Tweaking the SMACOF Engine

Jan de Leeuw

Version 06, October 10, 2017

Abstract

The *smacof* algorithm for (metric, Euclidean, least squares) multidimensional scaling is rewritten so that all computation is done in C, with only the data management, memory allocation, iteration counting, and I/O handled by R. All symmetric matrices use compact, lower triangular, column-wise storage. Second derivatives of the loss function are provided, but non-metric scaling, individual differences, and constraints still have to be added.

Contents

Note: This is a working paper which will be expanded/updated frequently. All suggestions for improvement are welcome. The directory gifi.stat.ucla.edu/tweaking has a pdf version, the bib file, the complete Rmd file with the code chunks, and the R and C source code.

1 Introduction

The smacof algorithm is a majorization (or MM) algorithm to minimize the least squares loss function

$$\sigma(X) := \frac{1}{2} \sum_{1<=i<j<=n} \sum w_{ij}(\delta_{ij} - d_{ij}(X))^2 \tag{1}$$

over *configurations* $X \in \mathbb{R}^{n\times p}$. Here the *weights* w_{ij} and *dissimilarities* δ_{ij} are known non-negative numbers, and the $d_{ij}(X)$ are the Euclidean distances between the rows of X, i.e.

$$d_{ij}(X) := \sqrt{(e_i - e_j)'XX'(e_i - e_j)}, \tag{2}$$

with e_i and e_j unit vectors in $\mathbb{R}^n$, with a single element equal to one and all other elements equal to zero.

The smacof algorithm generates a sequence of configurations $X^{(k)}$ by the rule

$$X^{(k+1)} = V^+ B(X^{(k)})X^{(k)} \tag{3}$$

Here

$$V := \sum_{1<=i<j<=n} \sum w_{ij} A_{ij}, \tag{4}$$

and

$$B(X) := \sum_{1<=i<j<=n} \sum w_{ij} \frac{\delta_{ij}}{d_{ij}(X)} A_{ij}, \tag{5}$$

where $A_{ij} = (e_i - e_j)(e_i - e_j)'$ and V^+ is the Moore-Penrose inverse of V. Note that V, V^+, and $B(X)$ are all postitive semi-definite and doubly centered. Moreover V has rank $n-1$ if the weights are irreducible, i.e. the nonzero weights are not the direct sum of a number of smaller matrices. The only vector in the null-space of V is e, a vector with n elements all equal to one. Thus in the irreducible case

$$X^{(k+1)} = (V + \frac{1}{n} ee')^{-1} B(X^{(k)})X^{(k)}. \tag{6}$$

Also, in the case of unit weights $V = nI - ee'$, with e a vector with n elements all equal to one, so that

$$X^{(k+1)} = \frac{1}{n} B(X^{(k)}) X^{(k)}. \tag{7}$$

For unit weights, i.e. $w_{ij} = 1$ for all $i < j$, algorithm (3) was originally proposed by Guttman (1968), as a simple rewrite of the stationary equations $\mathcal{D}\sigma(X) = 0$. Guttman did not show the algorithm actually converged. For general non-negative weights it was shown to converge to stationary points by De Leeuw (1977), even though there clearly are configurations where (1) is not differentiable. Differentiability of (1) at local minima was shown in De Leeuw (1984). In De Leeuw (1977), De Leeuw and Heiser (1977), and De Leeuw and Heiser (1980) the smacof algorithm is extended to non-euclidean distances, to individual difference scaling, and to various types of constrained configurations. De Leeuw (1988) shows that in general the smacof algorithm has a linear convergence rate. A comprehensive implementation in R was published by De Leeuw and Mair (2009).

One important characteristic of smacof, next to its global convergence and monotone convergence, is that it is easily implemented in a matrix language such as R or MATLAB. The definition (4) may look expensive computationally, but, expandng the w_{ij} to a symmetric hollow matrix, it is actually

$$v_{ij} = \begin{cases} -w_{ij} & \text{if } i \neq j, \\ \sum_{j \neq i} w_{ij} & \text{if } i = j, \end{cases} \tag{8}$$

and a similar result applies for $B(X)$.

The iterations (3) is thus a straightforward matrix operation, which require at most $\frac{3}{2}n^2p$ multiplications. As Guttman (1968) already observed, there is a formal similarity to the power method for computing some of the dominant eigenvalues and corresponding eigenvectors, although in smacof the matrix $B(X)$ changes at each iteration. At a stationary point X, which is a fixed point of the smacof iterations, we have $V^+B(X)X = X$, i.e. $V^+B(X)$ has p eigenvalues equal to one, with X corresponding eigenvectors. De Leeuw (2014) shows that if the p eigenvalues equal to one are actually the largest eigenvalues, then X is a global minimum of stress.

If we use a matrix language, then we will most likely use the symmetric matrices V and $B(X)$ in our computations, and store them as full symmetric matrices. This, of course, is redundant, because of symmetry. It is possible to compute only the elements on and below the main diagonal and then store corresponding elements above the diagonal as well. This may save some time, but it does not save storage. In this paper we go one step further, and we compute and store all symmetric matrices in compact lower triangular column-wise form. The matrix operations now become more complicated, involving loops and index calculations, and they are most naturally done in C (or some other compiled language). Thus we save storage, which may be important in large examples. We may not save computing time, because the compact storage calculations are inherently more complicated, and matrix computations in R are

already calls to efficient LAPACK C routines. One part of this paper is a time comparison of a standard smacof implementation in pure R and several implementations which use compact storage of the symmetric matrices, with all the necessary computations done in C.

2 Implementation

One thing to remember is that we do not compare R and C implementations of smacof, we compare implementations in Jan's R and Jan's C. As in many other papers, we use a standard template for iterative routines in R. The C routines use inline functions to map matrix indexing from the colum-wise format of R to the row-wise format of C. We do not use any matrices (i.e. any double indexing) in C. Also, we use the calling conventions of the .C() interface in R, i.e. all functions return void, and all arguments are passed by reference (as pointers). The C routines do not handle I/O or dynamic memory allocation, that is all the responsibility of R. This implies that if a routine needs any working memory, then it is created in R, and a pointer is passed to C. The driver routine for the iterations is also always in R.

This seems a natural division of labor between C and R. We could have decided to use .Call() or Rcpp, but we would like our C routines to be usuable with C drivers, in cases where R is not around at all. Also, if I had wanted to learn C++ I would have done it thirty years ago. Now it is too late, thank God.

In the appendix there is code for four R implementations of smacof. `smacofR()` only uses basic R and no user written C code, symmetric matrices are stored as full matrices. `smacofRC()` uses compact lower-triangular storage, and calls a number of small C routines within each iteration to update the configuration. `smacofRCU()` is similar to `smacofRC()`, but it combines the small C routines into a single larger C routine, in order to minimize the number of .C() calls. This program was included following a suggestion of Patrick Groenen. `smacofRCU64()` uses the `.C64()` interface call described by Gerber, Moesinger, and Furrer (2016) and Gerber, Moesinger, and Furrer (2017). Compared to `C()` this new interface promises less copying of the arguments, and thus more efficiency, both in computing times and in memory use. We have checked that in all cases for a given random start the two programs carry out the exactly same iterations and converge to exactly the same solution, although of course from different random starts they can converge to different stationary points.

3 Timing

For our timing we use the example where all weights and all dissimilarities are one. In a sense, this is a worst-case example, because it corresponds with power method iterations in the case that all eigenvalues are equal. If X is a solution to the stationary equations, then any permutation of its rows is a solutions as well, with the same loss function value. Since, from De Leeuw (1984), at a local minimum all rows of X are different, this means there are at least $n!$ local minima with the same function value, and specifically at least $n!$ global minima.

For all runs we use a maximum of 100 iterations, and we stop if the maximum absolute value of $X^{(k)} - X^{(k+1)}$ is less than `1e-15`. Of course 100 iterations are not enough for convergence, but they are perfectly fine for time comparisons, because all three algorithms produce exactly the same sequence of iterations. Using 100 starting points and 100 iterations per starting point basically means timing 10,000 smacof updates.

We start with $n = 4$, using 100 random starts. The median user time over starts, measured by `system.time()`, for the pure R implementation `smacofR()` is 0.005, for the compact storage `smacofRC()` it is 0.071, for `smacofRCU()` it is 0.019 and for `smacofRCU64()` it is 0.021. `smacofR()` is 16 times faster than `smacofRC()`, but only 4 times faster than `smacofRCU()`. Because `smacofRCU64()` has more overhead it is a tiny bit slower than `smacofRCU()` in this small example.

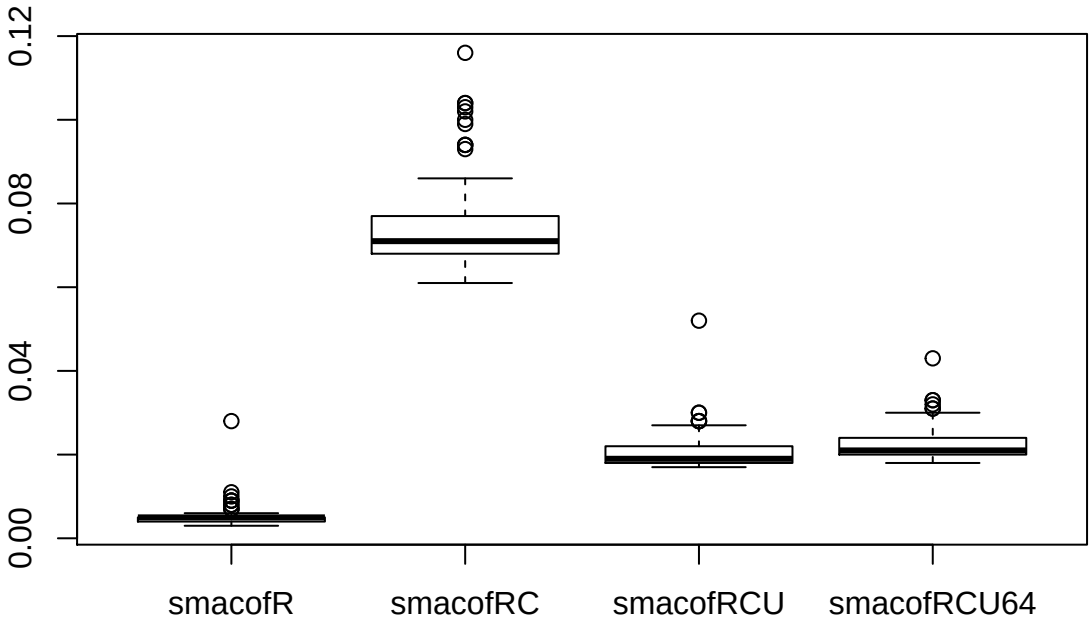

Figure 1: Boxplot for n = 4

The four medians for $n = 10$ are 0.008 for `smacofR()`, 0.105 for `smacofRC()`, 0.028 for `smacofRCU()`, and 0.031 for `smacofRCU64()`. This still gives roughly the same conclusions on relative times as for $n = 4$.

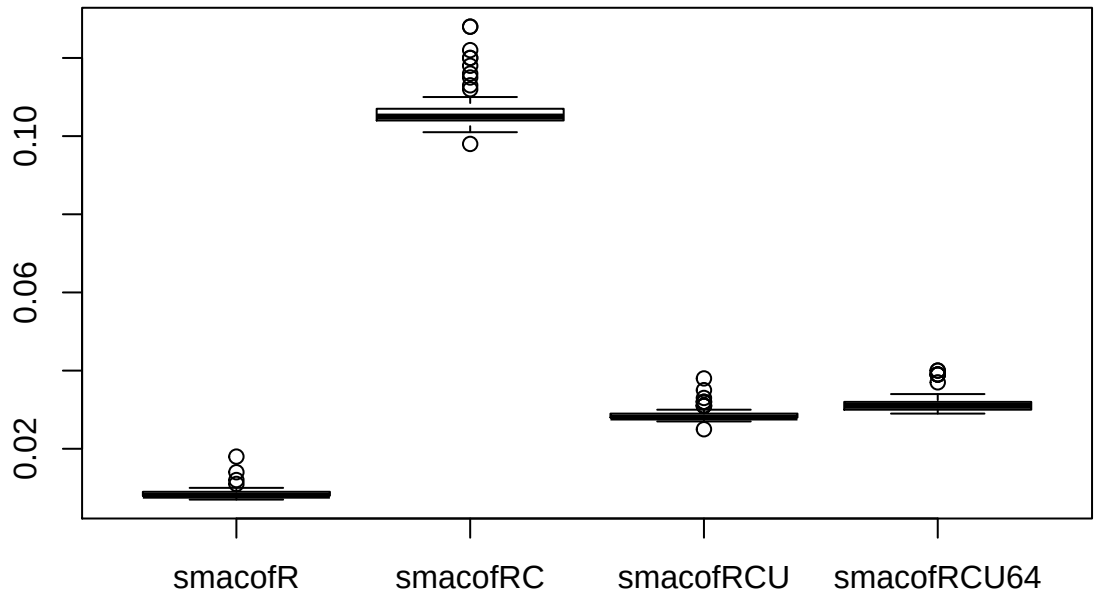

Figure 2: Boxplot for n = 10

The medians for $n = 50$ are 0.043 for `smacofR()`, 0.114 for `smacofRC()`, 0.033 for `smacofRCU()`, and 0.035 for `smacofRCU64()`. `smacofR()` is still 3 times faster than `smacofRC()`, but now `smacofRCU()` has caught up with `smacofR()`. Time for `smacofRCU64()` is the same as for `smacofRCU()`.

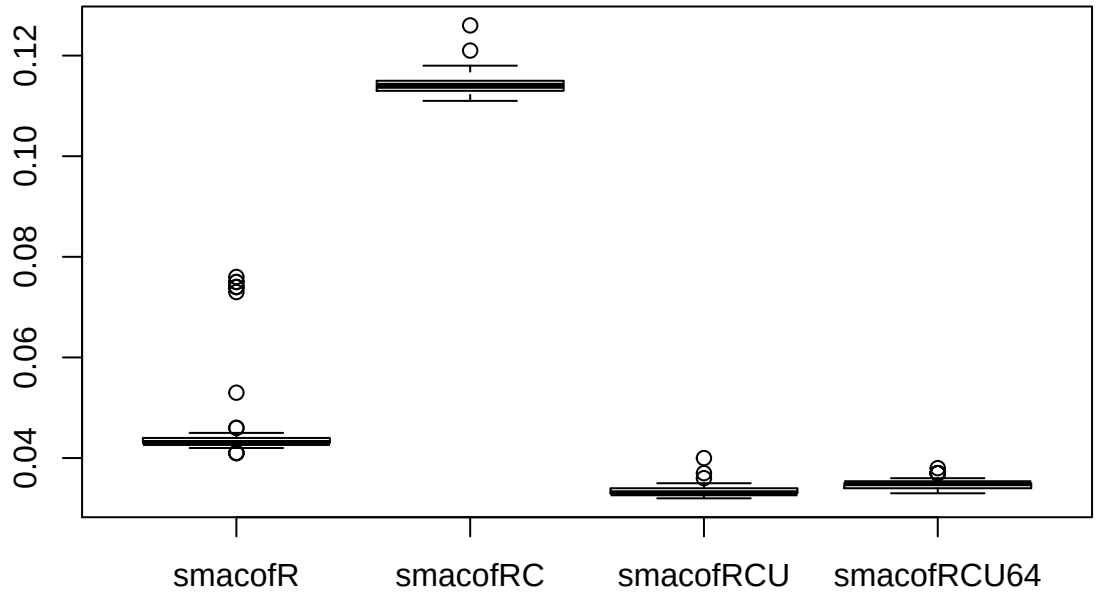

Figure 3: Boxplot for n = 50

For $n = 100$ things begin to change. The medians are 0.2025 for `smacofR()`, 0.143 for `smacofRC()`, 0.055 for `smacofRCU()`, and 0.05 for `smacofRCU64()`. `smacofR()` is now the slowest of the three, and `smacofRCU()` remains three times faster than `smacofRC()`. `smacofRCU64()` is now about 15% faster than `smacofRCU()`.

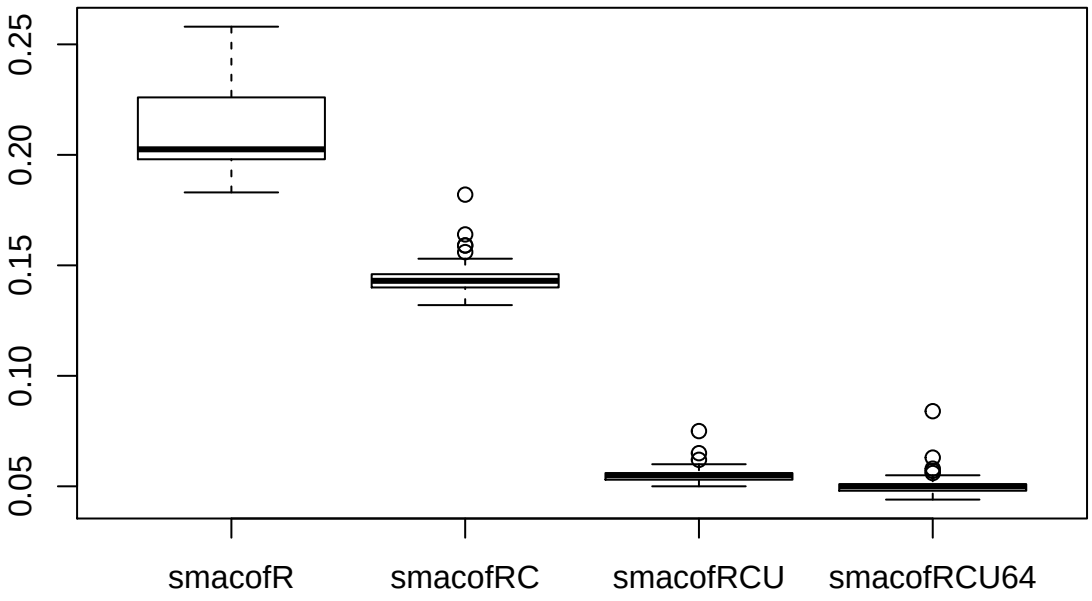

Figure 4: Boxplot for n = 100

The medians for $n = 250$ are 1.2745 for `smacofR()`, 0.321 for `smacofRC()`, 0.19 for `smacofRCU()`, and 0.145 for `smacofRCU64()`. `smacofR()` is losing ground, and `smacofRC()` and `smacofRCU()` are getting closer, although `smacofRCU()` still has the clear advantage. `smacofRCU64()` is about 20% faster than `smacofRCU()`.

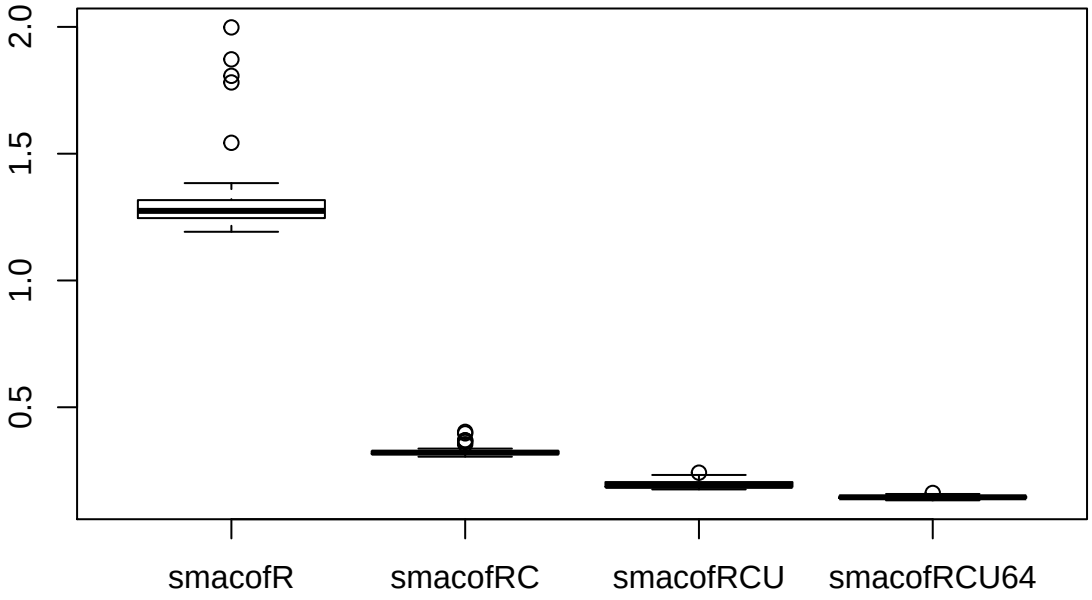

Figure 5: Boxplot for n = 250

4 Conclusion

More extensive comparisons would be useful. We have not used weights, and consequently did not look at multidimensional scaling problems such as unfolding. We did not include non-metric options or individual difference options yet. This will come later.

Also note that the code in the appendix includes both pure R and compact storage C versions to compute the Hessian of (1). Again, this is for future use, to implement variations of Newton's method and to draw pseudo-confidence intervals around the points of the configuration at the solution. There are also various utilities included, plus an interface to a subset of LAPACKE (Author (2016)). Not all of this is used in the current project.

For now it seems that for large n the compact storage routines in C are comparable in speed to pure R smacof, if not faster, especially if the number of .C() calls within an iteration are limited. And they are obviously superior in terms of storage, although these days that is hardly a consideration any more. Ultimately, however, wasting space is just inelegant, and the interesting exercise was how much execution time we had to sacrifice to please our esthetic prejudices.

5 Appendix: Code

5.1 Pure R code

5.1.1 smacofR.R

```
library(MASS)

smacofLossR <- function (d, w, delta) {
  return (sum (w * (delta - d) ^ 2) / 4)
}

smacofBmatR <- function (d, w, delta) {
  dd <- ifelse (d == 0, 0, 1 / d)
  b <- -dd * w * delta
  diag (b) <- -rowSums (b)
  return(b)
}

smacofVmatR <- function (w) {
  v <- -w
  diag(v) <- -rowSums(v)
  return (v)
}
```

```
smacofGuttmanR <- function (x, b, vinv) {
  return (vinv %*% b %*% x)
}

smacofGradientR <- function (x, b, v) {
  return ((v - b) %*% x)
}

smacofHmatR <- function (x, b, v, d, w, delta) {
  n <- nrow (x)
  p <- ncol (x)
  r <- n * p
  h <- matrix (0, r, r)
  dd <- ifelse (d == 0, 0, 1 / d)
  cc <- w * delta * (dd ^ 3)
  for (s in 1:p) {
    for (t in 1:s) {
      cst <- matrix (0, n, n)
      for (i in 1:n) {
        for (j in 1:n) {
          cst[i, j] <- cc[i, j] * (x[i, s] - x[j, s]) * (x[i, t] - x[j, t])
        }
      }
      cst <- -cst
      diag(cst) <- -rowSums(cst)
      if (s == t) {
        h[(s - 1) * n + 1:n, (s - 1) * n + 1:n] <- b - cst
      } else {
        h[(s - 1) * n + 1:n, (t - 1) * n + 1:n] <- -cst
        h[(t - 1) * n + 1:n, (s - 1) * n + 1:n] <- -cst
      }
    }
  }
  return (h)
}

smacofHessianR <- function (x, b, v, d, w, delta) {
  n <- nrow (x)
  p <- ncol (x)
  h <- -smacofHmatR (x, b, v, d, w, delta)
  for (s in 1:p) {
    h[(s - 1) * n + 1:n, (s - 1) * n + 1:n] <-
      h[(s - 1) * n + 1:n, (s - 1) * n + 1:n] + v
  }
```

```
  return(h)
}

smacofInitialR <- function (delta, p) {
  n <- nrow(delta)
  delta <- delta ^ 2
  rw <- rowSums (delta) / n
  sw <- sum (delta) / (n ^ 2)
  h <- -(delta - outer (rw, rw, "+") + sw) / 2
  e <- eigen (h)
  ea <- e$values
  ev <- e$vector
  ea <- ifelse (ea > 0, sqrt (abs(ea)), 0)[1:p]
  return (ev[, 1:p] %*% diag (ea))
}

smacofR <-
  function (w,
            delta,
            p,
            xold = smacofInitialR(delta, p),
            itmax = 100,
            eps = 1e-10,
            verbose = FALSE) {
    itel <- 1
    dold <- as.matrix (dist (xold))
    sold <- smacofLossR (dold, w, delta)
    bold <- smacofBmatR (dold, w, delta)
    vinv <- ginv (smacofVmatR (w))
    repeat {
      xnew <- smacofGuttmanR (xold, bold, vinv)
      eiff <- max (abs (xold - xnew))
      dnew <- as.matrix (dist (xnew))
      bnew <- smacofBmatR (dnew, w, delta)
      snew <- smacofLossR (dnew, w, delta)
      if (verbose) {
        cat(
          "itel ",
          formatC(itel, width = 4, format = "d"),
          "eiff ",
          formatC(
            eiff,
            width = 15,
            digits = 10,
```

```
            format = "f"
          ),
          "sold ",
          formatC(
            sold,
            width = 15,
            digits = 10,
            format = "f"
          ),
          "snew ",
          formatC(
            snew,
            width = 15,
            digits = 10,
            format = "f"
          ),
          "\n"
        )
      }
      if ((eiff < eps) || (itel == itmax)) {
        break
      }
      itel <- itel + 1
      xold <- xnew
      bold <- bnew
      dold <- dnew
      sold <- snew
    }
    return (list (
      x = xnew,
      d = dnew,
      b = bnew,
      s = snew,
      itel = itel
    ))
  }
```

5.2 R Glue

5.2.1 smacofRC.R

```
smacofLossRC <- function (d, w, delta) {
  m <- length (d)
  h <-
    .C(
      "smacofLossC",
      as.double (d),
      as.double (w),
      as.double (delta),
      as.integer (m),
      stress = as.double (0)
    )
  return (h$stress)
}

smacofDistRC <- function (x, p) {
  n <- round (length (x) / p)
  m <- n * (n - 1) / 2
  h <-
    .C("smacofDistC",
       as.double (x),
       as.integer (n),
       as.integer(p),
       dist = as.double (rep(0, m)))
  return (h$dist)
}

smacofInitialRC <- function (delta, p) {
  m <- length (delta)
  n <- round ((1 + sqrt (1 + 8 * m)) / 2)
  r <- n * (n + 1) / 2
  h <-
    .C(
      "smacofInitialC",
      as.double(delta),
      as.integer(n),
      as.integer(p),
      as.double (rep(0, n)),
      as.double (rep(0, r)),
      as.double (rep(0, n * n)),
      as.double (rep (0, n)),
      x = as.double (rep (0, n * p))
    )
  return (h$x)
}
```

```
smacofBmatRC <- function (d, w, delta) {
  m <- length (w)
  n <- round ((1 + sqrt (1 + 8 * m)) / 2)
  r <- n * (n + 1) / 2
  h <-
    .C(
      "smacofBmatC",
      as.double(d),
      as.double(w),
      as.double(delta),
      as.integer (n),
      bmat = as.double(rep(0, r))
    )
  return (h$bmat)
}

smacofVmatRC <- function (w) {
  m <- length (w)
  n <- round ((1 + sqrt (1 + 8 * m)) / 2)
  r <- n * (n + 1) / 2
  h <-
    .C("smacofVmatC",
       as.double(w),
       as.integer (n),
       vmat = as.double(rep(0, r)))
  return (h$vmat)
}

smacofGuttmanRC <- function (x, b, vinv) {
  m <- length (b)
  n <- round ((sqrt (1 + 8 * m) - 1) / 2)
  r <- length (x)
  p <- round (r / n)
  h <-
    .C(
      "smacofGuttmanC",
      as.double(x),
      as.double (b),
      as.double (vinv),
      as.integer(n),
      as.integer(p),
      as.double (rep(0, r)),
      y = as.double (rep(0, r))
    )
```

```
  return (h$y)
}

smacofGradientRC <- function (x, b, v) {
  m <- length (b)
  n <- round ((sqrt (1 + 8 * m) - 1) / 2)
  r <- length (x)
  p <- round (r / n)
  h <-
    .C(
      "smacofGradientC",
      as.double(x),
      as.double (b),
      as.double (v),
      as.integer(n),
      as.integer(p),
      y = as.double (rep(0, r))
    )
  return (h$y)
}

smacofHmatRC <- function (x, b, v, d, w, delta) {
  m <- length (w)
  n <- round ((1 + sqrt (1 + 8 * m)) / 2)
  q <- n * (n + 1) / 2
  r <- length (x)
  p <- round (r / n)
  h <-
    .C(
      "smacofHmatC",
      as.double(x),
      as.double (b),
      as.double (v),
      as.double (w),
      as.double (delta),
      as.double (d),
      as.integer(n),
      as.integer(p),
      as.double (rep (0, q)),
      hmat = as.double (rep (0, r * (r + 1) / 2))
    )
  return (h$hmat)
}
```

```r
smacofHessianRC <- function (x, b, v, d, w, delta) {
  m <- length (w)
  n <- round ((1 + sqrt (1 + 8 * m)) / 2)
  q <- n * (n + 1) / 2
  r <- length (x)
  p <- round (r / n)
  h <-
    .C(
      "smacofHessianC",
      as.double(x),
      as.double (b),
      as.double (v),
      as.double (w),
      as.double (delta),
      as.double (d),
      as.integer(n),
      as.integer(p),
      as.double (rep (0, q)),
      hmat = as.double (rep (0, r * (r + 1) / 2))
    )
  return (h$hmat)
}

smacofUpdateRC <- function (x, w, delta, b, vinv) {
  m <- length (b)
  n <- round ((sqrt (1 + 8 * m) - 1) / 2)
  r <- length (x)
  p <- round (r / n)
  q <- n * (n - 1) / 2
  h <-
    .C(
      "smacofUpdateC",
      as.double (x),
      as.double (w),
      as.double (delta),
      as.double (vinv),
      as.integer (n),
      as.integer (p),
      dnew = as.double (rep(0, q)),
      bnew = as.double (b),
      work = as.double (rep (0, n * p)),
      snew = as.double (0),
      xnew = as.double (rep (0, n * p))
    )
```

```
  return (h)
}

smacofUpdateRC64 <- function (x, w, delta, b, vinv) {
  m <- length (b)
  n <- round ((sqrt (1 + 8 * m) - 1) / 2)
  r <- length (x)
  p <- round (r / n)
  q <- n * (n - 1) / 2
  h <-
    .C64(
      "smacofUpdateC",
      SIGNATURE = c(rep("double", 4), rep("integer", 2), rep("double", 5)),
      x = x,
      w = w,
      delta = delta,
      vinv = vinv,
      n = n,
      p = p,
      dnew = numeric_dc (q),
      bnew = as.double (b),
      work = numeric_dc (n * p),
      snew = numeric_dc (1),
      xnew = numeric_dc (n * p),
      INTENT = c(rep("r", 6), "w", "rw", rep ("w", 3)),
      NAOK = TRUE
    )
  return (h)
}

smacofRC <-
  function (w,
            delta,
            p,
            xold = smacofInitialRC(delta, p),
            itmax = 100,
            eps = 1e-10,
            verbose = FALSE) {
    itel <- 1
    dold <- smacofDistRC (xold, p)
    sold <- smacofLossRC (dold, w, delta)
    bold <- smacofBmatRC (dold, w, delta)
    vinv <- mpowerRC (smacofVmatRC (w), -1)
    repeat {
```

```
    xnew <- smacofGuttmanRC (xold, bold, vinv)
    eiff <- max (abs (xold - xnew))
    dnew <- smacofDistRC (xnew, p)
    bnew <- smacofBmatRC (dnew, w, delta)
    snew <- smacofLossRC (dnew, w, delta)
    if (verbose) {
      cat(
        "itel ",
        formatC(itel, width = 4, format = "d"),
        "eiff ",
        formatC(
          eiff,
          width = 15,
          digits = 10,
          format = "f"
        ),
        "sold ",
        formatC(
          sold,
          width = 15,
          digits = 10,
          format = "f"
        ),
        "snew ",
        formatC(
          snew,
          width = 15,
          digits = 10,
          format = "f"
        ),
        "\n"
      )
    }
    if ((eiff < eps) || (itel == itmax)) {
      break
    }
    itel <- itel + 1
    xold <- xnew
    bold <- bnew
    dold <- dnew
    sold <- snew
  }
  return (list (
    x = xnew,
```

```
      d = dnew,
      b = bnew,
      s = snew,
      itel = itel
    ))
  }

smacofRCU <-
  function (w,
            delta,
            p,
            xold = smacofInitialRC(delta, p),
            itmax = 100,
            eps = 1e-10,
            verbose = FALSE) {
    itel <- 1
    dold <- smacofDistRC (xold, p)
    sold <- smacofLossRC (dold, w, delta)
    bold <- smacofBmatRC (dold, w, delta)
    vinv <- mpowerRC (smacofVmatRC (w), -1)
    repeat {
      h <- smacofUpdateRC(xold, w, delta, bold, vinv)
      xnew <- h$xnew
      dnew <- h$dnew
      snew <- h$snew
      bnew <- h$bnew
      eiff <- max (abs (xold - xnew))
      if (verbose) {
        cat(
          "itel ",
          formatC(itel, width = 4, format = "d"),
          "eiff ",
          formatC(
            eiff,
            width = 15,
            digits = 10,
            format = "f"
          ),
          "sold ",
          formatC(
            sold,
            width = 15,
            digits = 10,
            format = "f"
```

```
          ),
          "snew ",
          formatC(
            snew,
            width = 15,
            digits = 10,
            format = "f"
          ),
          "\n"
        )
      }
      if ((eiff < eps) || (itel == itmax)) {
        break
      }
      itel <- itel + 1
      xold <- xnew
      bold <- bnew
      dold <- dnew
      sold <- snew
    }
    return (list (
      x = xnew,
      d = dnew,
      b = bnew,
      s = snew,
      itel = itel
    ))
  }

smacofRCU64 <-
  function (w,
            delta,
            p,
            xold = smacofInitialRC(delta, p),
            itmax = 100,
            eps = 1e-10,
            verbose = FALSE) {
    itel <- 1
    dold <- smacofDistRC (xold, p)
    sold <- smacofLossRC (dold, w, delta)
    bold <- smacofBmatRC (dold, w, delta)
    vinv <- mpowerRC (smacofVmatRC (w), -1)
    repeat {
      h <- smacofUpdateRC64(xold, w, delta, bold, vinv)
```

```
    xnew <- h$xnew
    dnew <- h$dnew
    snew <- h$snew
    bnew <- h$bnew
    eiff <- max (abs (xold - xnew))
    if (verbose) {
      cat(
        "itel ",
        formatC(itel, width = 4, format = "d"),
        "eiff ",
        formatC(
          eiff,
          width = 15,
          digits = 10,
          format = "f"
        ),
        "sold ",
        formatC(
          sold,
          width = 15,
          digits = 10,
          format = "f"
        ),
        "snew ",
        formatC(
          snew,
          width = 15,
          digits = 10,
          format = "f"
        ),
        "\n"
      )
    }
    if ((eiff < eps) || (itel == itmax)) {
      break
    }
    itel <- itel + 1
    xold <- xnew
    bold <- bnew
    dold <- dnew
    sold <- snew
  }
  return (list (
    x = xnew,
```

```
      d = dnew,
      b = bnew,
      s = snew,
      itel = itel
    ))
  }
```

5.2.2 utilsRC.R

```
dorpolRC <- function (n) {
  h <-
    .C("dorpol", as.integer(n), as.double (rep(0, n * n)))
  return (matrix(h[[2]], n, n))
}

docentRC <- function (x) {
  m <- length (x)
  n <- round((sqrt (1 + 8 * m) - 1) / 2)
  h <-
    .C("docent", as.integer (n), as.double (x), as.double (rep(0, m)))
  return(h$y)
}

primatRC <- function (n,
                      m,
                      x,
                      width = 6,
                      precision = 4) {
  h <-
    .C(
      "primat",
      as.integer (n),
      as.integer (m),
      as.integer (width),
      as.integer (precision),
      as.double (x)
    )
}

pritrlRC <- function (x,
                      width = 6,
                      precision = 4) {
  m <- length (x)
```

```
  n <- round ((sqrt (1 + 8 * m) + 1) / 2)
  h <-
    .C("pritrl",
       as.integer (n),
       as.integer (width),
       as.integer (precision),
       as.double (x))
}


pritruRC <- function (x,
                      width = 6,
                      precision = 4) {
  m <- length (x)
  n <- round ((sqrt (1 + 8 * m) - 1) / 2)
  h <-
    .C("pritru",
       as.integer (n),
       as.integer (width),
       as.integer (precision),
       as.double (x))
}


priarrRC <- function (n,
                      m,
                      r,
                      x,
                      width = 6,
                      precision = 4) {
  h <-
    .C(
      "primat",
      as.integer (n),
      as.integer (m),
      as.integer (r),
      as.integer(width),
      as.integer (precision),
      as.double (x)
    )
}

mpowerRC <- function (x, p) {
  m <- length (x)
```

```
  n <- round ((sqrt (1 + 8 * m) - 1) / 2)
  h <-
    .C("mpower",
       as.integer(n),
       as.double (x),
       as.double(p),
       xpow = as.double(rep(0, m)))
  return (h$xpow)
}

trimatRC <- function (x) {
  m <- length (x)
  n <- round ((sqrt (1 + 8 * m) - 1) / 2)
  h <-
    .C("trimat", as.integer(n), as.double (x), y = as.double (rep(0, n * n)))
  return(h$y)
}

mattriRC <- function (x) {
  n <- round (sqrt (length (x)))
  m <- n * (n + 1) / 2
  h <-
    .C("mattri", as.integer(n), as.double (x), y = as.double (rep(0, m)))
  return(h$y)
}

mutrmaRC <- function (n, m, a, x) {
  h <-
    .C("mutrma",
       as.integer(n),
       as.integer(m),
       as.double (a),
       as.double (x),
       y = as.double (rep(0, n * m)))
  return (h$y)
}
```

5.2.3 jacobiRC.R

```
jacobi <- function (a, itmax = 100, eps = 1e-10) {
  m <- length (a)
  n <- (sqrt (1 + 8 * m) - 1) / 2
  i <- 1:n
```

```
  j <- (-(i ^ 2) / 2) + (n + (3 / 2)) * i - n
  h <-
    .C(
      "jacobiC",
      as.integer (n),
      a = as.double (a),
      e = as.double (rep (0, n * n)),
      as.double (rep(0, n)),
      as.double (rep(0, n)),
      as.integer (itmax),
      as.double (eps)
    )
  return (list (values = h$a[j], vectors = matrix (h$e, n, n)))
}
```

5.2.4 lapackeRC.R

```
dposvR <- function (a, b) {
  n <- nrow (a)
  m <- max (ncol (b), 1)
  h <-
    .C("dposv",
       as.integer(n),
       as.integer(m),
       as.double (a),
       as.double (b))
  if (is.null(ncol(b))) {
    return (h[[4]])
  } else {
    return (matrix(h[[4]], n, m))
  }
}

dsyevdR <- function (a) {
  n <- nrow (a)
  x <- matrix (0, n, n)
  h <- .C("dsyevd", as.integer(n), as.double (a), as.double (x))
  return (list(values = h[[3]][1:n], vectors = matrix(h[[2]], n, n)))
}

dgeqrR <- function (x) {
  n <- nrow (x)
  m <- ncol (x)
```

```
  h <-
    .C("dgeqrf",
       as.integer(n),
       as.integer(m),
       z = as.double (x),
       tau = as.double (rep (0, m)))
  return (list (z = matrix(h$z, n, m), tau = h$tau))
}

dorgqrR <- function (z, tau) {
  n <- nrow (z)
  m <- ncol (z)
  h <-
    .C("dorgqr",
       as.integer(n),
       as.integer(m),
       as.double (z),
       as.double (tau))
  return (matrix(h[[3]], n, m))
}

dorthoR <- function(x) {
  n <- nrow (x)
  m <- ncol (x)
  h <-
    .C("dortho",
       as.integer(n),
       as.integer(m),
       as.double (x),
       as.double(rep(0, m)))
  return (matrix(h[[3]], n, m))
}
```

5.3 C code

5.3.1 smacof.h

```
#ifndef SMACOF_H
#define SMACOF_H

#include <lapacke.h>
#include <math.h>
#include <stdbool.h>
```

```
#include <stdio.h>
#include <stdlib.h>

static inline int VINDEX(const int);
static inline int MINDEX(const int, const int, const int);
static inline int SINDEX(const int, const int, const int);
static inline int TINDEX(const int, const int, const int);
static inline int AINDEX(const int, const int, const int, const int, const int);

static inline double SQUARE(const double);
static inline double THIRD(const double);
static inline double FOURTH(const double);
static inline double FIFTH(const double);

static inline double MAX(const double, const double);
static inline double MIN(const double, const double);
static inline int IMIN(const int, const int);
static inline int IMAX(const int, const int);
static inline int IMOD(const int, const int);

void dposv(const int *, const int *, double *, double *);
void dsyevd(const int *, double *, double *);
void dgeqrf(const int *, const int *, double *, double *);
void dorgqr(const int *, const int *, double *, double *);
void dortho(const int *, const int *, double *);
void dorpol(const int *, double *);
void primat(const int *, const int *, const int *, const int *, const double *);
void priarr(const int *, const int *, const int *, const int *, const int *,
            const double *);
void pritrl(const int *, const int *, const int *, const double *);
void pritru(const int *, const int *, const int *, const double *);
void mpinve(const int *, double *, double *);
void mpower(const int *, double *, double *, double *);
void mpinvt(const int *, double *, double *);
void trimat(const int *, const double *, double *);
void mattri(const int *, const double *, double *);
void mutrma(const int *, const int *, const double *, const double *, double *);

void jacobiC(const int *, double *, double *, double *, double *, const int *,
             const double *);

void smacofDistC(const double *, const int *, const int *, double *);
void smacofLossC(const double *, const double *, const double *, const int *,
                 double *);
```

```
void smacofBmatC(const double *, const double *, const double *, const int *,
                 double *);
void smacofVmatC(const double *, const int *, double *);
void smacofGuttmanC(const double *, const double *, const double *, const int *,
                    const int *, double *, double *);
void smacofGradientC(const double *, const double *, const double *,
                     const int *, const int *, double *, double *);
void smacofHmatC(const double *, const double *, const double *, const double *,
                 const double *, const double *, const int *, const int *,
                 double *, double *);
void smacofHessianC(const double *, const double *, const double *,
                    const double *, const double *, const double *, const int *,
                    const int *, double *, double *);
void smacofInitialC(const double *, const int *, const int *, double *,
                    double *, double *, double *, double *);

static inline int VINDEX(const int i) { return i - 1; }

static inline int MINDEX(const int i, const int j, const int n) {
    return (i - 1) + (j - 1) * n;
}

static inline int AINDEX(const int i, const int j, const int k, const int n,
                         const int m) {
    return (i - 1) + (j - 1) * n + (k - 1) * n * m;
}

static inline int SINDEX(const int i, const int j, const int n) {
    return ((j - 1) * n) - (j * (j - 1) / 2) + (i - j) - 1;
}

static inline int TINDEX(const int i, const int j, const int n) {
    return ((j - 1) * n) - ((j - 1) * (j - 2) / 2) + (i - (j - 1)) - 1;
}

static inline double SQUARE(const double x) { return x * x; }
static inline double THIRD(const double x) { return x * x * x; }
static inline double FOURTH(const double x) { return x * x * x * x; }
static inline double FIFTH(const double x) { return x * x * x * x * x; }

static inline double MAX(const double x, const double y) {
    return (x > y) ? x : y;
}
```

```
static inline double MIN(const double x, const double y) {
    return (x < y) ? x : y;
}

static inline int IMAX(const int x, const int y) { return (x > y) ? x : y; }

static inline int IMIN(const int x, const int y) { return (x < y) ? x : y; }

static inline int IMOD(const int x, const int y) {
    return (((x % y) == 0) ? y : (x % y));
}

#endif /* SMACOF_H */
```

5.3.2 smacof.c

```
#include "smacof.h"

void smacofDistC(const double *x, const int *n, const int *p, double *d) {
    int k = 1, nn = *n, pp = *p;
    for (int j = 1; j <= nn - 1; j++) {
        for (int i = j + 1; i <= nn; i++) {
            double dij = 0.0;
            for (int s = 1; s <= pp; s++) {
                dij += SQUARE(x[MINDEX(i, s, nn)] - x[MINDEX(j, s, nn)]);
            }
            d[VINDEX(k)] = sqrt(dij);
            k++;
        }
    }
}

void smacofLossC(const double *dist, const double *w, const double *delta,
                 const int *m, double *stress) {
    int mm = *m;
    *stress = 0.0;
    for (int k = 1; k <= mm; k++) {
        *stress += w[VINDEX(k)] * SQUARE(delta[VINDEX(k)] - dist[VINDEX(k)]);
    }
    *stress /= 2.0;
    return;
}
```

```
void smacofBmatC(const double *dist, const double *w, const double *delta,
                 const int *n, double *b) {
    int nn = *n;
    for (int i = 1; i <= nn; i++) {
        b[TINDEX(i, i, nn)] = 0.0;
    }
    for (int j = 1; j <= nn - 1; j++) {
        for (int i = j + 1; i <= nn; i++) {
            int k = SINDEX(i, j, nn);
            double dinv = (dist[k] == 0.0) ? 0.0 : 1.0 / dist[k];
            double elem = w[k] * delta[k] * dinv;
            b[TINDEX(i, j, nn)] = -elem;
            b[TINDEX(i, i, nn)] += elem;
            b[TINDEX(j, j, nn)] += elem;
        }
    }
    return;
}

void smacofVmatC(const double *w, const int *n, double *v) {
    int nn = *n;
    for (int i = 1; i <= nn; i++) {
        v[TINDEX(i, i, nn)] = 0.0;
    }
    for (int j = 1; j <= nn - 1; j++) {
        for (int i = j + 1; i <= nn; i++) {
            double elem = w[SINDEX(i, j, nn)];
            v[TINDEX(i, j, nn)] = -elem;
            v[TINDEX(i, i, nn)] += elem;
            v[TINDEX(j, j, nn)] += elem;
        }
    }
    return;
}

void smacofHmatC(const double *x, const double *bmat, const double *vmat,
                 const double *w, const double *delta, const double *dist,
                 const int *n, const int *p, double *work, double *h) {
    int nn = *n, pp = *p, np = nn * pp;
    for (int s = 1; s <= pp; s++) {
        for (int t = 1; t <= s; t++) {
            for (int j = 1; j <= nn; j++) {
                for (int i = j; i <= nn; i++) {
                    work[TINDEX(i, j, nn)] = 0.0;
```

```
                }
            }
            for (int j = 1; j <= nn - 1; j++) {
                for (int i = j + 1; i <= nn; i++) {
                    double f1 = (x[MINDEX(i, s, nn)] - x[MINDEX(j, s, nn)]);
                    double f2 = (x[MINDEX(i, t, nn)] - x[MINDEX(j, t, nn)]);
                    double f3 = THIRD(dist[SINDEX(i, j, nn)]);
                    double f4 = delta[SINDEX(i, j, nn)] * w[SINDEX(i, j, nn)];
                    f3 = (f3 < 1e-10) ? 0 : 1 / f3;
                    double elem = f1 * f2 * f4 * f3;
                    work[TINDEX(i, j, nn)] = -elem;
                    work[TINDEX(i, i, nn)] += elem;
                    work[TINDEX(j, j, nn)] += elem;
                }
            }
            if (s == t) {
                for (int j = 1; j <= nn; j++) {
                    for (int i = j; i <= nn; i++) {
                        h[TINDEX((s - 1) * nn + i, (s - 1) * nn + j, nn * pp)] =
                            bmat[TINDEX(i, j, nn)] - work[TINDEX(i, j, nn)];
                    }
                }
            }
            if (s != t) {
                for (int i = 1; i <= nn; i++) {
                    for (int j = 1; j <= nn; j++) {
                        int ij = IMIN(i, j);
                        int ji = IMAX(i, j);
                        int sj = (s - 1) * nn + j;
                        int si = (t - 1) * nn + i;
                        h[TINDEX(sj, si, np)] = -work[TINDEX(ji, ij, nn)];
                    }
                }
            }
        }
    }
    return;
}

void smacofGuttmanC(const double *x, const double *bmat, const double *vinv,
                    const int *n, const int *p, double *work, double *y) {
    (void)mutrma(n, p, bmat, x, work);
    (void)mutrma(n, p, vinv, work, y);
    return;
```

```
}

void smacofGradientC(const double *x, const double *bmat, const double *vmat,
                     const int *n, const int *p, double *work, double *y) {
    int nn = *n, pp = *p;
    (void)mutrma(n, p, vmat, x, y);
    (void)mutrma(n, p, bmat, x, work);
    for (int i = 1; i <= nn; i++) {
        for (int s = 1; s <= pp; s++) {
            y[MINDEX(i, s, nn)] -= work[MINDEX(i, s, nn)];
        }
    }
    return;
}

void smacofHessianC(const double *x, const double *bmat, const double *vmat,
                    const double *w, const double *delta, const double *dist,
                    const int *n, const int *p, double *work, double *h) {
    int nn = *n, pp = *p, np = nn * pp;
    (void)smacofHmatC(x, bmat, vmat, w, delta, dist, n, p, work, h);
    for (int j = 1; j <= np; j++) {
        for (int i = j; i <= np; i++) {
            h[TINDEX(i, j, np)] = -h[TINDEX(i, j, np)];
        }
    }
    for (int s = 1; s <= pp; s++) {
        for (int j = 1; j <= nn; j++) {
            for (int i = j; i <= nn; i++) {
                h[TINDEX((s - 1) * nn + i, (s - 1) * nn + j, np)] +=
                    vmat[TINDEX(i, j, nn)];
            }
        }
    }
    return;
}

void smacofNewtonC() { return; }

void smacofInitialC(const double *delta, const int *n, const int *p,
                    double *work1, double *work2, double *work3, double *work4,
                    double *x) {
    int nn = *n, pp = *p, itmax = 100;
    double s, ss = 0.0, eps = 1e-6;
    for (int i = 1; i <= nn; i++) {
```

```
        work1[VINDEX(i)] = 0.0;
        for (int j = 1; j <= nn; j++) {
            if (i == j) continue;
            int ij = IMAX(i, j);
            int ji = IMIN(i, j);
            s = SQUARE(delta[SINDEX(ij, ji, nn)]);
            ss += s;
            work1[VINDEX(i)] += s;
            if (j < i) continue;
            work2[TINDEX(ij, ji, nn)] = s;
        }
        work1[VINDEX(i)] /= (double)nn;
    }
    ss /= SQUARE((double)nn);
    for (int j = 1; j <= nn; j++) {
        for (int i = j; i <= nn; i++) {
            work2[TINDEX(i, j, nn)] -= work1[VINDEX(i)];
            work2[TINDEX(i, j, nn)] -= work1[VINDEX(j)];
            work2[TINDEX(i, j, nn)] += ss;
            work2[TINDEX(i, j, nn)] *= -0.5;
        }
    }
    (void)jacobiC(n, work2, work3, work1, work4, &itmax, &eps);
    for (int i = 1; i <= nn; i++) {
        for (int j = 1; j <= pp; j++) {
            s = work2[TINDEX(j, j, nn)];
            if (s <= 0) continue;
            x[MINDEX(i, j, nn)] = work3[MINDEX(i, j, nn)] * sqrt(s);
        }
    }
    return;
}

void smacofUpdateC(const double *xold, const double *w, const double *delta,
                   const double *vinv, const int *n, const int *p, double *dnew,
                   double *bmat, double *work, double *snew, double *xnew) {
    int nn = *n, pp = *p, mm = nn * (nn - 1) / 2;
    (void)mutrma(n, p, bmat, xold, work);
    (void)mutrma(n, p, vinv, work, xnew);
    for (int j = 1; j <= nn - 1; j++) {
        for (int i = j + 1; i <= nn; i++) {
            double dij = 0.0;
            for (int s = 1; s <= pp; s++) {
                dij += SQUARE(xnew[MINDEX(i, s, nn)] - xnew[MINDEX(j, s, nn)]);
```

```
            }
            dnew[SINDEX(i, j, nn)] = sqrt(dij);
        }
    }
    for (int i = 1; i <= nn; i++) {
        bmat[TINDEX(i, i, nn)] = 0.0;
    }
    for (int j = 1; j <= nn - 1; j++) {
        for (int i = j + 1; i <= nn; i++) {
            int k = SINDEX(i, j, nn);
            double dinv = (dnew[k] == 0.0) ? 0.0 : 1.0 / dnew[k];
            double elem = w[k] * delta[k] * dinv;
            bmat[TINDEX(i, j, nn)] = -elem;
            bmat[TINDEX(i, i, nn)] += elem;
            bmat[TINDEX(j, j, nn)] += elem;
        }
    }
    *snew = 0.0;
    for (int k = 1; k <= mm; k++) {
        *snew += w[VINDEX(k)] * SQUARE(delta[VINDEX(k)] - dnew[VINDEX(k)]);
    }
    *snew /= 2.0;
}
```

5.3.3 utils.c

```
#include "smacof.h"

// complete set of orthogonal polynomials

void dorpol(const int *n, double *p) {
    for (int i = 1; i <= *n; i++) {
        p[MINDEX(i, 1, *n)] = 1.0;
        double di = (double)i;
        for (int j = 2; j <= *n; j++) {
            p[MINDEX(i, j, *n)] = p[MINDEX(i, j - 1, *n)] * di;
        }
    }
    (void)dortho(n, n, p);
    return;
}

// double centering
```

```
void docent(const int *n, double *x, double *y) {
    int nn = *n;
    double *ssum = (double *)calloc((size_t)nn, sizeof(double)), t = 0.0;
    for (int i = 1; i <= nn; i++) {
        double s = 0.0;
        for (int j = 1; j <= nn; j++) {
            int ij = IMAX(i, j);
            int ji = IMIN(j, i);
            s += x[TINDEX(ij, ji, nn)];
            t += x[TINDEX(ij, ji, nn)];
        }
        ssum[VINDEX(i)] = s / ((double)nn);
    }
    t /= (double)SQUARE(nn);
    for (int j = 1; j <= nn; j++) {
        for (int i = j; i <= nn; i++) {
            y[TINDEX(i, j, nn)] =
                -(x[TINDEX(i, j, nn)] - ssum[VINDEX(i)] - ssum[VINDEX(j)] + t) /
                2.0;
        }
    }
    free(ssum);
    return;
}

// print a general matrix

void primat(const int *n, const int *m, const int *w, const int *p,
            const double *x) {
    for (int i = 1; i <= *n; i++) {
        for (int j = 1; j <= *m; j++) {
            printf(" %+*.*f ", *w, *p, x[MINDEX(i, j, *n)]);
        }
        printf("\n");
    }
    printf("\n\n");
    return;
}

// print strict lower triangle

void pritrl(const int *n, const int *w, const int *p, const double *x) {
    for (int i = 1; i <= *n; i++) {
        for (int j = 1; j <= i; j++) {
```

```
            if (i == j) {
                for (int k = 1; k <= *w + 2; k++) {
                    printf("%c", '*');
                }
            }
            if (i > j) {
                printf(" %+*.*f ", *w, *p, x[SINDEX(i, j, *n)]);
            }
        }
        printf("\n");
    }
    printf("\n\n");
    return;
}

// print inclusive lower triangle

void pritru(const int *n, const int *w, const int *p, const double *x) {
    for (int i = 1; i <= *n; i++) {
        for (int j = 1; j <= i; j++) {
            printf(" %+*.*f ", *w, *p, x[TINDEX(i, j, *n)]);
        }
        printf("\n");
    }
    printf("\n\n");
    return;
}

// print general array

void priarr(const int *n, const int *m, const int *r, const int *w,
            const int *p, const double *x) {
    for (int k = 1; k <= *r; k++) {
        for (int i = 1; i <= *n; i++) {
            for (int j = 1; j <= *m; j++) {
                printf(" %+*.*f ", *w, *p, x[AINDEX(i, j, k, *n, *m)]);
            }
            printf("\n");
        }
        printf("\n\n");
    }
    printf("\n\n\n");
    return;
}
```

```c
// arbitrary power of a matrix

void mpower(const int *n, double *x, double *power, double *xpow) {
    int nn = *n, itmax = 100;
    double eps = 1e-6;
    double *e = (double *)calloc((size_t)SQUARE(nn), sizeof(double));
    double *oldi = (double *)calloc((size_t)nn, sizeof(double));
    double *oldj = (double *)calloc((size_t)nn, sizeof(double));
    (void)jacobiC(n, x, e, oldi, oldj, &itmax, &eps);
    int k = 1;
    for (int i = 1; i <= nn; i++) {
        double s = x[VINDEX(k)];
        oldi[VINDEX(i)] = (s > 1e-10) ? pow(s, *power) : 0.0;
        k += (nn - (i - 1));
    }
    for (int j = 1; j <= nn; j++) {
        for (int i = j; i <= nn; i++) {
            double s = 0.0;
            for (int k = 1; k <= nn; k++) {
                s +=
                    e[MINDEX(i, k, nn)] * e[MINDEX(j, k, nn)] * oldi[VINDEX(k)];
            }
            xpow[TINDEX(i, j, nn)] = s;
        }
    }
    free(e);
    free(oldi);
    free(oldj);
    return;
}

// inclusive lower triangle to symmetric

void trimat(const int *n, const double *x, double *y) {
    int nn = *n;
    for (int i = 1; i <= nn; i++) {
        for (int j = 1; j <= nn; j++) {
            y[MINDEX(i, j, nn)] =
                (i >= j) ? x[TINDEX(i, j, nn)] : x[TINDEX(j, i, nn)];
        }
    }
    return;
}
```

```
// symmetric to inclusive lower triangular

void mattri(const int *n, const double *x, double *y) {
    int nn = *n;
    for (int j = 1; j <= nn; j++) {
        for (int i = j; i <= nn; i++) {
            y[TINDEX(i, j, nn)] = x[MINDEX(i, j, nn)];
        }
    }
    return;
}

// premultiply matrix by symmetric matrix in inclusive triangular storage

void mutrma(const int *n, const int *m, const double *a, const double *x,
            double *y) {
    int nn = *n, mm = *m;
    for (int i = 1; i <= nn; i++) {
        for (int j = 1; j <= mm; j++) {
            double s = 0.0;
            for (int k = 1; k <= nn; k++) {
                int ik = IMAX(i, k);
                int ki = IMIN(i, k);
                s += a[TINDEX(ik, ki, nn)] * x[MINDEX(k, j, nn)];
            }
            y[MINDEX(i, j, nn)] = s;
        }
    }
}

// direct sum of two matrices -- can be used recursively

void dirsum(const int *n, const int *m, const double *a, const double *b,
            double *c) {
    int nn = *n, mm = *m, mn = nn + mm;
    for (int j = 1; j <= nn; j++) {
        for (int i = j; i <= nn; i++) {
            c[TINDEX(i, j, mn)] = a[TINDEX(i, j, nn)];
        }
    }
    for (int j = 1; j <= mm; j++) {
        for (int i = j; i <= mm; i++) {
            c[TINDEX(nn + i, nn + j, mn)] = b[TINDEX(i, j, mm)];
        }
```

```
    }
    return;
}
```

5.3.4 jacobi.c

```
#include "smacof.h"

void jacobiC(const int *nn, double *a, double *evec, double *oldi, double *oldj,
             const int *itmax, const double *eps) {
    int n = *nn, itel = 1;
    double d = 0.0, s = 0.0, t = 0.0, u = 0.0, v = 0.0, p = 0.0, q = 0.0,
           r = 0.0;
    double fold = 0.0, fnew = 0.0;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            evec[MINDEX(i, j, n)] = (i == j) ? 1.0 : 0.0;
        }
    }
    for (int i = 1; i <= n; i++) {
        fold += SQUARE(a[TINDEX(i, i, n)]);
    }
    while (true) {
        for (int j = 1; j <= n - 1; j++) {
            for (int i = j + 1; i <= n; i++) {
                p = a[TINDEX(i, j, n)];
                q = a[TINDEX(i, i, n)];
                r = a[TINDEX(j, j, n)];
                if (fabs(p) < 1e-10) continue;
                d = (q - r) / 2.0;
                s = (p < 0) ? -1.0 : 1.0;
                t = -d / sqrt(SQUARE(d) + SQUARE(p));
                u = sqrt((1 + t) / 2);
                v = s * sqrt((1 - t) / 2);
                for (int k = 1; k <= n; k++) {
                    int ik = IMIN(i, k);
                    int ki = IMAX(i, k);
                    int jk = IMIN(j, k);
                    int kj = IMAX(j, k);
                    oldi[VINDEX(k)] = a[TINDEX(ki, ik, n)];
                    oldj[VINDEX(k)] = a[TINDEX(kj, jk, n)];
                }
                for (int k = 1; k <= n; k++) {
```

```
                    int ik = IMIN(i, k);
                    int ki = IMAX(i, k);
                    int jk = IMIN(j, k);
                    int kj = IMAX(j, k);
                    a[TINDEX(ki, ik, n)] =
                        u * oldi[VINDEX(k)] - v * oldj[VINDEX(k)];
                    a[TINDEX(kj, jk, n)] =
                        v * oldi[VINDEX(k)] + u * oldj[VINDEX(k)];
                }
                for (int k = 1; k <= n; k++) {
                    oldi[VINDEX(k)] = evec[MINDEX(k, i, n)];
                    oldj[VINDEX(k)] = evec[MINDEX(k, j, n)];
                    evec[MINDEX(k, i, n)] =
                        u * oldi[VINDEX(k)] - v * oldj[VINDEX(k)];
                    evec[MINDEX(k, j, n)] =
                        v * oldi[VINDEX(k)] + u * oldj[VINDEX(k)];
                }
                a[TINDEX(i, i, n)] =
                    SQUARE(u) * q + SQUARE(v) * r - 2 * u * v * p;
                a[TINDEX(j, j, n)] =
                    SQUARE(v) * q + SQUARE(u) * r + 2 * u * v * p;
                a[TINDEX(i, j, n)] =
                    u * v * (q - r) + (SQUARE(u) - SQUARE(v)) * p;
            }
        }
        fnew = 0.0;
        for (int i = 1; i <= n; i++) {
            fnew += SQUARE(a[TINDEX(i, i, n)]);
        }
        if (((fnew - fold) < *eps) || (itel == *itmax)) break;
        fold = fnew;
        itel++;
    }
    return;
}
```

5.3.5 lapacke.c

```
#include "smacof.h"

void dposv(const int *n, const int *m, double *a, double *b) {
    lapack_int nn = (lapack_int)*n, mm = (lapack_int)*m;
    (void)LAPACKE_dposv(LAPACK_COL_MAJOR, 'U', nn, mm, a, nn, b, nn);
```

```
    return;
}

void dsyevd(const int *n, double *a, double *x) {
    lapack_int nn = (lapack_int)*n;
    (void)LAPACKE_dsyevd(LAPACK_COL_MAJOR, 'V', 'U', nn, a, nn, x);
    return;
}

void dgeqrf(const int *n, const int *m, double *a, double *tau) {
    lapack_int nn = (lapack_int)*n, mm = (lapack_int)*m;
    (void)LAPACKE_dgeqrf(LAPACK_COL_MAJOR, nn, mm, a, nn, tau);
    return;
}

void dorgqr(const int *n, const int *m, double *a, double *tau) {
    lapack_int nn = (lapack_int)*n, mm = (lapack_int)*m;
    (void)LAPACKE_dorgqr(LAPACK_COL_MAJOR, nn, mm, mm, a, nn, tau);
    return;
}

void dortho(const int *n, const int *m, double *a) {
    lapack_int nn = (lapack_int)*n, mm = (lapack_int)*m;
    double *tau = calloc((size_t)mm, sizeof(double));
    (void)LAPACKE_dgeqrf(LAPACK_COL_MAJOR, nn, mm, a, nn, tau);
    (void)LAPACKE_dorgqr(LAPACK_COL_MAJOR, nn, mm, mm, a, nn, tau);
    free(tau);
    return;
}
```

References

Author. 2016. "The LAPACKE C Interface to LAPACK." Accessed April 28. http://www.netlib.org/lapack/lapacke.html.

De Leeuw, J. 1977. "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. http://www.stat.ucla.edu/~deleeuw/janspubs/1977/chapters/deleeuw_C_77.pdf.

———. 1984. "Differentiability of Kruskal's Stress at a Local Minimum." *Psychometrika* 49: 111–13. http://www.stat.ucla.edu/~deleeuw/janspubs/1984/articles/deleeuw_A_84f.pdf.

———. 1988. "Convergence of the Majorization Method for Multidimensional Scaling."

Journal of Classification 5: 163–80. http://www.stat.ucla.edu/~deleeuw/janspubs/1988/articles/deleeuw_A_88b.pdf.

———. 2014. "Bounding, and Sometimes Finding, the Global Minimum in Multidimensional Scaling." UCLA Department of Statistics. doi:10.13140/RG.2.1.5068.4248.

De Leeuw, J., and W. J. Heiser. 1980. "Multidimensional Scaling with Restrictions on the Configuration." In *Multivariate Analysis, Volume V*, edited by P.R. Krishnaiah, 501–22. Amsterdam, The Netherlands: North Holland Publishing Company. http://www.stat.ucla.edu/~deleeuw/janspubs/1980/chapters/deleeuw_heiser_C_80.pdf.

De Leeuw, J., and W.J. Heiser. 1977. "Convergence of Correction Matrix Algorithms for Multidimensional Scaling." In *Geometric Representations of Relational Data*, edited by J.C. Lingoes, 735–53. Ann Arbor, Michigan: Mathesis Press. http://www.stat.ucla.edu/~deleeuw/janspubs/1977/chapters/deleeuw_heiser_C_77.pdf.

De Leeuw, J., and P. Mair. 2009. "Multidimensional Scaling Using Majorization: SMACOF in R." *Journal of Statistical Software* 31 (3): 1–30. http://www.stat.ucla.edu/~deleeuw/janspubs/2009/articles/deleeuw_mair_A_09c.pdf.

Gerber, F., K. Moesinger, and R. Furrer. 2016. "dotCall64: An efficient interface to compiled C/C++ and Fortran code supporting long vectors." *R journal (submitted)*.

———. 2017. "Extending R packages to support 64-bit compiled code: An illustration with spam64 and GIMMS NDVI3g data." *Computers & Geosciences* 104: 109–19.

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