Yet Another Smacof - Square Symmetric Case

Jan de Leeuw

June 8, 2025

We rewrite the metric/nonmetric and weighted/unweighted versions of the smacof program for square symmetric data as four monolithic C programs, with R used for taking care of the setup, the I/O, and of issuing a single call to *.C()* to start the computations. This makes this new *smacofSS()* program five to fifty times as fast (for our examples) as the *smacofSym()* function from the R smacof package. Utilities for various initial configurations and plots are included in the package.

Table of contents

Note: This is a working manuscript which will be expanded/updated frequently. All suggestions for improvement are welcome. All Rmd, tex, html, pdf, R, and C files are in the public domain. Attribution will be appreciated, but is not required. The files can be found at https://github.com/deleeuw/smacofFlat

1 Introduction

In Multidimensional Scaling (MDS) we start with n *objects*. Objects can be anything: people, animals, molecules, locations, time points, stimuli, political parties, and so on. We have information about the *similarities* or *dissimilarities* of some or all of the $\binom{n}{2}$ pairs of objects. Say we have information about m dissimilarities δ_k, with $k = 1, \cdots m$. Thus each index k refers to a pair of indices (i, j), with $1 \leq i \leq n$ and $1 \leq j \leq n$. The information is of the form $\delta \in \Delta$, where Δ is a known subset of the non-negative orthant of $\mathbb{R}^m$. If the dissimilarities are known numbers and Δ has only a single element the MDS problem is *metric*, in all other cases the problem is *non-metric*.

In MDS we want to map the objects into *points* in p-dimensional Euclidean space in such a way that the distances d_k between the points approximate the dissimilarities δ_k.
The quality of the approximation is given by the least squares loss function[1]

$$\sigma(X, \Delta) := \frac{\sum_{k=1}^{m} w_k(\delta_k - d_k(X))^2}{\sum_{k=1}^{m} w_k \delta_k^2}, \tag{1}$$

Traditionally, this loss function is called *stress* (Kruskal (1964a), Kruskal (1964b)).

In definition (1)

- δ_k are m non-negative *pseudo-distances*;
- w_k are m positive *weights*;
- X is an $n \times p$ *configuration*, with coordinates of n *points* in $\mathbb{R}^p$;
- $d_k(X)$ are the (Euclidean) *distances* between the rows of the configuration matrix.

The (unconstrained, least squares, square, symmetric, Euclidean, $p-$dimensional) MDS problem is to minimize σ of (1) over all $n \times p$ configurations and over the set $\Delta \in \mathbb{R}_+^m$ of non-negative pseudo-distances.

[1] The symbol := is used for definitions.

2 MDS Data Structure

Data management and algorithm initialization is handled by R (R Core Team (2025)). We start with an R object of class *dist* containing dissimilarities. Here is a small example of order four.

```
  1 2 3
2 1
3 3 1
4 2 3 1
```

Turn this into MDS data with the utility *makeMDSData()*, which creates an object of class *smacofSSData*.

```
smallData <- makeMDSData(small)
print(smallData)
```

```
$iind
[1] 2 3 4 4 3 4

$jind
[1] 1 2 3 1 1 2

$delta
[1] 1 1 1 2 3 3

$blocks
[1] 3 0 0 1 2 0

$weights
[1] 1 1 1 1 1 1

$nobj
[1] 4

$ndat
[1] 6

attr(,"class")
[1] "smacofSSData"
```

Note that the data in column *delta* are increasing, and that *blocks* has tie blocks. Also, the *weights* are always there in some form or another, even for MDS analyses that are unweighted (i.e. when all w_k are equal).

An object of class *smacofSSData* is *complete* if all $\binom{n}{2}$ dissimilarities are present and *unweighted* if all w_k are equal to one. Note the *weights* are always part of the *smacofSSData* object, even for MDS analyses that are unweighted. *makeMDSData()* can handle missing data and nontrivial weights. If our example is

```
   1  2  3
2 NA
3  3  1
4 NA  3  1
```

and we add weights, also of class *dist*,

```
  1 2 3
2 1
3 1 3
4 2 1 0
```

then our *smacofSSData* object becomes

```
smallData <- makeMDSData(smallMissing, smallWeights)
print(smallData)
```

```
$iind
[1] 3 3 4

$jind
[1] 2 1 2

$delta
[1] 1 3 3

$blocks
[1] 1 2 0

$weights
```

```
[1] 3 1 1

$nobj
[1] 4

$ndat
[1] 3

attr(,"class")
[1] "smacofSSData"
```

In the dist objects *delta* and *weights* the missing data are coded as zero weights, or as dissimilarities and/or weights that are *NA*. Zero dissimilarities do not indicate missing data. Thus weights are always strictly positive and missing data do not enter into the data object at all.

These conventions make it possible to also handle rectangular off-diagonal data, such as this *delta* and *weights*.

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


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

Now *makeMDSData()* gives

```
$iind
 [1] 5 6 7 5 5 6 7 7 5 6 6 7
```

```
$jind
 [1] 1 2 2 3 4 1 3 1 2 3 4 4

$delta
 [1] 1 1 1 1 1 2 2 3 3 3 3 3

$blocks
 [1] 5 0 0 0 0 2 0 5 0 0 0 0

$weights
 [1] 1 1 1 1 1 1 1 1 1 1 1 1

$nobj
[1] 7

$ndat
[1] 12

attr(,"class")
[1] "smacofSSData"
```

Notehowever that handling rectangular data with square symmetric MDS is inefficient, and it is better to use smacof programs specifically intended for rectangular data.

It is of course also possible to *smacofSSData* objects in other ways, and to edit the objects generated by *makeMDSData()*, for instance by deleting/adding observations or transforming weights/dissimilarities. As long as the conventions are obeyed that no index pair (i, j) occurs more than once, that the dissimilarities remain sorted, and that the tie blocks faithfully reflect ties in the sorted dissimilarities. We do need $i \neq j$, but it is not necessary that always $i > j$.

3 Arguments

The smacofSS() function has the following arguments, with default values,

1. *theData*, a *smacofSSData* object.

2. *ndim=2*, dimensionality of MDS analysis.
3. *xinit=NULL*, initial configuration, NULL or an *nobj* by *ndim* matrix.
4. *weighted = FALSE*, TRUE/FALSE for weighted/unweighted least squares.
5. *ordinal = FALSE*, FALSE or 0 for numerical, 1,2,3 for ordinal.
6. *itmax = 1000*, maximum number of iterations.
7. *eps = 1e-10*, if stress changes less than *eps*, stop.
8. *digits = 10*, digits stress print if verbose is TRUE,
9. *width = 12*, width stress print if verbose is TRUE,
10. *verbose = FALSE*, TRUE/FALSE print stress for each iteration to *stdout*.

In *ordinal* we use 1 for the primary approach to ties, 2 for the secondary approach, and 3 for the tertiary approach (De Leeuw (1977b)). The default initial configuration is the classical scaling solution, with missing data imputed as average non-missing dissimilarities.

smacofSS() is a front end that reads the parameters of the problem and then calls one of four separate R programs. They are

1. *smacofSSUR()*, unweighted numerical;
2. *smacofSSWR()*, weighted numerical;
3. *smacofSSUO()*, unweighted ordinal;
4. *smacofSSWO()*, weighted ordinal.

Although using four separate programs involves duplicating some code it does allow us to make the programs smaller and faster. For example, we are not have to do weighted updates with unit weights in the unweighted case, and we don't have to branch intoma different part of the code each time for either ordinal and numerical.

In what follows "foo" can be "UR", "WR", "UO", or "WO". Each of the four R programs *smacofSSfoo()* calls a corresponding shared library, called *smacofSSfooEngine.so*, which contains the compiled code of the C routine *smacofSSfooEngine()*.

Each of the four R programs *smacofSSfoo()* can also be called directly, without using the *smacofSS()* frontend. Their arguments and defualts are the same as for *smacofSS()*, except that *weighted* and *ordinal* are not there, while *smacofSSUO()* and *smacofSSWO()* have the additional argument *ties*, equal to 1 (default), 2, or 3.

4 Details

The C routines *smacofSSfooEngine()* by themselves are complete smacof MDS programs. It would be easy to write main programs in C that do the same job as the R frontend, and compile to stand-alone executables.

The C code uses the *.C()* calling conventions and is otherwise not dependent on R in any way. It is important that the *.C()* call from R is only executed one time in a *smacofSS()* job, when loading the shared library, and after that there is only compiled code running until the end of the job. In other words, everything done in R is either front-end or back-end, and is only done once. Earlier versions of the R/C program had R also managing the iterations. In each iteration there was a *.C()* call to update the configuration and in the ordinal case an additional *.C()* call to update the pseudo-distances. This turned out to be unsatisfactory, because it did not give enough speedup compared to the *smacofSym()* in the *smacof* package on CRAN (De Leeuw and Mair (2009), Mair, Groenen, and De Leeuw (2022)).

If *ordinal* is TRUE the monotone regression in each iteration uses the C version of the *monotone()* algorithm of Busing (2022). The *monotone()* algorithm is called in three separate C routines *primaryApproach()*, *secondaryApproach()*, and *tertiaryApproach()*, which in their turn are called in each iteration by the *smacofSSUOEngine()* or *smacofSSWOEngine()* C routines.

The default initial configuration is set to NULL for *smacofSS()*. This is mean to emphasize the importance of choosing an initial configuration. But to make life easier for the user if *xinit* is null, then an initial configuration is computed by the *smacofTorgerson()* routine in *smacofUtils.R*.

smacofTorgerson() is classical MDS, with the missing dissimilarities imputed using the average non-missing dissimilarity. The eigenvalues and eigenvectors are computed with *eigs-sym()* from the *RSpectra* package (Qiu and Mei (2024)). The Torgerson configuration works best for complete and unweighted data, but may be satisfactory for a small number of randomly missing data and for weights that are not too unequal. Also remember that in the case of a really bad fit classical MDS may run into the problem of negative eigenvalues. This will give the initial configuration a dimension less than p, and since smacof iterations never increase dimensionality this will mean the final configuration will also have dimension less than p. This makes the Torgerson initial configuration unsuitable, for example, for full-dimensional scaling (De Leeuw, Groenen, and Mair (2016)).

There are some additional initial configuration routines included. We can generate random missing data with *smacofRandomConfiguration()* which is mainly useful for theoretical studies of convergence and local minima. The Guttman-Lingoes initial configuration (Guttman (1968)), which handles missing data and unequal weights, is provided as *smacofGuttman()*. It is closely related to Hayashi's Quantification Method IV (Takane (1977)) and does not have the negative eignevalue problem.

In addition there is *smacofElegant()*, which implements an optimized version of the algorithm of De Leeuw (1975), as described recently in De Leeuw (2025). *smacofElegant()* is a stand-alone iterative metric scaling method, implemented in R. Having it converge is a lot of work just to obtain an initial configuration for *smacofSS()*. On the other hand it is crucial to have an initial configuration which is as good as possible, because that is the best guarantee against local minima. Finally, we have the option to uses the p dominant dimensions of the full-dimensiponal scaling solution as initial configuration. We know that full-dimensional scaling converges to the global minimum of stress in $n-1$ dimensions, and actually to the global minimum for all $p \geq r$, where r is the Gower rank of the data (De Leeuw (2016)). This will provide a good initial estimate if r is close to p. This final option again takes missing data and weights fully into account.

In terms of strategy we suggest that in an MDS analysis the researcher first computes one of more initial configurations (which are all independent MDS solutions anyway, except for the random configuration), and then give one or more of these initial configurations as the *xinit* argument to *smacofSS()*.

5 Value

The list of objects returned by *smacofSS()* mimics, as much as possible, the list returned by *smacofSym()*.

1. *delta*, dissimilarities, vector of length m.
2. *dhat*, final pseudo-distances, vector of length m.
3. *confdist*, final distances, vector of length m.
4. *conf* , final configuration, $n \times p$ matrix.
5. *weightmat*, weights, vector of length m.
6. *stress*, final stress.
7. *ndim*, number of dimensions.
8. *init*, initial configuration, $n \times p$ matrix..
9. *niter*, number of iterations.
10. *nobj*, number of objects.
11. *iind*, row indices, vector of length m.
12. *jind*, column indices, vector of length m.
13. *weighted*, was the analysis weighted.
14. *ordinal*, was the analysis ordinal (and if so, which tie approach).

One important special case should be kept in mind. In the ordinal case with the primary approach to ties, the data are (potentially) reordered within tie blocks in each iteration. In each iteration we have to reorder iind, jind, and in the weighted case the weights. There is no need to reorder delta and the blocks, because the only ordering changes are within blocks. Ultimately this means that if smacofSS() returns a list *h*, then $hiind*$, $*h$jind, and in the weighted case *h$weightmat* will be ordered differently from the corresponding columns in the MDS data structure. No reordering is going on using the secondary and tertiary approaches.

6 Utilities

6.1 Data Utilities

There are two functions in smacofDataUtilities.R

1. *makeMDSData(delta, weights = NULL)*. Returns *smacofSSData* object from dist objects.
2. *fromMDSData(theData)*. Returns *dist* objects of dissimilarities and weights from *smacofSSData* object.

6.2 Auxilaries

There are several computational routines in *smacofUtils.R*. Ideally this should be C routines as well, but the R versions are only used once in a job and they rely on compiled code in their calculations anyway. So translating them to C will be no major speedup.

The Moore-Penrose inverse of V uses the regular inverse $V^+ = (V + n^{-1}ee')^{-1} - n^{-1}ee'$, which is valid because the weights are assumed to be irreducible (De Leeuw (1977a)). It is the only place in *smacofSS()* where a full symmetric matrix of order n is used. What is passed to C is the strictly lower-diagonal part of V in a vector. The matrix multiplications in C by the B matrix and the V matrix do not need the diagonal elements.

1. *makeMPInverseV(theData)*. Returns vector with the lower diagonal of the Moore-Penrose inverse of V matrix.
2. *smacofTorgerson(theData, ndim)*. Returns initial configuration in $nobj \times ndim$ matrix using classical MDS.
3. *smacofGuttman(theData, ndim)*. Returns the Guttman-Lingoes initial configuration in $nobj \times ndim$ matrix.
4. *smacofElegant(theData, ndim)*. Returns the "elegant" initial configuration in $nobj \times ndim$ matrix.

6.3 Plotting

There are three plot routines in the file *smacofPlots.R*.

1. *smacofShepardPlot(h, main = "ShepardPlot", fitlines = TRUE, colline = "RED", colpoint = "BLUE", resolution = 100, lwd = 2, cex = 1, pch = 16)*
2. *smacofConfigurationPlot(h, main = "ConfigurationPlot", labels = NULL, dim1 = 1, dim2 = 2, pch = 16, col = "RED", cex = 1)*

3. *smacofDistDhatPlot(h, fitlines = TRUE, colline = "RED", colpoint = "BLUE", main = "Dist-Dhat Plot", cex = 1, lwd = 2, pch = 16)*

The Shepard plot has the original dissimilarities on the horizontal axis, and both the pseudo-distances and the distances on the vertical axis. Pseudo-distances are plotted as points of color *colline*, and are connected by a line of color *colline*. Distances are plotted as points if color *colpoint*. If *fitlines* is TRUE then black vertical lines from the distance point to the pseudo-distance point are drawn.

The configuration plot plots two dimensions *dim1* and *dim2* of the configuration. If *labels* is NULL points are drawn with symbol *pch*, otherwise a vector of labels is used.

A DistDhat-plot has distances on the horizontal axis and pseudo-distances on the vertical axes. Points are drawn using symbol *pch* in color *colpoint*. The line through the origin with slope one is drawn in color *colline*. If *fitlines* is true then black lines from the points to their orthogonal projections on the line are drawn.

7 Example Data Sets

There are a number of example data sets in the directory smacofSSData. They are mostly of the 20th century boomer type: small and collected from the aggregated judgments of human subjects. All data are available both as dist objects and as MDS data structures.

7.1 Ekman Data

Similarity data between 14 colors from Ekman (1954). Rating scale similarity judgments averaged over subjects, linearly converted to unit interval dissimilarities.

7.2 Morse Data

Dissimilarity between the Morse codes of 36 letters and numbers from Rothkopf (1957). Confusion probabilities transformed to dissimilarities.

7.3 Gruijter Data

Dissimilarities of nine Dutch political parties in 1967, from De Gruijter (1967). Collected using the method of triads and averaged over 100 subjects.

7.4 Wish Data

Rating scale similarities between 12 nations, collected by Wish (1971). Averaged over subjects, subtracted from maximum scale value to form dissimilarities.

7.5 Iris Data

Classical iris data from Anderson (1936) via Fisher (1936). Euclidean distances over four measurements on 150 irises.

8 Comparisons

8.1 Outcome

In this section we compare the output (final stress, number of iterations) of smacofSym() and smacofSS() using the Ekman and Morse data. Note that smacofSym() reports the square root of the final stress, following Kruskal (1964a), so for comparison purposes we square it again. All runs are started with the two-dimensional torgerson solution and have the stop criteria *eps* equal to 1e-10 and *itmax* equal to 1000 (except when using the tertiary approach when *itmax* is 10000). In the weighted case the weights for Ekman are $w_k = \delta_k^2$ and those for Morse are $w_k = \delta_k^{-1}$.

Table 1: Comparison smacofSym() and smacofSS() results Ekman data

	Sym stress	Sym niter	SS stress	SS niter
unweighted numerical	0.0172132	25	0.0172132	25
unweighted ordinal ties = 1	0.0005337	103	0.0005337	103
unweighted ordinal ties = 2	0.0009977	51	0.0009977	51
unweighted ordinal ties = 3	0.0000001	2556	0.0000001	2556
weighted numerical	0.0105187	22	0.0105187	22
weighted ordinal ties = 1	0.0003205	78	0.0003205	78
weighted ordinal ties = 2	0.0007063	64	0.0007063	64
weighted ordinal ties = 3	0.0000002	4650	0.0000002	4650

Table 2: Comparison smacofSym() and smacofSS() results Morse data

	Sym stress	Sym niter	SS stress	SS niter
unweighted numerical	0.0899492	238	0.0899492	238
unweighted ordinal ties = 1	0.0326557	143	0.0326557	143
unweighted ordinal ties = 2	0.0406405	135	0.0406405	135
unweighted ordinal ties = 3	0.0000018	351	0.0000018	351
weighted numerical	0.0977124	317	0.0977124	317
weighted ordinal ties = 1	0.0346208	117	0.0346208	117
weighted ordinal ties = 2	0.0425777	99	0.0425777	99
weighted ordinal ties = 3	0.0000025	289	0.0000025	289

The conclusion is clear. For the Ekman and Morse examples the results of *smacofSym()* and *smacofSS()* are identical.

8.2 Time

The *microbenchmark* package (Mersmann (2024)) is intended to time small pieces of code, not complete programs. Nevertheless we will use it to compare *smacofSym()* and *smacofSS()*, until something better comes along. We again use the Ekman and Morse data with default options. From the microbenchmark output we find the median time over 100 runs each of the two programs.

Table 3: Comparison smacofSym() and smacofSS() running times Ekman Data

	SS time	Sym time	ratio Sym/SS
unweighted numerical	112114.5	1230348	10.974035
unweighted ordinal ties = 1	119699.5	6789887	56.724439
unweighted ordinal ties = 2	97190.5	2365782	24.341700
unweighted ordinal ties = 3	153832.0	119421376	776.310368
weighted numerical	242166.5	1102716	4.553543
weighted ordinal ties = 1	249587.5	5251710	21.041561
weighted ordinal ties = 2	233864.0	3004767	12.848352
weighted ordinal ties = 3	312092.0	222468276	712.829151

The entries in the table for the tertiary approach are suspect, because it uses a huge number of iterations to arrive at a perfect solution. In many of the microbenchmark runs it may actually use the full 10000 iterations. From the better conditioned numerical and ordinalsolutions we see that smacofSS is between 4.5 and 56.7 times as fast as smacofSym. The improvment is largest in the unweighted analyses.

Table 4: Comparison smacofSym() and smacofSS() running times Morse data

	SS time	Sym time	ratio Sym/SS
unweighted numerical	1040150	17952322	17.259367
unweighted ordinal ties = 1	3860826	18041804	4.673042
unweighted ordinal ties = 2	1074262	11143452	10.373128
unweighted ordinal ties = 3	2382961	28658303	12.026342
weighted numerical	2479168	23882869	9.633423
weighted ordinal ties = 1	4084974	14449650	3.537269
weighted ordinal ties = 2	1824520	7695454	4.217795
weighted ordinal ties = 3	2928814	23933340	8.171682

We see that smacofSS() is four to eighteen times faster than smacofSym(). The difference is largest in the numerical case, and it is also larger for unweighted than for weighted. Strangely

enough for the ordinal options the weighted case generally seems faster than the corresponding non-weighted case. We also see that primary is faster than secondary and secondary is faster than tertiary. Again, there is a need for some tinkering with the code to see if these relations are real and stable or are a result of poor coding and various artefacts. Note that we have used microbenchmark in such a way that smacofSS() is always run after the corresponding smacofSym(), and that the eight different basic combinations or weighted and ordinal are always run in the same order.

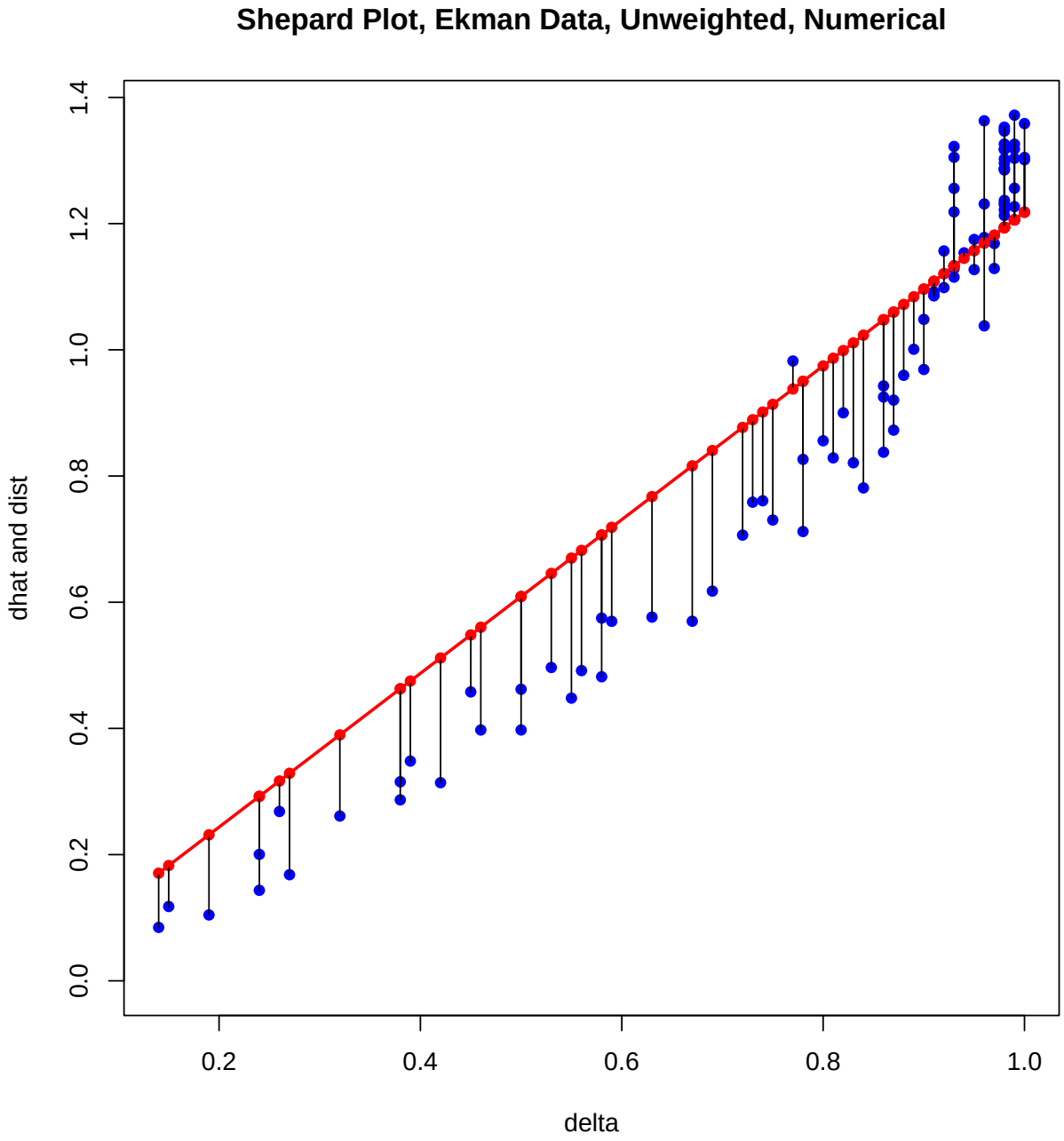
9 Real Examples

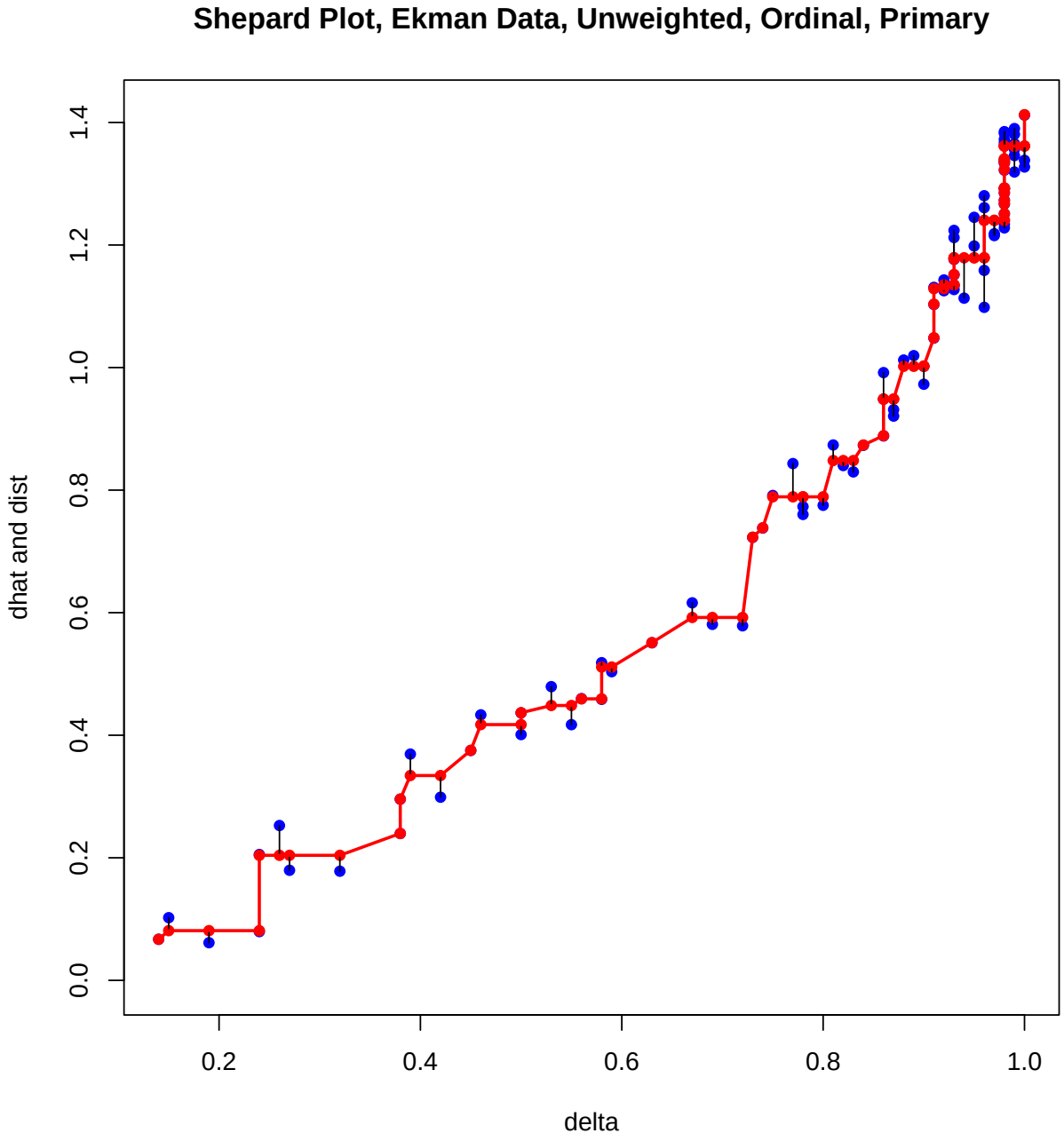
To show some plots we repeat eight possible analyses using the Ekman and Morse data. The number of iterations and the final stress have already been reported previously in the comparisons section. Both data sets have a fairly large number of ties, so we expect the choice of the ties approach to have some effect.

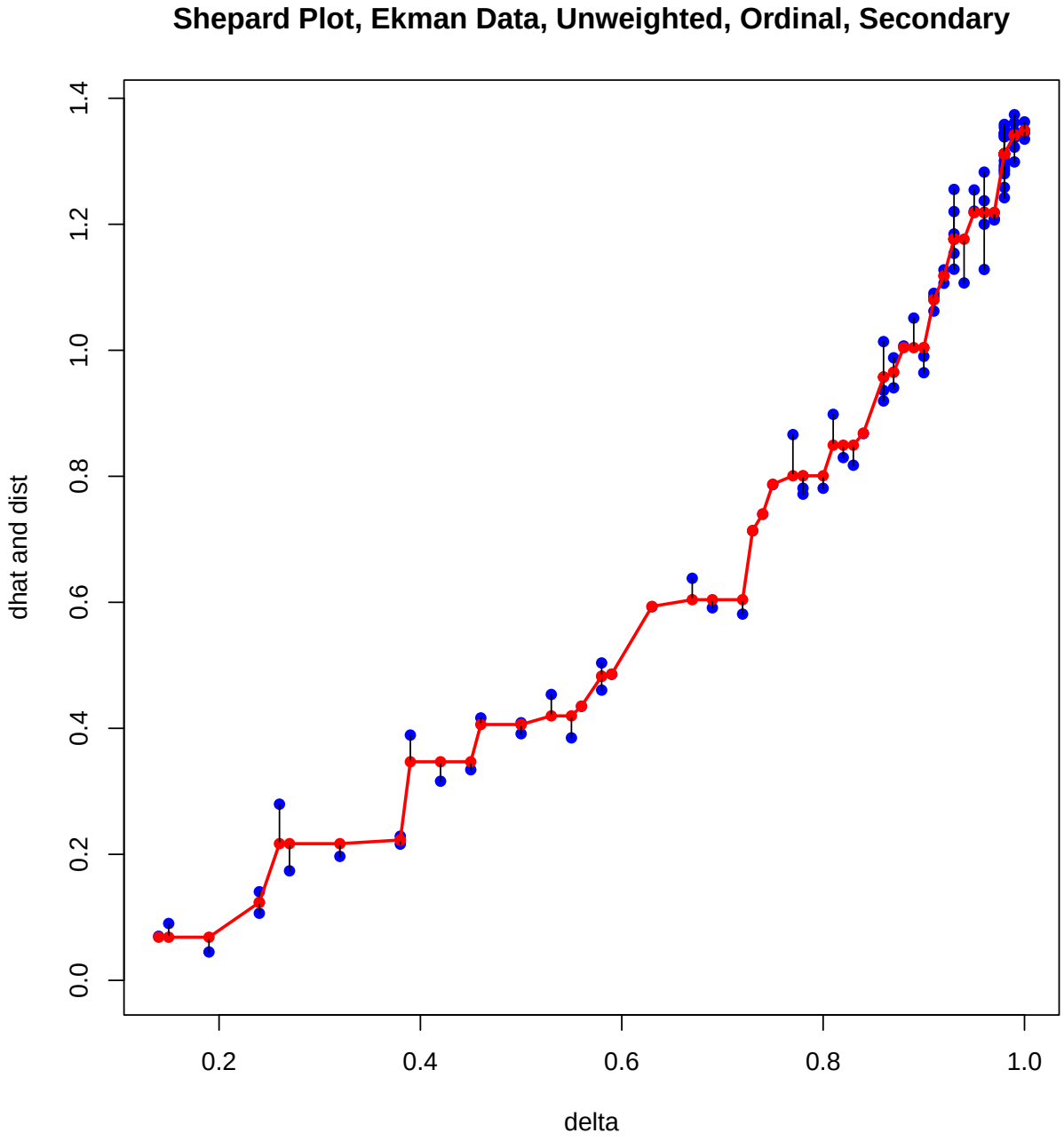
The next eight pages have Shepard plots and DistDhat plots for each of the four unweighted analyses. No plots are given for the weighted analyses, because they would illustrate essentially the same points. We made the plots large so they show some detail. Shepard plots have *fitlines* equal to TRUE, DistDhat plots have *fitlines* equal to TRUE for the Ekman data and FALSE for the Morse data. The sum of the squared lengths of the vertical fitlines in the Shepard plots is the stress. In the DistDhat plots the sum of squares of distances of the points to their orthogonal projections on the line is also the stress. Thus we can see from the plots where the largest residuals are, although the plot does not show which pair of points the fitlines correspond with. That information can easily be obtained from the numerical output.

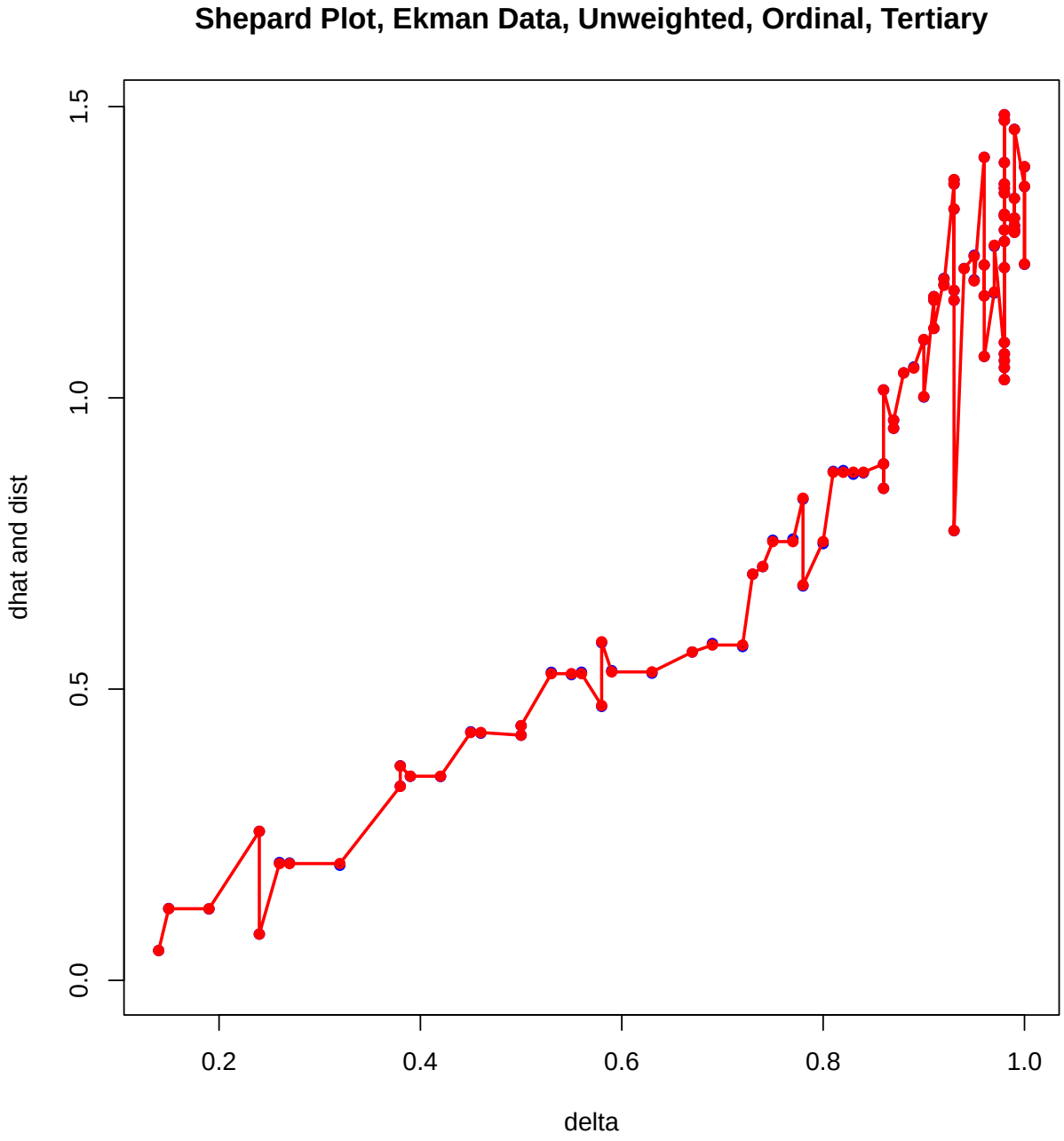
The Ekman example has an exceptionally good fit, even in the numerical case. Still,allowing for ordinal transformations gives a major improvement. Especially the DistDhat plots in the ordinal case show the different ways of handling tie blocks. The tertiary approach gives what is essentially a perfect fit, but the Shepard plot shows that in order to achieve this deviations from monotonicity are required.

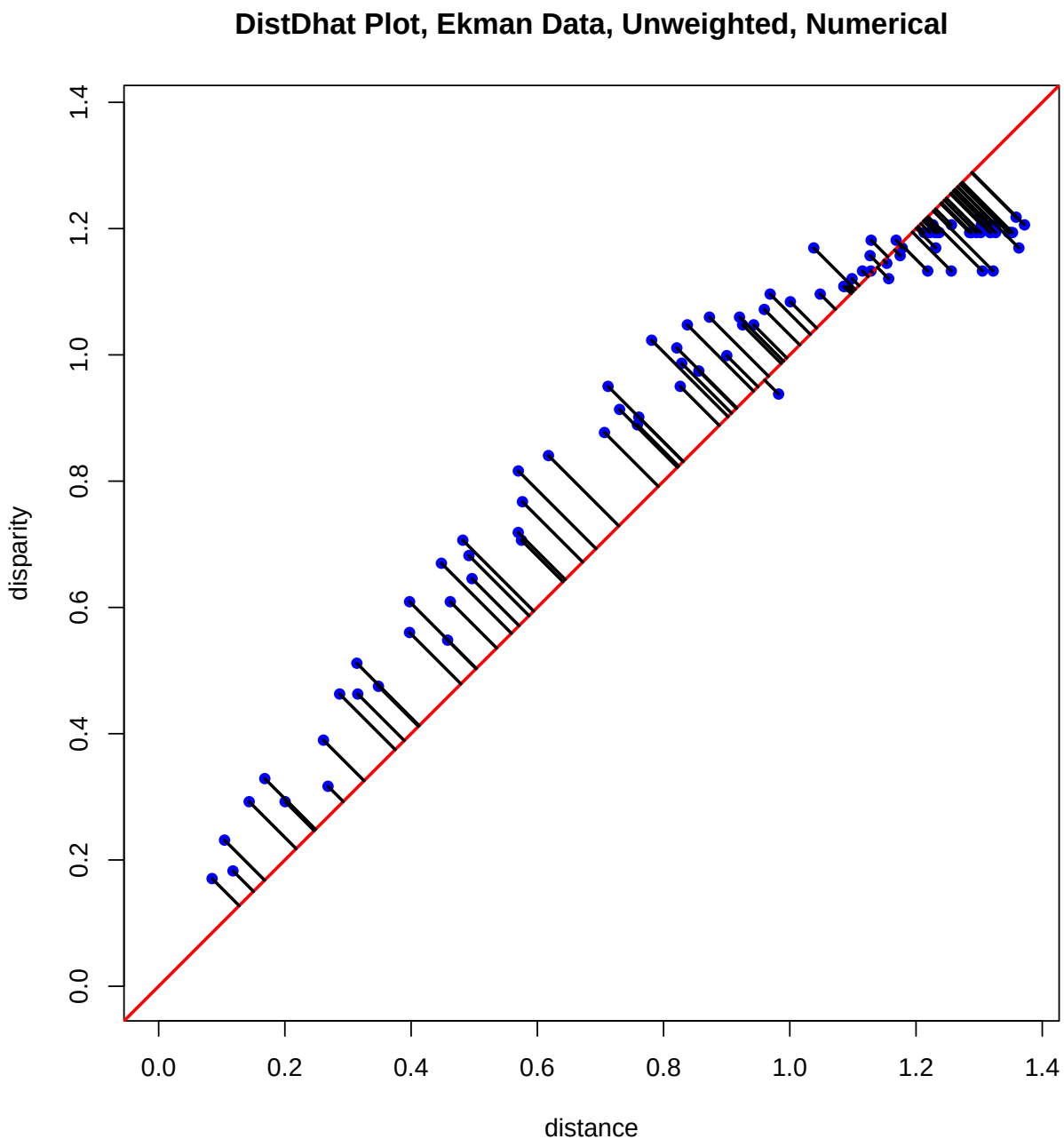
The Morse example has a bad numerical fit, and the improvements by the oidinal options are huge. There are no fitlines in the DistDhat plots, because that would mainly result in big black blobs. As in the Ekman example the tertiary approach gives close to perfect fit, at the cost of many deviations from monotonicity.

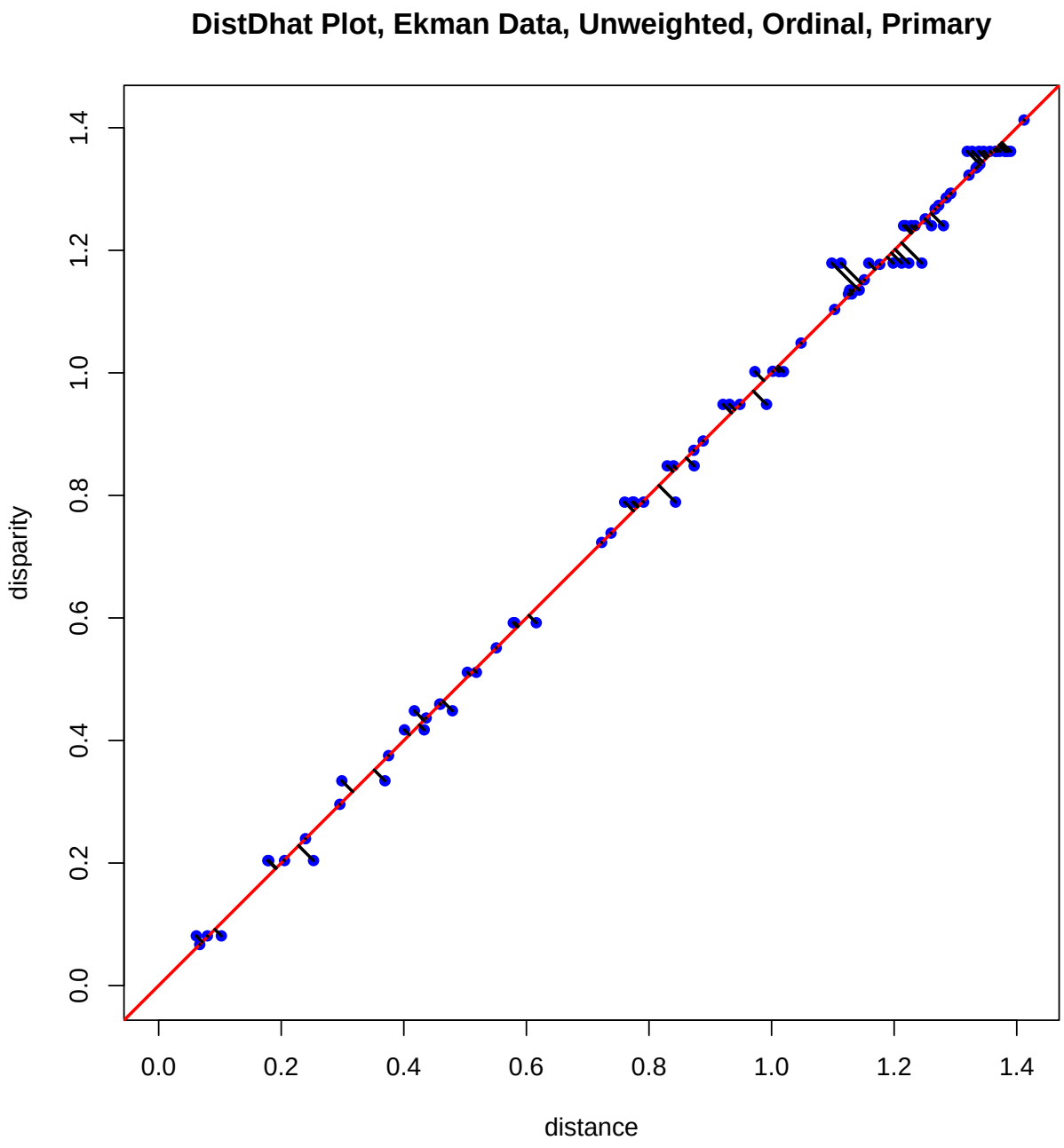

Shepard Plot, Ekman Data, Unweighted, Numerical
dhat and dist
delta
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0.2
0.4
0.6
0.8
1.0

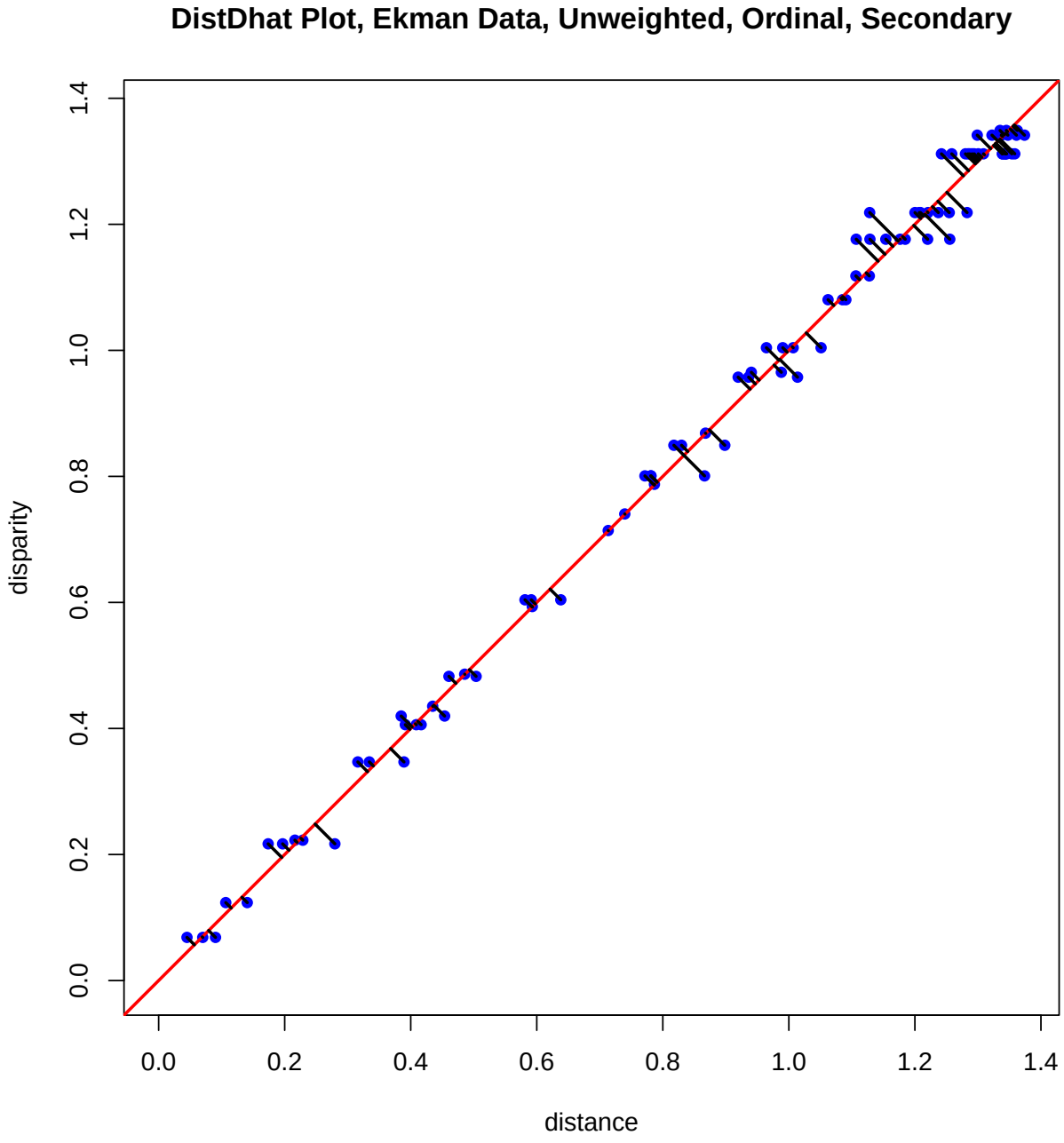

Shepard Plot, Ekman Data, Unweighted, Ordinal, Primary
dhat and dist
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0.2
0.4
0.6
0.8
1.0
delta

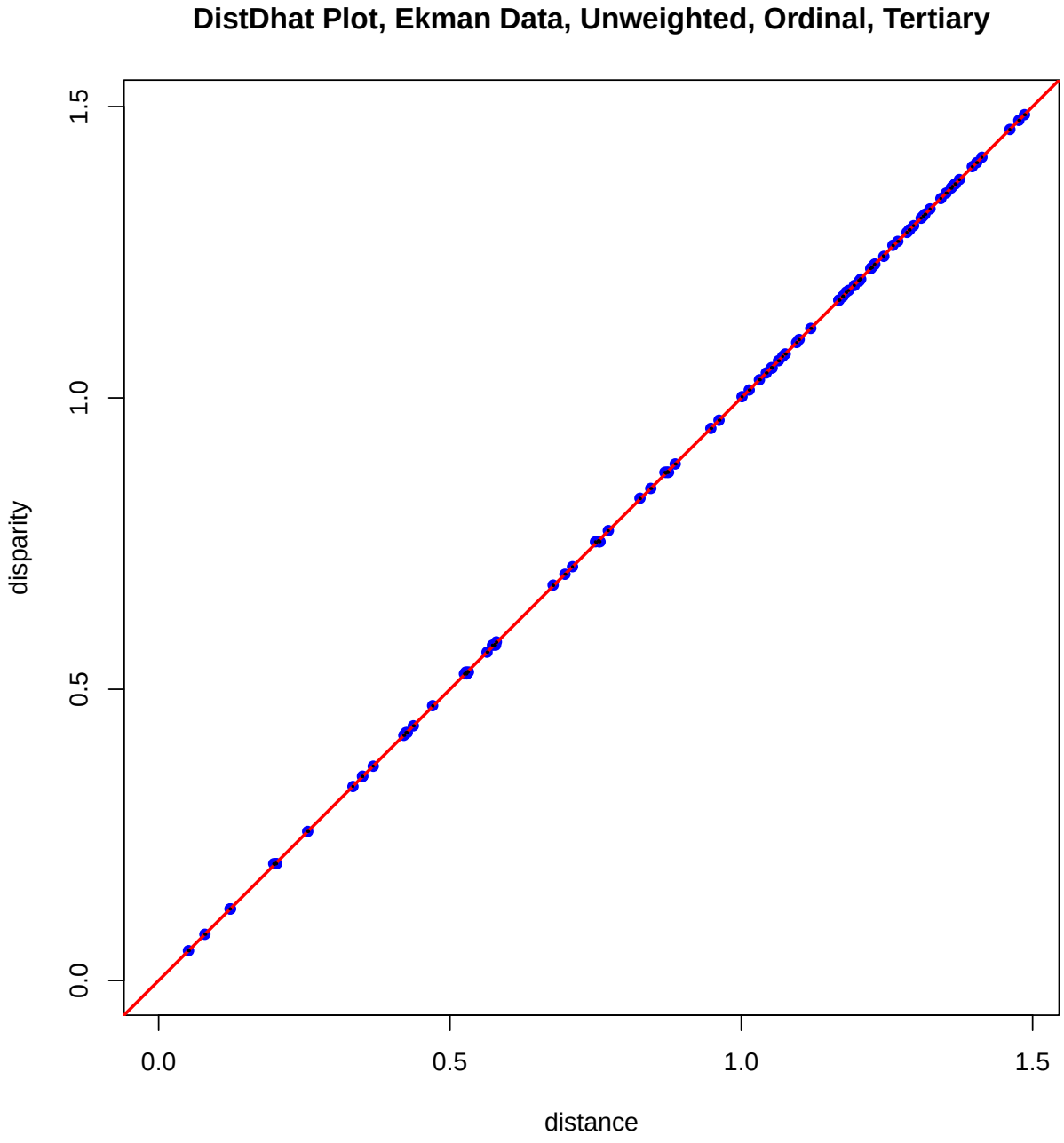

Shepard Plot, Ekman Data, Unweighted, Ordinal, Secondary
dhat and dist
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
0.2
0.4
0.6
0.8
1.0
delta

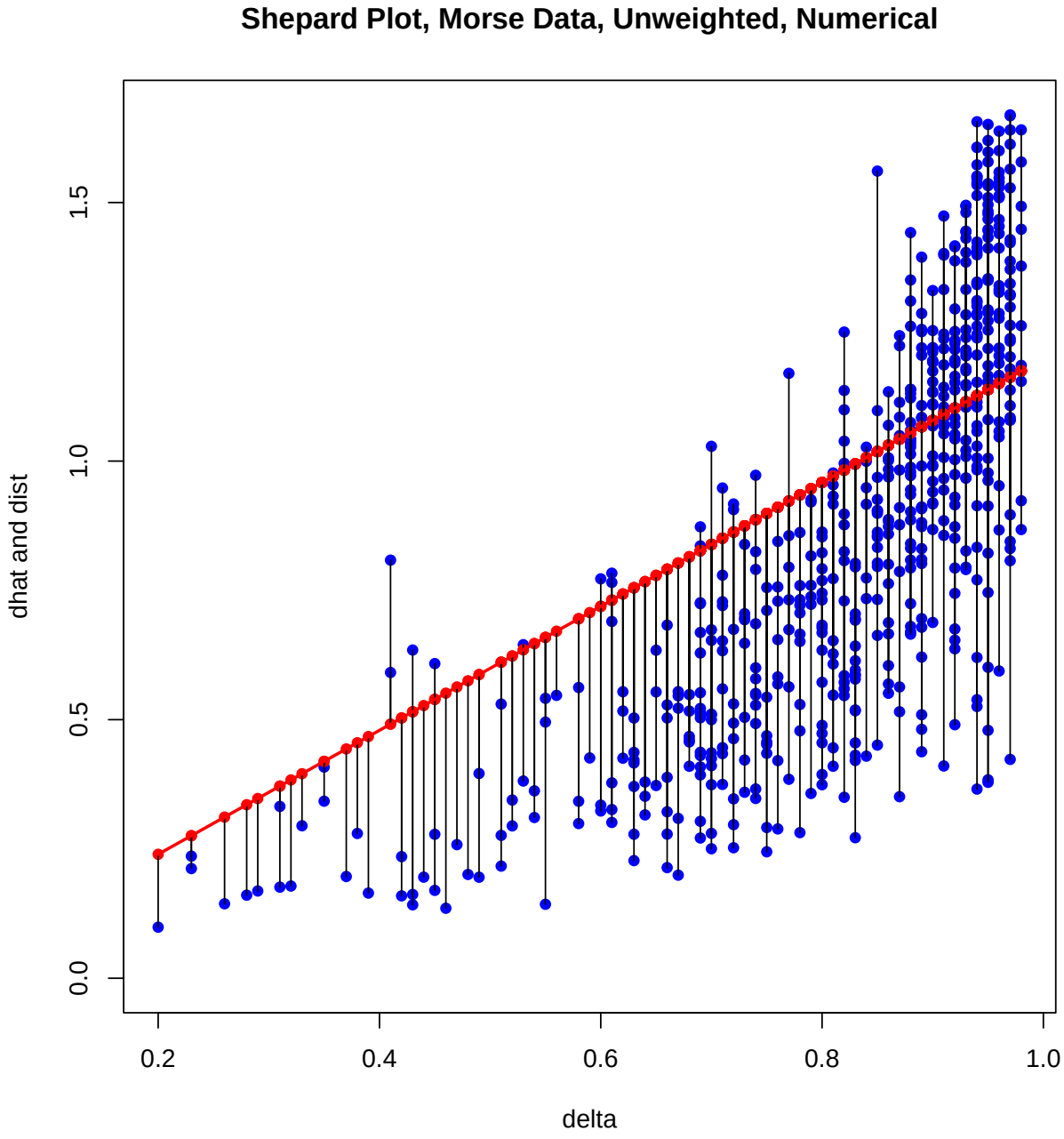

Shepard Plot, Ekman Data, Unweighted, Ordinal, Tertiary
dhat and dist
0.0
0.5
1.0
1.5
0.2
0.4
0.6
0.8
1.0
delta

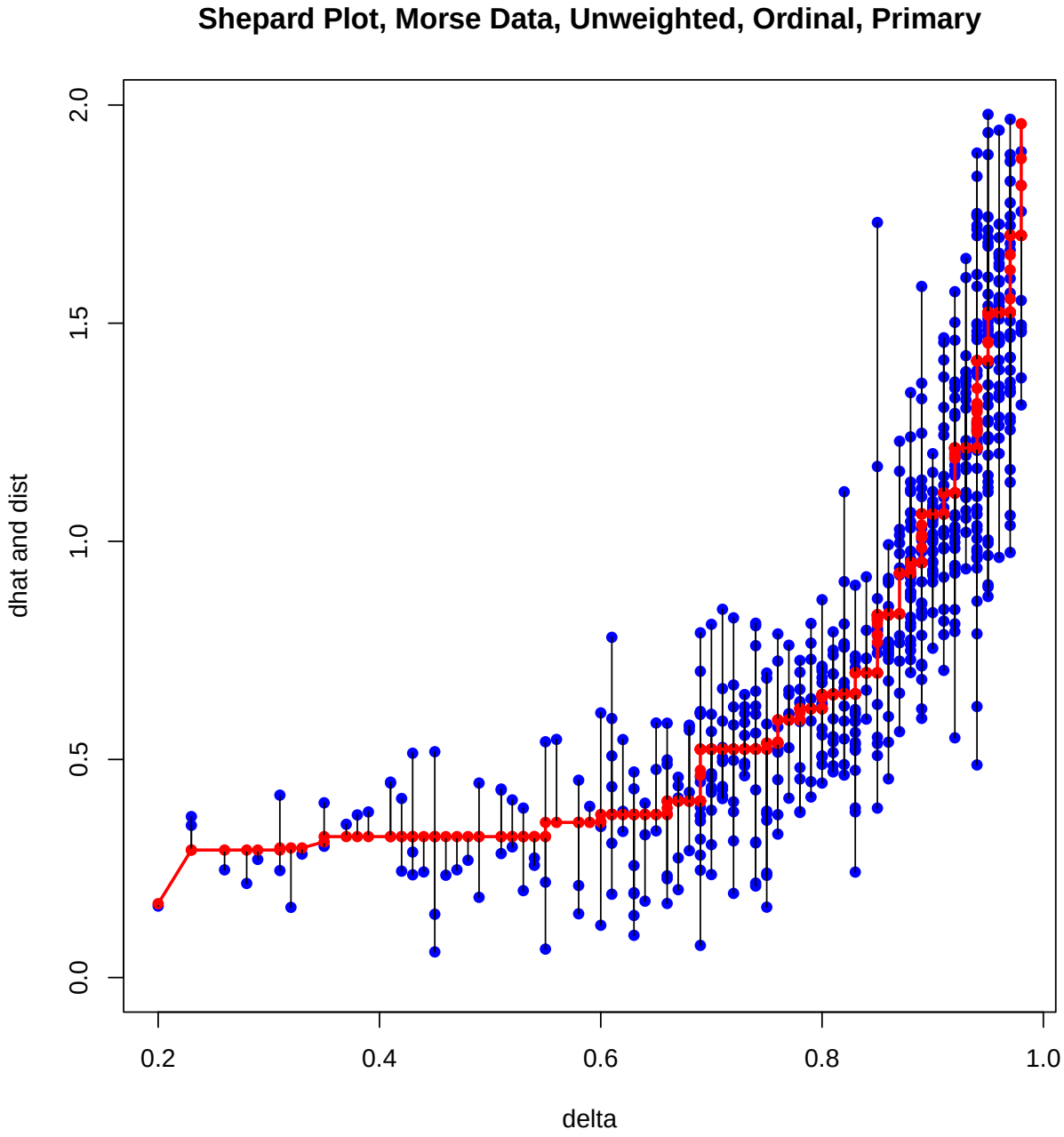

DistDhat Plot, Ekman Data, Unweighted, Numerical
disparity
distance
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4

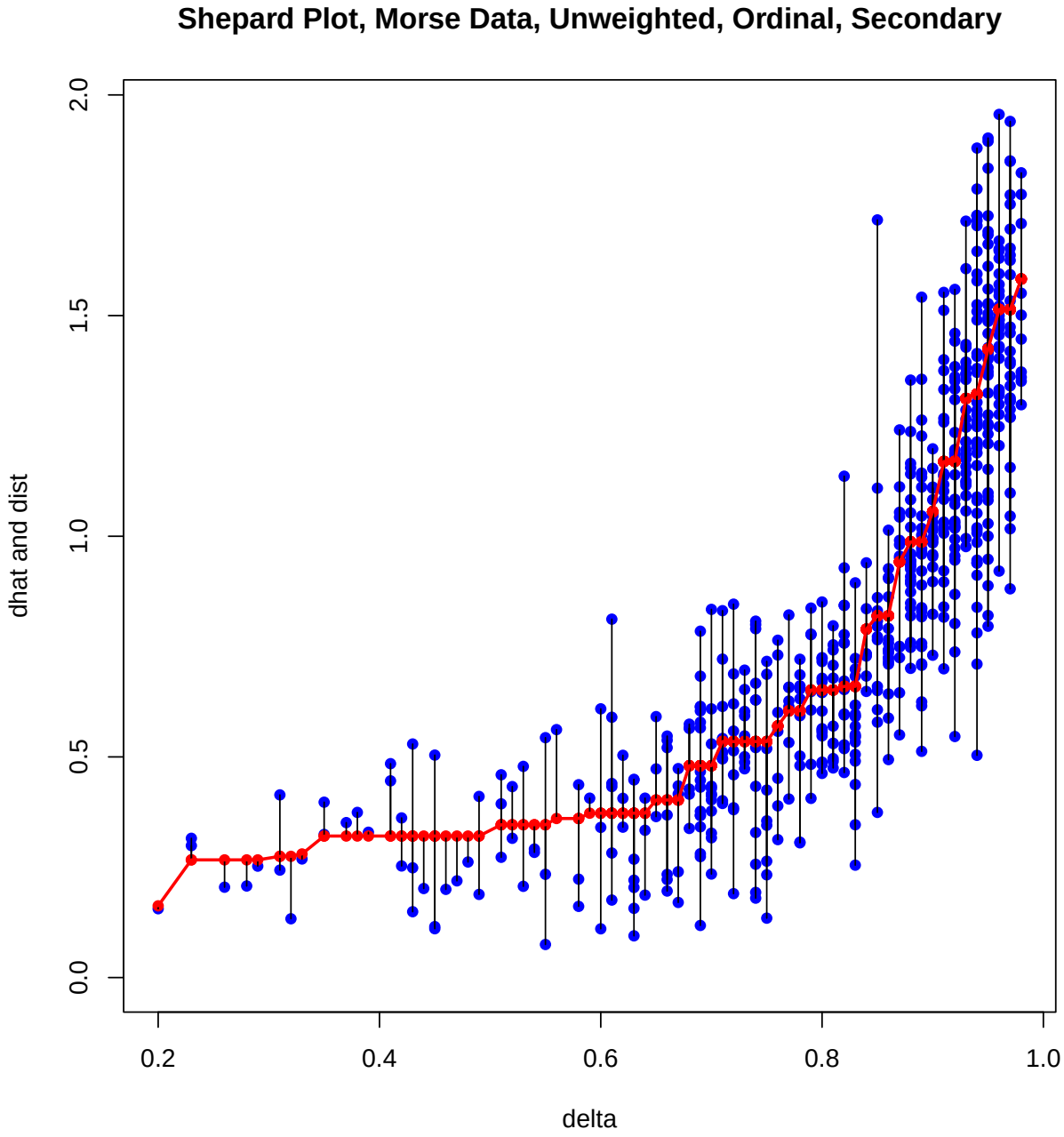

DistDhat Plot, Ekman Data, Unweighted, Ordinal, Primary
disparity
distance
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4

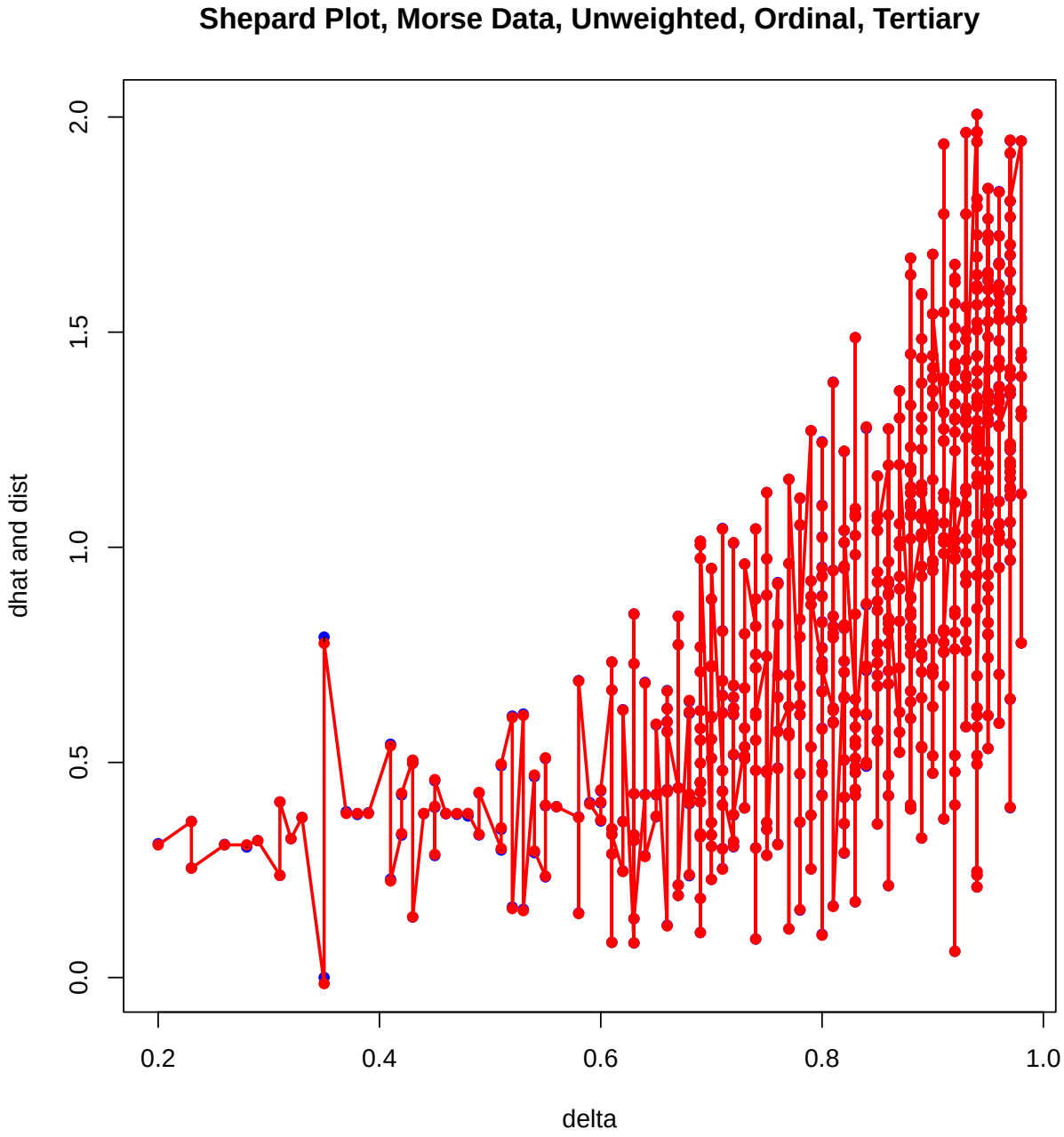

DistDhat Plot, Ekman Data, Unweighted, Ordinal, Secondary
disparity
distance
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4

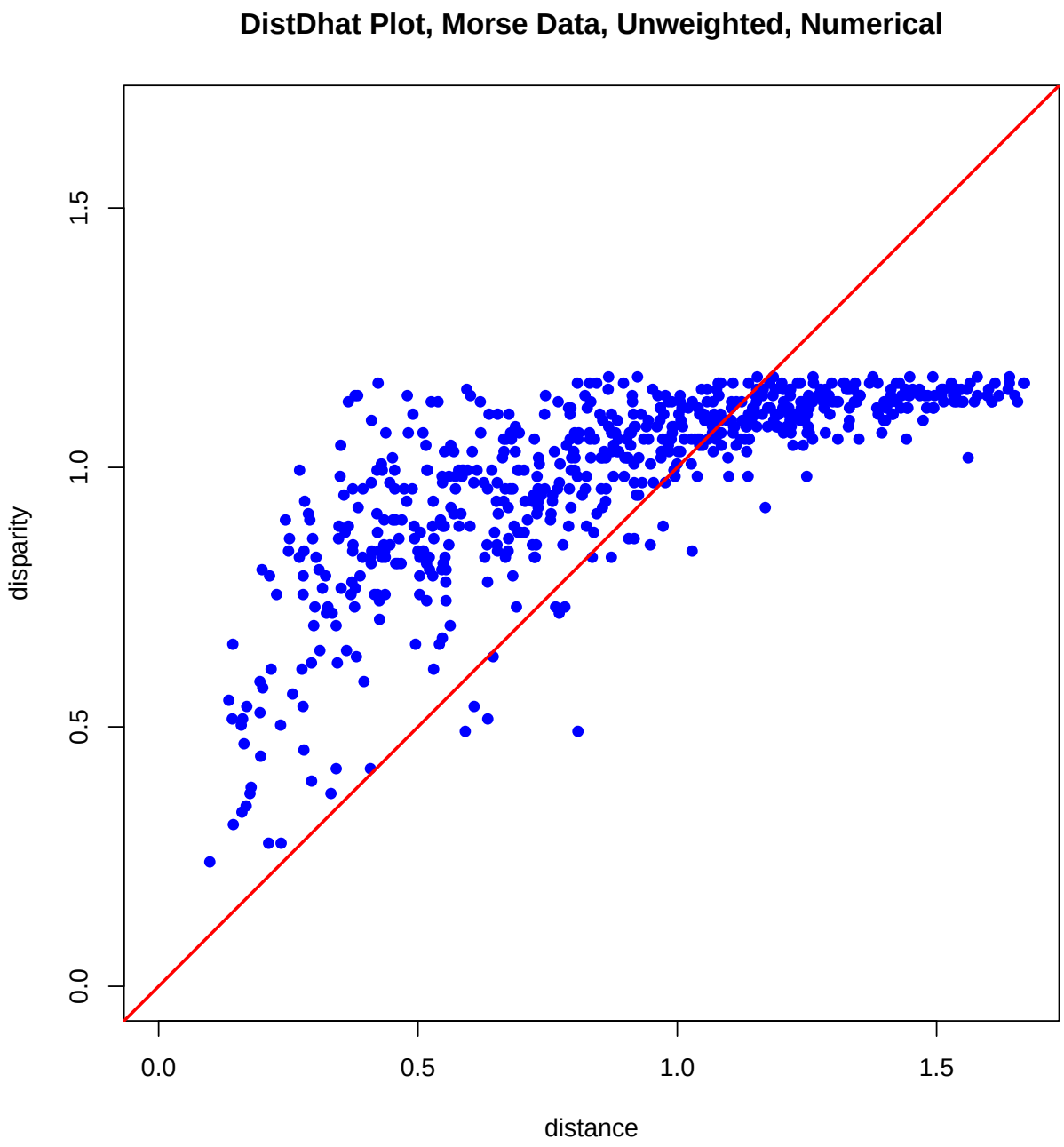

DistDhat Plot, Ekman Data, Unweighted, Ordinal, Tertiary
disparity
distance
0.0
0.5
1.0
1.5

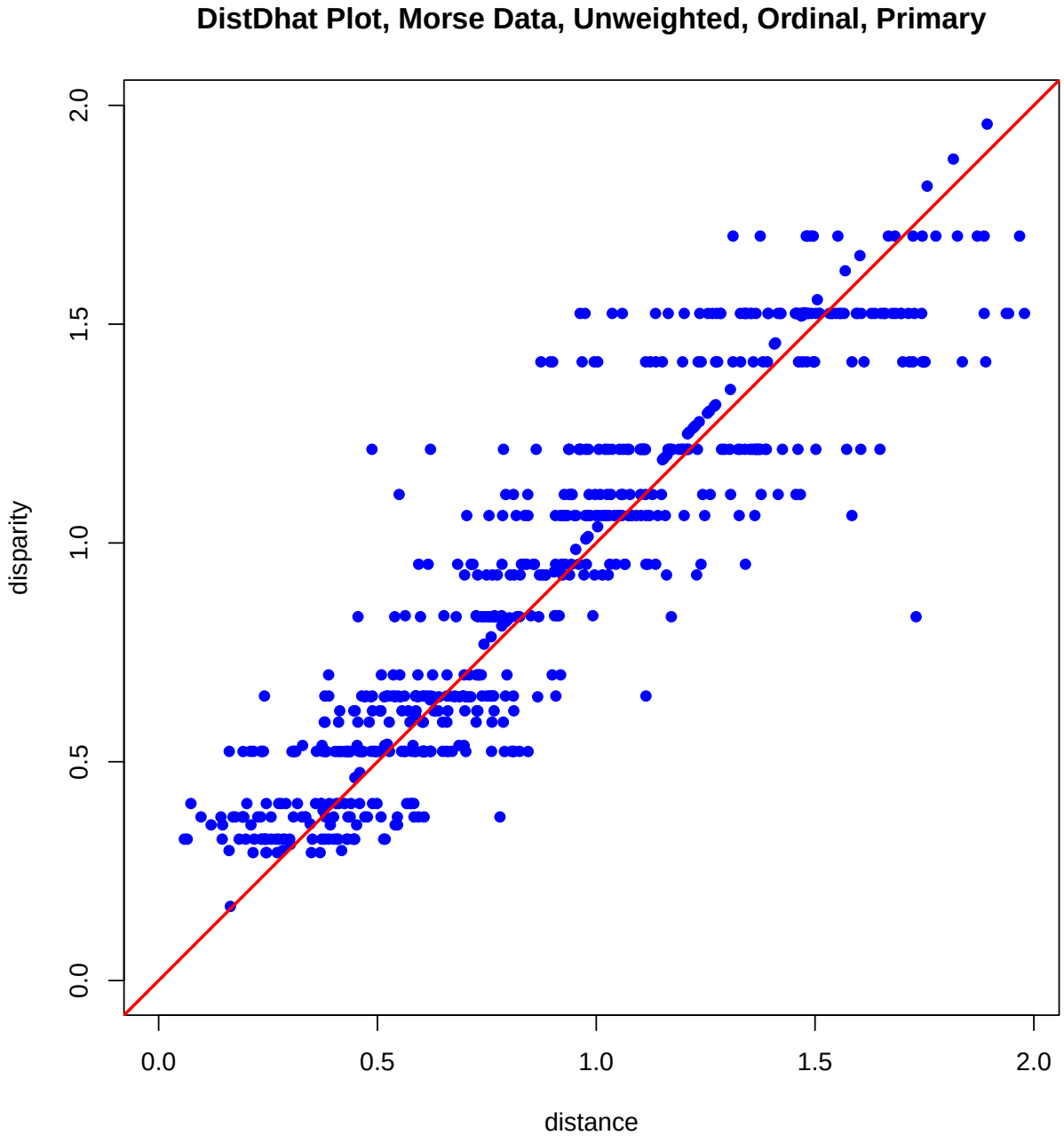

Shepard Plot, Morse Data, Unweighted, Numerical
dhat and dist
0.0
0.5
1.0
1.5
0.2
0.4
0.6
0.8
1.0
delta

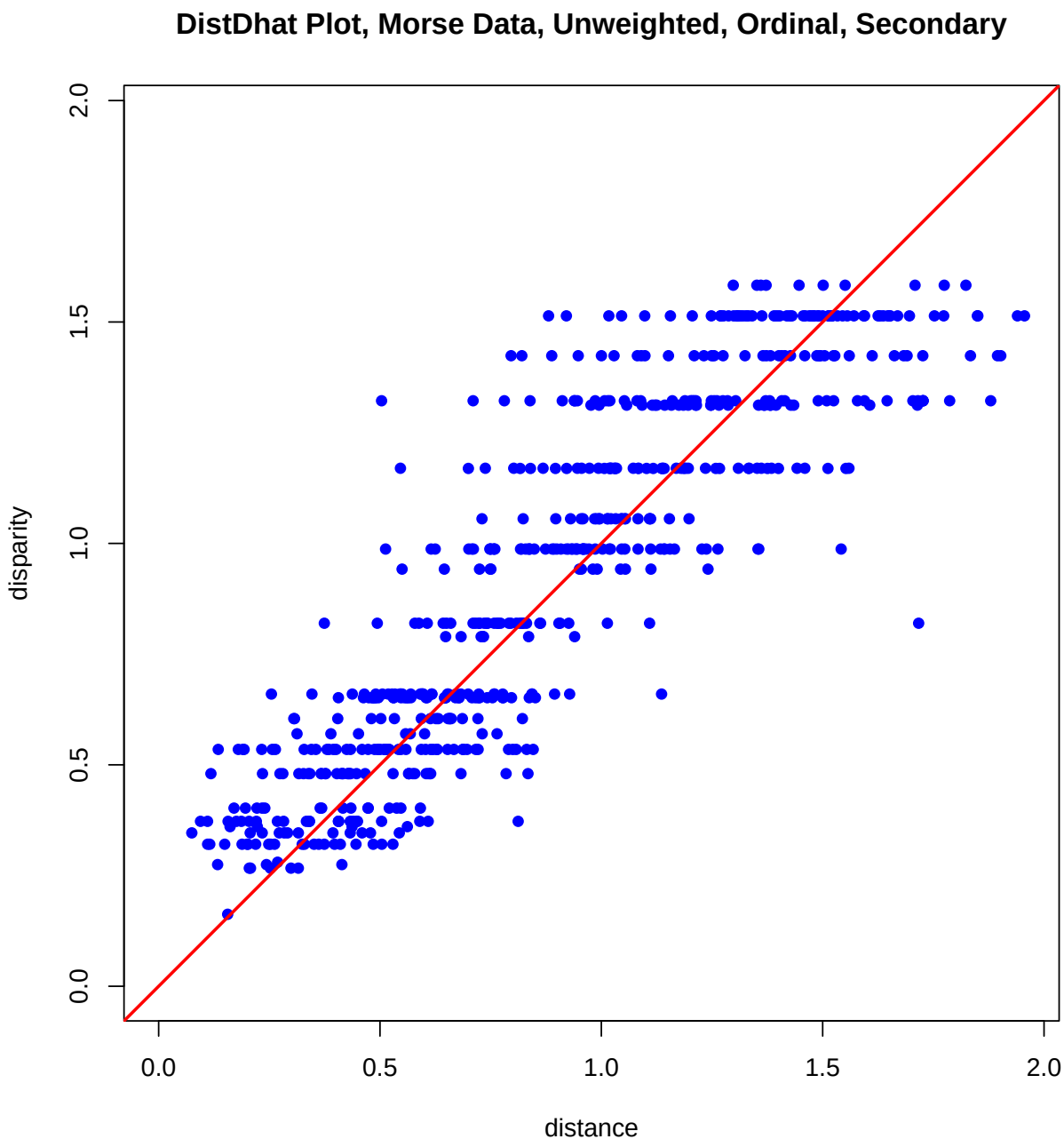

Shepard Plot, Morse Data, Unweighted, Ordinal, Primary
dhat and dist
0.0
0.5
1.0
1.5
2.0
0.2
0.4
0.6
0.8
1.0
delta

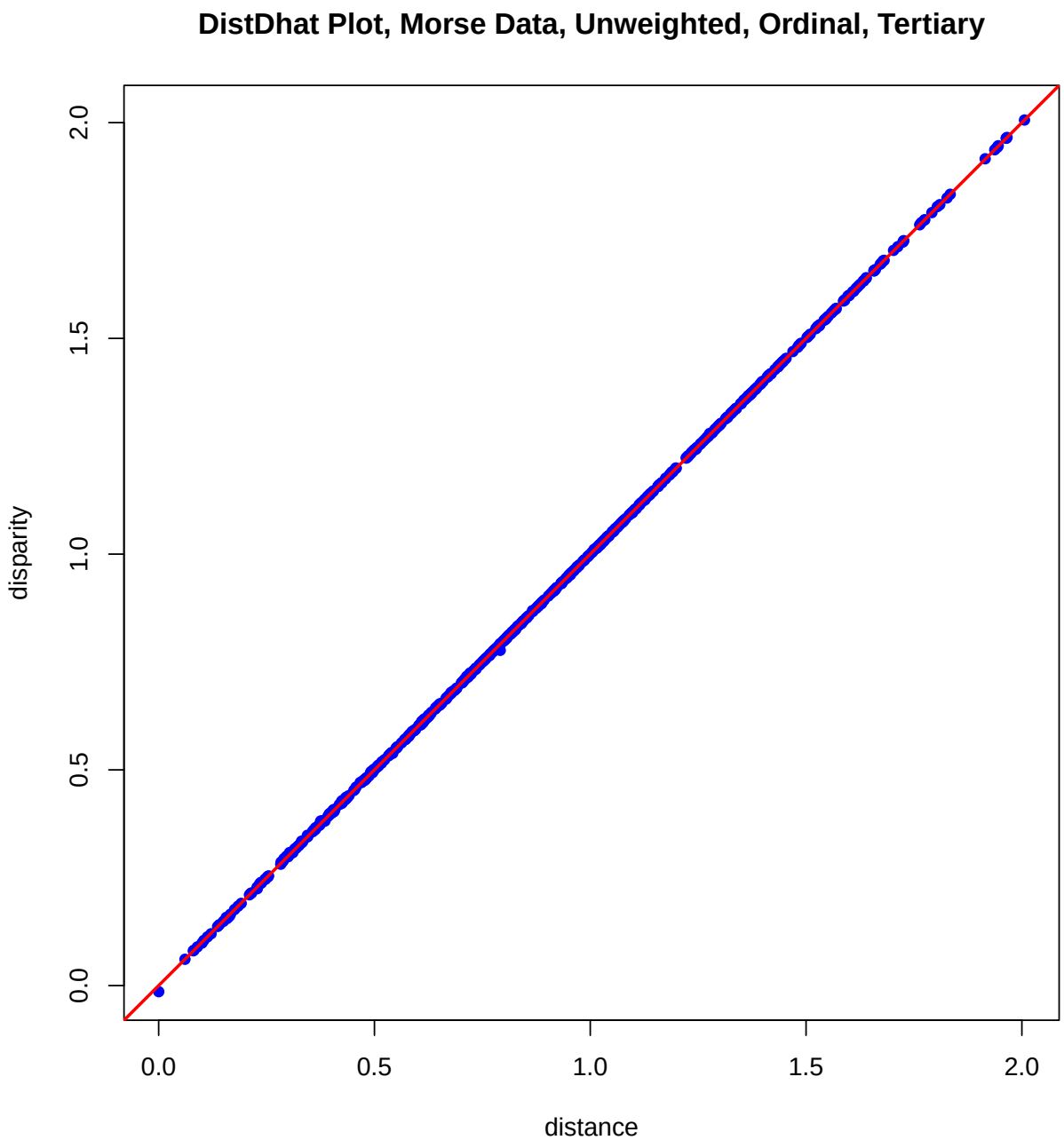

Shepard Plot, Morse Data, Unweighted, Ordinal, Secondary
dhat and dist
0.0
0.5
1.0
1.5
2.0
0.2
0.4
0.6
0.8
1.0
delta


Shepard Plot, Morse Data, Unweighted, Ordinal, Tertiary
dhat and dist
0.0
0.5
1.0
1.5
2.0
0.2
0.4
0.6
0.8
1.0
delta


DistDhat Plot, Morse Data, Unweighted, Numerical
disparity
distance
0.0
0.5
1.0
1.5
0.0
0.5
1.0
1.5


DistDhat Plot, Morse Data, Unweighted, Ordinal, Primary
disparity
distance
0.0
0.5
1.0
1.5
2.0


DistDhat Plot, Morse Data, Unweighted, Ordinal, Secondary
disparity
distance
0.0
0.5
1.0
1.5
2.0




```
source("smacofSSData/irisData.R")
xinit <- smacofTorgerson(irisData, 2)
h1 <- smacofSym(irisDist, ndim = 2, init = xinit, type = "ratio", eps = 1e-10, itmax =
h2 <- smacofSS(irisData, xinit = xinit, verbose = FALSE)
print(h1$stress)
```

```
[1] 0.03271481
```

```
print(h1$niter)
```

```
[1] 155
```

```
print(h2$stress)
```

```
[1] 0.001070259
```

```
print(h2$niter)
```

```
[1] 155
```

10 Code

10.1 R Code

10.1.1 smacofSS.R

```
source("smacofAuxiliaries.R")
source("smacofDataUtilities.R")
source("smacofPlots.R")

smacofSS <- function(theData,
                     ndim = 2,
                     xinit = NULL,
                     weighted = FALSE,
                     ordinal = FALSE,
                     itmax = 1000,
                     eps = 1e-10,
                     digits = 10,
                     width = 12,
                     verbose = TRUE) {
  if (!weighted && !ordinal) {
    source("smacofSSUR.R")
    return(
      smacofSSUR(
        theData = theData,
        ndim = ndim,
        xinit = xinit,
        itmax = itmax,
        eps = eps,
        digits = digits,
        width = width,
        verbose = verbose
      )
    )
  }
  if (weighted && !ordinal) {
    source("smacofSSWR.R")
    return(
      smacofSSWR(
```

```
      theData = theData,
      ndim = ndim,
      xinit = xinit,
      itmax = itmax,
      eps = eps,
      digits = digits,
      width = width,
      verbose = verbose
    )
  )
}
if (!weighted && ordinal) {
  source("smacofSSUO.R")
  return(
    smacofSSUO(
      theData = theData,
      ndim = ndim,
      xinit = xinit,
      ties = ordinal,
      itmax = itmax,
      eps = eps,
      digits = digits,
      width = width,
      verbose = verbose
    )
  )
}
if (weighted && ordinal) {
  source("smacofSSWO.R")
  return(
    smacofSSWO(
      theData = theData,
      ndim = ndim,
      xinit = xinit,
      ties = ordinal,
      itmax = itmax,
      eps = eps,
      digits = digits,
      width = width,
      verbose = verbose
```

```
      )
    )
  }
}
```

10.1.2 smacofSSUR.R

```
dyn.load("smacofSSUREngine.so")

source("smacofAuxiliaries.R")
source("smacofDataUtilities.R")
source("smacofPlots.R")

smacofSSUR <- function(theData,
                       ndim = 2,
                       xinit = NULL,
                       itmax = 1000,
                       eps = 1e-10,
                       digits = 10,
                       width = 15,
                       verbose = TRUE) {
  if (is.null(xinit)) {
    xinit <- smacofTorgerson(theData, 2)
  }
  xold <- xinit
  nobj <- theData$nobj
  ndat <- theData$ndat
  itel <- 1
  iind <- theData$iind
  jind <- theData$jind
  dhat <- theData$delta
  edis <- rep(0, ndat)
  for (k in 1:ndat) {
    i <- iind[k]
    j <- jind[k]
    edis[k] <- sqrt(sum((xold[i, ] - xold[j, ])^2))
  }
  dhat <- dhat * sqrt(ndat / sum(dhat^2))
```

```
  sdd <- sum(edis^2)
  sde <- sum(dhat * edis)
  lbd <- sde / sdd
  edis <- lbd * edis
  xold <- lbd * xold
  sold <- sum((dhat - edis)^2) / ndat
  snew <- 0.0
  xold <- as.vector(xold)
  xnew <- rep(0, nobj * ndim)
  h <- .C(
    "smacofSSUREngine",
    nobj = as.integer(nobj),
    ndim = as.integer(ndim),
    ndat = as.integer(ndat),
    itel = as.integer(itel),
    itmax = as.integer(itmax),
    digits = as.integer(digits),
    width = as.integer(width),
    verbose = as.integer(verbose),
    sold = as.double(sold),
    snew = as.double(snew),
    eps = as.double(eps),
    iind = as.integer(iind),
    jind = as.integer(jind),
    edis = as.double(edis),
    dhat = as.double(dhat),
    xold = as.double(xold),
    xnew = as.double(xnew)
  )
  result <- list(
    delta = theData$delta,
    dhat = h$dhat,
    confdist = h$edis,
    conf = matrix(h$xnew, nobj, ndim),
    weightmat = theData$weights,
    stress = h$snew,
    ndim = ndim,
    init = xinit,
    niter = h$itel,
    nobj = nobj,
```

```
    iind = h$iind,
    jind = h$jind,
    weighted = FALSE,
    ordinal = FALSE
  )
  class(result) <- c("smacofSSResult", "smacofSSURResult")
  return(result)
}
```

10.1.3 smacofSSWR.R

```
dyn.load("smacofSSWREngine.so")

source("smacofAuxiliaries.R")
source("smacofDataUtilities.R")
source("smacofPlots.R")

smacofSSWR <- function(theData,
                       ndim = 2,
                       xinit = NULL,
                       itmax = 1000,
                       eps = 1e-10,
                       digits = 10,
                       width = 15,
                       verbose = TRUE) {
  if (is.null(xinit)) {
    xinit <- smacofTorgerson(theData, 2)
  }
  xold <- xinit
  nobj <- theData$nobj
  ndat <- theData$ndat
  itel <- 1
  iind <- theData$iind
  jind <- theData$jind
  dhat <- theData$delta
  wght <- theData$weights
  wsum <- sum(wght)
  vinv <- makeMPInverseV(theData)
```

```
  edis <- rep(0, ndat)
  for (k in 1:ndat) {
    i <- iind[k]
    j <- jind[k]
    edis[k] <- sqrt(sum((xold[i, ] - xold[j, ])^2))
  }
  dhat <- dhat * sqrt(wsum / sum(wght * dhat^2))
  sdd <- sum(wght * edis^2)
  sde <- sum(wght * dhat * edis)
  lbd <- sde / sdd
  edis <- lbd * edis
  xold <- lbd * xold
  sold <- sum(wght * (dhat - edis)^2) / wsum
  snew <- 0.0
  xold <- as.vector(xold)
  xnew <- rep(0, nobj * ndim)
  h <- .C(
    "smacofSSWREngine",
    nobj = as.integer(nobj),
    ndim = as.integer(ndim),
    ndat = as.integer(ndat),
    itel = as.integer(itel),
    itmax = as.integer(itmax),
    digits = as.integer(digits),
    width = as.integer(width),
    verbose = as.integer(verbose),
    wsum = as.double(wsum),
    sold = as.double(sold),
    snew = as.double(snew),
    eps = as.double(eps),
    iind = as.integer(iind),
    jind = as.integer(jind),
    edis = as.double(edis),
    dhat = as.double(dhat),
    wght = as.double(wght),
    vinv = as.double(vinv),
    xold = as.double(xold),
    xnew = as.double(xnew)
  )
  result <- list(
```

```
    delta = theData$delta,
    dhat = h$dhat,
    confdist = h$edis,
    conf = matrix(h$xnew, nobj, ndim),
    weightmat = theData$weights,
    stress = h$snew,
    ndim = ndim,
    init = xinit,
    niter = h$itel,
    nobj = nobj,
    iind = h$iind,
    jind = h$jind,
    weighted = TRUE,
    ordinal = FALSE
  )
  class(result) <- c("smacofSSResult", "smacofSSWRResult")
  return(result)
}
```

10.1.4 smacofSSUO.R

```
dyn.load("smacofSSUOEngine.so")

source("smacofAuxiliaries.R")
source("smacofDataUtilities.R")
source("smacofPlots.R")

smacofSSUO <- function(theData,
                       ndim = 2,
                       xinit = NULL,
                       ties = 1,
                       itmax = 1000,
                       eps = 1e-10,
                       digits = 10,
                       width = 15,
                       verbose = TRUE) {
  if (is.null(xinit)) {
    xinit <- smacofTorgerson(theData, 2)
```

```
}
xold <- xinit
nobj <- theData$nobj
ndat <- theData$ndat
itel <- 1
iind <- theData$iind
jind <- theData$jind
dhat <- theData$delta
blks <- theData$blocks
edis <- rep(0, ndat)
for (k in 1:ndat) {
  i <- iind[k]
  j <- jind[k]
  edis[k] <- sqrt(sum((xold[i, ] - xold[j, ])^2))
}
dhat <- dhat * sqrt(ndat / sum(dhat^2))
sdd <- sum(edis^2)
sde <- sum(dhat * edis)
lbd <- sde / sdd
edis <- lbd * edis
xold <- lbd * xold
sold <- sum((dhat - edis)^2) / ndat
snew <- 0.0
xold <- as.vector(xold)
xnew <- rep(0, nobj * ndim)
h <- .C(
  "smacofSSUOEngine",
  nobj = as.integer(nobj),
  ndim = as.integer(ndim),
  ndat = as.integer(ndat),
  itel = as.integer(itel),
  ties = as.integer(ties),
  itmax = as.integer(itmax),
  digits = as.integer(digits),
  width = as.integer(width),
  verbose = as.integer(verbose),
  sold = as.double(sold),
  snew = as.double(snew),
  eps = as.double(eps),
  iind = as.integer(iind),
```

```
    jind = as.integer(jind),
    blks = as.integer(blks),
    edis = as.double(edis),
    dhat = as.double(dhat),
    xold = as.double(xold),
    xnew = as.double(xnew)
  )
  result <- list(
    delta = theData$delta,
    dhat = h$dhat,
    confdist = h$edis,
    conf = matrix(h$xnew, nobj, ndim),
    weightmat = theData$weights,
    stress = h$snew,
    ndim = ndim,
    init = xinit,
    niter = h$itel,
    nobj = nobj,
    iind = h$iind,
    jind = h$jind,
    weighted = FALSE,
    ordinal = TRUE
  )
  class(result) <- c("smacofSSResult", "smacofSSUOResult")
  return(result)
}
```

10.1.5 smacofSSWO.R

```
dyn.load("smacofSSWOEngine.so")

source("smacofAuxiliaries.R")
source("smacofDataUtilities.R")
source("smacofPlots.R")

smacofSSWO <- function(theData,
                       ndim = 2,
                       xinit = NULL,
```

```
                      ties = 1,
                      itmax = 1000,
                      eps = 1e-10,
                      digits = 10,
                      width = 15,
                      verbose = TRUE) {
if (is.null(xinit)) {
  xinit <- smacofTorgerson(theData, 2)
}
xold <- xinit
nobj <- theData$nobj
ndat <- theData$ndat
itel <- 1
iind <- theData$iind
jind <- theData$jind
dhat <- theData$delta
blks <- theData$blocks
wght <- theData$weights
wsum <- sum(wght)
vinv <- makeMPInverseV(theData)
edis <- rep(0, ndat)
for (k in 1:ndat) {
  i <- iind[k]
  j <- jind[k]
  edis[k] <- sqrt(sum((xold[i, ] - xold[j, ])^2))
}
dhat <- dhat * sqrt(wsum / sum(wght * dhat^2))
sdd <- sum(wght * edis^2)
sde <- sum(wght * dhat * edis)
lbd <- sde / sdd
edis <- lbd * edis
xold <- lbd * xold
sold <- sum(wght * (dhat - edis)^2) / wsum
snew <- 0.0
xold <- as.vector(xold)
xnew <- rep(0, nobj * ndim)
h <- .C(
  "smacofSSWOEngine",
  nobj = as.integer(nobj),
  ndim = as.integer(ndim),
```

```
    ndat = as.integer(ndat),
    itel = as.integer(itel),
    ties = as.integer(ties),
    itmax = as.integer(itmax),
    digits = as.integer(digits),
    width = as.integer(width),
    verbose = as.integer(verbose),
    wsum = as.double(wsum),
    sold = as.double(sold),
    snew = as.double(snew),
    eps = as.double(eps),
    iind = as.integer(iind),
    jind = as.integer(jind),
    blks = as.integer(blks),
    edis = as.double(edis),
    dhat = as.double(dhat),
    wght = as.double(wght),
    vinv = as.double(vinv),
    xold = as.double(xold),
    xnew = as.double(xnew)
  )
  result <- list(
    delta = theData$delta,
    dhat = h$dhat,
    confdist = h$edis,
    conf = matrix(h$xnew, nobj, ndim),
    weightmat = h$wght,
    stress = h$snew,
    ndim = ndim,
    init = xinit,
    niter = h$itel,
    nobj = nobj,
    iind = h$iind,
    jind = h$jind,
    weighted = TRUE,
    ordinal = TRUE
  )
  class(result) <- c("smacofSSResult", "smacofSSWOResult")
  return(result)
}
```

10.1.6 smacofAuxiliaries.R

```
library(RSpectra)

makeMPInverseV <- function(theData) {
  nobj <- theData$nobj
  ndat <- theData$ndat
  wght <- matrix(0, nobj, nobj)
  for (k in 1:ndat) {
    i <- theData$iind[k]
    j <- theData$jind[k]
    wght[i, j] <- wght[j, i] <- theData$weights[k]
  }
  vmat <- -wght
  diag(vmat) <- -rowSums(vmat)
  vinv <- solve(vmat + (1 / nobj)) - (1 / nobj)
  return(as.vector(as.dist(vinv)))
}

smacofTorgerson <- function(theData, ndim) {
  nobj <- theData$nobj
  ndat <- theData$ndat
  dmat <- matrix(0, nobj, nobj)
  mdel <- mean(theData$delta)^2
  dmat <- mdel * (1 - diag(nobj))
  for (k in 1:ndat) {
    i <- theData$iind[k]
    j <- theData$jind[k]
    dmat[i, j] <- dmat[j, i] <- theData$delta[k]
  }
  dmat <- dmat^2
  dr <- apply(dmat, 1, mean)
  dm <- mean(dmat)
  bmat <- -(dmat - outer(dr, dr, "+") + dm) / 2
  ev <- eigs_sym(bmat, ndim, which = "LA")
  evev <- diag(sqrt(pmax(0, ev$values)))
  x <- ev$vectors[, 1:ndim] %*% evev
  return(x)
}
```

```
smacofGuttman <- function(theData, ndim) {
  nobj <- theData$nobj
  ndat <- theData$ndat
  delta <- theData$delta
  wght <- theData$weights
  bmat <- matrix(0, nobj, nobj)
  for (k in 1:ndat) {
    i <- theData$iind[k]
    j <- theData$jind[k]
    bmat[i, j] <- bmat[j, i] <- wght[k] * delta[k]^2
  }
  bmat <- -bmat
  diag(bmat) <- -rowSums(bmat)
  ev <- eigs_sym(bmat, ndim, which = "LA")
  evev <- diag(sqrt(pmax(0, ev$values)))
  x <- ev$vectors[, 1:ndim] %*% evev
}

smacofRandomConfiguration <- function(theData, ndim = 2) {
  nobj <- theData$nobj
  x <- matrix(rnorm(nobj * ndim), nobj, ndim)
  return(columnCenter(x))
}

columnCenter <- function(x) {
  apply(x, 2, function(x) x - mean(x))
}

matrixPrint <- function(x,
                        digits = 6,
                        width = 8,
                        format = "f",
                        flag = "+") {
  print(noquote(
    formatC(
      x,
      digits = digits,
      width = width,
      format = format,
      flag = flag
```

```
    )
  ))
}
```

10.1.7 smacofDataUtilities.R

```
makeMDSData <- function(delta, weights = NULL) {
  nobj <- attr(delta, "Size")
  if (is.null(weights)) {
    weights <- as.dist(1 - diag(nobj))
  }
  theData <- NULL
  k <- 1
  for (j in 1:(nobj - 1)) {
    for (i in (j + 1):nobj) {
      if ((weights[k] > 0) &&
          (!is.na(weights[k])) && (!is.na(delta[k]))) {
        theData <- rbind(theData, c(i, j, delta[k], 0, weights[k]))
      }
      k <- k + 1
    }
  }
  colnames(theData) <- c("i", "j", "delta", "blocks", "weights")
  ndat <- nrow(theData)
  theData <- theData[order(theData[, 3]), ]
  dvec <- theData[, 3]
  k <- 1
  repeat {
    m <- length(which(dvec == dvec[k]))
    theData[k, 4] <- m
    k <- k + m
    if (k > ndat) {
      break
    }
  }
  result <- list(
    iind = theData[, 1],
    jind = theData[, 2],
```

```
    delta = theData[, 3],
    blocks = theData[, 4],
    weights = theData[, 5],
    nobj = nobj,
    ndat = ndat
  )
  class(result) <- "smacofSSData"
  return(result)
}

fromMDSData <- function(theData) {
  ndat <-theData$ndat
  nobj <- theData$nobj
  delta <- matrix(0, nobj, nobj)
  weights <- matrix(0, nobj, nobj)
  for (k in 1:ndat) {
    i <- theData$iind[k]
    j <- theData$jind[k]
    delta[i, j] <- delta[j, i] <- theData$delta[k]
    weights[i, j] <- weights[j, i] <- theData$weights[k]
  }
  return(list(delta = as.dist(delta), weights = as.dist(weights)))
}
```

10.1.8 smacofPlots.R

```
smacofShepardPlot <-
  function(h,
           main = "ShepardPlot",
           fitlines = TRUE,
           colline = "RED",
           colpoint = "BLUE",
           resolution = 100,
           lwd = 2,
           cex = 1,
           pch = 16) {
    maxDelta <- max(h$delta)
    minDelta <- min(h$delta)
```

```
    x <- h$delta
    y <- h$dhat
    z <- h$confdist
    plot(
      rbind(cbind(x, z), cbind(x, y)),
      xlim = c(minDelta, maxDelta),
      ylim = c(0, max(c(h$confdist, h$dhat))),
      xlab = "delta",
      ylab = "dhat and dist",
      main = main,
      type = "n"
    )
    points(x,
           z,
           col = colpoint,
           cex = cex,
           pch = pch)
    points(x,
           y,
           col = colline,
           cex = cex,
           pch = pch)
    if (fitlines) {
      for (i in 1:length(x)) {
        lines(x = c(x[i], x[i]), y = c(y[i], z[i]))
      }
    }
    lines(x,
          y,
          type = "l",
          lwd = lwd,
          col = colline)
  }

smacofConfigurationPlot <-
  function(h,
           main = "ConfigurationPlot",
           labels = NULL,
           dim1 = 1,
           dim2 = 2,
```

```
            pch = 16,
            col = "RED",
            cex = 1.0) {
    xnew <- h$conf
    if (is.null(labels)) {
      plot(
        xnew[, c(dim1, dim2)],
        xlab = paste("dimension", dim1),
        ylab = paste("dimension", dim2),
        main = main,
        pch = pch,
        col = col,
        cex = cex
      )
    }
    else {
      plot(
        xnew[, c(dim1, dim2)],
        xlab = paste("dimension", dim1),
        ylab = paste("dimension", dim2),
        main = main,
        type = "n"
      )
      text(xnew[, c(dim1, dim2)], labels, col = col, cex = cex)
    }
  }

smacofDistDhatPlot <- function(h,
                               fitlines = TRUE,
                               colline = "RED",
                               colpoint = "BLUE",
                               main = "Dist-Dhat Plot",
                               cex = 1.0,
                               lwd = 2,
                               pch = 16) {
  uppe <- max(c(h$confdist, h$dhat))
  plot(
    h$confdist,
    h$dhat,
    xlab = "distance",
```

```
    ylab = "disparity",
    xlim = c(0, uppe),
    ylim = c(0, uppe),
    main = main,
    cex = cex,
    pch = pch,
    col = colpoint
  )
  abline(0, 1, col = colline, lwd = lwd)
  if (fitlines) {
    m <- length(h$confdist)
    for (i in 1:m) {
      x <- h$confdist[i]
      y <- h$dhat[i]
      z <- (x + y) / 2
      a <- matrix(c(x, z, y, z), 2, 2)
      lines(a, lwd = lwd)
    }
  }
}
```

10.1.9 smacofElegant.R

```
library(RSpectra)

smacofElegant <- function(theData,
                          ndim = 2,
                          itmax = 1000,
                          eps = 1e-10,
                          verbose = TRUE) {
  nobj <- theData$nobj
  ndat <- theData$ndat
  iind <- theData$iind
  jind <- theData$jind
  wght <- theData$weights
  wsum <- sum(wght)
  delta <- (theData$delta)^2
  delta <- delta * sqrt(wsum / sum(wght * delta^2))
```

```
theData$delta <- delta
cinit <- smacofDoubleCenter(theData)
lbda <- smacofBound(theData)
evev <- eigs_sym(cinit, ndim)
cinit <- tcrossprod(evev$vectors %*% diag(sqrt(evev$values)))
dold <- rep(0, ndat)
for (k in 1:ndat) {
  i <- iind[k]
  j <- jind[k]
  dold[k] = cinit[i, i] + cinit[j, j] - 2 * cinit[i, j]
}
sold <- sum(wght * (delta - dold)^2) / wsum
itel <- 1
cold <- cinit
repeat {
  cmaj <- matrix(0, nobj, nobj)
  for (k in 1:ndat) {
    i <- iind[k]
    j <- jind[k]
    cmaj[i, j] <- cmaj[j, i] <- wght[k] * (delta[k] - dold[k])
  }
  cmaj <- -cmaj
  diag(cmaj) <- -rowSums(cmaj)
  cmaj <- cold + cmaj / lbda
  evev <- eigs_sym(cmaj, ndim)
  xvev <- evev$vectors %*% diag(sqrt(evev$values))
  cnew <- tcrossprod(xvev)
  dnew <- rep(0, ndat)
  for (k in 1:ndat) {
    i <- iind[k]
    j <- jind[k]
    dnew[k] = cnew[i, i] + cnew[j, j] - 2 * cnew[i, j]
  }
  snew <- sum(wght * (delta - dnew)^2) / wsum
  if (verbose) {
    cat(
      "itel ",
      formatC(itel, format = "d"),
      "sold ",
      formatC(sold, digits = 10, format = "f"),
```

```
        "snew ",
        formatC(snew, digits = 10, format = "f"),
        "\n"
      )
    }
    if ((itel == itmax) || ((sold - snew) < eps)) {
      break
    }
    itel <- itel + 1
    dold <- dnew
    sold <- snew
    cold <- cnew
  }
  return(xvev)
}

smacofDoubleCenter <- function(theData) {
  nobj <- theData$nobj
  ndat <- theData$ndat
  iind <- theData$iind
  jind <- theData$jind
  delta <- (theData$delta)^2
  cini <- matrix(0, nobj, nobj)
  for (k in 1:ndat) {
    i <- iind[k]
    j <- jind[k]
    cini[i, j] <- cini[j, i] <- delta[k]
  }
  rc <- apply(cini, 1, mean)
  rm <- mean(cini)
  cini <- -(cini - outer(rc, rc, "+") + rm) /  2
  return(cini)
}

smacofBound <- function(theData, quick = FALSE) {
  ndat <- theData$ndat
  nobj <- theData$nobj
  iind <- theData$iind
  jind <- theData$jind
  wght <- theData$weights
```

```
  if (quick) {
    return(2 * nobj * max(wght))
  }
  asum <- 4 * diag(ndat)
  for (k in 1:(ndat - 1)) {
    kind <- c(iind[k], jind[k])
    for (l in (k + 1):ndat) {
      lind <- c(iind[l], jind[l])
      asum[k, l] <- asum[l, k] <- sum(outer(kind, lind, "=="))
    }
  }
  asum <- asum * sqrt(outer(wght, wght))
  return(eigs_sym(asum, 1)$values)
}
```

10.2 C Code

10.3 smacofSSUREngine.c

```
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SQUARE(x) ((x) * (x))

void smacofSSUREngine(int *nobj, int *ndim, int *ndat, int *itel, int *itmax,
                      int *digits, int *width, bool *verbose, double *sold,
                      double *snew, double *eps, int *iind, int *jind,
                      double *edis, double *dhat, double *xold, double *xnew) {
    int Ndat = *ndat, Nobj = *nobj, Ndim = *ndim;
    while (true) {
        for (int k = 0; k < Nobj * Ndim; k++) {
            xnew[k] = 0.0;
        }
        for (int k = 0; k < Ndat; k++) {
            if (edis[k] == 0.0) {
```

```
                continue;
            }
            int is = iind[k] - 1, js = jind[k] - 1;
            double elem = dhat[k] / edis[k];
            for (int s = 0; s < Ndim; s++) {
                double add = elem * (xold[is] - xold[js]);
                xnew[is] += add;
                xnew[js] -= add;
                is += Nobj;
                js += Nobj;
            }
        }
        for (int i = 0; i < Nobj; i++) {
            int is = i;
            for (int s = 0; s < Ndim; s++) {
                xnew[is] = xnew[is] / (double)Nobj;
                is += Nobj;
            }
        }
        for (int k = 0; k < Ndat; k++) {
            int is = iind[k] - 1, js = jind[k] - 1;
            double sum = 0.0;
            for (int s = 0; s < Ndim; s++) {
                sum += SQUARE(xnew[is] - xnew[js]);
                edis[k] = sqrt(sum);
                is += Nobj;
                js += Nobj;
            }
        }
        *snew = 0.0;
        for (int k = 0; k < Ndat; k++) {
            *snew += SQUARE(dhat[k] - edis[k]);
        }
        *snew /= ((double) Ndat);
        if (*verbose) {
            printf("itel %4d sold %*.*f snew %*.*f\n", *itel, *width, *digits,
                   *sold, *width, *digits, *snew);
        }
        if ((*itel == *itmax) || ((*sold - *snew) < *eps)) {
            break;
```

```
        }
        xold = memcpy(xold, xnew, (size_t) Nobj * Ndim * sizeof(double));
        *sold = *snew;
        *itel += 1;
    }
}
```

10.4 smacofSSWREngine.c

```
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SQUARE(x) ((x) * (x))

void smacofSSWREngine(int *nobj, int *ndim, int *ndat, int *itel, int *itmax,
                      int *digits, int *width, bool *verbose, double *wsum, double *so
                      double *snew, double *eps, int *iind, int *jind,
                      double *edis, double *dhat, double *wght, double *vinv,
                      double *xold, double *xnew) {
    int Ndat = *ndat, Nobj = *nobj, Ndim = *ndim;
    while (true) {
        double *xtmp = (double *)calloc(Nobj * Ndim, sizeof(double));
        for (int k = 0; k < Ndat; k++) {
            if (edis[k] == 0.0) {
                continue;
            }
            int is = iind[k] - 1, js = jind[k] - 1;
            double elem = wght[k] * dhat[k] / edis[k];
            for (int s = 0; s < Ndim; s++) {
                double add = elem * (xold[is] - xold[js]);
                xtmp[is] += add;
                xtmp[js] -= add;
                is += Nobj;
                js += Nobj;
            }
```

```
        }
        for (int k = 0; k < Nobj * Ndim; k++) {
            xnew[k] = 0.0;
        }
        int k = 0;
        for (int j = 0; j < Nobj - 1; j++) {
            for (int i = j + 1; i < Nobj; i++) {
                double elem = vinv[k];
                int is = i, js = j;
                for (int s = 0; s < Ndim; s++) {
                    double add = elem * (xtmp[is] - xtmp[js]);
                    xnew[is] += add;
                    xnew[js] -= add;
                    is += Nobj;
                    js += Nobj;
                }
                k++;
            }
        }
        free(xtmp);
        for (int k = 0; k < Ndat; k++) {
            int is = iind[k] - 1, js = jind[k] - 1;
            double sum = 0.0;
            for (int s = 0; s < Ndim; s++) {
                sum += SQUARE(xnew[is] - xnew[js]);
                is += Nobj;
                js += Nobj;
            }
            edis[k] = sqrt(sum);
        }
        *snew = 0.0;
        for (int k = 0; k < Ndat; k++) {
            *snew += wght[k] * SQUARE(dhat[k] - edis[k]);
        }
        *snew /= *wsum;
        if (*verbose) {
            printf("itel %4d sold %*.*f snew %*.*f\n", *itel, *width, *digits,
                   *sold, *width, *digits, *snew);
        }
        if ((*itel == *itmax) || ((*sold - *snew) < *eps)) {
```

```
            break;
        }
        xold = memcpy(xold, xnew, (size_t) Nobj * Ndim * sizeof(double));
        *sold = *snew;
        *itel += 1;
    }
}
```

10.5 smacofSSUOEngine.c

```
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SQUARE(x) ((x) * (x))

void primaryApproach(int *, int *, double *, double *, double *, int *, int *);
void secondaryApproach(int *, int *, double *, double *);
void tertiaryApproach(int *, int *, double *, double *);
void tieBlockAverages(int *, int *, int *, double *, double *, double *,
                      double *, int *, double *);
void monotone(int *, double *, double *);
int myComp(const void *, const void *);
void mySort(double *, double *, double *, int *, int *, int *);

struct quintuple {
    double value;
    double weight;
    double dist;
    int index;
    int jndex;
};

void smacofSSUOEngine(int *nobj, int *ndim, int *ndat, int *itel, int *ties,
                      int *itmax, int *digits, int *width, bool *verbose,
                      double *sold, double *snew, double *eps, int *iind,
```

```
                    int *jind, int *blks, double *edis, double *dhat,
                    double *xold, double *xnew) {
    int Ndat = *ndat, Nobj = *nobj, Ndim = *ndim;
    while (true) {
        for (int k = 0; k < Nobj * Ndim; k++) {
            xnew[k] = 0.0;
        }
        for (int k = 0; k < Ndat; k++) {
            if (edis[k] == 0.0) {
                continue;
            }
            int is = iind[k] - 1, js = jind[k] - 1;
            double elem = dhat[k] / edis[k];
            for (int s = 0; s < Ndim; s++) {
                double add = elem * (xold[is] - xold[js]);
                xnew[is] += add;
                xnew[js] -= add;
                is += Nobj;
                js += Nobj;
            }
        }
        for (int i = 0; i < Nobj; i++) {
            int is = i;
            for (int s = 0; s < Ndim; s++) {
                xnew[is] = xnew[is] / (double)Nobj;
                is += Nobj;
            }
        }
        for (int k = 0; k < Ndat; k++) {
            int is = iind[k] - 1, js = jind[k] - 1;
            double sum = 0.0;
            for (int s = 0; s < Ndim; s++) {
                sum += SQUARE(xnew[is] - xnew[js]);
                edis[k] = sqrt(sum);
                is += Nobj;
                js += Nobj;
            }
        }
        double smid = 0.0;
        for (int k = 0; k < Ndat; k++) {
```

```
            smid += SQUARE(dhat[k] - edis[k]);
        }
        smid /= (double) Ndat;
        double *wght = (double *)calloc(Ndat, sizeof(double));
        for (int k = 0; k < Ndat; k++) {
            wght[k] = 1.0;
        }
        dhat = memcpy(dhat, edis, (size_t)(Ndat * sizeof(double)));
        if (*ties == 1) {
            (void)primaryApproach(ndat, blks, dhat, wght, edis, iind, jind);
        }
        if (*ties == 2) {
            (void)secondaryApproach(ndat, blks, dhat, wght);
        }
        if (*ties == 3) {
            (void)tertiaryApproach(ndat, blks, dhat, wght);
        }
        double ssq = 0.0;
        for (int k = 0; k < Ndat; k++) {
            ssq += SQUARE(dhat[k]);
        }
        *snew = 0.0;
        for (int k = 0; k < Ndat; k++) {
            dhat[k] *= sqrt(((double) Ndat) / ssq);
            *snew += SQUARE(dhat[k] - edis[k]);
        }
        *snew /= (double) Ndat;
        if (*verbose) {
            printf("itel %4d sold %*.*f smid %*.*f snew %*.*f\n", *itel, *width,
                   *digits, *sold, *width, *digits, smid, *width, *digits,
                   *snew);
        }
        if ((*itel == *itmax) || ((*sold - *snew) < *eps)) {
            break;
        }
        xold = memcpy(xold, xnew, (size_t) Nobj * Ndim * sizeof(double));
        *sold = *snew;
        *itel += 1;
        free(wght);
    }
```

```
}

void primaryApproach(int *ndat, int *blks, double *x, double *w, double *d,
                     int *iind, int *jind) {
    int Ndat = *ndat;
    for (int i = 0; i < Ndat; i++) {
        int blksize = blks[i];
        if (blksize > 0) {
            double *extracx = (double *)calloc(blksize, sizeof(double));
            double *extracw = (double *)calloc(blksize, sizeof(double));
            double *extracd = (double *)calloc(blksize, sizeof(double));
            int *extraci = (int *)calloc(blksize, sizeof(int));
            int *extracj = (int *)calloc(blksize, sizeof(int));
            for (int j = 0; j < blksize; j++) {
                extracx[j] = x[i + j];
                extracw[j] = w[i + j];
                extracd[j] = d[i + j];
                extraci[j] = iind[i + j];
                extracj[j] = jind[i + j];
            }
            (void)mySort(extracx, extracw, extracd, extraci, extracj, &blksize);
            for (int j = 0; j < blksize; j++) {
                x[i + j] = extracx[j];
                w[i + j] = extracw[j];
                d[i + j] = extracd[j];
                iind[i + j] = extraci[j];
                jind[i + j] = extracj[j];
            }
            free(extracx);
            free(extracw);
            free(extracd);
            free(extraci);
            free(extracj);
        }
    }
    double *ww = (double *)calloc(Ndat, sizeof(double));
    for (int i = 0; i < Ndat; i++) {
        ww[i] = w[i];
    }
    (void)monotone(ndat, x, ww);
```

```
    free(ww);
    return;
}

void secondaryApproach(int *ndat, int *blks, double *x, double *w) {
    int nblk = 0, Ndat = *ndat;
    for (int k = 0; k < Ndat; k++) {
        if (blks[k] > 0) {
            nblk++;
        }
    }
    double *xsum = (double *)calloc((size_t)nblk, sizeof(double));
    double *wsum = (double *)calloc((size_t)nblk, sizeof(double));
    double *xave = (double *)calloc((size_t)nblk, sizeof(double));
    int *csum = (int *)calloc((size_t)nblk, sizeof(int));
    (void)tieBlockAverages(ndat, &nblk, blks, x, w, xsum, wsum, csum, xave);
    (void)monotone(&nblk, xave, wsum);
    int l = 1;
    for (int k = 0; k < nblk; k++) {
        for (int i = l; i <= l + csum[k] - 1; i++) {
            x[i - 1] = xave[k];
        }
        l += csum[k];
    }
    free(xsum);
    free(wsum);
    free(csum);
    free(xave);
    return;
}

void tertiaryApproach(int *ndat, int *blks, double *x, double *w) {
    int nblk = 0, Ndat = *ndat;
    for (int k = 0; k < Ndat; k++) {
        if (blks[k] > 0) {
            nblk++;
        }
    }
    double *xsum = (double *)calloc((size_t)nblk, sizeof(double));
    double *wsum = (double *)calloc((size_t)nblk, sizeof(double));
```

```
    double *xave = (double *)calloc((size_t)nblk, sizeof(double));
    double *yave = (double *)calloc((size_t)nblk, sizeof(double));
    int *csum = (int *)calloc((size_t)nblk, sizeof(int));
    (void)tieBlockAverages(ndat, &nblk, blks, x, w, xsum, wsum, csum, xave);
    for (int k = 0; k < nblk; k++) {
        yave[k] = xave[k];
    }
    (void)monotone(&nblk, xave, wsum);
    int l = 1;
    for (int k = 0; k < nblk; k++) {
        for (int i = l; i <= l + csum[k] - 1; i++) {
            x[i - 1] = xave[k] + (x[i - 1] - yave[k]);
        }
        l += csum[k];
    }
    free(xsum);
    free(wsum);
    free(xave);
    free(yave);
    free(csum);
    return;
}

void tieBlockAverages(int *ndat, int *nblk, int *blks, double *x, double *w,
                      double *xsum, double *wsum, int *csum, double *xave) {
    int iblk = 0, Ndat = *ndat, Nblk = *nblk;
    for (int k = 0; k < Ndat; k++) {
        if (blks[k] > 0) {
            double sum1 = 0.0, sum2 = 0.0;
            for (int l = k; l < k + blks[k]; l++) {
                sum1 += w[l] * x[l];
                sum2 += w[l];
            }
            xsum[iblk] = sum1;
            wsum[iblk] = sum2;
            csum[iblk] = blks[k];
            iblk++;
        }
    }
    for (int i = 0; i < Nblk; i++) {
```

```
        xave[i] = xsum[i] / wsum[i];
    }
}

int myComp(const void *px, const void *py) {
    double x = ((struct quintuple *)px)->value;
    double y = ((struct quintuple *)py)->value;
    return (int)copysign(1.0, x - y);
}

void mySort(double *x, double *w, double *d, int *iind, int *jind, int *n) {
    int nn = *n;
    struct quintuple *xi = (struct quintuple *)calloc(
        (size_t)nn, (size_t)sizeof(struct quintuple));
    for (int i = 0; i < nn; i++) {
        xi[i].value = x[i];
        xi[i].weight = w[i];
        xi[i].dist = d[i];
        xi[i].index = iind[i];
        xi[i].jndex = jind[i];
    }
    (void)qsort(xi, (size_t)nn, (size_t)sizeof(struct quintuple), myComp);
    for (int i = 0; i < nn; i++) {
        x[i] = xi[i].value;
        w[i] = xi[i].weight;
        d[i] = xi[i].dist;
        iind[i] = xi[i].index;
        jind[i] = xi[i].jndex;
    }
    free(xi);
}

void monotone(int *n, double *x, double *w)
// Function monotone(),
// performs simple linear ordered monotone regression
// Copyright (C) 2020 Frank M.T.A. Busing (e-mail: busing at fsw dot leidenuniv
// dot nl) This function is free software: you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation. This program is distributed in the hope that it
// will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty
```

```
// of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General
// Public License for more details. You should have received a copy of the GNU
// General Public License along with this function. If not, see
// <https://www.gnu.org/licenses/>.
{
    double *rx = &x[-1];
    double *rw = &w[-1];
    size_t *idx = (size_t *)calloc(*n + 1, sizeof(size_t));
    idx[0] = 0;
    idx[1] = 1;
    size_t b = 1;
    double xbm1 = rx[b];
    double wbm1 = rw[b];
    for (size_t i = 2; i <= *n; i++) {
        b++;
        double xb = rx[i];
        double wb = rw[i];
        if (xbm1 > xb) {
            b--;
            double sb = wbm1 * xbm1 + wb * xb;
            wb += wbm1;
            xb = sb / wb;
            while (i < *n && xb >= rx[i + 1]) {
                i++;
                sb += rw[i] * rx[i];
                wb += rw[i];
                xb = sb / wb;
            }
            while (b > 1 && rx[b - 1] > xb) {
                b--;
                sb += rw[b] * rx[b];
                wb += rw[b];
                xb = sb / wb;
            }
        }
        rx[b] = xbm1 = xb;
        rw[b] = wbm1 = wb;
        idx[b] = i;
    }
    size_t from = *n;
```

```
    for (size_t k = b; k > 0; k--) {
        const size_t to = idx[k - 1] + 1;
        const double xk = rx[k];
        for (size_t i = from; i >= to; i--) rx[i] = xk;
        from = to - 1;
    }
    free(idx);
}  // monotone
```

10.6 smacofSSWOEngine.c

```
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SQUARE(x) ((x) * (x))

void primaryApproach(int *, int *, double *, double *, double *, int *, int *);
void secondaryApproach(int *, int *, double *, double *);
void tertiaryApproach(int *, int *, double *, double *);
void tieBlockAverages(int *, int *, int *, double *, double *, double *,
                      double *, int *, double *);
void monotone(int *, double *, double *);
int myComp(const void *, const void *);
void mySort(double *, double *, double *, int *, int *, int *);

struct quintuple {
    double value;
    double weight;
    double dist;
    int index;
    int jndex;
};

void smacofSSWOEngine(int *nobj, int *ndim, int *ndat, int *itel, int *ties,
                      int *itmax, int *digits, int *width, bool *verbose, double *wsum
```

```
                    double *sold, double *snew, double *eps, int *iind,
                    int *jind, int *blks, double *edis, double *dhat,
                    double *wght, double *vinv, double *xold, double *xnew) {
    int Ndat = *ndat, Nobj = *nobj, Ndim = *ndim;
    while (true) {
        double *xtmp = (double *)calloc(Nobj * Ndim, sizeof(double));
        for (int k = 0; k < Ndat; k++) {
            if (edis[k] == 0.0) {
                continue;
            }
            int is = iind[k] - 1, js = jind[k] - 1;
            double elem = wght[k] * dhat[k] / edis[k];
            for (int s = 0; s < Ndim; s++) {
                double add = elem * (xold[is] - xold[js]);
                xtmp[is] += add;
                xtmp[js] -= add;
                is += Nobj;
                js += Nobj;
            }
        }
        for (int k = 0; k < Nobj * Ndim; k++) {
            xnew[k] = 0.0;
        }
        int k = 0;
        for (int j = 0; j < Nobj - 1; j++) {
            for (int i = j + 1; i < Nobj; i++) {
                double elem = vinv[k];
                int is = i, js = j;
                for (int s = 0; s < Ndim; s++) {
                    double add = elem * (xtmp[is] - xtmp[js]);
                    xnew[is] += add;
                    xnew[js] -= add;
                    is += Nobj;
                    js += Nobj;
                }
                k++;
            }
        }
        free(xtmp);
        for (int k = 0; k < Ndat; k++) {
```

```
            int is = iind[k] - 1, js = jind[k] - 1;
            double sum = 0.0;
            for (int s = 0; s < Ndim; s++) {
                sum += SQUARE(xnew[is] - xnew[js]);
                is += Nobj;
                js += Nobj;
            }
            edis[k] = sqrt(sum);
        }
        double smid = 0.0;
        for (int k = 0; k < Ndat; k++) {
            smid += wght[k] * SQUARE(dhat[k] - edis[k]);
        }
        smid /= *wsum;
        dhat = memcpy(dhat, edis, (size_t)(Ndat * sizeof(double)));
        if (*ties == 1) {
            (void)primaryApproach(ndat, blks, dhat, wght, edis, iind, jind);
        }
        if (*ties == 2) {
            (void)secondaryApproach(ndat, blks, dhat, wght);
        }
        if (*ties == 3) {
            (void)tertiaryApproach(ndat, blks, dhat, wght);
        }
        double ssq = 0.0;
        for (int k = 0; k < Ndat; k++) {
            ssq += wght[k] * SQUARE(dhat[k]);
        }
        *snew = 0.0;
        for (int k = 0; k < Ndat; k++) {
            dhat[k] *= sqrt(*wsum / ssq);
            *snew += wght[k] * SQUARE(dhat[k] - edis[k]);
        }
        *snew /= *wsum;
        if (*verbose) {
            printf("itel %4d sold %*.*f smid %*.*f snew %*.*f\n", *itel, *width,
                   *digits, *sold, *width, *digits, smid, *width, *digits,
                   *snew);
        }
        if ((*itel == *itmax) || ((*sold - *snew) < *eps)) {
```

```
            break;
        }
        xold = memcpy(xold, xnew, (size_t) Nobj * Ndim * sizeof(double));
        *sold = *snew;
        *itel += 1;
    }
}

void primaryApproach(int *ndat, int *blks, double *x, double *w, double *d,
                     int *iind, int *jind) {
    int Ndat = *ndat;
    for (int i = 0; i < Ndat; i++) {
        int blksize = blks[i];
        if (blksize > 0) {
            double *extracx = (double *)calloc(blksize, sizeof(double));
            double *extracw = (double *)calloc(blksize, sizeof(double));
            double *extracd = (double *)calloc(blksize, sizeof(double));
            int *extraci = (int *)calloc(blksize, sizeof(int));
            int *extracj = (int *)calloc(blksize, sizeof(int));
            for (int j = 0; j < blksize; j++) {
                extracx[j] = x[i + j];
                extracw[j] = w[i + j];
                extracd[j] = d[i + j];
                extraci[j] = iind[i + j];
                extracj[j] = jind[i + j];
            }
            (void)mySort(extracx, extracw, extracd, extraci, extracj, &blksize);
            for (int j = 0; j < blksize; j++) {
                x[i + j] = extracx[j];
                w[i + j] = extracw[j];
                d[i + j] = extracd[j];
                iind[i + j] = extraci[j];
                jind[i + j] = extracj[j];
            }
            free(extracx);
            free(extracw);
            free(extracd);
            free(extraci);
            free(extracj);
        }
```

```
    }
    double *ww = (double *)calloc(Ndat, sizeof(double));
    for (int i = 0; i < Ndat; i++) {
        ww[i] = w[i];
    }
    (void)monotone(ndat, x, ww);
    free(ww);
    return;
}

void secondaryApproach(int *ndat, int *blks, double *x, double *w) {
    int nblk = 0, Ndat = *ndat;
    for (int k = 0; k < Ndat; k++) {
        if (blks[k] > 0) {
            nblk++;
        }
    }
    double *xsum = (double *)calloc((size_t)nblk, sizeof(double));
    double *wsum = (double *)calloc((size_t)nblk, sizeof(double));
    double *xave = (double *)calloc((size_t)nblk, sizeof(double));
    int *csum = (int *)calloc((size_t)nblk, sizeof(int));
    (void)tieBlockAverages(ndat, &nblk, blks, x, w, xsum, wsum, csum, xave);
    (void)monotone(&nblk, xave, wsum);
    int l = 1;
    for (int k = 0; k < nblk; k++) {
        for (int i = l; i <= l + csum[k] - 1; i++) {
            x[i - 1] = xave[k];
        }
        l += csum[k];
    }
    free(xsum);
    free(wsum);
    free(csum);
    free(xave);
    return;
}

void tertiaryApproach(int *ndat, int *blks, double *x, double *w) {
    int nblk = 0, Ndat = *ndat;
    for (int k = 0; k < Ndat; k++) {
```

```
        if (blks[k] > 0) {
            nblk++;
        }
    }
    double *xsum = (double *)calloc((size_t)nblk, sizeof(double));
    double *wsum = (double *)calloc((size_t)nblk, sizeof(double));
    double *xave = (double *)calloc((size_t)nblk, sizeof(double));
    double *yave = (double *)calloc((size_t)nblk, sizeof(double));
    int *csum = (int *)calloc((size_t)nblk, sizeof(int));
    (void)tieBlockAverages(ndat, &nblk, blks, x, w, xsum, wsum, csum, xave);
    for (int k = 0; k < nblk; k++) {
        yave[k] = xave[k];
    }
    (void)monotone(&nblk, xave, wsum);
    int l = 1;
    for (int k = 0; k < nblk; k++) {
        for (int i = l; i <= l + csum[k] - 1; i++) {
            x[i - 1] = xave[k] + (x[i - 1] - yave[k]);
        }
        l += csum[k];
    }
    free(xsum);
    free(wsum);
    free(xave);
    free(yave);
    free(csum);
    return;
}

void tieBlockAverages(int *ndat, int *nblk, int *blks, double *x, double *w,
                      double *xsum, double *wsum, int *csum, double *xave) {
    int iblk = 0, Ndat = *ndat, Nblk = *nblk;
    for (int k = 0; k < Ndat; k++) {
        if (blks[k] > 0) {
            double sum1 = 0.0, sum2 = 0.0;
            for (int l = k; l < k + blks[k]; l++) {
                sum1 += w[l] * x[l];
                sum2 += w[l];
            }
            xsum[iblk] = sum1;
```

```
            wsum[iblk] = sum2;
            csum[iblk] = blks[k];
            iblk++;
        }
    }
    for (int i = 0; i < Nblk; i++) {
        xave[i] = xsum[i] / wsum[i];
    }
}

int myComp(const void *px, const void *py) {
    double x = ((struct quintuple *)px)->value;
    double y = ((struct quintuple *)py)->value;
    return (int)copysign(1.0, x - y);
}

void mySort(double *x, double *w, double *d, int *iind, int *jind, int *n) {
    int nn = *n;
    struct quintuple *xi = (struct quintuple *)calloc(
        (size_t)nn, (size_t)sizeof(struct quintuple));
    for (int i = 0; i < nn; i++) {
        xi[i].value = x[i];
        xi[i].weight = w[i];
        xi[i].dist = d[i];
        xi[i].index = iind[i];
        xi[i].jndex = jind[i];
    }
    (void)qsort(xi, (size_t)nn, (size_t)sizeof(struct quintuple), myComp);
    for (int i = 0; i < nn; i++) {
        x[i] = xi[i].value;
        w[i] = xi[i].weight;
        d[i] = xi[i].dist;
        iind[i] = xi[i].index;
        jind[i] = xi[i].jndex;
    }
    free(xi);
}

void monotone(int *n, double *x, double *w)
// Function monotone(),
```

```
// performs simple linear ordered monotone regression
// Copyright (C) 2020 Frank M.T.A. Busing (e-mail: busing at fsw dot leidenuniv
// dot nl) This function is free software: you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation. This program is distributed in the hope that it
// will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty
// of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General
// Public License for more details. You should have received a copy of the GNU
// General Public License along with this function. If not, see
// <https://www.gnu.org/licenses/>.
{
    double *rx = &x[-1];
    double *rw = &w[-1];
    size_t *idx = (size_t *)calloc(*n + 1, sizeof(size_t));
    idx[0] = 0;
    idx[1] = 1;
    size_t b = 1;
    double xbm1 = rx[b];
    double wbm1 = rw[b];
    for (size_t i = 2; i <= *n; i++) {
        b++;
        double xb = rx[i];
        double wb = rw[i];
        if (xbm1 > xb) {
            b--;
            double sb = wbm1 * xbm1 + wb * xb;
            wb += wbm1;
            xb = sb / wb;
            while (i < *n && xb >= rx[i + 1]) {
                i++;
                sb += rw[i] * rx[i];
                wb += rw[i];
                xb = sb / wb;
            }
            while (b > 1 && rx[b - 1] > xb) {
                b--;
                sb += rw[b] * rx[b];
                wb += rw[b];
                xb = sb / wb;
            }
```

```
        }
        rx[b] = xbm1 = xb;
        rw[b] = wbm1 = wb;
        idx[b] = i;
    }
    size_t from = *n;
    for (size_t k = b; k > 0; k--) {
        const size_t to = idx[k - 1] + 1;
        const double xk = rx[k];
        for (size_t i = from; i >= to; i--) rx[i] = xk;
        from = to - 1;
    }
    free(idx);
}  // monotone
```

References

Anderson, E. 1936. “The Species Problem in Iris.” *Annals of the Missouri Botanical Garden* 23: 457–509. https://biostor.org/reference/11559.

Busing, F. M. T. A. 2022. “Monotone Regression: A Simple and Fast O(n) PAVA Implementation.” *Journal of Statistical Software* 102 (Code Snippet 1). https://www.jstatsoft.org/index.php/jss/article/view/v102c01/4306.

De Gruijter, D. N. M. 1967. “The Cognitive Structure of Dutch Political Parties in 1966.” Report E019-67. Psychological Institute, University of Leiden.

De Leeuw, J. 1975. “An Alternating Least Squares Approach to Squared Distance Scaling.” Department of Data Theory FSW/RUL.

———. 1977a. “Applications of Convex Analysis to Multidimensional Scaling.” In *Recent Developments in Statistics*, edited by J. R. Barra, F. Brodeau, G. Romier, and B. Van Cutsem, 133–45. Amsterdam, The Netherlands: North Holland Publishing Company.

———. 1977b. “Correctness of Kruskal’s Algorithms for Monotone Regression with Ties.” *Psychometrika* 42: 141–44.

———. 2016. “Gower Rank.” 2016. https://jansweb.netlify.app/publication/deleeuw-e-16-k/deleeuw-e-16-k.pdf.

———. 2025. “Squared Distance Scaling.” 2025.

De Leeuw, J., P. Groenen, and P. Mair. 2016. “Full-Dimensional Scaling.” 2016. https://jansweb.netlify.app/publication/deleeuw-groenen-mair-e-16-e/deleeuw-groenen-mair-e-16-e.pdf.

De Leeuw, J., and P. Mair. 2009. “Multidimensional Scaling Using Majorization: SMACOF in R.” *Journal of Statistical Software* 31 (3): 1–30. https://www.jstatsoft.org/article/view/v031i03.

Ekman, G. 1954. “Dimensions of Color Vision.” *Journal of Psychology* 38: 467–74.

Fisher, R. A. 1936. “The Use of Multiple Measurements in Taxonomic Problems.” *Annals of Eugenics* 7: 179–88. https://onlinelibrary.wiley.com/doi/epdf/10.1111/j.1469-1809.1936.tb02137.x.

Guttman, L. 1968. “A General Nonmetric Technique for Fitting the Smallest Coordinate Space for a Configuration of Points.” *Psychometrika* 33: 469–506.

Kruskal, J. B. 1964a. “Multidimensional Scaling by Optimizing Goodness of Fit to a Nonmetric Hypothesis.” *Psychometrika* 29: 1–27.

———. 1964b. “Nonmetric Multidimensional Scaling: a Numerical Method.” *Psychometrika* 29: 115–29.

Mair, P., P. J. F. Groenen, and J. De Leeuw. 2022. “More on Multidimensional Scaling in R: smacof Version 2.” *Journal of Statistical Software* 102 (10): 1–47. https://www.jstatsoft.org/article/view/v102i10.

Mersmann, O. 2024. *microbenchmark: Accurate Timing Functions*. https://CRAN.R-project.org/package=microbenchmark.

Qiu, Y., and J. Mei. 2024. *RSpectra: Solvers for Large-Scale Eigenvalue and SVD Problems*. https://CRAN.R-project.org/package=RSpectra.

R Core Team. 2025. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

Rothkopf, E. Z. 1957. "A Measure of Stimulus Similarity and Errors in some Paired-associate Learning." *Journal of Experimental Psychology* 53: 94–101.

Takane, Y. 1977. "On the Relations among Four Methods of Multidimensional Scaling." *Behaviormetrika* 4: 29–42.

Wish, M. 1971. "Individual Differences in Perceptions and Preferences Among Nations." In *Attitude Research Reaches New Heights*, edited by C. W. King and D. Tigert, 312–28. American Marketing Association.