Higher Partials of fStress. Who Needs Them ?

Jan de Leeuw

Version 01, August 05, 2017

Abstract

Third

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/third 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 multidimensional scaling (MDS) loss function *fStress* (Groenen, De Leeuw, and Mathar (1995)) is defined as

$$\sigma(x) := \sum_{1\leq i<j\leq n}\sum w_{ij}(\delta_{ij} - f(x'A_{ij}x))^2 \tag{1}$$

for some real-valued f. The w_{ij} are positive weights, the δ_{ij} are *dissimilarities*. This does not necessarily imply that they are actual dissimilarity judgments or measurements, they can be arbitrary transformations of such judgments or measurements as well. Both w_{ij} and δ_{ij} are fixed and known numbers, the A_{ij} are fixed and known symmetric matrices. Note there is no constraint that either the dissimilarities δ_{ij} or the fDistances $f(x'A_{ij}x)$ are non-negative, or that f is increasing. Thus minimizing fStress can also be used, for example, to fit fDistances to similarities

Our notation is somewhat different from standard MDS notation, so let's discuss some translations. We use $x = \mathbf{vec}(X)$, with X the MDS configuration of n points in p dimensions. If we define $\nu(i,s) = (s-1)*n+i$, then $\{X\}_{is} = x_{\nu(i,s)}$. For most of our formulas it is easier to work with vectors than it is to work with matrices, so we will use x instead of X. It is also helpful for our software development, since matrices and higher-dimensional arrays are stored as vectors anyway.

The $np \times np$ matrices A_{ij} are defined using unit vectors e_i and e_j of length n, all zero except for one element that is equal to one. If you like, the e_i are the columns of the identity matrix. We first define $n \times n$ matrices $\mathcal{A}_{ij}$ by

$$\mathcal{A}_{ij} := (e_i - e_j)(e_i - e_j)',$$

and use p copies of $\mathcal{A}_{ij}$ to make the direct sum

$$A_{ij} := \underbrace{\mathcal{A}_{ij} \oplus \cdots \oplus \mathcal{A}_{ij}}_{p \text{ times}}.$$

Thus the squared Euclidean distance between point i and j in the configuration X is $\mathbf{tr}\ X'\mathcal{A}_{ij}X = x'A_{ij}x$.

1.1 fDistances

An *fDistance* is a function of the form $g(x) = f(x'Ax)$ for some real-valued f. Thus minimzinf fStress is fitting fDistances to dissimilarities, using weighted least squares. fStress properly generalizes the usual *stress* function initially proposed by Kruskal (1964a) and Kruskal (1964b), which uses $g(x) = \sqrt{x'Ax}$. The *sStress* loss function of Takane, Young, and De Leeuw (1977) uses the identity $g(x) = x'Ax$, and the *lStress* function of Ramsay (1977) and Ramsay (1982) uses the logarithm $g(x) = \log(x'Ax)$.

For g a general power, i.e. $g(x) = (x'Ax)^r$, we get *rStress*, sometimes also known as *qStress*, studied in a host of (unpublished, my fault) papers by De Leeuw, Groenen, and Pietersz (2006), Groenen and De Leeuw (2010), De Leeuw (2014), De Leeuw, Groenen, and Mair (2016c), De Leeuw, Groenen, and Mair (2016a), De Leeuw, Groenen, and Mair (2016d), De Leeuw, Groenen, and Mair (2016b). URL's and/or DOI's are in the references section.

In Groenen, De Leeuw, and Mathar (1995) first and second partials of fStress are given. We add third and fourth order partials. It is unclear if the higher order partials have any practical applications. In De Leeuw, Groenen, and Mair (2016d) there are some applications of the second derivatives of rStress. We briefly discuss some possible applications of our higher order partials to majorization (a.k.a. MM) algorithms.

2 Partials, Partials Everywhere

2.1 Derivatives of fDistances

We give expressions for the first four derivatives of an arbitrary, but sufficiently many times differentiable, fDistance. In fact, we first address a more general problem, which in its turn is a special case of the first four terms of the multivariate Faà di Bruno formula (see, for instance, Constantine and Savits (1996), Leipnik and Pearce (2007)). We then apply those results to fDistances.

2.2 Faà di Bruno

Suppose $h : \mathcal{R}^n \to \mathcal{R}$, $g : \mathcal{R}^n \to \mathcal{R}$, and $f : \mathcal{R} \to \mathcal{R}$, such that $h(x) = f(g(x))$. We will give expressions for the partials of h of orders up to four. Our formulas separate the parts that depend on f from the parts that depend only on g.

2.2.1 First

The first partials can be written in the form

$$\mathcal{D}_i h(x) = \mathcal{D} f(g(x)) h_{11}(x),$$

with

$$h_{11}(x) := \mathcal{D}_i g(x).$$

2.2.2 Second

Next

$$\mathcal{D}_{ij} h(x) = \mathcal{D} f(g(x)) h_{21}(x) + \mathcal{D}^2 f(g(x)) h_{22}(x),$$

$$h_{21}(x) := \mathcal{D}_{ij}g(x),$$
$$h_{22}(x) := \mathcal{D}_i g(x)\mathcal{D}_j g(x).$$

2.2.3 Third

Next

$$\mathcal{D}_{ijk}h(x) = \mathcal{D}f(g(x))h_{31}(x) + \mathcal{D}^2 f(g(x))h_{32}(x) + \mathcal{D}^3 f(g(x))h_{33}(x),$$

with

$$h_{31}(x) := \mathcal{D}_{ijk}g(x),$$
$$h_{32}(x) := \mathcal{D}_j g(x)\mathcal{D}_{ik}g(x) + \mathcal{D}_i g(x)\mathcal{D}_{jk}g(x) + \mathcal{D}_k g(x)\mathcal{D}_{ij}g(x),$$
$$h_{33}(x) := \mathcal{D}_i g(x)\mathcal{D}_j g(x)\mathcal{D}g_k(x).$$

2.2.4 Fourth

And finally

$$\mathcal{D}_{ijkl}h(x) = \mathcal{D}f(g(x))h_{41}(x) + \mathcal{D}^2 f(g(x))h_{42}(x) + \mathcal{D}^3 f(g(x))h_{43}(x) + \mathcal{D}^4 f(g(x))h_{44}(x),$$

with

$$\begin{aligned}
h_{41}(x) &:= \mathcal{D}_{ijkl}g(x),\\
h_{42}(x) &:= \mathcal{D}_{jl}g(x)\mathcal{D}_{ik}g(x) + \mathcal{D}_{il}g(x)\mathcal{D}_{jk}g(x)\\
&\quad + \mathcal{D}_j g(x)\mathcal{D}_{ikl}g(x) + \mathcal{D}_i g(x)\mathcal{D}_{jkl}g(x) + \mathcal{D}_k g(x)\mathcal{D}_{ijl}g(x) + \mathcal{D}_l g(x)\mathcal{D}_{ijk}g(x),\\
h_{43}(x) &:= \mathcal{D}_{il}g(x)\mathcal{D}_j g(x)\mathcal{D}g_k(x) + \mathcal{D}_i g(x)\mathcal{D}_{jl}g(x)\mathcal{D}g_k(x) + \mathcal{D}_i g(x)\mathcal{D}_j g(x)\mathcal{D}_{kl}g(x)\\
&\quad + \mathcal{D}_j g(x)\mathcal{D}_{ik}g(x)\mathcal{D}_l g(x) + \mathcal{D}_i g(x)\mathcal{D}_{jk}g(x)\mathcal{D}_l g(x) + \mathcal{D}_k g(x)\mathcal{D}_{ij}g(x)\mathcal{D}_l g(x),\\
h_{44}(x) &= \mathcal{D}_i g(x)\mathcal{D}_j g(x)\mathcal{D}_k g(x)\mathcal{D}g_l(x).
\end{aligned}$$

2.3 Faà Goes Quadratic

2.3.1 Quadratic Forms

If $g(x) = x'Ax$ for some symmetric A, as it is in MDS, then $\mathcal{D}_i g(x) = 2\{Ax\}_i$ and $\mathcal{D}_{ij}g(x) = 2a_{ij}$. Also $\mathcal{D}_{ijk}g(x) = 0$ and $\mathcal{D}_{ijkl}g(x) = 0$. Note that we write $\{Ax\}_i$ for element i of the vector Ax. Also note that the results in this section apply for any symmetric matrix A, even if it is not positive-semidefinite.

For the first partials

$$\mathcal{D}_i h(x) = 2\mathcal{D}f(x'Ax)\{Ax\}_i,$$

for the second partials

$$\mathcal{D}_{ij} h(x) = 2\mathcal{D}f(x'Ax)a_{ij} + 4\mathcal{D}^2 f(x'Ax))\{Ax\}_i\{Ax\}_j,$$

and for the third partials

$$\mathcal{D}_{ijk} h(x) = 4\mathcal{D}^2 f(x'Ax)\{\{Ax\}_i a_{jk} + \{Ax\}_j a_{ik} + \{Ax\}_k a_{ij}\} + 8\mathcal{D}^3 f(x'Ax)\{Ax\}_i\{Ax\}_j\{Ax\}_k.$$

For the fourth partials we again write

$$\mathcal{D}_{ijkl} h(x) = \mathcal{D}f(g(x))h_{41}(x) + \mathcal{D}^2 f(g(x))h_{42}(x) + \mathcal{D}^3 f(g(x))h_{43}(x) + \mathcal{D}^4 f(g(x))h_{44}(x),$$

and now we have

$$\begin{aligned}
h_{41}(x) &:= 0,\\
h_{42}(x) &:= 4\{a_{jl}a_{ik} + a_{il}a_{jk}\},\\
h_{43}(x) &:= 8\{a_{il}\{Ax\}_j\{Ax\}_k + a_{jl}\{Ax\}_i\{Ax\}_k + a_{kl}\{Ax\}_i\{Ax\}_j +\\
&\quad + a_{ik}\{Ax\}_j\{Ax\}_l + a_{jk}\{Ax\}_i\{Ax\}_l + a_{ij}\{Ax\}_k\{Ax\}_l\},\\
h_{44}(x) &:= 16\{Ax\}_i\{Ax\}_j\{Ax\}_k\{Ax\}_l.
\end{aligned}$$

2.4 Derivatives of Stress

If we expand fStress we find, using notation due to De Leeuw (1977),

$$\sigma(x) = C - \rho(x) + \eta(x),$$

with

$$\rho(x) := \sum_{k=1}^K w_k z_k f(x'A_k x),$$

$$\eta(x) := \frac{1}{2}\sum_{k=1}^K w_k f^2(x'A_k x),$$

and with C a constant not depending on x.

Clearly both ρ and η^2 are linear combinations of fDistances, where the fDistances in η^2 are the squares of the fDistances in ρ. Thus the partial derivatives are also linear combinations of the partial derivatives of the fDistances. Again, for the derivatives of the squares of the fDistances we can use a univariate version of Faà di Bruno.

We now look at the partials of fStress. For this we use the actual definition of the A_{ij}.

$$\mathcal{D}\sigma(x) = 2(V(x) - B(x))x,$$

with

$$B(x) := \sum_{k=1}^{K} w_k z_k \mathcal{D} f(x' A_k x) A_k,$$

and

$$V(x) := \sum_{k=1}^{K} w_k f(x' A_k x) \mathcal{D} f(x' A_k x) A_k.$$

$$\mathcal{D}^2 \rho(x) = 2 \sum_{k=1}^{K} w_k z_k \left\{ \mathcal{D} f(x' A_k x) A_k + 2 \mathcal{D}^2 f(x' A_k x) A_k x x' A_k \right\}$$

3 Standard MDS Basis

For fStress the matrices A_{ij} have direct sum structure, with all p diagonal blocks equal to $\mathcal{A}_{ij}$.

$$\{A_{ij} x\} = \begin{bmatrix} (x_{i1} - x_{j1})(e_i - e_j) \\ \vdots \\ (x_{ip} - x_{jp})(e_i - e_j) \end{bmatrix}$$

and thus

$$\{A_{ij} x\}_\alpha = (x_{is} - x_{js})(\delta^{i\alpha} - \delta^{j\alpha})$$

$$\{A_{ij} x\}_\alpha \{A_{ij} x\}_\beta = (x_{is} - x_{js})(x_{it} - x_{jt}) \{\mathcal{A}_{ij}\}_{\alpha\beta}$$

$$\{A_{ij} x\}_\alpha \{A_{ij} x\}_\beta \{A_{ij} x\}_\gamma = (x_{is} - x_{js})(x_{it} - x_{jt}) x_{it} - x_{jt}) \{(e_i - e_j) \otimes (e_i - e_j) \otimes (e_i - e_j)\}_{\alpha\beta\gamma}$$

Also

$$\{A_{ij}\}_{\alpha\beta} =$$

4 Implementation

5 Appendix: Code

5.1 R Code

5.1.1 fStress.R

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

library("numDeriv")
```

```
fLogR <- function (x) {
  h <-
    .C(
      "fStressLog",
      x = as.double (x),
      f0 = as.double(0),
      f1 = as.double(0),
      f2 = as.double(0),
      f3 = as.double(0),
      f4 = as.double(0)
    )
  return (list (
    x = h$x,
    f0 = h$f0,
    f1 = h$f1,
    f2 = h$f2,
    f3 = h$f3,
    f4 = h$f4
  ))
}

fIdeR <- function (x) {
  h <-
    .C(
      "fStressIdentity",
      x = as.double (x),
      f0 = as.double(0),
      f1 = as.double(0),
      f2 = as.double(0),
      f3 = as.double(0),
      f4 = as.double(0)
    )
  return (list (
    x = h$x,
    f0 = h$f0,
    f1 = h$f1,
    f2 = h$f2,
    f3 = h$f3,
    f4 = h$f4
  ))
}

fExpR <- function (x) {
  h <-
```

```
    .C(
      "fStressExponent",
      x = as.double (x),
      f0 = as.double(0),
      f1 = as.double(0),
      f2 = as.double(0),
      f3 = as.double(0),
      f4 = as.double(0)
    )
  return (list (
    x = h$x,
    f0 = h$f0,
    f1 = h$f1,
    f2 = h$f2,
    f3 = h$f3,
    f4 = h$f4
  ))
}

fBndR <- function (x) {
  h <-
    .C(
      "fStressBounded",
      x = as.double (x),
      f0 = as.double(0),
      f1 = as.double(0),
      f2 = as.double(0),
      f3 = as.double(0),
      f4 = as.double(0)
    )
  return (list (
    x = h$x,
    f0 = h$f0,
    f1 = h$f1,
    f2 = h$f2,
    f3 = h$f3,
    f4 = h$f4
  ))
}

fLgoR <- function (x) {
  h <-
    .C(
      "fStressLogPlusOne",
```

```
      x = as.double (x),
      f0 = as.double(0),
      f1 = as.double(0),
      f2 = as.double(0),
      f3 = as.double(0),
      f4 = as.double(0)
    )
  return (list (
    x = h$x,
    f0 = h$f0,
    f1 = h$f1,
    f2 = h$f2,
    f3 = h$f3,
    f4 = h$f4
  ))
}

fPowR <- function (x, fNumber, pPower) {
  h <-
    .C(
      "fStressPower",
      x = as.double(x),
      fNumber = as.integer (fNumber),
      ppower = as.double (pPower),
      f0 = as.double(0),
      f1 = as.double(0),
      f2 = as.double(0),
      f3 = as.double(0),
      f4 = as.double(0)
    )
  return (list (
    x = h$x,
    f0 = h$f0,
    f1 = h$f1,
    f2 = h$f2,
    f3 = h$f3,
    f4 = h$f4
  ))
}

faaR <- function(x, fNumber, pPower, a) {
  n <- length (x)
  h <-
    .C(
```

```
      "faa_di_bruno",
      x = as.double(x),
      n = as.integer(n),
      fNumber = as.integer(fNumber - 1),
      pPower = as.double(pPower),
      a = as.double(a),
      ax = as.double (rep(0, n)),
      gx = as.double(0),
      h0 = as.double (0),
      h1 = as.double(rep (0, n)),
      h2 = as.double (rep(0, n ^ 2)),
      h3 = as.double (rep (0, n ^ 3)),
      h4 = as.double (rep(0, n ^ 4))
    )
  return (list (
    gx = h$gx,
    h0 = h$h0,
    h1 = h$h1,
    h2 = h$h2,
    h3 = h$h3,
    h4 = h$h4
  ))
}

fStressR <- function (x, w, delta, p, fNumber, pPower) {
  r <- length (x)
  m <- length (w)
  n <- round (r / p)
  hh <-
    .C(
      "fStressPartials",
      x = as.double (x),
      w = as.double (w),
      delta = as.double (delta),
      n = as.integer (n),
      p = as.integer (p),
      fNumber = as.integer (fNumber - 1),
      pPower = as.double (pPower),
      stress = as.double (0),
      qdist = as.double (rep(0, m)),
      fdist = as.double (rep(0, m)),
      h1 = as.double (rep(0, r)),
      h2 = as.double (rep(0, r ^ 2)),
      h3 = as.double (rep(0, r ^ 3)),
```

```
      h4 = as.double (rep(0, r ^ 4))
    )
  return (
    list(
      x = x,
      stress = hh$stress,
      qdist = hh$qdist,
      fdist = hh$fdist,
      h1 = hh$h1,
      h2 = hh$h2,
      h3 = hh$h3,
      h4 = hh$h4
    )
  )
}
```

5.1.2 check.R

```
fList <- list (function (x)
  log(x),
  function (x)
    x,
  function (x)
    exp(x),
  function (x)
    x / (1 + x),
  function (x)
    log (1 + x))

checkD <- function (expr, order, value) {
  DD <- function(expr, name, order = 1) {
    if (order < 1)
      stop("'order' must be >= 1")
    if (order == 1)
      D(expr, name)
    else
      DD(D(expr, name), name, order - 1)
  }
  dd <- DD(expr, "x", order)
  x <- value
  return (eval (dd))
}
```

```
checkF <- function (x, fNumber, pPower, a) {
  f <- function (x, fNumber, pPower, a) {
    g <- sum (a * outer (x, x))
    return ((fList[[fNumber]](g)) ^ pPower)
  }
  h0 <- f(x, fNumber, pPower, a)
  h1 <- grad (f,
              x,
              fNumber = fNumber,
              pPower = pPower,
              a = a)
  h2 <- hessian (f,
                 x,
                 fNumber = fNumber,
                 pPower = pPower,
                 a = a)
  return (list (
    x = x,
    h0 = h0,
    h1 = h1,
    h2 = h2
  ))
}

checkS <- function (x, w, delta, p, fNumber, pPower) {
  f <- function (x, w, delta, p, fNumber, pPower) {
    r <- length (x)
    n <- round (r / p)
    d <- as.vector(dist(matrix(x, n, p))) ^ 2
    e <- fList[[fNumber]](d) ^ pPower
    sum (w * (delta - e) ^ 2) / 2
  }
  h0 <-
    f(
      x,
      w = w,
      delta = delta,
      p = p,
      fNumber = fNumber,
      pPower = pPower
    )
  h1 <-
    grad (
      f,
```

```
      x,
      w = w,
      delta = delta,
      p = p,
      fNumber = fNumber,
      pPower = pPower
    )
  h2 <-
    hessian (
      f,
      x,
      w = w,
      delta = delta,
      p = p,
      fNumber = fNumber,
      pPower = pPower
    )
  return (list (
    x = x,
    h0 = h0,
    h1 = h1,
    h2 = h2
  ))
}
```

5.2 C Code

5.2.1 fStress.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 int CINDEX(const int, const int, const int, const int, const int, const in

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);

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

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));
}

static inline int ASEEK(const int n, const int p, const int u, const int v, const int i,
    int value = 0;
    if (abs (i - j) < n) {
        for (int s = 1; s <= p; s++) {
            int k = (s - 1) * n;
            if ((i == (u + k)) && (j == (u + k))) {
                value = 1;
            }
            if ((i == (u + k)) && (j == (v + k))) {
                value = -1;
            }
            if ((i == (v + k)) && (j == (u + k))) {
                value = -1;
            }
            if ((i == (v + k)) && (j == (v + k))) {
                value = 1;
            }
        }
    }
    return (value);
}


#endif /* SMACOF_H */
```

5.2.2 fStress.c

```
#include "fStress.h"

void fStressLog(const double *x, double *f0, double *f1, double *f2, double *f3,
                double *f4) {
```

```
    double xx = *x;
    *f0 = log(xx);
    *f1 = 1 / xx;
    *f2 = -1 / SQUARE(xx);
    *f3 = 2 / THIRD(xx);
    *f4 = -6 / FOURTH(xx);
    return;
}

void fStressIdentity(const double *x, double *f0, double *f1, double *f2,
                     double *f3, double *f4) {
    double xx = *x;
    *f0 = xx;
    *f1 = 1.0;
    *f2 = 0.0;
    *f3 = 0.0;
    *f4 = 0.0;
    return;
}

void fStressExponent(const double *x, double *f0, double *f1, double *f2,
                     double *f3, double *f4) {
    double xx = *x;
    *f0 = exp(xx);
    *f1 = exp(xx);
    *f2 = exp(xx);
    *f3 = exp(xx);
    *f4 = exp(xx);
    return;
}

void fStressBounded(const double *x, double *f0, double *f1, double *f2,
                    double *f3, double *f4) {
    double xx = *x, xz = 1.0 + xx;
    *f0 = xx / xz;
    *f1 = 1.0 / SQUARE(xz);
    *f2 = -2.0 / THIRD(xz);
    *f3 = 6.0 / FOURTH(xz);
    *f4 = -24.0 / FIFTH(xz);
    return;
}

void fStressLogPlusOne(const double *x, double *f0, double *f1, double *f2,
                       double *f3, double *f4) {
```

```
    double xx = *x, xp = 1.0 + xx, xz = 1 / xp;
    *f0 = log(xp);
    *f1 = xz;
    *f2 = -SQUARE(xz);
    *f3 = 2.0 * THIRD(xz);
    *f4 = -6.0 * FOURTH(xz);
}

void (*fStressTable[5])(const double *, double *, double *, double *, double *,
                        double *) = {fStressLog, fStressIdentity,
                                     fStressExponent, fStressBounded,
                                     fStressLogPlusOne};

void fStressPower(const double *x, const int *fNumber, const double *ppar,
                  double *g0, double *g1, double *g2, double *g3, double *g4) {
    double f0, f1, f2, f3, f4, pp = *ppar;
    (void)fStressTable[*fNumber](x, &f0, &f1, &f2, &f3, &f4);
    *g0 = pow(f0, pp);
    *g1 = pp * f1 * pow(f0, pp - 1.0);
    *g2 = pp * (pp - 1.0) * pow(f0, pp - 2.0) * SQUARE(f1);
    *g2 += pp * pow(f0, pp - 1.0) * f2;
    *g3 = pp * (pp - 1.0) * (pp - 2.0) * pow(f0, pp - 3.0) * THIRD(f1);
    *g3 += 3.0 * pp * (pp - 1.0) * pow(f0, pp - 2.0) * f1 * f2;
    *g3 += pp * pow(f0, pp - 1.0) * f3;
    *g4 = pp * (pp - 1.0) * (pp - 2.0) * (pp - 3.0) * pow(f0, pp - 4.0) *
          FOURTH(f1);
    *g4 += 6.0 * pp * (pp - 1.0) * (pp - 2.0) * pow(f0, pp - 3.0) * SQUARE(f1) *
           f2;
    *g4 += 4.0 * pp * (pp - 1.0) * pow(f0, pp - 2.0) * (f1 * f3);
    *g4 += 3.0 * pp * (pp - 1.0) * pow(f0, pp - 2.0) * SQUARE(f2);
    *g4 += pp * pow(f0, pp - 1.0) * f4;
}

void fStressFaaDiBruno(const double *x, const int *n, const int *p,
                       const int *u, const int *v, const int *fNumber,
                       const double *par, double *ax, double *gx, double *h0,
                       double *h1, double *h2, double *h3, double *h4) {
    double f0, f1, f2, f3, f4;
    int nn = *n, uu = *u, vv = *v, pp = *p, np = nn * pp;
    *gx = 0.0;
    for (int i = 1; i <= np; i++) {
        ax[VINDEX(i)] = 0.0;
    }
    for (int s = 1; s <= pp; s++) {
```

```
        double xuv = x[MINDEX(uu, s, nn)] - x[MINDEX(vv, s, nn)];
        ax[MINDEX(uu, s, nn)] = xuv;
        ax[MINDEX(vv, s, nn)] = -xuv;
    }
    for (int i = 1; i <= np; i++) {
        *gx += ax[VINDEX(i)] * x[VINDEX(i)];
    }
    (void)fStressPower(gx, fNumber, par, &f0, &f1, &f2, &f3, &f4);
    *h0 = f0;
    for (int i = 1; i <= np; i++) {
        h1[VINDEX(i)] = 2.0 * f1 * ax[VINDEX(i)];
    }
    for (int i = 1; i <= np; i++) {
        for (int j = 1; j <= np; j++) {
            h2[MINDEX(i, j, np)] = 2.0 * f1 * ASEEK(nn, pp, uu, vv, i, j) +
                                   4.0 * f2 * ax[VINDEX(i)] * ax[VINDEX(j)];
        }
    }
    for (int i = 1; i <= np; i++) {
        for (int j = 1; j <= np; j++) {
            for (int k = 1; k <= np; k++) {
                h3[AINDEX(i, j, k, np, np)] =
                    4.0 * f2 *
                    ((ax[VINDEX(i)] * ASEEK(nn, pp, uu, vv, j, k)) +
                     (ax[VINDEX(j)] * ASEEK(nn, pp, uu, vv, i, k)) +
                     (ax[VINDEX(k)] * ASEEK(nn, pp, uu, vv, i, j)));
                h3[AINDEX(i, j, k, nn, nn)] +=
                    8.0 * f3 * ax[VINDEX(i)] * ax[VINDEX(j)] * ax[VINDEX(k)];
            }
        }
    }
    for (int i = 1; i <= np; i++) {
        for (int j = 1; j <= np; j++) {
            for (int k = 1; k <= np; k++) {
                for (int l = 1; l <= np; l++) {
                    h4[CINDEX(i, j, k, l, np, np, np)] =
                        4.0 * f2 *
                        (ASEEK(nn, pp, uu, vv, j, l) *
                             ASEEK(nn, pp, uu, vv, i, k) +
                         ASEEK(nn, pp, uu, vv, i, l) *
                             ASEEK(nn, pp, uu, vv, j, k));
                    h4[CINDEX(i, j, k, l, nn, nn, nn)] +=
                        16.0 * f4 * ax[VINDEX(i)] * ax[VINDEX(j)] *
                        ax[VINDEX(k)] * ax[VINDEX(l)];
```

```
                    h4[CINDEX(i, j, k, l, nn, nn, nn)] +=
                        8.0 * f3 * ASEEK(nn, pp, uu, vv, i, l) * ax[VINDEX(j)] *
                        ax[VINDEX(k)];
                    h4[CINDEX(i, j, k, l, nn, nn, nn)] +=
                        8.0 * f3 * ASEEK(nn, pp, uu, vv, j, l) * ax[VINDEX(i)] *
                        ax[VINDEX(k)];
                    h4[CINDEX(i, j, k, l, nn, nn, nn)] +=
                        8.0 * f3 * ASEEK(nn, pp, uu, vv, k, l) * ax[VINDEX(j)] *
                        ax[VINDEX(j)];
                    h4[CINDEX(i, j, k, l, nn, nn, nn)] +=
                        8.0 * f3 * ASEEK(nn, pp, uu, vv, i, k) * ax[VINDEX(j)] *
                        ax[VINDEX(l)];
                    h4[CINDEX(i, j, k, l, nn, nn, nn)] +=
                        8.0 * f3 * ASEEK(nn, pp, uu, vv, j, k) * ax[VINDEX(i)] *
                        ax[VINDEX(l)];
                    h4[CINDEX(i, j, k, l, nn, nn, nn)] +=
                        8.0 * f3 * ASEEK(nn, pp, uu, vv, i, j) * ax[VINDEX(k)] *
                        ax[VINDEX(l)];
                }
            }
        }
    }
    return;
}

void faa_di_bruno(const double *x, const int *n, const int *fNumber,
                  const double *par, const double *a, double *ax, double *gx,
                  double *h0, double *h1, double *h2, double *h3, double *h4) {
    double f0, f1, f2, f3, f4;
    int nn = *n;
    *gx = 0.0;
    for (int i = 1; i <= nn; i++) {
        ax[VINDEX(i)] = 0.0;
        for (int j = 1; j <= nn; j++) {
            ax[VINDEX(i)] += a[MINDEX(i, j, nn)] * x[VINDEX(j)];
        }
        *gx += ax[VINDEX(i)] * x[VINDEX(i)];
    }
    (void)fStressPower(gx, fNumber, par, &f0, &f1, &f2, &f3, &f4);
    *h0 = f0;
    for (int i = 1; i <= nn; i++) {
        h1[VINDEX(i)] = 2.0 * f1 * ax[VINDEX(i)];
    }
    for (int i = 1; i <= nn; i++) {
```

```
        for (int j = 1; j <= nn; j++) {
            h2[MINDEX(i, j, nn)] = 2.0 * f1 * a[MINDEX(i, j, nn)] +
                                   4.0 * f2 * ax[VINDEX(i)] * ax[VINDEX(j)];
        }
    }
    for (int i = 1; i <= nn; i++) {
        for (int j = 1; j <= nn; j++) {
            for (int k = 1; k <= nn; k++) {
                h3[AINDEX(i, j, k, nn, nn)] =
                    4.0 * f2 *
                    ((ax[VINDEX(i)] * a[MINDEX(j, k, nn)]) +
                     (ax[VINDEX(j)] * a[MINDEX(i, k, nn)]) +
                     (ax[VINDEX(k)] * a[MINDEX(i, j, nn)]));
                h3[AINDEX(i, j, k, nn, nn)] +=
                    8.0 * f3 * ax[VINDEX(i)] * ax[VINDEX(j)] * ax[VINDEX(k)];
            }
        }
    }
    for (int i = 1; i <= nn; i++) {
        for (int j = 1; j <= nn; j++) {
            for (int k = 1; k <= nn; k++) {
                for (int l = 1; l <= nn; l++) {
                    h4[CINDEX(i, j, k, l, nn, nn, nn)] =
                        4.0 * f2 *
                        (a[MINDEX(j, l, nn)] * a[MINDEX(i, k, nn)] +
                         a[MINDEX(i, l, nn)] * a[MINDEX(j, k, nn)]);
                    h4[CINDEX(i, j, k, l, nn, nn, nn)] +=
                        16.0 * f4 * ax[VINDEX(i)] * ax[VINDEX(j)] *
                        ax[VINDEX(k)] * ax[VINDEX(l)];
                    h4[CINDEX(i, j, k, l, nn, nn, nn)] +=
                        8.0 * f3 * a[MINDEX(i, l, nn)] * ax[VINDEX(j)] *
                        ax[VINDEX(k)];
                    h4[CINDEX(i, j, k, l, nn, nn, nn)] +=
                        8.0 * f3 * a[MINDEX(j, l, nn)] * ax[VINDEX(i)] *
                        ax[VINDEX(k)];
                    h4[CINDEX(i, j, k, l, nn, nn, nn)] +=
                        8.0 * f3 * a[MINDEX(k, l, nn)] * ax[VINDEX(j)] *
                        ax[VINDEX(j)];
                    h4[CINDEX(i, j, k, l, nn, nn, nn)] +=
                        8.0 * f3 * a[MINDEX(i, k, nn)] * ax[VINDEX(j)] *
                        ax[VINDEX(l)];
                    h4[CINDEX(i, j, k, l, nn, nn, nn)] +=
                        8.0 * f3 * a[MINDEX(j, k, nn)] * ax[VINDEX(i)] *
                        ax[VINDEX(l)];
```

```
                    h4[CINDEX(i, j, k, l, nn, nn, nn)] +=
                        8.0 * f3 * a[MINDEX(i, j, nn)] * ax[VINDEX(k)] *
                        ax[VINDEX(l)];
                }
            }
        }
    }
    return;
}

void fStressPartials(const double *x, const double *w, const double *delta,
                     const int *n, const int *p, const int *fNumber,
                     const double *par, double *stress, double *d, double *fd,
                     double *f1, double *f2, double *f3, double *f4) {
    double par2 = 2 * *par, nn = *n, pp = *p, np = nn * pp, gx, hx, h0, g0;
    double *ax = (double *)calloc((size_t)np, sizeof(double));
    double *h1 = (double *)calloc((size_t)np, sizeof(double));
    double *h2 = (double *)calloc((size_t)SQUARE(np), sizeof(double));
    double *h3 = (double *)calloc((size_t)THIRD(np), sizeof(double));
    double *h4 = (double *)calloc((size_t)FOURTH(np), sizeof(double));
    double *g1 = (double *)calloc((size_t)np, sizeof(double));
    double *g2 = (double *)calloc((size_t)SQUARE(np), sizeof(double));
    double *g3 = (double *)calloc((size_t)THIRD(np), sizeof(double));
    double *g4 = (double *)calloc((size_t)FOURTH(np), sizeof(double));
    *stress = 0.0;
    for (int j = 1; j <= nn - 1; j++) {
        for (int i = j + 1; i <= nn; i++) {
            int k = SINDEX(i, j, nn);
            (void)fStressFaaDiBruno(x, n, p, &i, &j, fNumber, par, ax, &hx, &h0,
                                    h1, h2, h3, h4);
            d[k] = hx;
            fd[k] = h0;
            *stress += w[k] * SQUARE(delta[k] - fd[k]);
            (void)fStressFaaDiBruno(x, n, p, &i, &j, fNumber, &par2, ax, &gx,
                                    &g0, g1, g2, g3, g4);
            for (int r = 1; r <= np; r++) {
                int ind = VINDEX(r);
                f1[ind] += w[k] * (0.5 * g1[ind] - delta[k] * h1[ind]);
                for (int s = 1; s <= np; s++) {
                    int ind = MINDEX(r, s, np);
                    f2[ind] += w[k] * (0.5 * g2[ind] - delta[k] * h2[ind]);
                    for (int t = 1; t <= np; t++) {
                        int ind = AINDEX(r, s, t, np, np);
                        f3[ind] += w[k] * (0.5 * g3[ind] - delta[k] * h3[ind]);
```

```
                        for (int u = 1; u <= np; u++) {
                            int ind = CINDEX(r, s, t, u, np, np, np);
                            f4[ind] +=
                                w[k] * (0.5 * g4[ind] - delta[k] * h4[ind]);
                        }
                    }
                }
            }
        }
    }
    *stress /= 2.0;
    free(ax);
    free(h1);
    free(h2);
    free(h3);
    free(h4);
    free(g1);
    free(g2);
    free(g3);
    free(g4);
}
```

References

Constantine, G.M., and T.H. Savits. 1996. "A Multivariate Faà di Bruno Formula with Applications." *Transactions of the American Mathematical Society* 348 (2): 503–20.

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.

De Leeuw, J. 2014. "Minimizing rStress Using Nested Majorization." UCLA Department of Statistics. http://www.stat.ucla.edu/~deleeuw/janspubs/2014/notes/deleeuw_U_14c.pdf.

De Leeuw, J., P.J.F Groenen, and R. Pietersz. 2006. "Optimizing Functions of Squared Distances." UCLA Department of Statistics. http://www.stat.ucla.edu/~deleeuw/janspubs/2006/notes/deleeuw_groenen_pietersz_U_06.pdf.

De Leeuw, J., P.J.F. Groenen, and P. Mair. 2016a. "Differentiability of rStress at a Local Minimum." doi:10.13140/RG.2.1.1249.8968.

———. 2016b. "Minimizing qStress for Small q." doi:10.13140/RG.2.1.4843.1764.

———. 2016c. "Minimizing rStress Using Majorization." doi:10.13140/RG.2.1.3871.3366.

———. 2016d. "Second Derivatives of rStress, with Applications." doi:10.13140/RG.2.1.1058.4081.

Groenen, P.J.F., and J. De Leeuw. 2010. "Power-Stress for Multidimensional Scaling."

http://www.stat.ucla.edu/~deleeuw/janspubs/2010/notes/groenen_deleeuw_U_10.pdf.

Groenen, P.J.F., J. De Leeuw, and R. Mathar. 1995. "Least Squares Multidimensional Scaling with Transformed Distances." In *From Data to Knowledge: Theoretical and Practical Aspects of Classification, Data Analysis and Knowledge Organization*, edited by W. Gaul and D. Pfeifer. Berlin, Germany: Springer Verlag. http://www.stat.ucla.edu/~deleeuw/janspubs/1995/chapters/groenen_deleeuw_mathar_C_95.pdf.

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.

Leipnik, R.B, and C.E.M. Pearce. 2007. "The Multivarite Faà di Bruno Formula and Multivariate Taylor Expansions with Explicit Integral Remainder Term." *Australian and New Zealand Industrial and Applied Mathematics Journal* 48: 327–41.

Ramsay, J. O. 1982. "Some Statistical Approaches to Multidimensional Scaling Data." *J. Roy. Statist. Soc. Ser. A* 145 (3): 285–312.

Ramsay, J.O. 1977. "Maximum Likelihood Estimation in MDS." *Psychometrika* 42: 241–66.

Takane, Y., F.W. Young, and J. De Leeuw. 1977. "Nonmetric Individual Differences in Multidimensional Scaling: An Alternating Least Squares Method with Optimal Scaling Features." *Psychometrika* 42: 7–67. http://www.stat.ucla.edu/~deleeuw/janspubs/1977/articles/takane_young_deleeuw_A_77.pdf.