Convergence Rate of ELEGANT Algorithms

Jan de Leeuw

Version 20, December 02, 2016

Abstract

We study the convergence rate of the ELEGANT algorithm for squared distance scaling by using both observed convergence rates and an analytical expression for the derivative of the algorithmic map.

Contents

Note: This is a working paper which will be expanded/updated frequently. All suggestions for improvement are welcome. The directory deleeuwpdx.net/pubfolders/speed has a pdf version, the complete Rmd file with all code chunks, the bib file, and the R source code.

1 Introduction

The multidimensional scaling (MDS) loss function sstress is defined on the set $\mathbb{R}^{n\times p}$ of $n \times p$ *configuration* matrices as

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

where $\Delta = \{\delta_{ij}\}$ and $W = \{w_{ij}\}$ are symmetric, non-negative, and hollow matrices of *dissimilarities* and *weights* (a matrix is *hollow* if its diagonal is zero). The $d_{ij}^2(X)$ in (1) are squared Euclidean distances, defined as

$$d_{ij}^2(X) := (x_i - x_j)'(x_i - x_j) = \mathbf{tr}\ X'A_{ij}X = \mathbf{tr}\ A_{ij}C = c_{ii} + c_{jj} - 2c_{ij}, \tag{2}$$

where $C = XX'$ and $A_{ij} := (u_i - u_j)(u_i - u_j)'$, with the u_i unit vectors (element i is +1, the other elements are zero).

The majorization method to minimize sstress (1) computes the update of the configuration matrix in iteration k using the iteration function

$$\Phi(X) := \Gamma_p(B_\beta(X)). \tag{3}$$

Here Γ_p computes the $n \times p$ matrix whose row-wise cross-product provides a best least squares rank p approximation of its argument. Thus, with **SSQ** for the (unweighted) sum of squares,

$$\Gamma_p(B) := \mathop{\mathbf{argmin}}_{X \in \mathbb{R}^{n\times p}} \mathbf{SSQ}\ (B - XX').$$

The stationary equation for this minimization problem are $BX = X(X'X)$. Suppose $B = K\Lambda K'$ is a complete eigen decomposition of B, and suppose the p largest eigenvalues are non-negative. Collect them in the diagonal matrix Λ_p, with the corresponding p eigenvectors in K_p. Then $K_p\Lambda_p^{\frac{1}{2}}$ is a minimizer of $\mathbf{SSQ}(B - XX')$. It is unique, up to rotation, if $\lambda_p > \lambda_{p+1}$.

The function B_β is defined, using a step-size or relaxation type parameter β, by

$$B_\beta(X) := XX' + \frac{1}{\beta}R(X), \tag{4}$$

$$R(X) := \sum_{i=1}^n \sum_{j=1}^n w_{ij}(\delta_{ij}^2 - d_{ij}^2(X))A_{ij}. \tag{5}$$

Note that the rank of $\Phi(X)$ is equal to the minimum of p and the number of positive eigenvalues of $B(X)$. If X has rank p and β is large enough, then $B(X)$ will also be of rank p. Matrix $B_\beta(X)$ is invariant under rotation of X. We have $\mathcal{D}\sigma(X) = -2R(X)X$ and consequently X is a stationary point of σ if and only if $B(X)X = X(X'X)$, i.e. if and only if X is a fixed point of Φ. Also note that if X is centered (columnwise) then $B(X)$ is doubly-centered.

We call (3) the *ELEGANT transform*, because it derives from the ELEGANT algorithm to minimize sstress (De Leeuw (1975)). The ELEGANT iteration is

$$X^{(k+1)} = \Phi(X^{(k)}) = \Gamma_p(B(X^{(k)})). \tag{6}$$

For a modernized and improved presentation of ELEGANT we refer to De Leeuw, Groenen, and Pietersz (2016). In the original (unweighted) ELEGANT algorithm $\beta = 4n^2$, in the improved majorization algorithm of De Leeuw, Groenen, and Pietersz (2016) it is $\beta = 4n$.

2 Derivatives of the ELEGANT transform

2.1 Derivatives of Gamma

We start with the definition $\Gamma_p(B) = K_p\Lambda_p^{\frac12}$, where that $B = K\Lambda K'$ is the complete eigen decomposition. Assume all λ_t are different, and define $\Xi := K'EK$. Using results from, for example, De Leeuw (2008), we see that

$$\begin{aligned}\lambda_s^{\frac12}(B+E) &= \lambda_s^{\frac12} + \frac12\lambda_s^{\frac12}\xi_{ss} + o(\|E\|),\\ k_s(B+E) &= k_s - (B-\lambda_s I)^+ E k_s + o(\|E\|).\end{aligned}$$

Thus column s of $Y := \mathcal{D}\Gamma(X)(E)$ is

$$y_s = \frac12\lambda_s^{-\frac12}\xi_{ss}k_s - \lambda_s^{\frac12}\sum_{\substack{t=1\\t\neq s}}^{n}\frac{1}{\lambda_t-\lambda_s}\xi_{st}k_t \tag{7}$$

The formula is implemented in the R function `map_dgamma`. We can check equation (7) by computing numerical derivatives, using the function `jacobian` from the numDeriv package (Gilbert and Varadhan (2016)).

```
set.seed(12345)
b <- crossprod (matrix (rnorm(400), 100, 4)) / 100
b <- doubleCenter (b)
e <- aij (2, 3, 4)
func <- function (z, b, e)
  return (map_gamma (b + z * e, p = 2))
mprint (matrix (jacobian (func, 0, b = b, e = e), 4, 2))
```

```
##       [,1]             [,2]
## [1,]   -1.6998187769   -0.1816313139
## [2,]    3.0764177033    0.9917597934
## [3,]   -0.7816264815    1.8727796964
## [4,]   -0.5949724449   -2.6829081760
```

```
mprint (map_dgamma (b, e, p = 2))
```

```
##       [,1]             [,2]
## [1,]   -1.6998187769    0.1816313139
## [2,]    3.0764177033   -0.9917597934
## [3,]   -0.7816264815   -1.8727796964
## [4,]   -0.5949724449    2.6829081759
```

2.2 Derivatives of B

The ELEGANT transform is a composition of the map Γ_p and the map B_β. Because

$$B_\beta(X+E)=B_\beta(X)+XE'+EX'-\frac{2}{\beta}\sum_{i=1}^n\sum_{j=1}^n w_{ij}(\mathbf{tr}\ X'A_{ij}E)A_{ij}+o(\|E\|),$$

we find

$$\mathcal{D}B_\beta(X)(E)=XE'+EX'-\frac{2}{\beta}\sum_{i=1}^n\sum_{j=1}^n w_{ij}(x_i-x_j)'(e_i-e_j)A_{ij} \tag{8}$$

Note that this derivative does not depend on the δ_{ij}. Also note that, because B_β is invariant under rotation, if $E=XS$ with S anti-symmetric, then $\mathcal{D}B_\beta(X)(E)=0$.

Again, we check the formula implemented in the function `map_db` numerically using `jacobian`.

```r
set.seed(12345)
delta <- 1 - diag (4)
a <- center (matrix (rnorm (8), 4, 2))
b <- center (matrix (rnorm (8), 4, 2))
func <- function (z, a, b) {
    return (map_b (delta, x = a + z * b, w = 1 - diag (4), beta = 16))
}
mprint (matrix (jacobian (func, 0, a = a, b = b), 4, 4))
```

```
##       [,1]            [,2]            [,3]            [,4]
## [1,]    0.6771742494   -0.7255711172   -0.7145306231    0.7629274909
## [2,]   -0.7255711172   -0.2659852113    1.0607440880   -0.0691877594
## [3,]   -0.7145306231    1.0607440880   -0.0318313857   -0.3143820791
## [4,]    0.7629274909   -0.0691877594   -0.3143820791   -0.3793576523
```

```r
mprint (map_db (a, b, w = delta, beta = 16))
```

```
##       [,1]            [,2]            [,3]            [,4]
## [1,]    0.6771742494   -0.7255711172   -0.7145306231    0.7629274909
## [2,]   -0.7255711172   -0.2659852113    1.0607440880   -0.0691877594
## [3,]   -0.7145306231    1.0607440880   -0.0318313857   -0.3143820791
## [4,]    0.7629274909   -0.0691877594   -0.3143820791   -0.3793576523
```

2.3 Chain Rule

We can combine (7) and (8) using the chain rule

$$\Phi(X+E)=\Phi(X)+\mathcal{D}\Gamma_p(B_\beta(X))(\mathcal{D}B_\beta(X)(E))+o(\|E\|), \tag{9}$$

where for computational purposes we substitute (7) and (8) in (9).

Formula (9) is implemented in `map_dphi` and can be tested with `jacobian`.

```
set.seed (12345)
delta <- 1 - diag (4)
x <- center (matrix (rnorm (8), 4, 2)) / 10
y <- center (matrix (rnorm (8), 4, 2)) / 10
mm <- matrixBasis (4, 2)
aa <- drop (as.vector (x) %*% mm)
bb <- drop (as.vector (y) %*% mm)
func <- function (x, a, b) {
  return (map_phi (delta, a + x * b, w = 1 - diag (4), p = 2, beta = 16, basis = matrixI
}
mprint (matrix (mm %*% jacobian (func, 0, a = aa, b = bb), 4, 2))
```

```
##      [,1]          [,2]
## [1,]  0.2360189702 -0.0949984808
## [2,] -0.0241459647 -0.0452932309
## [3,] -0.2355205805  0.2879923869
## [4,]  0.0236475749 -0.1477006751
```

```
mprint (matrix (mm %*% map_dphi (delta, aa, bb), 4, 2))
```

```
##      [,1]          [,2]
## [1,] -0.2360189702  0.0949984810
## [2,]  0.0241459646  0.0452932309
## [3,]  0.2355205805 -0.2879923873
## [4,] -0.0236475748  0.1477006754
```

3 Computing Convergence Rates

The obvious way to measure the rate of convergence of iterative methods with convergence order one (i.e. linear convergence, or convergence at the rate of a geometric progression) is to compute the sequence

$$\eta_Q^{(k)}=\frac{\|X^{(k)}-X^{(k-1)}\|}{\|X^{(k-1)}-X^{(k-2)}\|},$$

and then estimate $\lim_{k\rightarrow\infty}\eta_Q^{(k)}$. Unfortunately in practice this may not be as easy as it looks. The sequence $\{\eta_Q^{(k)}\}$ depends on the norm chosen, but also on the sequence $\{X^{(k)}\}$. Sequence $\{\eta_Q^{(k)}\}$ may not converge at all, or have more than one converging subsequence. Different sequences converging to the same limit point X_∞, for instance from different random starts, will lead to different sequences $\{\eta_Q^{(k)}\}$, possibly with different limits. Computation of $\eta_Q^{(k)}$ in later iterations is numerically problematic, because it requires us to compute the ratio of two very small quantities. The version of ELEGANT used in this paper, implemented in the R function `beyond`, computes this estimate of the convergence rate, but if we require to much precision in our MDS solution we do see a lot numerical instability.

The theoretical convergence rate is the spectral radius of the derivative at the solution. i.e. the eigenvalue of maximum modulus of the Jacobian. Thus this theoretical rate is defined only if Γ_p is differentiable at $B(X)$, which requires that the first p eigenvalues of $B(X)$ are different, and $\lambda_p > \lambda_{p+1}$.

3.1 Small Example

Our first example has four objects, with squared dissimilarities

$$\begin{bmatrix}0&1&2&3\\1&0&1&2\\2&1&0&1\\3&2&1&0\end{bmatrix}.$$

```
delta <- abs (outer (1:4, 1:4, "-"))
hs <- beyond(delta, bound = "eval", verbose = FALSE, eps = 1e-6, itmax = 5000)
hg <- beyond(delta, bound = "trace", verbose = FALSE, eps = 1e-6, itmax = 5000)
```

We iterate ELEGANT transforms until the norm of the difference between successive solutions is less than 10^{-6}. The theoretical convergence rate is computed by the function `power`, an ad-hoc version of the power method that iterates eigenvectors as matrices. For the original ELEGANT bound $\beta = 4n^2 = 64$ the observed convergence rate is 0.9204716502, the theoretical convergence rate is 0.9407957901, and we need 87 iterations. For $\beta = 4n = 16$ the observed convergence rate is 0.7598695799, the theoretical convergence rate is 1.0311634416.

We use the function `janJacobian` to compute the Jacobian and its eigenvalues for $\beta = 4n = 16$ and $\beta = 4n^2 = 64$.

```
h1 <- janJacobian (delta, hs$a, beta = 16)
h2 <- janJacobian (delta, hg$a, beta = 64)
ea <- eigen (h1)
eb <- eigen (h2)
mprint (Mod(ea$values))
```

```
## [1]    0.7599223785    0.6225704947    0.6144170594    0.4999996330
## [5]    0.2118440380    0.0000000000
```

```
mprint (Mod(eb$values))
```

```
## [1]    0.9407953252    0.9177247789    0.9089519333    0.8749994492
## [5]    0.8031848002    0.0000000000
```

We can also use the `jacobian` function from the numDeriv package and find the eigenvalues of the numerical Jacobian.

```
func <- function (alpha, beta) {
  return (map_phi (delta, alpha, w = 1 - diag (4), p = 2, beta = beta, basis = matrixBa
}
h1 <- jacobian (func, hs$a, beta = 16)
h2 <- jacobian (func, hg$a, beta = 64)
e1 <- eigen (h1)
e2 <- eigen (h2)
mprint(Mod(e1$values))
```

```
## [1]    0.7599223785    0.6225704948    0.6144170594    0.4999996329
## [5]    0.2118440380    0.0000000000
```

```
mprint(Mod(e2$values))
```

```
## [1]    0.9407953252    0.9177247789    0.9089519333    0.8749994493
## [5]    0.8031848002    0.0000000000
```

3.2 Ekman Example

The second example uses the familiar color data from Ekman (1954). We use four different values of β, compute the solution minimizing sstress, and compute the empirical and theoretical convergence rates at the solution.

```
## bound   728  ||  itel  1172  sstress  3.3187855776  erate  0.9960504503  frate  0.996

## bound    56  ||  itel   136  sstress  3.3187849642  erate  0.9502152593  frate  0.951

## bound    25  ||  itel    64  sstress  3.3187849612  erate  0.8858979427  frate  0.888

## bound    10  ||  itel    24  sstress  3.3187849607  erate  0.6913989976  frate  0.693
```

All four values of β give the same solution. The first two are the original ELEGANT value and the maximum eigenvalue bound of the majorization method of De Leeuw, Groenen, and Pietersz (2016). For these two values the convergence is guaranteed by our theoretical results. The iterations with smaller beta still converge monotonically (i.e. the iterations consistently decrease sstress values). For values of β less than ten convergence becomes unstable. An interesting problem for further study is how low we can generally go.

4 Appendix: Code

4.1 auxilary.R

```
ei <- function (i, n) {
  return (ifelse(i == (1:n), 1, 0))
}

aij <- function (i, j, n) {
```

```r
  u <- ei(i, n) - ei (j, n)
  return (outer (u, u))
}

kdelta <- function (i, j) {
  ifelse (i == j, 1 , 0)
}

mprint <- function (x, d = 10, w = 15) {
  print (noquote (formatC (
    x, di = d, wi = w, fo = "f"
  )))
}

mnorm <- function (x) {
  return (sqrt (sum (x ^ 2)))
}

anorm <- function (x) {
  return (max (abs (x)))
}

basis <- function (i, j, n) {
  s <- sqrt (2) / 2
  a <- matrix (0, n, n)
  if (i == j)
    a[i, i] <- 1
  else {
    a[i, j] <- a[j, i] <- s
  }
  return (a)
}

center <- function (x) {
  return (apply (x, 2, function (z) z - mean (z)))
}

doubleCenter <- function (x) {
  n <- nrow (x)
  j <- diag(n) - (1 / n)
  return (j %*% x %*% j)
}

squareDist <- function (x) {
  d <- diag (x)
  return (outer (d, d, "+") - 2 * x)
}

lowerTrapezoidal <- function (x) {
  n <- nrow (x)
  p <- ncol (x)
  if (p == 1) return (x)
  for (i in 1:(p - 1))
    for (j in (i + 1):p) {
      a <- diag (p)
      y <- x[i, c(i, j)]
      y <- y / sqrt (sum (y ^ 2))
      a[i, c (i, j)] <- c(1, -1) * y
      a[j, c (j, i)] <- y
      x <- x %*% a
    }
  return (x)
}

symmetricFromTriangle <- function (x, lower = TRUE, diagonal = TRUE) {
  k <- length (x)
  if (diagonal)
    n <- (sqrt (1 + 8 * k) - 1) / 2
  else
    n <- (sqrt (1 + 8 * k) + 1) / 2
  if (n != as.integer (n))
    stop ("input error")
  nn <- 1:n
  if (diagonal && lower)
    m <- outer (nn, nn, ">=")
  if (diagonal && (!lower))
    m <- outer (nn, nn, "<=")
  if ((!diagonal) && lower)
    m <- outer (nn, nn, ">")
  if ((!diagonal) && (!lower))
    m <- outer (nn, nn, "<")
  b <- matrix (0, n, n)
  b[m] <- x
  b <- b + t(b)
  if (diagonal)
    diag (b) <- diag(b) / 2
  return (b)
}
```

```r
triangleFromSymmetric <- function (x, lower = TRUE, diagonal = TRUE) {
  n <- ncol (x)
  nn <- 1:n
  if (diagonal && lower)
    m <- outer (nn, nn, ">=")
  if (diagonal && (!lower))
    m <- outer (nn, nn, "<=")
  if ((!diagonal) && lower)
    m <- outer (nn, nn, ">")
  if ((!diagonal) && (!lower))
    m <- outer (nn, nn, "<")
  return (x[m])
}

columnBasis <- function (n) {
  x <- matrix (0, n, n - 1)
  x[outer (1:n, 1:(n - 1), "<=")] <- -1
  x[outer (1:n, 1:(n - 1), function (i, j)
    i - j == 1)] <- 1:(n - 1)
  return (apply (x, 2, function (y)
    y / sqrt (sum (y ^ 2))))
}

matrixBasis <- function (n, p) {
  x <- matrix (0, n * p, p * (n - 1))
  for (j in 1:p) {
    x [((j - 1) * n) + (1:n), ((j - 1) * (n - 1)) + (1:(n - 1))] <-
      columnBasis (n)
  }
  return (x)
}
```

4.2 elegant.R

```r
suppressPackageStartupMessages(library(mgcv, quietly = TRUE))

torgerson <- function (delta, p = 2) {
  z <- slanczos(-doubleCenter(delta / 2), p)
  v <- pmax(z$values, 0)
  return(z$vectors %*% diag(sqrt(v)))
}

beyond <-
```

```r
function (delta,
          w = 1 - diag (nrow (delta)),
          p = 2,
          xold = torgerson (delta, p),
          bound = "eval",
          basis = matrixBasis (nrow(delta), p),
          itmax = 1000,
          eps = 1e-8,
          verbose = TRUE) {
  n <- nrow (delta)
  itel <- 1
  vv <- matrix (0, n ^ 2, n ^ 2)
  if (is.numeric (bound))
    lbd <- bound
  if (bound == "eval") {
    for (i in 1:n)
      for (j in 1:n)
        vv <-
          vv + w[i, j] * kronecker (aij (i, j, n), aij (i, j, n))
      lbd <- slanczos(vv, 1)$values
  }
  if (bound == "trace")
    lbd <- 4 * sum (w)
  eold <- Inf
  cold <- tcrossprod (xold)
  dold <- squareDist (cold)
  sold <- sum (w * (delta - dold) ^ 2)
  aold <- drop (as.vector (xold) %*% basis)
  repeat {
    anew <- map_phi (delta, aold, w, p, lbd, basis)
    xnew <- matrix (basis %*% anew, n, p)
    enew <- mnorm (anew - aold)
    rnew <- enew / eold
    cnew <- tcrossprod (xnew)
    dnew <- squareDist (cnew)
    snew <- sum (w * (delta - dnew) ^ 2)
    if (verbose) {
      cat (
        formatC (itel, width = 4, format = "d"),
        formatC (
          sold,
          digits = 10,
          width = 15,
          format = "f"
        ),
        formatC (
          snew,
          digits = 10,
          width = 15,
          format = "f"
        ),
        formatC (
          enew,
          digits = 10,
          width = 15,
          format = "f"
        ),
        formatC (
          rnew,
          digits = 10,
          width = 15,
          format = "f"
        ),
        "\n"
      )
    }
    if ((itel == itmax) || (enew < eps))
      break
    itel <- itel + 1
    xold <- xnew
    aold <- anew
    dold <- dnew
    cold <- cnew
    eold <- enew
    sold <- snew
  }
  return (list (
    x = xnew,
    a = anew,
    c = cnew,
    d = dnew,
    bound = lbd,
    itel = itel,
    e = enew,
    r = rnew,
    s = snew
  ))
}
```

4.3 partials.R

```r
map_b <- function (delta, x, w, beta) {
  s <- tcrossprod (x)
  d <- squareDist (s)
  r <- -2 * w * (delta - d)
  diag (r) <- -rowSums (r)
  return (s + r / beta)
}

map_gamma <- function (b, p) {
  e <-  slanczos (b, p)
  ea <- e$values
  ev <- e$vectors
  return (ev[, 1:p] %*% diag (sqrt (pmax (0, ea[1:p]))))
}

map_phi <- function (delta,
                     alpha,
                     w = 1 - diag (nrow (delta)),
                     p = 2,
                     beta = 4 * nrow (delta),
                     basis = matrixBasis (nrow(delta), p)) {
  n <- nrow (delta)
  x <- matrix (basis %*% alpha, n, p)
  b <- map_b (delta, x, w, beta)
  x <- map_gamma (b, p)
  return (drop (as.vector(x) %*% basis))
}

map_db <- function (x, y, w, beta) {
  n <- nrow (x)
  da <- tcrossprod (x, y) + tcrossprod (y, x)
  dd <- matrix (0, n, n)
  for (i in 1:n)
    for (j in 1:n)
      dd <-
    dd + w[i, j] * sum ((x[i,] - x[j,]) * (y[i,] - y[j,])) * aij (i, j, n)
  return (da - 2 * dd / beta)
}

map_dgamma <- function (b, db, p) {
  n <- nrow (b)
  e <- eigen (b)
  l <- e$values
  k <- e$vectors
  xi <- crossprod (k, db %*% k)
  dg <- matrix (0, n, p)
  for (s in 1:p)
  {
    dgs <- (1 / (2 * sqrt (l[s]))) * xi[s, s] * k[, s]
    for (t in 1:n) {
      if (t == s)
        next
      dgs <- dgs - (sqrt (l[s]) / (l[t] - l[s])) * xi[s, t] * k[, t]
    }
    dg[, s] <- dgs
  }
  return (dg)
}

map_dphi <-
  function (delta,
            alpha,
            epsilon,
            w = 1 - diag (nrow (delta)),
            p = 2,
            beta = 4 * nrow (delta),
            basis = matrixBasis (nrow(delta), p)) {
    n <- nrow (delta)
    x <- matrix (basis %*% alpha, n, p)
    y <- matrix (basis %*% epsilon, n, p)
    b <- map_b (delta, x, w, beta)
    db <- map_db (x, y, w, beta)
    dg <- map_dgamma (b, db, p)
    return (drop (as.vector(dg) %*% basis))
  }


janJacobian <- function (delta,
                         alpha,
                         w = 1 - diag (nrow (delta)),
                         p = 2,
                         beta = 4 * nrow (delta),
                         basis = matrixBasis (nrow(delta), p)) {
  n <- nrow (delta)
  m <- p * (n - 1)
  j <- matrix (0, m, m)
  for (i in 1:m) {
    j[i, ] <- map_dphi (delta, alpha, ei (i, m), w, p, beta, basis)
  }
  return (j)
}

power <-
  function (delta,
            x,
            w = 1 - diag (nrow (x)),
            beta = 4 * nrow (x),
            itmax = 1000,
            eps = 1e-10,
            verbose = FALSE,
            basis = matrixBasis (nrow(x), ncol(x))) {
    n <- nrow (x)
    p <- ncol (x)
    set.seed (12345)
    y <- drop (as.vector (x) %*% basis)
    eold <- ei (1, ncol (basis))
    lold <- -Inf
    itel <- 1
    repeat {
      enew <- map_dphi (delta, y, eold, w, p, beta, basis)
      lnew <- sum (eold * enew)
      enew <- enew / mnorm (enew)
      edif <- 1 - abs (sum (eold * enew))
      if (verbose) {
        cat (
          formatC (itel, width = 4, format = "d"),
          formatC (
            lold,
            digits = 10,
            width = 15,
            format = "f"
          ),
          formatC (
            lnew,
            digits = 10,
            width = 15,
            format = "f"
          ),
          formatC (
            edif,
            digits = 10,
            width = 15,
            format = "f"
          ),
          "\n"
        )
      }
      if ((itel == itmax) || (edif < eps))
        break
      eold <- enew
      lold <- lnew
      itel <- itel + 1
    }
    return (list (e = enew, l = lnew, itel = itel))
  }
```

References

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

———. 2008. "Derivatives of Fixed-Rank Approximations." Preprint Series 547. Los Angeles, CA: UCLA Department of Statistics. http://deleeuwpdx.net/janspubs/2008/reports/deleeuw_R_08b.pdf.

De Leeuw, J., P. Groenen, and R. Pietersz. 2016. "An Alternating Least Squares Approach to Squared Distance Scaling." doi:10.13140/RG.2.2.15357.97766.

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

Gilbert, P., and R. Varadhan. 2016. *numDeriv: Accurate Numerical Derivatives*. https://R-Forge.R-project.org/projects/optimizer/.