(Uni- and Bi-)Dimensional Scaling: A Toolbox for MATLAB

Version: July 17, 2011

Contents

List of Tables

List of Figures

1 Introduction

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

The title given to this monograph includes the phrase "bidimensional scaling." The prefix "bi" is justified by considering an additive combination of two distinct unidimensional scales — or to use jargon common to the multidimensional scaling literature, a unidimensional model is extended to one involving a two-dimensional city-block metric. In theory, city-block scaling can be extended to more than two dimensions, but the convenient planar graphical representation would be lost; we therefore choose to limit our discussion to just the two-dimensional case.

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

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

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

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

```
.00  .38  .34  .37  .67  .64  .75  .86  .85
.38  .00  .28  .29  .45  .53  .57  .75  .76
.34  .28  .00  .22  .53  .51  .57  .72  .74
.37  .29  .22  .00  .45  .50  .56  .69  .71
```

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

1.2 Additional Data Sets Used for Illustration

In addition to the data on the Supreme Court just described, there are five more data sets used throughout this monograph to provide examples of usage for specific M-files introduced. One is from Johnson and Tversky (1984) on perceptions of risk, and is in the form of an 18×18 symmetric proximity matrix. In our circular scaling representations, we will rely on the famous Morse Code digit data from Rothkopf (1957). The third proximity matrix for our discussion of multidimensional scaling (really, in two dimensions) based on the Euclidean and/or city-block metric, was derived from an inter-town distance matrix among twenty-seven Hampshire County cities (in England), published by John Norden in 1625. Penultimately, we give six (two-mode or rectangular) (10×15) proximity matrices from Osgood and Luria (1954) that provide semantic differential data on a well-known case of multiple personality (the *Three Faces of Eve*). There are two replications given for each of the separate personalities of Eve White, Eve Black, and Jane; we will chose one of these matrices for purposes of illustration. Finally, to have two skew-symmetric matrices to scale these kinds of data unidimensionally, we use an

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

Table 1: Dissimilarities Among Nine Supreme Court Justices.

old data set from Thurstone (1959) involving the comparison of relative seriousness of thirteen offenses. These comparisons were done by school children before and after seeing a movie showing what the life of a gambler was like.

1.2.1 Risk Perception

The data set on risk perception (an 18×18 proximity matrix) is generated from Johnson and Tversky (1984) and given in Table 2. It involves eighteen risks:

1: accidental falls
2: airplane accidents
3: electrocution
4: fire
5: flood
6: heart disease
7: homicide
8: leukaemia
9: lightning
10: lung cancer
11: nuclear accident
12: stomach cancer
13: stroke
14: terrorism
15: tornados
16: toxic chemical spill
17: traffic accidents
18: war

The original proximity matrix was obtained by averaging the ratings from a group of subjects who evaluated risk pairs on a scale from one (very dissimilar) to nine (very similar). To key these as dissimilarities, the similarities were subtracted from 10.0 to produce the entries given in Table 2 (the `ascii` data file is called `risk_rate.dat`).

1.2.2 Morse Code Digit Data

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

1.2.3 Hampshire County (in England) Inter-town Distances

Table 4 provides a 27×27 dissimilarity matrix among twenty-seven Hampshire County towns (the first name listed is from Figure 1; current spellings are given in parentheses):

1: Bramfhot (Bramshott)
2: Hertford bridge (Hartfordbridge)
3: Stoke-bridge (Stockbridge)
4: Whit-church (Whitchurch)
5: Micheldouer (Micheldever)
6: Odyam (Odiham)

7: Lymington (Lymington)
8: Beaulieu (Beaulieu)
9: Titchefeild (Titchefield)
10: Wickham (Wickham)
11: Ouerton (Overton)
12: Bafingftoke (Basingstoke)
13: S. Hampton (Southhamption)
14: Chrift-Church (ChristChurch)
15: Ryngwood (Ringwood)
16: Fording-Bridge (Fordingbridge)
17: Rumfey (Romsey)
18: Andouer (Andover)
19: Kingefelere (Kingsclere)
20: B. Waltham (Bishops Waltham)
21: Alresforde (Alresford)
22: Alton (Alton)
23: Petersfeild (Petersfield)
24: Hauant (Havant)
25: Fareham (Fareham)
26: Portefmouth (Portesmouth)
27: Winchefter (Winchester)

These data are from John Norden's book (published 1625), *England, an Intended Guyde for English Travailers*. A facsimile page is given in Figure 1 (with its interesting reverse column order compared to Table 4); a (more or less current map) of the county appears in Figure 2 (Cobbett's Hampshire Map from 1830) that shows many of the twenty-seven county towns. Hampshire County is in south-central England and borders on the English Channel. It includes the port of Southampton (from whence the Titanic sailed on its maiden voyage (April 10, 1912; see Figure 3)) (the `ascii` data file is called `hampshire_proximities.dat`).

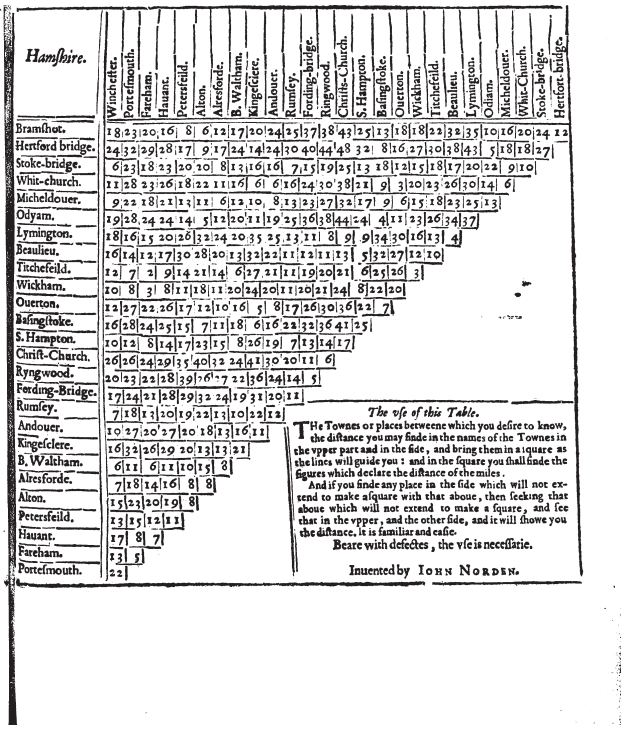


Figure 1: Facsimile of John Norden's Distance Map for Hampshire County (1625).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	0.0	7.6	6.9	5.4	6.2	7.5	6.4	7.1	6.2	7.5	7.1	8.2	6.0	7.7	6.8	6.9	4.8	7.5
2	7.6	0.0	7.0	7.0	5.8	8.1	6.9	7.9	5.8	7.9	6.5	8.0	7.6	5.7	6.6	6.4	4.7	6.2
3	6.9	7.0	0.0	3.0	5.9	7.6	7.4	7.9	3.3	8.3	6.5	7.9	7.7	7.0	6.1	6.7	7.0	7.1
4	5.4	7.0	3.0	0.0	7.0	8.3	6.1	7.4	3.8	7.3	6.5	7.7	7.2	6.3	5.4	5.7	5.8	6.1
5	6.2	5.8	5.9	7.0	0.0	8.5	5.3	6.5	3.3	7.7	6.5	8.2	7.9	7.0	2.3	6.9	7.2	7.3
6	7.5	8.1	7.6	8.3	8.5	0.0	7.5	4.4	5.8	6.9	8.1	4.1	3.0	8.2	8.1	7.6	6.7	8.0
7	6.4	6.9	7.4	6.1	5.3	7.5	0.0	7.7	8.0	8.0	4.0	8.2	7.5	3.8	7.5	6.8	5.3	3.0
8	7.1	7.9	7.9	7.4	6.5	4.4	7.7	0.0	8.1	6.1	6.9	3.8	4.8	8.4	8.0	6.7	7.7	7.4
9	6.2	5.8	3.3	3.8	3.3	5.8	8.0	8.1	0.0	7.8	6.5	7.1	6.3	7.0	2.4	6.5	7.1	7.0
10	7.5	7.9	8.3	7.3	7.7	6.9	8.0	6.1	7.8	0.0	8.0	4.3	4.1	7.7	8.6	7.0	6.9	7.4
11	7.1	6.5	6.5	6.5	6.5	8.1	4.0	6.9	6.5	8.0	0.0	6.9	8.3	7.1	6.3	2.7	6.8	4.3
12	8.2	8.0	7.9	7.7	8.2	4.1	8.2	3.8	7.1	4.3	6.9	0.0	7.1	8.1	7.9	7.0	7.8	7.9
13	6.0	7.6	7.7	7.2	7.9	3.0	7.5	4.8	6.3	4.1	8.3	7.1	0.0	7.1	4.9	7.4	5.7	7.8
14	7.7	5.7	7.0	6.3	7.0	8.2	3.8	8.4	7.0	7.7	7.1	8.1	7.1	0.0	7.1	6.4	7.1	2.5
15	6.8	6.6	6.1	5.4	2.3	8.1	7.5	8.0	2.4	8.6	6.3	7.9	4.9	7.1	0.0	5.5	7.5	7.8
16	6.9	6.4	6.7	5.7	6.9	7.6	6.8	6.7	6.5	7.0	2.7	7.0	7.4	6.4	5.5	0.0	5.8	7.8
17	4.8	4.7	7.0	5.8	7.2	6.7	5.3	7.7	7.1	6.9	6.8	7.8	5.7	7.1	7.5	5.8	0.0	7.0
18	7.5	6.2	7.1	6.1	7.3	8.0	3.0	7.4	7.0	7.4	4.3	7.9	7.8	2.5	7.8	7.8	7.0	0.0

Table 2: Dissimilarities Among Eighteen Risks.

Table 3: A Proximity Matrix, morse_digits, for the Ten Morse Code Symbols Representing the First Ten Digits.

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

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1	0	12	24	20	16	10	35	32	22	18	18	13	25	43	38	37	25	24	20	17	12	6	8	16	20	23	18
2	12	0	27	18	18	5	43	38	30	27	16	8	32	48	44	40	30	24	14	24	17	9	17	28	29	32	24
3	24	27	0	10	9	22	20	17	18	15	12	18	13	25	19	15	7	16	16	13	8	20	20	23	18	23	6
4	20	18	10	0	6	14	30	26	23	20	3	9	21	38	30	24	16	6	6	16	11	22	18	26	23	28	11
5	16	18	9	6	0	13	25	23	18	15	6	9	17	32	27	23	13	8	10	12	6	11	13	21	18	22	9
6	10	5	22	14	13	0	37	34	26	23	11	4	24	44	38	36	25	19	11	20	12	5	14	24	24	28	19
7	35	43	20	30	25	37	0	4	13	16	30	34	9	9	8	11	13	25	35	20	24	32	26	20	15	16	18
8	32	38	17	26	23	34	4	0	10	12	27	32	5	13	11	12	11	22	32	13	20	28	30	17	12	14	16
9	22	30	18	23	18	26	13	10	0	3	26	25	6	21	20	19	11	21	27	6	14	21	14	9	2	7	12
10	18	27	15	20	15	23	16	12	3	0	20	22	8	24	21	20	11	20	24	20	11	18	11	8	3	8	10
11	18	16	12	3	6	11	30	27	26	20	0	7	22	36	30	26	17	8	5	16	10	12	17	26	22	27	12
12	13	8	18	9	9	4	34	32	25	22	7	0	25	41	36	32	22	16	6	18	11	7	15	25	24	28	16
13	25	32	13	21	17	24	9	5	6	8	22	25	0	17	14	13	7	19	26	8	15	23	17	14	8	12	10
14	43	48	25	38	32	44	9	13	21	24	36	41	17	0	6	11	20	30	41	24	32	40	35	29	24	26	26
15	38	44	19	30	27	38	8	11	20	21	30	36	14	6	0	5	14	24	36	22	27	26	39	28	22	23	20
16	37	40	15	24	23	36	11	12	19	20	26	32	13	11	5	0	11	20	31	19	24	32	29	28	21	24	17
17	25	30	7	16	13	25	13	11	11	11	17	22	7	20	14	11	0	12	22	10	13	22	19	20	13	18	7
18	24	24	16	6	8	19	25	22	21	20	8	16	19	30	24	20	12	0	11	16	13	18	20	27	20	27	10
19	20	14	16	6	10	11	35	32	27	24	5	6	26	41	36	31	22	11	0	21	13	13	20	29	26	32	16
20	17	24	13	16	12	20	20	13	6	20	16	18	8	24	22	19	10	16	21	0	8	15	10	11	6	11	6
21	12	17	8	11	6	12	24	20	14	11	10	11	15	32	27	24	13	13	13	8	0	8	8	16	14	18	7
22	6	9	20	22	11	5	32	28	21	18	12	7	23	40	26	32	22	18	13	15	8	0	8	19	20	23	15
23	8	17	20	18	13	14	26	30	14	11	17	15	17	35	39	29	19	20	20	10	8	8	0	11	12	15	13
24	16	28	23	26	21	24	20	17	9	8	26	25	14	29	28	28	20	27	29	11	16	19	11	0	7	8	17
25	20	29	18	23	18	24	15	12	2	3	22	24	8	24	22	21	13	20	26	6	14	20	12	7	0	5	13
26	23	32	23	28	22	28	16	14	7	8	27	28	12	26	23	24	18	27	32	11	18	23	15	8	5	0	22
27	18	24	6	11	9	19	18	16	12	10	12	16	10	26	20	17	7	10	16	6	7	15	13	17	13	22	0

Table 4: A Dissimilarity Matrix Among Twenty-seven Hampshire County Towns.

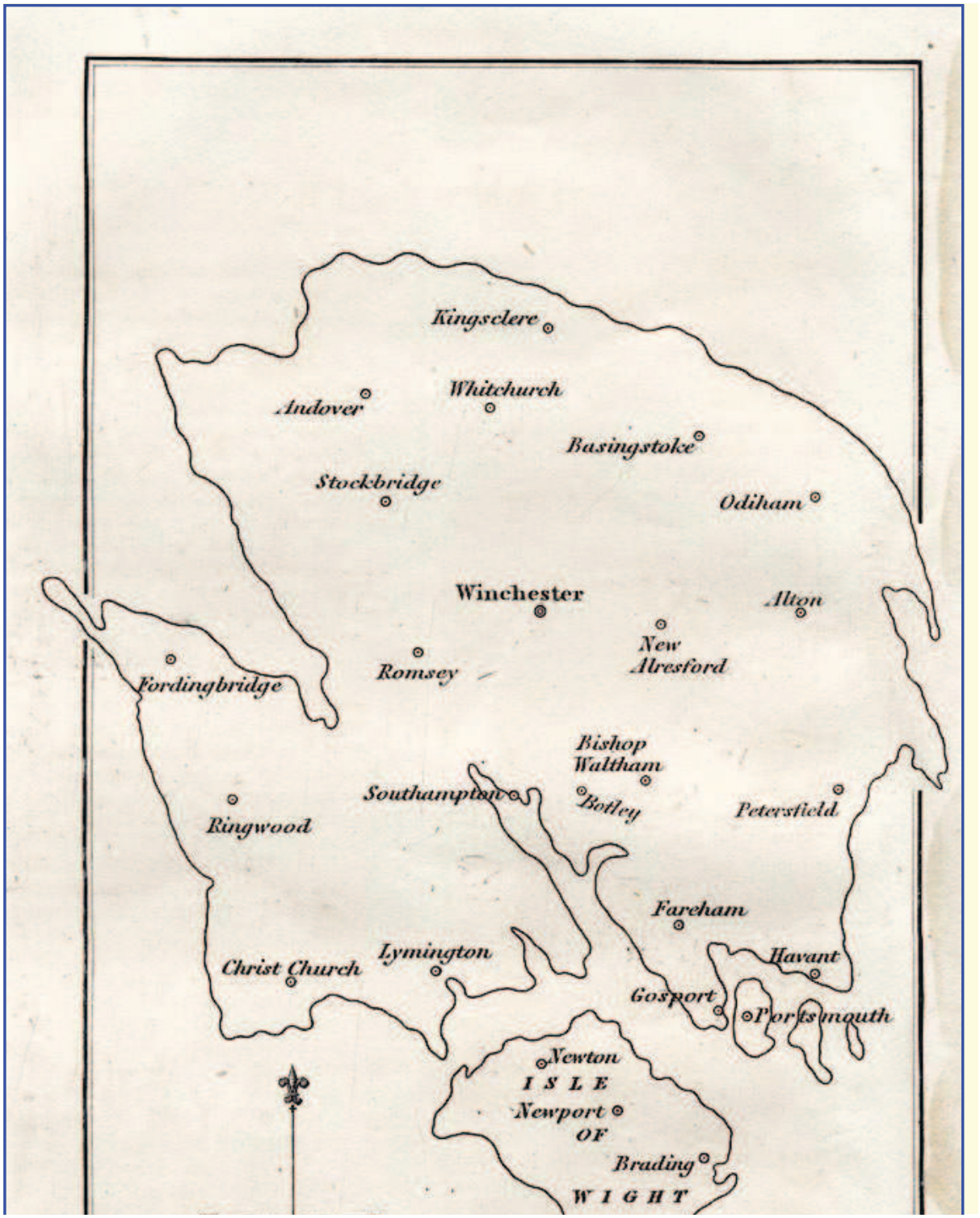


Figure 2: Cobbett's 1830 Map of Hampshire County.

http://www.maritimequest.com/liners/titanic/photos/art/04_titanic.jpg

http://www.maritimequest.com/liners/titanic/photos/art/04_titanic.jpg [3/28/2007 2:11:52 PM]

Figure 3: A Famous Sailing from Southampton.

concepts	1(a)	2(b)	3(c)	4(d)	5(e)	6(f)	7(g)	8(h)	9(i)	10(j)	11(k)	12(l)	13(m)	14(n)	15(o)
scales															
1	4.0	1.0	7.0	7.0	1.0	1.0	1.0	7.0	6.5	2.5	3.5	6.0	7.0	4.0	1.0
2	7.0	7.0	1.0	1.0	7.0	7.0	4.5	1.0	2.0	7.0	4.5	1.5	1.0	7.0	7.0
3	1.0	1.0	7.0	7.0	4.0	2.5	4.0	7.0	6.0	4.0	2.0	6.5	1.0	1.5	1.0
4	1.0	7.0	7.0	4.0	5.5	4.0	4.0	7.0	6.5	2.5	6.0	6.5	1.0	4.0	1.0
5	7.0	4.0	1.0	1.0	7.0	1.0	1.0	1.0	1.5	7.0	2.0	1.0	4.0	4.5	4.0
6	2.0	1.0	7.0	7.0	4.0	1.5	7.0	7.0	5.0	4.0	4.5	6.0	7.0	1.5	4.0
7	1.0	7.0	7.0	7.0	1.0	1.0	4.0	7.0	6.0	4.0	5.0	6.5	1.0	1.0	1.0
8	7.0	7.0	1.5	1.0	7.0	7.0	7.0	1.0	2.0	5.5	3.5	2.0	1.0	4.0	7.0
9	7.0	7.0	1.0	1.0	7.0	4.0	7.0	1.0	1.5	4.0	2.0	1.5	7.0	4.0	7.0
10	7.0	7.0	1.0	1.0	7.0	7.0	1.0	1.0	1.5	7.0	4.5	1.5	1.0	4.0	7.0

Table 5: A Two-Mode Dissimilarity Matrix for Eve Black Between Ten Scales and Fifteen Concepts.

1.2.4 Semantic Differential Data on the "Three Faces of Eve"

To have available a two-mode (10×15) matrix for purposes of illustration, we chose the first replication of Eve Black's Semantic Differential data given in Table 5 (`eve_black_one.dat`). There are six such exemplars available:

`eve_white_one.dat`; `eve_white_two.dat`; `eve_black_one.dat`
`eve_black_two.dat`; `jane_one.dat`; `jane_two.dat`

Readers are welcome to experiment with these as they see fit. All have the same ten rows (or scales) with the table entries being dissimilarities for the left-most member of the bipolar pair:

1) cold-hot
2) valuable-worthless
3) tense-relaxed
4) small-large
5) fast-slow
6) dirty-clean
7) weak-strong
8) tasty-distasteful
9) deep-shallow
10) active-passive

The fifteen columns correspond to the concepts being rated:

1(a): love
2(b): child
3(c): my doctor
4(d): me
5(e): my job
6(f): mental sickness
7(g): my mother
8(h): peace of mind
9(i): fraud
10(j): my spouse
11(k): self-control
12(l): hatred
13(m): my father

14(n): confusion
15(o): sex

To provide some background for these data, we give a short plot summary for the 1957 movie, *The Three Faces of Eve*; Joanne Woodward won an Oscar for her portrayal of Eve White/Eve Black/Jane:

Eve White is a quite, mousy, unassuming wife and mother who keeps suffering from headaches and occasional black outs. Eventually she is sent to see psychiatrist Dr. Luther, and, while under hypnosis, a whole new personality emerges: the racy, wild, fun-loving Eve Black. Under continued therapy, yet a third personality appears, the relatively stable Jane. This film, based on the true-life case of a multiple personality, chronicles Dr. Luther's attempts to reconcile the three faces of Eve's multiple personalities.

The real reason, however, that this particular data matrix exemplar is used from among the six we have available, is to give two quotes from the movie:

Ralph White: I've never seen you take a drink in your life.
Eve Black: Honey, there are a lot of things you ain't never seen me do; that's no sign I don't do 'em.

The Soldier: When I spend eight bucks on a dame, I don't just go home with the morning paper, y'know what I mean?

1.2.5 Skew-Symmetric Matrices from Thurstone

To have (two) skew-symmetric matrices for use in our examples, we begin with a very old data set, originally collected in 1929 in a study of the influence of motion pictures on children's attitudes (see Thurstone, 1959, pp. 309–319). Both before and after seeing a film entitled *Street of Chance*, which depicted the life of a gambler, 240 school children were asked to compare the relative seriousness of thirteen offenses presented in all 78 possible pairs:

1): bankrobber
2): gambler

3): pickpocket
4): drunkard
5): quack doctor
6): bootlegger
7): beggar
8): gangster
9): tramp
10): speeder
11): petty thief
12): kidnaper
13): smuggler

The data are given below, where the entries show the proportion of the school children who rated the offense listed in the column to be more serious than the offense listed in the row. The above-diagonal entries were obtained before the showing of the film; those below were collected after. The obvious substantive question here involves the effect of the film on the assessment of the offense of being a gambler.

1:bankrobber	x	.07	.08	.05	.27	.29	.01	.50	.00	.06	.02	.73	.21
2:gambler	.79	x	.71	.52	.76	.92	.07	.92	.05	.41	.49	.90	.81
3:pickpocket	.93	.51	x	.25	.67	.75	.02	.86	.02	.39	.42	.87	.68
4:drunkard	.95	.70	.70	x	.81	.95	.01	.92	.03	.37	.62	.91	.87
5:quack doctor	.67	.36	.28	.16	x	.49	.02	.70	.02	.12	.22	.64	.55
6:bootlegger	.70	.31	.30	.13	.50	x	.00	.79	.01	.09	.26	.68	.50
7:beggar	.98	.95	.97	.94	.98	.98	x	.96	.42	.86	.96	1.0	.99
8:gangster	.50	.18	.13	.11	.32	.27	.01	x	.02	.08	.08	.36	.31
9:tramp	1.0	.96	.98	.96	.99	.98	.64	.99	x	.91	.97	.99	1.0
10:speeder	.94	.73	.68	.67	.89	.90	.21	.94	.13	x	.58	.90	.92
11:petty thief	.97	.64	.62	.47	.81	.76	.06	.89	.05	.36	x	.98	.78
12:kidnaper	.38	.27	.16	.08	.35	.30	.02	.62	.01	.08	.03	x	.27
13:smuggler	.73	.31	.30	.16	.46	.49	.02	.66	.02	.11	.24	.64	x

To generate two skew-symmetric matrices from the data just given, the entry defined for each pair of offenses in the "before" and "after" skew-symmetric matrices, is a difference in the proportions of rating one offense

more serious than the other. For example, because the proportion judging a bootlegger more serious than a bankrobber is .29 before the movie was shown, an entry of $.29 - .79 = -.42$ is present for the (bankrobber, bootlegger) above-diagonal pair in the before matrix: the negative value above the diagonal indicates that the second member (bootlegger) is less serious than the first (bankrobber) (the two data matrices are called

`thurstone_skew_symmetric_before.dat`;
`thurstone_skew_symmetric_after.dat`.

2 The Basics of Linear Unidimensional Scaling (LUS)

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

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

is minimized. In turn, this optimization suggested by (1) can be rephrased as two separate problems to be solved simultaneously: find a set of n numbers, $x_1 \leq x_2 \leq \cdots \leq x_n$, *and* a permutation on the first n integers, $\rho(\cdot) \equiv \rho$, for which

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

is minimized. Thus, a set of locations (coordinates) is defined along a continuum as represented in ascending order by the sequence $x_1, x_2, \ldots, x_n$; the n objects are allocated to these locations by the permutation ρ, so object $O_{\rho(i)}$ is placed at location i.

The minimization of (2) can be carried out directly by the maximization of the single term, $\sum_i (t_i^{(\rho)})^2$ (under the mild regularity condition that all off-diagonal proximities in $\mathbf{P}$ are positive and not merely nonnegative), where

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

for

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

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

and

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

In words, $u_i^{(\rho)}$ is the sum of the entries within row $\rho(i)$ of $\{p_{\rho(i)\rho(j)}\}$ from the extreme left up to the main diagonal; $v_i^{(\rho)}$ is the sum from the main diagonal to the extreme right. If ρ^* denotes the permutation that maximizes $\sum_i (t_i^{(\rho)})^2$, then we can let $x_i = t_i^{(\rho^*)}$, with the order induced by $t_1^{(\rho^*)}, \ldots, t_n^{(\rho^*)}$ being consistent with the constraint, $x_1 \leq x_2 \leq \cdots \leq x_n$. In short, the minimization of (2) reduces to the combinatorial optimization of the single term $\sum_i (t_i^{(\rho)})^2$, and where the coordinate estimation is completed as an automatic byproduct.

2.1 Iterative Quadratic Assignment

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

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

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

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

the two optimization subproblems to be solved simultaneously of identifying an optimal permutation and a set of coordinates can be separated:

(a) assuming that an ordering of the objects is known (and denoted, say, as ρ^0 for the moment), find those values $x_1^0 \leq \cdots \leq x_n^0$ to minimize $\sum_i (x_i^0 - t_i^{(\rho^0)})^2$.

If the permutation ρ^0 produces a *monotonic* form for the matrix $\{p_{\rho^0(i)\rho^0(j)}\}$ in the sense that $t_1^{(\rho^0)} \leq t_2^{(\rho^0)} \leq \cdots \leq t_n^{(\rho^0)}$, the coordinate estimation is immediate by letting $x_i^0 = t_i^{(\rho^0)}$, in which case $\sum_i (x_i^0 - t_i^{(\rho^0)})^2$ is zero.

(b) assuming that the locations $x_1^0 \leq \cdots \leq x_n^0$ are known, find the permutation ρ^0 to maximize $\sum_i x_i t_i^{(\rho^0)}$. Any such permutation which even only locally maximizes $\sum_i x_i t_i^{(\rho^0)}$, in the sense that no adjacently placed pair of objects in ρ^0 could be interchanged to increase the index, will produce a monotonic form for the nonnegative matrix $\{p_{\rho^0(i)\rho^0(j)}\}$. Also, the task of finding the permutation ρ^0 to maximize $\sum_i x_i t_i^{(\rho^0)}$ is actually a quadratic assignment (QA) task, discussed extensively in the literature of operations research. As usually defined, a QA problem involves two $n \times n$ matrices, $\mathbf{A} = \{a_{ij}\}$ and $\mathbf{B} = \{b_{ij}\}$, and we seek a permutation ρ to maximize

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

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

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

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

2.2 An M-file for Performing LUS Through Iterative QA

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

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

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

the built-in MATLAB random permutation generator, `randperm(9)`, is invoked for a starting permutation.

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

supreme_agree =

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

>> help uniscalqa.m

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

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

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

```
   coordinates and object permutation.


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

>> outperm

outperm =

     1     2     3     4     5     6     7     8     9

>> coord

coord =

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

>> diff

diff =

    0.4691
```

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

2.3 A Useful Utility for the QA Task Generally

For the QA problem in (5), the attempt to find ρ to maximize $\Gamma(\rho)$, reorganizes the (proximity) matrix as $\mathbf{A}_\rho = \{a_{\rho(i)\rho(j)}\}$, which hopefully shows the

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

The type of heuristic optimization strategy we use for the QA task in `order.m`, implements simple object interchange/rearrangement operations. Based on given matrices $\mathbf{A}$ and $\mathbf{B}$, and beginning with some permutation (possibly chosen at random), local interchanges and rearrangements of a particular type are implemented until no improvement in the index can be made. By repeatedly initializing such a process randomly, a distribution over a set of local optima can be achieved. Three different classes of local operations are used in the M-file, `order.m`: (i) the pairwise interchanges of objects in the current permutation defining the row and column order of the data matrix $\mathbf{A}$. All possible such interchanges are generated and considered in turn, and whenever an increase in the cross-product index would result from a particular interchange, it is made immediately. The process continues until the current permutation cannot be improved upon by any such pairwise object interchange. The procedure then proceeds to (ii): the local operations considered are all reinsertions of from 1 to `kblock` (which is less than n and set by the user) consecutive objects somewhere in the permutation defining the current row and column order of the data matrix. When no further improvement can be made, we move to (iii): the local operations are now all possible rotations (or inversions) of from 2 to `kblock` consecutive objects in the current row/column order of the data matrix. (We suggest a use of `kblock` equal to 3 as a reasonable compromise between the extensiveness of local search, speed of execution, and quality of solution.) The three collections of local changes are revisited (in order) until no alteration is possible in the final permutation obtained.

The use of `order.m` is illustrated in the verbatim recording below on the

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

```
>> load supreme_agree.dat

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

outperm =

     1     2     3     4     5     6     7     8     9


rawindex =

  145.1200


index =

    19
```

3 Confirmatory and Nonmetric LUS

In developing linear unidimensional scaling (as well as other types of) representations for a proximity matrix, it is convenient to have a general mechanism available for solving linear (in)equality constrained least-squares tasks. The two such instances discussed in this section involve (a) the confirmatory fitting of a given object order to a proximity matrix (through an M-file called linfit.m), and (b) the construction of an optimal monotonic transformation of a proximity matrix in relation to a given unidimensional ordering (through an M-file called proxmon.m). In both these cases, we rely on what

can be called the Dykstra-Kaczmarz method for solving linear inequality constrained least-squares problems (the latter solution strategy is developed in greater detail in Section 9.9).

3.1 The Confirmatory Fitting of a Given Order

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

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

the matrix $\{|x_j - x_i|\}$ is referred to as the fitted matrix (`fit`); `coord` gives the ordered coordinates; and `diff` is the value of the least-squares criterion. The fitted matrix is found through the Dykstra-Kaczmarz method where the equality constraints defined by distances along a continuum are imposed to construct the fitted matrix, i.e., if $i < j < k$, then $|x_i - x_j| + |x_j - x_k| = |x_i - x_k|$. Once found, the actual ordered coordinates are retrieved by the usual $t_i^{(\rho^0)}$ formula but computed on `fit`. In the example below of the use of `linfit.m`, the identity permutation obtained from the use of `uniscalqa.m` is used as the input permutation.

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

fit =

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

```
diff =

    0.4691


coord =

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

3.2 The Monotonic Transformation of a Proximity Matrix

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

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

Here, `proxpermut` is the input proximity matrix (which may have been subjected to an initial row/column permutation, hence the suffix "`permut`"); `fitted` is a given target matrix; the output matrix `monproxpermut` is closest to `fitted` in a least-squares sense and obeys the order constraints obtained from each pair of entries in (the upper-triangular portion of) `proxpermut` (and where the inequality constrained optimization is carried out using the Dykstra-Kaczmarz iterative projection strategy); `vaf` denotes "variance-accounted-for" and indicates how much variance in `monproxpermut` can be accounted for by `fitted`; finally, `diff` is the value of the least-squares loss function and is the sum of squared differences between the entries in `fitted` and `monproxpermut` (actually, `diff` is one-half of such a sum because the loss function is over $i < j$).

When fitting a given order, `fitted` would correspond to the matrix $\{|x_j - x_i|\}$, where $x_1 \leq x_2 \leq \cdots \leq x_n$; the input `proxpermut` would be $\{p_{\rho^0(i)\rho^0(j)}\}$;

`monproxpermut` would be $\{f(p_{\rho^0(i)\rho^0(j)})\}$, where the function $f(\cdot)$ satisfies the monotonicity constraints, i.e., if $p_{\rho^0(i)\rho^0(j)} < p_{\rho^0(i')\rho^0(j')}$ for $1 \leq i < j \leq n$ and $1 \leq i' < j' \leq n$, then $f(p_{\rho^0(i)\rho^0(j)}) \leq f(p_{\rho^0(i')\rho^0(j')})$. The transformed proximity matrix $\{f(p_{\rho^0(i)\rho^0(j)})\}$ minimizes the least-squares criterion (`diff`) of

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

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

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

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

3.2.1 An Application Incorporating proxmon.m

The script M-file listed below gives an application of `proxmon.m` using the identity permutation for our `supreme_agree` matrix. First, `linfit.m` is invoked to obtain a fitted matrix (`fit`); `proxmon.m` then generates the monotonically transformed proximity matrix (`monprox`) with `vaf` = .9869 and `diff` = .0349. The strategy is then repeated one-hundred times (i.e., finding a fitted matrix based on the monotonically transformed proximity matrix, finding a new monotonically transformed matrix, and so on). To avoid degeneracy (where all matrices would converge to zeros), the sum of squares of the fitted matrix is normalized. As indicated in the output below, the final `vaf` is .9934 with a `diff` of .0190. (Although the permutation found earlier for `supreme_agree` remains the same throughout the construction of the optimal monotonic transformation, in this particular example it would also remain optimal with the same VAF if the unidimensional scaling were repeated with `monprox` now considered the input proximity matrix. Even though probably rare, other data sets might not have such an invariance, and it may be desirable to initiate an iterative routine that finds a unidimensional scaling [i.e., an object ordering] in addition to monotonically transforming the proximity matrix.)

```
>> type uniscale_monotone_test

load supreme_agree.dat
inperm = [1 2 3 4 5 6 7 8 9];
proxpermut = supreme_agree(inperm,inperm);
[fit,diff,coord] = linfit(proxpermut,1:9)
[monprox,vaf,diff] = proxmon(proxpermut,fit)
sumfitsq = sum(sum(fit.^2));

for i = 1:100

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

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

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

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

end

fit
coord
vaf
diff
monprox


disparities = squareform(monprox);

dissimilarities = squareform(proxpermut);

distances = squareform(fit);

[dum, ord] = sortrows([disparities(:) dissimilarities(:)]);

plot(dissimilarities, distances, 'bo', dissimilarities(ord),disparities(ord),'r.-')

xlabel('Dissimilarities')

ylabel('Distances/Disparities')

legend({'Distances' 'Disparities'},'Location','NorthWest');
```

```
>> uniscale_monotone_test

fit =

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


diff =

    0.4691


coord =

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


monprox =

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

```
    1.0117    0.8089    0.7444    0.6733    0.4044    0.3030    0.1911         0    0.0478
    1.0117    0.8567    0.7922    0.7211    0.4092    0.3030    0.1774    0.0478         0


vaf =

    0.9869


diff =

    0.0349


fit =

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


coord =

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


vaf =

    0.9934
```

```
diff =

    0.0190


monprox =

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

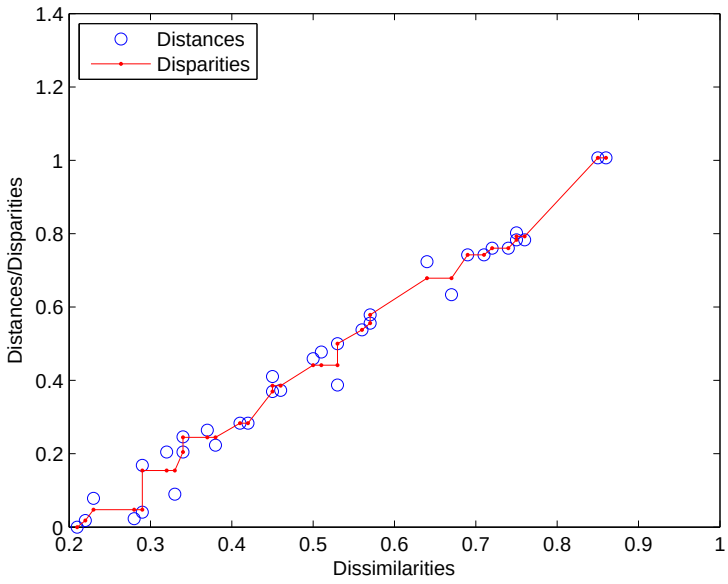


Figure 4: Plot of the Monotonically Transformed Proximities (Disparities) and Fitted Values (Distances) (y-axis) Against the Original Supreme Court Proximities (Dissimilarities) (x-axis).

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

`type uniscale_monotone_test`

The commands are performed by typing the script file name in the command window. We also note that a rather arbitrary number of iterations of the fitting process were carried out (i.e., one-hundred). An alternative strategy would have been to exit upon a minimal change in, say, the VAF value. Second, we show how to plot the entries in the various matrices (`monprox`, `supreme_agree`, and `fit`) by first changing them to vector form through the MATLAB M-function, `squareform.m`. Figure 4 then shows the plot of both `monprox` (called "disparities") and `fit` (called "distances"0 against `supreme_agree` (called "dissimilarities"). Such a (joint) plot is called a Shepard diagram in the literature of multidimensional scaling.

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

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

```
>> load supreme_agree.dat

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

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

6 iterations, Final stress criterion = 0.213021

4 iterations, Final stress criterion = 0.511745

4 iterations, Final stress criterion = 0.213232

4 iterations, Final stress criterion = 0.222266

5 iterations, Final stress criterion = 0.373536

2 iterations, Final stress criterion = 0.607790

2 iterations, Final stress criterion = 0.579076

2 iterations, Final stress criterion = 0.567206

3 iterations, Final stress criterion = 0.459125

2 iterations, Final stress criterion = 0.602695

*remaining starts deleted*


coord =

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


stress =

    0.2125
```

3.4 A Convenient Utility for Plotting a LUS Representation

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

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

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

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

fit =

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


diff =

    0.4691


coord =

   -0.5400
   -0.3611
   -0.2967
   -0.2256
    0.0622
    0.1611
```

```
    0.2567
    0.4478
    0.4956

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

linearlength =

    1.0356
```

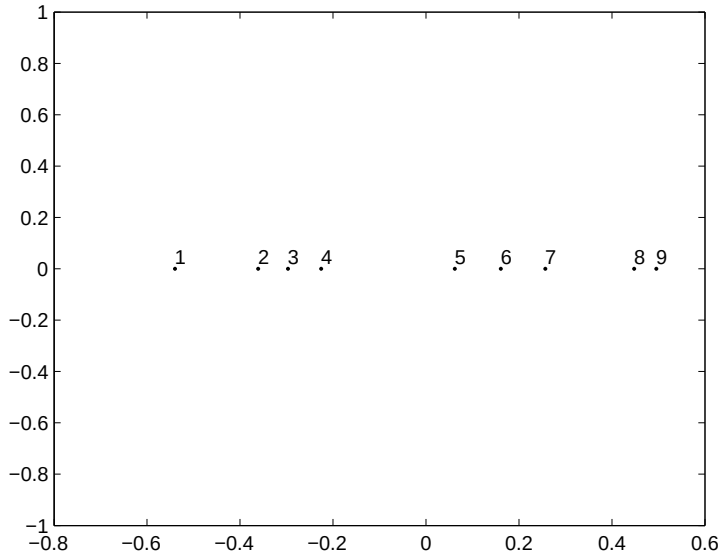


Figure 5: The LUS Representation Using linearplot.m with the Coordinates Obtained from linfit.m on the supreme_agree Proximities.

4 Incorporating an Additive Constant in LUS

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

form

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

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

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

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

The explicit inclusion of two constants, c_1 and c_2, in (7) rather than adding these two together and including a single additive constant, c, deserves some additional introductory explanation. As would be the case in fitting a single LUS structure using the loss function in (6), two interpretations exist for the role of the additive constant, c. We could consider $\{|x_j - x_i|\}$ to be fitted to the translated proximities $\{p_{ij} + c\}$, or alternatively, $\{|x_j - x_i| - c\}$ to be fitted to the original proximities $\{p_{ij}\}$, where the constant c becomes part of the actual model. Although these two interpretations do not lead to any algorithmic differences in how we would proceed with minimizing the loss function in (6), a consistent use of the second interpretation suggests that we frame extensions to the use of multiple LUS structures as we did in (7), where it is explicit that the constants c_1 and c_2 are part of the actual models to be fitted to the (untransformed) proximities $\{p_{ij}\}$. Once c_1 and c_2 are obtained, they could be summed as $c = c_1 + c_2$, and an interpretation

made that we have attempted to fit a transformed set of proximities $\{p_{ij} + c\}$ by the sum $\{|x_{j1} - x_{i1}| + |x_{j2} - x_{i2}|\}$ (and in this latter case, a more usual terminology would be one of a two-dimensional scaling (MDS) based on the city-block distance function). However, such a further interpretation is unnecessary and could lead to at least some small terminological confusion in further extensions that we might wish to pursue. For instance, if some type of (optimal nonlinear) transformation, say $f(\cdot)$, of the proximities is also sought (e.g., a monotonic function of some form as we did in Section 3.2), in addition to fitting multiple LUS structures, and where p_{ij} in (7) is replaced by $f(p_{ij})$, and $f(\cdot)$ is to be constructed, the first interpretation would require the use of a "doubly transformed" set of proximities $\{f(p_{ij}) + c\}$ to be fitted by the sum $\{|x_{j1} - x_{i1}| + |x_{j2} - x_{i2}|\}$. In general, it seems best to avoid the need to incorporate the notion of a double transformation in this context, and instead merely consider the constants c_1 and c_2 to be part of the models being fitted to a transformed set of proximities $f(p_{ij})$.

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

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

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

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

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

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

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

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

fit =

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


vaf =

    0.9796


coord =

   -0.3462
   -0.2158
   -0.1998
   -0.1771
    0.0622
```

```
    0.1127
    0.1598
    0.3021
    0.3021


addcon =

   -0.2180

>> supreme_agree_stan = proxstd(supreme_agree,0.0)

supreme_agree_stan =

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

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

fit =

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


vaf =

    0.9796


coord =
```

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


addcon =

    1.5480
```

4.2 Increasing the Computational Speed of the M-file, linfitac.m

The iterative projection optimization process implemented in `linfitac.m`, alternates between finding the additive constant and the coordinates. As discussed in Section 9.7, it is also possible to fit the LUS model through partitions consistent with a given object order, and without alternating between the additive constant and coordinate estimation. This can dramatically increase the speed of `linfitac.m` if this alternative computational routine is used for the optimization engine (e.g., some 75 times as fast in a few test problems we have tried). We have implemented this alternative in the M-file, `linfitac_altcomp.m`, with exactly the same syntax as `linfitac.m` (note that for usage elsewhere and in comparison to `linfitac.m`, the `addcon` in `linfitac_altcomp.m` is opposite in sign and is also included in `fit`). The reader is encouraged to use this alternative M-file whenever the problem size warrants and computational quickness is at a high premium.

5 Circular Unidimensional Scaling (CUS)

Circular unidimensional scaling (CUS) has the objective of placing n objects around a closed continuum such that the reconstructed distance between each pair of objects, defined by the minimum length over the two possible

paths that join the objects, reflects the given proximities as well as possible. Explicitly, and in analogy with the loss function for linear unidimensional scaling (LUS), we wish to find a set of coordinates, $x_1, \ldots, x_n$, plus an $(n+1)^{st}$ value (a circumference), $x_0 \geq |x_j - x_i|$ for all $1 \leq i \neq j \leq n$, minimizing

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

or equivalently,

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

where c is again some constant to be estimated. The value x_0 represents the total length of the closed continuum, and the expression, $\min\{|x_j - x_i|, x_0 - |x_j - x_i|\}$, gives the minimum length over the two possible paths joining objects O_i and O_j.

5.1 The Circular Unidimensional Scaling Utilities

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

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

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

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

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

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

where c is an estimated additive constant (assumed equal to zero in `cirfit.m`), $x_{\rho(1)} \leq x_{\rho(2)} \leq \cdots \leq x_{\rho(n)} \leq x_0$, and the last coordinate, x_0, is the circumference of the circular structure. We can obtain these latter coordinates from the adjacent spacings in the output matrix `fit`. As an example, we applied `cirfit.m` to the `morse_digits` proximity matrix with an assumed identity input permutation; the spacings around the circular structure between the placements for objects 1 and 2 is .5337; 2 and 3: .7534; 3 and 4: .6174; 4 and 5: .1840; 5 and 6: .5747; 6 and 7: .5167; 7 and 8: .3920; 8 and 9: .5467; 9 and 10: .1090; and back around between 10 and 1: .5594 (the sum of all these adjacent spacings is 4.787 and is the circumference (x_0) of the circular structure). For `cirfitac.m`, the additive constant was estimated as -.8031 with a `vaf` of .7051; here, the spacings around the circular structure between the placements for objects 1 and 2 is .2928; 2 and 3: .4322; 3 and 4: .2962; 4 and 5: .0234; 5 and 6: .3338; 6 and 7: .2758; 7 and 8: .2314; 8 and 9: .2800; 9 and 10: .0000; and back around between 10 and 1: .2124 (here, x_0 has a value of 2.378).

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

morse_digits =

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

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

fit =

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


diff =
```

```
    7.3898

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

fit =

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


vaf =

    0.7051


addcon =

   -0.8031
```

5.1.1 The M-function, unicirac.m

The function M-file, `unicirac.m`, carries out a circular unidimensional scaling of a symmetric dissimilarity matrix (with the estimation of an additive constant) using an iterative quadratic assignment strategy (and thus, can be viewed as an analogue of `uniscalqa.m` for the LUS task). We begin with an equally-spaced circular target constructed using the M-file, `targcir.m` (that could be invoked with the command `targcir(10)`), a (random) starting permutation, and then use a sequential combination of the pairwise interchange/rotation/insertion heuristics; the target matrix is re-estimated based on the identified (locally optimal) permutation. The whole process is repeated until no changes can be made in the target or the identified (locally optimal) permutation. The explicit usage syntax is

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

where the various terms should now be familiar. The given starting permutation, `inperm`, is of the first n integers (assumed to be around the circle); `find` is the least-squares optimal matrix (with variance-accounted-for of `vaf`) to `prox` having the appropriate circular form for the row and column object ordering given by the final permutation, `outperm`. The spacings between the

objects are given by the entries immediately above the main diagonal in `find` (and the extreme $(1, n)$ entry in `find`). The block size in the use the iterative quadratic assignment routine is `kblock`; the additive constant for the model is given by `addcon`.

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

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

find =

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


vaf =

    0.7190

outperm =

     4     5     6     7     8     9    10     1     2     3


addcon =

   -0.7964
```

The plotting function circularplot.m

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

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

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

As an example, Figure 6 provides an application of `circularplot.m` to the just given example of `unicirac.m`. The text output also appears below:

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

circum =

    2.4126


radius =

    0.3840


coord =

         0    0.3840
    0.0247    0.3832
    0.3107    0.2256
    0.3821   -0.0380
    0.2293   -0.3080
    0.0481   -0.3810
```

```
    0.0481   -0.3810
   -0.1649   -0.3468
   -0.3600   -0.1336
   -0.3254    0.2038


degrees =

    0.0644
    0.8783
    0.7273
    0.8315
    0.5146
    0.0000
    0.5695
    0.7716
    0.9148
    1.0113


cumdegrees =

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

6 LUS for Two-Mode (Rectangular) Proximity Data

The proximity data considered thus far for obtaining some type of structure, such as a LUS or CUS, have been assumed to be on one intact set of objects, $S = \{O_1, \ldots, O_n\}$, and complete in the sense that proximity values are present between all object pairs. Suppose now that the available proximity data are two-mode, and between two distinct object sets, $S_A = \{O_{1A}, \ldots, O_{n_a A}\}$ and $S_B = \{O_{1B}, \ldots, O_{n_b B}\}$, containing n_a and n_b objects, respectively, given by

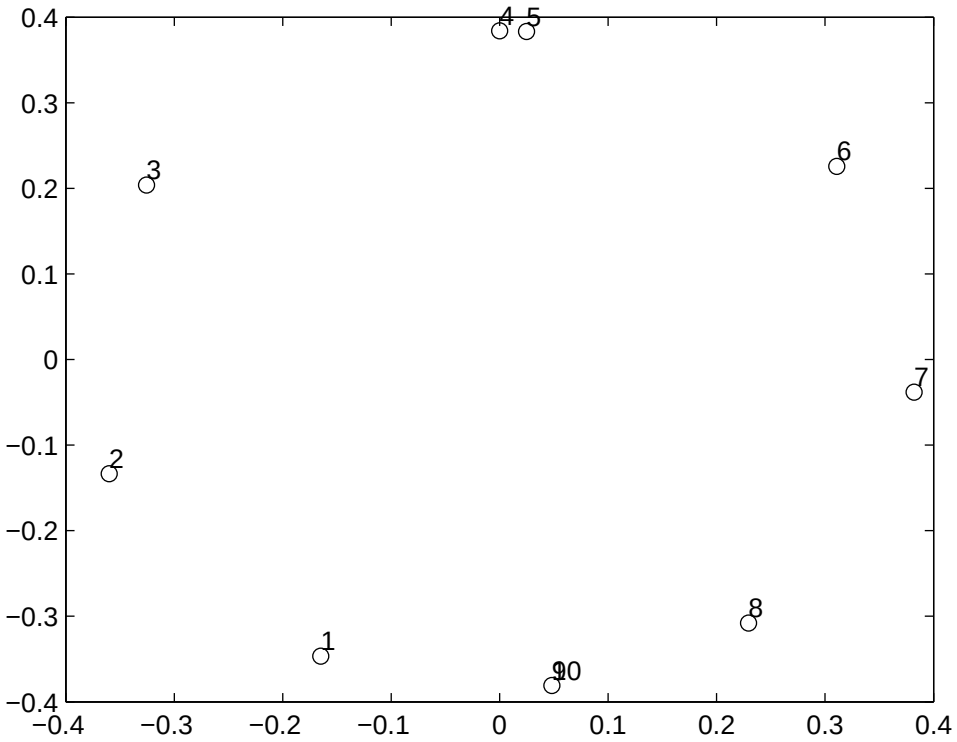


Figure 6: Two-dimensional Circular Plot for the morse_digits Data Obtained Using circularplot.m.

an $n_a \times n_b$ proximity matrix $\mathbf{Q} = \{q_{rs}\}$. Again, we assume that the entries in $\mathbf{Q}$ are keyed as dissimilarities, and a joint structural representation is desired for the combined set $S_A \cup S_B$. We might caution at the outset of the need to have legitimate proximities to make the analyses to follow very worthwhile or interpretable. There are many numerical elicitation schemes where subjects (e.g., raters) are asked to respond to some set of objects (e.g., items). If the elicitation is for, say, preference, then proximity may be a good interpretation for the numerical values. If, on the other hand, the numerical value is merely a rating given on some more-or-less objective criterion where only errors of observation induce the variability from rater to rater, then probably not.

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

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

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

6.1 Reordering Two-Mode Proximity Matrices

Given an $n_a \times n_b$ two-mode proximity matrix, $\mathbf{Q}$, defined between the two distinct sets, S_A and S_B, it may be desirable to reorder separately the rows and columns of $\mathbf{Q}$ to display some type of pattern that may be present in its entries, or to obtain some joint permutation of the n $(= n_a + n_b)$ row and column objects to effect some further type of simplified representation. These kinds of reordering tasks will be approached with a variant of the quadratic assignment heuristics of the earlier LUS discussion applied to a

square, $(n_a + n_b) \times (n_a + n_b)$, proximity matrix, $\mathbf{P}^{(tm)}$, in which a two-mode matrix, $\mathbf{Q}_{(dev)}$ and its transpose (where $\mathbf{Q}_{(dev)}$ is constructed from $\mathbf{Q}$ by deviating its entries from the mean proximity), form the upper-right- and lower-left-hand portions, respectively, with zeros placed elsewhere. (This use of zero in the presence of deviated proximities, appears a reasonable choice generally in identifying good reorderings of $\mathbf{P}^{(tm)}$. Without this type of deviation strategy, there would typically be no "mixing" of the row and column objects in the permutations that we would identify for the combined [row and column] object set.) Thus, for $\mathbf{0}$ denoting (an appropriately dimensioned) matrix of all zeros,

$$\mathbf{P}^{(tm)} = \begin{bmatrix} \mathbf{0}_{n_a \times n_a} & \mathbf{Q}_{(dev)n_a \times n_b} \\ \mathbf{Q}'_{(dev)n_b \times n_a} & \mathbf{0}_{n_b \times n_b} \end{bmatrix},$$

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

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

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

where the two-mode proximity matrix `proxtm` (with its entries deviated from the mean proximity within the use of the M-file) forms the upper-right- and lower-left-hand portions of a defined square $(n \times n)$ proximity matrix (`squareprox`) with a dissimilarity interpretation, and with zeros placed elsewhere (n = number of rows + number of columns of `proxtm` $= n_a + n_b$); three separate local operations are used to permute the rows and columns of the square proximity matrix to maximize the cross-product index with respect to a square target matrix `targ`: (a) pairwise interchanges of objects in the permutation defining the row and column order of the square proximity matrix; (b) the insertion of from 1 to `kblock` (which is less than or equal to $n - 1$) consecutive objects in the permutation defining the row and column order of the data matrix; (c) the rotation of from 2 to `kblock` (which is less than or equal to $n - 1$) consecutive objects in the permutation defining the row and column order of the data matrix. The beginning input permutation

(a permutation of the first n integers) is `inperm`; `proxtm` is the two-mode $n_a \times n_b$ input proximity matrix; `targ` is the $n \times n$ input target matrix. The final permutation of `squareprox` is `outperm`, having the cross-product index `rawindex` with respect to `targ`; `allperms` is a cell array containing `index` entries corresponding to all the permutations identified in the optimization from `allperms{1} = inperm` to `allperms{index} = outperm`.

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

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

supreme_agree5x4 =

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

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

outperm =

     1     6     2     7     3     8     4     5     9


rawindex =

   14.1420



index =
```

```
    17

>> squareprox(outperm,outperm)

ans =

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

6.2 Fitting a Two-Mode Unidimensional Scale

It is possible to fit unidimensional scales to two-mode proximity data based on a given permutation of the combined row and column object set. Specifically, if $\rho(\cdot)$ denotes some given permutation of the first n integers (where the first n_a integers denote row objects labeled $1, 2, \ldots, n_a$, and the remaining n_b integers denote column objects, labeled $n_a + 1, n_a + 2, \ldots, n_a + n_b$ $(= n)$), we seek a set of coordinates, $x_1 \leq x_2 \leq \cdots \leq x_n$, such that using the reordered square proximity matrix, $\mathbf{P}_{\rho_0}^{(tm)} = \{p_{\rho_0(i)\rho_0(j)}^{(tm)}\}$, 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 inequality constraints generated from how the row and column objects are intermixed by the given permutation $\rho_0(\cdot)$. To give a schematic representation of how these constraints are generated, suppose r_1 and r_2 (c_1 and c_2) denote two arbitrary row (column) objects, and suppose the following 2×2 matrix represents what is to be fitted to the four proximity values present between r_1, r_2 and c_1, c_2:

	c_1	c_2
r_1	a	b
r_2	c	d

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

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

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

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

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

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

inperm =

     1     6     2     7     3     8     4     5     9

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

fit =
```

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


diff =

    0.2849


rowperm =

     1
     2
     3
     4
     5


colperm =

     1
     2
     3
     4


coord =

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

In complete analogy with the LUS discussion (where the M-file, `linfitac.m`, generalizes `linfit.m` by fitting an additive constant along with the absolute

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

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

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

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

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

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

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

```
fit =

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


vaf =

    0.9911


rowperm =

     1
     2
     3
     4
     5


colperm =

     1
     2
     3
     4


addcon =

   -0.2132


coord =

         0
    0.0974
    0.1405
    0.1405
    0.3568
    0.4469
    0.4777
```

Table 6: The Two Unidimensional Scalings of the supreme_agree5x4 Data Matrix.

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

```
0.6325
0.6325
```

6.3 Increasing the Computational Speed of the M-file, linfittmac.m

Just as a computationally better (i.e., one that is faster) M-file was developed for `linfitac.m` in the form of `linfitac_altcomp`, a computational alternative to `linfittmac.m` is available as `linfittmac_altcomp.m`. Also, a complete two-mode unidimensional scaling routine incorporating `linfitmac_altcomp.m` and `ordertm.m`, is given in `uniscaltmac_altcomp.m`, with syntax:

```
[find,vaf,outperm,rowperm,colperm,addcon,coord] = ...
    uniscaltmac_altcomp(proxtm,inperm,kblock)
```

As the following output from using the `supreme_agree5x4` data matrix shows (and in contrast to what was obtained for `linfittmac.m`), the additive constant is now given with the opposite sign, the coordinates are deviated from the mean (so, they sum to 0.0), and the additive constant is included with the identified fitted matrix. For larger data sets, the use of `linfittmac_altcomp.m` and `uniscaltmac_altcomp.m` are almost mandatory. The original routines can be very slow, and with some added numerical instability and convergence difficulties when the matrices are big.

```
>> load supreme_agree5x4.dat
>> [find,vaf,outperm,rowperm,colperm,addcon,coord] = ...
 uniscaltmac_altcomp(supreme_agree5x4,randperm(9),3)

find =

    0.3106    0.3537    0.6600    0.8457
    0.2563    0.2132    0.5196    0.7052
    0.4726    0.4295    0.3033    0.4889
    0.5935    0.5504    0.2441    0.3680
    0.7483    0.7052    0.3988    0.2132


vaf =

    0.9911


outperm =

     9     5     4     8     3     7     2     6     1


rowperm =

     5
     4
     3
     2
     1


colperm =

     4
     3
     2
     1


addcon =

    0.2132
```

```
coord =

   -0.3075
   -0.3075
   -0.1527
   -0.1219
   -0.0318
    0.1845
    0.1845
    0.2275
    0.3249
```

7 The Analysis of Skew-Symmetric Proximity Matrices

Suppose the original proximity information given between the pairs of objects in $S = \{O_1, \ldots, O_n\}$ is nonsymmetric and in the form of an $n \times n$ nonnegative matrix, say $\mathbf{D} = \{d_{ij}\}$. This latter matrix will be decomposed into the sum of its symmetric and skew-symmetric components as

$$\mathbf{D} = [(\mathbf{D} + \mathbf{D}')/2] + [(\mathbf{D} - \mathbf{D}')/2] \ ,$$

and each of these components will always be addressed separately. The matrix $(\mathbf{D} + \mathbf{D}')/2$ is a nonnegative symmetric matrix (in our notation, $\mathbf{P}$), and all the methods appropriate for such a proximity measure can be applied. The second component, $(\mathbf{D} - \mathbf{D}')/2$, is an $n \times n$ matrix denoted by $\mathbf{P}^{ss} = \{p_{ij}^{ss}\}$, where $p_{ij}^{ss} = (d_{ij} - d_{ji})/2$ for $1 \leq i, j \leq n$, and the superscript "ss" signifies "skew-symmetric," i.e., $p_{ij}^{ss} = -p_{ji}^{ss}$ for all $1 \leq i, j \leq n$. Thus, there is an explicit directionality to the (dominance) relationship specified between any two objects depending on the order of the subscripts, i.e., p_{ij}^{ss} and p_{ji}^{ss} are equal in absolute magnitude but differ in sign. It may also be worth noting here that any skew-symmetric matrix $\{p_{ij}^{ss}\}$ can itself be interpreted as containing two separate sources of information: one is the directionality of the dominance relationship given by $\{\text{sign}(p_{ij}^{ss})\}$, where $\text{sign}(y) = 1$ if $y > 0$; $= 0$ if $y = 0$; and $= -1$ if $y < 0$; the second is in the absolute magnitude of dominance given by $\{\mid p_{ij}^{ss} \mid\}$. This latter matrix is symmetric and can be viewed as a dissimilarity matrix and analyzed as such.

Measures of matrix patterning for a skew-symmetric matrix, $\mathbf{P}^{ss}$, that might be derivable indirectly from the generation of some type of coordinate representation, have an interestingly different status than they do for a symmetric matrix $\mathbf{P}$. In the skew-symmetric framework, closed-form least-squares solutions are possible, thus eliminating the need for some auxiliary optimization strategy. For example, suppose we wish to find a set of n coordinate values, $x_1, \ldots, x_n$, such that the least-squares criterion,

$$\sum_{i<j} (p_{ij}^{ss} - (x_j - x_i))^2 \ ,$$

is minimized. An optimal set of coordinates can be obtained analytically from the average proximity within column j, i.e., $x_j = (1/n) \sum_i p_{ij}^{ss}$, with a minimum loss value of

$$\sum_{i<j} (p_{ij}^{ss})^2 - (1/n) \sum_j (\sum_i p_{ij}^{ss})^2 \ .$$

Also, because the sum of the entries in $\mathbf{P}^{ss}$ is zero, the variance-accounted-for (vaf) can be given simply as:

$$\text{vaf} = \frac{(1/n) \sum_j (\sum_i p_{ij}^{ss})^2}{\sum_{i<j} (p_{ij}^{ss})^2} \ .$$

Thus, an optimal row/column reordering of $\mathbf{P}^{ss}$ can be obtained just by using the order of the optimal coordinates from smallest (most negative) to largest (most positive). Similarly, if we consider an equally-spaced coordinate representation obtained by minimizing the least-squares loss function:

$$\sum_{i<j} (p_{ij}^{ss} - \alpha(x_j - x_i))^2 \ ,$$

where $x_1, \ldots, x_n$ are the integers $1, \ldots, n$ in some order, and α is some multiplicative constant to be estimated, the optimal row/column reordering of $\mathbf{P}^{ss}$ induced by the integer coordinates would again be generated by the ordering of $(1/n) \sum_i p_{ij}^{ss}$ for $1 \leq j \leq n$.

The M-file we have written to carry-out these closed-form scaling procedures on a skew-symmetric proximity matrix is `skew_symmetric_scaling.m`, with syntax:

```
[coord,sort_coord,permut,prox_permut,vaf,...
  alpha_multiplier,alpha_vaf,alpha_coord] = ...
  skew_symmetric_scaling(prox)
```

Here, COORD contains the least-squares coordinates which are sorted from smallest to largest in SORT_COORD; VAF is the variance they account for; PERMUT is the object permutation corresponding to SORT_COORD with PROX_PERMUT the reordered skew-symmetric proximity matrix. For equally-spaced coordinates, ALPHA_MULTIPLIER defines the multiplicative constant on the integer-valued coordinates; a collection of equally-spaced coordinates is given by ALPHA_COORD with ALPHA_VAF the variance they account for.

The two Thurstone skew-symmetric matrices are used in the MATLAB recording below. One can see the "gambler" offense moving from a fifth to a seventh position in the "before" and "after" analyses. It does appear that the movie has had an effect.

```
  >> load thurstone_skew_symmetric_before.dat
>> load thurstone_skew_symmetric_after.dat

>> [coord,sort_coord,permut,prox_permut,vaf,alpha_multiplier,alpha_vaf,...
alpha_coord] = skew_symmetric_scaling(thurstone_skew_symmetric_before)

coord =

    0.5708   -0.2292   -0.0215   -0.2569    0.2692    0.3185   -0.7785    0.5854   -0.8162   -0.3246   -0.1754    0.5677    0.29


sort_coord =

   -0.8162   -0.7785   -0.3246   -0.2569   -0.2292   -0.1754   -0.0215    0.2692    0.2908    0.3185    0.5677    0.5708    0.58


permut =

     9
     7
    10
     4
     2
    11
     3
     5
    13
     6
    12
     1
     8


prox_permut =

         0    0.1600    0.8200    0.9400    0.9000    0.9400    0.9600    0.9600    1.0000    0.9800    0.9800    1.0000    0.97
```

```
   -0.1600         0    0.7200    0.9800    0.8600    0.9200    0.9600    0.9600    0.9800    1.0000    1.0000    0.9800    0.92
   -0.8200   -0.7200         0    0.2600    0.1800    0.1600    0.2200    0.7600    0.8400    0.8200    0.8000    0.8800    0.84
   -0.9400   -0.9800   -0.2600         0   -0.0400    0.2400    0.5000    0.6200    0.7400    0.9000    0.8200    0.9000    0.84
   -0.9000   -0.8600   -0.1800    0.0400         0   -0.0200    0.4200    0.5200    0.6200    0.8400    0.8000    0.8600    0.84
   -0.9400   -0.9200   -0.1600   -0.2400    0.0200         0    0.1600    0.5600    0.5600    0.4800    0.9600    0.9600    0.84
   -0.9600   -0.9600   -0.2200   -0.5000   -0.4200   -0.1600         0    0.3400    0.3600    0.5000    0.7400    0.8400    0.72
   -0.9600   -0.9600   -0.7600   -0.6200   -0.5200   -0.5600   -0.3400         0    0.1000   -0.0200    0.2800    0.4600    0.40
   -1.0000   -0.9800   -0.8400   -0.7400   -0.6200   -0.5600   -0.3600   -0.1000         0         0    0.4600    0.5800    0.38
   -0.9800   -1.0000   -0.8200   -0.9000   -0.8400   -0.4800   -0.5000    0.0200         0         0    0.3600    0.4200    0.58
   -0.9800   -1.0000   -0.8000   -0.8200   -0.8000   -0.9600   -0.7400   -0.2800   -0.4600   -0.3600         0   -0.4600    0.28
   -1.0000   -0.9800   -0.8800   -0.9000   -0.8600   -0.9600   -0.8400   -0.4600   -0.5800   -0.4200    0.4600         0
   -0.9700   -0.9200   -0.8400   -0.8400   -0.8400   -0.8400   -0.7200   -0.4000   -0.3800   -0.5800   -0.2800         0


vaf =

    0.9126


alpha_multiplier =

    0.1205


alpha_vaf =

    0.8690


alpha_coord =

   -0.7231
   -0.6026
   -0.4821
   -0.3616
   -0.2410
   -0.1205
         0
    0.1205
    0.2410
    0.3616
    0.4821
    0.6026
    0.7231

>> [coord,sort_coord,permut,prox_permut,vaf,alpha_multiplier,alpha_vaf,...
alpha_coord] = skew_symmetric_scaling(thurstone_skew_symmetric_after)

coord =

    0.5446    0.0200   -0.0492   -0.2569    0.2815    0.2677   -0.7446    0.5508   -0.8138   -0.3908   -0.2092    0.5138    0.28


sort_coord =

   -0.8138   -0.7446   -0.3908   -0.2569   -0.2092   -0.0492    0.0200    0.2677    0.2815    0.2862    0.5138    0.5446    0.55


permut =

     9
     7
    10
     4
```

```
    11
     3
     2
     6
     5
    13
    12
     1
     8


prox_permut =

         0    0.2800    0.7400    0.9200    0.9000    0.9600    0.9200    0.9600    0.9800    0.9600    0.9800    1.0000    0.98
   -0.2800         0    0.5800    0.8800    0.8800    0.9400    0.9000    0.9600    0.9600    0.9600    0.9600    0.9600    0.98
   -0.7400   -0.5800         0    0.3400    0.2800    0.3600    0.4600    0.8000    0.7800    0.7800    0.8400    0.8800    0.88
   -0.9200   -0.8800   -0.3400         0    0.0600    0.4000    0.4000    0.7400    0.6800    0.6800    0.8400    0.9000    0.78
   -0.9000   -0.8800   -0.2800   -0.0600         0    0.2400    0.2800    0.5200    0.6200    0.5200    0.9400    0.9400    0.78
   -0.9600   -0.9400   -0.3600   -0.4000   -0.2400         0    0.0200    0.4000    0.4400    0.4000    0.6800    0.8600    0.74
   -0.9200   -0.9000   -0.4600   -0.4000   -0.2800   -0.0200         0    0.3800    0.2800    0.3800    0.4600    0.5800    0.64
   -0.9600   -0.9600   -0.8000   -0.7400   -0.5200   -0.4000   -0.3800         0         0    0.0200    0.4000    0.4000    0.46
   -0.9800   -0.9600   -0.7800   -0.6800   -0.6200   -0.4400   -0.2800         0         0    0.0800    0.3000    0.3400    0.36
   -0.9600   -0.9600   -0.7800   -0.6800   -0.5200   -0.4000   -0.3800   -0.0200   -0.0800         0    0.2800    0.4600    0.32
   -0.9800   -0.9600   -0.8400   -0.8400   -0.9400   -0.6800   -0.4600   -0.4000   -0.3000   -0.2800         0   -0.2400    0.24
   -1.0000   -0.9600   -0.8800   -0.9000   -0.9400   -0.8600   -0.5800   -0.4000   -0.3400   -0.4600    0.2400         0
   -0.9800   -0.9800   -0.8800   -0.7800   -0.7800   -0.7400   -0.6400   -0.4600   -0.3600   -0.3200   -0.2400         0


vaf =

    0.9340


alpha_multiplier =

    0.1164


alpha_vaf =

    0.8927


alpha_coord =

   -0.6982
   -0.5819
   -0.4655
   -0.3491
   -0.2327
   -0.1164
         0
    0.1164
    0.2327
    0.3491
    0.4655
    0.5819
    0.6982
```

Given a skew-symmetric matrix $\{p_{ij}^{ss}\}$ (or possibly, just $\{\text{sign}(p_{ij}^{ss}\})$, an obvious class of measures for matrix pattern based on matrix reordering would

be the sum of the above-diagonal entries:

$$\sum_{i<j} p_{ij}^{ss} (\text{or} \sum_{i<j} \text{sign}(p_{ij}^{ss})) \ .$$

In fact, because $p_{ij}^{ss} = \text{sign}|p_{ij}^{ss}|$, the index $\sum_{i<j} p_{ij}^{ss}$ can be interpreted merely as a weighted version of the one based just on $\text{sign}(p_{ij}^{ss})$. We will assume in our discussion below that $\sum_{i<j} p_{ij}^{ss}$ is being considered, but the obvious replacement of p_{ij}^{ss} by $\text{sign}(p_{ij}^{ss})$ could incorporate the use of $\sum_{i<j} \text{sign}(p_{ij}^{ss})$ directly.

A dynamic programming solution to the optimization task of reordering the rows/columns of a matrix to maximize the sum of above-diagonal entries, was first sketched by Lawler (1964) in identifying minimum feedback arc sets in a directed graph. The M-file that we use below on the Thurstone skew-symmetric matrices, `skew_symmetric_lawler_dp.m`, implements the dynamic programming recursion. The optimization task itself, however, has several other distinct substantive incarnations, e.g., in maximum likelihood paired comparison ranking (Flueck and Korsh, 1974), or to triangulating an input-output matrix (Korte and Oberhofer, 1971). For an extensive review of the variety of possible applications up through the middle 1970's, the reader is referred to Hubert (1976).

The syntax of `skew_symmetric_lawler_dp.m` is as follows:

```
[permut,prox_permut,cumobfun] = skew_symmetric_lawler_dp(prox)
```

Here, `PROX` is the input skew-symmetric proximity matrix (with a zero main diagonal); `PERMUT` is the order of the objects in the optimal permutation; `PROX_PERMUT` is the reordered proximity matrix using `PERMUT`; `CUMOBFUN` gives the cumulative values of the objective function for the successive placements of the objects in the optimal permutation. The verbatim output below is an implementation of `skew_symmetric_lawler_dp.m` on the two Thurstone "before" and "after"' skew-symmetric proximity matrices. The offense of "gambler" (number 2) moves appropriately after the movie has been shown.

```
   >> [permut,prox_permut,cumobfun] = skew_symmetric_lawler_dp(thurstone_skew_symmetric_before)
Elapsed time is 0.258336 seconds.
```

```
permut =

     9
     7
    10
     2
     4
    11
     3
     6
     5
    13
     1
    12
     8


prox_permut =

         0    0.1600    0.8200    0.9000    0.9400    0.9400    0.9600    0.9800    0.9600    1.0000    1.0000    0.9800    0.97
   -0.1600         0    0.7200    0.8600    0.9800    0.9200    0.9600    1.0000    0.9600    0.9800    0.9800    1.0000    0.92
   -0.8200   -0.7200         0    0.1800    0.2600    0.1600    0.2200    0.8200    0.7600    0.8400    0.8800    0.8000    0.84
   -0.9000   -0.8600   -0.1800         0    0.0400   -0.0200    0.4200    0.8400    0.5200    0.6200    0.8600    0.8000    0.84
   -0.9400   -0.9800   -0.2600   -0.0400         0    0.2400    0.5000    0.9000    0.6200    0.7400    0.9000    0.8200    0.84
   -0.9400   -0.9200   -0.1600    0.0200   -0.2400         0    0.1600    0.4800    0.5600    0.5600    0.9600    0.9600    0.84
   -0.9600   -0.9600   -0.2200   -0.4200   -0.5000   -0.1600         0    0.5000    0.3400    0.3600    0.8400    0.7400    0.72
   -0.9800   -1.0000   -0.8200   -0.8400   -0.9000   -0.4800   -0.5000         0    0.0200         0    0.4200    0.3600    0.58
   -0.9600   -0.9600   -0.7600   -0.5200   -0.6200   -0.5600   -0.3400   -0.0200         0    0.1000    0.4600    0.2800    0.40
   -1.0000   -0.9800   -0.8400   -0.6200   -0.7400   -0.5600   -0.3600         0   -0.1000         0    0.5800    0.4600    0.38
   -1.0000   -0.9800   -0.8800   -0.8600   -0.9000   -0.9600   -0.8400   -0.4200   -0.4600   -0.5800         0    0.4600
   -0.9800   -1.0000   -0.8000   -0.8000   -0.8200   -0.9600   -0.7400   -0.3600   -0.2800   -0.4600   -0.4600         0    0.28
   -0.9700   -0.9200   -0.8400   -0.8400   -0.8400   -0.8400   -0.7200   -0.5800   -0.4000   -0.3800         0   -0.2800


cumobfun =

         0
    0.1600
    1.7000
    3.6400
    5.8600
    8.1000
   11.3200
   16.8400
   21.5800
   26.7800
   34.6600
   42.3200
   49.9300

>> [permut,prox_permut,cumobfun] = skew_symmetric_lawler_dp(thurstone_skew_symmetric_after)
Elapsed time is 0.234843 seconds.

permut =

     9
     7
    10
     4
    11
     3
     2
     5
     6
```

```
    13
     1
    12
     8


prox_permut =

         0    0.2800    0.7400    0.9200    0.9000    0.9600    0.9200    0.9800    0.9600    0.9600    1.0000    0.9800    0.98
   -0.2800         0    0.5800    0.8800    0.8800    0.9400    0.9000    0.9600    0.9600    0.9600    0.9600    0.9600    0.98
   -0.7400   -0.5800         0    0.3400    0.2800    0.3600    0.4600    0.7800    0.8000    0.7800    0.8800    0.8400    0.88
   -0.9200   -0.8800   -0.3400         0    0.0600    0.4000    0.4000    0.6800    0.7400    0.6800    0.9000    0.8400    0.78
   -0.9000   -0.8800   -0.2800   -0.0600         0    0.2400    0.2800    0.6200    0.5200    0.5200    0.9400    0.9400    0.78
   -0.9600   -0.9400   -0.3600   -0.4000   -0.2400         0    0.0200    0.4400    0.4000    0.4000    0.8600    0.6800    0.74
   -0.9200   -0.9000   -0.4600   -0.4000   -0.2800   -0.0200         0    0.2800    0.3800    0.3800    0.5800    0.4600    0.64
   -0.9800   -0.9600   -0.7800   -0.6800   -0.6200   -0.4400   -0.2800         0         0    0.0800    0.3400    0.3000    0.36
   -0.9600   -0.9600   -0.8000   -0.7400   -0.5200   -0.4000   -0.3800         0         0    0.0200    0.4000    0.4000    0.46
   -0.9600   -0.9600   -0.7800   -0.6800   -0.5200   -0.4000   -0.3800   -0.0800   -0.0200         0    0.4600    0.2800    0.32
   -1.0000   -0.9600   -0.8800   -0.9000   -0.9400   -0.8600   -0.5800   -0.3400   -0.4000   -0.4600         0    0.2400
   -0.9800   -0.9600   -0.8400   -0.8400   -0.9400   -0.6800   -0.4600   -0.3000   -0.4000   -0.2800   -0.2400         0    0.24
   -0.9800   -0.9800   -0.8800   -0.7800   -0.7800   -0.7400   -0.6400   -0.3600   -0.4600   -0.3200         0   -0.2400


cumobfun =

         0
    0.2800
    1.6000
    3.7400
    5.8600
    8.7600
   11.7400
   16.4800
   21.2400
   26.0200
   33.3400
   40.2600
   47.4200
```

8 Order-Constrained Partition Construction

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

order will be constructed with the (heuristic) unidimensional scaling routine, `uniscalqa.m`, discussed in Section 2.2.

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

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

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

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

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

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

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

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

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

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

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

8.1 The Dynamic Programming Implementation

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

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

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

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

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

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

  PROX is the input proximity matrix (with a zero main diagonal
  and a dissimilarity interpretation); LINCON is the given
```

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



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

membership =

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


objectives =

         0
   73.8432
   83.2849
   86.9479
   88.1095
   88.6861
   89.0559
```

```
   89.2241
   89.3166


permmember =

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


clusmeasure =

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


cluscoord =

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


residsumsq =
```

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

To interpret this example further, it appears that five ordered classes may be a good "stopping point" for the clustering process — moving to four gives a noticeable drop in the achievable objective function value. The fifth row of `membership` is the vector `5 4 4 4 3 2 2 1 1`, and thus the justice partitioning of {{St},{Br, Gi, So},{Oc},{Ke,Re},{Sc,Th}}, clearly placing O'Connor in a separate "swing" class. The `objectives` value for this partition is 88.1095, and can be reconstructed from the values in `clusmeasure` that delineate the extent of the various classes, i.e., the values in this matrix at positions (1,1), (2,4), (5,5), (6,7), (8,9): 23.6196 + 21.0675 + .3136 + 7.0688 + 36.0401 = 88.1096 ($\approx$ 88.1095 to rounding). The coordinates for the classes are given in these same positions in the matrix `cluscoord`: $-.5400$; $-.2944$; .0622; .2089; .4717. Weighting these by the class sizes of 1, 3, 1, 2, 2, respectively, and then summing, gives the value (to rounding) of 0.0 (i.e., the constraint $\sum_k n_k x_k^* = 0$ is satisfied). Finally, the residual sum-of-squares of .6033 is reconstructible from (12) as $10.3932 - (1/9)88.1096$, where 10.3932 is $\sum_{i<j} p_{ij}^2$, and is always given as the first entry in `residsumsq` corresponding to only one class that must be placed at a coordinate value of 0.0 (because of the constraint $\sum_k n_k x_k^* = 0$).

8.2 Two Utility Functions For Coordinate Estimation

When constructing ordered partitions through optimizing the measure in (10), the coordinates were generated as a byproduct through the closed-form expression in (11). For some extensions we contemplate, and particularly to multiple dimensions and the imposition of ordered partitions on each, it would be useful to have a fitting mechanism that would not depend on the presence of nonnegative proximities. To this end we provide two util-

ity M-files, `linfit_tied.m` and `linfitac_tied.m`, that for a given ordered partition and underlying constraining object order, will fit the M coordinates, $x_1 \leq \cdots \leq x_M$ (and an additional additive constant c in the case of `linfitac_tied.m`) by minimizing

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

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

```
>> load supreme_agree.dat

>> help linfit_tied.m

   LINFIT_TIED does a confirmatory fitting of a given
   unidimensional order using Dykstra's
   (Kaczmarz's) iterative projection least-squares method. This
   includes the possible imposition of tied coordinates.

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

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



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

fit =

         0    0.2456    0.2456    0.2456    0.6022    0.7489    0.7489    1.0117    1.0117
    0.2456         0         0         0    0.3567    0.5033    0.5033    0.7661    0.7661
```

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


diff =

    0.6032


coord =

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

>> help linfitac_tied.m

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

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

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

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

fit =

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


vaf =

    0.9671


coord =

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


addcon =

   -0.2297
```

As can be seen in the preceding output, the coordinates given earlier for the five-class ordered partition can be retrieved using `linfit_tied.m` along with the least-squares loss value, `diff`, of 0.6032 (this is within a .0001 rounding

error of the previously given five-class residual sum-of-squares of 0.6033). In incorporating the estimation of an additive constant, c, as part of the model that is fitted with `linfitac_tied`, a legitimate variance-accounted-for (VAF) measure can be given and used in place of the unnormalized least-squares loss value. Here, we would define the VAF measure as

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

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

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

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

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

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

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

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

>> for i = 1:9

tiedcoord = permmember(10-i,:);

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

 fits{i} = fit;

 vafs{i} = vaf;


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

 end

>> for i = 1:9

 vaf = vafs{i}

end

vaf =

    0.9796

... (output deleted)

vaf =

    0.6759


vaf =

     0
```

8.3 Extensions to Generalized Ultrametrics

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

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

tion of nonnegative weights $\alpha_1, \ldots, \alpha_{T-1}$, just as in the more restrictive ultrametric context. Thus, although the same number of weights may be needed to construct $\mathbf{P}_\alpha$ as for an ultrametric, the structures definable through ordered partitions restricted only by the class contiguity constraint are broader than those possible through the concept of an ultrametric.

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

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

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

membership =

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

```
>> help orderpartitionfit.m

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

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

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



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

fit =

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


weights =

    0.2388
    0.0383
    0.0533
    0.0550
         0
         0
```

```
    0.0450
    0.2100


vaf =

    0.7339
```

9 Some Possible LUS and CUS Generalizations

9.1 Additive Representation Through Multiple Structures

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

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

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

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

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

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

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

9.2 Individual Differences

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

9.3 Incorporating Transformations of the Proximities

In the use of either a one- or two-mode proximity matrix, the data were assumed 'as is', and without any preliminary transformation. It was noted that some analyses leading to negative values might be more pleasingly interpretable if an additive constant could be fitted along with the LUS or CUS structures. In other words, the structures fit to proximity matrices then have an invariance with respect to linear transformations of the proximities. A more general transformation will be discussed briefly in a later section where a centroid (metric), fit as part of the whole representational structure, has the effect of double-centering (i.e., making the rows and columns sum to

zero). Considering the input proximity matrix deviated from the centroid, zero sums are present within rows or columns. The analysis methods could iterate between fitting a LUS or CUS structure and a centroid, attempting to squeeze out every last bit of VAF. A more direct strategy (and one that would most likely not affect substantive interpretations materially) would be to initially double-center (either a one- or two-mode matrix), and then treat the later to the analyses we wish to carry out, without again revisiting the double-centering operation during the iterative process.

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

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

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

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

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

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

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

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

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

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

As an illustration of the M-files just introduced, Table 7 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 LUS structure obtained with `linfitac_missing.m` has VAF of 86.78%, and is given in Figure 7 plotted with `linearplot`. Notice that because of the missing values in `fit`, the coordinates were entered 'by hand' in the vector `coord` before plotted with `linearplot`.

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

supreme_agree_2005_6 =

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

>> supreme_agree_2005_6_missing

supreme_agree_2005_6_missing =

     0     1     1     1     1     1     1     1     1     1
     1     0     1     1     1     1     1     1     1     1
     1     1     0     1     1     1     1     1     1     1
     1     1     1     0     1     1     1     1     1     1
     1     1     1     1     0     1     1     1     0     1
     1     1     1     1     1     0     1     1     1     1
     1     1     1     1     1     1     0     1     1     1
     1     1     1     1     1     1     1     0     1     1
```

```
     1     1     1     1     0     1     1     1     0     1
     1     1     1     1     1     1     1     1     1     0

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

outperm =

     1     4     3     2     5     6     7     8     9    10

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

fit =

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


vaf =

    0.8678

addcon =

   -0.1567

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

coord =

         0    0.0967    0.1553    0.1620    0.3146    0.4873    0.5783    0.5783    0.5983    0.6490

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

inperm =

     1     4     3     2     5     6     7     8     9    10

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

linearlength =

    0.6490
```

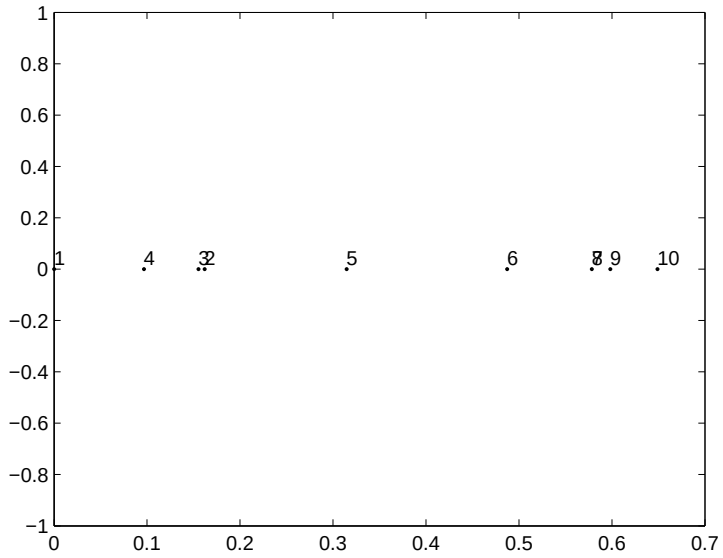


Figure 7: The LUS Representation for the supreme_agree_2005_6 Proximities Using linearplot.m with the Coordinates Constructed from linfitac_missing.m.

9.5 Obtaining Good Object Orders Through a Dynamic Programming Strategy

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

The syntax for this optimization strategy is

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

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

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

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

permut =

     1
     2
     3
     4
     5
     6
     7
     8
     9


cumobfun =

   23.6196
   34.1821
   41.3110
   45.4319
```

```
45.7455
47.8480
53.1841
69.4250
89.3166
```

9.6 Extending LUS and CUS Representations Through Additively Imposed Centroid Matrices

In a companion Toolbox on Cluster Analysis (Hubert, Köhn, & Steinley, 2009), the notion of a matrix representing an additive tree was introduced and characterized by a certain four-point condition that its entries must satisfy. Alternatively, it was noted that any such matrix could be represented (in many ways) as a sum of two matrices, say $\mathbf{U} = \{u_{ij}\}$ and $\mathbf{C} = \{c_{ij}\}$, where $\mathbf{U}$ is an ultrametric matrix (and whose entries satisfy a certain three-point condition), and $c_{ij} = g_i + g_j$ for $1 \leq i \neq j \leq n$, and $c_{ii} = 0$ for $1 \leq i \leq n$, based on some set of values, $g_1, \ldots, g_n$. We will call $\mathbf{C}$ a centroid metric, for convenience (and will continue to do so even though some entries in $\mathbf{C}$ may be negative because of possible negative values for $g_1, \ldots, g_n$).

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

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

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

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

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

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

fit =

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


vaf =

    0.1908


lengths =

    0.3700    0.2486    0.2343    0.2171    0.1957    0.1857    0.1943    0.3114    0.3129

>> residual_supreme_agree = supreme_agree - fit

residual_supreme_agree =
```

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

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

fit =

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


vaf =

    0.1090


lengths =

    0.3080
    0.1980
    0.1880
    0.1905
    0.2880
    0.2375
    0.2275
    0.1895
    0.2835

>> residual_supreme_agree5x4 = supreme_agree5x4 - fit

residual_supreme_agree5x4 =

   -0.2455   -0.1655    0.1425    0.2685
   -0.1555   -0.2055    0.1225    0.2385
```

```
    0.0245    0.0345   -0.0475   -0.0115
    0.1420    0.1420   -0.1500   -0.1340
    0.2345    0.1945   -0.0675   -0.3615

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

outperm =

     9     8     7     6     5     2     4     3     1

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

fit =

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


vaf =

    0.9288


coord =

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

```
addcon =

    0.2232


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

     6     1     7     2     3     8     4     5     9

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

fit =

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


vaf =

    0.9394


rowperm =

     1
     2
     3
     4
     5


colperm =

     1
     2
     3
     4
```

```
addcon =

    0.2094


coord =

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

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

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

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

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

```
>> [find,vaf,outperm,targone,targtwo,lengthsone,coordtwo,addcontwo] = ...
```

```
cent_linearfnd(supreme_agree,randperm(9))

find =

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


vaf =

    0.9856


outperm =

     8     9     7     6     5     2     4     1     3


targone =

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


targtwo =

         0         0    0.1383    0.1974    0.2486    0.4855    0.5000    0.5235    0.5295
         0         0    0.1383    0.1974    0.2486    0.4855    0.5000    0.5235    0.5295
    0.1383    0.1383         0    0.0591    0.1103    0.3472    0.3617    0.3852    0.3912
    0.1974    0.1974    0.0591         0    0.0512    0.2881    0.3026    0.3261    0.3321
    0.2486    0.2486    0.1103    0.0512         0    0.2369    0.2514    0.2749    0.2809
```

```
    0.4855    0.4855    0.3472    0.2881    0.2369         0    0.0146    0.0380    0.0440
    0.5000    0.5000    0.3617    0.3026    0.2514    0.0146         0    0.0234    0.0294
    0.5235    0.5235    0.3852    0.3261    0.2749    0.0380    0.0234         0    0.0060
    0.5295    0.5295    0.3912    0.3321    0.2809    0.0440    0.0294    0.0060         0


lengthsone =

    0.2353    0.2367    0.2168    0.2335    0.2508    0.2699    0.2322    0.3683    0.2266


coordtwo =

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


addcontwo =

    0.2611

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

find =

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


vaf =
```

```
    0.9841


outperm =

     1     2     3     4     5     6     7     8     9


targone =

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


targtwo =

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


lengthsone =

    0.3663    0.2578    0.2457    0.2363    0.2490    0.2317    0.2149    0.2334    0.2349


coordtwo =

   -0.2316
   -0.2186
   -0.2156
```

```
   -0.1976
    0.0410
    0.0922
    0.1513
    0.2895
    0.2895


addcontwo =

    0.2574
```

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

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

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

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

As an example below, the least-squares fitting of the identity permutation on `supreme_agree` with `linfitac.m` is replicated with `partitionfit.m`. The central matrix is `member`, where the first eight rows correspond to the eight separations between the justices along the line (in the jargon of graph theory, we have eight 'cuts' of a graph, each defined by two disjoint [and exhaustive] subsets, and characterized by a 0/1 dissimilarity matrix with 1's indicating objects present across the two separate subsets). The last row of `member` is

the disjoint partition representing an additive constant, and producing a single 0/1 dissimilarity matrix with all 1's in the off-diagonal positions. Thus, to move from one object to another along the continuum, the various 'gaps' must be traversed separating the two objects. To construct the approximation, the weights attached to the gaps are summed to produce the complete path; an additional additive constant is then imposed. Because we are using nonnegative least-squares to obtain that weights and the additive constant, an obtained zero value for the additive constant (i.e., we have an estimation at the boundary) suggests the need to augment the original proximities by a positive value before `partitionfit.m` is used.

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

fit =

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


vaf =

    0.9796


coord =

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

```
    0.3021
    0.3021


addcon =

   -0.2180

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

member =

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

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

fitted =

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


vaf =

    0.9796
```

```
weights =

    0.1304
    0.0160
    0.0227
    0.2393
    0.0504
    0.0471
    0.1423
         0
    0.2180


end_condition =

     0
```

In addition to `partitionfit.m`, an M-file is available, `partitionfit_addcon.m`, that fits an additive constrant without subjecting it to a nonnegativity constraint. The syntax is similar to `partitionfit.m`, but it has one additional output value, `addcon`, representing the unrestricted additive constant:

```
[fitted,vaf,weights,addcon,end_condition] = ...
  partitionfit_addcon(prox,member)
```

For two-mode matrices, there are the M-files, `partitionfit_twomode.m` and `partitionfit_twomode_addcon.m`, with self-explanatory syntaxes:

```
[fitted,vaf,weights,end_condition] = ...
  partitionfit_twomode(proxtm,member)

[fitted,vaf,weights,addcon,end_condition] = ...
  partitionfit_twomode_addcon(proxtm,member)
```

9.8 Concave and Convex Monotonic (Isotonic) Regression

This section gives four M-files that implement extensions to the monotonic regression task by imposing additional convexity and/or concavity restrictions on the regression function. The basic optimization problem involves two $n \times 1$ vectors, say $\mathbf{x} = \{x_i\}$ (considered as an independent variable) and

$\mathbf{y} = \{y_i\}$ (considered as a dependent variable). We wish to find an $n \times 1$ vector $\hat{\mathbf{y}} = \{\hat{y}_i\}$, such that the least-squares criterion,

$$\sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \ ,$$

is minimized, subject to $\hat{\mathbf{y}}$ satisfying some particular set of constraints. For convenience, and without-loss-of-any-generality, assume the entries in $\mathbf{x}$ are indexed, so: $x_i \leq x_{i+1}$ for $i = 1, \ldots, n-1$. (It is easier to specify the type of additional constraints we wish to impose if this initial ordering is assumed. In the various M-files introduced below, the input arguments are $\mathbf{x}$ and $\mathbf{y}$, but no preliminary ordering of the values in $\mathbf{x}$ is needed (or assumed).)

Monotonic Regression (A) if $x_i < x_{i+1}$, for $i = 1, \ldots, n-1$, then $\hat{y}_i \leq \hat{y}_{i+1}$;

Convex Monotonic Regression (B) in addition to Condition (A), if $x_{i-1} < x_i < x_{i+1}$, for $i = 2, \ldots, n-1$,

$$\hat{y}_i \leq (\frac{x_{i+1} - x_i}{x_{i+1} - x_{i-1}})\hat{y}_{i-1} + (\frac{x_i - x_{i-1}}{x_{i+1} - x_{i-1}})\hat{y}_{i+1} \ .$$

Condition (B) implies that the fitted value, $\hat{y}_i$, associated with x_i, lies at or below the line segment connecting $(x_{i-1}, \hat{y}_{i-1})$ and $(x_{i+1}, \hat{y}_{i+1})$. Or, equivalently, the slope of the line segment connecting $(x_{i-1}, \hat{y}_{i-1})$ and $(x_i, \hat{y}_i)$ is no greater than that connecting $(x_i, \hat{y}_i)$ and $(x_{i+1}, \hat{y}_{i+1})$.

Concave Monotonic Regression (C) in addition to Condition (A), if $x_{i-1} < x_i < x_{i+1}$, for $i = 2, \ldots, n-1$,

$$\hat{y}_i \geq (\frac{x_{i+1} - x_i}{x_{i+1} - x_{i-1}})\hat{y}_{i-1} + (\frac{x_i - x_{i-1}}{x_{i+1} - x_{i-1}})\hat{y}_{i+1} \ .$$

As can be seen, Condition (C) merely changes the sense of the inequality in (B), and implies that the fitted value, $\hat{y}_i$, associated with x_i, lies at or above the line segment connecting $(x_{i-1}, \hat{y}_{i-1})$ and $(x_{i+1}, \hat{y}_{i+1})$. Or, equivalently, the slope of the line segment connecting $(x_{i-1}, \hat{y}_{i-1})$ and $(x_i, \hat{y}_i)$ is no less than that connecting $(x_i, \hat{y}_i)$ and $(x_{i+1}, \hat{y}_{i+1})$.

Convex-Concave Monotonic Regression To define Condition (D), a monotonic convex function is fit up to the median object (based on the assumed ordered objects in $\mathbf{x}$) and a concave monotonic function thereafter (so, an "ogive-like" shape is given). Explicitly, an `upper_limit` is defined on the object triples considered in fitting convexity, and a `lower_limit` on the triples considered in fitting concavity. For n odd, `upper_limit` $= (n+3)/2$; `lower_limit` $= (n-1)/2$; for n even, `upper_limit` $= (n/2)+1$; `lower_limit` $= n/2$.

The four M-files implementing these four fitting constraints have more or less the same syntax, and are named:

Condition (A): `monotonic_regression_dykstra.m`
Condition (B): `convex_monotonic_regression_dykstra.m`
Condition (C): `concave_monotonic_regression_dykstra.m`
Condition (D): `convex_concave_monotonic_regression_dykstra.m`

For example, syntax for `convex_concave_monotonic_regression_dykstra.m` is

```
[yhat,vaf_yhat] = convex_concave_monotonic_regression_dykstra(x,y)
```

with the vector of values in `YHAT` weakly monotonic with respect to the values in the independent input vector `X` (which is *not* necessarily ordered); defining a convex function up to the median observation in `X` and a concave function thereafter; and which minimizes the least-squares loss measure, $\sum_i (y_i - \hat{y}_i)^2$. The variance-accounted-for in `Y` from `YHAT` is given by `VAF_YHAT`, and defined as

$$1 - \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2} \ ,$$

where $\bar{y}$ is the mean value in $\mathbf{y}$.

The least-squares optimization strategy implemented in the four M-files of this section are all based on an iterative projection strategy developed by Dykstra (1983). The first routine dealing with monotonic regression per se, `monotonic_regression_dykstra.m`, is a direct competitor to one that would implement a "pool adjacent violators" algorithm; the latter has been the standard strategy ever since implemented by Kruskal (1964a,b) in his approach to nonmetric multidimensional scaling. For example, the MATLAB Statistical

Toolbox has an M-file for nonmetric multidimensional scaling, `mdscale.m`. that relies on a (private) function, `lsqisotonic.m`, which is a least-squares isotonic regression strategy built on the "pool adjacent violators" method. To see how one of the four M-files of this section would work in `mdscale.m`, the call to `lsqisotonic.m` could be replaced by one of these. In effect, we can easily implement a nonmetric multidimensional scaling routine that would allow constraints beyond monotonic to convex, concave, or a combination of convex and concave. Generally, these further constraints seem to "smooth" out the monotonic regression function quite dramatically; this has been the goal of using monotonic splines in exactly this same context (e.g., see Ramsay, 1988). In effect, we have solved with these mechanisms rather directly and without splines, one of the major problems identified by Shepard (1974) in the Psychometric Society Presidential Address:

> The problem that remains, however, is to impose the desired condition of smoothness in some general and nonarbitrary way without having to specify a particular functional form — such as the polynomial (which can become nonmonotonic) or the exponential (which is often too restrictive) (p. 398).

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

The Dykstra-Kaczmarz (DK) method for solving (in)equality constrained least-squares tasks, can be characterized as a very general iterative projection strategy that considers each linear constraint successively (and repeatedly), until convergence. It serves as *the* method for all of our least-squares fitting requirements in the present Toolbox. We first review very cryptically in the section below, the basic argument for the iterative strategy; this then serves as the template that must be specialized for a particular data analysis context and followed in detail. For example, when fitting structures such as ultrametrics, additive trees, anti-Robinson forms, and the like, the fitting procedure reviewed is directly programmed into an M-file. A second section discusses a general MATLAB M-file based on converting a Fortran subroutine from Wollan and Dykstra (1987). This is a general optimization routine that can be used in the spirit of other such M-files from the Optimization Toolbox.

It is written very generally, and not for just one particular application (as we have done, for example, in fitting ultrametrics with the dedicated M-file, `ultrafit.m`). It accepts general arguments to define the constrained least-squares task, such as the characterizing constraint matrix, and produces a solution. At times, a general implementation such as this may be problematic if, say, the constraint matrix is too enormous in size. In these cases, there may be no other choice than to program the specific application without storing the constraint matrix.

9.9.1 A Review of the DK Strategy

The Kaczmarz (1937) method deals only with equality constraints; Dykstra's (1983) method generalizes the restrictions that can be handled by Kaczmarz's strategy to the use of inequality constraints.

Kaczmarz's method can be characterized as follows:

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

Dykstra's method can be characterized as follows:

Given $\mathbf{A} = \{a_{ij}\}$ of order $m \times n$, $\mathbf{x}_0' = \{x_{01}, \ldots, x_{0n}\}$, $\mathbf{b}' = \{b_1, \ldots, b_m\}$, and $\mathbf{w}' = \{w_1, \ldots, w_n\}$, where $w_j > 0$ for all j, find $\mathbf{x}_0^*$ such that $\mathbf{a}_i'\mathbf{x}_0^* \leq b_i$ for $1 \leq i \leq m$ and $\sum_{j=1}^{n} w_j(x_{0j} - x_{0j}^*)^2$ is minimized. Again, (re)define the (closed convex) sets $C_i = \{\mathbf{x} \mid \sum_{j=1}^{n} a_{ij}x_j \leq b_i, 1 \leq j \leq n\}$ and when a vector $\mathbf{y} \notin C_i$, its projection onto C_i (in the metric defined by the weight vector $\mathbf{w}$) is $\mathbf{y} - (\mathbf{a}_i'\mathbf{y} - b_i)\mathbf{a}_i\mathbf{W}^{-1}(\mathbf{a}_i'\mathbf{W}^{-1}\mathbf{a}_i)^{-1}$, where $\mathbf{W}^{-1} = \text{diag}\{w_1^{-1}, \ldots, w_n^{-1}\}$. We again initialize the process with the vector $\mathbf{x}_0$ and each set $C_1, \ldots, C_m$ is considered in turn. If the vector being carried forward to this point when

C_i is (re)considered does not satisfy the constraint defining C_i, a projection onto C_i occurs. The sets $C_1, \ldots, C_m$ are cyclically and repeatedly considered but with one difference from the operation of Kaczmarz's method — each time a constraint set C_i is revisited, any changes from the previous time C_i was reached are first "added back". This last process ensures convergence to a (globally) optimal solution $\mathbf{x}_0^*$.

9.9.2 A General M-file for Solving Linear Inequality Constrained Least-Squares Tasks

The Fortran subroutine that we have rewritten into an M-file from Wollan and Dykstra (1987) is called `least_squares_dykstra.m`, and is meant to find a solution to the following problem:

suppose $\mathbf{g}$ is a given $n \times 1$ vector (the "dependent" vector); $\mathbf{S}$ a given $n \times n$ positive-definite matrix (thus, it has an inverse; it may, for example, be a variance-covariance matrix, in which case we are minimizing Mahalanobis distances in the formulation below); $\mathbf{A}$ is a known $m \times n$ matrix where each row of $\mathbf{A}$ defines one of m constraints; $\mathbf{b}$ is a known $m \times 1$ vector giving the right-hand-sides of the inequality constraints. The task: find an $n \times 1$ vector $\mathbf{x}$, subject to $\mathbf{Ax} \leq \mathbf{b}$, to minimize

$$(\mathbf{g} - \mathbf{x})' \mathbf{S}^{-1} (\mathbf{g} - \mathbf{x}) \ . \tag{14}$$

Commonly, $\mathbf{S}$ will be the identity, in which case we have unweighted least-squares; if $\mathbf{S}$ is diagonal, weighted least-squares is obtained. We also note that in the actual application of the M-file, equality constraints can be enforced directly.

The syntax for the M-file, `least_squares_dykstra.m`, is as follows:

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

Here, the input arguments are an $n \times 1$ vector, `DATA`; an $n \times n$ positive-definite matrix, `COVARIANCE`; an $m \times n$ constraint matrix, `CONSTRAINT_ARRAY`; the $m \times 1$ right-hand-side constraint vector, `CONSTRAINT_CONSTANT`; and an $m \times 1$

EQUALITY_FLAG vector with values of 0 when a corresponding constraint is an inequality, and 1 if an equality. The weighted least-squares criterion (with weights defined by the inverse of COVARIANCE) is minimized by a SOLUTION that satisfies CONSTRAINT_ARRAY * SOLUTION being in EQUALITY_FLAG relation to CONSTRAINT_CONSTANT. As additional output arguments, there is a $m \times 1$ KUHN_TUCKER vector (useful for some applications); the number of ITERATIONS taken (maximum default value is ITERMAX $= 1.0e+04$, set in the program), and an END_CONDITION flag: 0: no error; 1: ITERMAX exceeded; 2: invalid constant; 3: invalid constraint function.

9.9.3 A Few Applications of least_squares_dykstra.m

There are a few very useful application of least_squares_dykstra.m that we review briefly here:

(A) Consider the minimization of

$$(\mathbf{g} - \mathbf{x})' \mathbf{S}^{-1} (\mathbf{g} - \mathbf{x}) \ , \tag{15}$$

subject to

$$\mathbf{x} = \sum_{i=1}^{m} \alpha_i \mathbf{d}_i, \ \alpha_i \geq 0 \text{ for } 1 \leq i \leq m,$$

and the specified collection of $n \times 1$ vectors, $\mathbf{d}_1, \ldots, \mathbf{d}_k$. To use the general M-file for this task, define the constraint array to be,

$$\mathbf{A} = \begin{bmatrix} \mathbf{d}_1' \mathbf{S}^{-1} \\ \vdots \\ \mathbf{d}_m' \mathbf{S}^{-1} \end{bmatrix} .$$

Letting $\mathbf{x}^*$ denote the solution of the original task in (17), the solution here is given by $\mathbf{g} - \mathbf{x}^*$, with the α_i's being 1/2 the values in the Kuhn-Tucker vector. We might note that these fitting ideas were implemented in the M-file, partitionfit.m, used heavily in the Cluster Analysis Toolbox to generalize the notion of an ultrametric.

(B) Consider the usual linear model formulation: $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \epsilon$, where $\mathbf{y}$ is the $n \times 1$ vector of observations, $\mathbf{X}$ an $n \times m$ full-rank design matrix, and ϵ contains n independent, $N(0, \sigma^2)$, random variables. If we wish to impose

linear inequality constraints on the least-squares estimate of the $m \times 1$ vector $\boldsymbol{\beta}$, say, $\hat{\boldsymbol{\beta}}$: $\mathbf{A}\hat{\boldsymbol{\beta}} \leq \mathbf{b}$, in our original problem formulation in (18), we can let $\mathbf{S}^{-1} = \mathbf{X}'\mathbf{X}$, $\mathbf{g} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$, and use the given constraint array and vector, $\mathbf{A}$ and $\mathbf{b}$.

If we wish only nonnegativity of $\hat{\boldsymbol{\beta}}$, then (18) can be used with $\mathbf{g} = \mathbf{y}$, $\mathbf{S} = \mathbf{I}$, and the $\mathbf{d}_1, \ldots, \mathbf{d}_m$ given by the k columns of $\mathbf{X}$. When halved, the minimizing α_i's provide the desired nonnegative estimate, $\hat{\boldsymbol{\beta}} \geq \mathbf{0}$.

9.10 The L_1 Fitting of Unidimensional Scales (with an Additive Constant)

The linear unidimensional scaling task in the L_1 norm can be phrased as one of finding a set of coordinates, $x_1, \ldots, x_n$, such that the L_1 criterion,

$$\sum_{i<j} |p_{ij} - (|x_j - x_i| - c)| \ , \tag{16}$$

is minimized, where we now immediately include the possibility of an additive constant in the model (in what follows, c can just be set to 0 for the more elemental model without an additive constant). As an alternative reformulation of the optimization task in (16) that will prove convenient as a point of departure in our development of computational routines (much as what we did within the L_2 norm), we subdivide (16) into the two separate problems of finding a set of n numbers, $x_1 \leq \cdots \leq x_n$, and a permutation on the first n integers, $\rho(\cdot) \equiv \rho$, for which

$$\sum_{i<j} |p_{\rho(i)\rho(j)} - ((x_j - x_i) - c)| \tag{17}$$

is minimized. Again, we can impose the additional constraint that $\sum_{i=1}^{n} x_i = 0$.

Assuming for now that the permutation ρ is given, the task of finding $x_1 \leq \cdots \leq x_n$ to minimize (17) is a linear programming problem. Without loss of generality, we let ρ be the identity permutation and first rewrite $\sum_{i<j} |p_{ij} - (|x_j - x_i| - c)|$ as the loss criterion $\sum_{i<j}(z_{ij}^{+} + z_{ij}^{-})$, where

$$z_{ij}^{+} = \frac{1}{2}\{|p_{ij} - (|x_j - x_i| - c)| - (p_{ij} - (|x_j - x_i| - c))\};$$

$$z_{ij}^{-} = \frac{1}{2}\{|p_{ij} - (|x_j - x_i| - c)| + (p_{ij} - (|x_j - x_i| - c))\},$$

for $1 \leq i < j \leq n$. The unknowns are c, $x_1, \ldots, x_n$, and for $1 \leq i < j \leq n$, z_{ij}^{+}, z_{ij}^{-}, and y_{ij} ($\equiv |x_j - x_i|$). The constraints of the linear program take the form:

$$-z_{ij}^{+} + z_{ij}^{-} + y_{ij} - c = p_{ij};$$

$$-x_j + x_i + y_{ij} = 0;$$

$$z_{ij}^{+} \geq 0, z_{ij}^{-} \geq 0, y_{ij} \geq 0,$$

for $1 \leq i < j \leq n$, and

$$x_1 + \cdots + x_n = 0.$$

The MATLAB Functions linfitl1.m and linfitl1ac.m Based on the linear programming reformulation just given for finding a set of ordered coordinates for a fixed object permutation, the M-functions, `linfitl1.m` and `linfitl1ac.m`, where the latter includes an additive constant in the model and the former does not, serve to setup the relevant (constraint) matrices for the associated linear programming task; the actual linear programming optimization is carried out by invoking `linprog.m` from the MATLAB Optimization Toolbox.

The syntax for `linfitl1.m` is

```
[fit diff coord exitflag] = linfitl1(prox,inperm)
```

where if we denote the given permutation as $\rho^0(\cdot)$ (`INPERM`), we seek a set of coordinates $x_1 \leq \cdots \leq x_n$ (`COORD`) to minimize (at a value of `DIFF`)

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

`FIT` refers to the matrix $\{|x_j - x_i|\}$, and `EXITFLAG` describes the exit condition of the linear program optimization (greater than 0 for convergence; 0 denotes the maximum number of function evaluations or iterations was exceeded; less than 0 indicates a failure of convergence to a solution). For using `linfitl1ac.m`, the syntax is

```
[fit dev coord addcon exitflag] = linfitl1ac(prox,inperm)
```

Here, we minimize

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

where c is given by `ADDCON` and `DEV` refers to the deviance(-accounted-for) defined by the normalized L_1 loss value:

$$\text{DEV} = 1 - \frac{\sum_{i<j} |p_{\rho^0(i)\rho^0(j)} - (|x_j - x_i| - c)|}{\sum_{i<j} |p_{ij} - p_{med}|},$$

where p_{med} is the median of the off-diagonal proximity values.

We illustrate the use of `linfitl1.m` and `linfitl1ac.m` on the `supreme_agree` proximity matrix using the identity permutation as the input object order.

```
>> inperm = 1:9

inperm =

     1     2     3     4     5     6     7     8     9

>> [fit,diff,coord,exitflag] = linfitl1(supreme_agree,inperm)
Optimization terminated.

fit =

         0    0.1911    0.2582    0.3367    0.6198    0.7045    0.8119    1.0092    1.0727
    0.1911         0    0.0671    0.1456    0.4287    0.5134    0.6207    0.8181    0.8816
    0.2582    0.0671         0    0.0785    0.3616    0.4463    0.5537    0.7511    0.8145
    0.3367    0.1456    0.0785         0    0.2831    0.3678    0.4752    0.6726    0.7360
    0.6198    0.4287    0.3616    0.2831         0    0.0847    0.1921    0.3895    0.4529
    0.7045    0.5134    0.4463    0.3678    0.0847         0    0.1073    0.3047    0.3681
    0.8119    0.6207    0.5537    0.4752    0.1921    0.1073         0    0.1974    0.2608
    1.0092    0.8181    0.7511    0.6726    0.3895    0.3047    0.1974         0    0.0634
    1.0727    0.8816    0.8145    0.7360    0.4529    0.3681    0.2608    0.0634         0


diff =

    3.4600


coord =
```

```
   -0.5560
   -0.3649
   -0.2978
   -0.2193
    0.0638
    0.1485
    0.2559
    0.4532
    0.5167


exitflag =

     1

>> [fit,dev,coord,addcon,exitflag] = linfitl1ac(supreme_agree,inperm)
Optimization terminated.

fit =

         0    0.3239    0.3490    0.3664    0.6134    0.6588    0.7185    0.8649    0.8705
    0.3239         0    0.2287    0.2461    0.4931    0.5385    0.5983    0.7446    0.7503
    0.3490    0.2287         0    0.2211    0.4681    0.5135    0.5732    0.7196    0.7252
    0.3664    0.2461    0.2211         0    0.4507    0.4961    0.5558    0.7022    0.7078
    0.6134    0.4931    0.4681    0.4507         0    0.2490    0.3088    0.4551    0.4608
    0.6588    0.5385    0.5135    0.4961    0.2490         0    0.2634    0.4098    0.4154
    0.7185    0.5983    0.5732    0.5558    0.3088    0.2634         0    0.3500    0.3557
    0.8649    0.7446    0.7196    0.7022    0.4551    0.4098    0.3500         0    0.2093
    0.8705    0.7503    0.7252    0.7078    0.4608    0.4154    0.3557    0.2093         0


dev =

    0.8763


coord =

   -0.3484
   -0.2282
   -0.2031
   -0.1857
    0.0613
    0.1067
    0.1664
```

```
    0.3128
    0.3184


addcon =

   -0.2037


exitflag =

     1
```

9.10.1 Iterative Linear Programming

Given the availability of the two linear programming based M-functions for fitting given unidimensional scales defined by specific input object permutations, it is possible to embed these two routines in a search strategy for actually finding the (at least hopefully) best such permutations in the first place. This embedding is analogous to adopting iterative quadratic assignment in `uniscalqa.m` and attempting to locate good unidimensional scalings in the L_2 norm. Here, we have an iterative use of linear programming in `uniscallp.m` and `uniscallpac.m` to identify the good unidimensional scales in the L_1 norm, without and with, respectively, an additive constant in the fitted model. The usage syntax of both M-functions are as follows:

```
[outperm coord diff fit] = uniscallp(prox,inperm)
[outperm coord dev fit addcon] = uniscallpac(prox,inperm)
```

Both M-functions begin with a given object ordering (`INPERM`) and evaluate the effect of pairwise object interchanges on the current permutation carried forward to that point. If an object interchange is identified that improves the L_1 loss value, that interchange is made and the changed permutation becomes the current one. When no pairwise object interchange can reduce `DIFF` in `uniscallp.m`, or increase `DEV` in `uniscallpac.m` over its current value, that ending permutation is provided as `OUTPERM` along with its coordinates (`COORD`) and the matrix `FIT` (the absolute differences of the ordered coordinates). In `uniscallpac.m`, the additive constant (`ADDCON`) is also given.

The numerical example that follows relies on `supreme_agree` to provide the proximity matrix, and initializes both the M-functions with the random permutations. From other random starts that we have tried, the resulting scales we give below are (almost undoubtedly) L_1-norm optimal. We might note that the (optimal) object orderings differ depending on whether or not an additive constant is included in the model, and the one without an additive constant involves a small departure from the identity permutation (an interchange of O'Connor (5) and Kennedy (6)).

```
>> load supreme_agree.dat
>> [outperm,coord,diff,fit] = uniscallp(supreme_agree,randperm(9))

outperm =

     1     2     3     4     6     5     7     8     9


coord =

   -0.5539
   -0.3210
   -0.2942
   -0.2342
    0.0803
    0.1259
    0.2638
    0.4456
    0.4875


diff =

    3.4000


fit =

         0    0.2329    0.2597    0.3197    0.6342    0.6798    0.8177    0.9995    1.0414
    0.2329         0    0.0268    0.0868    0.4013    0.4469    0.5849    0.7666    0.8085
    0.2597    0.0268         0    0.0600    0.3745    0.4201    0.5580    0.7398    0.7817
    0.3197    0.0868    0.0600         0    0.3145    0.3601    0.4980    0.6798    0.7217
    0.6342    0.4013    0.3745    0.3145         0    0.0456    0.1835    0.3653    0.4072
    0.6798    0.4469    0.4201    0.3601    0.0456         0    0.1379    0.3197    0.3616
```

```
    0.8177    0.5849    0.5580    0.4980    0.1835    0.1379         0    0.1818    0.2237
    0.9995    0.7666    0.7398    0.6798    0.3653    0.3197    0.1818         0    0.0419
    1.0414    0.8085    0.7817    0.7217    0.4072    0.3616    0.2237    0.0419         0

>> [outperm,coord,dev,fit,addcon] = uniscallpac(supreme_agree,randperm(9))

outperm =

     9     8     7     6     5     4     3     2     1


coord =

   -0.3184
   -0.3128
   -0.1664
   -0.1067
   -0.0613
    0.1857
    0.2031
    0.2282
    0.3484


dev =

    0.8763


fit =

         0    0.2093    0.3557    0.4154    0.4608    0.7078    0.7252    0.7503    0.8705
    0.2093         0    0.3500    0.4098    0.4551    0.7022    0.7196    0.7446    0.8649
    0.3557    0.3500         0    0.2634    0.3088    0.5558    0.5732    0.5983    0.7185
    0.4154    0.4098    0.2634         0    0.2490    0.4961    0.5135    0.5385    0.6588
    0.4608    0.4551    0.3088    0.2490         0    0.4507    0.4681    0.4931    0.6134
    0.7078    0.7022    0.5558    0.4961    0.4507         0    0.2211    0.2461    0.3664
    0.7252    0.7196    0.5732    0.5135    0.4681    0.2211         0    0.2287    0.3490
    0.7503    0.7446    0.5983    0.5385    0.4931    0.2461    0.2287         0    0.3239
    0.8705    0.8649    0.7185    0.6588    0.6134    0.3664    0.3490    0.3239         0


addcon =

   -0.2037
```

Although the M-functions are provided to either fit or find the best unidimensional scalings in the L_1 norm, we do not suggest their routine use. The finding of the best unidimensional scales in L_1 is extremely expensive computationally (given the use of the repetitive linear programming subtasks), and without any obvious advantage over L_2, it is not clear why the L_1 approach should be pursued. This same set of conclusions exist as well for the L_1 finding and fitting of multidimensional unidimensional scales. (We might also mention a possible issue with ill-conditioning for some uses of `linfitl1.m` and `linfitl1ac.m` if the simplex option is chosen [and not the default interior-point algorithm] for the optimization method implemented in `linprog.m`. The end results appear generally to be fine, but the intermediate warnings in either the finding or fitting of these unidimensional scales within L_1 is disconcerting. So the use of the default interior-point strategy is recommended whenever `linprog.m` is called.)

9.10.2 The L_1 Finding and Fitting of Multiple Unidimensional Scales

In analogy to the L_2 fitting of multiple unidimensional structures, the use of the L_1 norm can again be done by (repetitive) successive residualization, but now with a reliance on the M-function, `linfitl1ac.m`, to fit each separate unidimensional structure with its additive constant. The M-function, `biscallp.m`, is a two-(or bi-)dimensional scaling strategy for the L_1 loss-function

$$\sum_{i<j} |p_{ij} - [|x_{j1} - x_{i1}| - c_1] - [|x_{j2} - x_{i2}| - c_2]|, \tag{18}$$

with syntax (and all variables) similar to `biscalqa.m`, including a provision for the confirmatory fitting of two given input orders (by setting `NOPT = 0`). As noted earlier, in principle one can use the L_1 norm in city-block scaling, but given the increased computational expense of doing so with no apparent advantage over L_2, it is suggested that the use of L_1 generally be avoided in favor of L_2.

9.11 The Confirmatory Fitting of Tied Coordinate Patterns in Bidimensional City-Block Scaling

One of the options in the M-file, `biscalqa.m`, allows the confirmatory fitting of two given input permutations by setting the `nopt` switch to zero. An extension of this M-file to `biscalqa_tied.m` is presented here that further allows the imposition of given patterns of tied coordinates along the two axes. Equal integer values that are present in `tiedcoordone` and `tiedcoordtwo` impose equal coordinate values in the optimization process.

We give two examples below of using `biscalqa_tied.m` on the Hampshire proximities. The first illustration does a confirmatory fitting of the two object permutations that led to the best VAF identified earlier of .9499 and with no imposed tied coordinates (so both `tiedcoordone` and `tiedcoordtwo` consist of the integers from 1 to 27). The exact same VAF is obtained in the confirmatory fit, but interestingly, there are slight variations in the additive constants and the coordinates at the level of the third or fourth decimal places. So we have small variations that apparently "cancel" and lead to the same best VAF as before. The second illustration imposes tied coordinates along both axes in groups of threes, so `tiedcoordone` and `tiedcoordtwo` both have the pattern of `[1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6 7 7 7 8 8 8 9 9 9]`. There is an expected drop in the VAF to .8734 (but not too severe given the major reduction in model complexity). The resulting configuration of Hampshire towns with these tied coordinate structures imposed is given in Figure 8.

```
load hampshire_proximities.dat

inpermone = [2 6 12 19 22 1 11 4 5 23 21 18 3 27 20 24 17 10 25 9 26 13 8 16 7 15 14];
inpermtwo = [18 4 19 16 3 11 15 17 5 14 27 7 8 13 12 21 20 6 2 22 9 10 25 23 1 26 24];

[outpermone,outpermtwo,coordone,coordtwo,fitone,fittwo,addconone,addcontwo,vaf] = ...
    biscalqa_tied(hampshire_proximities,targlin(27),targlin(27),...
    inpermone,inpermtwo,1:27,1:27,3,0)

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

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

[outpermone,outpermtwo,coordone,coordtwo,fitone,fittwo,addconone,addcontwo,vaf] = ...
    biscalqa_tied(hampshire_proximities,targlin(27),targlin(27),inpermone,inpermtwo,...
    tiedcoordone,tiedcoordtwo,3,0)

    axes = zeros(27,2);

        for i = 1:27
```

```
            axes(outpermone(i),1) = coordone(i);
            axes(outpermtwo(i),2) = coordtwo(i);
        end

        plot(axes(1:27,1),axes(1:27,2),'ko')

        hold on

        for i = 1:27

            objectlabels{i,1} = int2str(i);

        end

        text(axes(1:27,1),axes(1:27,2),objectlabels,'fontsize',10,...
        'verticalalignment','bottom')


>> script_tied_coordinates_biscalqa

outpermone =

  Columns 1 through 26

     2     6    12    19    22     1    11     4     5    23    21    18     3    27    20    24    17    10    25     9    26

  Column 27

    14


outpermtwo =

  Columns 1 through 26

    18     4    19    16     3    11    15    17     5    14    27     7     8    13    12    21    20     6     2    22     9

  Column 27

    24


coordone =

  -18.7759
  -14.5533
  -13.3525
  -12.7337
  -10.7167
  -10.4930
   -9.1591
   -7.5284
   -6.3342
   -5.0568
   -4.5961
   -3.9200
   -0.5998
   -0.5998
    1.0959
    2.9431
    4.1361
    4.3892
    5.0837
    6.1704
```

```
    6.7689
    7.1508
   12.0935
   13.8611
   15.3806
   17.2922
   22.0538


coordtwo =

   -7.1706
   -5.9026
   -5.3941
   -4.9316
   -4.9034
   -4.3384
   -3.5486
   -3.5486
   -2.3517
   -2.1241
   -1.8610
   -0.8060
   -0.7654
   -0.6970
   -0.0768
    0.8199
    0.9487
    1.8872
    2.3601
    2.7219
    3.3431
    3.5634
    4.0580
    5.9584
    6.8936
    7.4461
    8.4196



addconone =

   -6.4555


addcontwo =

    5.0634


vaf =

    0.9499


outpermone =

  Columns 1 through 26

     2     6    12    19    22     1    11     4     5    23    21    18     3    27    20    24    17    10    25     9    26

  Column 27
```

```
    14


outpermtwo =

  Columns 1 through 26

    18     4    19    16     3    11    15    17     5    14    27     7     8    13    12    21    20     6     2    22     9

  Column 27

    24


coordone =

  -14.9857
  -14.9857
  -14.9857
  -10.6768
  -10.6768
  -10.6768
   -7.2769
   -7.2769
   -7.2769
   -4.6149
   -4.6149
   -4.6149
   -0.0553
   -0.0553
   -0.0553
    3.6327
    3.6327
    3.6327
    5.6893
    5.6893
    5.6893
   10.6332
   10.6332
   10.6332
   17.6544
   17.6544
   17.6544


coordtwo =

   -6.1101
   -6.1101
   -6.1101
   -4.9751
   -4.9751
   -4.9751
   -2.6275
   -2.6275
   -2.6275
   -1.5461
   -1.5461
   -1.5461
   -0.2714
   -0.2714
   -0.2714
    0.9779
```

```
    0.9779
    0.9779
    3.0910
    3.0910
    3.0910
    4.2537
    4.2537
    4.2537
    7.2075
    7.2075
    7.2075


addconone =

   -7.1024


addcontwo =

    4.8676


vaf =

    0.8734
```

9.12 Matrix Color Coding and Presentation

A current method of data presentation is through the use of color, where a collection of varying numerical values is mapped into a corresponding collection of colors that can then be displayed. A `colormap` is an $m \times 3$ matrix of real numbers between 0.0 and 1.0, with each row giving an RGB (Red/Green/Blue) vector defining a single color. For example, if `map` is the name of the $m \times 3$ matrix, then the k^{th} row, `map(k,:) = [r(k) g(k) b(k)]`, provides the intensities of red, green, and blue components for the k^{th} color.

A number of built-in colormaps are available in MATLAB (or, alternatively, custom colormaps can be constructed by a user, possibly through the MATLAB GUI, `colormapeditor`); all of these ready-made colormaps accept a value of m for the number of rows (number of colors) of the colormap. For example, the command, `colormap(bone(256))`, produces a (built-in) `bone` colormap with 256 colors; the latter is a grayscale with a higher value of the blue component, adding an "electronic" look to the grayscale images. Asking for MATLAB help on "colormap" will produce the collection of supported colormaps (e.g., `bone`, `gray`, `copper`, `spring`, among many others).

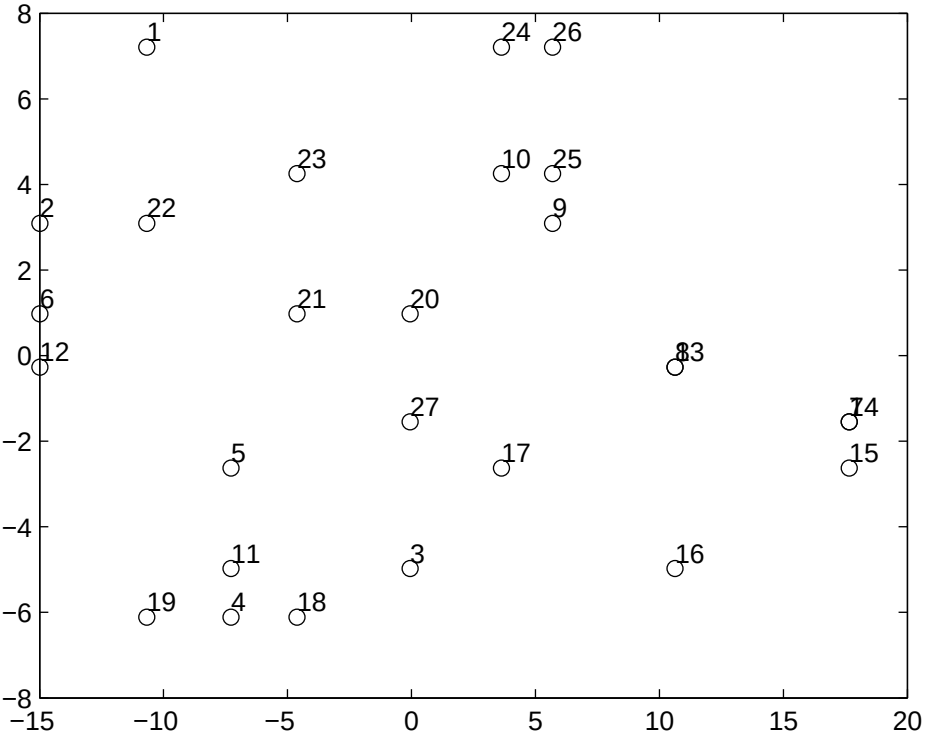


Figure 8: The City-Block Scaling of the Hampshire Towns Based on the Given Tied Coordinate Patterns (in Groups of Three) Obtained with biscalqa_tied.m.

The MATLAB function, `pcolor`, is used to produce our checkerboard colorcoded plot of the entries in a given matrix. The minimum and maximum matrix entries are assigned the first and last colors in the colormap, with the remaining entries determined by a linear mapping from the numerical values to the colormap elements. The MATLAB command, `colorbar`, reproduces a legend giving a visual correspondence between color and numerical value. Also, the MATLAB command, `axis ij`, puts the representation into "matrix axes mode", where the coordinate system origin is at the upper-left corner. As is standard for a matrix, the i axis is vertical and numbered from top to bottom; the j axis is horizontal and numbered from left to right.

The M-file we have written, `matrix_colorcode.m`, has syntax:

```
matrix_colorcode(datamatrix,rowperm,colperm,cmap)
```

Here, a color representation is constructed for the values in an $n \times m$ matrix, `DATAMATRIX`. The rows and columns of `DATAMATRIX` are permuted by `ROWPERM` and `COLPERM`, respectively, with `CMAP` the input colormap for the representation (e.g., `bone(256)`).

There are several other uses of the `pcolor` command that are interesting from a data analytic perspective. One is to produce a movie that traces the course of some optimization method. For example, the quadratic assignment routine, `order.m`, produces a cell array of permutations that transform a symmetric data matrix to be as close as possible to a given target matrix. To show the effects of these permutations in a color-coded manner, we have the M-file, `matrix_movie.m`, with the syntax:

```
matrix_movie(datamatrix,perms,numperms,cmap)
```

A color movie is constructed of the effects of a series of permutations on an $n \times n$ symmetric proximity matrix, `DATAMATRIX`; `PERMS` is a cell array containing `NUMPERMS` permutations; `CMAP` is the input colormap used for the representation. The actual movie is produced and played based on the MATLAB commands of `getframe` (for capturing the movie frames) and `movie` (for replaying the recorded frames).

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

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

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

these comparisons, one might argue that a continuous representation does much better than a categorical one, at least for this particular proximity matrix.

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

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

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

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

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

findultra =

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


vafultra =

    0.7369


conperm =

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

```
vafarob =

    0.9955


arobprox =

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


fitlinear =

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


vaflinear =

    0.9796


coord =

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


addcon =

   -0.2180
```

There is one somewhat unresolved issue as to whether the LUS and ultrametric representations incorporate the same number of "weights,' because otherwise, direct comparison of VAF values may be considered problematic. We argue that, indeed, the number of weights are the same; there are $n-1$ distinct values in an ultrametric matrix and $n-1$ separations along a line in

a LUS model. The one additional additive constant for LUS that is needed to insure invariance to linear transformations of the proximities, should not count against the representation. The ultrametric model automatically has such invariance, and should not be given an inherent advantage just because of this.

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

The fitting strategies offered by `linfitac.m` and `cirfitac.m` (as well as `ultrafit.m` from a companion Cluster Analysis Toolbox – Hubert, Köhn, and Steinley, 2009), all allow unequal spacings to generate the least-squares approximations. This, in effect, requires multiple weights to be constructed. At times, it may be of interest to see how a much simpler model might fair, based only on one 'free weight'. In LUS, we would have equal spacings along a line; for CUS, there would be equal spacings around a circular structure; and for an ultrametric, only multiples of the integer-valued levels (typically, n minus the number of classes in a partition) at which new subsets are formed. In the examples below of `eqspace_linfitac.m`, `eqspace_cirfitac.m`, and `eqspace_ultrafit.m`, the addition of the prefix "`eq`" shows the `vaf`, `fit`, or `addcon` for the equally-spaced alternatives. (All of the latter, we might add, are based on simple regression, rather than on any iterative fitting strategy.) The usual non-equally-spaced alternatives are also given for comparison. We found the high VAF of 83.83% interesting for the equally-spaced LUS model; it is remarkable that only one weight is necessary to generate such a value.

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

fit =

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

```
    0.5060    0.3756    0.3596    0.3369    0.0976    0.0471         0    0.1423    0.1423
    0.6483    0.5179    0.5019    0.4792    0.2399    0.1894    0.1423         0         0
    0.6483    0.5179    0.5019    0.4792    0.2399    0.1894    0.1423         0         0


vaf =

    0.9796


coord =

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


addcon =

   -0.2180


eqfit =

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


eqvaf =

    0.8383
```

```
eqaddcon =

   -0.2181

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

fit =

         0    0.0247    0.3620    0.6413    0.9605    1.1581    1.1581    1.0358    0.7396    0
    0.0247         0    0.3373    0.6165    0.9358    1.1334    1.1334    1.0606    0.7643    0
    0.3620    0.3373         0    0.2793    0.5985    0.7961    0.7961    1.0148    1.1016    0
    0.6413    0.6165    0.2793         0    0.3193    0.5169    0.5169    0.7355    1.0318    1
    0.9605    0.9358    0.5985    0.3193         0    0.1976    0.1976    0.4163    0.7125    1
    1.1581    1.1334    0.7961    0.5169    0.1976         0    0.0000    0.2187    0.5149    0
    1.1581    1.1334    0.7961    0.5169    0.1976    0.0000         0    0.2187    0.5149    0
    1.0358    1.0606    1.0148    0.7355    0.4163    0.2187    0.2187         0    0.2963    0
    0.7396    0.7643    1.1016    1.0318    0.7125    0.5149    0.5149    0.2963         0    0
    0.3883    0.4131    0.7503    1.0296    1.0638    0.8662    0.8662    0.6475    0.3513


vaf =

    0.7190


addcon =

   -0.7964


eqfit =

         0    0.2208    0.4416    0.6624    0.8833    1.1041    0.8833    0.6624    0.4416    0
    0.2208         0    0.2208    0.4416    0.6624    0.8833    1.1041    0.8833    0.6624    0
    0.4416    0.2208         0    0.2208    0.4416    0.6624    0.8833    1.1041    0.8833    0
    0.6624    0.4416    0.2208         0    0.2208    0.4416    0.6624    0.8833    1.1041    0
    0.8833    0.6624    0.4416    0.2208         0    0.2208    0.4416    0.6624    0.8833    1
    1.1041    0.8833    0.6624    0.4416    0.2208         0    0.2208    0.4416    0.6624    0
    0.8833    1.1041    0.8833    0.6624    0.4416    0.2208         0    0.2208    0.4416    0
    0.6624    0.8833    1.1041    0.8833    0.6624    0.4416    0.2208         0    0.2208    0
    0.4416    0.6624    0.8833    1.1041    0.8833    0.6624    0.4416    0.2208         0    0
    0.2208    0.4416    0.6624    0.8833    1.1041    0.8833    0.6624    0.4416    0.2208
```

```
eqvaf =

    0.5518


eqaddcon =

   -0.8371

>> load sc_completelink_integertarget.dat
>> sc_completelink_integertarget

sc_completelink_integertarget =

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

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

fit =

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


vaf =

    0.7369
```

```
eqfit =

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


eqvaf =

    0.5927
```

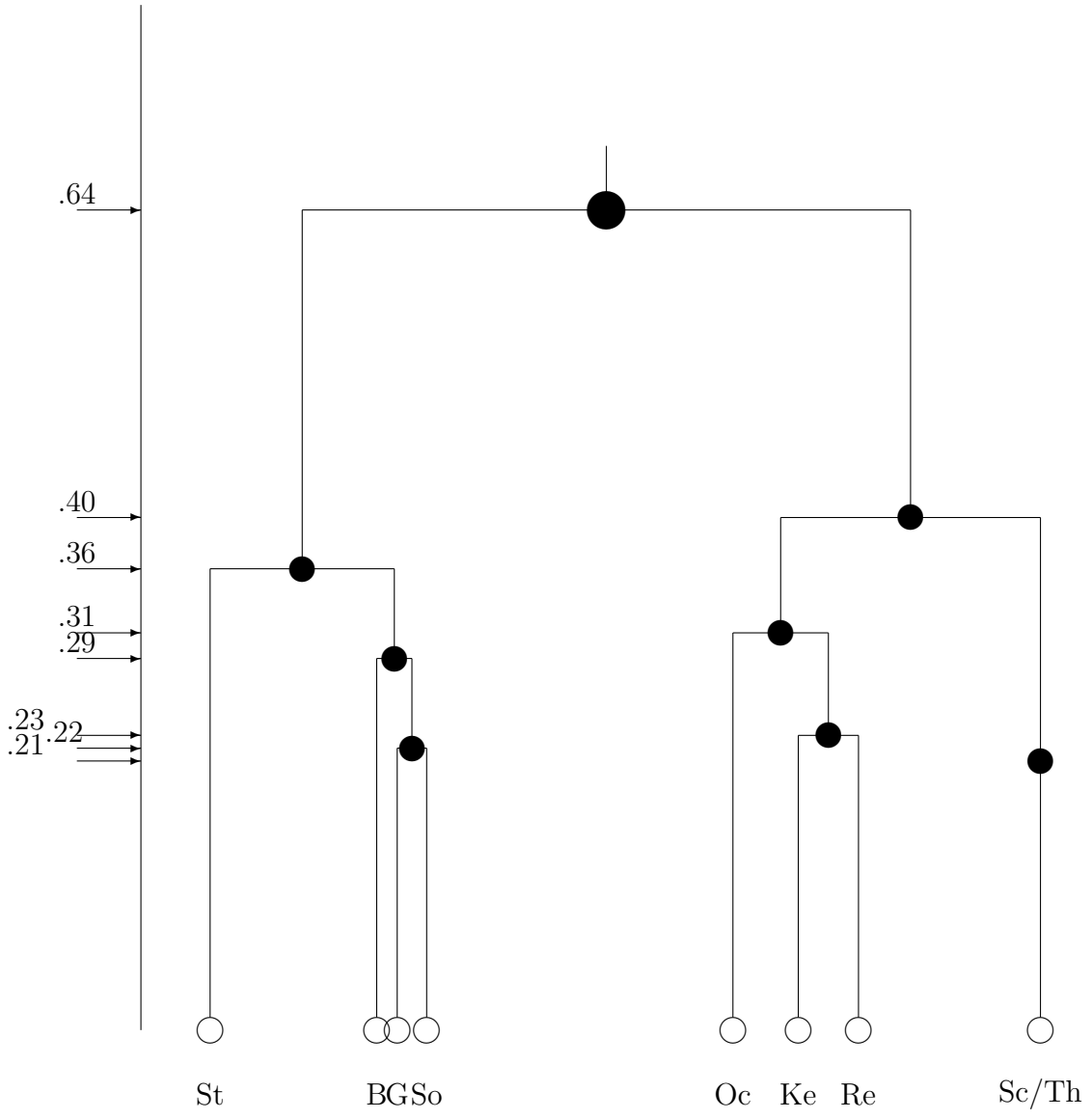
10.2 Representing an Order-Constrained LUS and an Ultrametric on the Same Graph

Whenever the same constraining object order is used to generate both a LUS and ultrametric structure, it is possible to represent them jointly within the same graphical display. The usual dendrogram showing when the new groups form in the hierarchical clustering is given for the ultrametric, but the latter also has its terminal nodes separated according to the coordinates constructed for the LUS. An example is given in Figure 9 using the `supreme_agree` data and the earlier analyses given with `cat_vs_con_orderfnd`. We believe this provides a very nice combined representation for both the discrete and continuous models found for a proximity matrix.

11 The Representation of Proximity Matrices by Structures Dependent on Order (Only)

This section concentrates on an alternative approach to understanding what a given proximity matrix may be depicting about the objects on which it was constructed, and one that does not require a prior commitment to the sole use of either some form of dimensional model (as in nonmetric multidimensional

Figure 9: A Joint Order-Constrained LUS and Ultrametric Representation for the supreme_agree Proximity Matrix



scaling (NMDS)), or one that is strictly classificatory (as in the use of a partition hierarchy and the implicit fitting of an ultrametric that serves as the representational mechanism for the hierarchical clustering). The method of analysis is based on approximating a given proximity matrix additively by a sum of matrices, where each component in the sum is subject to specific patterning restrictions on its entries. The restrictions imposed on each component of the decomposition (to be referred to as matrices with AR, SAR, CAR, or CSAR forms) are very general and encompass interpretations that might be dimensional, or classificatory, or some combination of both (e.g., through object classes that are themselves placed dimensionally in some space). Thus, as one special case — and particularly when an (optimal) transformation of the proximities is also permitted (as we will generally allow), proximity matrices that are well interpretable through NMDS should also be interpretable through an additive decomposition of the (transformed) proximity matrix. Alternatively, when classificatory structures of various kinds might underlie a set of proximities (and the direct use of NMDS could possibly lead to a degeneracy), additive decompositions may still provide an analysis strategy for elucidating the structure.

11.1 Anti-Robinson (AR) Matrices for Symmetric Proximity Data

Denoting an arbitrary symmetric $n \times n$ matrix by $\mathbf{A} = \{a_{ij}\}$, where the main diagonal entries are considered irrelevant and assumed to be zero (i.e., $a_{ii} = 0$ for $1 \leq i \leq n$), $\mathbf{A}$ is said to have an anti-Robinson (AR) form if after some reordering of the rows and columns of $\mathbf{A}$, the entries within each row and column have a distinctive pattern: moving away from the zero main diagonal entry within any row or any column, the entries never decrease. Generally, matrices having AR forms can appear both in spatial representations for a set of proximities as functions of the absolute differences in coordinate values along some axis, or for classificatory structures that are characterized through an ultrametric.

To illustrate, we first let $\mathbf{P} = \{p_{ij}\}$ be a given $n \times n$ proximity (dissimilarity) matrix among the distinct pairs of n objects in a set $S = \{O_1, O_2, \ldots, O_n\}$ (where $p_{ii} = 0$ for $1 \leq i \leq n$). Then, suppose, for example, a two-dimensional

Euclidean representation is possible for $\mathbf{P}$ and its entries are very well representable by the distances in this space, so

$$p_{ij} \approx \sqrt{(x_{1i} - x_{1j})^2 + (x_{2i} - x_{2j})^2} \ ,$$

where x_{ki} and x_{kj} are the coordinates on the k^{th} axis (for $k = 1$ and 2) for objects O_i and O_j (and the symbol $\approx$ is used to indicate approximation). Here, a simple monotonic transformation (squaring) of the proximities should then be fitted well by the sum of two matrices both having AR forms, i.e.,

$$\{p_{ij}^2\} \approx \{(x_{1i} - x_{1j})^2\} + \{(x_{2i} - x_{2j})^2\}.$$

In a classificatory framework, if $\{p_{ij}\}$ were well representable, say, as a sum of two matrices, $\mathbf{A}_1 = \{a_{ij}^{(1)}\}$ and $\mathbf{A}_2 = \{a_{ij}^{(2)}\}$, each satisfying the ultrametric inequality, i.e., $a_{ij}^{(k)} \leq \max\{a_{ih}^{(k)}, a_{hj}^{(k)}\}$ for $k = 1$ and 2, then

$$\{p_{ij}\} \approx \{a_{ij}^{(1)}\} + \{a_{ij}^{(2)}\},$$

and each of the constituent matrices can be reordered to display an AR form. As shown in the Cluster Analysis Toolbox, any matrix whose entries satisfy the ultrametric inequality can be represented by a sequence of partitions that are hierarchically related.

Given some proximity matrix $\mathbf{P}$, the task of approximating it as a sum of matrices each having an AR form is implemented through an iterative optimization strategy based on a least-squares loss criterion that is discussed in detail by Hubert and Arabie (1994). Given the manner in which the optimization process is carried out sequentially, *each successive AR matrix* in any decomposition generally accounts for less and less of the patterning of the original proximity information (and very analogous to what is typically observed in a principal component decomposition of a covariance matrix). In fact, it has been found empirically that for the many data sets we have analyzed, only a very small number of such AR matrices are ever necessary to represent almost all of the patterning in the given proximities. As a succinct summary that we could give to this empirical experience: no more than three AR matrices are ever necessary; two are usually sufficient; and sometimes one will suffice.

The substantive challenge that remains, once a well-fitting decomposition is found for a given proximity matrix, is to interpret substantively what each term in the decomposition might be depicting. The strategy that could be followed would approximate each separate AR matrix by ones having a more restrictive form, and usually those representing some type of unidimensional scale (a dimensional interpretation) or partition hierarchy (a classificatory interpretation).

11.1.1 Interpreting the Structure of an AR matrix

In representing a proximity matrix $\mathbf{P}$ as a sum, $\mathbf{A}_1 + \cdots + \mathbf{A}_K$, the interpretive task remains to explain substantively what each term of the decomposition might be depicting. We suggest four possible strategies below, with the first two attempting to understand the structure of an AR matrix directly and without much loss of detail; the last two require the imposition of strictly parameterized approximations in the form of either an ultrametric or a unidimensional scale. In the discussion below, $\mathbf{A} = \{a_{ij}\}$ will be assumed to have an AR form that is displayed by the given row and column order.

(A) Complete representation and reconstruction through a collection of subsets and associated subset diameters:

The entries in any AR matrix $\mathbf{A}$ can be reconstructed exactly through a collection of M subsets of the original object set $S = \{O_1, \ldots, O_n\}$, denoted by $S_1, \ldots, S_M$, and where M is determined by the particular pattern of tied entries, if any, in $\mathbf{A}$. These M subsets have the following characteristics:

(i) each S_m, $1 \leq m \leq M$, consists of a sequence of (two or more) consecutive integers so that $M \leq n(n-1)/2$. (This bound holds because the number of different subsets having consecutive integers for any given fixed ordering is $n(n-1)/2$, and will be achieved if all the entries in the AR matrix $\mathbf{A}$ are distinct).

(ii) each S_m, $1 \leq m \leq M$, has a diameter, denoted by $d(S_m)$, so that for all object pairs within S_m, the corresponding entries in $\mathbf{A}$ are less than or equal to the diameter. The subsets, $S_1, \ldots, S_M$, can be assumed ordered as $d(S_1) \leq d(S_2) \leq \cdots \leq d(S_M)$, and if $S_m \subseteq S_{m'}$, $d(S_m) \leq d(S_{m'})$.

(iii) each entry in $\mathbf{A}$ can be reconstructed from $d(S_1), \ldots, d(S_M)$, i.e., for $1 \leq i, j \leq n$,

$$a_{ij} = \min_{1 \leq m \leq M} \{d(S_m) \mid O_i, O_j \in S_m\},$$

so that the minimum diameter for subsets containing an object pair $O_i, O_j \in S$ is equal to a_{ij}. Given $\mathbf{A}$, the collection of subsets $S_1, \ldots, S_M$ and their diameters can be identified by inspection through the use of an increasing threshold that starts from the smallest entry in $\mathbf{A}$, and observing which subsets containing contiguous objects emerge from this process. The substantive interpretation of what $\mathbf{A}$ is depicting reduces to explaining why those subsets with the smallest diameters are so homogenous. For convenience of reference, the subsets $S_1, \ldots, S_M$ could be referred to as the set of *AR reconstructive subsets*.

(B) Representation by a strongly anti-Robinson matrix:

If the matrix $\mathbf{A}$ has a somewhat more restrictive form than just being AR, and is also *strongly* anti-Robinson (SAR), a convenient graphical representation can be given to the collection of AR reconstructive subsets $S_1, \ldots, S_M$ and their diameters, and how they can serve to retrieve $\mathbf{A}$. Specifically, $\mathbf{A}$ is said to be strongly anti-Robinson (SAR) if (considering the above-diagonal entries of $\mathbf{A}$) whenever two entries in adjacent columns are equal ($a_{ij} = a_{i(j+1)}$), those in the same two adjacent columns in the previous row are also equal ($a_{(i-1)j} = a_{(i-1)(j+1)}$ for $1 \leq i - 1 < j \leq n - 1$); also, whenever two entries in adjacent rows are equal ($a_{ij} = a_{(i+1)j}$), those in the same two adjacent rows in the succeeding column are also equal ($a_{i(j+1)} = a_{(i+1)(j+1)}$ for $2 \leq i + 1 < j \leq n - 1$).

When $\mathbf{A}$ is SAR, the collection of subsets, $S_1, \ldots, S_M$, and their diameters, and how these serve to reconstruct $\mathbf{A}$ can be modeled graphically. The internal nodes (represented by solid circles) in each of these figures are at a height equal to the diameter of the respective subset; the consecutive objects forming that subset are identifiable by downward paths from the internal nodes to the terminal nodes corresponding to the objects in $S = \{O_1, \ldots, O_n\}$ (represented by labeled open circles). An entry a_{ij} in $\mathbf{A}$ can be reconstructed as the minimum node height of a subset for which a path can be constructed from O_i up to that internal node and then back down to O_j. (To prevent un-

due graphical "clutter", only the most homogenous subsets from $S_1, \ldots, S_M$ having the smallest diameters should actually be included in the graphical representation of an SAR matrix; each figure would explicitly show only how the smallest entries in $\mathbf{A}$ can be reconstructed, although each could be easily extended to include all of $\mathbf{A}$. The calibrated vertical axis in such figures could routinely include the heights at which the additional internal nodes would have to be placed to effect such a complete reconstruction.)

Given an arbitrary AR matrix $\mathbf{A}$, a least-squares SAR approximating matrix to $\mathbf{A}$ can be found using the heuristic optimization search strategy developed in Hubert, Arabie, and Meulman (1998), and illustrated in a section to follow. This latter source also discusses in detail (through counterexample) why strongly AR conditions need to be imposed to obtain a consistent graphical representation.

(C) Representation by a unidimensional scale:

To obtain greater graphical simplicity for an eventual substantive interpretation than offered by an SAR matrix, one possibility is to use approximating unidimensional scales. To be explicit, one very simple form that an AR matrix $\mathbf{A}$ may assume is interpretable by a single dimension and through a unidimensional scale in which the entries have the parameterized form, $\mathbf{A} = \{a_{ij}\} = \{\mid x_j - x_i \mid + c\}$, where the coordinates are ordered as $x_1 \leq x_2 \leq \cdots \leq x_n$ and c is an estimated constant. Given any proximity matrix, a least-squares approximating unidimensional scale can be obtained through the optimization strategies discussed in earlier chapters, and would be one (dimensional) method that could be followed in attempting to interpret what a particular AR component of a decomposition might be revealing.

(D) Representation by an ultrametric:

A second simple form that an AR matrix $\mathbf{A}$ could have is strictly classificatory in which the entries in $\mathbf{A}$ satisfy the ultrametric condition: $a_{ij} \leq \max\{a_{ik}, a_{jk}\}$ for all $O_i, O_j, O_k \in S$. As a threshold is increased from the smallest entry in $\mathbf{A}$, a sequence of partitions of S is identified in which each partition is constructed from the previous one by uniting pairs of subsets from the latter. A partition identified at a given threshold level has equal values in $\mathbf{A}$ between each given pair of subsets, and all the within subset

values are not greater than the between subset values. The reconstructive subsets $S_1, \ldots, S_M$ that would represent the AR matrix $\mathbf{A}$ are now the new subsets that are formed in the sequence of partitions, and have the property that if $d(S_m) \leq d(S_{m'})$, then $S_m \subseteq S_{m'}$ or $S_m \cap S_{m'} = \oslash$. Given any proximity matrix, a least-squares approximating ultrametric can be constructed by the heuristic optimization routines developed in the Cluster Analysis Toolbox, and would be another (classificatory) strategy for interpreting what a particular AR component of a decomposition might be depicting. As might be noted, there are generally $n-1$ subsets (each of size greater than one) in the collection of reconstructive subsets for any ultrametric, and thus $n-1$ values need to be estimated in finding the least-squares approximation (which is the same number needed for a least-squares approximating unidimensional scale, based on obtaining the $n-1$ non-negative separation values between x_i and x_{i+1} for $1 \leq i \leq n-1$).

11.2 Fitting a Given AR Matrix in the L_2-Norm

The function M-file, `arobfit.m`, fits an anti-Robinson matrix using iterative projection to a symmetric proximity matrix in the L_2-norm. The usage syntax is of the form

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

where `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`. A recording of a MATLAB session using the `supreme_agree` data file and object ordering given by the identity permutation follows:

```
>> load supreme_agree.dat
>> inperm = 1:9;
>> [fit,vaf] = arobfit(supreme_agree,inperm)

fit =

         0    0.3600    0.3600    0.3700    0.6550    0.6550    0.7500    0.8550    0.8550
```

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

vaf =

    0.9955
```

11.3 Finding an AR Matrix in the L_2-Norm

The *fitting* of a given AR matrix by the M-function, `arobfit.m`, requires the presence of an initial permutation to direct the optimization process. Thus, the *finding* of a best-fitting AR matrix reduces to the identification of an appropriate object permutation to use *ab initio*. We suggest the adoption of `order.m`, which carries out an iterative Quadratic Assignment (QA) maximization task using a given square, $n \times n$, proximity matrix `PROX` (with a zero main diagonal and a dissimilarity interpretation). 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`: (a) pairwise interchanges of objects in the permutation defining the row and column order of the square proximity matrix; (b) the insertion of from 1 to `KBLOCK` (which is less than or equal to $n-1$) consecutive objects in the permutation defining the row and column order of the data matrix; and (c) the rotation of from 2 to `KBLOCK` (which is less than or equal to $n-1$) consecutive objects in the permutation defining the row and column order of the data matrix. The usage syntax has the form

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

where `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`. The cell array `ALLPERMS` contains `INDEX`

entries corresponding to all the permutations identified in the optimization from ALLPERMS{1} = INPERM to ALLPERMS{INDEX} = OUTPERM.

A recording of a MATLAB session using order.m is listed below with the beginning INPERM given as the identity permutation, TARG by an equally-spaced object placement along a line, and KBLOCK = 3. Using the generated OUTPERM, arobfit.m is then invoked to fit an AR form having final VAF of .9955.

```
>> load supreme_agree.dat
>> [outperm,rawindex,allperms,index] = order(supreme_agree, targlin(9),randperm(9),3);

outperm =

     9     8     7     6     5     4     3     2     1

>> [fit,vaf] = arobfit(supreme_agree,outperm)

fit =

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


vaf =

    0.9955
```

The M-file, arobfnd.m is our preferred method for actually identifying a single AR form, and incorporates an initial equally-spaced target and uses the iterative QA routine of order.m to generate better permutations; the obtained AR forms are then used as new targets against which possibly even better permutations might be identified, until convergence (i.e., the identified permutations remain the same). The syntax is as follows:

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

where 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 the iterative quadratic assignment routine.

As seen from the example below, and starting from a random initial permutation, the same AR form is found as with just one application of order.m reported above.

```
>> [find,vaf,outperm] = arobfnd(supreme_agree, randperm(9),2)

find =

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


vaf =

    0.9955


outperm =

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

11.4 Fitting and Finding a Strongly Anti-Robinson (SAR) Matrix in the L_2-Norm

The M-functions, sarobfit.m and sarobfnd.m, are direct analogues of arobfit.m and arobfnd.m, respectively, but are concerned with fitting and finding *strongly* anti-Robinson forms. The syntax for sarobfit.m, which fits a

strongly anti-Robinson matrix using iterative projection to a symmetric proximity matrix in the L_2-norm, is

```
[fit, vaf] = sarobfit(prox, inperm)
```

where, again, `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 a strongly anti-Robinson form for the row and column object ordering given by `INPERM`.

An example follows using the same identity permutation as was implemented in fitting an AR form with `arobfit.m`; as might be expected from using the more restrictive strongly anti-Robinson form, the `VAF` drops to .9862 from .9955.

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

fit =

         0    0.3600    0.3600    0.3700    0.6550    0.7025    0.7025    0.8550    0.8550
    0.3600         0    0.2800    0.2900    0.4900    0.5450    0.5450    0.7425    0.7425
    0.3600    0.2800         0    0.2200    0.4900    0.5450    0.5450    0.7425    0.7425
    0.3700    0.2900    0.2200         0    0.4500    0.5300    0.5300    0.7000    0.7000
    0.6550    0.4900    0.4900    0.4500         0    0.3100    0.3100    0.4600    0.4600
    0.7025    0.5450    0.5450    0.5300    0.3100         0    0.2300    0.4150    0.4150
    0.7025    0.5450    0.5450    0.5300    0.3100    0.2300         0    0.3300    0.3300
    0.8550    0.7425    0.7425    0.7000    0.4600    0.4150    0.3300         0    0.2100
    0.8550    0.7425    0.7425    0.7000    0.4600    0.4150    0.3300    0.2100         0


vaf =

    0.9862
```

The M-function, `sarobfnd.m`, finds and fits a strongly 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. The function has the expected syntax

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

where, again, 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 a strongly anti-Robinson form for the row and column object ordering given by the ending permutation OUTPERM. As usual, KBLOCK defines the block size in the use the iterative quadratic assignment routine.

In the MATLAB recording below, and starting from a random permutation, the same strongly anti-Robinson form is found with a VAF of .9862.

```
>> [find,vaf,outperm] = sarobfnd(supreme_agree,randperm(9),2)

find =

         0    0.3600    0.3600    0.3700    0.6550    0.7025    0.7025    0.8550    0.8550
    0.3600         0    0.2800    0.2900    0.4900    0.5450    0.5450    0.7425    0.7425
    0.3600    0.2800         0    0.2200    0.4900    0.5450    0.5450    0.7425    0.7425
    0.3700    0.2900    0.2200         0    0.4500    0.5300    0.5300    0.7000    0.7000
    0.6550    0.4900    0.4900    0.4500         0    0.3100    0.3100    0.4600    0.4600
    0.7025    0.5450    0.5450    0.5300    0.3100         0    0.2300    0.4150    0.4150
    0.7025    0.5450    0.5450    0.5300    0.3100    0.2300         0    0.3300    0.3300
    0.8550    0.7425    0.7425    0.7000    0.4600    0.4150    0.3300         0    0.2100
    0.8550    0.7425    0.7425    0.7000    0.4600    0.4150    0.3300    0.2100         0


vaf =

    0.9862


outperm =

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

11.5 Representation Through Multiple (Strongly) AR Matrices

The representation of a proximity matrix by a single anti-Robinson structure extends easily to the additive use of multiple matrices. The M-function, biarobfnd.m, fits the sum of two anti-Robinson matrices using iterative projection to a symmetric proximity matrix in the L_2-norm based on permuta-

tions identified through the use of iterative quadratic assignment. The usage syntax is

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

where, as before, `PROX` is the input proximity matrix ($n \times n$ with a zero main diagonal and a dissimilarity interpretation); `INPERM` is a given starting permutation of the first n integers; `FIND` is the least-squares optimal matrix (with variance-accounted-for of `VAF`) to `PROX` and is the sum of the two anti-Robinson matrices `TARGONE` and `TARGTWO` based on the two row and column object orderings given by the ending permutations `OUTPERMONE` and `OUTPERMTWO`. As before, `KBLOCK` defines the block size in the use of the iterative quadratic assignment routine.

For finding multiple SAR forms, `bisarobfnd.m` has usage syntax

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

with all the various terms the same as for `biarobfnd.m` but now for strongly AR (SAR) structures.

11.6 l_p Fitted Distance Metrics Based on Given Object Orders

The emphasis in this Toolbox has been on obtaining object orders along a continuum, and secondarily, in generating explicit coordinates for a city-block representation. The object orders are primary with the coordinates coming along more or less automatically. To generalize, it is possible to fit to the given proximities, a Minkowski distance function of the form:

$$\sum_{i<j} (|x_i - x_j|^p + |y_i - y_j|^p)^{\frac{1}{p}} \ ,$$

where $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ are coordinates along a first and second axes, respectively; both sets of coordinates are constrained to represent the given object orders that are given as input.

The M-function, `lpfit.m`, provides the two-dimensional coordinates to construct l_p distances fit (in a confirmatory manner) to the given proximities. The syntax is as follows:

```
[fitted, coordone,coordtwo,addcon,fval,vaf,exitflag] = ...
    lpfit(prox,permone,permtwo,start_vector,p)
```

As input, there is the $n \times n$ symmetric dissimilarity matrix `PROX` with zero main diagonal entries; two fixed permutations, `PERMONE` and `PERMTWO`, of the objects along the two dimensions; the value of p to obtain the l_p distances ($p = 1$: city-block distances; $p = 2$: Euclidean distances; large p: dominance distances). As outputs, the values fitted to the proximity matrix are in `FITTED`; the object coordinates along the two dimensions, `COORDONE` and `COORDTWO`; the additive constant, `ADDCON`; the value of the least-squares criterion, `FVAL`, obtained in finding the coordinates; the variance-accounted-for measure, `VAF`; the exit condition, `EXITFLAG`, from using the MATLAB optimization routine, `fmincon.m`.

To give an example, the reader is advised to run the script file, `script_lpfit.m`. It fits a Euclidean distance function for the `morse_digit` data based on the city-block solution obtained with `biscalqa.m`. It also shows how a start vector may be generated rationally using the given input orders even though an all-zero start vector is then used in calling `lpfit.m`. Note that in `lpfit.m`, an option file is defined:

```
options = optimset('Algorithm','interior-point')
```

which calls an interior-point solver in `fmincon.m`. Other optimization options would be an active-set method (default), `'active-set'`, or a sequential quadratic programming method, `'sqp'`.

12 Circular-Anti-Robinson (CAR) Matrices for Symmetric Proximity Data

In the approximation of a proximity matrix $\mathbf{P}$ by one that is row/column reorderable to an AR form, the interpretation of the fitted matrix in general had to be carried out by identifying a set of subsets through an increasing threshold variable; each of the subsets contained objects that were contiguous with respect to a given *linear* ordering along a continuum, and had a diameter defined by the maximum fitted value within the subset. To provide

a further representation depicting the fitted values as lengths of paths in a graph, an approximation was sought that satisfied the additional constraints of an SAR matrix; still, the subsets thus identified had to contain objects contiguous with respect to a linear ordering. As one possible generalization of both the AR and SAR constraints, we can define what will be called circular anti-Robinson (CAR) and circular strongly-anti-Robinson (CSAR) forms that allow the subsets identified from increasing a threshold variable to be contiguous with respect to a *circular* ordering of the objects around a closed continuum. Approximation matrices that are row/column reorderable to display an AR or SAR form, respectively, will also be (trivially) row/column reorderable to display what is formally characterized below as a CAR or a CSAR form, but not conversely. (Historically, there is a large literature on the possibility of circular structures emerging from and being identifiable in a given proximity matrix. For a variety of references, the reader is referred to the American Psychological Association sponsored volume edited by Plutchik and Conte (1997). The extension of CAR forms to those that are also CSAR, however, has apparently not been a topic discussed in the literature before the appearance of Hubert, Arabie, and Meulman [1998]; this latter source forms the basis for much of the present section.)

To be explicit, an arbitrary symmetric matrix $\mathbf{Q} = \{q_{ij}\}$, where $q_{ii} = 0$ for $1 \leq i, j \leq n$, is said to be row/column reorderable to a circular anti-Robinson form (or, for short, $\mathbf{Q}$ is a circular anti-Robinson (CAR) matrix) if there exists a permutation, $\rho(\cdot)$, on the first n integers such that the reordered matrix $\mathbf{Q}_\rho = \{q_{\rho(i)\rho(j)}\}$ satisfies the conditions given in (II):

(II): for $1 \leq i \leq n-3$, and $i+1 < j \leq n-1$,

if $q_{\rho(i+1)\rho(j)} \leq q_{\rho(i)\rho(j+1)}$, then
$q_{\rho(i+1)\rho(j)} \leq q_{\rho(i)\rho(j)}$ and $q_{\rho(i+1)\rho(j)} \leq q_{\rho(i+1)\rho(j+1)}$;
if $q_{\rho(i+1)\rho(j)} \geq q_{\rho(i)\rho(j+1)}$, then
$q_{\rho(i)\rho(j)} \geq q_{\rho(i)\rho(j+1)}$ and $q_{\rho(i+1)\rho(j+1)} \geq q_{\rho(i)\rho(j+1)}$,
and, for $2 \leq i \leq n-2$,
if $q_{\rho(i+1)\rho(n)} \leq q_{\rho(i)\rho(1)}$, then
$q_{\rho(i+1)\rho(n)} \leq q_{\rho(i)\rho(n)}$ and $q_{\rho(i+1)\rho(n)} \leq q_{\rho(i+1)\rho(1)}$;
if $q_{\rho(i+1)\rho(n)} \geq q_{\rho(i)\rho(1)}$, then

$q_{\rho(i)\rho(n)} \geq q_{\rho(i)\rho(1)}$ and $q_{\rho(i+1)\rho(1)} \geq q_{\rho(i)\rho(1)}$.

Interpretatively, within each row of $\mathbf{Q}_\rho$ moving to the right from the main diagonal and then wrapping back around to re-enter the same row from the left, the entries never decrease until a maximum is reached and then never increase moving away from the maximum until the main diagonal is again reached. Given the symmetry of $\mathbf{P}$, a similar pattern of entries would be present within each column as well. As noted above, any AR matrix is CAR but not conversely.

In analogy to the SAR conditions that permit graphical representation, a symmetric matrix $\mathbf{Q}$ is said to be row/column reorderable to a circular strongly-anti-Robinson form (or, for short, $\mathbf{Q}$ is a *circular strongly-anti-Robinson* (CSAR) matrix) if there exists a permutation, $\rho(\cdot)$, on the first n integers such that the reordered matrix $\mathbf{Q}_\rho = \{q_{\rho(i)\rho(j)}\}$ satisfies the conditions given by (II), *and*

for $1 \leq i \leq n-3$, and $i+1 < j \leq n-1$,

if $q_{\rho(i+1)\rho(j)} \leq q_{\rho(i)\rho(j+1)}$,

then $q_{\rho(i+1)\rho(j)} = q_{\rho(i)\rho(j)}$ implies $q_{\rho(i+1)\rho(j+1)} = q_{\rho(i)\rho(j+1)}$, and $q_{\rho(i+1)\rho(j)} = q_{\rho(i+1)\rho(j+1)}$ implies $q_{\rho(i)\rho(j)} = q_{\rho(i)\rho(j+1)}$;

if $q_{\rho(i+1)\rho(j)} \geq q_{\rho(i)\rho(j+1)}$,

then $q_{\rho(i)\rho(j+1)} = q_{\rho(i+1)\rho(j+1)}$ implies $q_{\rho(i)\rho(j)} = q_{\rho(i+1)\rho(j)}$, and $q_{\rho(i)\rho(j)} = q_{\rho(i)\rho(j+1)}$ implies $q_{\rho(i+1)\rho(j)} = q_{\rho(i+1)\rho(j+1)}$,

and for $2 \leq i \leq n-2$,

if $q_{\rho(i+1)\rho(n)} \leq q_{\rho(i)\rho(1)}$, then $q_{\rho(i+1)\rho(n)} = q_{\rho(i)\rho(n)}$ implies $q_{\rho(i+1)\rho(1)} = q_{\rho(i)\rho(1)}$, and $q_{\rho(i+1)\rho(n)} = q_{\rho(i+1)\rho(1)}$ implies $q_{\rho(i)\rho(n)} = q_{\rho(i)\rho(1)}$;

if $q_{\rho(i+1)\rho(n)} \geq q_{\rho(i)\rho(1)}$, then $q_{\rho(i)\rho(1)} = q_{\rho(i+1)\rho(1)}$ implies $q_{\rho(i)\rho(n)} = q_{\rho(i+1)\rho(n)}$, and $q_{\rho(i)\rho(n)} = q_{\rho(i)\rho(1)}$ implies $q_{\rho(i+1)\rho(n)} = q_{\rho(i+1)\rho(1)}$.

Again, the imposition of the stronger CSAR conditions avoids graphical anomalies, i.e., when two fitted values that are adjacent within a row are equal, the fitted values in the same two adjacent columns must also be equal for a row that is either its immediate predecessor (if $q_{\rho(i+1)\rho(j)} \leq q_{\rho(i)\rho(j+1)}$), or successor (if $q_{\rho(i+1)\rho(j)} \geq q_{\rho(i)\rho(j+1)}$); a similar condition is imposed when

two fitted values that are adjacent within a column are equal. As noted, any SAR matrix is CSAR but not conversely.

The computational strategy we suggest for identifying a best-fitting CAR or CSAR approximation matrix is based on an initial circular unidimensional scaling obtained through the optimization strategy developed by Hubert, Arabie, and Meulman (1997). Specifically, we first institute a combination of combinatorial search for good matrix reorderings and heuristic iterative projection to locate the points of inflection when minimum distance calculations change directionality around a closed circular structure. Approximation matrices to $\mathbf{P}$ are found through a least-squares loss criterion, and they have the parameterized form

$$\mathbf{Q}_\rho = \{\min(\mid x_{\rho(j)} - x_{\rho(i)} \mid,\ x_0 - \mid x_{\rho(j)} - x_{\rho(i)} \mid)\ + c\},$$

where c is an estimated additive constant, $x_{\rho(1)} \leq x_{\rho(2)} \leq \cdots \leq x_{\rho(n)} \leq x_0$, and the last coordinate, x_0, is the circumference of the circular structure. Based on the inequality constraints implied by such a collection of coordinates, a CAR approximation matrix can be fitted to $\mathbf{P}$ directly; then, beginning with this latter CAR approximation, the identification and imposition of CSAR constraints proceeds through the heuristic use of iterative projection, directly analogous to the way SAR constraints in the linear ordering context were identified and fitted, beginning with a best approximation matrix satisfying just the AR restrictions.

12.1 Fitting a Given CAR Matrix in the L_2-Norm

The function M-file, `cirarobfit.m`, fits a circular anti-Robinson (CAR) matrix using iterative projection to a symmetric proximity matrix in the L_2-norm. Usage syntax is

```
[fit, vaf] = cirarobfit(prox,inperm,targ)
```

where `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 (around a circle); `TARG` is a given $n \times n$ matrix having the circular anti-Robinson form that guides the direction in which distances are taken around

the circle. The matrix FIT is the least-squares optimal approximation (with variance-accounted-for of VAF) to PROX having an circular anti-Robinson form for the row and column object ordering given by INPERM.

12.2 Finding a CAR Matrix in the L_2-Norm

The M-file, cirarobfnd.m, is our suggested strategy for identifying a best-fitting CAR matrix for a symmetric proximity matrix in the L_2-norm based on a permutation that is initially identified through the use of iterative quadratic assignment. Based on an equally-spaced circular target matrix, order.m is first invoked to obtain a good (circular) permutation, which in turn is then used to construct a new circular target matrix with cirfit.m. (We will mention here but not illustrate with an example, an alternative to the use of cirarobfnd.m called cirarobfnd_ac.m; the latter M-file has the same syntax as cirarobfnd.m but uses cirfitac.m rather than cirfit.m internally to obtain the new circular target matrices.) The final output is generated from cirarobfit.m. The usage syntax for cirarobfnd.m is as follows:

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

where PROX is the input proximity matrix ($n \times n$ with a zero main diagonal and a dissimilarity interpretation); INPERM is a given starting permutation (assumed to be around the circle) of the first n integers; FIND is the least-squares optimal matrix (with variance-accounted-for of VAF) to PROX having a circular anti-Robinson form for the row and column object ordering given by the concluding permutation OUTPERM. Again, KBLOCK defines the block size in the use of the iterative quadratic assignment routine.

12.3 Representation Through Multiple (Strongly) CAR Matrices

Just as we discussed representing of proximity matrices through multiple (strongly) AR matrices, the analysis of a proximity matrix by a single (strongly) circular-anti-Robinson structure extends easily to the additive use of multiple matrices. The M-function, bicirarobfnd.m, fits the sum of two circular-anti-Robinson matrices using iterative projection to a symmetric proximity matrix

in the L_2-norm based on permutations identified through the use of iterative quadratic assignment. The syntax usage is

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

where, as before, `PROX` is the input proximity matrix ($n \times n$ with a zero main diagonal and a dissimilarity interpretation); `INPERM` is a given initial permutation of the first n integers; `FIND` is the least-squares optimal matrix (with variance-accounted-for of `VAF`) to `PROX` and is the sum of the two circular-anti-Robinson matrices `TARGONE` and `TARGTWO` based on the two row and column object orderings given by the final permutations `OUTPERMONE` and `OUTPERMTWO`. As before, `KBLOCK` defines the block size in the use of the iterative quadratic assignment routine.

For finding multiple CSAR forms, `bicirsarobfnd.m` has usage syntax

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

with all the various terms the same as for `bicirarobfnd.m` but now for strongly CAR (CSAR) structures.

13 Anti-Robinson (AR) Matrices for Two-Mode Proximity Data

In direct analogy to the extensions of Linear Unidimensional Scaling (LUS), it is possible to find and fit (more general) anti-Robinson (AR) forms to two-mode proximity matrices. The same type of reordering strategy implemented by `ordertm.m` would be used, but the more general AR form would be fitted to the reordered square proximity matrix, $\mathbf{P}_{\rho_0}^{(tm)} = \{p_{\rho_0(i)\rho_0(j)}^{(tm)}\}$; the least-squares criterion

$$\sum_{i,j=1}^{n} w_{\rho_0(i)\rho_0(j)}(p_{\rho_0(i)\rho_0(j)}^{(tm)} - \hat{p}_{ij})^2,$$

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 $\{\hat{p}_{ij}\}$ fitted to $\mathbf{P}_{\rho_0}^{(tm)}$ are AR in form (and which correspond to nonzero values of the weight

function $w_{\rho_0(i)\rho_0(j)}$), and thus satisfy certain linear inequality constraints generated from how the row and column objects are intermixed by the given permutation $\rho_0(\cdot)$. We note here and discuss this more completely in the section to follow that the patterning of entries in $\{\hat{p}_{ij}\}$ fitted to the original two-mode proximity matrix, with appropriate row and column permutations extracted from ρ_0, is called an anti-Q-form.

13.1 Fitting and Finding Two-Mode AR Matrices

The M-file `arobfittm.m` does a confirmatory two-mode anti-Robinson fitting of a given ordering of the row and column objects of a two-mode proximity matrix using the Dykstra-Kaczmarz iterative projection least-squares method. The usage syntax has the form

```
[fit,vaf,rowperm,colperm] = arobfittm(proxtm,inperm)
```

where `PROXTM` is the input two-mode proximity matrix; `INPERM` is the given ordering of the row and column objects together; `FIT` is an $n_a \times n_b$ (number of rows by number of columns) matrix fitted to `PROXTM(ROWPERM,COLPERM)` with `VAF` being the variance-accounted-for based on the (least-squares criterion) sum of squared discrepancies between `PROXTM(ROWPERM,COLMEAN)` and `FIT`; `ROWPERM` and `COLPERM` are the row and column object orderings derived from `INPERM`.

The matrix given by `FIT` that is intended to approximate the row and column permuted two-mode proximity matrix, `PROXTM(ROWPERM,COLPERM)`, displays a particularly important patterning of its entries called an anti-Q-form in the literature (see Hubert and Arabie, 1995, for an extended discussion of this type of patterning for a two-mode matrix). Specifically, a matrix is said to have the anti-Q-form (for rows and columns) if within each row and column the entries are nonincreasing to a minimum and thereafter nondecreasing. *Matrices satisfying the anti-Q-form have a convenient interpretation presuming an underlying unidimensional scale that jointly represents both the row and column objects.* Explicitly, suppose a matrix has been appropriately row-ordered to display the anti-Q-form for columns. Any dichotomization of the entries within a column at some threshold value (using

0 for entries below the threshold and 1 if at or above), produces a matrix that has the consecutive zeros property within each column, that is, all zeros within a column occur consecutively, uninterrupted by intervening ones. In turn, any matrix with the consecutive zeros property for columns suggests the existence of a perfect scale (error-free), where row objects can be ordered along a continuum (using the same row order for the matrix that actually reflects the anti-Q-form for columns), and each column object is representable as an interval along the continuum (encompassing those consecutive row objects corresponding to zeros). Historically, the type of pattern represented by the anti-Q-form has played a major role in the literature of (unidimensional) unfolding, and for example, is the basis of a Coombs (1964, Chapter 4) parallelogram structure for a two-mode proximity matrix. The reader is referred to Hubert (1974) for a review of some of these connections.

13.2 Multiple Two-Mode AR Reorderings and Fittings

The M-file, `biarobfndtm.m`, finds and fits the sum of two anti-Q-forms (extracted from fitting two anti-Robinson matrices) using iterative projection to a two-mode proximity matrix in the L_2-norm based on permutations identified through the use of iterative quadratic assignment. In the usage

```
[find,vaf,targone,targtwo,outpermone,outpermtwo, ...
        rowpermone,colpermone,rowpermtwo,colpermtwo] = ...
    biarobfndtm(proxtm,inpermone,inpermtwo,kblock)
```

`PROXTM` is the usual input two-mode proximity matrix ($n_a \times n_b$) with a dissimilarity interpretation, and `FIND` is the least-squares optimal matrix (with variance-accounted-for of `VAF`) to `PROXTM`. The latter matrix `PROXTM` is the sum of the two matrices `TARGONE` and `TARGTWO` based on the two row and column object orderings given by the ending permutations `OUTPERMONE` and `OUTPERMTWO`. The two ending permutations of `OUTPERMONE` and `OUTPERMTWO` contain the ending row and column object orderings of `ROWPERMONE` and `ROWPERMTWO` and `COLPERMONE` and `COLPERMTWO`. `KBLOCK` defines the block size in the use the iterative quadratic assignment routine; the input permutations are `INPERMONE` and `INPERMTWO`.

14 Some Bibliographic Comments

There are a number of book-length presentations of (multi)dimensional scaling methods available (encompassing differing collections of subtopics within the field). We list several of the better ones to consult in the reference section to follow, and note these here in chronological order: Kruskal & Wish (1978); Shiffman, Reynolds, & Young (1981); Everitt & Rabe-Hesketh (1997); Carroll & Arabie (1998); Cox & Cox (2001); Lattin, Carroll, & Green (2003); Hand (2004); Borg & Groenen (2005). The items that would be closest to the approaches taken here with MATLAB and the emphasis on least-squares, would be the monograph by Hubert, Arabie, and Meulman (2006), and the reviews by Hubert, Arabie, & Meulman (1997; 2001; 2002); Hubert, Köhn, & Steinley (2009, 2010); and Steinley & Hubert (2005).

References

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

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

[3] Coombs, C. H. (1964). *A theory of data*. New York: Wiley.

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

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

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

[7] Flueck, J. A., & Korsh, J. F. (1974). A branch search algorithm for maximum likelihood paired comparison ranking. *Biometrika*, *61*, 621–626.

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

[9] Hubert, L. J. (1974). Problems of seriation using a subject by item response matrix. *Psychological Bulletin*, *81*, 976–983.

[10] Hubert, L. J. (1976). Seriation using asymmetric proximity meaures. *British Journal of Mathematical and Statistical Psychology*, *29*, 32–52.

[11] 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.

[12] Hubert, L. J., & Arabie, P. (1995). The approximation of two-mode proximity matrices by sums of order-constrained matrices. *Psychometrika*, *60*, 573–605.

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

[14] Hubert, L. J., Arabie, P., & Meulman, J. J. (1998) Graph-theoretic representations for proximity matrices through strongly-anti-Robinson or circular strongly-anti-Robinson matrices. *Psychometrika*, *63*, 341–358.

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

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

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

[18] Hubert, L., Köhn, H.-F., & Steinley, D. (2009). Cluster Analysis: A Toolbox for MATLAB. In R. Millsap & A. Maydeu-Olivares (Eds.), *Handbook of quantitative methods in psychology* (pp. 444–512). Riverside, CA: Sage.

[19] Hubert, L., Köhn, H.-F., & Steinley, D. (2010). Order-constrained proximity matrix representations: Ultrametric generalizations and constructions with MATLAB. In S. Kolenikov, D. Steinley, & L. Thombs (Eds.), *Current methodological developments of statistics in the social sciences*(pp. 81–112). New York: Wiley.

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

[21] Johnson, E. C., & Tversky, A. (1984). Representations of perceptions of risk. *Journal of Experimental Psychology: General*, *113*, 55–70.

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

[23] Korte, B., & Oberhofer, W. (1971). Triangularizing input-output matrices and the structures of production. *European Economic Review*, *2*, 493–522.

[24] Kruskal, J. B. (1964a). Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, *29*, 1–27.

[25] Kruskal, J. B. (1964b). Nonmetric multidimensional scaling: A numerical method. *Psychometrika*, *29*, 115–129.

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

[27] Lawler, E. L. (1964). A comment on minimum feedback are sets. *IEEE Transactions on Circuit Theory*, *11*, 296–297.

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

[29] Osgood, C. E., & Luria, Z. (1954). A blind analysis of a case of multiple personality. *Journal of Abnormal and Social Psychology*, *49*, 579–591.

[30] Plutchik, R. & Conte, H. R. (Eds.). (1997). *Circumplex models of personality and emotions*. Washington, DC: American Psychological Association.

[31] Ramsay, J. O. (1988). Monotone regression splines in action. *Statistical Science*, *3*, 425–441.

[32] Rothkopf, E. Z. (1957). A measure of stimulus similarity and errors in some paired-associate learning tasks. *Journal of Experimental Psychology*, *53*, 94–101.

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

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

[35] Thurstone, L. L. (1959). *The measurement of values* Chicago: The University of Chicago Press.

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

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

arobfit.m

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

arobfnd.m

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

arobfittm.m

```
%  AROBFITTM does a confirmatory two-mode anti-Robinson fitting of a
%  given ordering of the row and column objects of a two-mode
%  proximity matrix PROXTM using Dykstra's (Kaczmarz's)
```

```
%  iterative projection least-squares method.
%
%  syntax: [fit,vaf,rowperm,colperm] = arobfittm(proxtm,inperm)
%
%  INPERM is the given ordering of the row and column objects
%  together; FIT is an nrow (number of rows) by ncol (number of
%  columns) matrix fitted to PROXTM(ROWPERM,COLPERM)
%  with VAF being the variance-accounted for and
%  based on 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.
```

arobfndtm.m

```
% AROBFNDTM finds and fits an anti-Robinson
% form using iterative projection to
% a two-mode proximity matrix in the $L_{2}$-norm based on a
% permutation identified through the use of iterative quadratic
% assignment.
%
% syntax: [find, vaf, outperm, rowperm, colperm] = ...
% arobfndtm(proxtm, inperm, kblock)
%
% PROXTM is the input two-mode proximity matrix
% ($n_{a} \times n_{b}$ with a dissimilarity interpretation);
% INPERM is a given starting permutation
% of the first $n = n_{a} + n_{b}$ integers;
% FIND is the least-squares optimal matrix (with
% variance-accounted-for of VAF) to PROXTM having the 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. ROWPERM and COLPERM are the resulting
% row and column permutations for the objects.
```

biarobfnd.m

```
% BIAROBFND finds and fits the sum of two
% anti-Robinson matrices using iterative projection to
% a symmetric proximity matrix in the $L_{2}$-norm
% based on permutations identified through
% the use of iterative quadratic assignment.
%
```

```
% syntax: [find,vaf,targone,targtwo,outpermone, ...
%     outpermtwo] = biarobfnd(prox,inperm,kblock)
%
% PROX is the input proximity matrix ($n \times n$ with a zero
% main diagonal and a dissimilarity interpretation);
% INPERM is a given starting permutation of the first $n$ integers;
% FIND is the least-squares optimal matrix (with
% variance-accounted-for of VAF)
% to PROX and is the sum of the two anti-Robinson matrices
% TARGONE and TARGTWO based on the two row and column
% object orderings given by the ending permutations OUTPERMONE
% and OUTPERMTWO. KBLOCK defines the block size in the use of the
% iterative quadratic assignment routine.
```

bisarobfnd.m

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

biarobfndtm.m

```
% BIAROBFNDTM finds and fits the sum of
% two anti-Robinson matrices using iterative projection to
% a two-mode proximity matrix in the $L_{2}$-norm based on
% permutations identified through the use of
% iterative quadratic assignment.
%
```

```
% syntax: [find,vaf,targone,targtwo,outpermone,outpermtwo, ...
%        rowpermone,colpermone,rowpermtwo,colpermtwo] = ...
%   biarobfndtm(proxtm,inpermone,inpermtwo,kblock)
%
% PROXTM is the input two-mode proximity matrix ($nrow \times ncol$)
% with a dissimilarity interpretation);
% FIND is the least-squares optimal matrix (with variance-
% accounted-for of VAF) to PROXTM and is the sum of the two matrices
% TARGONE and TARGTWO based on the two row and column
% object orderings given by the ending permutations OUTPERMONE
% and OUTPERMTWO, and in turn ROWPERMONE and ROWPERMTWO and
% COLPERMONE and COLPERMTWO. KBLOCK defines the block size
% in the use of the iterative quadratic assignment routine;
% the input permutations are INPERMONE and INPERMTWO.
```

bicirarobfnd.m

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

bicirsarobfnd.m

```
% BICIRSAROBFND fits the sum of two strongly circular anti-Robinson
% matrices using iterative projection to a symmetric proximity
% matrix in the $L_{2}$-norm based on permutations
% identified through the use of iterative quadratic assignment.
%
% syntax: [find,vaf,targone,targtwo,outpermone,outpermtwo] = ...
```

```
%      bicirsarobfnd(prox,inperm,kblock)
%
% PROX is the input proximity matrix ($n \times n$ with a zero main
% diagonal and a dissimilarity interpretation);
% INPERM is a given starting permutation of the first $n$ integers;
% FIND is the least-squares optimal matrix (with variance-
% accounted-for of VAF) to PROX and is the
% sum of the two strongly circular anti-Robinson matrices;
% TARGONE and TARGTWO are based on the two row and column
% object orderings given by the ending permutations OUTPERMONE
% and OUTPERMTWO. KBLOCK defines the block size in the use of the
% iterative quadratic assignment routine.
```

bicirac.m

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

biscallp.m

```
%BISCALLP carries out a bidimensional scaling of a symmetric proximity
%  matrix using iterative linear programming.
%  PROX is the input proximity matrix (with a zero main diagonal and a
%  dissimilarity interpretation);
%  INPERMONE is the input beginning permutation for the first dimension
```

```
%  (a permuation of the first $n$ integers); INPERMTWO is the input beginning
%  permutation for the second dimension;
%  NOPT controls the confirmatory or exploratory fitting of the unidimensional
%  scales; a value of NOPT = 0 will fit in a confirmatory manner the two scales
%  indicated by INPERMONE and INPERMTWO; a value of NOPT = 1 uses iterative LP
%  to locate the better permutations to fit;
%  OUTPERMONE is the final object permutation for the first dimension;
%  OUTPERMTWO is the final object permutation for the second dimension;
%  COORDONE is the set of first dimension coordinates in ascending order;
%  COORDTWO is the set of second dimension coordinates in ascending order;
%  ADDCONONE is the additive constant for the first dimensional model;
%  ADDCONTWO is the additive constant for the second dimensional model;
%  DEV is the variance-accounted-for in PROX by the bidimensional scaling.
```

biscalqa.m

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

```
%  first dimension; OUTPERMTWO is the final object permutation
%  for the second dimension;
%  COORDONE is the set of first dimension coordinates
%  in ascending order; COORDTWO is the set of second dimension
%  coordinates in ascending order;
%  ADDCONONE is the additive constant for the first
%  dimensional model; ADDCONTWO is the additive constant for
%  the second dimensional model;
%  VAF is the variance-accounted-for in PROX by
%  the bidimensional scaling.
```

biscal_tied.m

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

```
%  first dimension; OUTPERMTWO is the final object permutation
%  for the second dimension;
%  COORDONE is the set of first dimension coordinates
%  in ascending order; COORDTWO is the set of second dimension
%  coordinates in ascending order;
%  ADDCONONE is the additive constant for the first
%  dimensional model; ADDCONTWO is the additive constant for
%  the second dimensional model;
%  VAF is the variance-accounted-for in PROX by
%  the bidimensional scaling.
```

biscaltmac.m

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

```
% indicated by INPERMONE and INPERMTWO;
% a value of NOPT = 1 uses iterative QA
% to locate the better permutations to fit.
```

cat_vs_con_orderfit.m

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

cat_vs_con_orderfnd.m

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

cent_linearfit

```
% CENT_LINEARFIT fits a structure to a proximity matrix by first fitting
% a centroid metric and secondly a linear unidimensional scale
% to the residual matrix where the latter is constrained by a given object order.
```

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

cent_linearfnd

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

centfit.m

```
% CENTFIT finds the least-squares fitted centroid metric (FIT) to
% PROX, the input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation).
%
% syntax: [fit,vaf,lengths] = centfit(prox)
%
% The $n$ values that serve to define the approximating sums,
% $g_{i} + g_{j}$, are given in the vector LENGTHS of size $n \times 1$.
```

centfittm.m

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

cirarobfit.m

```
% CIRAROBFIT fits a circular anti-Robinson matrix using iterative
% projection to a symmetric proximity matrix in the $L_{2}$-norm.
%
% syntax: [fit, vaf] = cirarobfit(prox,inperm,targ)
%
% 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 (around
% a circle); TARG is a given $n \times n$ matrix having the
% circular anti-Robinson form that guides the direction in which
% distances are taken around the circle.
% FIT is the least-squares optimal matrix (with variance-
% accounted-for of VAF) to PROX having a circular anti-Robinson
% form for the row and column object ordering given by INPERM.
```

cirarobfnd.m

```
% CIRAROBFND finds and fits a circular
% 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] = cirarobfnd(prox, inperm, kblock)
%
% PROX is the input proximity matrix ($n \times n$ with a zero
% main diagonal and a dissimilarity interpretation);
% INPERM is a given starting permutation (assumed to be around the
```

```
% circle) of the first $n$ integers; FIT is the least-squares optimal
% matrix (with variance-accounted-for of VAF) to PROX having a
% circular anti-Robinson form for the row and column
% object ordering given by the ending permutation OUTPERM.
% KBLOCK defines the block size in the use of the iterative
% quadratic assignment routine.
```

cirsarobfit.m

```
% CIRSAROBFIT fits a strongly circular anti-Robinson matrix
% using iterative projection to
% a symmetric proximity matrix in the $L_{2}$-norm.
%
% syntax: [fit, vaf] = cirsarobfit(prox,inperm,target)
%
% 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
% (around a circle);
% TARGET is a given $n \times n$ matrix having the circular
% anti-Robinson form that guides the direction in which distances
% are taken around the circle.
% FIT is the least-squares optimal matrix (with variance-
% accounted-for of VAF) to PROX having a strongly circular
% anti-Robinson form for the row and column object ordering
% given by INPERM.
```

cirsarobfnd.m

```
% CIRSAROBFND finds and fits a strongly circular
% 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] = cirsarobfnd(prox, inperm, kblock)
%
% PROX is the input proximity matrix ($n \times n$ with a zero
% main diagonal and a dissimilarity interpretation);
% INPERM is a given starting permutation (assumed to be around the
% circle) of the first $n$ integers;
% FIT is the least-squares optimal matrix (with variance-
% accounted-for of VAF) to PROX having a strongly
% circular 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.
```

circularplot.m

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

cirfit.m

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

cirfitac.m

```
%  CIRFITAC does a confirmatory fitting (including
%  the estimation of an additive constant) for a given order
%  (assumed to reflect a circular ordering around a closed
%  unidimensional structure) using the Dykstra--Kaczmarz
%  iterative projection least-squares method.
```

```
%
%  syntax: [fit, vaf, addcon] = cirfitac(prox,inperm)
%
%  INPERM is the given order; FIT is an $n \times n$ matrix that
%  is fitted to PROX(INPERM,INPERM) with variance-accounted-for of
%  VAF; ADDCON is the estimated additive constant.
```

class_scaledp.m

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

concave_monotonic_regression_dykstra.m

```
% CONCAVE_MONOTONIC_REGRESSION_DYKSTRA returns a vector of values in YHAT that are
% weakly monotonic with respect to the values in an independent input
% vector X, define a concave function, and minimize the least-squares
% loss to the values in the dependent input vector Y.  The
% variance-accounted-for in Y from YHAT is given by VAF_YHAT.
%
% syntax: [yhat,vaf_yhat] = concave_monotonic_regression_dykstra(x,y)
%
% The least-squares optimization is carried out through Dykstra's method of
```

```
% iterative projection.
```

convex_concave_monotonic_regression_dykstra.m

```
% CONVEX_CONCAVE_MONOTONIC_REGRESSION_DYKSTRA returns a vector of values
% in YHAT that are weakly monotonic with respect to the values in an
% independent input vector X, define a convex function up to the median
% observation in X and a concave function thereafter, and minimize the
% least-squares loss to the values in the dependent input vector Y.  The
% variance-accounted-for in Y from YHAT is given by VAF_YHAT.
%
% syntax: [yhat,vaf_yhat] = convex_concave_monotonic_regression_dykstra(x,y)
%
% The least-squares optimization is carried out through Dykstra's method of
% iterative projection.
```

convex_monotonic_regression_dykstra.m

```
% CONVEX_MONOTONIC_REGRESSION_DYKSTRA returns a vector of values in YHAT that are
% weakly monotonic with respect to the values in an independent input
% vector X, define a convex function, and minimize the least-squares
% loss to the values in the dependent input vector Y.  The
% variance-accounted-for in Y from YHAT is given by VAF_YHAT.
%
% syntax: [yhat,vaf_yhat] = convex_monotonic_regression_dykstra(x,y)
%
% The least-squares optimization is carried out through Dykstra's method of
% iterative projection.
```

eqspace_cirfitac.m

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

```
%  VAF; ADDCON is the estimated additive constant. The equally-spaced
%  output alternatives are prefixed with an EQ.
```

eqspace_linfitac.m

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

eqspace_ultrafit.m

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

least_squares_dykstra.m

```
% LEAST_SQUARES_DYKSTRA carries out an inequality constrained least-squares
```

```
% task using iterative projection.
%
% syntax: [solution, kuhn_tucker, iterations, end_condition] = ...
%    least_squares_dykstra(data,covariance,constraint_array, ...
%    constraint_constant,equality_flag)
%
%  The input arguments are an $n \times 1$ vector, DATA; an $n \times n$
%  positive-definite matrix, COVARIANCE; a $k \times n$ constraint matrix,
%  CONSTRAINT_ARRAY; the $k \times 1$ right-hand-side constraint vector,
%  CONSTRAINT_CONSTANT; and a $k \times 1$ EQUALITY_FLAG vector with
%  values of 0 when a corresponding constraint is an inequality; and 1 if
%  an equality.
%  The weighted least-squares criterion (with weights defined by the
%  inverse of COVARIANCE) is minimized by a SOLUTION that satisfies
%  CONSTRAINT_ARRAY * SOLUTION being in EQUALITY_FLAG relation to
%  CONSTRAINT_CONSTANT.  As additional output arguments, there is a $k
%  \times 1$ KUHN_TUCKER vector (useful for some applications); the number
%  of ITERATIONS taken (maximum default value is ITERMAX = 1.0e+04 set in
%  the program), and an END_CONDITION flag: 0: no error; 1: itermax
%  exceeded; 2: invalid constant; 3: invalid constraint function.
```

linear_order_member.m

```
% LINEAR_ORDER_MEMBER provides an (n-1) x n partition membership matrix
% called MEMBER based on the input permutation INPERM of size n.  MEMBER
% defines (with values of 1 and 2) those objects before and after each of
% the n-1 separations in INPERM.  MEMBER is used in the various
% PARTITIONFIT routines.
%
% syntax: [member] = linear_order_member(inperm)
```

linearplot.m

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

linfit.m

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

linfit_tied.m

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

linfitac.m

```
%  LINFITAC does a confirmatory fitting of a given unidimensional order
%  using the Dykstra--Kaczmarz iterative projection
%  least-squares method, but differing from linfit.m in
%  including the estimation of an additive constant.
%
%  syntax: [fit, vaf, coord, addcon] = linfitac(prox,inperm)
%
%  INPERM is the given order;
%  FIT is an $n \times n$ matrix that is fitted to
%  PROX(INPERM,INPERM) with variance-accounted-for VAF;
```

```
%  COORD gives the ordered coordinates whose absolute differences
%  could be used to reconstruct FIT; ADDCON is the estimated
%  additive constant that can be interpreted as being added to PROX.
```

linfitac_altcomp.m

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

linfitac_missing.m

```
%  LINFITAC_MISSING does a confirmatory fitting of a given unidimensional order
%  using the Dykstra--Kaczmarz iterative projection
%  least-squares method, but differing from linfit.m in
%  including the estimation of an additive constant;also, missing entries
%  in the input proximity matrix PROX are given values of zero.
%
%  syntax: [fit, vaf, addcon] = ...
%                linfitac_missing(prox,inperm,proxmiss)
%
%  INPERM is the given order;
%  FIT is an $n \times n$ matrix that is fitted to
%  PROX(INPERM,INPERM) with variance-accounted-for VAF;
%  ADDCON is the estimated additive constant that can be interpreted
%  as being added to PROX.  PROXMISS is the same size as PROX (with main
%  diagonal entries all zero); an off-diagonal entry of 1.0 denotes an
%  entry in PROX that is present and 0.0 if it is absent.
```

linfitac_tied.m

```
%  LINFITAC_TIED does a confirmatory fitting of a given unidimensional order
```

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

linfitl1.m

```
%LINFITL1 does a confimatory fitting in the $L_{1}$ norm of a given unidimensional
%  order using linear programming.
%  INPERM is the given order; FIT is an $n \times n$ matrix that is fitted to
%  PROX(INPERM,INPERM) with $L_{1}$ value DIFF; COORD gives the ordered coordinates
%  whose absolute differences could be used to reconstruct FIT.
%  EXITFLAG indicates the success of the optimization ( > 0 indicates convergence;
%  0 indicates that the maximum number of function evaluations or iterations were
%  reached; and < 0 denotes nonconvergence).
```

linfitl1ac.m

```
%LINFITL1AC does a confimatory fitting in the $L_{1}$ norm of a given unidimensional
%  order using linear programming, with the estimation of
%  an additive constant (ADDCON).
%  INPERM is the given order; FIT is an $n \times n$ matrix that is fitted to
%  PROX(INPERM,INPERM) with deviance DEV; COORD gives the ordered coordinates
%  whose absolute differences could be used to reconstruct FIT.
%  EXITFLAG indicates the success of the optimization ( > 0 indicates convergence;
%  0 indicates that the maximum number of function evaluations or iterations were
%  reached; and < 0 denotes nonconvergence).
```

linfittm.m

```
%  LINFITTM does a confirmatory two-mode fitting of a given
```

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

linfittmac.m

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

linfittmac_altcomp.m

```
%  LINFITTMAC_ALTCOMP does a confirmatory two-mode fitting of a given
```

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

lpfit.m

```
%lpfit.m provides the two-dimensional coordinates used to construct lp
%distances fit (in a confirmatory manner) to the given proximities.

%The inputs:
%
%the n x n symmetric dissimilarity matrix with zero main diagonal entries;
%two fixed permutations,  PERMONE and PERMTWO, of the objects along the
%two dimensions;
% the value of p to obtain the p distances (p = 1: city-block distances; p
% = 2: Euclidean distances; large p: dominance distances).

%The outputs:
%
%the values fitted to the proximity matrix are in FITTED;
%the object coordinates along the two dimensions, COORDONE and COORDTWO;
%the additive constant, ADDCON;
%the values of the least-squares criterion, FVAL, in finding the
%coordinates;
%the variance-accounted-for measure, VAF;
%the exit condition, EXITFLAG, from using the MATLAB optimization routine,
```

```
%fmincon.m.
%
% syntax: [fitted, coordone,coordtwo,addcon,fval,vaf,exitflag] = ...
%    lpfit(prox,permone,permtwo,start_vector,p)
```

lsqisotonic.m – from the MATLAB Statistics Toolbox

```
%LSQISOTONIC Isotonic least squares.
%   YHAT = LSQISOTONIC(X,Y) returns a vector of values that minimize the
%   sum of squares (Y - YHAT).^2 under the monotonicity constraint that
%   X(I) > X(J) => YHAT(I) >= YHAT(J), i.e., the values in YHAT are
%   monotonically non-decreasing with respect to X (sometimes referred
%   to as "weak monotonicity").  LSQISOTONIC uses the "pool adjacent
%   violators" algorithm.
%
%   If X(I) == X(J), then YHAT(I) may be <, ==, or > YHAT(J) (sometimes
%   referred to as the "primary approach").  If ties do occur in X, a plot
%   of YHAT vs. X may appear to be non-monotonic at those points.  In fact,
%   the above monotonicity constraint is not violated, and a reordering
%   within each group of ties, by ascending YHAT, will produce the desired
%   appearance in the plot.
%
%   YHAT = LSQISOTONIC(X,Y,W) performs weighted isotonic regression using
%   the non-negative weights in W.

%   Copyright 2003-2006 The MathWorks, Inc.
%   $Revision: 1.1.6.5 $  $Date: 2006/06/20 20:51:42 $

%   References:
%      [1] Kruskal, J.B. (1964) "Nonmetric multidimensional scaling: a
%          numerical method", Psychometrika 29:115-129.
%      [2] Cox, R.F. and Cox, M.A.A. (1994) Multidimensional Scaling,
%          Chapman&Hall.
```

matrix_colorcode.m

```
% MATRIX_COLORCODE constructs a color representation for the values in an
% $n \times m$ matrix, DATAMATRIX.  The rows and columns of DATAMATRIX are
% permuted by ROWPERM and COLPERM, respectively.  CMAP is the input
% colormap for the representation (e.g., bone(256)).
```

matrix_movie.m

```
% MATRIX_MOVIE constructs a color movie of the effects of a series
% of permutations on a proximity matrix.
% DATAMATRIX is an n by n symmetric proximity matrix; PERMS is
% a cell array containing NUMPERMS permutations; CMAP
% is the input colormap used for the representation (e.g.,
% bone(256)).
```

monotonic_regression_dykstra.m

```
% MONOTONIC_REGRESSION_DYKSTRA returns a vector of values in YHAT that are
% weakly monotonic with respect to the values in an independent input
% vector X, and minimize the least-squares loss to the values in the
% dependent input vector Y.  The variance-accounted-for in Y from YHAT is
% given by VAF_YHAT.
%
% syntax: [yhat,vaf_yhat] = monotonic_regression_dykstra(x,y)
%
% The least-squares optimization is carried out through Dykstra's method of
% iterative projection.
```

order.m

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

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

order_missing.m

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

orderpartitionfit.m

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

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

orderpartitionfnd.m

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

ordertm.m

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

partitionfit.m

```
% PARTITIONFIT provides a least-squares approximation to a proximity
% matrix based on a given collection of partitions.
%
% syntax: [fitted,vaf,weights,end_condition] = partitionfit(prox,member)
```

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

partitionfit_addcon.m

```
% PARTITIONFIT provides a least-squares approximation to a proximity
% matrix based on a given collection of partitions.
%
% syntax: [fitted,vaf,weights,end_condition] = partitionfit_addcon(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.  ADDCON is an automatically estimated additive constrant.
```

partitionfit_twomode.m

```
% PARTITIONFIT_TWOMODE provides a least-squares approximation to a two-mode
% proximity matrix based on a given collection of partitions.
%
% syntax: [fitted,vaf,weights,end_condition] = partitionfit_twomode(prox,member)
%
% PROXTM is the nrow x ncol two-mode input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation); MEMBER is the m x n (n = nrow+ ncol) 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 order used for PROXTM, with rows first and columns to follow.
```

```
% FITTED is an nrow x ncol matrix fitted to PROXTM (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 PROXTM by the fitted matrix FITTED.
% END_CONDITION should be zero for a normal termination of the optimization
% process.
```

partitionfit_twomode_addcon

```
% PARTITIONFIT_TWOMODE_ADDCON provides a least-squares approximation to a two-mode
% proximity matrix based on a given collection of partitions.
%
% syntax: [fitted,vaf,weights,end_condition] = partitionfit_twomode_addcon(prox,member)
%
% PROXTM is the nrow x ncol two-mode input proximity matrix (with a zero main diagonal
% and a dissimilarity interpretation); MEMBER is the m x n (n = nrow+ ncol) 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 order used for PROXTM, with rows first and columns to follow.
% FITTED is an nrow x ncol matrix fitted to PROXTM (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 PROXTM by the fitted matrix FITTED.
% END_CONDITION should be zero for a normal termination of the optimization
% process.  ADDCON is an automatically estimated additive constant.
```

proxmon.m

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

proxmontm.m

```
%  PROXMONTM produces a monotonically transformed
%  two-mode proximity matrix (MONPROXPERMUTTM)
%  from the order constraints obtained
%  from each pair of entries in the input two-mode
%  proximity matrix PROXPERMUTTM (with a dissimilarity
%  interpretation).
%
%  syntax: [monproxpermuttm, vaf, diff] = ...
%       proxmontm(proxpermuttm, fittedtm)
%
%  MONPROXPERMUTTM is close to the $nrow \times ncol$
%  matrix FITTEDTM in the least-squares sense;
%  The variance accounted for (VAF) is how much variance
%  in MONPROXPERMUTTM can be accounted for by FITTEDTM;
%  DIFF is the value of the least-squares criterion.
```

proxstd.m

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

sarobfit.m

```
% SAROBFIT fits a strongly anti-Robinson matrix using
% iterative projection to a symmetric proximity matrix
% in the $L_{2}$-norm.
% PROX is the input proximity matrix ($n \times n$ with a
% zero main diagonal and a dissimilarity interpretation).
%
% syntax: [fit, vaf] = sarobfit(prox, inperm)
%
% 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 a strongly
% anti-Robinson form for the row and column
% object ordering given by INPERM.
```

sarobfnd.m

```
% SAROBFND finds and fits a strongly
% 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] = sarobfnd(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 a strongly
% 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.
```

skew_symmetric_lawler_dp.m

```
%  SKEW_SYMMETRIC_LAWLER_DP carries out a reordering of a skew-symmetric proximity
%  matrix using dynamic programming, and by maximizing the above-diagonal
%  sum of proximities.
%  PROX is the input skew-symmetric proximity matrix (with a zero main
%  diagonal);
%  PERMUT is the order of the objects in the optimal permutation;
%  PROX_PERMUT is the reordered proximity matrix using PERMUT;
%  CUMOBFUN gives the cumulative values of the objective function for
%  the successive placements of the objects in the optimal permutation.

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

skew_symmetric_scaling.m

```
% SKEW_SYMMETRIC_SCALING performs a closed-form least squares scaling of a
% skew-symmetric proximity matrix.
%
% syntax: [coord,sort_coord,permut,prox_permut,vaf,...
%   alpha_multiplier,alpha_vaf,alpha_coord] = skew_symmetric_scaling(prox)
%
% COORD contains the least-squares coordinates which are sorted from
% smallest to largest in SORT_COORD; VAF is the variance they account for;
% PERMUT is the object permutation
% corresponding to SORT_COORD with PROX_PERMUT the reordered skew-symmetric
% proximity matrix.  For equally-spaced coordinates, ALPHA_MULTIPLIER
% defines the multiplicative constant on the integer-valued coordinates; a
% collection of equally-spaced coordinates is given by ALPHA_COORD with
% ALPHA_VAF the variance they account for.
```

targcir.m

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

targlin.m

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

ultrafit.m

```
% ULTRAFIT fits a given ultrametric using iterative projection to
% a symmetric proximity matrix in the $L_{2}$-norm.
%
% syntax: [fit,vaf] = ultrafit(prox,targ)
%
% PROX is the input proximity matrix (with a zero main diagonal
```

```
% and a dissimilarity interpretation);
% TARG is an ultrametric matrix of the same size as PROX;
% FIT is the least-squares optimal matrix (with
% variance-accounted-for of VAF) to PROX satisfying the ultrametric
% constraints implicit in TARG.
```

unicirac.m

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

uniscallp.m

```
%UNISCALLP carries out a unidimensional scaling of a symmetric proximity
%  matrix using iterative linear programming.
%  PROX is the input proximity matrix (with a zero main diagonal and a
%  dissimilarity interpretation);
%  INPERM is the input beginning permutation (a permuation of the first $n$ integers).
%  OUTPERM is the final permutation of PROX.
%  COORD is the set of coordinates of the unidimensional scaling
%  in ascending order;
%  DIFF is the value of the l1 loss function for the
%  coordinates and object permutation; and FIT is the matrix of absolute
%  coordinate differences being fit to PROX(OUTPERM,OUTPERM).
```

uniscallpac.m

```
%UNISCALLPAC carries out a unidimensional scaling of a symmetric proximity
%  matrix using iterative linear programming, with the inclusion of an
%  additive constant (ADDCON) in the model.
%  PROX is the input proximity matrix (with a zero main diagonal and a
%  dissimilarity interpretation);
%  INPERM is the input beginning permutation (a permuation of the first $n$ integers).
%  OUTPERM is the final permutation of PROX.
%  COORD is the set of coordinates of the unidimensional scaling
%  in ascending order;
%  DEV is the value of deviance (the normalized $L_{1}$ loss function) for the
%  coordinates and object permutation; and FIT is the matrix being fit to
%  PROX(OUTPERM,OUTPERM) with the given deviance.
```

uniscaltmac_altcomp.m

```
% UNISCALTMAC finds and fits a linear
% unidimensional scale using iterative projection to
% a two-mode proximity matrix in the $L_{2}$-norm based on a
% permutation identified through the use of iterative
% quadratic assignment.
%
% syntax: [find,vaf,outperm,rowperm,colperm,addcon,coord] = ...
%    uniscaltmac_altcomp(proxtm,inperm,kblock)
%
% PROXTM is the input two-mode proximity matrix
% ($n_{a} \times n_{b}$ with a zero main diagonal
% and a dissimilarity interpretation);
% INPERM is a given starting permutation of the
% first $n = n_{a} + n_{b}$ integers;
% FIND is the least-squares optimal matrix (with
% variance-accounted-for of VAF) to PROXTM having a linear
% unidimensional form for the row and column
% object ordering given by the ending permutation OUTPERM.
% The spacings among the objects are given by the entries in FIND.
% KBLOCK defines the block size in the use of the iterative
% quadratic assignment routine.
% The additive constant for the model is given by ADDCON.
% ROWPERM and COLPERM are the resulting row and column
% permutations for the objects.  The nrow + ncol coordinates
% (ordered with the smallest set at a value of zero)
% are given in COORD.
```

uniscalqa.m

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