality is equivalent to the statement that for any distinct triple of subscripts, i, j, and k, the largest two proximities among u_{ij}, u_{ik}, and u_{jk} are equal and [therefore] not less than the third). Any ultrametric can be associated with the specific partition hierarchy it induces, having the form $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_T$, where $\mathcal{P}_1$ and $\mathcal{P}_T$ are now the two trivial partitions that respectively contain all objects in separate classes and all objects in the same class, and $\mathcal{P}_k$ is formed from $\mathcal{P}_{k-1}$ ($2 \leq k \leq T$) by (agglomeratively) uniting certain (and possibly more than two) subsets in $\mathcal{P}_{k-1}$. For those subsets merged in $\mathcal{P}_{k-1}$ to form $\mathcal{P}_k$, all between-subset ultrametric values must be equal, and no less than any other ultrametric value associated with an object pair within a class in $\mathcal{P}_{k-1}$. Thus, individual partitions in the hierarchy can be identified by merely increasing a threshold variable starting at zero, and observing that $\mathcal{P}_k$ for $1 \leq k \leq T$ is defined by a set of subsets in which all within-subset ultrametric values are less than or equal to some specific threshold value, and all ultrametric values between subsets are strictly greater. Conversely, any partition hierarchy of the form $\mathcal{P}_1, \ldots, \mathcal{P}_T$ can be identified with the equivalence class of all ultrametric matrices that induce it. We note that if only a *single* pair of subsets can be united in $\mathcal{P}_{k-1}$ to form $\mathcal{P}_k$ for $2 \leq k \leq T$, then $T = n$, and we could then revert to the characterization of a full partition hierarchy $\mathcal{P}_1, \ldots, \mathcal{P}_n$ used earlier.

Given some fixed partition hierarchy $\mathcal{P}_1, \ldots, \mathcal{P}_T$, there are an infinite number of ultrametric matrices that induce it, but all can be generated by (restricted) monotonic functions of what might be called the basic ultrametric matrix $\mathbf{U}^0$ defined earlier. Explicitly, any ultrametric in the equivalence class whose members induce the same fixed hierarchy, $\mathcal{P}_1, \ldots, \mathcal{P}_T$, can be obtained by a strictly increasing monotonic function of the entries in $\mathbf{U}^0$, where the function maps zero to zero. Moreover, because u_{ij}^0 for $i \neq j$ can be only one of the integer values from 1 to $T - 1$, each ultrametric in the equivalence class that generates the fixed hierarchy may be defined by one of $T - 1$ distinct values. When these $T - 1$ values are ordered from the smallest to the largest, the $(k-1)^{st}$ smallest value corresponds to the partition $\mathcal{P}_k$ in the partition hierarchy $\mathcal{P}_1, \ldots, \mathcal{P}_T$, and implicitly to all object pairs that appear together for the first time within a subset in $\mathcal{P}_k$.

To provide an alternative interpretation, the basic ultrametric matrix can also be characterized as defining a collection of linear equality and inequality constraints that any ultrametric in a specific equivalence class must satisfy. Specifically, for each object triple there is (a) a specification of which ultrametric values among the three must be equal plus two additional inequality constraints so that the third is not greater; (b) an inequality or equality constraint for every pair of ultrametric values based on their order relationship in the basic ultrametric matrix; and (c) an equality constraint of zero for the main diagonal entries in $\mathbf{U}$. In any case, given these fixed equality and inequality constraints, standard L_p regression methods (such as those given in Späth, 1991, could be adapted to generate a best-fitting ultrametric, say $\mathbf{U}^* = \{u^*_{ij}\}$, to the given proximity matrix $\mathbf{P} = \{p_{ij}\}$. Concretely, we might find $\mathbf{U}^*$ by minimizing

$$\sum_{i<j}(p_{ij} - u_{ij})^2, \ \sum_{i<j} \mid p_{ij} - u_{ij} \mid, \text{ or possibly, } \max_{i<j} \mid p_{ij} - u_{ij} \mid .$$

(As a convenience here and later, it is assumed that $p_{ij} > 0$ for all $i \neq j$, to avoid the technicality of possibly locating best-fitting 'ultrametrics' that could violate the condition that $u_{ij} = 0$ if and only if $i = j$.)

1.1 Fitting a Given Ultrametric in the L_2 Norm

The MATLAB function, `ultrafit.m`, listed in Appendix A.1 and with usage

```
[fit,vaf] = ultrafit(prox,targ)
```

generates (using iterative projection based on the linear (in)equality constraints obtained from the fixed ultrametric — see Section 1.1.5 in the LUS Toolbox) the best-fitting ultrametric in the L_2-norm (`FIT`) within the same equivalence class as that of a given ultrametric matrix `TARG`. The matrix `PROX` contains the symmetric input proximities and `VAF` is the variance-accounted-for (defined throughout this Toolbox, as usual, by normalizing the obtained L_2-norm loss value:

$$\text{variance} - \text{accounted} - \text{for} = 1 - \frac{\sum_{i<j}(p_{ij} - u^*_{ij})^2}{\sum_{i<j}(p_{ij} - \bar{p})^2},$$

where $\bar{p}$ is the mean off-diagonal proximity in $\mathbf{P}$, and $\mathbf{U}^* = \{u^*_{ij}\}$ is the best-fitting ultrametric.

In the example below, the target matrix is `numcltarg` obtained from the complete-link hierarchical clustering of `number`; the `VAF` generated by these ultrametric constraints is .4781. Comparing the target matrix `numcltarg` and `fit`, the particular monotonic function, say

$f(\cdot)$, of the entries in the basic ultrametric matrix that generates the fitted matrix is: $f(1) = .0590$, $f(2) = .2630$, $f(3) = .2980$, $f(4) = .3065$, $f(5) = .4000$, $f(6) = .4210$, $f(7) = .4808$, $f(8) = .5535$, $f(9) = .6761$.

```
load number.dat
load numcltarg.dat
[fit,vaf] = ultrafit(number,numcltarg)

fit =

  Columns 1 through 6

         0    0.4210    0.6761    0.6761    0.6761    0.6761
    0.4210         0    0.6761    0.6761    0.6761    0.6761
    0.6761    0.6761         0    0.5535    0.0590    0.5535
    0.6761    0.6761    0.5535         0    0.5535    0.4808
    0.6761    0.6761    0.0590    0.5535         0    0.5535
    0.6761    0.6761    0.5535    0.4808    0.5535         0
    0.6761    0.6761    0.5535    0.2980    0.5535    0.4808
    0.6761    0.6761    0.5535    0.4808    0.5535    0.4000
    0.6761    0.6761    0.3065    0.5535    0.3065    0.5535
    0.6761    0.6761    0.5535    0.2630    0.5535    0.4808

  Columns 7 through 10

    0.6761    0.6761    0.6761    0.6761
    0.6761    0.6761    0.6761    0.6761
    0.5535    0.5535    0.3065    0.5535
    0.2980    0.4808    0.5535    0.2630
    0.5535    0.5535    0.3065    0.5535
    0.4808    0.4000    0.5535    0.4808
         0    0.4808    0.5535    0.2980
    0.4808         0    0.5535    0.4808
    0.5535    0.5535         0    0.5535
    0.2980    0.4808    0.5535         0


vaf =

    0.4781
```

1.2 Finding an Ultrametric in the L_2 Norm

The m-file `ultrafnd.m` in Appendix A.2 implements a heuristic search strategy using iterative projection to locate a best-fitting ultrametric in the L_2-norm. The method used is from Hubert and Arabie (1995); this latter source should be consulted for the explicit algorithmic details implemented in `ultrafnd.m` (as well as for many of the other m-files to be presented in this Toolbox). The m-file usage has the form

```
[find,vaf] = ultrafnd(prox,inperm)
```

where `FIND` is the ultrametric identified having variance-accounted-for `VAF`. The matrix `PROX` contains the symmetric input proximities; `INPERM` is a permutation that defines an order in which the constraints are considered over all object triples. In the example below, for instance, `INPERM` is simply set as the MATLAB built-in random permutation function `randperm(n)` (using the size $n = 10$ explicitly for the `number` illustration). Thus, the search can be rerun with the same specification but now using a different random starting sequence. Two such searches are shown below leading to vafs of .4941 and .4781 (the latter is the same as obtained from fitting the best ultrametric in Section 1.1 using `numcltarg` for a fixed set of constraints; the former provides a slightly different and better-fitting ultrametric).

```
[find,vaf] = ultrafnd(number,randperm(10))

find =

  Columns 1 through 6

         0    0.7300    0.7300    0.7300    0.7300    0.7300
    0.7300         0    0.5835    0.5835    0.5835    0.5835
    0.7300    0.5835         0    0.5535    0.0590    0.5535
    0.7300    0.5835    0.5535         0    0.5535    0.4808
    0.7300    0.5835    0.0590    0.5535         0    0.5535
    0.7300    0.5835    0.5535    0.4808    0.5535         0
    0.7300    0.5835    0.5535    0.2980    0.5535    0.4808
    0.7300    0.5835    0.5535    0.4808    0.5535    0.4000
    0.7300    0.5835    0.3065    0.5535    0.3065    0.5535
    0.7300    0.5835    0.5535    0.2630    0.5535    0.4808

  Columns 7 through 10

    0.7300    0.7300    0.7300    0.7300
    0.5835    0.5835    0.5835    0.5835
    0.5535    0.5535    0.3065    0.5535
    0.2980    0.4808    0.5535    0.2630
    0.5535    0.5535    0.3065    0.5535
    0.4808    0.4000    0.5535    0.4808
         0    0.4808    0.5535    0.2980
    0.4808         0    0.5535    0.4808
    0.5535    0.5535         0    0.5535
    0.2980    0.4808    0.5535         0


vaf =

    0.4941

[find,vaf] = ultrafnd(number,randperm(10))

find =

  Columns 1 through 6

         0    0.4210    0.6761    0.6761    0.6761    0.6761
    0.4210         0    0.6761    0.6761    0.6761    0.6761
    0.6761    0.6761         0    0.5535    0.0590    0.5535
    0.6761    0.6761    0.5535         0    0.5535    0.4808
    0.6761    0.6761    0.0590    0.5535         0    0.5535
    0.6761    0.6761    0.5535    0.4808    0.5535         0
    0.6761    0.6761    0.5535    0.2980    0.5535    0.4808
    0.6761    0.6761    0.5535    0.4808    0.5535    0.4000
    0.6761    0.6761    0.3065    0.5535    0.3065    0.5535
    0.6761    0.6761    0.5535    0.2630    0.5535    0.4808

  Columns 7 through 10

    0.6761    0.6761    0.6761    0.6761
    0.6761    0.6761    0.6761    0.6761
    0.5535    0.5535    0.3065    0.5535
    0.2980    0.4808    0.5535    0.2630
    0.5535    0.5535    0.3065    0.5535
    0.4808    0.4000    0.5535    0.4808
         0    0.4808    0.5535    0.2980
    0.4808         0    0.5535    0.4808
    0.5535    0.5535         0    0.5535
    0.2980    0.4808    0.5535         0
```

```
vaf =

    0.4781
```

1.3 Representing an Ultrametric (Graphically)

Once an ultrametric matrix has been identified, there are two common ways in which the information within the matrix might be displayed. The first is to perform a simple reordering of the rows and columns of the given matrix to make apparent the sequence of partitions being induced by the ultrametric. The form desired is typically called anti-Robinson (see, for example, Hubert and Arabie, 1994, for a very complete discussion of using and fitting such matrix orderings). When a matrix is in anti-Robinson form, the entries within each row (and column) are non-decreasing moving away from the main diagonal in either direction. As the example given below will show, any ultrametric matrix can be put into such a form easily (and nonuniquely). The second strategy for representing an ultrametric relies on the graphical form of a tree (or as it is typically called in the classification literature, a dendrogram), and where one can read the values of the ultrametric directly from the displayed structure. We give an example of such a tree below (and provide in the *.tex version of this Toolbox document the LaTeX code [within the `picture` environment] to generate the graphical structure).

To give the illustration of reordering an ultrametric matrix to display its anti-Robinson form, the example found in Section 1.2 with a vaf of .4941 will be used, along with a short m-file, `ultraorder.m` given in Appendix A.10. This function implements a simple mechanism of first generating a unidimensional equally-spaced target matrix from the utility m-file `ransymat.m` within the LUS Toolbox, and then reorders heuristically the given ultrametric matrix against this given target with the quadratic assignment functions `pairwiseqa.m` and `insertqa.m` (the latter uses a block size of 1 for `kblock`). The explicit usage is

```
[orderprox,orderperm] = ultraorder(prox)
```

where `PROX` is a supposedly ultrametric matrix; `ORDERPERM` is a permutation used to display the anti-Robinson form in `ORDERPROX` where `orderprox = prox(orderperm,orderperm)`.

```
load number.dat
[find,vaf] = ultrafnd(number,randperm(10))

find =

  Columns 1 through 6

         0    0.7300    0.7300    0.7300    0.7300    0.7300
```

```
    0.7300         0    0.5835    0.5835    0.5835    0.5835
    0.7300    0.5835         0    0.5535    0.0590    0.5535
    0.7300    0.5835    0.5535         0    0.5535    0.4808
    0.7300    0.5835    0.0590    0.5535         0    0.5535
    0.7300    0.5835    0.5535    0.4808    0.5535         0
    0.7300    0.5835    0.5535    0.2980    0.5535    0.4808
    0.7300    0.5835    0.5535    0.4808    0.5535    0.4000
    0.7300    0.5835    0.3065    0.5535    0.3065    0.5535
    0.7300    0.5835    0.5535    0.2630    0.5535    0.4808

  Columns 7 through 10

    0.7300    0.7300    0.7300    0.7300
    0.5835    0.5835    0.5835    0.5835
    0.5535    0.5535    0.3065    0.5535
    0.2980    0.4808    0.5535    0.2630
    0.5535    0.5535    0.3065    0.5535
    0.4808    0.4000    0.5535    0.4808
         0    0.4808    0.5535    0.2980
    0.4808         0    0.5535    0.4808
    0.5535    0.5535         0    0.5535
    0.2980    0.4808    0.5535         0


vaf =

    0.4941

[orderprox,orderperm] = ultraorder(find)

orderprox =

  Columns 1 through 6

         0    0.7300    0.7300    0.7300    0.7300    0.7300
    0.7300         0    0.3065    0.3065    0.5535    0.5535
    0.7300    0.3065         0    0.0590    0.5535    0.5535
    0.7300    0.3065    0.0590         0    0.5535    0.5535
    0.7300    0.5535    0.5535    0.5535         0    0.2630
    0.7300    0.5535    0.5535    0.5535    0.2630         0
    0.7300    0.5535    0.5535    0.5535    0.2980    0.2980
    0.7300    0.5535    0.5535    0.5535    0.4808    0.4808
    0.7300    0.5535    0.5535    0.5535    0.4808    0.4808
    0.7300    0.5835    0.5835    0.5835    0.5835    0.5835

  Columns 7 through 10

    0.7300    0.7300    0.7300    0.7300
    0.5535    0.5535    0.5535    0.5835
    0.5535    0.5535    0.5535    0.5835
    0.5535    0.5535    0.5535    0.5835
    0.2980    0.4808    0.4808    0.5835
    0.2980    0.4808    0.4808    0.5835
         0    0.4808    0.4808    0.5835
    0.4808         0    0.4000    0.5835
    0.4808    0.4000         0    0.5835
    0.5835    0.5835    0.5835         0


orderperm =

     1     9     3     5    10     4     7     8     6     2
```

The reordered matrix using the row and column order of $0 \prec 8 \prec 2 \prec 4 \prec 9 \prec 3 \prec 6 \prec 7 \prec 5 \prec 1$ is given below; here the blocks of equal-valued entries are highlighted, indicating the partition hierarchy (also given below) induced by the ultrametric.

	0	8	2	4	9	3	6	7	5	1
0	x	.73	.73	.73	.73	.73	.73	.73	.73	.73
8	.73	x	.31	.31	.55	.55	.55	.55	.55	.58
2	.73	.31	x	.06	.55	.55	.55	.55	.55	.58
4	.73	.31	.06	x	.55	.55	.55	.55	.55	.58
9	.73	.55	.55	.55	x	.26	.30	.48	.48	.58
3	.73	.55	.55	.55	.26	x	.30	.48	.48	.58
6	.73	.55	.55	.55	.30	.30	x	.48	.48	.58
7	.73	.55	.55	.55	.48	.48	.48	x	.40	.58
5	.73	.55	.55	.55	.48	.48	.48	.40	x	.58
1	.73	.58	.58	.58	.58	.58	.58	.58	.58	x

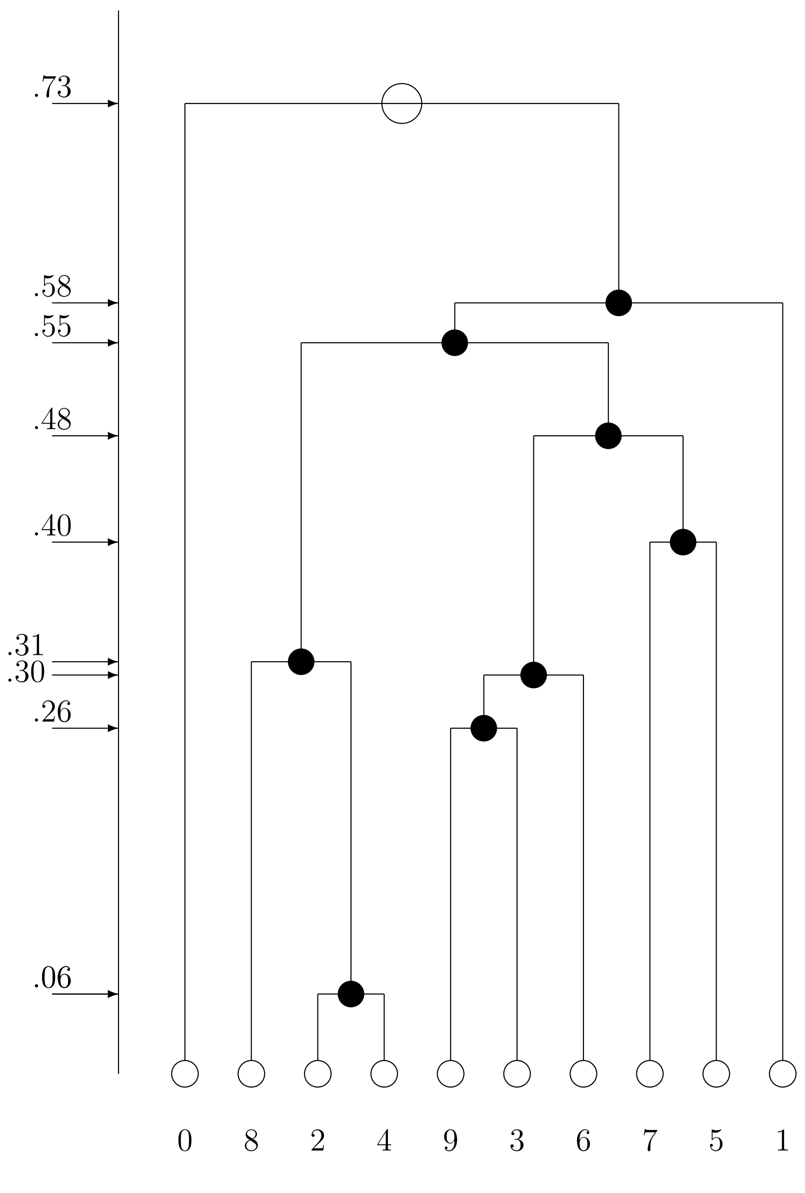
Partition	Level Formed
{{0,8,2,4,9,3,6,7,5,1}}	.73
{{0},{8,2,4,9,3,6,7,5,1}}	.58
{{0},{8,2,4,9,3,6,7,5},{1}}	.55
{{0},{8,2,4},{9,3,6,7,5},{1}}	.48
{{0},{8,2,4},{9,3,6},{7,5},{1}}	.40
{{0},{8,2,4},{9,3,6},{7},{5},{1}}	.31
{{0},{8},{2,4},{9,3,6},{7},{5},{1}}	.30
{{0},{8},{2,4},{9,3},{6},{7},{5},{1}}	.26
{{0},{8},{2,4},{9},{3},{6},{7},{5},{1}}	.06
{{0},{8},{2},{4},{9},{3},{6},{7},{5},{1}}	—

For the partition hierarchy just given, the alternative structure of a dendrogram (or tree) for its representation is given in the figure that follows. The terminal "nodes" of this structure, indicated by open circles, correspond to the ten digits; the filled circles are internal "nodes" reflecting the level at which certain new classes in a partition hierarchy are constructed. For instance, using the calibration given on the long vertical line at the left, a new class consisting of the digits {9,3,6,7,5} is formed at level .48 by uniting the two classes {9,3,6} and {7,5}. Thus, in the ultrametric matrix given earlier, the values between the entries in these two classes are all a constant .48.

The dendrogram just given can be modified (or at least in how it should be interpreted) to motivate the representational form of an additive tree to be introduced in Chapter 2: (a) the values in the calibration along the long vertical axis need to be cut in half; (b) all horizontal lines are now to be understood as having no length and are present only for graphical convenience; (c) a spot on the dendrogram is indicated (here, by a large open circle), called the "root". A crucial characterization feature of an ultrametric is that the root is equidistant from all terminal nodes.

Given these interpretive changes, the ultrametric values for each object pair can now be reconstructed by the length of the path in the tree connecting the two relevant objects. Thus, an ultrametric is reconstructed from the lengths of paths between objects in a tree; and the special form of a tree for an ultrametric is one in which there exists a root that is equidistant from all terminal nodes. In the generalization to an additive tree of Chapter 2, the condition of requiring the existence of an equidistant root is removed, which amounts to allowing the branches attached to the terminal nodes to be stretched or shrunk at will.

Figure 1.1: A Dendrogram (Tree) Representation for the Ultrametric Described in the Text Having Vaf of .4941



Chapter 2

Additive Trees for Symmetric Proximity Data

A currently popular alternative to the use of a simple ultrametric in classification, and which might be considered a natural extension of the notion of an ultrametric, is that of an additive tree; comprehensive discussions can be found in Mirkin (1996, Chapter 7) or throughout Barthélemy and Guénoche (1991). Generalizing the earlier characterization of an ultrametric, an $n \times n$ matrix $\mathbf{A} = \{a_{ij}\}$ can be called an additive tree (metric or matrix) if the three-object (or three-point) ultrametric condition is replaced by a four-object (or four-point) condition: $a_{ij} + a_{kl} \leq \max\{a_{ik} + a_{jl}, a_{il} + a_{jk}\}$ for $1 \leq i, j, k, l \leq n$; equivalently, for any object quadruple O_i, O_j, O_k, and O_l, the largest two values among the sums $a_{ij} + a_{kl}$, $a_{ik} + a_{jl}$, and $a_{il} + a_{jk}$ must be equal.

Any additive tree matrix $\mathbf{A}$ can 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 $c_{ij} = g_i + g_j$ for $1 \leq i \neq j \leq n$ and $c_{ii} = 0$ for $1 \leq i \leq n$, based on some set of values $g_1, \ldots, g_n$. The multiplicity of such possible decompositions results from the choice of where to essentially place the root in the type of graphical tree representation we will generally use. Generally, for us, the root will be placed half-way along the longest path in the tree, generating a decomposition of the matrix $\mathbf{A}$ using a procedure from Barthélemy and Guénoche (1991, Section 3.3.3):

(a) Given $\mathbf{A}$, let $O_{i*}, O_{j*} \in S$ denote the two objects between which the longest path is defined in the tree, i.e., the pair of objects O_{i*} and O_{j*} is associated with the largest entry in $\mathbf{A}$, say a_{i*j*}.

(b) Define $\mathbf{U}$ by letting

$$u_{ij} = a_{ij} - (g_i + g_j), \text{ where } g_i = \max\{a_{ii*}, a_{jj*}\} - M,$$

with M chosen so that $u_{ij} > 0$ for $i \neq j$. The matrix $\mathbf{C} = \{c_{ij}\}$ is then constructed by letting $c_{ii} = 0$ for $1 \leq i \leq n$, and $c_{ij} = g_i + g_j$ for $1 \leq i \neq j \leq n$. (If M is set equal to the largest entry a_{i*j*}, the values in $\mathbf{U}$ would have to be positive, and two values among $g_1, \ldots, g_n$ would be zero with the remainder less than or equal to zero. Thus, a value for M

less than a_{i*j*} is usually found by trial-and-error that will give positive entries within $\mathbf{U}$ and as many positive values as possible for $g_1, \ldots, g_n$.)

To construct the type of graphical additive tree representation we will give below, the process followed is to first graph the dendrogram induced by $\mathbf{U}$, where (as for any ultrametric) the chosen root is equidistant from all terminal nodes. The branches connecting the terminal nodes are then lengthened or shortened depending on the signs and absolute magnitudes of $g_1, \ldots, g_n$. If one were willing to consider the (arbitrary) inclusion of a sufficiently large additive constant to the entries of $\mathbf{A}$, the values of $g_1, \ldots, g_n$ could be assumed non-negative. In this case, the matrix $\mathbf{C}$ would represent what is now commonly called a centroid metric (see, for example, the usage in Barthélemy and Guénoche, 1991, Chapter 3); although having some advantages (particularly for some of the graphical representations we give in avoiding the issue of presenting negative branch lengths), such a restriction is not absolutely necessary for what we do in the sequel. In fact, even though some of the entries among $g_1, \ldots, g_n$ may be negative, for convenience we will still routinely refer to a centroid metric (component) even though some of the defined distances may actually be negative.

2.1 Fitting a Given Additive Tree in the L_2 Norm

The MATLAB function, `atreefit.m`, listed in Appendix A.3 and with usage

```
[fit,vaf] = atreefit(prox,targ)
```

parallels that of `ultrafit.m` of Section 1.1; it generates (again using iterative projection based on the linear (in)equality constraints obtained from a fixed additive tree — see Section 1.1.5 in the LUS Toolbox) the best-fitting additive tree in the L_2-norm (`FIT`) within the same equivalence class as that of a given additive tree matrix `TARG`. The matrix `PROX` contains the symmetric input proximities and `VAF` is the variance-accounted-for.

In the example below, the target matrix is again `numcltarg` obtained from the complete-link hierarchical clustering of `number.dat`; the `VAF` generated by these (now considered as additive tree) constraints is .6249 (and, as to be expected, is a value larger than for the corresponding best-fitting ultrametric value of .4781).

```
[fit,vaf] = atreefit(number,numcltarg)

fit =

  Columns 1 through 6

         0    0.4210    0.7185    0.7371    0.7092    0.8188
    0.4210         0    0.5334    0.5520    0.5241    0.6337
    0.7185    0.5334         0    0.4882    0.0590    0.5700
    0.7371    0.5520    0.4882         0    0.4790    0.4337
    0.7092    0.5241    0.0590    0.4790         0    0.5607
    0.8188    0.6337    0.5700    0.4337    0.5607         0
    0.7116    0.5265    0.4627    0.2506    0.4535    0.4082
    0.8670    0.6818    0.6181    0.4818    0.6089    0.4000
    0.7549    0.5698    0.3111    0.5247    0.3019    0.6064
    0.8318    0.6467    0.5830    0.2630    0.5737    0.5284

  Columns 7 through 10

    0.7116    0.8670    0.7549    0.8318
    0.5265    0.6818    0.5698    0.6467
    0.4627    0.6181    0.3111    0.5830
    0.2506    0.4818    0.5247    0.2630
    0.4535    0.6089    0.3019    0.5737
    0.4082    0.4000    0.6064    0.5284
         0    0.4563    0.4992    0.3454
    0.4563         0    0.6546    0.5766
    0.4992    0.6546         0    0.6194
    0.3454    0.5766    0.6194         0

vaf =

    0.6249
```

2.2 Finding an Additive Tree in the L_2 Norm

In analogy to the m-file, `ultrafnd.m`, from Section 1.2 for identifying best-fitting ultrametrics, Appendix A.4 provides `atreefnd.m` that again implements the Hubert and Arabie (1995) heuristic search strategy using iterative projection but now for constructing the best-fitting additive trees in the L_2-norm. The usage has the form

```
[find,vaf] = atreefnd(prox,inperm)
```

where `FIND` is the identified additive tree with variance-accounted-for `VAF`. Again, the matrix `PROX` contains the symmetric input proximities, and `INPERM` is a permutation that defines an order in which the constraints are considered over all object quadruples. In the example below, two such searches are shown starting with random permutations (through the use of `randperm(10)`) that give vafs of .6359 and .6249.

```
[find,vaf] = atreefnd(number,randperm(10))

find =
  Columns 1 through 6

         0    0.4210    0.6467    0.6448    0.6374    0.8049
    0.4210         0    0.4616    0.4596    0.4523    0.6198
    0.6467    0.4616         0    0.3634    0.0590    0.5235
    0.6448    0.4596    0.3634         0    0.3542    0.4385
    0.6374    0.4523    0.0590    0.3542         0    0.5143
    0.8049    0.6198    0.5235    0.4385    0.5143         0
    0.7523    0.5671    0.4709    0.3858    0.4617    0.4132
    0.9263    0.7412    0.6449    0.5599    0.6357    0.5872
    0.8634    0.6783    0.5820    0.4970    0.5728    0.5244
    0.8733    0.6881    0.5919    0.5068    0.5827    0.5342

  Columns 7 through 10

    0.7523    0.9263    0.8634    0.8733
    0.5671    0.7412    0.6783    0.6881
    0.4709    0.6449    0.5820    0.5919
    0.3858    0.5599    0.4970    0.5068
    0.4617    0.6357    0.5728    0.5827
    0.4132    0.5872    0.5244    0.5342
         0    0.3930    0.3301    0.3400
    0.3930         0    0.4000    0.4569
    0.3301    0.4000         0    0.3941
    0.3400    0.4569    0.3941         0

vaf =

    0.6359

[find,vaf] = atreefnd(number,randperm(10))

find =

  Columns 1 through 6

         0    0.4210    0.7185    0.7371    0.7092    0.8188
    0.4210         0    0.5334    0.5520    0.5241    0.6337
    0.7185    0.5334         0    0.4882    0.0590    0.5700
    0.7371    0.5520    0.4882         0    0.4790    0.4337
    0.7092    0.5241    0.0590    0.4790         0    0.5607
    0.8188    0.6337    0.5700    0.4337    0.5607         0
    0.7116    0.5265    0.4627    0.2506    0.4535    0.4082
    0.8670    0.6818    0.6181    0.4818    0.6089    0.4000
    0.7549    0.5698    0.3111    0.5247    0.3019    0.6064
    0.8318    0.6467    0.5830    0.2630    0.5737    0.5284

  Columns 7 through 10

    0.7116    0.8670    0.7549    0.8318
    0.5265    0.6818    0.5698    0.6467
    0.4627    0.6181    0.3111    0.5830
    0.2506    0.4818    0.5247    0.2630
    0.4535    0.6089    0.3019    0.5737
    0.4082    0.4000    0.6064    0.5284
         0    0.4563    0.4992    0.3454
    0.4563         0    0.6546    0.5766
    0.4992    0.6546         0    0.6194
    0.3454    0.5766    0.6194         0

vaf =

    0.6249
```

2.3 Decomposing an Additive Tree

The m-file given in Appendix A.5, `atreedec.m`, decomposes a given additive tree matrix into an ultrametric and a centroid metric matrix (where the root is half-way along the longest path). The form of the usage is

```
[ulmetric,ctmetric] = atreedec(prox,constant)
```

where `PROX` is the input (additive tree) proximity matrix (with a zero main diagonal and a dissimilarity interpretation); `CONSTANT` is a nonnegative number (less than or equal to the maximum proximity value) that controls the positivity of the constructed ultrametric values; `ULMETRIC` is the ultrametric component of the decomposition; `CTMETRIC` is the centroid metric component (given by values $g_1, \ldots, g_n$ assigned to each of the objects, some of which may actually be negative depending on the input proximity matrix used). In the example below, the additive tree matrix identified earlier with a vaf of .6359, is decomposed using a value of .70 for the constant to control the positivity of the ultrametric values.

```
load number.dat
```

```
[find,vaf] = atreefnd(number,randperm(10))

find =

  Columns 1 through 7

         0    0.4210    0.6467    0.6448    0.6374    0.8049    0.7523
    0.4210         0    0.4616    0.4596    0.4523    0.6198    0.5671
    0.6467    0.4616         0    0.3634    0.0590    0.5235    0.4709
    0.6448    0.4596    0.3634         0    0.3542    0.4385    0.3858
    0.6374    0.4523    0.0590    0.3542         0    0.5143    0.4617
    0.8049    0.6198    0.5235    0.4385    0.5143         0    0.4132
    0.7523    0.5671    0.4709    0.3858    0.4617    0.4132         0
    0.9263    0.7412    0.6449    0.5599    0.6357    0.5872    0.3930
    0.8634    0.6783    0.5820    0.4970    0.5728    0.5244    0.3301
    0.8733    0.6881    0.5919    0.5068    0.5827    0.5342    0.3400

  Columns 8 through 10

    0.9263    0.8634    0.8733
    0.7412    0.6783    0.6881
    0.6449    0.5820    0.5919
    0.5599    0.4970    0.5068
    0.6357    0.5728    0.5827
    0.5872    0.5244    0.5342
    0.3930    0.3301    0.3400
         0    0.4000    0.4569
    0.4000         0    0.3941
    0.4569    0.3941         0


vaf =

    0.6359

[ulmetric,ctmetric] = atreedec(find,.70)

ulmetric =

  Columns 1 through 7

         0    0.1536    0.4737    0.4737    0.4737    0.4737    0.4737
    0.1536         0    0.4737    0.4737    0.4737    0.4737    0.4737
    0.4737    0.4737         0    0.4720    0.1749    0.4720    0.4720
    0.4737    0.4737    0.4720         0    0.4720    0.3888    0.3888
    0.4737    0.4737    0.1749    0.4720         0    0.4720    0.4720
    0.4737    0.4737    0.4720    0.3888    0.4720         0    0.2560
    0.4737    0.4737    0.4720    0.3888    0.4720    0.2560         0
    0.4737    0.4737    0.4720    0.3888    0.4720    0.2560    0.1144
    0.4737    0.4737    0.4720    0.3888    0.4720    0.2560    0.1144
    0.4737    0.4737    0.4720    0.3888    0.4720    0.2560    0.1144

  Columns 8 through 10

    0.4737    0.4737    0.4737
    0.4737    0.4737    0.4737
    0.4720    0.4720    0.4720
    0.3888    0.3888    0.3888
    0.4720    0.4720    0.4720
    0.2560    0.2560    0.2560
    0.1144    0.1144    0.1144
         0    0.0103    0.0574
    0.0103         0    0.0574
    0.0574    0.0574         0


ctmetric =

    0.2263
    0.0412
   -0.0533
   -0.0552
   -0.0626
    0.1049
    0.0523
    0.2263
    0.1634
    0.1733

[orderprox,orderperm] = ultraorder(ulmetric)


orderprox =
  Columns 1 through 7

         0    0.1536    0.4737    0.4737    0.4737    0.4737    0.4737
    0.1536         0    0.4737    0.4737    0.4737    0.4737    0.4737
    0.4737    0.4737         0    0.1749    0.4720    0.4720    0.4720
    0.4737    0.4737    0.1749         0    0.4720    0.4720    0.4720
    0.4737    0.4737    0.4720    0.4720         0    0.2560    0.2560
    0.4737    0.4737    0.4720    0.4720    0.2560         0    0.1144
    0.4737    0.4737    0.4720    0.4720    0.2560    0.1144         0
    0.4737    0.4737    0.4720    0.4720    0.2560    0.1144    0.0574
    0.4737    0.4737    0.4720    0.4720    0.2560    0.1144    0.0574
    0.4737    0.4737    0.4720    0.4720    0.3888    0.3888    0.3888

  Columns 8 through 10

    0.4737    0.4737    0.4737
    0.4737    0.4737    0.4737
    0.4720    0.4720    0.4720
    0.4720    0.4720    0.4720
    0.2560    0.2560    0.3888
    0.1144    0.1144    0.3888
    0.0574    0.0574    0.3888
         0    0.0103    0.3888
    0.0103         0    0.3888
    0.3888    0.3888         0


orderperm =

     2     1     3     5     6     7    10     9     8     4
```

2.4 Representing an Additive Tree (Graphically)

The information present in an additive tree can be provided in several ways. First, given the decomposition into an ultrametric and a centroid metric, the partition hierarchy induced by the ultrametric could be given explicitly, along with the levels at which the various new subsets in the partitions are formed. The fitted additive tree values could then be identified as a sum of (a) the level at which an object pair, say O_i and O_j, first appear together within a common subset of the hierarchy, and (b) the sum of g_i and g_j for the pair from the centroid metric component. As an illustration using the example just given in Section 2.3, the partition hierarchy has the form:

Partition	Level Formed
{{1,0,2,4,5,6,9,8,7,3}}	.47
{{1,0},{2,4},{5,6,9,8,7,3}}	.39
{{1,0},{2,4},{5,6,9,8,7},{3}}	.26
{{1,0},{2,4},{5},{6,9,8,7},{3}}	.17
{{1,0},{2},{4},{5},{6,9,8,7},{3}}	.15
{{1},{0},{2},{4},{5},{6,9,8,7},{3}}	.11
{{1},{0},{2},{4},{5},{6},{9,8,7},{3}}	.06
{{1},{0},{2},{4},{5},{6},{9},{8,7},{3}}	.01
{{1},{0},{2},{4},{5},{6},{9},{8},{7},{3}}	.01

with centroid metric values of:

digit	g_i
0	.23
1	.04
2	-.05
3	-.06
4	-.06
5	.10
6	.05
7	.23
8	.16
9	.17

Thus, the additive tree value for the digit pair (3,6) of .39 [.3858] is formed from the level .39 [.3883] at which 3 and 6 first appear together in the hierarchy, plus the sum of the g_is for the two digits of -.05 [-.0552] and .05 [.0523]. A dendrogram representation for the partition hierarchy is given in Figure 2.1.

A graphical representation for the additive tree is given in Figure 2.2 which was obtained from the dendrogram of Figure 2.1 by stretching and shrinking the branches attached to the terminal nodes by the g_i values (and cutting the vertical scale given in the dendrogram by half). Thus, the length of a path in the tree from one terminal node to another (ignoring all horizontal lines as having lengths of zero), would generate the values given in the additive tree matrix.

Figure 2.1: A Dendrogram (Tree) Representation for the Ultrametric Component of the Additive Tree Described in the Text Having Vaf of .6359

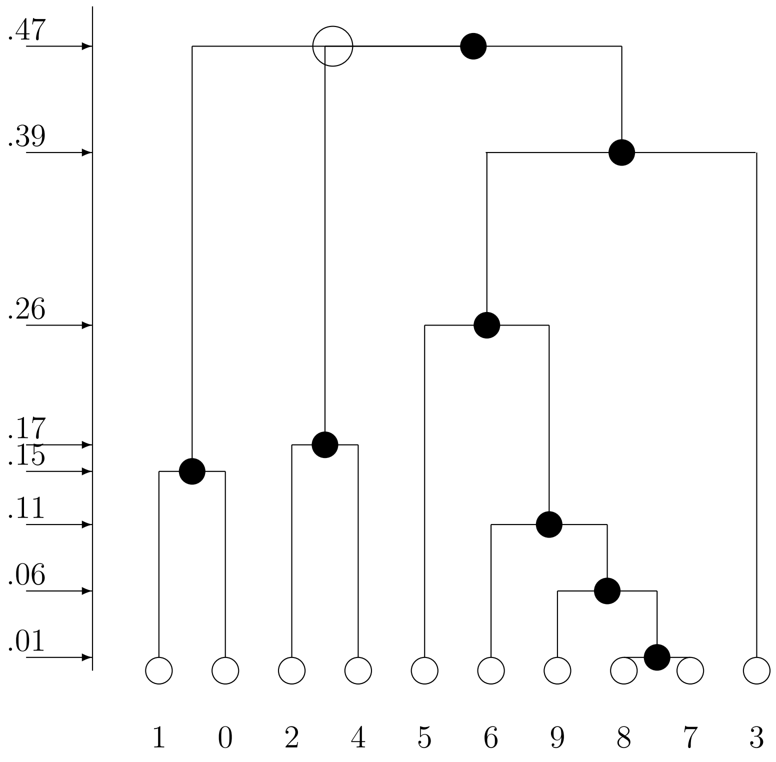
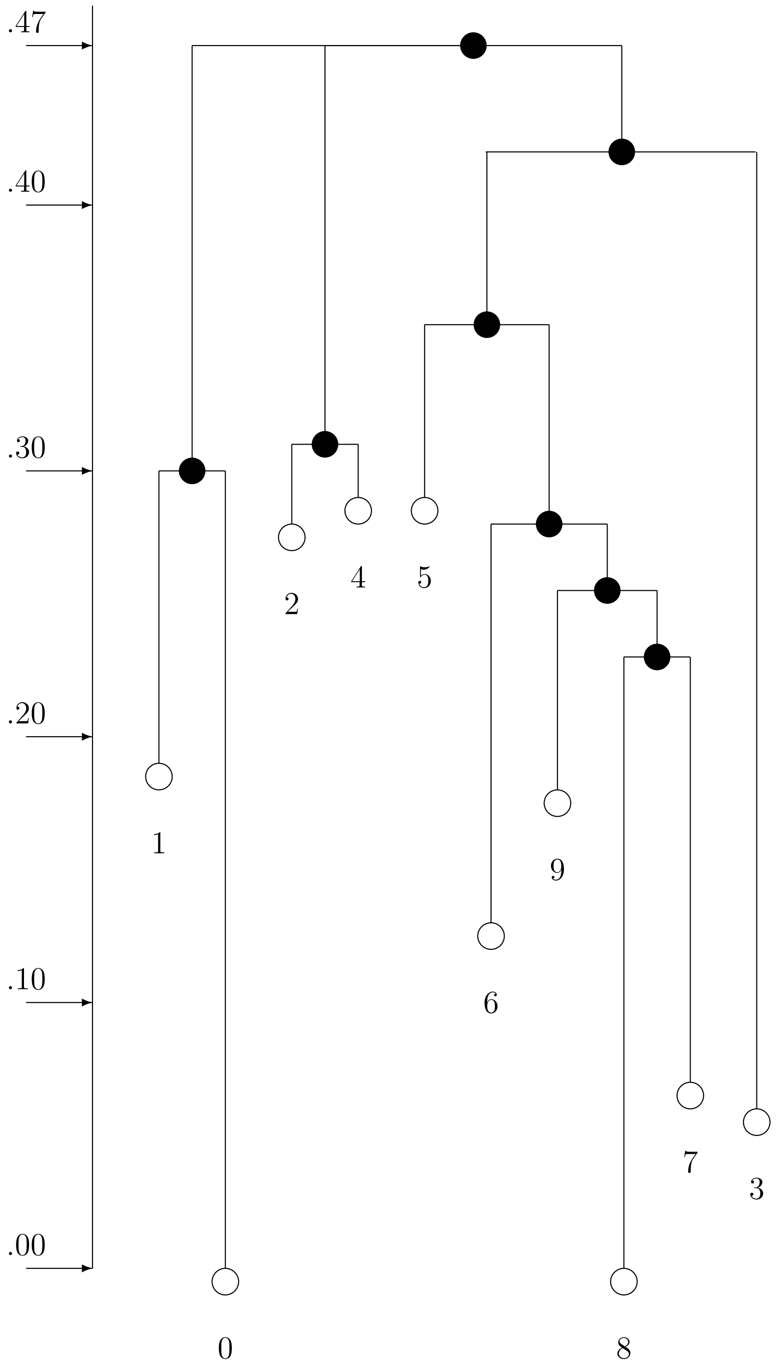


Figure 2.2: A Graph-Theoretic Representation for the Additive Tree Described in the Text Having Vaf of .6359



2.5 An Alternative for Finding an Additive Tree in the L_2 Norm (Based on Combining a Centroid Metric and an Ultrametric)

If the four-point condition characterizing an additive tree is strengthened so that all the sums in the defining conditions for all object quadruples are equal (and not only for the largest two such sums), the additive tree matrix so obtained has entries representable as $g_i + g_j$, for a collection of values $g_1, \ldots, g_n$. This specially constrained additive tree is usually referred to as a centroid metric, and as noted by Carroll and Pruzansky (1980) and De Soete, DeSarbo, Furnas, & Carroll (1984), can be fitted to a proximity matrix in the L_2-norm through closed-form expressions. Specifically, if $\mathbf{P}$ denotes the proximity matrix, then g_i can be given as the i^{th} row sum of $\mathbf{P}$ excluding the diagonal entry, divided by $n - 2$, minus the total off-diagonal sum divided by $2(n-1)(n-2)$.

Appendix A.6 gives an m-file, `centfit.m`, for obtaining the best-fitting centroid metric in the L_2-norm, with usage

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

where `PROX` is the usual input proximity matrix (with a zero main diagonal and a dissimilarity interpretation); the n values that serve to define the approximating sums, $g_i + g_j$, present in the fitted matrix `FIT`, are given in the vector `LENGTHS` of size $n \times 1$. The example below uses `centfit.m` with the `number.dat` data set, leading to an additive tree with vaf of .3248; the latter could be represented graphically as a "star" tree with one internal node and spokes having the lengths given in the output vector `LENGTHS`

```
load number.dat
[fit,vaf,lengths] = centfit(number)

fit =

  Columns 1 through 7

         0    0.7808    0.6877    0.6709    0.6784    0.7647    0.6589
    0.7808         0    0.5026    0.4858    0.4933    0.5796    0.4738
    0.6877    0.5026         0    0.3927    0.4002    0.4864    0.3807
    0.6709    0.4858    0.3927         0    0.3834    0.4697    0.3639
    0.6784    0.4933    0.4002    0.3834         0    0.4772    0.3714
    0.7647    0.5796    0.4864    0.4697    0.4772         0    0.4577
    0.6589    0.4738    0.3807    0.3639    0.3714    0.4577         0
    0.8128    0.6277    0.5346    0.5178    0.5253    0.6116    0.5058
    0.7499    0.5648    0.4717    0.4549    0.4624    0.5487    0.4429
    0.7657    0.5806    0.4874    0.4707    0.4782    0.5644    0.4587
  Columns 8 through 10

    0.8128    0.7499    0.7657
    0.6277    0.5648    0.5806
    0.5346    0.4717    0.4874
    0.5178    0.4549    0.4707
    0.5253    0.4624    0.4782
    0.6116    0.5487    0.5644
    0.5058    0.4429    0.4587
         0    0.5968    0.6126
    0.5968         0    0.5497
    0.6126    0.5497         0

vaf =

    0.3248

lengths =

  Columns 1 through 7

    0.4830    0.2978    0.2047    0.1880    0.1955    0.2817    0.1760

  Columns 8 through 10

    0.3298    0.2670    0.2827
```

An alternative strategy for identifying good-fitting additive trees (and one that will be used in a slightly different form on two-mode proximity data in Section 4.2) relies on the possible decomposition of an additive tree into an ultrametric and centroid metric. The m-file in Appendix A.7, `atreectul.m`, first fits a centroid metric in closed form; on the residual matrix an ultrametric is then identified. The sum of these two matrices is an additive tree. The usage would follow that of `atreefnd.m`:

```
[find,vaf] = atreectul(prox,inperm)
```

where `FIND` is the identified additive tree with variance-accounted-for `VAF`. Again, the matrix `PROX` contains the symmetric input proximities, and `INPERM` is a permutation that defines an order in which the constraints are considered over all object triples in the identification of the ultrametric component. In the example below, one search is shown starting with a random permutation (through the use of `randperm(10)`) that gives the same additive tree identified earlier with a vaf of .6249.

```
[find,vaf] = atreectul(number,randperm(10))

find =

  Columns 1 through 7

         0    0.4210    0.7185    0.7371    0.7092    0.8188    0.7116
    0.4210         0    0.5334    0.5520    0.5241    0.6337    0.5265
    0.7185    0.5334         0    0.4882    0.0590    0.5700    0.4627
    0.7371    0.5520    0.4882         0    0.4790    0.4337    0.2506
    0.7092    0.5241    0.0590    0.4790         0    0.5607    0.4535
    0.8188    0.6337    0.5700    0.4337    0.5607         0    0.4082
    0.7116    0.5265    0.4627    0.2506    0.4535    0.4082         0
    0.8670    0.6818    0.6181    0.4818    0.6089    0.4000    0.4563
    0.7549    0.5698    0.3111    0.5247    0.3019    0.6064    0.4992
    0.8318    0.6467    0.5830    0.2630    0.5737    0.5284    0.3454

  Columns 8 through 10

    0.8670    0.7549    0.8318
    0.6818    0.5698    0.6467
    0.6181    0.3111    0.5830
    0.4818    0.5247    0.2630
    0.6089    0.3019    0.5737
    0.4000    0.6064    0.5284
    0.4563    0.4992    0.3454
         0    0.6546    0.5766
    0.6546         0    0.6194
    0.5766    0.6194         0

vaf =

    0.6249
```

Chapter 3

Fitting Multiple Tree Structures to a Symmetric Proximity Matrix

The use of multiple structures, whether they be ultrametrics or additive trees, to additively represent a given proximity matrix, 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 variance-accounted-for by the sum of both fitted matrices no longer changes by a set amount (the value of 1.0e-006 is used in the m-files of the next two sections).

3.1 Multiple Ultrametrics

The m-file, `biultrafnd`, in Appendix A.8 fits (additively) two ultrametric matrices in the L_2-norm. The explicit usage is

```
[find,vaf,targone,targtwo] = biultrafnd(prox,inperm)
```

where `PROX` is the given input proximity matrix (with a zero main diagonal and a dissimilarity interpretation); `INPERM` is a permutation that determines the order in which the inequality constraints are considered (and thus can be made random to search for different locally optimal representations); `FIND` is the found least-squares matrix (with variance-accounted-for of `VAF`) to `PROX`, and is the sum of the two ultrametric matrices `TARGONE` and `TARGTWO`.

In the example to follow, a vaf of .8001 was achieved for the two identified ultrametrics (and where one needs to add an (arbitrary) constant [e.g., of .40] to the entries in `TARGTWO` to satisfy the technical requirement that ultrametric values should be nonnegative). It might be noted substantively that the first ultrametric matrix (in `TARGONE`) reflects the structural properties of the digits; the second ultrametric matrix (in `TARGTWO`) is completely consistent with digit magnitude. This is a very nice mixture of ultrametric structures with a convenient substantive interpretation for both components.

```
[find,vaf,targone,targtwo] = biultrafnd(number,randperm(10))

find =

  Columns 1 through 7

         0    0.4210    0.5266    0.6075    0.7685    0.7685    0.8694
    0.4210         0    0.3414    0.4224    0.5834    0.5834    0.6843
    0.5266    0.3414         0    0.3791    0.0589    0.5401    0.6410
    0.6075    0.4224    0.3791         0    0.5401    0.4511    0.3669
    0.7685    0.5834    0.0589    0.5401         0    0.4090    0.6410
    0.7685    0.5834    0.5401    0.4511    0.4090         0    0.5519
    0.8694    0.6843    0.6410    0.3669    0.6410    0.5519         0
    0.8694    0.6843    0.6410    0.5519    0.6410    0.4000    0.3640
    0.8694    0.6843    0.3065    0.6410    0.3065    0.6410    0.3500
    0.8694    0.6843    0.6410    0.2630    0.6410    0.5519    0.2293

  Columns 8 through 10

    0.8694    0.8694    0.8694
    0.6843    0.6843    0.6843
    0.6410    0.3065    0.6410
    0.5519    0.6410    0.2630
    0.6410    0.3065    0.6410
    0.4000    0.6410    0.5519
    0.3640    0.3500    0.2293
         0    0.4530    0.4143
    0.4530         0    0.5034
    0.4143    0.5034         0


vaf =

    0.8001


targone =
  Columns 1 through 7

         0    0.7796    0.7796    0.7796    0.7796    0.7796    0.7796
    0.7796         0    0.5945    0.5945    0.5945    0.5945    0.5945
    0.7796    0.5945         0    0.5512    0.0701    0.5512    0.5512
    0.7796    0.5945    0.5512         0    0.5512    0.4622    0.2772
    0.7796    0.5945    0.0701    0.5512         0    0.5512    0.5512
    0.7796    0.5945    0.5512    0.4622    0.5512         0    0.4622
    0.7796    0.5945    0.5512    0.2772    0.5512    0.4622         0
    0.7796    0.5945    0.5512    0.4622    0.5512    0.3103    0.4622
    0.7796    0.5945    0.2168    0.5512    0.2168    0.5512    0.5512
    0.7796    0.5945    0.5512    0.1733    0.5512    0.4622    0.2772

  Columns 8 through 10

    0.7796    0.7796    0.7796
    0.5945    0.5945    0.5945
    0.5512    0.2168    0.5512
    0.4622    0.5512    0.1733
    0.5512    0.2168    0.5512
    0.3103    0.5512    0.4622
    0.4622    0.5512    0.2772
         0    0.5512    0.4622
    0.5512         0    0.5512
    0.4622    0.5512         0


targtwo =

  Columns 1 through 7

         0   -0.3586   -0.2531   -0.1721   -0.0111   -0.0111    0.0897
   -0.3586         0   -0.2531   -0.1721   -0.0111   -0.0111    0.0897
   -0.2531   -0.2531         0   -0.1721   -0.0111   -0.0111    0.0897
   -0.1721   -0.1721   -0.1721         0   -0.0111   -0.0111    0.0897
   -0.0111   -0.0111   -0.0111   -0.0111         0   -0.1422    0.0897
   -0.0111   -0.0111   -0.0111   -0.0111   -0.1422         0    0.0897
    0.0897    0.0897    0.0897    0.0897    0.0897    0.0897         0
    0.0897    0.0897    0.0897    0.0897    0.0897    0.0897   -0.0982
    0.0897    0.0897    0.0897    0.0897    0.0897    0.0897   -0.2012
    0.0897    0.0897    0.0897    0.0897    0.0897    0.0897   -0.0479
  Columns 8 through 10

    0.0897    0.0897    0.0897
    0.0897    0.0897    0.0897
    0.0897    0.0897    0.0897
    0.0897    0.0897    0.0897
    0.0897    0.0897    0.0897
    0.0897    0.0897    0.0897
   -0.0982   -0.2012   -0.0479
         0   -0.0982   -0.0479
   -0.0982         0   -0.0479
   -0.0479   -0.0479         0

[orderprox,orderperm] = ultraorder(targone)

orderprox =

  Columns 1 through 7

         0    0.7796    0.7796    0.7796    0.7796    0.7796    0.7796
    0.7796         0    0.2168    0.2168    0.5512    0.5512    0.5512
    0.7796    0.2168         0    0.0701    0.5512    0.5512    0.5512
    0.7796    0.2168    0.0701         0    0.5512    0.5512    0.5512
    0.7796    0.5512    0.5512    0.5512         0    0.1733    0.2772
    0.7796    0.5512    0.5512    0.5512    0.1733         0    0.2772
    0.7796    0.5512    0.5512    0.5512    0.2772    0.2772         0
    0.7796    0.5512    0.5512    0.5512    0.4622    0.4622    0.4622
    0.7796    0.5512    0.5512    0.5512    0.4622    0.4622    0.4622
    0.7796    0.5945    0.5945    0.5945    0.5945    0.5945    0.5945

  Columns 8 through 10

    0.7796    0.7796    0.7796
    0.5512    0.5512    0.5945
    0.5512    0.5512    0.5945
    0.5512    0.5512    0.5945
    0.4622    0.4622    0.5945
    0.4622    0.4622    0.5945
    0.4622    0.4622    0.5945
         0    0.3103    0.5945
    0.3103         0    0.5945
    0.5945    0.5945         0
```

```
orderperm =

     1     9     3     5    10     4     7     8     6     2

[orderprox,orderperm] = ultraorder(targtwo)

orderprox =

  Columns 1 through 7

         0   -0.3586   -0.2531   -0.1721   -0.0111   -0.0111    0.0897
   -0.3586         0   -0.2531   -0.1721   -0.0111   -0.0111    0.0897
   -0.2531   -0.2531         0   -0.1721   -0.0111   -0.0111    0.0897
   -0.1721   -0.1721   -0.1721         0   -0.0111   -0.0111    0.0897
   -0.0111   -0.0111   -0.0111   -0.0111         0   -0.1422    0.0897
   -0.0111   -0.0111   -0.0111   -0.0111   -0.1422         0    0.0897
    0.0897    0.0897    0.0897    0.0897    0.0897    0.0897         0
    0.0897    0.0897    0.0897    0.0897    0.0897    0.0897   -0.0982
    0.0897    0.0897    0.0897    0.0897    0.0897    0.0897   -0.0982
    0.0897    0.0897    0.0897    0.0897    0.0897    0.0897   -0.0479

  Columns 8 through 10

    0.0897    0.0897    0.0897
    0.0897    0.0897    0.0897
    0.0897    0.0897    0.0897
    0.0897    0.0897    0.0897
    0.0897    0.0897    0.0897
    0.0897    0.0897    0.0897
   -0.0982   -0.0982   -0.0479
         0   -0.2012   -0.0479
   -0.2012         0   -0.0479
   -0.0479   -0.0479         0


orderperm =

     1     2     3     4     5     6     8     7     9    10
```

3.2 Multiple Additive Trees

The m-file, `biatreefnd`, in Appendix A.9 fits (additively) two additive tree matrices in the L_2-norm. The explicit usage is

```
[find,vaf,targone,targtwo] = biatreefnd(prox,inperm)
```

where `PROX` is the given input proximity matrix (with a zero main diagonal and a dissimilarity interpretation); `INPERM` is a permutation that determines the order in which the inequality constraints are considered (and thus can be made random to search for different locally optimal representations); `FIND` is the found least-squares matrix (with variance-accounted-for of `VAF`) to `PROX`, and is the sum of the two additive tree matrices `TARGONE` and `TARGTWO`.

In the example to follow, a vaf of .9003 was achieved for the two identified additive trees (and where one needs as in the multiple ultrametric case, to add an (arbitrary) constant [e.g., of .40] to the entries in `TARGTWO` to satisfy the technical requirement that additive tree values should be nonnegative). Similarly as in the interpretation for the example of the last section, it might be noted substantively that the second additive tree matrix (in `TARGTWO`) reflects the structural properties of the digits; the first matrix (in `TARGONE`) is completely consistent with digit magnitude. So, again we have a very nice mixture of structures with convenient substantive interpretations for both components.

```
load number.dat
[find,vaf,targone,targtwo] = biatreefnd(number,randperm(10))
find =

  Columns 1 through 7

         0    0.4210    0.4955    0.6196    0.7487    0.7882    0.7083
    0.4210         0    0.3725    0.3411    0.6256    0.5723    0.5852
    0.4955    0.3725         0    0.4483    0.0590    0.6169    0.4747
    0.6196    0.3411    0.4483         0    0.4683    0.4150    0.4279
    0.7487    0.6256    0.0590    0.4683         0    0.4090    0.3766
    0.7882    0.5723    0.6169    0.4150    0.4090         0    0.4784
    0.7083    0.5852    0.4747    0.4279    0.3766    0.4784         0
    0.9558    0.7399    0.7845    0.5826    0.6864    0.4000    0.4013
    0.9246    0.8015    0.3556    0.6442    0.2574    0.6947    0.3078
    0.9084    0.6299    0.7371    0.2630    0.6390    0.5857    0.3539

  Columns 8 through 10

    0.9558    0.9246    0.9084
    0.7399    0.8015    0.6299
    0.7845    0.3556    0.7371
    0.5826    0.6442    0.2630
    0.6864    0.2574    0.6390
    0.4000    0.6947    0.5857
    0.4013    0.3078    0.3539
         0    0.4000    0.3947
    0.4000         0    0.4563
    0.3947    0.4563         0


vaf =

    0.9003


targone =

  Columns 1 through 7

         0    0.4439    0.4582    0.6163    0.7114    0.7976    0.7699
    0.4439         0    0.2731    0.4312    0.5262    0.6125    0.5848
    0.4582    0.2731         0    0.3227    0.4178    0.5040    0.4763
    0.6163    0.4312    0.3227         0    0.3427    0.4290    0.4013
    0.7114    0.5262    0.4178    0.3427         0    0.2961    0.3782
    0.7976    0.6125    0.5040    0.4290    0.2961         0    0.4644
    0.7699    0.5848    0.4763    0.4013    0.3782    0.4644         0
    0.9652    0.7801    0.6716    0.5966    0.5735    0.6598    0.3874
    0.9023    0.7172    0.6087    0.5337    0.5106    0.5969    0.3245
    0.9051    0.7200    0.6115    0.5365    0.5134    0.5997    0.3273

  Columns 8 through 10

    0.9652    0.9023    0.9051
    0.7801    0.7172    0.7200
    0.6716    0.6087    0.6115
    0.5966    0.5337    0.5365
    0.5735    0.5106    0.5134
    0.6598    0.5969    0.5997
    0.3874    0.3245    0.3273
         0    0.3022    0.4087
    0.3022         0    0.3458
    0.4087    0.3458         0
```

```
targtwo =

  Columns 1 through 7

        0   -0.0229    0.0373    0.0033    0.0373   -0.0094   -0.0616
  -0.0229         0    0.0994   -0.0901    0.0994   -0.0402    0.0005
   0.0373    0.0994         0    0.1256   -0.3588    0.1129   -0.0016
   0.0033   -0.0901    0.1256         0    0.1256   -0.0140    0.0267
   0.0373    0.0994   -0.3588    0.1256         0    0.1129   -0.0016
  -0.0094   -0.0402    0.1129   -0.0140    0.1129         0    0.0139
  -0.0616    0.0005   -0.0016    0.0267   -0.0016    0.0139         0
  -0.0094   -0.0402    0.1129   -0.0140    0.1129   -0.2598    0.0139
   0.0222    0.0843   -0.2532    0.1105   -0.2532    0.0978   -0.0167
   0.0033   -0.0901    0.1256   -0.2735    0.1256   -0.0140    0.0267

  Columns 8 through 10

  -0.0094    0.0222    0.0033
  -0.0402    0.0843   -0.0901
   0.1129   -0.2532    0.1256
  -0.0140    0.1105   -0.2735
   0.1129   -0.2532    0.1256
  -0.2598    0.0978   -0.0140
   0.0139   -0.0167    0.0267
        0    0.0978   -0.0140
   0.0978         0    0.1105
  -0.0140    0.1105         0

[ulmetricone,ctmetricone] = atreedec(targone,0.0)

ulmetricone =

  Columns 1 through 7

        0   -1.3014   -1.1786   -0.9652   -0.9652   -0.9652   -0.9652
  -1.3014         0   -1.1786   -0.9652   -0.9652   -0.9652   -0.9652
  -1.1786   -1.1786         0   -0.9652   -0.9652   -0.9652   -0.9652
  -0.9652   -0.9652   -0.9652         0   -0.9850   -0.9850   -0.9850
  -0.9652   -0.9652   -0.9652   -0.9850         0   -1.2128   -1.1031
  -0.9652   -0.9652   -0.9652   -0.9850   -1.2128         0   -1.1031
  -0.9652   -0.9652   -0.9652   -0.9850   -1.1031   -1.1031         0
```

```
   -0.9652   -0.9652   -0.9652   -0.9850   -1.1031   -1.1031   -1.3477
   -0.9652   -0.9652   -0.9652   -0.9850   -1.1031   -1.1031   -1.3477
   -0.9652   -0.9652   -0.9652   -0.9850   -1.1031   -1.1031   -1.3477

  Columns 8 through 10

   -0.9652   -0.9652   -0.9652
   -0.9652   -0.9652   -0.9652
   -0.9652   -0.9652   -0.9652
   -0.9850   -0.9850   -0.9850
   -1.1031   -1.1031   -1.1031
   -1.1031   -1.1031   -1.1031
   -1.3477   -1.3477   -1.3477
         0   -1.5653   -1.4617
   -1.5653         0   -1.4617
   -1.4617   -1.4617         0


ctmetricone =

    0.9652
    0.7801
    0.6716
    0.6163
    0.7114
    0.7976
    0.7699
    0.9652
    0.9023
    0.9051

[ulmetrictwo,ctmetrictwo] = atreedec(targtwo,0.0)

ulmetrictwo =

  Columns 1 through 7

         0   -0.1596   -0.1256   -0.1596   -0.1256   -0.1596   -0.1256
   -0.1596         0   -0.1256   -0.3151   -0.1256   -0.2525   -0.1256
   -0.1256   -0.1256         0   -0.1256   -0.6099   -0.1256   -0.1539
   -0.1596   -0.3151   -0.1256         0   -0.1256   -0.2525   -0.1256
   -0.1256   -0.1256   -0.6099   -0.1256         0   -0.1256   -0.1539
   -0.1596   -0.2525   -0.1256   -0.2525   -0.1256         0   -0.1256
   -0.1256   -0.1256   -0.1539   -0.1256   -0.1539   -0.1256         0
   -0.1596   -0.2525   -0.1256   -0.2525   -0.1256   -0.4855   -0.1256
   -0.1256   -0.1256   -0.4893   -0.1256   -0.4893   -0.1256   -0.1539
   -0.1596   -0.3151   -0.1256   -0.5247   -0.1256   -0.2525   -0.1256

  Columns 8 through 10

   -0.1596   -0.1256   -0.1596
   -0.2525   -0.1256   -0.3151
   -0.1256   -0.4893   -0.1256
   -0.2525   -0.1256   -0.5247
   -0.1256   -0.4893   -0.1256
   -0.4855   -0.1256   -0.2525
   -0.1256   -0.1539   -0.1256
         0   -0.1256   -0.2525
   -0.1256         0   -0.1256
   -0.2525   -0.1256         0


ctmetrictwo =

    0.0373
    0.0994
    0.1256
    0.1256
    0.1256
    0.1129
    0.0267
    0.1129
    0.1105
    0.1256

[orderproxone,orderpermone] = ultraorder(ulmetricone)

orderproxone =

  Columns 1 through 7

         0   -1.1786   -1.1786   -0.9652   -0.9652   -0.9652   -0.9652
   -1.1786         0   -1.3014   -0.9652   -0.9652   -0.9652   -0.9652
   -1.1786   -1.3014         0   -0.9652   -0.9652   -0.9652   -0.9652
   -0.9652   -0.9652   -0.9652         0   -1.2128   -1.1031   -1.1031
   -0.9652   -0.9652   -0.9652   -1.2128         0   -1.1031   -1.1031
   -0.9652   -0.9652   -0.9652   -1.1031   -1.1031         0   -1.4617
   -0.9652   -0.9652   -0.9652   -1.1031   -1.1031   -1.4617         0
   -0.9652   -0.9652   -0.9652   -1.1031   -1.1031   -1.4617   -1.5653
   -0.9652   -0.9652   -0.9652   -1.1031   -1.1031   -1.3477   -1.3477
   -0.9652   -0.9652   -0.9652   -0.9850   -0.9850   -0.9850   -0.9850

  Columns 8 through 10

   -0.9652   -0.9652   -0.9652
   -0.9652   -0.9652   -0.9652
   -0.9652   -0.9652   -0.9652
   -1.1031   -1.1031   -0.9850
   -1.1031   -1.1031   -0.9850
   -1.4617   -1.3477   -0.9850
   -1.5653   -1.3477   -0.9850
         0   -1.3477   -0.9850
   -1.3477         0   -0.9850
   -0.9850   -0.9850         0


orderpermone =

     3     2     1     5     6    10     9     8     7     4

[orderproxtwo,orderpermtwo] = ultraorder(ulmetrictwo)

orderproxtwo =

  Columns 1 through 7

         0   -0.1539   -0.1539   -0.1539   -0.1256   -0.1256   -0.1256
   -0.1539         0   -0.4893   -0.4893   -0.1256   -0.1256   -0.1256
   -0.1539   -0.4893         0   -0.6099   -0.1256   -0.1256   -0.1256
   -0.1539   -0.4893   -0.6099         0   -0.1256   -0.1256   -0.1256
   -0.1256   -0.1256   -0.1256   -0.1256         0   -0.5247   -0.3151
   -0.1256   -0.1256   -0.1256   -0.1256   -0.5247         0   -0.3151
   -0.1256   -0.1256   -0.1256   -0.1256   -0.3151   -0.3151         0
   -0.1256   -0.1256   -0.1256   -0.1256   -0.2525   -0.2525   -0.2525
   -0.1256   -0.1256   -0.1256   -0.1256   -0.2525   -0.2525   -0.2525
   -0.1256   -0.1256   -0.1256   -0.1256   -0.1596   -0.1596   -0.1596
  Columns 8 through 10

   -0.1256   -0.1256   -0.1256
   -0.1256   -0.1256   -0.1256
   -0.1256   -0.1256   -0.1256
   -0.1256   -0.1256   -0.1256
   -0.2525   -0.2525   -0.1596
   -0.2525   -0.2525   -0.1596
   -0.2525   -0.2525   -0.1596
         0   -0.4855   -0.1596
   -0.4855         0   -0.1596
   -0.1596   -0.1596         0


orderpermtwo =

     7     9     5     3    10     4     2     8     6     1
```

Chapter 4

Ultrametrics and Additive Trees for Two-Mode (Rectangular) Proximity Data

The proximity data considered thus far for obtaining some type of structural representation, such as an ultrametric or an additive tree, 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, that is *between* two distinct object sets, $S_A = \{O_{1A}, \ldots, O_{n_aA}\}$ and $S_B = \{O_{1B}, \ldots, O_{n_bB}\}$, containing n_a and n_b objects, respectively, and defined through an $n_a \times n_b$ proximity matrix $\mathbf{Q} = \{q_{rs}\}$, where again, for convenience, we assume that the entries in $\mathbf{Q}$ are keyed as dissimilarities. What may be desirable is a joint structural representation of the set $S_A \cap S_B$ (considered as a single object set S containing $n_a + n_b = n$ objects), but one that is based only on the available proximities between the sets S_A and S_B.

Conditions have been proposed in the literature for when the entries in a matrix fitted to $\mathbf{Q}$ characterize an ultrametric or an additive tree representation. In particular, suppose a $n_a \times n_b$ matrix $\mathbf{F} = \{f_{rs}\}$ is fitted to $\mathbf{Q}$ through least squares subject to the constraints that follow:

Ultrametric (Furnas, 1980):

for all distinct object quadruples, O_{rA}, O_{sA}, O_{rB}, O_{sB}, where O_{rA}, $O_{sA} \in S_A$ and O_{rB}, O_{sB}, $\in S_B$, and considering the entries in $\mathbf{F}$ corresponding to the pairs, (O_{rA}, O_{rB}), (O_{rA}, O_{sB}), (O_{sA} O_{rB}), and (O_{sA}, O_{sB}), say $f_{r_Ar_B}$, $f_{r_As_B}$, $f_{s_Ar_B}$, $f_{s_As_B}$, respectively, the largest two must be equal.

Additive trees (Brossier, 1987):

for all distinct object sextuples, O_{rA}, O_{sA}, O_{tA}, O_{rB}, O_{sB}, O_{tB}, where O_{rA}, O_{sA}, $O_{tA} \in S_A$ and O_{rB}, O_{sB}, O_{tB}, $\in S_B$, and considering the entries in $\mathbf{F}$ corresponding to the pairs (O_{rA}, O_{rB}), (O_{rA}, O_{sB}), (O_{rA}, O_{tB}), (O_{sA}, O_{rB}), (O_{sA}, O_{sB}), (O_{sA}, O_{tB}), (O_{tA}, O_{rB}), (O_{tA}, O_{sB}), and (O_{tA}, O_{tB}), say $f_{r_Ar_B}$, $f_{r_As_B}$, $f_{r_At_B}$, $f_{s_Ar_B}$, $f_{s_As_B}$, $f_{s_At_B}$, $f_{t_Ar_B}$, $f_{t_As_B}$, $f_{t_At_B}$, respectively, the largest two of the following sums must be equal:

$f_{r_Ar_B} + f_{s_As_B} + f_{t_At_B}$;
$f_{r_Ar_B} + f_{s_At_B} + f_{t_As_B}$;
$f_{r_As_B} + f_{s_Ar_B} + f_{t_At_B}$;
$f_{r_As_B} + f_{s_At_B} + f_{t_Ar_B}$;
$f_{r_At_B} + f_{s_Ar_B} + f_{t_As_B}$;
$f_{r_At_B} + f_{s_As_B} + f_{t_Ar_B}$.

In both cases of ultrametric and additive trees for two-mode proximity data, the necessary constraints characterizing a solution are linear and define closed convex sets in which a solution must lie. Thus, the application of iterative projection as a heuristic search strategy for the best-fitting solutions is fairly direct, and an example of an ultrametric found and fitted to a two-mode matrix will be given in Section 4.1. We will not, however, give a comparable example of fitting the additive tree constraints to such a proximity matrix; the (scratch) storage requirements necessitated by iterative projection in directly using the additive tree constraints given above and keeping track of the various augmentations made in the course of the heuristic search can become rather onerous for moderate-sized data matrices. For general use, an alternative approach to the fitting of additive trees is preferable that again uses iterative projection but with the ultrametric conditions in conjunction with a secondary centroid metric; this strategy avoids any major (scratch) storage difficulties and will be reviewed and illustrated in Section 4.2.

We might note that the process of fitting two-mode proximity data by additive trees or ultrametrics using iterative projection heuristics may generate a rather large number of distinct locally optimal solutions, particularly in contrast to the situation usually observed for symmetric proximity data. Although this abundance is not inevitably the case and obviously depends on the particular data set being considered, it is not unusual and should be expected by a user.

4.1 Fitting and Finding Two-Mode Ultrametrics

To illustrate the fitting of a given two-mode ultrametric, a two-mode target is generated by the upper-right 6×4 portion of the 10×10 ultrametric target matrix, `numcltarg`, used in Section 1.1. This file will be called `numcltarg6x4.dat`, and has contents as follows:

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

The six rows correspond to the digits 0, 1, 2, 3, 4, and 5; the four columns to 6, 7, 8, and 9. As the two-mode 6×4 proximity matrix, the appropriate upper-right portion of the `number` proximity matrix will be used in the fitting process; the corresponding file is called `number6x4.dat`, with contents:

```
.788 .909 .821 .850
.758 .630 .791 .625
.421 .796 .367 .808
.300 .592 .804 .263
.388 .742 .246 .683
.396 .400 .671 .592
```

The m-file, `ultrafittm.m`, of Appendix A.11 fits a given ultrametric to a two-mode proximity matrix (using iterative projection in the L_2-norm); it has usage

```
[fit,vaf] = ultrafittm(proxtm,targ)
```

where `PROXTM` is the two-mode (rectangular) input proximity matrix (with a dissimilarity interpretation); `TARG` is an ultrametric matrix of the same size as `PROXTM`; `FIT` is the least-squares optimal matrix (with variance-accounted-for of `VAF`) to `PROXTM` satisfying the two-mode ultrametric constraints implicit in `TARG`. An example follows using `numcltarg6x4` for `TARG` and `number6x4` as `proxtm`:

```
load number6x4.dat
number6x4

number6x4 =

    0.7880    0.9090    0.8210    0.8500
    0.7580    0.6300    0.7910    0.6250
    0.4210    0.7960    0.3670    0.8080
    0.3000    0.5920    0.8040    0.2630
    0.3880    0.7420    0.2460    0.6830
    0.3960    0.4000    0.6710    0.5920

load numcltarg6x4.dat
numcltarg6x4

numcltarg6x4 =

     9     9     9     9
     9     9     9     9
     8     8     4     8
     3     7     8     2
     8     8     4     8
     7     5     8     7


[fit,vaf] = ultrafittm(number6x4,numcltarg6x4)

fit =

    0.7715    0.7715    0.7715    0.7715
    0.7715    0.7715    0.7715    0.7715
    0.6641    0.6641    0.3065    0.6641
    0.3000    0.5267    0.6641    0.2630
    0.6641    0.6641    0.3065    0.6641
    0.5267    0.4000    0.6641    0.5267


vaf =

    0.6978
```

A vaf of .6978 was obtained for the fitted ultrametric; we give the hierarchy below with indications of when the partitions were formed in the L_2-norm fitted ultrametric (in `FIT`) and in the original target (in `cltarg6X4`):

Partition	Level Formed (L_2)	Level Formed (Target)
{{0,1,2,4,8,3,9,6,5,7}}	.7715	9
{{0},{1},{2,4,8,3,9,6,5,7}}	.6641	8
{{0},{1},{2,4,8},{3,9,6,5,7}}	.5267	7
{{0},{1},{2,4,8},{3,9,6},{5,7}}	.4000	5
{{0},{1},{2,4,8},{3,9,6},{5},{7}}	.3065	4
{{0},{1},{2},{4},{8},{3,9,6},{5},{7}}	.3000	3
{{0},{1},{2},{4},{8},{3,9},{6},{5},{7}}	.2630	2
{{0},{1},{2},{4},{8},{3},{9},{6},{5},{7}}	—	—

The m-file, `ultrafndtm.m` in Appendix A.12 relies on iterative projection heuristically to locate a best-fitting two-mode ultrametric. The usage is

```
[find,vaf] = ultrafndtm(proxtm,inpermrow,inpermcol)
```

where `PROXTM` is the two-mode input proximity matrix (with a dissimilarity interpretation); `INPERMROW` and `INPERMCOL` are permutations for the row and column objects that determine the order in which the inequality constraints are considered; `FIND` is the found least-squares matrix (with variance-accounted-for of `VAF`) to `PROXTM` satisfying the ultrametric constraints. The example below for the `number6x4` two-mode data (using random permutations for `INPERMROW` and `INPERMCOL`), finds an ultrametric with vaf of .7448.

```
[find,vaf] = ultrafndtm(number6x4,randperm(6),randperm(4))

find =

    0.8420    0.8420    0.8420    0.8420
    0.7010    0.7010    0.7010    0.7010
    0.6641    0.6641    0.3670    0.6641
    0.3000    0.5267    0.6641    0.2630
    0.6641    0.6641    0.2460    0.6641
    0.5267    0.4000    0.6641    0.5267


vaf =

    0.7448
```

The partition hierarchy identified is similar to that found for the fixed target `numcltarg6x4`, although there is some minor variation in how the digits 0 and 1 are treated:

Partition	Level Formed (L_2)
{{0,1,2,4,8,3,9,6,5,7}}	.8420
{{0},{1,2,4,8,3,9,6,5,7}}	.7010
{{0},{1},{2,4,8,3,9,6,5,7}}	.6641
{{0},{1},{2,4,8},{3,9,6,5,7}}	.5267
{{0},{1},{2,4,8},{3,9,6},{5,7}}	.4000
{{0},{1},{2,4,8},{3,9,6},{5},{7}}	.3670
{{0},{1},{2},{4,8},{3,9,6},{5},{7}}	.3000
{{0},{1},{2},{4,8},{3,9},{6},{5},{7}}	.2630
{{0},{1},{2},{4,8},{3},{9},{6},{5},{7}}	.2460
{{0},{1},{2},{4},{8},{3},{9},{6},{5},{7}}	—

4.2 Finding Two-Mode Additive Trees

As noted in the introductory material to the current Chapter 4, the identification of a best-fitting two-mode additive tree will be done somewhat differently (because of storage considerations) than for a two-mode ultrametric representation. Specifically, a (two-mode) centroid metric and a (two-mode) ultrametric matrix will be identified so that their sum is a good-fitting two-mode additive tree. Because a centroid metric can be obtained in closed-form, we first illustrate the fitting of just a centroid metric to a two-mode proximity matrix with the m-file, `centfittm.m`, in Appendix A.13. Its usage is of the form

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

which gives the least-squares fitted two-mode centroid metric (`FIT`) to `PROXTM`, the two-mode rectangular input proximity matrix (with a dissimilarity interpretation). The n values (where n = number of rows(n_a) + number of columns(n_b)) serve to define the approximating sums, $u_r + v_s$, where the u_r are for the n_a rows and the v_s for the n_b columns; these are given in the vector `LENGTHS` of size $n \times 1$, with row values first followed by the column values. The closed-form formula used for u_r (or v_s) can be given simply as the r^{th} row (or s^{th} column) mean of `PROXTM` minus one-half the grand mean (see Carroll & Pruzansky, 1980, and De Soete et al., 1984, for a further discussion). In the example given below using the two-mode matrix, `number6x4`, a two-mode centroid metric by itself has a vaf of .4737.

```
load number6x4.dat
[fit,vaf,lengths] = centfittm(number6x4)

fit =

    0.7405    0.9101    0.8486    0.8688
    0.5995    0.7691    0.7076    0.7278
    0.4965    0.6661    0.6046    0.6248
    0.3882    0.5579    0.4964    0.5165
    0.4132    0.5829    0.5214    0.5415
    0.4132    0.5829    0.5214    0.5415


vaf =

    0.4737


lengths =

    0.5370
    0.3960
    0.2930
    0.1847
    0.2097
    0.2097
    0.2035
    0.3731
    0.3116
    0.3318
```

The finding of a two-mode additive tree with the m-file, `atreefndtm.m`, in Appendix A.14 proceeds iteratively. A two-mode centroid metric is first found and the original two-mode proximity matrix residualized; a two-mode ultrametric is then identified for the residual matrix. The process repeats with the centroid and ultrametric components alternatingly being refit until a small change in the overall vaf occurs (a value less than 1.0e-006 is used). The m-file has the explicit usage

```
[find,vaf,ultrafit,lengths] = atreefndtm(proxtm,inpermrow,inpermcol)
```

and as noted above, relies on iterative projection heuristically to find a two-mode ultrametric component that is added to a two-mode centroid metric to produce a two-mode additive tree. Here, `PROXTM` is the rectangular input proximity matrix (with a dissimilarity interpretation); `INPERMROW` and `INPERMCOL` are permutations for the row and column objects that determine the order in which the inequality constraints are considered; `FIND` is the found least-squares matrix (with variance-accounted-for of `VAF`) to `PROXTM` satisfying the two-mode additive tree constraints. The vector `LENGTHS` contains the row followed by column values for the two-mode centroid metric component; `ULTRA` is the ultrametric component. In the example given below, the identified two-mode additive-tree for `number6x4` has a vaf of .9053, with a nice structural interpretation of the digits along with some indication now of odd and even digit groupings. The partition hierarchy is reported below the MATLAB output along with an indication of when the various partitions are formed.

```
[find,vaf,ultra,lengths] = atreefndtm(number6x4,randperm(6),randperm(4))

find =

    0.6992    0.9029    0.9104    0.8561
    0.6298    0.6300    0.8411    0.7029
    0.4398    0.8160    0.3670    0.7692
    0.4549    0.5748    0.6661    0.2630
    0.3692    0.7453    0.2460    0.6985
    0.4582    0.4000    0.6694    0.5313


vaf =

    0.9053
```

```
ultra =

    0.1083    0.0520    0.1083    0.0520
    0.1083   -0.1516    0.1083   -0.0318
   -0.0078    0.1083   -0.2919    0.1083
    0.1083   -0.0318    0.1083   -0.2968
   -0.0078    0.1083   -0.3422    0.1083
    0.1083   -0.2099    0.1083   -0.0318


lengths =

    0.4570
    0.3876
    0.3138
    0.2127
    0.2431
    0.2160
    0.1339
    0.3939
    0.3451
    0.3471
```

Partition	Level Formed
{{6,4,8,2,9,3,5,7,1,0}}	.1083
{{6,4,8,2},{9,3,5,7,1,0}}	.0520
{{6,4,8,2},{9,3,5,7,1},{0}}	-.0078
{{6},{4,8,2},{9,3,5,7,1},{0}}	-.0318
{{6},{4,8,2},{9,3},{5,7,1},{0}}	-.1516
{{6},{4,8,2},{9,3},{5,7},{1},{0}}	-.2099
{{6},{4,8,2},{9,3},{5},{7},{1},{0}}	-.2919
{{6},{4,8},{2},{9,3},{5},{7},{1},{0}}	-.2968
{{6},{4,8},{2},{9},{3},{5},{7},{1},{0}}	-.3422
{{6},{4},{8},{2},{9},{3},{5},{7},{1},{0}}	—

4.3 Reordering a Two-Mode Rectangular Proximity Matrix (with a Quadratic Assignment Strategy)

Given a $n_a \times n_b$ two-mode proximity matrix, $\mathbf{Q}$, defined between the two distinct sets S_A and S_B, it may be desirable at times to have some means for separately reordering 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 LUS Toolbox 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 $\mathbf{P}^{(tm)}$.) 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`, in Appendix A.15 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 to be 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`: 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 beginning input permutation (a permutation of the first n integers). `PROXTM` is the two-mode $n_a \times n_b$ input proximity matrix. `TARG` is the $n \times n$ 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`.

In the example to follow, `ordertm.m`, is used (analogously to how `ultraorder.m` was applied to a symmetric proximity matrix) to identify the anti-Robinson pattern for the ultrametric component (called `ultra`) of the two-mode additive tree of the last section (4.2). The square equally-spaced target matrix is obtained from the LUS utility, `ransymat.m`.

The listing given of the (reordered) matrix, `squareprox(outperm,outperm)`, shows clearly the anti-Robinson pattern for the ultrametric component of the additive tree (and is the permutation used in listing the partition hierarchy of the last section):

```
load number6x4.dat
[find,vaf,ultra,lengths] = atreefndtm(number6x4,randperm(6),randperm(4));

ultra

ultra =

    0.1083    0.0520    0.1083    0.0520
    0.1083   -0.1516    0.1083   -0.0318
   -0.0078    0.1083   -0.2919    0.1083
    0.1083   -0.0318    0.1083   -0.2968
   -0.0078    0.1083   -0.3422    0.1083
    0.1083   -0.2099    0.1083   -0.0318

[prox10,targlin,targcir] = ransymat(10);

targlin

targlin =

     0     1     2     3     4     5     6     7     8     9
     1     0     1     2     3     4     5     6     7     8
     2     1     0     1     2     3     4     5     6     7
     3     2     1     0     1     2     3     4     5     6
     4     3     2     1     0     1     2     3     4     5
     5     4     3     2     1     0     1     2     3     4
     6     5     4     3     2     1     0     1     2     3
     7     6     5     4     3     2     1     0     1     2
     8     7     6     5     4     3     2     1     0     1
     9     8     7     6     5     4     3     2     1     0

[outperm,rawindex,allperms,index,squareprox] = ...
ordertm(ultra,targlin,1:10,1)

outperm =

     7     5     9     3    10     4     6     8     2     1
```

```
rawindex =

   11.0000


allperms =

  Columns 1 through 4

    [1x10 double]    [1x10 double]    [1x10 double]    [1x10 double]

  Columns 5 through 8

    [1x10 double]    [1x10 double]    [1x10 double]    [1x10 double]

  Columns 9 through 12

    [1x10 double]    [1x10 double]    [1x10 double]    [1x10 double]

  Columns 13 through 16

    [1x10 double]    [1x10 double]    [1x10 double]    [1x10 double]

    [1x10 double]


index =

    17


squareprox =

  Columns 1 through 7

         0         0         0         0         0         0    0.1083
         0         0         0         0         0         0    0.1083
         0         0         0         0         0         0   -0.0078
         0         0         0         0         0         0    0.1083
         0         0         0         0         0         0   -0.0078
         0         0         0         0         0         0    0.1083
    0.1083    0.1083   -0.0078    0.1083   -0.0078    0.1083         0
    0.0520   -0.1516    0.1083   -0.0318    0.1083   -0.2099         0
    0.1083    0.1083   -0.2919    0.1083   -0.3422    0.1083         0
    0.0520   -0.0318    0.1083   -0.2968    0.1083   -0.0318         0

  Columns 8 through 10

    0.0520    0.1083    0.0520
   -0.1516    0.1083   -0.0318
    0.1083   -0.2919    0.1083
   -0.0318    0.1083   -0.2968
    0.1083   -0.3422    0.1083
   -0.2099    0.1083   -0.0318
         0         0         0
         0         0         0
         0         0         0
         0         0         0

squareprox(outperm,outperm)

ans =

  Columns 1 through 7

         0   -0.0078         0   -0.0078         0    0.1083    0.1083
   -0.0078         0   -0.3422         0    0.1083         0         0
         0   -0.3422         0   -0.2919         0    0.1083    0.1083
   -0.0078         0   -0.2919         0    0.1083         0         0
         0    0.1083         0    0.1083         0   -0.2968   -0.0318
    0.1083         0    0.1083         0   -0.2968         0         0
    0.1083         0    0.1083         0   -0.0318         0         0
         0    0.1083         0    0.1083         0   -0.0318   -0.2099
    0.1083         0    0.1083         0   -0.0318         0         0
    0.1083         0    0.1083         0    0.0520         0         0

  Columns 8 through 10

         0    0.1083    0.1083
    0.1083         0         0
         0    0.1083    0.1083
    0.1083         0         0
         0   -0.0318    0.0520
   -0.0318         0         0
   -0.2099         0         0
         0   -0.1516    0.0520
   -0.1516         0         0
    0.0520         0         0
```

4.4 Fitting a Unidimensional Scale to Two-Mode Proximity Data (Based on a Given Permutation of the Combined Row and Column Object Set)

Besides finding best-fitting two-mode ultrametrics and additive trees, it is possible to fit through iterative projection, best-fitting (in the L_2-norm) 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, $\mathbf{P}_{\rho_0}^{(tm)} = \{p_{\rho_0(i)\rho_0(j)}^{(tm)}\}$, the least-squares criterion

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

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 $\mathbf{P}_{\rho_0}^{(tm)}$ 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 (in)equality constraints generated from how the row and column objects are intermixed by the given permutation $\rho_0(\cdot)$. To give a schematic 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 fit 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 in terms of 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`, of Appendix A.16, with usage

```
[fit,diff,rowperm,colperm] = 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 a $n_a \times n_b$ 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,COLMEAN)`; `ROWPERM` and `COLPERM` are the row and column object orderings derived from `INPERM`.

There are two examples given below. The first uses a permutation obtained from `ordertm.m` on `number6x4`; the second is the identity permutation. The particular unidimensional scalings given by the spacings between the ten objects follow the MATLAB output.

```
load number6x4.dat
[prox10,targlin,targcir] = ransymat(10);
[outperm,rawindex,allperms,index,squareprox] = ...
ordertm(number6x4,targlin,1:10,1)

outperm =

     1     2    10     4     8     6     7     5     3     9


rawindex =

   15.3192


allperms =

  Columns 1 through 4

    [1x10 double]    [1x10 double]    [1x10 double]    [1x10 double]

  Columns 5 through 8

    [1x10 double]    [1x10 double]    [1x10 double]    [1x10 double]
```

```
index =

     8


squareprox =

  Columns 1 through 7

         0         0         0         0         0         0    0.1780
         0         0         0         0         0         0    0.1480
         0         0         0         0         0         0   -0.1890
         0         0         0         0         0         0   -0.3100
         0         0         0         0         0         0   -0.2220
         0         0         0         0         0         0   -0.2140
    0.1780    0.1480   -0.1890   -0.3100   -0.2220   -0.2140         0
    0.2990    0.0200    0.1860   -0.0180    0.1320   -0.2100         0
    0.2110    0.1810   -0.2430    0.1940   -0.3640    0.0610         0
    0.2400    0.0150    0.1980   -0.3470    0.0730   -0.0180         0

  Columns 8 through 10

    0.2990    0.2110    0.2400
    0.0200    0.1810    0.0150
    0.1860   -0.2430    0.1980
   -0.0180    0.1940   -0.3470
    0.1320   -0.3640    0.0730
   -0.2100    0.0610   -0.0180
         0         0         0
         0         0         0
         0         0         0
         0         0         0

[fit,diff,rowperm,colperm] = linfittm(number6x4,outperm)

fit =

    0.4975    0.6841    0.8907    1.2956
    0.3565    0.5431    0.7497    1.1546
    0.0000    0.1865    0.3932    0.7981
    0.3257    0.1392    0.0675    0.4724
    0.7195    0.5330    0.3263    0.0785
    0.7350    0.5485    0.3418    0.0631


diff =

    1.3933


rowperm =

     1
     2
     4
     6
     5
     3


colperm =
     4
     2
     1
     3

[fit,diff,rowperm,colperm] = linfittm(number6x4,1:10)

fit =

    0.7405    0.8758    0.8758    0.8758
    0.5995    0.7348    0.7348    0.7348
    0.4965    0.6318    0.6318    0.6318
    0.4049    0.5403    0.5403    0.5403
    0.4049    0.5403    0.5403    0.5403
    0.4049    0.5403    0.5403    0.5403


diff =

    0.5398


rowperm =
     1
     2
     3
     4
     5
     6


colperm =

     1
     2
     3
     4
```

Figure 4.1: Unidimensional Scaling (Using linfittm.m) of the Ten Digits Based on the Permutation [1 2 10 4 8 6 7 5 3 9] and the 6 × 4 Two-Mode Proximity Matrix; the Spacings Between Locations are Indicated Above the Line

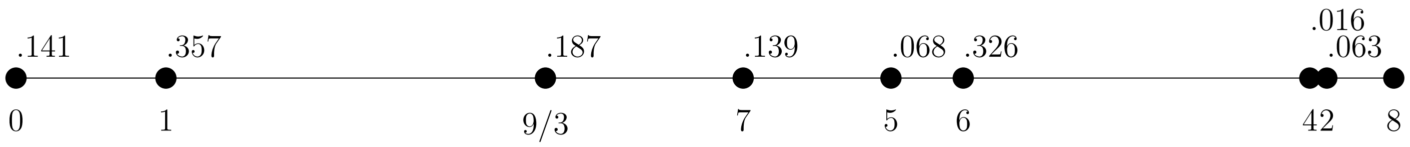
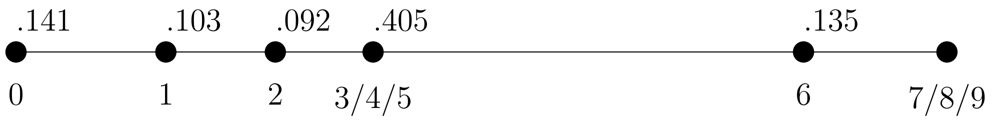


Figure 4.2: Unidimensional Scaling (Using linfittm.m) of the Ten Digits Based on the Identity Permutation [1 2 3 4 5 6 7 8 9 10] and the 6 × 4 Two-Mode Proximity Matrix; the Spacings Between Locations are Indicated Above the Line



In complete analogy to what was done in the LUS Toolbox (with the m-file `linfitac.m` generalizing `linfit.m` by fitting an additive constant along with the absolute coordinate differences), the more general unidimensional scaling model can be fit with an additive constant using the m-file of Appendix A.17, `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] = 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 Dykstra's (Kaczmarz's) iterative projection least-squares method. It differs from `linfittm.m` by including the estimation of an additive constant, and thus allowing the variance-accounted-for (`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 two exemplar permutations are used below, and following the MATLAB output, the schematics of the unidimensional scalings are given with the additive constant and variance-accounted-for specified.

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

inperm =

     1     2    10     4     8     6     7     5     3     9

[fit,vaf,rowperm,colperm,addcon] = linfittmac(number6x4,inperm)

fit =

    0.3267    0.3994    0.4801    0.6499
    0.1857    0.2584    0.3391    0.5089
         0    0.0727    0.1533    0.3232
    0.1185    0.0459    0.0348    0.2047
    0.3138    0.2411    0.1605    0.0094
    0.3230    0.2504    0.1697    0.0002


vaf =

    0.6706


rowperm =

     1
     2
     4
     6
     5
     3


colperm =

     4
     2
     1
     3


addcon =

   -0.3780

[fit,vaf,rowperm,colperm,addcon] = linfittmac(number6x4,1:10)

fit =

    0.7405    0.8758    0.8758    0.8758
    0.5995    0.7348    0.7348    0.7348
    0.4965    0.6318    0.6318    0.6318
    0.4049    0.5403    0.5403    0.5403
    0.4049    0.5403    0.5403    0.5403
    0.4049    0.5403    0.5403    0.5403
```

```
vaf =

    0.4602


rowperm =

     1
     2
     3
     4
     5
     6


colperm =

     1
     2
     3
     4


addcon =

 -2.7756e-017
```

Figure 4.3: Unidimensional Scaling (Using linfittmac.m) of the Ten Digits Based on the Permutation [1 2 10 4 8 6 7 5 3 9] and the 6 $\times$ 4 Two-Mode Proximity Matrix; the Spacings Between Locations are Indicated Above the Line. The Variance-Accounted-For is .6706 Using an Additive Constant of -.3780

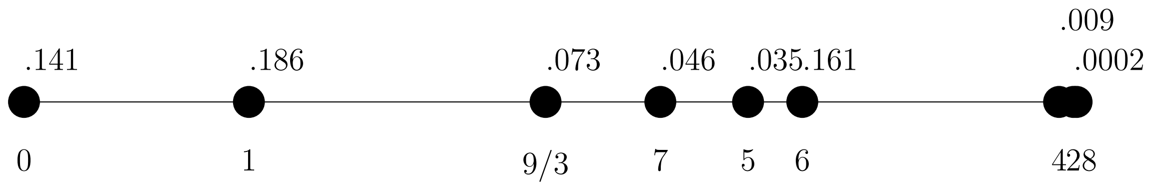
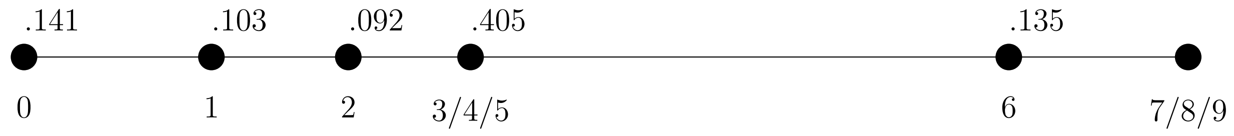


Figure 4.4: Unidimensional Scaling (Using linfittmac.m) of the Ten Digits Based on the Identity Permutation [1 2 3 4 5 6 7 8 9 10] and the 6 $\times$ 4 Two-Mode Proximity Matrix; the Spacings Between Locations are Indicated Above the Line. The Variance-Accounted-For is .4602 Using an Additive Constant of 0.0 — thus, the Figure is Identical to 4.2



4.5 Some Useful Utilities for Transforming Two-Mode Proximity Matrices

This section gives several miscellaneous m-functions that carry out various operations on a two-mode proximity matrix, and for which no other section seemed appropriate. The first two, `proxstdtm.m` and `proxrandtm.m`, given in Appendix A.18 and A.19, are very simple and provide standardized and randomly (entry-)permuted two-mode proximity matrices, respectively, that might be useful, for example, in testing the various m-functions we give. The syntax

```
[stanproxtm stanproxmulttm] = proxstd(proxtm,mean)
```

is intended to suggest that `STANPROXTM` provides a linear transformation of the entries in `PROXTM` to a standard deviation of one and a mean of `MEAN`; `STANPROXMULTTM` is a multiplicative transformation so the entries in this $n_a \times n_b$ matrix have a sum-of-squares of $n_a n_b$. For the second utility m-function

```
[randproxtm] = proxrandtm(proxtm)
```

implies that the two-mode matrix `RANDPROXTM` has its entries as a random permutation of the entries in `PROXTM`. The illustration below using `number6x4.dat` should make both of these usages clear.

```
load number6x4.dat
[stanproxtm, stanproxmulttm] = proxstdtm(number6x4,5.0)

stanproxtm =

    5.8718    6.4646    6.0335    6.1755
    5.7248    5.0978    5.8865    5.0733
    4.0739    5.9110    3.8093    5.9698
    3.4811    4.9116    5.9502    3.2999
    3.9122    5.6465    3.2166    5.3574
    3.9514    3.9710    5.2986    4.9116


stanproxmulttm =

    6.0010    6.9225    6.2524    6.4732
    5.7726    4.7978    6.0239    4.7597
    3.2061    6.0620    2.7949    6.1534
    2.2847    4.5084    6.1229    2.0029
    2.9548    5.6507    1.8734    5.2014
    3.0158    3.0462    5.1100    4.5084

[randproxtm] = proxrandtm(number6x4)

randproxtm =

    0.3000    0.9090    0.8210    0.4000
    0.7580    0.6300    0.8500    0.6250
    0.4210    0.7910    0.3670    0.6710
    0.7960    0.5920    0.8080    0.7880
    0.3880    0.8040    0.2630    0.5920
    0.7420    0.2460    0.6830    0.3960
```

A third utility function, `proxmontm.m`, given in Appendix A.20, provides a monotonically transformed two-mode proximity matrix that is close in a least-squares sense to a given input two-mode matrix. The syntax is

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

Here, `PROXPERMUTTM` is the original input two-mode proximity matrix (which may have been subjected to initial row and column permutations, hence the suffix '`PERMUTTM`'), and `FITTEDTM` is a given two-mode target matrix; the output matrix `MONPROXPERMUTTM` is closest to `FITTEDTM` in a least-squares sense and obeys the order constraints obtained from each pair of entries in `PROXPERMUTTM` (and where the inequality constrained optimization is carried out using the Dykstra-Kaczmarz iterative projection strategy); as usual, `VAF` denotes 'variance-accounted-for' and indicates how much variance in `MONPROXPERMUTTM` can be accounted for by `FITTEDTM`; finally `DIFF` is the value of the least-squares loss function and is the sum of squared differences between the entries in `MONPROXPERMUTTM` and `FITTEDTM`.

The script m-file listed below gives an application of `proxmontm.m` using the permutation found previously with `ordertm.m` for our `number6x4` matrix. First, `linfittm.m` is invoked to obtain a fitted matrix (`fit`); `proxmontm.m` then generates the monotonically transformed proximity matrix (`monproxpermuttm`) with `vaf` = .6599 and `diff` = .6751. The strategy is then repeated cyclically (i.e., finding a fitted matrix based on the monotonically transformed proximity matrix, finding a new monotonically tranformed matrix, and so on). To avoid degeneracy (where all matrices would converge to zeros), the sum of squares of the fitted matrix is kept the same as it was initially; convergence is based on observing a minimal change (less than 1.0e-006) in the `vaf`. As indicated in the output below, the final vaf is .7771.

```
load number6x4.dat
inperm = [1 2 10 4 8 6 7 5 3 9]
[fit,diff,rowperm,colperm] = linfittm(number6x4,inperm)
[monproxpermuttm, vaf, diff] = proxmontm(number6x4(rowperm,colperm), fit)
```

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


while (abs(prevvaf-vaf) >= 1.0e-006)

   prevvaf = vaf;
   mondata(rowperm,colperm) = monproxpermuttm;
   [fit diff rowperm colperm] = linfittm(mondata,inperm);
   sumnewfitsq = sum(sum(fit.^2));
   fit = sqrt(sumfitsq)*(fit/sqrt(sumnewfitsq));

   [monproxpermuttm vaf diff] = proxmontm(number6x4(rowperm,colperm), fit);


end

fit
vaf
monproxpermuttm
mondata
inperm =

     1     2    10     4     8     6     7     5     3     9



fit =

    0.4975    0.6841    0.8907    1.2956
    0.3565    0.5431    0.7497    1.1546
    0.0000    0.1865    0.3932    0.7981
    0.3257    0.1392    0.0675    0.4724
    0.7195    0.5330    0.3263    0.0785
    0.7350    0.5485    0.3418    0.0631


diff =

    1.3933


rowperm =
     1
     2
     4
     6
     5
     3


colperm =

     4
     2
     1
     3


monproxpermuttm =

    0.8258    0.8258    0.8254    0.8258
    0.3565    0.5077    0.7497    0.8254
    0.0393    0.2642    0.1979    0.8254
    0.3257    0.1979    0.1979    0.5077
    0.6263    0.6263    0.1979    0.0393
    0.8254    0.8254    0.2642    0.1979


vaf =

    0.6599


diff =

    0.6751


fit =

    0.7513    0.8413    1.0727    1.4538
    0.2422    0.3322    0.5636    0.9447
         0    0.0900    0.3214    0.7025
    0.2041    0.1141    0.1173    0.4984
    0.6177    0.5277    0.2963    0.0848
    0.6890    0.5990    0.3676    0.0135


vaf =

    0.7771


monproxpermuttm =

    1.0154    1.0154    0.8016    1.0154
    0.2422    0.3322    0.5696    0.8016
    0.0424    0.2206    0.1674    0.8016
    0.2206    0.1759    0.1759    0.4984
    0.5696    0.5696    0.1759    0.0424
    0.8016    0.8016    0.2206    0.1674


mondata =

    0.8016    1.0154    1.0154    1.0154
    0.5696    0.3322    0.8016    0.2422
    0.2206    0.8016    0.1674    0.8016
    0.1674    0.2206    0.8016    0.0424
    0.1759    0.5696    0.0424    0.5696
    0.1759    0.1759    0.4984    0.2206
```

Bibliography

[1] Barthélemy, J.-P. & Guénouche, A. (1991). *Trees and proximity representations.* Chichester: Wiley.

[2] Brossier, G. (1987). Étude des matrices de proximité rectangulaires en vue de la classification [A study of rectangular proximity matrices from the point of view of classification]. *Revue de Statistiques Appliquées*, *35(4)*, 43–68.

[3] Carroll, J. D. (1976). Spatial, non-spatial and hybrid models for scaling. *Psychometrika*, *41*, 439–463.

[4] Carroll, J. D., Clark, L. A., & DeSarbo, W. S. (1984). The representation of three-way proximity data by single and multiple tree structure models. *Journal of Classification*, *1*, 25–75.

[5] Carroll, J. D. & Pruzansky, S. (1980). Discrete and hybrid scaling models. In E. D. Lantermann & H. Feger (Eds.), *Similarity and choice* (pp. 108–139). Bern: Hans Huber.

[6] Day, W. H. E. (1987). Computational complexity of inferring phylogenies from dissimilarity matrices. *Bulletin of Mathematical Biology*, *49*, 461–467.

[7] De Soete, G. (1983). A least squares algorithm for fitting additive trees to proximity data. *Psychometrika*, *48*, 621–626.

[8] De Soete, G. (1984a). A least squares algorithm for fitting an ultrametric tree to a dissimilarity matrix. *Pattern Recognition Letters*, *2*, 133–137.

[9] De Soete, G. (1984b). Ultrametric tree representations of incomplete dissimilarity data. *Journal of Classification*, *1*, 235–242.

[10] De Soete, G. (1984c). Additive tree representations of incomplete dissimilarity data. *Quality and Quantity*, *18*, 387–393.

[11] De Soete, G., Carroll, J. D., & DeSarbo, W. S. (1987). Least squares algorithms for constructing constrained ultrametric and additive tree representations of symmetric proximity data. *Journal of Classification*, *4*, 155–173.

[12] De Soete, G., DeSarbo, W. S., Furnas, G. W., & Carroll, J. D. (1984). The estimation of ultrametric and path length trees from rectangular proximity data. *Psychometrika*, *49*, 289–310.

[13] Furnas, G. W. (1980). Objects and their features: The metric representation of two class data. Unpublished doctoral dissertation, Stanford University.

[14] Hubert, L. J., & Arabie, P. (1994). The analysis of proximity matrices through sums of matrices having (anti-)Robinson forms. *British Journal of Mathematical and Statistical Psychology*, *47*, 1–40.

[15] Hubert, L. J., & Arabie, P. (1995). Iterative projection strategies for the least-squares fitting of tree structures to proximity data. *British Journal of Mathematical and Statistical Psychology*, *48*, 281–317.

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

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

[18] Hutchinson, J. W. (1989). NETSCAL: A network scaling algorithm for nonsymmetric proximity data. *Psychometrika*, *54*, 25–51.

[19] Klauer, K. C., & Carroll, J. D. (1989). A mathematical programming approach to fitting general graphs. *Journal of Classification*, *6*, 247–270.

[20] Klauer, K. C., & Carroll, J. D. (1991). A comparison of two approaches to fitting directed graphs to nonsymmetric proximity measures. *Journal of Classification*, *8*, 251–268.

[21] Krivánek, M. (1986). On the computational complexity of clustering. In E. Diday, Y. Escoufier, L. Lebart, J. P. Pagès, Y. Schektman, & R. Tomassone (Eds.), *Data analysis and informatics, IV* (pp. 89–96). Amsterdam: North-Holland.

[22] Krivànek, M., & Moravek, J. (1986). NP-hard problems in hierarchical-tree clustering. *Acta Informatica*, *23*, 311–323.

[23] Mirkin, B. (1996). *Mathematical classification and clustering.* Dordrecht: Kluwer.

[24] Späth, H. (1991). *Mathematical algorithms for linear regression.* New York: Academic Press.

Appendix A

main program files

A.1 ultrafit.m

```
function [fit,vaf] = ultrafit(prox,targ)

% ULTRAFIT fits a given ultrametric using iterative projection to
% a symmetric proximity matrix in the $L_{2}$-norm.
% 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.

n = size(prox,1);
work = zeros(n*(n-1)*n*(n-1),1);
fit = prox;
cr = 1.0;

while (cr >= 1.0e-006)
   cr = 0.0;
   indexll = 0;

   for jone = 1:(n-2)
      for jtwo = (jone+1):(n-1)
         for jthree = (jtwo+1):n

            p1 = fit(jone,jtwo);
            p2 = fit(jone,jthree);
            p3 = fit(jtwo,jthree);

            fit(jone,jtwo) = fit(jone,jtwo) - work(indexll+1);
            fit(jone,jthree) = fit(jone,jthree) - work(indexll+2);
            fit(jtwo,jthree) = fit(jtwo,jthree) - work(indexll+3);

            if((targ(jone,jtwo) <= targ(jone,jthree)) & ...
                  (targ(jone,jtwo) <= targ(jtwo,jthree)))
               ave = (fit(jone,jthree) + fit(jtwo,jthree))/2.0;

               work(indexll+1) = 0.0;
               work(indexll+2) = ave - fit(jone,jthree);
               work(indexll+3) = ave - fit(jtwo,jthree);

               fit(jone,jthree) = ave;
               fit(jtwo,jthree) = ave;

               indexll = indexll + 3;

             elseif((targ(jone,jthree) <= targ(jone,jtwo)) & ...
                  (targ(jone,jthree) <= targ(jtwo,jthree)))

               ave = (fit(jone,jtwo) + fit(jtwo,jthree))/2.0;

               work(indexll+1) = ave - fit(jone,jtwo);
               work(indexll+2) = 0.0;
               work(indexll+3) = ave - fit(jtwo,jthree);

               fit(jone,jtwo) = ave;
               fit(jtwo,jthree) = ave;

               indexll = indexll + 3;

             elseif((targ(jtwo,jthree) <= targ(jone,jthree)) & ...
                  (targ(jtwo,jthree) <= targ(jone,jtwo)))

               ave = (fit(jone,jthree) + fit(jone,jtwo))/2.0;

               work(indexll+1) = ave - fit(jone,jtwo);
               work(indexll+2) = ave - fit(jone,jthree);
               work(indexll+3) = 0.0;

               fit(jone,jtwo) = ave;
               fit(jone,jthree) = ave;

               indexll = indexll + 3;
            end

               cr = cr + abs(p1-fit(jone,jtwo)) + abs(p2-fit(jone,jthree)) ...
               + abs(p3-fit(jtwo,jthree));

         end
      end
   end

   for jone = 1:(n-1)
      for jtwo = (jone+1):n
         for jthree = 1:(n-1)
   for jfour = (jthree+1):n

      if((jone ~= jthree) | (jtwo ~= jfour))

         p1 = fit(jone,jtwo);
         p2 = fit(jthree,jfour);
         fit(jone,jtwo) = fit(jone,jtwo) - work(indexll+1);
         fit(jthree,jfour) = fit(jthree,jfour) - work(indexll+2);

         if(abs(targ(jone,jtwo) - targ(jthree,jfour)) <= ...
               1.0e-006)

            ave = (fit(jone,jtwo) + fit(jthree,jfour))/2.0;
            work(indexll+1) = ave - fit(jone,jtwo);
            work(indexll+2) = ave - fit(jthree,jfour);
            fit(jone,jtwo) = ave;
            fit(jthree,jfour) = ave;

            indexll = indexll + 2;

         elseif((abs(targ(jone,jtwo) - targ(jthree,jfour)) ...
               > 1.0e-006) & (targ(jone,jtwo) < ...
               targ(jthree,jfour)))

            if(fit(jone,jtwo) < fit(jthree,jfour))

               work(indexll+1) = 0;
               work(indexll+2) = 0;

               indexll = indexll + 2;

            elseif(fit(jone,jtwo) >= fit(jthree,jfour))

               ave = (fit(jone,jtwo) + fit(jthree,jfour))/2.0;
               work(indexll+1) = ave - fit(jone,jtwo);
               work(indexll+2) = ave - fit(jthree,jfour);
               fit(jone,jtwo) = ave;
               fit(jthree,jfour) = ave;

               indexll = indexll + 2;

            end

         elseif((abs(targ(jone,jtwo) - targ(jthree,jfour)) ...
               > 1.0e-006) & (targ(jone,jtwo) > ...
               targ(jthree,jfour)))

            if(fit(jone,jtwo) > fit(jthree,jfour))

               work(indexll+1) = 0;
               work(indexll+2) = 0;
```

```
                        indexll = indexll + 2;

                     elseif(fit(jone,jtwo) <= fit(jthree,jfour))

                        ave = (fit(jone,jtwo) + fit(jthree,jfour))/2.0;
                        work(indexll+1) = ave - fit(jone,jtwo);
                        work(indexll+2) = ave - fit(jthree,jfour);
                        fit(jone,jtwo) = ave;
                        fit(jthree,jfour) = ave;

                        indexll = indexll + 2;

                     end
                  end

               cr = cr + abs(p1-fit(jone,jtwo)) + ...
                  abs(p2-fit(jthree,jfour));
            end


            end
         end
      end
   end
end




   for jone = 1:(n-1)
      for jtwo = (jone+1):n

         fit(jtwo,jone) = fit(jone,jtwo);

      end
   end

  aveprox = sum(sum(prox))/(n*(n-1));

for i = 1:n
   for j = 1:n
      if( i ~= j)
         proxave(i,j) = aveprox;
      else
         proxave(i,j) = 0;
      end
   end
end
```

```
diff = sum(sum((prox - fit).^2));

denom = sum(sum((prox - proxave).^2));

vaf = 1 - (diff/denom);
```

A.2 ultrafnd.m

```
function [find,vaf] = ultrafnd(prox,inperm)

% ULTRAFND finds and fits an ultrametric using iterative projection
% heuristically on a symmetric proximity matrix in the $L_{2}$-norm.
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% INPERM is a permutation that determines the order in which the
% inequality constraints are considered;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROX satisfying the ultrametric constraints.

n = size(prox,1);
work = zeros(n*(n-1)*(n-2),1);
find = prox;
cr = 1.0;
iterate = 0;

while (cr >= 1.0e-006)
   cr = 0.0;
   indexll = 0;
   iterate = iterate + 1;


   for ione = 1:(n-2)
      for itwo = (ione+1):(n-1)
        for ithree = (itwo+1):n

            indxord(1) = inperm(ione);
            indxord(2) = inperm(itwo);
            indxord(3) = inperm(ithree);

            for i = 1:2
               for j = (i+1):3

                  if(indxord(i) > indxord(j))

                     temp = indxord(i);
                     indxord(i) = indxord(j);
                     indxord(j) = temp;

                  end
               end
```

```
end


jone = indxord(1);
jtwo = indxord(2);
jthree = indxord(3);

p1 = find(jone,jtwo);
p2 = find(jone,jthree);
p3 = find(jtwo,jthree);

if(iterate <= 100)

find(jone,jtwo) = find(jone,jtwo) - work(indexll+1);
find(jone,jthree) = find(jone,jthree) - work(indexll+2);
find(jtwo,jthree) = find(jtwo,jthree) - work(indexll+3);

end


if((find(jone,jtwo) <= find(jone,jthree)) & ...
      (find(jone,jtwo) <= find(jtwo,jthree)))

   ave = (find(jone,jthree) + find(jtwo,jthree))/2.0;

   work(indexll+1) = 0.0;
   work(indexll+2) = ave - find(jone,jthree);
   work(indexll+3) = ave - find(jtwo,jthree);

   find(jone,jthree) = ave;
   find(jtwo,jthree) = ave;

   indexll = indexll + 3;

 elseif((find(jone,jthree) <= find(jone,jtwo)) & ...
      (find(jone,jthree) <= find(jtwo,jthree)))

   ave = (find(jone,jtwo) + find(jtwo,jthree))/2.0;

   work(indexll+1) = ave - find(jone,jtwo);
   work(indexll+2) = 0.0;
   work(indexll+3) = ave - find(jtwo,jthree);

   find(jone,jtwo) = ave;
   find(jtwo,jthree) = ave;

   indexll = indexll + 3;

 elseif((find(jtwo,jthree) <= find(jone,jthree)) & ...
      (find(jtwo,jthree) <= find(jone,jtwo)))
```

```
               ave = (find(jone,jthree) + find(jone,jtwo))/2.0;

               work(indexll+1) = ave - find(jone,jtwo);
               work(indexll+2) = ave - find(jone,jthree);
               work(indexll+3) = 0.0;

               find(jone,jtwo) = ave;
               find(jone,jthree) = ave;

               indexll = indexll + 3;
            end

               cr = cr + abs(p1-find(jone,jtwo)) + abs(p2-find(jone,jthree)) ...
               + abs(p3-find(jtwo,jthree));

         end
      end
   end
end

for jone = 1:(n-1)
   for jtwo = (jone+1):n

      find(jtwo,jone) = find(jone,jtwo);

   end
end

[fit, vaf] = ultrafit(prox,find);

find = fit;

aveprox = sum(sum(prox))/(n*(n-1));

for i = 1:n
   for j = 1:n
      if( i ~= j)
         proxave(i,j) = aveprox;
      else
         proxave(i,j) = 0;
     end
   end
end

diff = sum(sum((prox - find).^2));

denom = sum(sum((prox - proxave).^2));

vaf = 1 - (diff/denom);
```

A.3 atreefit.m

```
function [fit,vaf] = atreefit(prox,targ)

% ATREEFIT fits a given additive tree using iterative projection to
% a symmetric proximity matrix in the $L_{2}$-norm.
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% TARG is an matrix of the same size as PROX with entries
% satisfying the four-point additive tree constraints;
% FIT is the least-squares optimal matrix (with variance-accounted-for
% of VAF) to PROX satisfying the additive tree constraints implicit in TARG.

n = size(prox,1);
work = zeros(n*(n-1)*(n-2)*(n-3),1);
fit = prox;
cr = 1.0;

while (cr >= 1.0e-006)
   cr = 0.0;
   indexll = 0;


   for jone = 1:(n-3)
      for jtwo = (jone+1):(n-2)
         for jthree = (jtwo+1):(n-1)
            for jfour = (jthree+1):n


            p1 = fit(jone,jtwo);
            p2 = fit(jone,jthree);
            p3 = fit(jone,jfour);
            p4 = fit(jtwo,jthree);
            p5 = fit(jtwo,jfour);
            p6 = fit(jthree,jfour);



            fit(jone,jtwo) = fit(jone,jtwo) - work(indexll+1);
            fit(jone,jthree) = fit(jone,jthree) - work(indexll+2);
            fit(jone,jfour) = fit(jone,jfour) - work(indexll+3);
            fit(jtwo,jthree) = fit(jtwo,jthree) - work(indexll+4);
            fit(jtwo,jfour) = fit(jtwo,jfour) - work(indexll+5);
            fit(jthree,jfour) = fit(jthree,jfour) - work(indexll+6);



            if(((targ(jone,jtwo) + targ(jthree,jfour)) <= ...
                  (targ(jone,jthree) + targ(jtwo,jfour))) & ...
                  ((targ(jone,jtwo) + targ(jthree,jfour)) <=  ...
                  (targ(jone,jfour) + targ(jtwo,jthree))))
               del = (fit(jone,jthree) + fit(jtwo,jfour) - ...
                  (fit(jone,jfour) + fit(jtwo,jthree)))/4.0;

               work(indexll+1) = 0.0;
               work(indexll+2) = -del;
               work(indexll+3) =  del;
               work(indexll+4) = del;
               work(indexll+5) = -del;
               work(indexll+6) = 0.0;

               fit(jone,jthree) = fit(jone,jthree) - del;
               fit(jtwo,jfour) = fit(jtwo,jfour) - del;
               fit(jone,jfour) = fit(jone,jfour) + del;
               fit(jtwo,jthree) = fit(jtwo,jthree) + del;

               indexll = indexll + 6;

             elseif(((targ(jone,jthree) + targ(jtwo,jfour)) <= ...
                  (targ(jone,jtwo) + targ(jthree,jfour))) & ...
                  ((targ(jone,jthree) + targ(jtwo,jfour)) <=  ...
                  (targ(jone,jfour) + targ(jtwo,jthree))))

               del = (fit(jone,jtwo) + fit(jthree,jfour) - ...
                  (fit(jone,jfour) + fit(jtwo,jthree)))/4.0;

               work(indexll+1) = -del;
               work(indexll+2) = 0.0;
               work(indexll+3) =  del;
               work(indexll+4) = del;
               work(indexll+5) = 0.0;
               work(indexll+6) = -del;

               fit(jone,jtwo) = fit(jone,jtwo) - del;
               fit(jthree,jfour) = fit(jthree,jfour) - del;
               fit(jone,jfour) = fit(jone,jfour) + del;
               fit(jtwo,jthree) = fit(jtwo,jthree) + del;

               indexll = indexll + 6;

            elseif(((targ(jone,jfour) + targ(jtwo,jthree)) <= ...
                  (targ(jone,jtwo) + targ(jthree,jfour))) & ...
                  ((targ(jone,jfour) + targ(jtwo,jthree)) <=  ...
                  (targ(jone,jthree) + targ(jtwo,jfour))))

               del = (fit(jone,jtwo) + fit(jthree,jfour) - ...
                  (fit(jone,jthree) + fit(jtwo,jfour)))/4.0;

               work(indexll+1) = -del;
               work(indexll+2) =  del;
               work(indexll+3) =  0.0;
```

```
               work(indexll+4) =  0.0;
               work(indexll+5) =  del;
               work(indexll+6) = -del;

               fit(jone,jtwo) = fit(jone,jtwo) - del;
               fit(jthree,jfour) = fit(jthree,jfour) - del;
               fit(jone,jthree) = fit(jone,jthree) + del;
               fit(jtwo,jfour) = fit(jtwo,jfour) + del;

               indexll = indexll + 6;
            end


               cr = cr + abs(p1-fit(jone,jtwo)) + abs(p2-fit(jone,jthree)) ...
                  + abs(p3-fit(jone,jfour)) + abs(p4-fit(jtwo,jthree)) ...
                  + abs(p5-fit(jtwo,jfour)) + abs(p6-fit(jthree,jfour));
            end
         end
      end
   end
end

for jone = 1:(n-1)
   for jtwo = (jone+1):n

      fit(jtwo,jone) = fit(jone,jtwo);

   end
end

aveprox = sum(sum(prox))/(n*(n-1));

for i = 1:n
   for j = 1:n
      if( i ~= j)
         proxave(i,j) = aveprox;
      else
         proxave(i,j) = 0;
      end
   end
end

diff = sum(sum((prox - fit).^2));

denom = sum(sum((prox - proxave).^2));

vaf = 1 - (diff/denom);
```

A.4 atreefnd.m

```
function [find,vaf] = atreefnd(prox,inperm)

% ATREEFIND finds and fits an additive tree using iterative projection
% heuristically on a symmetric proximity matrix in the $L_{2}$-norm.
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% INPERM is a permutation that determines the order in which the
% inequality constraints are considered;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROX satisfying the additive tree constraints.


n = size(prox,1);
work = zeros(n*(n-1)*(n-2)*(n-3),1);
find = prox;
cr = 1.0;
iterate = 0;

while (cr >= 1.0e-006)
   cr = 0.0;
   indexll = 0;
   iterate = iterate + 1;


   for ione = 1:(n-3)
      for itwo = (ione+1):(n-2)
         for ithree = (itwo+1):(n-1)
            for ifour = (ithree+1):n

            indxord(1) = inperm(ione);
            indxord(2) = inperm(itwo);
            indxord(3) = inperm(ithree);
            indxord(4) = inperm(ifour);

            for i = 1:3
               for j = (i+1):4

                  if(indxord(i) > indxord(j))

                     temp = indxord(i);
                     indxord(i) = indxord(j);
                     indxord(j) = temp;

                  end
               end
            end


            jone = indxord(1);
```

```
jtwo = indxord(2);
jthree = indxord(3);
jfour = indxord(4);

p1 = find(jone,jtwo);
p2 = find(jone,jthree);
p3 = find(jone,jfour);
p4 = find(jtwo,jthree);
p5 = find(jtwo,jfour);
p6 = find(jthree,jfour);


if(iterate <= 100)

find(jone,jtwo) = find(jone,jtwo) - work(indexll+1);
find(jone,jthree) = find(jone,jthree) - work(indexll+2);
find(jone,jfour) = find(jone,jfour) - work(indexll+3);
find(jtwo,jthree) = find(jtwo,jthree) - work(indexll+4);
find(jtwo,jfour) = find(jtwo,jfour) - work(indexll+5);
find(jthree,jfour) = find(jthree,jfour) - work(indexll+6);

end


if(((find(jone,jtwo) + find(jthree,jfour)) <= ...
      (find(jone,jthree) + find(jtwo,jfour))) & ...
      ((find(jone,jtwo) + find(jthree,jfour)) <=  ...
      (find(jone,jfour) + find(jtwo,jthree))))

   del = (find(jone,jthree) + find(jtwo,jfour) - ...
      (find(jone,jfour) + find(jtwo,jthree)))/4.0;

   work(indexll+1) = 0.0;
   work(indexll+2) = -del;
   work(indexll+3) =  del;
   work(indexll+4) = del;
   work(indexll+5) = -del;
   work(indexll+6) = 0.0;

   find(jone,jthree) = find(jone,jthree) - del;
   find(jtwo,jfour) = find(jtwo,jfour) - del;
   find(jone,jfour) = find(jone,jfour) + del;
   find(jtwo,jthree) = find(jtwo,jthree) + del;

   indexll = indexll + 6;

 elseif(((find(jone,jthree) + find(jtwo,jfour)) <= ...
      (find(jone,jtwo) + find(jthree,jfour))) & ...
      ((find(jone,jthree) + find(jtwo,jfour)) <=  ...
      (find(jone,jfour) + find(jtwo,jthree))))
```

```
               del = (find(jone,jtwo) + find(jthree,jfour) - ...
                  (find(jone,jfour) + find(jtwo,jthree)))/4.0;

               work(indexll+1) = -del;
               work(indexll+2) = 0.0;
               work(indexll+3) =  del;
               work(indexll+4) = del;
               work(indexll+5) = 0.0;
               work(indexll+6) = -del;

               find(jone,jtwo) = find(jone,jtwo) - del;
               find(jthree,jfour) = find(jthree,jfour) - del;
               find(jone,jfour) = find(jone,jfour) + del;
               find(jtwo,jthree) = find(jtwo,jthree) + del;

               indexll = indexll + 6;

            elseif(((find(jone,jfour) + find(jtwo,jthree)) <= ...
                  (find(jone,jtwo) + find(jthree,jfour))) & ...
                  ((find(jone,jfour) + find(jtwo,jthree)) <=  ...
                  (find(jone,jthree) + find(jtwo,jfour))))

               del = (find(jone,jtwo) + find(jthree,jfour) - ...
                  (find(jone,jthree) + find(jtwo,jfour)))/4.0;

               work(indexll+1) = -del;
               work(indexll+2) =  del;
               work(indexll+3) =  0.0;
               work(indexll+4) =  0.0;
               work(indexll+5) =  del;
               work(indexll+6) = -del;

               find(jone,jtwo) = find(jone,jtwo) - del;
               find(jthree,jfour) = find(jthree,jfour) - del;
               find(jone,jthree) = find(jone,jthree) + del;
               find(jtwo,jfour) = find(jtwo,jfour) + del;

               indexll = indexll + 6;
            end


               cr = cr + abs(p1-find(jone,jtwo)) + abs(p2-find(jone,jthree)) ...
                  + abs(p3-find(jone,jfour)) + abs(p4-find(jtwo,jthree)) ...
                  + abs(p5-find(jtwo,jfour)) + abs(p6-find(jthree,jfour));
            end
         end
      end
   end
end

for jone = 1:(n-1)
```

```
   for jtwo = (jone+1):n

      find(jtwo,jone) = find(jone,jtwo);

   end
end


[fit,vaf] = atreefit(prox,find);

find = fit;

aveprox = sum(sum(prox))/(n*(n-1));

for i = 1:n
   for j = 1:n
      if( i ~= j)
         proxave(i,j) = aveprox;
      else
         proxave(i,j) = 0;
      end
   end
end

diff = sum(sum((prox - find).^2));

denom = sum(sum((prox - proxave).^2));

vaf = 1 - (diff/denom);
```

A.5 atreedec.m

```
function [ulmetric,ctmetric] = atreedec(prox,constant)

% ATREEDEC decomposes a given additive tree matrix into an ultrametric and a
% centroid metric matrix (where the root is half-way along the longest path).
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% CONSTANT is a nonnegative number (less than or equal to the maximum
% proximity value) that controls the positivity of the constructed ultrametric values;
% ULMETRIC is the ultrametric component of the decomposition;
% CTMETRIC is the centoid metric component of the decomposition (given
% by values $g_{1},...,g_{n}$ for each of the objects, some of which
% may actually be negative depending on the input proximity matrix used).


n = size(prox,1);
imax = 0.0;
jmax = 0.0;
valmax = 0.0;
```

```
ctmetric = zeros(n,1);

for i = 1:(n-1)
   for j = (i+1):n

      if(prox(i,j) > valmax)

         valmax = prox(i,j);
         imax = i;
         jmax = j;

      end

   end
end

for i = 1:n

   ctmetric(i) = max([prox(i,imax) prox(i,jmax)]) - (constant);

end



for i = 1:n
   for j = 1:n

      if(i ~= j)

         ulmetric(i,j) = prox(i,j) - ctmetric(i) - ctmetric(j);

      else

         ulmetric(i,j) = 0.0;

      end

   end
end
```

A.6 centfit.m

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

% CENTFIT finds the least-squares fitted centroid metric (FIT) to
% PROX, the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% The $n$ values that serve to define the approximating sums,
% $g_{i} + g_{j}$, are given in  the vector LENGTHS of size n x 1.
```

```
n = size(prox,1);
lengths = zeros(n);
rsums = sum(prox);
tsum = sum(rsums);

lengths = (rsums/(n-2)) - (tsum/(2*(n-1)*(n-2)));

for i = 1:n
   for j = 1:n

      if (i ~= j)

         fit(i,j) = lengths(i) + lengths(j);

      else

         fit(i,j) = 0;
      end
   end
end


  aveprox = sum(sum(prox))/(n*(n-1));

for i = 1:n
   for j = 1:n
      if( i ~= j)
         proxave(i,j) = aveprox;
      else
         proxave(i,j) = 0;
      end
   end
end

diff = sum(sum((prox - fit).^2));

denom = sum(sum((prox - proxave).^2));

vaf = 1 - (diff/denom);
```

A.7 atreectul.m

```
function [find,vaf] = atreectul(prox,inperm)

% ATREEFINDCTUL finds and fits an additive tree by first fitting
% a centroid metric (using centfit.m) and secondly an ultrametric to the resudual
% matrix (using ultrafnd.m).
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% INPERM is a permutation that determines the order in which the
% inequality constraints are considered;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROX satisfying the additive tree constraints.

n = size(prox,1);

[ctfnd,ctvaf,lengths] = centfit(prox);

resprox = prox - ctfnd;

[ulfnd,ulvaf] = ultrafnd(resprox,inperm);

targ = ctfnd + ulfnd;

[find,vaf] = atreefit(prox,targ);
```

A.8 biultrafnd.m

```
function [find,vaf,targone,targtwo] = biultrafnd(prox,inperm)

% BIULTRAFND finds and fits the sum of two ultrametrics using iterative projection
% heuristically on a symmetric proximity matrix in the $L_{2}$-norm.
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% INPERM is a permutation that determines the order in which the
% inequality constraints are considered;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROX and is the sum of the two ultrametric matrices TARGONE and TARGTWO.

n = size(prox,1);

[targ1,vaftarg1] = ultrafnd(prox,inperm);

resprox = prox - targ1;

[targ2,vaftarg2] = ultrafnd(resprox,inperm);

find = targ1 + targ2;

aveprox = sum(sum(prox))/(n*(n-1));

for i = 1:n
    for j = 1:n
        if(i ~= j)
            proxave(i,j) = aveprox;
        else
            proxave(i,j) = 0.0;
        end
    end
```

```
end

diff = sum(sum((prox - find).^2));
denom = sum(sum((prox - proxave).^2));
vaf = 1 - (diff/denom);

targone = targ1;
targtwo = targ2;

vafdiff = 1.0;

while (vafdiff >= 1.0e-006)

    vafprev = vaf;

    resprox = prox - targtwo;
    [targone,vafone] = ultrafnd(resprox,inperm);

    resprox = prox - targone;
    [targtwo,vaftwo] = ultrafnd(resprox,inperm);

    find = targone + targtwo;

    diff = sum(sum((prox - find).^2));
    denom = sum(sum((prox - proxave).^2));
    vaf = 1 - (diff/denom);

    vafdiff = abs(vaf - vafprev);
end
```

A.9 biatreefnd.m

```
function [find,vaf,targone,targtwo] = biatreefnd(prox,inperm)


% BIATREEFND finds and fits the sum of two additive trees using iterative projection
% heuristically on a symmetric proximity matrix in the $L_{2}$-norm.
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% INPERM is a permutation that determines the order in which the
% inequality constraints are considered;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROX and is the sum of the two additive tree matrices TARGONE and TARGTWO.

n = size(prox,1);

[targ1,vaftarg1] = atreefnd(prox,inperm);

resprox = prox - targ1;
```

```
[targ2,vaftarg2] = atreefnd(resprox,inperm);

find = targ1 + targ2;

aveprox = sum(sum(prox))/(n*(n-1));

for i = 1:n
    for j = 1:n
        if(i ~= j)
            proxave(i,j) = aveprox;
        else
            proxave(i,j) = 0.0;
        end
    end
end

diff = sum(sum((prox - find).^2));
denom = sum(sum((prox - proxave).^2));
vaf = 1 - (diff/denom)

targone = targ1;
targtwo = targ2;

vafdiff = 1.0;

while (vafdiff >= 1.0e-006)

    vafprev = vaf;

    resprox = prox - targtwo;
    [targone,vafone] = atreefnd(resprox,inperm);


    resprox = prox - targone;
    [targtwo,vaftwo] = atreefnd(resprox,inperm);

    find = targone + targtwo;

    diff = sum(sum((prox - find).^2));
    denom = sum(sum((prox - proxave).^2));
    vaf = 1 - (diff/denom)

    vafdiff = abs(vaf - vafprev);
end
```

A.10 ultraorder.m

```
function [orderprox,orderperm] = ultraorder(prox)
```

```
% ULTRAORDER finds for the input proximity matrix PROX
% (assumed to be ultrametric with a zero main diagonal),
% a permutation ORDERPERM that displays the anti-
% Robinson form in the reordered proximity matrix
% ORDERPROX; thus, prox(orderperm,orderperm) = orderprox.

n = size(prox,1);

[proxn,targlin,targcir] = ransymat(n);

inperm = 1:n;

[orderperm,rawindex,allperms,index] = pairwiseqa(prox,targlin,inperm);

orderprox = prox(orderperm,orderperm);
```

A.11 ultrafittm.m

```
function [fit,vaf] = ultrafittm(proxtm,targ)


% ULTRAFITTM fits a given (two-mode) ultrametric using iterative projection to
% a two-mode (rectangular) proximity matrix in the $L_{2}$-norm.
% PROXTM is the input proximity matrix (with a dissimilarity interpretation);
% TARG is an ultrametric matrix of the same size as PROXTM;
% FIT is the least-squares optimal matrix (with variance-accounted-for
% of VAF) to PROXTM satisfying the ultrametric constraints implicit in TARG.

[nrow ncol] = size(proxtm);
work = zeros(3*nrow*nrow*ncol*ncol,1);
fit = proxtm;
cr = 1.0;
iterate = 0;

while (cr >= 1.0e-006)


   cr = 0.0;
   indexll = 0;
   iterate = iterate + 1;


   for jone = 1:(nrow-1)
      for jtwo = (jone+1):nrow
         for jthree = 1:(ncol-1)
            for jfour = (jthree+1):ncol



            p1 = fit(jone,jthree);
```

```
p2 = fit(jone,jfour);
p3 = fit(jtwo,jthree);
p4 = fit(jtwo,jfour);



fit(jone,jthree) = fit(jone,jthree) - work(indexll+1);
fit(jone,jfour) = fit(jone,jfour) - work(indexll+2);
fit(jtwo,jthree) = fit(jtwo,jthree) - work(indexll+3);
fit(jtwo,jfour) = fit(jtwo,jfour) - work(indexll+4);




if(((targ(jone,jthree) <= targ(jtwo,jthree)) & ...
      (targ(jone,jfour) <= targ(jtwo,jthree))) & ...
      ((targ(jone,jthree) <= targ(jtwo,jfour)) & ...
      (targ(jone,jfour) <= targ(jtwo,jfour))))

 ave = (fit(jtwo,jthree) + fit(jtwo,jfour))/2.0;

   work(indexll+1) = 0.0;
   work(indexll+2) = 0.0;
   work(indexll+3) = ave - fit(jtwo,jthree);
   work(indexll+4) = ave - fit(jtwo,jfour);

   fit(jtwo,jthree) = ave;
   fit(jtwo,jfour) = ave;

   indexll = indexll + 4;






 elseif(((targ(jone,jthree) >= targ(jtwo,jthree)) & ...
      (targ(jone,jfour) >= targ(jtwo,jthree))) & ...
      ((targ(jone,jthree) >= targ(jtwo,jfour)) & ...
      (targ(jone,jfour) >= targ(jtwo,jfour))))


   ave = (fit(jone,jthree) + fit(jone,jfour))/2.0;



   work(indexll+1) = ave - fit(jone,jthree);
   work(indexll+2) = ave - fit(jone,jfour);
   work(indexll+3) = 0.0;
   work(indexll+4) = 0.0;
```

```
  fit(jone,jthree) = ave;
  fit(jone,jfour) = ave;

  indexll = indexll + 4;



elseif(((targ(jone,jthree) <= targ(jone,jfour)) & ...
     (targ(jtwo,jthree) <= targ(jone,jfour))) & ...
     ((targ(jone,jthree) <= targ(jtwo,jfour)) & ...
     (targ(jtwo,jthree) <= targ(jtwo,jfour))))




  ave = (fit(jone,jfour) + fit(jtwo,jfour))/2.0;

  work(indexll+1) = 0.0;
  work(indexll+2) = ave - fit(jone,jfour);
  work(indexll+3) = 0.0;
  work(indexll+4) = ave - fit(jtwo,jfour);

  fit(jone,jfour) = ave;
  fit(jtwo,jfour) = ave;

  indexll = indexll + 4;



elseif(((targ(jone,jthree) >= targ(jone,jfour)) & ...
     (targ(jtwo,jthree) >= targ(jone,jfour))) & ...
     ((targ(jone,jthree) >= targ(jtwo,jfour)) & ...
     (targ(jtwo,jthree) >= targ(jtwo,jfour))))

  ave = (fit(jone,jthree) + fit(jtwo,jthree))/2.0;

  work(indexll+1) = ave - fit(jone,jthree);
  work(indexll+2) = 0.0;
  work(indexll+3) = ave - fit(jtwo,jthree);
  work(indexll+4) = 0.0;

  fit(jone,jthree) = ave;
  fit(jtwo,jthree) = ave;

  indexll = indexll + 4;



elseif(((targ(jone,jfour) <= targ(jone,jthree)) & ...
     (targ(jtwo, jthree) <= targ(jone,jthree))) & ...
     ((targ(jone,jfour) <= targ(jtwo,jfour)) & ...
      (targ(jtwo,jthree) <= targ(jtwo,jfour))))


   ave = (fit(jone,jthree) + fit(jtwo,jfour))/2.0;

   work(indexll+1) = ave - fit(jone,jthree);
   work(indexll+2) = 0.0;
   work(indexll+3) = 0.0;
   work(indexll+4) = ave - fit(jtwo,jfour);

   fit(jone,jthree) = ave;
   fit(jtwo,jfour) = ave;

   indexll = indexll + 4;



 elseif(((targ(jone,jfour) >= targ(jone,jthree)) & ...
      (targ(jtwo, jthree) >= targ(jone,jthree))) & ...
      ((targ(jone,jfour) >= targ(jtwo,jfour)) & ...
      (targ(jtwo,jthree) >= targ(jtwo,jfour))))




   ave = (fit(jone,jfour) + fit(jtwo,jthree))/2.0;

   work(indexll+1) = 0.0;
   work(indexll+2) = ave - fit(jone,jfour);
   work(indexll+3) = ave - fit(jtwo,jthree);
   work(indexll+4) = 0.0;

   fit(jone,jfour) = ave;
   fit(jtwo,jthree) = ave;

   indexll = indexll + 4;

else







end

cr = cr + abs(p1-fit(jone,jthree)) + ...
   abs(p2-fit(jone,jfour)) ...
   + abs(p3-fit(jtwo,jthree)) + ...
   abs(p4-fit(jtwo,jfour));
```

```
         end
      end
   end
end




   for jone = 1:nrow
      for jtwo = 1:ncol
         for jthree = jone:nrow
            for jfour = jtwo:ncol

               if((jone ~= jthree) | (jtwo ~= jfour))

                  p1 = fit(jone,jtwo);
                  p2 = fit(jthree,jfour);
                  fit(jone,jtwo) = fit(jone,jtwo) - work(indexll+1);
                  fit(jthree,jfour) = fit(jthree,jfour) - ...
                     work(indexll+2);

                  if(abs(targ(jone,jtwo) - targ(jthree,jfour)) <= ...
                        1.0e-006)

                     ave = (fit(jone,jtwo) + fit(jthree,jfour))/2.0;
                     work(indexll+1) = ave - fit(jone,jtwo);
                     work(indexll+2) = ave - fit(jthree,jfour);
                     fit(jone,jtwo) = ave;
                     fit(jthree,jfour) = ave;

                     indexll = indexll + 2;

                  elseif((abs(targ(jone,jtwo) - targ(jthree,jfour)) ...
                        > 1.0e-006) & (targ(jone,jtwo) < ...
                        targ(jthree,jfour)))

                     if(fit(jone,jtwo) < fit(jthree,jfour))

                        work(indexll+1) = 0;
                        work(indexll+2) = 0;

                        indexll = indexll + 2;

                     elseif(fit(jone,jtwo) >= fit(jthree,jfour))

                        ave = (fit(jone,jtwo) + fit(jthree,jfour))/2.0;
                        work(indexll+1) = ave - fit(jone,jtwo);
                        work(indexll+2) = ave - fit(jthree,jfour);
                        fit(jone,jtwo) = ave;
                        fit(jthree,jfour) = ave;

                        indexll = indexll + 2;

                     end

                  elseif((abs(targ(jone,jtwo) - targ(jthree,jfour)) ...
                        > 1.0e-006) & (targ(jone,jtwo) > ...
                        targ(jthree,jfour)))

                     if(fit(jone,jtwo) > fit(jthree,jfour))

                        work(indexll+1) = 0;
                        work(indexll+2) = 0;

                        indexll = indexll + 2;

                     elseif(fit(jone,jtwo) <= fit(jthree,jfour))

                        ave = (fit(jone,jtwo) + fit(jthree,jfour))/2.0;
                        work(indexll+1) = ave - fit(jone,jtwo);
                        work(indexll+2) = ave - fit(jthree,jfour);
                        fit(jone,jtwo) = ave;
                        fit(jthree,jfour) = ave;

                        indexll = indexll + 2;

                     end
                  end

               cr = cr + abs(p1-fit(jone,jtwo)) + ...
                  abs(p2-fit(jthree,jfour));
            end


            end
         end
      end
   end


  end




aveprox = sum(sum(proxtm))/(nrow*(ncol));
```

```
diff = sum(sum((proxtm - fit).^2));

denom = sum(sum((proxtm - aveprox).^2));

vaf = 1 - (diff/denom);
```

A.12 ultrafndtm.m

```
function [find,vaf] = ultrafndtm(proxtm,inpermrow,inpermcol)

% ULTRAFNDTM finds and fits a two-mode ultrametric using iterative projection
% heuristically on a rectangular proximity matrix in the $L_{2}$-norm.
% PROXTM is the input proximity matrix (with a dissimilarity interpretation);
% INPERMROW and INPERMCOL are permutations for the row and column
% objects that determine the order in which the
% inequality constraints are considered;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROXTM satisfying the ultrametric constraints.

[nrow ncol] = size(proxtm);
work = zeros(nrow*nrow*ncol*ncol,1);
find = proxtm;
cr = 1.0;
iterate = 0;

while (cr >= 1.0e-006)
   cr = 0.0;
   indexll = 0;
   iterate = iterate + 1;



   for ione = 1:(nrow-1)
      for itwo = (ione+1):nrow
         for ithree = 1:(ncol-1)
            for ifour = (ithree+1):ncol

               jone = inpermrow(ione);
               jtwo = inpermrow(itwo);
               jthree = inpermcol(ithree);
               jfour = inpermcol(ifour);



            p1 = find(jone,jthree);
            p2 = find(jone,jfour);
            p3 = find(jtwo,jthree);
            p4 = find(jtwo,jfour);

            if(iterate <= 100)
```

```
find(jone,jthree) = find(jone,jthree) - work(indexll+1);
find(jone,jfour) = find(jone,jfour) - work(indexll+2);
find(jtwo,jthree) = find(jtwo,jthree) - work(indexll+3);
find(jtwo,jfour) = find(jtwo,jfour) - work(indexll+4);

end


if(((find(jone,jthree) <= find(jtwo,jthree)) & ...
      (find(jone,jfour) <= find(jtwo,jthree))) & ...
      ((find(jone,jthree) <= find(jtwo,jfour)) & ...
      (find(jone,jfour) <= find(jtwo,jfour))))

 ave = (find(jtwo,jthree) + find(jtwo,jfour))/2.0;

   work(indexll+1) = 0.0;
   work(indexll+2) = 0.0;
   work(indexll+3) = ave - find(jtwo,jthree);
   work(indexll+4) = ave - find(jtwo,jfour);

   find(jtwo,jthree) = ave;
   find(jtwo,jfour) = ave;

   indexll = indexll + 4;





 elseif(((find(jone,jthree) >= find(jtwo,jthree)) & ...
      (find(jone,jfour) >= find(jtwo,jthree))) & ...
      ((find(jone,jthree) >= find(jtwo,jfour)) & ...
      (find(jone,jfour) >= find(jtwo,jfour))))


   ave = (find(jone,jthree) + find(jone,jfour))/2.0;



   work(indexll+1) = ave - find(jone,jthree);
   work(indexll+2) = ave - find(jone,jfour);
   work(indexll+3) = 0.0;
   work(indexll+4) = 0.0;

   find(jone,jthree) = ave;
   find(jone,jfour) = ave;

   indexll = indexll + 4;
```

```
elseif(((find(jone,jthree) <= find(jone,jfour)) & ...
     (find(jtwo,jthree) <= find(jone,jfour))) & ...
     ((find(jone,jthree) <= find(jtwo,jfour)) & ...
     (find(jtwo,jthree) <= find(jtwo,jfour))))




  ave = (find(jone,jfour) + find(jtwo,jfour))/2.0;

  work(indexll+1) = 0.0;
  work(indexll+2) = ave - find(jone,jfour);
  work(indexll+3) = 0.0;
  work(indexll+4) = ave - find(jtwo,jfour);

  find(jone,jfour) = ave;
  find(jtwo,jfour) = ave;

  indexll = indexll + 4;



elseif(((find(jone,jthree) >= find(jone,jfour)) & ...
     (find(jtwo,jthree) >= find(jone,jfour))) & ...
     ((find(jone,jthree) >= find(jtwo,jfour)) & ...
     (find(jtwo,jthree) >= find(jtwo,jfour))))

  ave = (find(jone,jthree) + find(jtwo,jthree))/2.0;

  work(indexll+1) = ave - find(jone,jthree);
  work(indexll+2) = 0.0;
  work(indexll+3) = ave - find(jtwo,jthree);
  work(indexll+4) = 0.0;

  find(jone,jthree) = ave;
  find(jtwo,jthree) = ave;

  indexll = indexll + 4;



elseif(((find(jone,jfour) <= find(jone,jthree)) & ...
     (find(jtwo, jthree) <= find(jone,jthree))) & ...
     ((find(jone,jfour) <= find(jtwo,jfour)) & ...
     (find(jtwo,jthree) <= find(jtwo,jfour))))


  ave = (find(jone,jthree) + find(jtwo,jfour))/2.0;
```

```
                 work(indexll+1) = ave - find(jone,jthree);
                 work(indexll+2) = 0.0;
                 work(indexll+3) = 0.0;
                 work(indexll+4) = ave - find(jtwo,jfour);

                 find(jone,jthree) = ave;
                 find(jtwo,jfour) = ave;

                 indexll = indexll + 4;



               elseif(((find(jone,jfour) >= find(jone,jthree)) & ...
                    (find(jtwo, jthree) >= find(jone,jthree))) & ...
                    ((find(jone,jfour) >= find(jtwo,jfour)) & ...
                    (find(jtwo,jthree) >= find(jtwo,jfour))))




                 ave = (find(jone,jfour) + find(jtwo,jthree))/2.0;

                 work(indexll+1) = 0.0;
                 work(indexll+2) = ave - find(jone,jfour);
                 work(indexll+3) = ave - find(jtwo,jthree);
                 work(indexll+4) = 0.0;

                 find(jone,jfour) = ave;
                 find(jtwo,jthree) = ave;

                 indexll = indexll + 4;

              end

              cr = cr + abs(p1-find(jone,jthree)) + ...
                 abs(p2-find(jone,jfour)) ...
                 + abs(p3-find(jtwo,jthree)) + ...
                 abs(p4-find(jtwo,jfour));

           end
        end
     end
  end
  end

  [fit,vaf] = ultrafittm(proxtm,find);

  find = fit;

  aveprox = sum(sum(proxtm))/(nrow*(ncol));
```

```
diff = sum(sum((proxtm- find).^2));

denom = sum(sum((proxtm- aveprox).^2));

vaf = 1 - (diff/denom);
```

A.13 centfittm.m

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

% CENTFITTM finds the least-squares fitted two-mode centroid metric (FIT) to
% PROXTM, the two-mode rectangular input proximity matrix (with
% a dissimilarity interpretation);
% 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 x 1,
% with row values first followed by the column values.


   [nrow ncol] = size(proxtm);
   n = nrow + ncol;
   lengths = zeros(n,1);
   fit = zeros(nrow,ncol);

   cmeans = sum(proxtm)/nrow;
   rmeans = sum(proxtm')/ncol;
   gmean = sum(sum(proxtm))/(nrow*ncol);

   for i = 1:nrow

      lengths(i) = rmeans(i) - ((1/2)*gmean);

   end

   for j = 1:ncol

      lengths(j+nrow) = cmeans(j) - ((1/2)*gmean);

   end

   for i = 1:nrow
      for j = 1:ncol

         fit(i,j) = lengths(i) + lengths(j+nrow);

      end
   end

   diff = sum(sum((proxtm - fit).^2));
   denom = sum(sum((proxtm - gmean).^2));
   vaf = 1 - (diff/denom);
```

A.14 atreefndtm.m

```
function [find,vaf,ultrafit,lengths] = atreefndtm(proxtm,inpermrow,inpermcol)

% ATREEFNDTM finds and fits a two-mode additive tree; iterative projection is used
% heuristically to find a two-mode ultrametric component that
% is added to a two-mode centroid metric to produce the two-mode additive tree.
% PROXTM is the input proximity matrix (with a dissimilarity interpretation);
% INPERMROW and INPERMCOL are permutations for the row and column
% objects that determine the order in which the
% inequality constraints are considered;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROXTM satisfying the additive tree constraints
% the vector LENGTHS contains the row followed by column values for the
% two-mode centroid metric component; ULTRA is the ultrametric component.


[nrow ncol] = size(proxtm);

[centfit,centvaf,lengths] = centfittm(proxtm);

resproxtm = proxtm - centfit;

[ultrafit,ultravaf] = ultrafndtm(resproxtm,inpermrow,inpermcol);

find = centfit + ultrafit;

aveprox = sum(sum(proxtm))/(nrow*(ncol));

diff = sum(sum((proxtm- find).^2));

denom = sum(sum((proxtm- aveprox).^2));

vaf = 1 - (diff/denom);


vafdiff = 1.0;

while (vafdiff >= 1.0e-006)

    vafprev = vaf;

    resproxtm = proxtm - ultrafit;
    [centfit,centvaf,lengths] = centfittm(resproxtm);


    resproxtm = proxtm - centfit;
    [ultrafit,ultravaf] = ultrafndtm(resproxtm,inpermrow,inpermcol);

    find = centfit + ultrafit;

    diff = sum(sum((proxtm - find).^2));
    denom = sum(sum((proxtm - aveprox).^2));
    vaf = 1 - (diff/denom);

    vafdiff = abs(vaf - vafprev);
end
```

A.15 ordertm.m

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

% 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);
% 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 permuation 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.

[nrow ncol] = size(proxtm);
n = nrow + ncol;
prox = zeros(n,n);
aveprox = sum(sum(proxtm))/(nrow*ncol);

for irow = 1:nrow
   for icol = 1:ncol

      prox(irow,icol+nrow) = proxtm(irow,icol) - aveprox;
      prox(icol+nrow,irow) = proxtm(irow,icol) - aveprox;

   end
end

outperm = inperm;
index = 1;
allperms{index} = inperm;
begindex = sum(sum(prox(inperm,inperm).*targ));

for iterate = 1:2

   nchange = 1;

while (nchange == 1)

   nchange=0;

   for k = 1:(n-1)
      for j = (k+1):n

         intrperm = outperm;

         intrperm(k) = outperm(j);
         intrperm(j) = outperm(k);

         tryindex = sum(sum(prox(intrperm,intrperm).*targ));

         if(tryindex > (begindex + 1.0e-008))
            nchange = 1;
            begindex = tryindex;
            outperm = intrperm;
            index = index + 1;
            allperms{index} = intrperm;
         end

      end
   end
end

rawindex = begindex;
nchange = 1;

while (nchange == 1)

   nchange=0;

   for k = 1:kblock
      for insertpt = 1:(n+1)
         for nlimlow = 1:(n+1-k)
```

```
            intrperm = outperm;

            if (nlimlow > insertpt)

               jtwo = 0;
               for j = insertpt:(insertpt+k-1)
                  intrperm(j) =outperm(nlimlow+jtwo);
                  jtwo = jtwo + 1;
               end

               jone = 0;
               for j = (insertpt+k):(nlimlow+k-1);
                  intrperm(j) = outperm(insertpt+jone);
                  jone = jone + 1;
               end

            elseif ((nlimlow+k) < insertpt)

               jtwo = 0;
               for j = (insertpt-k):(insertpt-1)
                  intrperm(j) = outperm(nlimlow+jtwo);
                  jtwo = jtwo + 1;
               end

               jone = 0;
               for j = nlimlow:(insertpt-k-1)
                  intrperm(j) = outperm(nlimlow+k+jone);
                  jone = jone + 1;
               end

            else

            end

            tryindex = sum(sum(prox(intrperm,intrperm).*targ));

            if(tryindex > (begindex + 1.0e-008))
               nchange = 1;
               begindex = tryindex;
               outperm = intrperm;
               index = index +1;
               allperms{index} = intrperm;
            end

         end
      end
   end
end


rawindex = begindex;
```

```
nchange = 1;


while (nchange == 1)

   nchange=0;

   for k = 2:kblock
      for nlimlow = 1:(n+1-k)

         intrperm = outperm;

         for j = 1:k
            intrperm(nlimlow+j-1) = outperm(nlimlow+k-j);
         end

         tryindex = sum(sum(prox(intrperm,intrperm).*targ));

         if(tryindex > (begindex + 1.0e-008))
            nchange = 1;
            begindex = tryindex;
            outperm = intrperm;
            index = index + 1;
            allperms{index} = intrperm;
         end

      end
   end
end



rawindex = begindex;
nchange = 1;


end


squareprox = prox;
```

A.16 linfittm.m

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

%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.
%  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.

[nrow ncol] = size(proxtm);
n = nrow + ncol;
work = zeros(n*(n-1)*(n-2)*(n-3),1);
tempfit = zeros(n,n);
fit = zeros(nrow,ncol);
mfit = zeros(n,n);

for irow = 1:nrow
   for icol = 1:ncol

      tempfit(irow,icol+nrow) = proxtm(irow,icol);
      tempfit(icol+nrow,irow) = proxtm(irow,icol);

   end
end

mfit = tempfit(inperm,inperm);

cr = 1.0;

while(cr >= 1.0e-005)

   cr = 0.0;
   indexll=0;

   for ione = 1:(n-3)
      for itwo = (ione+1):(n-2)
         for ithree = (itwo+1):(n-1)
            for ifour = (ithree+1):n

               if((((inperm(ione) <= nrow) & (inperm(itwo) <= nrow)) ...
                     & ((inperm(ithree) > nrow) & (inperm(ifour) ...
                     > nrow))) | ...
                     (((inperm(ione) > nrow) & (inperm(itwo) > nrow)) ...
                     & ((inperm(ithree) <= nrow) & (inperm(ifour) ...
                     <= nrow))))

                  p1 = mfit(ione,ithree);
                  p2 = mfit(ione,ifour);
                  p3 = mfit(itwo,ithree);
                  p4 = mfit(itwo,ifour);

                  mfit(ione,ithree) = mfit(ione,ithree) - work(indexll+1);
                  mfit(ione,ifour) = mfit(ione,ifour) - work(indexll+2);
```

```
  mfit(itwo,ithree) = mfit(itwo,ithree) - work(indexll+3);
  mfit(itwo,ifour) = mfit(itwo,ifour) - work(indexll+4);

  del = (mfit(itwo,ithree) + mfit(ione,ifour) - ...
     mfit(ione,ithree) - mfit(itwo,ifour))/4.0;

  mfit(itwo,ithree) = mfit(itwo,ithree) - del;
  mfit(ione,ifour) = mfit(ione,ifour) - del;
  mfit(ione,ithree) = mfit(ione,ithree) + del;
  mfit(itwo,ifour) = mfit(itwo,ifour) + del;

  work(indexll+1) = del;
  work(indexll+2) = -del;
  work(indexll+3) = -del;
  work(indexll+4) = del;

  cr = cr + abs(p1 - mfit(ione,ithree)) + ...
     abs(p2 - mfit(ione,ifour)) + ...
     abs(p3 - mfit(itwo,ithree)) + ...
     abs(p4 - mfit(itwo,ifour));

  indexll = indexll + 4;


elseif((((inperm(ione) <= nrow) & (inperm(itwo) > nrow)) ...
  & ((inperm(ithree) <= nrow) & (inperm(ifour) ...
  > nrow))) | ...
  (((inperm(ione) > nrow) & (inperm(itwo) <= nrow)) ...
  & ((inperm(ithree) > nrow) & (inperm(ifour) ...
  <= nrow))))

  p1 = mfit(ione,itwo);
  p2 = mfit(ione,ifour);
  p3 = mfit(itwo,ithree);
  p4 = mfit(ithree,ifour);

  mfit(ione,itwo) = mfit(ione,itwo) - work(indexll+1);
  mfit(ione,ifour) = mfit(ione,ifour) - work(indexll+2);
  mfit(itwo,ithree) = mfit(itwo,ithree) - work(indexll+3);
  mfit(ithree,ifour) = mfit(ithree,ifour) - work(indexll+4);

  del = (mfit(ione,itwo) + mfit(itwo,ithree) + ...
     mfit(ithree,ifour) - mfit(ione,ifour))/4.0;

  mfit(itwo,ithree) = mfit(itwo,ithree) - del;
  mfit(ione,ifour) = mfit(ione,ifour) + del;
  mfit(ione,itwo) = mfit(ione,itwo) - del;
  mfit(ithree,ifour) = mfit(ithree,ifour) - del;

  work(indexll+1) = -del;
  work(indexll+2) = del;
```

```
         work(indexll+3) = -del;
         work(indexll+4) = -del;

         cr = cr + abs(p1 - mfit(ione,itwo)) + ...
            abs(p2 - mfit(ione,ifour)) + ...
            abs(p3 - mfit(itwo,ithree)) + ...
            abs(p4 - mfit(ithree,ifour));

         indexll = indexll + 4;

        elseif((((inperm(ione) <= nrow) & (inperm(itwo) > nrow)) ...
         & ((inperm(ithree) > nrow) & (inperm(ifour) ...
         <= nrow))) | ...
         (((inperm(ione) > nrow) & (inperm(itwo) <= nrow)) ...
         & ((inperm(ithree) <= nrow) & (inperm(ifour) ...
         > nrow))))

         p1 = mfit(ione,itwo);
         p2 = mfit(ione,ithree);
         p3 = mfit(itwo,ifour);
         p4 = mfit(ithree,ifour);

         mfit(ione,itwo) = mfit(ione,itwo) - work(indexll+1);
         mfit(ione,ithree) = mfit(ione,ithree) - work(indexll+2);
         mfit(itwo,ifour) = mfit(itwo,ifour) - work(indexll+3);
         mfit(ithree,ifour) = mfit(ithree,ifour) - work(indexll+4);

         del = (mfit(ione,itwo) + mfit(itwo,ifour) - ...
            mfit(ione,ithree) - mfit(ithree,ifour))/4.0;

         mfit(ione,itwo) = mfit(ione,itwo) - del;
         mfit(itwo,ifour) = mfit(itwo,ifour) - del;
         mfit(ione,ithree) = mfit(ione,ithree) + del;
         mfit(ithree,ifour) = mfit(ithree,ifour) + del;

         work(indexll+1) = -del;
         work(indexll+2) = del;
         work(indexll+3) = -del;
         work(indexll+4) = del;

         cr = cr + abs(p1 - mfit(ione,itwo)) + ...
            abs(p2 - mfit(ione,ithree)) + ...
            abs(p3 - mfit(itwo,ifour)) + ...
            abs(p4 - mfit(ithree,ifour));

         indexll = indexll + 4;

      end

   end
end
```

```
   end
end

for ione = 1:(n-2)
   for itwo = (ione+1):(n-1)
      for ithree = (itwo+1):n

            if((((inperm(ione) <= nrow) & (inperm(itwo) <= nrow)) ...
                  & ((inperm(ithree) > nrow) )) | ...
                  (((inperm(ione) > nrow) & (inperm(itwo) > nrow)) ...
                  & ((inperm(ithree) <= nrow) )))

               p1 = mfit(ione,ithree);
               p2 = mfit(itwo,ithree);

               mfit(ione,ithree) = mfit(ione,ithree) - work(indexll+1);
               mfit(itwo,ithree) = mfit(itwo,ithree) - work(indexll+2);

               if (mfit(ione,ithree) >= mfit(itwo,ithree))
                   del = 0;
               else
                   del = (mfit(itwo,ithree) - mfit(ione,ithree))/2;
               end


               mfit(ione,ithree) = mfit(ione,ithree) + del;
               mfit(itwo,ithree) = mfit(itwo,ithree) - del;


               work(indexll+1) = del;
               work(indexll+2) = -del;


               cr = cr + abs(p1 - mfit(ione,ithree)) + ...
                  abs(p2 - mfit(itwo,ithree));

              indexll = indexll + 2;


             elseif((((inperm(ione) <= nrow) & (inperm(itwo) > nrow)) ...
               & ((inperm(ithree) > nrow) )) | ...
               (((inperm(ione) > nrow) & (inperm(itwo) <= nrow)) ...
               & ((inperm(ithree) <= nrow) )))

               p1 = mfit(ione,itwo);
               p2 = mfit(ione,ithree);


               mfit(ione,itwo) = mfit(ione,itwo) - work(indexll+1);
               mfit(ione,ithree) = mfit(ione,ithree) - work(indexll+2);
```

```
                if (mfit(ione,ithree) >= mfit(ione,itwo))
                    del = 0;
                else
                    del = (mfit(ione,itwo) - mfit(ione,ithree))/2;
                end


                mfit(ione,ithree) = mfit(ione,ithree) + del;
                mfit(ione,itwo) = mfit(ione,itwo) - del;


                work(indexll+1) = -del;
                work(indexll+2) = del;


                cr = cr + abs(p1 - mfit(ione,itwo)) + ...
                   abs(p2 - mfit(ione,ithree));

               indexll = indexll + 2;



            end

         end
      end
   end






for ione = 1:(n-1)
   for itwo = (ione+1):n

      if(((inperm(ione) <= nrow) & (inperm(itwo) > nrow)) | ...
            ((inperm(ione) > nrow) & (inperm(itwo) <= nrow)))

         p1 = mfit(ione,itwo);

         mfit(ione,itwo) = mfit(ione,itwo) - work( indexll+1);

         if(mfit(ione,itwo) < 0.0)

            work(indexll+1) - mfit(ione,itwo);
            mfit(ione,itwo) = 0.0;
         end

         indexll = indexll + 1;

         cr = cr + abs(p1 - mfit(ione,itwo));
         end
      end

   end
end



   for i = 1:(n-1)
      for j = (i+1):n

         mfit(j,i) = mfit(i,j);

      end
   end

   rowidx = 0;

   for i = 1:n

      if(inperm(i) <= nrow)

         rowidx = rowidx + 1;

         colidx = 0;

         for j = 1:n

            if(inperm(j) > nrow)

               colidx = colidx + 1;
               fit(rowidx,colidx) = mfit(i,j);
            end
         end
      end
   end

   rowperm = zeros(nrow,1);
   colperm = zeros(ncol,1);
   rowidx = 0;
   colidx = 0;

   for i = 1:n

      if(inperm(i) <= nrow)

         rowidx = rowidx + 1;
         rowperm(rowidx) = inperm(i);

      elseif(inperm(i) > nrow)
```

```
         colidx = colidx + 1;
         colperm(colidx) = inperm(i) - nrow;

      end
   end

   aveprox = sum(sum(proxtm))/(nrow*ncol);

   diff = sum(sum((proxtm(rowperm,colperm) - fit).^2));
   % denom = sum(sum((proxtm(rowperm,colperm) - aveprox).^2));

   % vaf = 1 - (diff/denom);
```

A.17 linfittmac.m

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

%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.
%  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).


[nrow ncol] = size(proxtm);
n = nrow + ncol;
tempfit = zeros(n,n);
fit = zeros(nrow,ncol);
mfit = zeros(n,n);

for irow = 1:nrow
   for icol = 1:ncol

      tempfit(irow,icol+nrow) = proxtm(irow,icol);
      tempfit(icol+nrow,irow) = proxtm(irow,icol);

   end
end

acondiff = 1.0;
addcon = 0.0;
```

```
while (acondiff >= 1.0e-005)

  for i =1:n
      for j=1:n

         if(i ~= j)
            mfit(i,j) = tempfit(inperm(i),inperm(j)) + addcon;
         else
            mfit(i,j) = 0.0;
         end
      end
   end

cr = 1.0;
work = zeros(n*(n-1)*(n-2)*(n-3),1);
addconpv = addcon;

while(cr >= 1.0e-005)

   cr = 0.0;
   indexll=0;

   for ione = 1:(n-3)
      for itwo = (ione+1):(n-2)
         for ithree = (itwo+1):(n-1)
            for ifour = (ithree+1):n

               if((((inperm(ione) <= nrow) & (inperm(itwo) <= nrow)) ...
                     & ((inperm(ithree) > nrow) & (inperm(ifour) ...
                     > nrow))) | ...
                     (((inperm(ione) > nrow) & (inperm(itwo) > nrow)) ...
                     & ((inperm(ithree) <= nrow) & (inperm(ifour) ...
                     <= nrow))))

                  p1 = mfit(ione,ithree);
                  p2 = mfit(ione,ifour);
                  p3 = mfit(itwo,ithree);
                  p4 = mfit(itwo,ifour);

                  mfit(ione,ithree) = mfit(ione,ithree) - work(indexll+1);
                  mfit(ione,ifour) = mfit(ione,ifour) - work(indexll+2);
                  mfit(itwo,ithree) = mfit(itwo,ithree) - work(indexll+3);
                  mfit(itwo,ifour) = mfit(itwo,ifour) - work(indexll+4);

                  del = (mfit(itwo,ithree) + mfit(ione,ifour) - ...
                     mfit(ione,ithree) - mfit(itwo,ifour))/4.0;

                  mfit(itwo,ithree) = mfit(itwo,ithree) - del;
                  mfit(ione,ifour) = mfit(ione,ifour) - del;
                  mfit(ione,ithree) = mfit(ione,ithree) + del;
                  mfit(itwo,ifour) = mfit(itwo,ifour) + del;
                  work(indexll+1) = del;
                  work(indexll+2) = -del;
                  work(indexll+3) = -del;
                  work(indexll+4) = del;

                  cr = cr + abs(p1 - mfit(ione,ithree)) + ...
                     abs(p2 - mfit(ione,ifour)) + ...
                     abs(p3 - mfit(itwo,ithree)) + ...
                     abs(p4 - mfit(itwo,ifour));

                  indexll = indexll + 4;


                elseif((((inperm(ione) <= nrow) & (inperm(itwo) > nrow)) ...
                  & ((inperm(ithree) <= nrow) & (inperm(ifour) ...
                  > nrow))) | ...
                  (((inperm(ione) > nrow) & (inperm(itwo) <= nrow)) ...
                  & ((inperm(ithree) > nrow) & (inperm(ifour) ...
                  <= nrow))))

                  p1 = mfit(ione,itwo);
                  p2 = mfit(ione,ifour);
                  p3 = mfit(itwo,ithree);
                  p4 = mfit(ithree,ifour);

                  mfit(ione,itwo) = mfit(ione,itwo) - work(indexll+1);
                  mfit(ione,ifour) = mfit(ione,ifour) - work(indexll+2);
                  mfit(itwo,ithree) = mfit(itwo,ithree) - work(indexll+3);
                  mfit(ithree,ifour) = mfit(ithree,ifour) - work(indexll+4);

                  del = (mfit(ione,itwo) + mfit(itwo,ithree) + ...
                     mfit(ithree,ifour) - mfit(ione,ifour))/4.0;

                  mfit(itwo,ithree) = mfit(itwo,ithree) - del;
                  mfit(ione,ifour) = mfit(ione,ifour) + del;
                  mfit(ione,itwo) = mfit(ione,itwo) - del;
                  mfit(ithree,ifour) = mfit(ithree,ifour) - del;

                  work(indexll+1) = -del;
                  work(indexll+2) = del;
                  work(indexll+3) = -del;
                  work(indexll+4) = -del;

                  cr = cr + abs(p1 - mfit(ione,itwo)) + ...
                     abs(p2 - mfit(ione,ifour)) + ...
                     abs(p3 - mfit(itwo,ithree)) + ...
                     abs(p4 - mfit(ithree,ifour));

                  indexll = indexll + 4;
```

```
            elseif((((inperm(ione) <= nrow) & (inperm(itwo) > nrow)) ...
              & ((inperm(ithree) > nrow) & (inperm(ifour) ...
              <= nrow))) | ...
              (((inperm(ione) > nrow) & (inperm(itwo) <= nrow)) ...
              & ((inperm(ithree) <= nrow) & (inperm(ifour) ...
              > nrow))))

              p1 = mfit(ione,itwo);
              p2 = mfit(ione,ithree);
              p3 = mfit(itwo,ifour);
              p4 = mfit(ithree,ifour);

              mfit(ione,itwo) = mfit(ione,itwo) - work(indexll+1);
              mfit(ione,ithree) = mfit(ione,ithree) - work(indexll+2);
              mfit(itwo,ifour) = mfit(itwo,ifour) - work(indexll+3);
              mfit(ithree,ifour) = mfit(ithree,ifour) - work(indexll+4);

              del = (mfit(ione,itwo) + mfit(itwo,ifour) - ...
                 mfit(ione,ithree) - mfit(ithree,ifour))/4.0;

              mfit(ione,itwo) = mfit(ione,itwo) - del;
              mfit(itwo,ifour) = mfit(itwo,ifour) - del;
              mfit(ione,ithree) = mfit(ione,ithree) + del;
              mfit(ithree,ifour) = mfit(ithree,ifour) + del;

              work(indexll+1) = -del;
              work(indexll+2) = del;
              work(indexll+3) = -del;
              work(indexll+4) = del;

              cr = cr + abs(p1 - mfit(ione,itwo)) + ...
                 abs(p2 - mfit(ione,ithree)) + ...
                 abs(p3 - mfit(itwo,ifour)) + ...
                 abs(p4 - mfit(ithree,ifour));

              indexll = indexll + 4;

            end

         end
      end
   end
end

for ione = 1:(n-2)
   for itwo = (ione+1):(n-1)
      for ithree = (itwo+1):n

            if((((inperm(ione) <= nrow) & (inperm(itwo) <= nrow)) ...
                  & ((inperm(ithree) > nrow) )) | ...
                  (((inperm(ione) > nrow) & (inperm(itwo) > nrow)) ...
                 & ((inperm(ithree) <= nrow) )))

              p1 = mfit(ione,ithree);
              p2 = mfit(itwo,ithree);

              mfit(ione,ithree) = mfit(ione,ithree) - work(indexll+1);
              mfit(itwo,ithree) = mfit(itwo,ithree) - work(indexll+2);

              if (mfit(ione,ithree) >= mfit(itwo,ithree))
                  del = 0;
              else
                  del = (mfit(itwo,ithree) - mfit(ione,ithree))/2;
              end


              mfit(ione,ithree) = mfit(ione,ithree) + del;
              mfit(itwo,ithree) = mfit(itwo,ithree) - del;


              work(indexll+1) = del;
              work(indexll+2) = -del;


              cr = cr + abs(p1 - mfit(ione,ithree)) + ...
                 abs(p2 - mfit(itwo,ithree));

             indexll = indexll + 2;


            elseif((((inperm(ione) <= nrow) & (inperm(itwo) > nrow)) ...
              & ((inperm(ithree) > nrow) )) | ...
              (((inperm(ione) > nrow) & (inperm(itwo) <= nrow)) ...
              & ((inperm(ithree) <= nrow) )))

              p1 = mfit(ione,itwo);
              p2 = mfit(ione,ithree);


              mfit(ione,itwo) = mfit(ione,itwo) - work(indexll+1);
              mfit(ione,ithree) = mfit(ione,ithree) - work(indexll+2);

              if (mfit(ione,ithree) >= mfit(ione,itwo))
                  del = 0;
              else
                  del = (mfit(ione,itwo) - mfit(ione,ithree))/2;
              end


              mfit(ione,ithree) = mfit(ione,ithree) + del;
              mfit(ione,itwo) = mfit(ione,itwo) - del;
```

```
                  work(indexll+1) = -del;
                  work(indexll+2) = del;


                  cr = cr + abs(p1 - mfit(ione,itwo)) + ...
                     abs(p2 - mfit(ione,ithree));

                 indexll = indexll + 2;



               end

            end
         end
      end





   for ione = 1:(n-1)
      for itwo = (ione+1):n

         if(((inperm(ione) <= nrow) & (inperm(itwo) > nrow)) | ...
               ((inperm(ione) > nrow) & (inperm(itwo) <= nrow)))

            p1 = mfit(ione,itwo);

            mfit(ione,itwo) = mfit(ione,itwo) - work( indexll+1);

            if(mfit(ione,itwo) < 0.0)

               work(indexll+1) - mfit(ione,itwo);
               mfit(ione,itwo) = 0.0;
            end

            indexll = indexll + 1;

            cr = cr + abs(p1 - mfit(ione,itwo));
         end
      end

   end
end



   for i = 1:(n-1)
      for j = (i+1):n
         mfit(j,i) = mfit(i,j);

      end
   end

   rowidx = 0;

   for i = 1:n

      if(inperm(i) <= nrow)

         rowidx = rowidx + 1;

         colidx = 0;

         for j = 1:n

            if(inperm(j) > nrow)

               colidx = colidx + 1;
               fit(rowidx,colidx) = mfit(i,j);
            end
         end
      end
   end

   rowperm = zeros(nrow,1);
   colperm = zeros(ncol,1);
   rowidx = 0;
   colidx = 0;

   for i = 1:n

      if(inperm(i) <= nrow)

         rowidx = rowidx + 1;
         rowperm(rowidx) = inperm(i);

      elseif(inperm(i) > nrow)

         colidx = colidx + 1;
         colperm(colidx) = inperm(i) - nrow;

      end
   end


   addcon = -sum(sum(proxtm(rowperm,colperm) - fit))/(nrow*ncol);

   acondiff = abs(addcon - addconpv);
```

```
   aveprox = sum(sum(proxtm))/(nrow*ncol);

   diff = sum(sum((proxtm(rowperm,colperm) - (fit-addcon)).^2));
   denom = sum(sum((proxtm(rowperm,colperm) - aveprox).^2));

   vaf = 1 - (diff/denom);

end
```

A.18 proxstdtm.m

```
function [stanproxtm, stanproxmulttm] = proxstdtm(proxtm,mean)

%PROXSTDTM produces a standardized two-mode proximity matrix (STANPROXTM) from the input
%  $nrow \times ncol$ two-mode proximity matrix (PROXTM) with a dissimilarity
%  interpretation.
%  STANPROXTM entries have unit variance (standard deviation of one) with a
%  mean of MEAN given as an input number;
%  STANPROXMULTTM entries have a sum of squares equal to
%  $nrow*rcol$.

[nrow ncol]  = size(proxtm);
aveprox = sum(sum(proxtm))/(nrow*ncol);
sumprxsq = sum(sum(proxtm.^2));
stddev = sqrt(((1/(nrow*ncol))*sumprxsq) - (aveprox*aveprox));
stanproxtm = zeros(nrow,ncol);
stanproxmulttm = zeros(nrow,ncol);

for i = 1:nrow
   for j = 1:ncol

         stanproxtm(i,j) = ((proxtm(i,j) - aveprox)/stddev) + mean;
         stanproxmulttm(i,j) = (nrow*ncol)*(proxtm(i,j)/sqrt(sumprxsq));

      end
   end
end
```

A.19 proxrandtm.m

```
function [randproxtm] = proxrandtm(proxtm)

%PROXRANDTM produces a two-mode proximity matrix having
%  entries that are a random permutation of those in the two-mode input proximity
%  matrix PROXTM.

[nrow ncol] = size(proxtm);
```

```
change = randperm(nrow*ncol);
randproxtm = proxtm;

for i = 1:nrow
   for j = 1:ncol

      k = i + j;

      for ione = 1:nrow
         for jone = 1:ncol

            kk = ione + jone;

            if(change(k) == kk)

               temp = randproxtm(i,j);
               randproxtm(i,j) = randproxtm(ione,jone);
               randproxtm(ione,jone) = temp;

            end
         end
      end

   end
end
```

A.20 proxmontm.m

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

%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).
%  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.

[nrow ncol] = size(proxpermuttm);
n = nrow + ncol;
work = zeros(n*(n-1)*n*(n-1),1);
targ = proxpermuttm;
fit = fittedtm;
cr = 1.0;

while (cr >= 1.0e-006)



   cr = 0.0;
   indexll = 0;
```

```
for jone = 1:nrow
   for jtwo = 1:ncol
      for jthree = 1:nrow
         for jfour = 1:ncol

            if((jone ~= jthree) | (jtwo ~= jfour))

               p1 = fit(jone,jtwo);
               p2 = fit(jthree,jfour);
               fit(jone,jtwo) = fit(jone,jtwo) - work(indexll+1);
               fit(jthree,jfour) = fit(jthree,jfour) - work(indexll+2);


               if((abs(targ(jone,jtwo) - targ(jthree,jfour)) ...
                     > 1.0e-006) & (targ(jone,jtwo) < ...
                     targ(jthree,jfour)))

                  if(fit(jone,jtwo) <= fit(jthree,jfour))

                     work(indexll+1) = 0;
                     work(indexll+2) = 0;



                  elseif(fit(jone,jtwo) > fit(jthree,jfour))

                     ave = (fit(jone,jtwo) + fit(jthree,jfour))/2.0;
                     work(indexll+1) = ave - fit(jone,jtwo);
                     work(indexll+2) = ave - fit(jthree,jfour);

                     fit(jone,jtwo) = ave;
                     fit(jthree,jfour) = ave;



                  end

               elseif((abs(targ(jone,jtwo) - targ(jthree,jfour)) ...
                     > 1.0e-006) & (targ(jone,jtwo) > ...
                     targ(jthree,jfour)))

                  if(fit(jone,jtwo) >= fit(jthree,jfour))

                     work(indexll+1) = 0;
                     work(indexll+2) = 0;



                  elseif(fit(jone,jtwo) < fit(jthree,jfour))
```

```
                        ave = (fit(jone,jtwo) + fit(jthree,jfour))/2.0;
                        work(indexll+1) = ave - fit(jone,jtwo);
                        work(indexll+2) = ave - fit(jthree,jfour);
                        fit(jone,jtwo) = ave;
                        fit(jthree,jfour) = ave;



                     end
                  end

               cr = cr + abs(p1-fit(jone,jtwo)) + ...
                  abs(p2-fit(jthree,jfour));
            end

               indexll = indexll + 2;

            end
         end
      end
   end
end




avefit = sum(sum(fit))/(nrow*ncol);

diff = sum(sum((fit - fittedtm).^2));

denom = sum(sum((fit - avefit).^2));

vaf = 1 - (diff/denom);

monproxpermuttm = fit;
```