Contents

List of Tables

List of Figures

8 Ultrametric Extensions by Fitting Partitions Containing Contiguous Subsets

The M-file, `partitionfit.m`, is a very general routine giving a least-squares approximation to a proximity matrix based on a given collection of partitions. Thus, no matter how the set of candidate partitions might be chosen, a least-squares fitted matrix to the given proximity matrix is achieved. For example, if we simply use the nested partitions constructed from an ultrametric, the ultrametric would be retrieved when the latter is used as the input proximity matrix. In this section, we show how `partitionfit.m` can also be used to select partitions from a predefined set (this selection is done by those partitions assigned strictly positive weights) that might serve to reconstruct the proximity matrix well. The M-file, `consec_subsetfit.m`, defines $(n(n-1)/2)-1$ candidate partitions each characterized by a single contiguous cluster of objects, with all objects before and after this contiguous set forming individual clusters of the partition (the minus 1 appears in the count due to the (conjoint) partition defined by a single contiguous set being excluded). The M-file, `consec_subsetfit_alter.m`, varies the specific definition of the partitions by including all objects before and all objects after the contiguous set (when nonempty) as separate individual clusters of the partitions.

As can be seen from the verbatim output provided below, the nonnegative weighted partitions from `consec_subsetfit.m`, producing a fitted matrix with VAF of 92.61%, are as follows:

Partition	Partition Increment
{{St,Br,Gi,So},{Oc},{Ke},{Re},{Sc},{Th}}	.1939
{{St,Br,Gi,So,Oc},{Ke},{Re},{Sc},{Th}}	.0300
{{St,Br,Gi,So,Oc,Ke},{Re},{Sc},{Th}}	.0389
{{St,Br,Gi,So,Oc,Ke,Re},{Sc},{Th}}	.1315
{{St},{Br,Gi,So,Oc,Ke,Re,Sc,Th}}	.1152
{{St},{Br},{Gi,So,Oc,Ke,Re,Sc,Th}}	.0052
{{St},{Br},{Gi},{So,Oc,Ke,Re,Sc,Th}}	.0153
{{St},{Br},{Gi},{So},{Oc,Ke,Re,Sc,Th}}	.2220
{{St},{Br},{Gi},{So},{Oc},{Ke,Re,Sc,Th}}	.0633
{{St},{Br},{Gi},{So},{Oc},{Ke},{Re,Sc,Th}}	.0030

Similarly, we have a very high VAF of 98.12% based on the more numerous partitions generated from `consec_subsetfit_alter.m`:

Partition	Partition Increment
{{St,Br},{Gi,So,Oc,Ke,Re,Sc,Th}}	.0021
{{St,Br,Gi},{So,Oc,Ke,Re,Sc,Th}}	.0001
{{St,Br,Gi,So},{Oc,Ke,Re,Sc,Th}}	.0001
{{St,Br,Gi,So,Oc,Ke},{Re,Sc,Th}}	.0100
{{St,Br,Gi,So,Oc,Ke,Re},{Sc,Th}}	.1218
{{St},{Br,Gi},{So,Oc,Ke,Re,Sc,Th}}	.0034
{{St},{Br,Gi,So,Oc},{Ke,Re,Sc,Th}}	.0056
{{St},{Br,Gi,So,Oc,Ke,Re},{Sc,Th}}	.0113
{{St},{Br,Gi,So,Oc,Ke,Re,Sc},{Th}}	.0038
{{St},{Br,Gi,So,Oc,Ke,Re,Sc,Th}}	.1170
{{St,Br},{Gi,So},{Oc,Ke,Re,Sc,Th}}	.0165
{{St,Br},{Gi,So,Oc,Ke,Re,Sc,Th}}	.0095
{{St,Br,Gi},{So,Oc},{Ke,Re,Sc,Th}}	.0197
{{St,Br,Gi},{So,Oc,Ke,Re,Sc,Th}}	.0115
{{St,Br,Gi,So},{Oc,Ke,Re,Sc,Th}}	.2294
{{St,Br,Gi,So,Oc},{Ke,Re,Sc,Th}}	.0353
{{St,Br,Gi,So,Oc,Ke},{Re,Sc,Th}}	.0400
{{St,Br,Gi,So,Oc,Ke,Re},{Sc,Th}}	.0132
{{St},{Br},{Gi},{So},{Oc},{Ke},{Re},{Sc},{Th}}	.2050

```
>> load supreme_agree.dat
>> [fitted,vaf,weights,end_condition,member] = consec_subsetfit(supreme_agree);
>> fitted

fitted =

         0    0.4239    0.4239    0.4239    0.6178    0.6478    0.6866    0.8181    0.8181
    0.4239         0    0.3087    0.3087    0.5026    0.5326    0.5715    0.7029    0.7029
    0.4239    0.3087         0    0.3035    0.4974    0.5274    0.5663    0.6977    0.6977
    0.4239    0.3087    0.3035         0    0.4821    0.5121    0.5510    0.6824    0.6824
    0.6178    0.5026    0.4974    0.4821         0    0.2901    0.3290    0.4604    0.4604
    0.6478    0.5326    0.5274    0.5121    0.2901         0    0.2657    0.3972    0.3972
    0.6866    0.5715    0.5663    0.5510    0.3290    0.2657         0    0.3942    0.3942
    0.8181    0.7029    0.6977    0.6824    0.4604    0.3972    0.3942         0    0.3942
    0.8181    0.7029    0.6977    0.6824    0.4604    0.3972    0.3942    0.3942         0

>> vaf

vaf =

    0.9261

>> weights

weights =

         0
         0
    0.1939
    0.0300
    0.0389
    0.1315
         0
         0
         0
         0
         0
         0
         0
    0.1152
         0
         0
         0
```

```
         0
         0
    0.0052
         0
         0
         0
         0
    0.0153
         0
         0
         0
    0.2220
         0
         0
    0.0633
         0
    0.0030
         0
         0

>> end_condition

end_condition =

     0

>> member

member =

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

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


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

fitted =

         0    0.3460    0.3740    0.4053    0.6347    0.6700    0.7200    0.8550    0.8550
    0.3460         0    0.2330    0.2677    0.4971    0.5380    0.5880    0.7342    0.7380
    0.3740    0.2330         0    0.2396    0.4855    0.5264    0.5764    0.7227    0.7264
    0.4053    0.2677    0.2396         0    0.4509    0.5114    0.5614    0.7076    0.7114
    0.6347    0.4971    0.4855    0.4509         0    0.2655    0.3155    0.4617    0.4655
    0.6700    0.5380    0.5264    0.5114    0.2655         0    0.2550    0.4012    0.4050
    0.7200    0.5880    0.5764    0.5614    0.3155    0.2550         0    0.3512    0.3550
    0.8550    0.7342    0.7227    0.7076    0.4617    0.4012    0.3512         0    0.2087
    0.8550    0.7380    0.7264    0.7114    0.4655    0.4050    0.3550    0.2087         0


vaf =

    0.9812


weights =

    0.0021
    0.0001
    0.0001
         0
    0.0100
    0.1218
         0
    0.0034
         0
    0.0056
         0
    0.0113
    0.0038
    0.1170
    0.0165
         0
         0
         0
         0
    0.0095
    0.0197
         0
         0
         0
    0.0115
         0
         0
         0
    0.2294
         0
         0
    0.0353
         0
    0.0400
    0.0132
    0.2050
```

```
end_condition =

     0


member =

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

To see how well one might do by choosing only eight partitions (i.e., the same number needed to define the order-constrained best-fitting ultrametric), the single (disjoint) partition defined by nine separate classes is chosen in both instances, plus the seven partitions having the highest assigned positive weights. For those picked from the partition pool identified by `consec_subsetfit.m`, the VAF drops slightly from 92.61% to 92.51% based on the partitions:

Partition	Partition Increment
{{St,Br,Gi,So},{Oc},{Ke},{Re},{Sc},{Th}}	.1923
{{St,Br,Gi,So,Oc},{Ke},{Re},{Sc},{Th}}	.0301
{{St,Br,Gi,So,Oc,Ke},{Re},{Sc},{Th}}	.0396
{{St,Br,Gi,So,Oc,Ke,Re},{Sc},{Th}}	.1316
{{St},{Br,Gi,So,Oc,Ke,Re,Sc,Th}}	.1224
{{St},{Br},{Gi},{So},{Oc,Ke,Re,Sc,Th}}	.2250
{{St},{Br},{Gi},{So},{Oc},{Ke,Re,Sc,Th}}	.0671
{{St},{Br},{Gi},{So},{Oc},{Ke},{Re},{Sc},{Th}}	.0000

For those selected from the set generated by `consec_subsetfit_alter.m`, the VAF again drops slightly from 98.12% to 97.97%. But in some absolute sense given the size of the VAF, the eight partitions listed below seem to be about all that can be extracted from this particular justice data set.

Partition	Partition Increment
{{St,Br,Gi,So,Oc,Ke,Re},{Sc,Th}}	.1466
{{St},{Br,Gi,So,Oc,Ke,Re,Sc,Th}}	.1399
{{St,Br},{Gi,So},{Oc,Ke,Re,Sc,Th}}	.0287
{{St,Br,Gi},{So,Oc},{Ke,Re,Sc,Th}}	.0326
{{St,Br,Gi,So},{Oc,Ke,Re,Sc,Th}}	.2269
{{St,Br,Gi,So,Oc},{Ke,Re,Sc,Th}}	.0316
{{St,Br,Gi,So,Oc,Ke},{Re,Sc,Th}}	.0500
{{St},{Br},{Gi},{So},{Oc},{Ke},{Re},{Sc},{Th}}	.2051

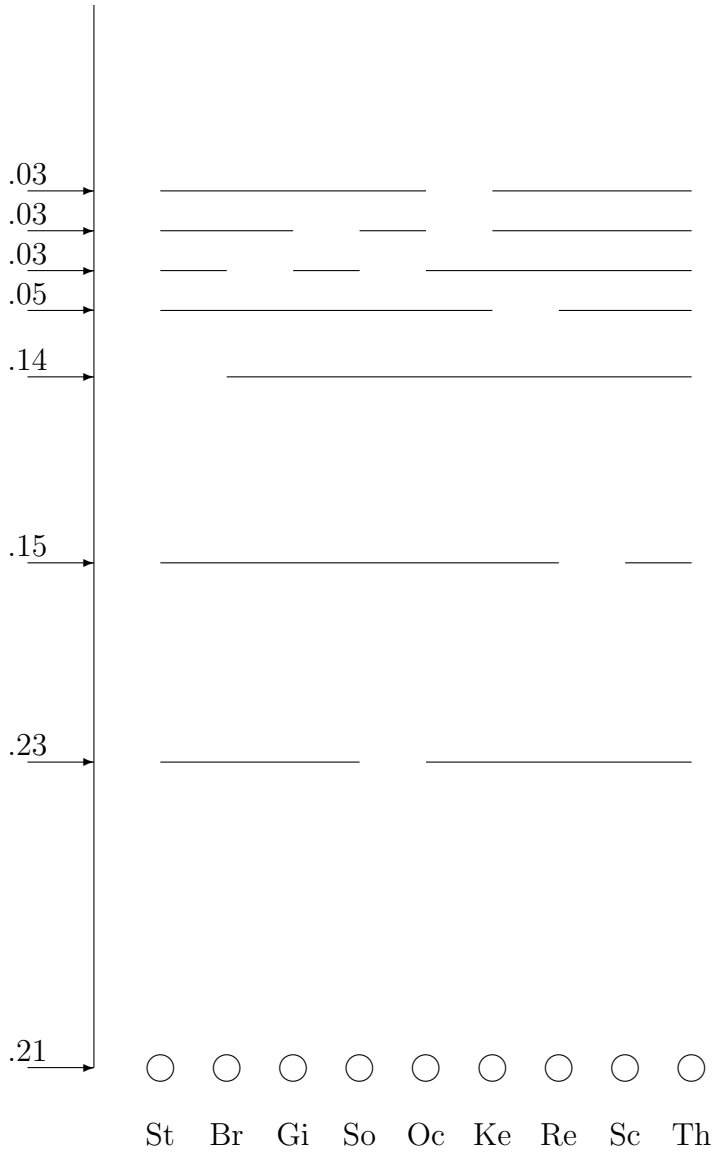
The three highest weighted partitions have very clear interpretations: {Sc,Th} versus the rest; {St} versus the rest; {St,Br,Gi,So} as the left versus {Oc,Ke,Re,Sc,Th} as the right. The few remaining partitions revolve around several other less salient (adjacent) object pairings that are also very interpretable in relation to the object ordering from liberal to conservative. We give a graphical representation for this latter culled collection of partitions in Figure 6. Again, the partition increments are not included in a fitted value whenever a continuous horizontal line encompasses the relevant objects in a defining cluster of the partition.

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

member =

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

Figure 6: A Representation for the Fitted Values of the (Generalized) Structure Described in the Text (Having VAF of 97.97%)



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

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

fitted =

         0    0.4245    0.4245    0.4245    0.6168    0.6469    0.6865    0.8181    0.8181
    0.4245         0    0.3021    0.3021    0.4944    0.5245    0.5641    0.6957    0.6957
    0.4245    0.3021         0    0.3021    0.4944    0.5245    0.5641    0.6957    0.6957
    0.4245    0.3021    0.3021         0    0.4944    0.5245    0.5641    0.6957    0.6957
    0.6168    0.4944    0.4944    0.4944         0    0.2895    0.3291    0.4607    0.4607
    0.6469    0.5245    0.5245    0.5245    0.2895         0    0.2620    0.3936    0.3936
    0.6865    0.5641    0.5641    0.5641    0.3291    0.2620         0    0.3936    0.3936
    0.8181    0.6957    0.6957    0.6957    0.4607    0.3936    0.3936         0    0.3936
    0.8181    0.6957    0.6957    0.6957    0.4607    0.3936    0.3936    0.3936         0


vaf =

    0.9251


weights =

    0.1923
    0.0301
    0.0396
    0.1316
    0.1224
    0.2350
    0.0671
         0


end_condition =

     0

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

member =

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

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

fitted =

         0    0.3450    0.3736    0.4062    0.6331    0.6647    0.7147    0.8613    0.8613
    0.3450         0    0.2337    0.2664    0.4933    0.5248    0.5748    0.7215    0.7215
    0.3736    0.2337         0    0.2377    0.4933    0.5248    0.5748    0.7215    0.7215
    0.4062    0.2664    0.2377         0    0.4606    0.5248    0.5748    0.7215    0.7215
    0.6331    0.4933    0.4933    0.4606         0    0.2693    0.3193    0.4659    0.4659
    0.6647    0.5248    0.5248    0.5248    0.2693         0    0.2551    0.4017    0.4017
    0.7147    0.5748    0.5748    0.5748    0.3193    0.2551         0    0.3517    0.3517
```

```
    0.8613    0.7215    0.7215    0.7215    0.4659    0.4017    0.3517         0    0.2051
    0.8613    0.7215    0.7215    0.7215    0.4659    0.4017    0.3517    0.2051         0


vaf =

    0.9797


weights =

    0.1466
    0.1399
    0.0287
    0.0326
    0.2269
    0.0316
    0.0500
    0.2051


end_condition =

     0
```

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

The proximity data considered thus far for obtaining some type of structure, 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, and 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, given by an $n_a \times n_b$ proximity matrix $\mathbf{Q} = \{q_{rs}\}$. Again, we assume that the entries in $\mathbf{Q}$ are keyed as dissimilarities, and a joint structural representation is desired for the combined set $S_A \cup S_B$. We might caution at the outset of the need to have legitimate proximities to make the analyses to follow very worthwhile or interpretable. There are many numerical elicitation schemes where subjects (e.g., raters) are asked to respond to some set of objects (e.g., items). If the elicitation is for, say, preference, then proximity may be a good interpretation for the numerical values. If, on the other hand, the numerical value is merely a rating given on some more-or-less objective criterion where only errors of observation induce

the variability from rater to rater, then probably not.

Conditions have been proposed in the literature for when the entries in a matrix fitted to **Q** characterize an ultrametric or an additive tree representation. In particular, suppose an $n_a \times n_b$ matrix, $\mathbf{F} = \{f_{rs}\}$, is fitted to **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 **F** corresponding to the pairs, (O_{rA}, O_{rB}), (O_{rA}, O_{sB}), (O_{sA} O_{rB}), and (O_{sA}, O_{sB}), say $f_{r_A r_B}$, $f_{r_A s_B}$, $f_{s_A r_B}$, $f_{s_A s_B}$, 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 **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 $f_{r_A r_B}$, $f_{r_A s_B}$, $f_{r_A t_B}$, $f_{s_A r_B}$, $f_{s_A s_B}$, $f_{s_A t_B}$, $f_{t_A r_B}$, $f_{t_A s_B}$, $f_{t_A t_B}$, respectively, the largest two of the following sums must be equal:

$f_{r_A r_B} + f_{s_A s_B} + f_{t_A t_B}$;
$f_{r_A r_B} + f_{s_A t_B} + f_{t_A s_B}$;
$f_{r_A s_B} + f_{s_A r_B} + f_{t_A t_B}$;
$f_{r_A s_B} + f_{s_A t_B} + f_{t_A r_B}$;
$f_{r_A t_B} + f_{s_A r_B} + f_{t_A s_B}$;
$f_{r_A t_B} + f_{s_A s_B} + f_{t_A r_B}$.

9.1 Two-Mode Ultrametrics

To illustrate the fitting of a given two-mode ultrametric, a two-mode target is generated by extracting a 5×4 portion from the 9×9 ultrametric target matrix, `sc_completelink_target`, used earlier. This file has contents as follows (`sc_completelink_target5x4.dat`):

```
0.3800    0.3800    0.8600    0.8600
0.2900    0.2200    0.8600    0.8600
0.8600    0.8600    0.3300    0.4600
0.8600    0.8600    0.2300    0.4600
0.8600    0.8600    0.4600    0.2100
```

The five rows correspond to the judges, St, Gi, Oc, Re, Th; the four columns to Br, So, Ke, Sc. As the two-mode 5×4 proximity matrix, the appropriate portion of the `supreme_agree` proximity matrix will be used in the fitting process; the corresponding file is called `supreme_agree5x4.dat`, with contents:

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

Because of the way the joint set of row and columns objects is numbered, the five rows are labeled from 1 to 5 and the four columns from 6 to 9. Thus, the correspondence between the justices and the numbers obviously differs from earlier applications:

1:St; 2:Gi; 3:Oc; 4:Re; 5:Th; 6:Br; 7:So; 8:Ke; 9:Sc

The M-file, `ultrafittm.m`, fits a given ultrametric to a two-mode proximity matrix, and 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 `sc_completelink_target5x4` for `targ` and `supreme_agree5x4` as `proxtm`:

```
>> load supreme_agree5x4.dat

>> load sc_completelink_target5x4.dat

>> supreme_agree5x4

supreme_agree5x4 =

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

>> sc_completelink_target5x4

sc_completelink_target5x4 =
```

```
    0.3800    0.3800    0.8600    0.8600
    0.2900    0.2200    0.8600    0.8600
    0.8600    0.8600    0.3300    0.4600
    0.8600    0.8600    0.2300    0.4600
    0.8600    0.8600    0.4600    0.2100

>> [fit,vaf] = ultrafittm(supreme_agree5x4,sc_completelink_target5x4)

fit =

    0.3350    0.3350    0.6230    0.6230
    0.2800    0.2200    0.6230    0.6230
    0.6230    0.6230    0.3300    0.4033
    0.6230    0.6230    0.2300    0.4033
    0.6230    0.6230    0.4033    0.2100


vaf =

    0.7441
```

A VAF of 74.41% was obtained for the fitted ultrametric; the easily interpretable hierarchy is given below with indications of when the partitions were formed using both the number and letter schemes to label the justices:

Partition	Level
{{4:Re,8:Ke,3:Oc,5:Th,9:Sc,1:St,2:Gi,7:So,6:Br}	.6230
{{4:Re,8:Ke,3:Oc,5:Th,9:Sc},{1:St,2:Gi,7:So,6:Br}}	.4033
{{4:Re,8:Ke,3:Oc},{5:Th,9:Sc},{1:St,2:Gi,7:So,6:Br}	.3350
{{4:Re,8:Ke,3:Oc},{5:Th,9:Sc},{1:St},{2:Gi,7:So,6:Br}}	.3300
{{4:Re,8:Ke},{3:Oc},{5:Th,9:Sc},{1:St},{2:Gi,7:So,6:Br}}	.2800
{{4:Re,8:Ke},{3:Oc},{5:Th,9:Sc},{1:St},{2:Gi,7:So},{6:Br}}	.2300
{{4:Re},{8:Ke},{3:Oc},{5:Th,9:Sc},{1:St},{2:Gi,7:So},{6:Br}}	.2200
{{4:Re},{8:Ke},{3:Oc},{5:Th,9:Sc},{1:St},{2:Gi},{7:So},{6:Br}}	.2100
{{4:Re},{8:Ke},{3:Oc},{5:Th},{9:Sc},{1:St},{2:Gi},{7:So},{6:Br}}	—

The M-file, `ultrafndtm.m` locates a best-fitting two-mode ultrametric with usage

```
[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 `supreme_agree5x4` (using random permutations for both `inpermrow` and `inpermcol`), finds exactly the same ultrametric as above with `vaf` of .7441.

```
>> [find,vaf] = ultrafndtm(supreme_agree5x4,randperm(5),randperm(4))

find =

    0.3350    0.3350    0.6230    0.6230
    0.2800    0.2200    0.6230    0.6230
    0.6230    0.6230    0.3300    0.4033
    0.6230    0.6230    0.2300    0.4033
    0.6230    0.6230    0.4033    0.2100


vaf =

    0.7441
```

9.2 Two-Mode Additive Trees

The identification of a best-fitting two-mode additive tree will be done somewhat differently than for a two-mode ultrametric representation, largely because of future storage considerations when huge matrices might be considered. 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`. Its usage is of the form

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

giving 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 u_r and v_s values 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. In the example below using the two-mode matrix, `supreme_agree5x4`, a two-mode centroid metric by itself has a (paltry) `vaf` of .1090.

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

fit =

    0.5455    0.5355    0.4975    0.5915
```

```
    0.4355    0.4255    0.3875    0.4815
    0.4255    0.4155    0.3775    0.4715
    0.4280    0.4180    0.3800    0.4740
    0.5255    0.5155    0.4775    0.5715


vaf =

    0.1090


lengths =

    0.3080
    0.1980
    0.1880
    0.1905
    0.2880
    0.2375
    0.2275
    0.1895
    0.2835
```

The identification of a two-mode additive tree with the M-file, `atreefndtm.m`, 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. The M-file has the explicit usage

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

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 `supreme_agree5x4` has `vaf` of .9953. The actual partition hierarchy is given in the next section along with an indication of when the various partitions are formed using a utility M-file that completes a two-mode ultrametric so it is defined over the entire joint set.

```
>> [find,vaf,ultra,lengths] = atreefndtm(supreme_agree5x4,randperm(5),randperm(4))

find =

    0.3000    0.3768    0.6533    0.8399
    0.2732    0.2200    0.5251    0.7117
    0.4625    0.4379    0.3017    0.4883
    0.5755    0.5510    0.2333    0.3400
    0.7488    0.7243    0.4067    0.2100


vaf =

    0.9953


ultra =

   -0.2658   -0.1644    0.1576    0.1576
   -0.1644   -0.1931    0.1576    0.1576
    0.1576    0.1576    0.0668    0.0668
    0.1576    0.1576   -0.1145   -0.1945
    0.1576    0.1576   -0.1145   -0.4978


lengths =

    0.3368
    0.2086
    0.0759
    0.1889
    0.3623
    0.2290
    0.2045
    0.1589
    0.3456
```

9.3 Completing a Two-Mode Ultrametric to One Defined on the Combined Object Set

Instead of relying only on our general intuition (and problem-solving skills) to transform a fitted two-mode ultrametric to one we could interpret directly as a sequence of partitions for the joint set $S_A \cup S_B$, the M-file, `ultracomptm.m`, provides the explicit completion of a given two-mode ultrametric matrix to a symmetric proximity matrix (defined on $S_A \cup S_B$ and satisfying the usual ultrametric constraints). Thus, this completion, in effect, estimates the (missing) ultrametric values that must be present between objects from the same cluster and from the same mode. The general syntax has the form

```
[ultracomp] = ultracomptm(ultraproxtm)
```

where `ultraproxtm` is the $n_a \times n_b$ fitted two-mode ultrametric matrix; `ultracomp` is the completed $n \times n$ proximity matrix having the usual ultrametric pattern for the complete object set of size $n = n_a + n_b$. As seen in the example below, the use of `ultrafndtm.m` on `supreme_agree5x4`, and the subsequent application of `ultracomptm.m` (plus `ultraorder.m` on `ultracomp`), leads directly to the partition hierarchy given following the verbatim output.

```
>> [ultracomp] = ultracomptm(ultra)

ultracomp =

         0   -0.1644    0.1576    0.1576    0.1576   -0.2658   -0.1644    0.1576    0.1576
   -0.1644         0    0.1576    0.1576    0.1576   -0.1644   -0.1931    0.1576    0.1576
    0.1576    0.1576         0    0.0668    0.0668    0.1576    0.1576    0.0668    0.0668
    0.1576    0.1576    0.0668         0   -0.1945    0.1576    0.1576   -0.1145   -0.1945
    0.1576    0.1576    0.0668   -0.1945         0    0.1576    0.1576   -0.1145   -0.4978
   -0.2658   -0.1644    0.1576    0.1576    0.1576         0   -0.1644    0.1576    0.1576
   -0.1644   -0.1931    0.1576    0.1576    0.1576   -0.1644         0    0.1576    0.1576
    0.1576    0.1576    0.0668   -0.1145   -0.1145    0.1576    0.1576         0   -0.1145
    0.1576    0.1576    0.0668   -0.1945   -0.4978    0.1576    0.1576   -0.1145         0

>> [orderprox,orderperm] = ultraorder(ultracomp)

orderprox =

         0   -0.2658   -0.1644   -0.1644    0.1576    0.1576    0.1576    0.1576    0.1576
   -0.2658         0   -0.1644   -0.1644    0.1576    0.1576    0.1576    0.1576    0.1576
   -0.1644   -0.1644         0   -0.1931    0.1576    0.1576    0.1576    0.1576    0.1576
   -0.1644   -0.1644   -0.1931         0    0.1576    0.1576    0.1576    0.1576    0.1576
    0.1576    0.1576    0.1576    0.1576         0   -0.4978   -0.1945   -0.1145    0.0668
    0.1576    0.1576    0.1576    0.1576   -0.4978         0   -0.1945   -0.1145    0.0668
    0.1576    0.1576    0.1576    0.1576   -0.1945   -0.1945         0   -0.1145    0.0668
    0.1576    0.1576    0.1576    0.1576   -0.1145   -0.1145   -0.1145         0    0.0668
    0.1576    0.1576    0.1576    0.1576    0.0668    0.0668    0.0668    0.0668         0


orderperm =

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

Partition	Level
{{6:Br,1:St,7:So,2:Gi,9:Sc,5:Th,4:Re,8:Ke,3:Oc}}	.1576
{{6:Br,1:St,7:So,2:Gi},{9:Sc,5:Th,4:Re,8:Ke,3:Oc}}	.0668
{{6:Br,1:St,7:So,2:Gi},{9:Sc,5:Th,4:Re,8:Ke},{3:Oc}}	−.1145
{{6:Br,1:St,7:So,2:Gi},{9:Sc,5:Th,4:Re},{8:Ke},{3:Oc}}	−.1644
{{6:Br,1:St},{7:So,2:Gi},{9:Sc,5:Th,4:Re},{8:Ke},{3:Oc}}	−.1931
{{6:Br,1:St},{7:So},{2:Gi},{9:Sc,5:Th,4:Re},{8:Ke},{3:Oc}}	−.1945
{{6:Br,1:St},{7:So},{2:Gi},{9:Sc,5:Th},{4:Re},{8:Ke},{3:Oc}}	−.2658
{{6:Br},{1:St},{7:So},{2:Gi},{9:Sc,5:Th},{4:Re},{8:Ke},{3:Oc}}	−.4978
{{6:Br},{1:St},{7:So},{2:Gi},{9:Sc},{5:Th},{4:Re},{8:Ke},{3:Oc}}	—

10 Some Possible Clustering Generalizations

10.1 Representation Through Multiple Tree Structures

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

The M-files, `biultrafnd.m` and `biatreefnd.m` fit (additively) two ultrametric or additive tree matrices in the least-squares sense. The explicit usages are

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

[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 obtained least-squares matrix (with variance-accounted-for of `vaf`) to `prox`, and is the sum of the two ultrametric or additive tree matrices `targone` and `targtwo`.

We will not given an explicit illustration of using two fitted structures to represent a proximity matrix. The data set we have been using, `supreme_agree`, is not a good example for multiple structures because only one such device is really needed to explain everything present in the data. More suitable proximity matrices would probably themselves be obtained by a mixture or

aggregation of other proximity matrices, reflecting somewhat different underlying structures; hopefully, these could be "teased apart" in an analysis using multiple additive structures.

10.2 Individual Differences and Ultrametrics

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

10.3 Incorporating Transformations of the Proximities

In the use of either a one- or two-mode proximity matrix, the data were assumed 'as is', and without any preliminary transformation. It was noted that some analyses leading to negative values might be more pleasingly interpretable if they were positive, and thus, a sufficiently large additive constant was imposed on the original proximities (without any real loss of generality).

In other words, the structures fit to proximity matrices have an invariance with respect to linear transformations of the proximities. A second type of transformation is implicit in the use of additive trees where a centroid (metric), fit as part of the whole representational structure, has the effect of double-centering (i.e., for the input proximity matrix deviated from the centroid, zero sums are present within rows or columns). In effect, the analysis methods iterate between fitting an ultrametric and a centroid, attempting to squeeze out every last bit of VAF. Maybe a more direct strategy (and one that would most likely not affect the substantive interpretations materially) would be to initially double-center (either a one- or two-mode matrix), and then treat the later to the analyses we wish to carry out, without again revisiting the double-centering operation during the iterative process.

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

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

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

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

tion and is one-half the squared differences between the entries in `fitted` (`fittedtm`) and `monproxpermut` (`monproxpermuttm`).

A script M-file is listed in the verbatim output below that gives an application of `proxmon.m` using the best-fitting ultrametric structure found for `supreme_agree`. First, `ultrafnd.m` is invoked to obtain a fitted matrix (`fit`) with `vaf` of .7369; `proxmon.m` then generates the monotonically transformed proximity matrix (`monproxpermut`) with `vaf` of .9264 and `diff` of .0622. The strategy is repeated one-hundred times (i.e., finding a fitted matrix based on the monotonically transformed proximity matrix, finding a new monotonically transformed matrix, and so on). To avoid degeneracy (where all matrices would just converge to zeros), the sum of squares of the fitted matrix is normalized (and held constant). Here, a perfect `vaf` of 1.0 is achieved (and a `diff` of 0.0); the structure of the proximities is now pretty flattened with only four new clusters explicitly identified in the partition hierarchy (the levels at which these are formed are given next to the clusters): (Sc,Th):.2149; (Gi,So):.2251; (Ke,Re):.2353; (Br,Gi,So):.2916. Another way of stating this is to observe that the monotonically transformed proximity matrix has only five distinct values, and over 86% are at a common value of .5588, the level at which the single all-inclusive subset is formed.

```
>> type ultra_monotone_test

load supreme_agree.dat [fit,vaf] =
ultrafnd(supreme_agree,randperm(9))

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

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

for i = 1:100

    [fit,vaf] = ultrafit(monprox,fit);

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

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

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

end

fit

vaf

diff

monprox
```

```
supreme_agree


>> ultra_monotone_test

fit =

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


vaf =

    0.7369


monprox =

         0    0.3761    0.3633    0.3761    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3761         0    0.2850    0.2850    0.5211    0.6405    0.6405    0.6405    0.6405
    0.3633    0.2850         0    0.2200    0.6405    0.6405    0.6405    0.6405    0.6405
    0.3761    0.2850    0.2200         0    0.5211    0.6405    0.6405    0.6405    0.6405
    0.6405    0.5211    0.6405    0.5211         0    0.3558    0.3100    0.5211    0.5211
    0.6405    0.6405    0.6405    0.6405    0.3558         0    0.2300    0.4017    0.4017
    0.6405    0.6405    0.6405    0.6405    0.3100    0.2300         0    0.3761    0.3558
    0.6405    0.6405    0.6405    0.6405    0.5211    0.4017    0.3761         0    0.2100
    0.6405    0.6405    0.6405    0.6405    0.5211    0.4017    0.3558    0.2100         0


vaf =

    0.9264


diff =

    0.0622


fit =

         0    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588
    0.5588         0    0.2916    0.2916    0.5588    0.5588    0.5588    0.5588    0.5588
    0.5588    0.2916         0    0.2251    0.5588    0.5588    0.5588    0.5588    0.5588
    0.5588    0.2916    0.2251         0    0.5588    0.5588    0.5588    0.5588    0.5588
    0.5588    0.5588    0.5588    0.5588         0    0.5588    0.5588    0.5588    0.5588
    0.5588    0.5588    0.5588    0.5588    0.5588         0    0.2353    0.5588    0.5588
    0.5588    0.5588    0.5588    0.5588    0.5588    0.2353         0    0.5588    0.5588
    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588         0    0.2149
    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588    0.2149         0


vaf =

     1
```

```
diff =

     0


monprox =

         0    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588
    0.5588         0    0.2916    0.2916    0.5588    0.5588    0.5588    0.5588    0.5588
    0.5588    0.2916         0    0.2251    0.5588    0.5588    0.5588    0.5588    0.5588
    0.5588    0.2916    0.2251         0    0.5588    0.5588    0.5588    0.5588    0.5588
    0.5588    0.5588    0.5588    0.5588         0    0.5588    0.5588    0.5588    0.5588
    0.5588    0.5588    0.5588    0.5588    0.5588         0    0.2353    0.5588    0.5588
    0.5588    0.5588    0.5588    0.5588    0.5588    0.2353         0    0.5588    0.5588
    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588         0    0.2149
    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588    0.5588    0.2149         0


supreme_agree =

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

10.4 Finding and Fitting Best Ultrametrics in the Presence of Missing Proximities

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

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

[fit,vaf] = ultrafit_missing(prox,targ,proxmiss)
```

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

Table 5: Dissimilarities Among Ten Supreme Court Justices for the 2005/6 Term. The Missing Entry Between O'Connor and Alito is Represented With an Asterisk.

```
[find,vaf] = ultrafnd_missing(prox,inperm,proxmiss)
```

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

As an illustration of the M-files just introduced, Table 5 provides data on the ten supreme court justices present at some point during the 2005/6 term, and the percentage of times justices disagreed in non-unanimous decisions during the year. (These data were in the *New York Times* on July 2, 2006, as part of a "first-page, above-the-fold" article bylined by Linda Greenhouse entitled "Roberts Is at Court's Helm, But He Isn't Yet in Control".) There is a single missing value in the table between O'Connor (Oc) and Alito (Al) because they shared a common seat for the term until Alito's confirmation by Congress. Roberts (Ro) served the full year as Chief Justice so no missing data entries involve him. As can be seen in the verbatim output to follow, an empirically obtained ordering (presumably from "left" to "right") using `order_missing.m` is

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

suggesting rather strongly that Kennedy will most likely now occupy the middle position (although possibly shifted somewhat to the right) once O'Connor

is removed from the court's deliberations. The best-fitting ultrametric obtained with `ultrafnd_missing.m` has VAF of 72.75%, and is given below in partition hierarchy form using the justice ordering from `order_missing.m`, except for the slight interchange of Sc and Al (this allows the fitted ultrametric to display its perfect AR form, as the verbatim output shows).

```
>> load supreme_agree_2005_6.dat

>> load supreme_agree_2005_6_missing.dat

>> supreme_agree_2005_6

supreme_agree_2005_6 =

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

>> supreme_agree_2005_6_missing

supreme_agree_2005_6_missing =

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

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

>> outperm

outperm =

    10     9     8     7     6     5     2     3     4     1

>> [find,vaf] = ultrafnd_missing(supreme_agree_2005_6,randperm(10),supreme_agree_2005_6_missing)

find =

         0    0.3633    0.3633    0.3100    0.3633    0.5954    0.5954    0.5954    0.5954    0.5954
    0.3633         0    0.2300    0.3633    0.1400    0.5954    0.5954    0.5954    0.5954    0.5954
    0.3633    0.2300         0    0.3633    0.2300    0.5954    0.5954    0.5954    0.5954    0.5954
    0.3100    0.3633    0.3633         0    0.3633    0.5954    0.5954    0.5954    0.5954    0.5954
    0.3633    0.1400    0.2300    0.3633         0    0.5954    0.5954    0.5954         0    0.5954
    0.5954    0.5954    0.5954    0.5954    0.5954         0    0.2950    0.2950    0.2950    0.2950
    0.5954    0.5954    0.5954    0.5954    0.5954    0.2950         0    0.1725    0.0900    0.1725
```

```
    0.5954    0.5954    0.5954    0.5954    0.5954    0.2950    0.1725         0    0.1725    0.1600
    0.5954    0.5954    0.5954    0.5954         0    0.2950    0.0900    0.1725         0    0.1725
    0.5954    0.5954    0.5954    0.5954    0.5954    0.2950    0.1725    0.1600    0.1725         0


vaf =

    0.7275

>> find([1 4 3 2 5 6 7 9 8 10],[1 4 3 2 5 6 7 9 8 10])

ans =

         0    0.3100    0.3633    0.3633    0.3633    0.5954    0.5954    0.5954    0.5954    0.5954
    0.3100         0    0.3633    0.3633    0.3633    0.5954    0.5954    0.5954    0.5954    0.5954
    0.3633    0.3633         0    0.2300    0.2300    0.5954    0.5954    0.5954    0.5954    0.5954
    0.3633    0.3633    0.2300         0    0.1400    0.5954    0.5954    0.5954    0.5954    0.5954
    0.3633    0.3633    0.2300    0.1400         0    0.5954    0.5954         0    0.5954    0.5954
    0.5954    0.5954    0.5954    0.5954    0.5954         0    0.2950    0.2950    0.2950    0.2950
    0.5954    0.5954    0.5954    0.5954    0.5954    0.2950         0    0.0900    0.1725    0.1725
    0.5954    0.5954    0.5954    0.5954         0    0.2950    0.0900         0    0.1725    0.1725
    0.5954    0.5954    0.5954    0.5954    0.5954    0.2950    0.1725    0.1725         0    0.1600
    0.5954    0.5954    0.5954    0.5954    0.5954    0.2950    0.1725    0.1725    0.1600         0
```

Partition	Level
{{1:St,4:Gi,3:Br,2:So,5:Oc,6:Ke,7:Ro,9:Al,8:Sc,10:Th}}	.5954
{{1:St,4:Gi,3:Br,2:So,5:Oc},{6:Ke,7:Ro,9:Al,8:Sc,10:Th}}	.3633
{{1:St,4:Gi},{3:Br,2:So,5:Oc},{6:Ke,7:Ro,9:Al,8:Sc,10:Th}}	.3100
{{1:St},{4:Gi},{3:Br,2:So,5:Oc},{6:Ke,7:Ro,9:Al,8:Sc,10:Th}}	.2950
{{1:St},{4:Gi},{3:Br,2:So,5:Oc},{6:Ke},{7:Ro,9:Al,8:Sc,10:Th}}	.2300
{{1:St},{4:Gi},{3:Br},{2:So,5:Oc},{6:Ke},{7:Ro,9:Al,8:Sc,10:Th}}	.1725
{{1:St},{4:Gi},{3:Br},{2:So,5:Oc},{6:Ke},{7:Ro,9:Al},{8:Sc,10:Th}}	.1600
{{1:St},{4:Gi},{3:Br},{2:So,5:Oc},{6:Ke},{7:Ro,9:Al},{8:Sc},{10:Th}}	.1400
{{1:St},{4:Gi},{3:Br},{2:So},{5:Oc},{6:Ke},{7:Ro,9:Al},{8:Sc},{10:Th}}	.0900
{{1:St},{4:Gi},{3:Br},{2:So},{5:Oc},{6:Ke},{7:Ro},{9:Al},{8:Sc},{10:Th}}	—

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

arobfit.m

```
function [fit, vaf] = arobfit(prox, inperm)

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

arobfnd.m

```
function [find, vaf, outperm] = arobfnd(prox, inperm, kblock)

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

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
% halfway along the longest path).
%
% syntax: [ulmetric,ctmetric] = atreedec(prox,constant)
%
% 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 centroid 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).
```

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.
%
% syntax: [fit,vaf] = atreefit(prox,targ)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% TARG is a 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.
```

atreefnd.m

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

% ATREEFND finds and fits an additive tree using iterative projection
% heuristically on a symmetric proximity matrix in the $L_{2}$-norm.
```

```
%
% syntax: [find,vaf] = atreefnd(prox,inperm)
%
% 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.
```

atreefndtm.m

```
function [find,vaf,ultra,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.
%
% syntax: [find,vaf,ultra,lengths] = ...
%      atreefndtm(proxtm,inpermrow,inpermcol)
%
% 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.
```

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.
%
% syntax: [find,vaf,targone,targtwo] = biatreefnd(prox,inperm)
```

```
%
% 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.
```

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.
%
% syntax: [find,vaf,targone,targtwo] = biultrafnd(prox,inperm)
%
% 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.
```

cent_ultrafnd_confit.m

```
function [find,vaf,outperm,targone,targtwo,lengthsone] = ...
    cent_ultrafnd_confit(prox,inperm,conperm)

% CENT_ULTRAFND_CONFIT finds and fits an additive tree by first fitting
% a centroid metric and secondly an ultrametric to the residual
% matrix where the latter is constrained by a given object order.
%
% syntax: [find,vaf,outperm,targone,targtwo,lengthsone] = ...
%      cent_ultrafnd_confit(prox,inperm,conperm)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation); CONPERM is the given
% input constraining order (permutation) which is also given
% as the output vector OUTPERM;
% INPERM is a permutation that determines the order in which the
```

```
% inequality constraints are considered in identifying the ultrametric;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROX satisfying the additive tree constraints. TARGTWO is
% the ultrametric component of the decomposition; TARGONE is the centroid
% metric component defined by the lengths in LENGTHSONE.
```

cent_ultrafnd_confnd.m

```
function [find,vaf,outperm,targone,targtwo,lengthsone] = ...
    cent_ultrafnd_confnd(prox,inperm)

% CENT_ULTRAFND_CONFND finds and fits an additive tree by first fitting
% a centroid metric and secondly an ultrametric to the residual
% matrix where the latter is displayed by a constraining object order that
% is also identified in the process.
%
% syntax: [find,vaf,outperm,targone,targtwo,lengthsone] = ...
%      cent_ultrafnd_confnd(prox,inperm)
%
% 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 in identifying the ultrametric;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROX satisfying the additive tree constraints. TARGTWO is
% the ultrametric component of the decomposition; TARGONE is the centroid
% metric component defined by the lengths in LENGTHSONE; OUTPERM is the
% identified constraining object order used to display the ultrametric
% component.
```

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

centfittm.m

```
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).
%
% syntax: [fit,vaf,lengths] = centfittm(proxtm)
%
% The $n$ values (where $n$ = number of rows + number of columns)
% serve to define the approximating sums,
% $u_{i} + v_{j}$, where the $u_{i}$ are for the rows and the $v_{j}$
% are for the columns; these are given in  the vector LENGTHS of size
% $n \times 1$, with row values first followed by the column values.
```

consec_subsetfit.m

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

% CONSEC_SUBSETFIT defines a collection of partitions involving
% consecutive subsets for the object set and then calls partitionfit.m
% to fit a least-squares approximation to the input proximity matrix based
% on these identified partitions.
%
% syntax [fitted,vaf,weights,end_condition,member] = consec_subsetfit(prox)
%
% PROX is the n x n input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation); MEMBER is the m x n matrix
% indicating cluster membership, where each row corresponds to a specific
% partition (there are m partitions in general); the columns of MEMBER
% are in the same input order used for PROX. The partitions are defined
% by a single contiguous cluster of objects, with all objects before and
% after this contiguous set forming individual clusters of the partitions.
% The value of m is (n*(n-1)/2) - 1; the partition defined by a single
% contiguous partition is excluded.
% FITTED is an n x n matrix fitted to PROX (through least-squares)
% constructed from the nonnegative weights given in the m x 1 WEIGHTS
% vector corresponding to each of the partitions.  VAF is the variance-
% accounted-for in the proximity matrix PROX by the fitted matrix FITTED.
% END_CONDITION should be zero for a normal termination of the optimization
% process.
```

consec_subsetfit_alter.m

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

% CONSEC_SUBSETFIT_ALTER defines a collection of partitions involving
% consecutive subsets for the object set and then calls partitionfit.m
% to fit a least-squares approximation to the input proximity matrix based
% on these identified partitions.
%
% syntax [fitted,vaf,weights,end_condition,member] = ...
%                       consec_subsetfit_alter(prox)
%
% PROX is the n x n input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation); MEMBER is the m x n matrix
% indicating cluster membership, where each row corresponds to a specific
% partition (there are m partitions in general); the columns of MEMBER
% are in the same input order used for PROX. The partitions are defined
% by a single contiguous cluster of objects, with all objects before and
% all objects after this contiguous set (when nonempty) forming
% separate individual clusters of the partitions.
% (These possible three-class partitions when before and after subsets are
% both nonempty) distinguish consec_subsetfit_alter.m from consec_subsetfit.m).
% The value of m is (n*(n-1)/2) - 1; the partition defined by a single
% contiguous partition is excluded.
% FITTED is an n x n matrix fitted to PROX (through least-squares)
% constructed from the nonnegative weights given in the m x 1 WEIGHTS
% vector corresponding to each of the partitions.  VAF is the variance-
% accounted-for in the proximity matrix PROX by the fitted matrix FITTED.
% END_CONDITION should be zero for a normal termination of the optimization
% process.
```

dykstra.m

```
code only; no help file

function [solution, kuhn_tucker, iterations, end_condition] = ...
    dykstra(data,covariance,constraint_array,constraint_constant,equality_flag)
```

insertqa.m

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

```
% INSERTQA carries out an iterative
% Quadratic Assignment maximization task using 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.
%
% syntax: [outperm, rawindex, allperms, index] = ...
%   insertqa(prox, targ, inperm, kblock)
%
% INPERM is the input beginning permutation
% (a permutation of the first $n$ integers).
% PROX is the $n \times n$ input proximity matrix.
% TARG is the $n \times n$ input target matrix.
% OUTPERM is the final permutation of PROX with the cross-product
% index RAWINDEX with respect to TARG.
% ALLPERMS is a cell array containing INDEX entries corresponding
% to all the permutations identified in the optimization from
% ALLPERMS{1} = INPERM to ALLPERMS{INDEX} = OUTPERM.
```

order.m

```
function [outperm,rawindex,allperms,index] =
order(prox,targ,inperm,kblock)

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

```
% of the first $n$ integers).
% OUTPERM is the final permutation of PROX with the
% cross-product index RAWINDEX
% with respect to TARG. ALLPERMS is a cell array containing INDEX
% entries corresponding to all the
% permutations identified in the optimization from ALLPERMS{1} =
% INPERM to ALLPERMS{INDEX} = OUTPERM.
```

order_missing.m

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

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

pairwiseqa.m

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

% PAIRWISEQA carries out an iterative
% Quadratic Assignment maximization task using the
% pairwise interchanges of objects in the
% permutation defining the row and column
% order of the data matrix.
%
% syntax: [outperm, rawindex, allperms, index] = ...
%   pairwiseqa(prox, targ, inperm)
%
% INPERM is the input beginning permutation
% (a permutation of the first $n$ integers).
% PROX is the $n \times n$ input proximity matrix.
% TARG is the $n \times n$ input target matrix.
% OUTPERM is the final permutation of
% PROX with the cross-product index RAWINDEX
% with respect to TARG.
% ALLPERMS is a cell array containing INDEX entries corresponding
% to all the permutations identified in the optimization from
% ALLPERMS{1} = INPERM to ALLPERMS{INDEX} = OUTPERM.
```

partitionfit.m

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

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

```
% END_CONDITION should be zero for a normal termination of the optimization
% process.
```

partitionfnd_averages.m

```
function [membership,objectives] = partitionfnd_averages(prox)

% PARTITIONFND_AVERAGES uses dynamic programming to
% construct a linearly constrained cluster analysis that
% consists of a collection of partitions with from 1 to
% n ordered classes.
%
% syntax: [membership,objectives] = partitionfnd_averages(prox)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% MEMBERSHIP is the n x n matrix indicating cluster membership,
% where rows correspond to the number of ordered clusters,
% and the columns are in the identity permutation input order
% used for PROX.
% OBJECTIVES is the vector of merit values minimized in the
% construction of the ordered partitions, each defined by the
% maximum over clusters of the average proximities within subsets.
```

partitionfnd_diameters.m

```
function [membership,objectives] = partitionfnd_diameters(prox)

% PARTITIONFND_DIAMETERS uses dynamic programming to
% construct a linearly constrained cluster analysis that
% consists of a collection of partitions with from 1 to
% n ordered classes.
%
% syntax: [membership,objectives] = partitionfnd_diameters(prox)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% MEMBERSHIP is the n x n matrix indicating cluster membership,
% where rows correspond to the number of ordered clusters,
% and the columns are in the identity permutation input order
% used for PROX.
% OBJECTIVES is the vector of merit values minimized in the
% construction of the ordered partitions, each defined by the
% maximum over clusters of the maximum proximities within subsets.
```

partitionfnd_kmeans.m

```
function [membership,objectives] = partitionfnd_kmeans(prox)

% PARTITIONFND_KMEANS uses dynamic programming to
% construct a linearly constrained cluster analysis that
% consists of a collection of partitions with from 1 to
% n ordered classes.
%
% syntax: [membership,objectives] = partitionfnd_kmeans(prox)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation);
% MEMBERSHIP is the n x n matrix indicating cluster membership,
% where rows correspond to the number of ordered clusters,
% and the columns are in the identity permutation input order
% used for PROX.
% OBJECTIVES is the vector of merit values minimized in the
% construction of the ordered partitions, each defined by the
% sum over clusters of the average (using a division by twice the
% number of objects in the class) of the proximities within subsets.
```

proxmon.m

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

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

proxmontm.m

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

sqeuclidean.m

```
code only; no help file

 function [sqeuclid] = sqeuclidean(data)
```

targlin.m

```
function [targlinear] = targlin(n)

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

ultracomptm.m

```
function [ultracomp] = ultracomptm(ultraproxtm)

% ULTRACOMPTM provides a completion of a given two-mode ultrametric
% matrix to a symmetric proximity matrix satisfying the
% usual ultrametric constraints.
%
% syntax: [ultracomp] = ultracomptm(ultraproxtm)
```

```
%
% ULTRAPROXTM is the $nrow \times ncol$ two-mode ultrametric matrix;
% ULTRACOMP is the completed symmetric
% $n \times n$ proximity matrix having the usual
% ultrametric pattern for $n = nrow + ncol$.
```

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

ultrafit_missing.m

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

% ULTRAFIT_MISSING fits a given ultrametric using iterative projection to
% a symmetric proximity matrix in the $L_{2}$-norm.
%
% syntax: [fit,vaf] = ultrafit_missing(prox,targ,proxmiss)
%
% PROX is the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation); also, missing entries in the input
% proximity matrix PROX are given values of zero.
% 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. PROXMISS is the same size as PROX (with main
% diagonal entries all zero); an off-diagonal entry of 1.0 denotes an
% entry in PROX that is present and 0.0 if it is absent.
```

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.
%
% syntax: [fit,vaf] = ultrafittm(proxtm,targ)
%
% 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.
```

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.
%
% syntax: [find,vaf] = ultrafnd(prox,inperm)
%
% 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.
```

ultrafnd_confit.m

```
function [find,vaf,vafarob,arobprox,vafultra] = ...
    ultrafnd_confit(prox,inperm,conperm)

% ULTRAFND_CONFIT finds and fits an ultrametric using iterative projection
% heuristically on a symmetric proximity matrix in the $L_{2}$-norm,
% constrained by a given object order.
%
% syntax: [find,vaf,vafarob,arobprox,vafultra] = ...
%    ultrafnd_confit(prox,inperm,conperm)
%
```

```
% 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 in obtaining the ultrametric;
% CONPERM is the given constraining object order;
% VAFAROB is the VAF of the anti-Robinson matrix fit, AROBPROX, to PROX;
% VAFULTRA is the VAF of the ultrametric fit to AROBPROX;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROX satisfying the ultrametric constraints, and given
% in CONPERM order.
```

ultrafnd_confnd.m

```
function [find,vaf,conperm,vafarob,arobprox,vafultra] = ...
    ultrafnd_confnd(prox,inperm)

% ULTRAFND_CONFND finds and fits an ultrametric using iterative projection
% heuristically on a symmetric proximity matrix in the $L_{2}$-norm, and
% also locates a initial constraining object order.
%
% syntax: [find,vaf,conperm,vafarob,arobprox,vafultra] = ...
%    ultrafnd_confnd(prox,inperm)
%
% 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 in obtaining the ultrametric;
% CONPERM is the identified constraining object order;
% VAFAROB is the VAF of the anti-Robinson matrix fit, AROBPROX, to PROX;
% VAFULTRA is the VAF of the ultrametric fit to AROBPROX;
% FIND is the found least-squares matrix (with variance-accounted-for
% of VAF) to PROX satisfying the ultrametric constraints, and given
% in CONPERM order.
```

ultrafnd_missing.m

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

% ULTRAFND_MISSING finds and fits an ultrametric using iterative projection
% heuristically on a symmetric proximity matrix in the $L_{2}$-norm.
%
% syntax: [find,vaf] = ultrafnd_missing(prox,inperm,proxmiss)
%
% PROX is the input proximity matrix (with a zero main diagonal
```

```
% and a dissimilarity interpretation); also, missing entries in the input
% proximity matrix PROX are given values of zero.
% 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. PROXMISS is
% the same size as PROX (with main
% diagonal entries all zero); an off-diagonal entry of 1.0 denotes an
% entry in PROX that is present and 0.0 if it is absent.
```

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.
%
% syntax: [find,vaf] = ultrafndtm(proxtm,inpermrow,inpermcol)
%
% 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.
```

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.
%
% syntax:  [orderprox,orderperm] = ultraorder(prox)
```

ultraplot.m

```
function [] = ultraplot(ultra)
```

```
% ULTRAPLOT gives a dendrogram plot for the input ultrametric
% dissimilarity matrix ULTRA.
%
% syntax: [] = ultraplot(ultra)
```