Simultaneous Diagonalization in R/C

Jan de Leeuw

Version June 19, 2018

Abstract

We present an R/C implementation of optimal simultaneous diagonalization of several real symmetric matrices using Jacobi plane rotations, with compact triangular storage of symmetric matrices.

Contents

1 Introduction **1**

2 Algorithm **2**

3 Illustration **5**

4 Examples **6**

4.1 Three Two by Two Matrices 6

4.2 One Matrix 7

4.3 General Case 7

5 Code **8**

5.1 R code 8

5.2 C Code 10

References **16**

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

1 Introduction

Suppose A_j are m real symmetric matrices of order n. Our problem is to find a square orthonormal K, i.e. a matrix with $KK' = K'K = I$, such that the matrices $H_j = K'A_jK$ are as diagonal as possible in the least squares sense. Thus we want to minimize the sum of squares of all off-diagonal elements of the H_j, or, equivalently, maximize the sum of squares of all diagonal elements.

It is a natural idea to use Jacabi plain rotations, the orthonormal version of cyclic coordinate descent, for this. The algorithm was proposed, possibly first, by De Leeuw and Pruzansky

(1978). It was subsequently programmed in FORTRAN for the Applied Statistics algorithms by Clarkson (1988). Numerical analysts arrived later on the scene, but contributed a number of essential refinements and extensions, such as an algorithm for Hermitian matrices and a proof of quadratic convergence (Bunse-Gerstner, Byers, and Mehrmann (1993), Cardoso and Souloumiac (1996)). There is a version of the algorithm in R (R Core Team (2018)) in De Leeuw and Ferrari (2008), together with a number of related algorithms using Jacobi plane rotations. De Leeuw (2017) has an R/C program (i.e. calling routine in in R, computational routines in C) for $m = 1$, the classical Jacobi algorithm, but using compact triangular storage of symmetric matrices. In this paper we provide an R/C compact storage version of the simultaneous diagonalization algorithm.

2 Algorithm

In the classical Jacobi method we cycle through all off-diagonal elements in a fixed order, minimizing the sum of squares over single-parameter plane rotations each time. After a cycle is completed we test for convergence. If there is still room for improvement we start a new cycle.

The plane rotation K for any $i \neq j$ is defined by

$$\begin{aligned} k_{ii} &= k_{jj} = x, \\ k_{ij} &= -k_{ji} = y, \end{aligned}$$

where $x^2 + y^2 = 1$. For $\ell \neq i, j$ we have $k_{\ell\ell} = 1$, and for $\nu \neq i, j$ and $\ell \neq i, j$ we have $k_{\nu\ell} = 0$. For $H = K'AK$ we find

$$\begin{aligned} h_{ii} &= a_{ii}x^2 - 2a_{ij}xy + a_{jj}y^2, \\ h_{ij} = h_{ji} &= a_{ij}(x^2 - y^2) + (a_{ii} - a_{jj})xy, \\ h_{jj} &= a_{jj}x^2 + 2a_{ij}xy + a_{ii}y^2, \end{aligned}$$

which implies

$$h_{ii}^2 + h_{jj}^2 + 2h_{ij}^2 = a_{ii}^2 + a_{jj}^2 + 2a_{ij}^2.$$

For $\ell \neq i, j$

$$\begin{aligned} h_{i\ell} &= h_{\ell i} = a_{i\ell}x - a_{j\ell}y, \\ h_{j\ell} &= h_{\ell j} = a_{i\ell}y + a_{j\ell}x, \end{aligned}$$

which implies

$$h_{i\ell}^2 + h_{j\ell}^2 = a_{i\ell}^2 + a_{j\ell}^2.$$

Thus for m of such matrices A_k we want to find the optimal plane rotation by minimizing

$$f(\theta) = \sum_{k=1}^{m} (a_{ijk}(x^2 - y^2) + (a_{iik} - a_{jjk})xy)^2 \tag{1}$$

over the circle $x^2 + y^2 = 1$. Now define

$$u = x^2 - y^2, \tag{2}$$
$$v = 2xy. \tag{3}$$

Note that $u^2 + v^2 = (x^2 + y^2)^2$ and thus $u^2 + v^2 = 1$ if and only if $x^2 + y^2 = 1$. In other words the map

$$\begin{bmatrix} x \\ y \end{bmatrix} \Rightarrow \begin{bmatrix} u \\ v \end{bmatrix}$$

maps a point on the circle to another point on the circle. If $u^2 + v^2 = 1$ then the inverse map consists of two antipodal points on the circle, of which one is

$$x = \sqrt{\frac{1+u}{2}}, \tag{4}$$
$$y = \mathbf{sign}(v)\sqrt{\frac{1-u}{2}}. \tag{5}$$

By (1) both antipodal points give the same loss function value, so we always choose the one in (4) and (4).

With our new variables the problem of finding the optimal rotation is

$$f(u, v) = \sum_{k=1}^{m}(ua_{ijk} + vd_{ijk})^2. \tag{6}$$

with $d_{ijk} \triangleq \frac{1}{2}(a_{iik} - a_{jjk})$. Define the 2×2 matrix S by

$$S = \begin{bmatrix} p & q \\ q & r \end{bmatrix},$$

with

$$
\begin{aligned}
p &= \sum_{k=1}^{m} a_{ijk}^2,\\
q &= \sum_{k=1}^{m} a_{ijk}d_{ijk},\\
r &= \sum_{k=1}^{m} d_{ijk}^2.
\end{aligned}
$$

Then

$$f(u,v) = pu^2 + 2quv + rv^2, \tag{7}$$

which we must minimize over $u^2 + v^2 = 1$. The minimizing (u,v) is a normalized eigenvector corresponding with the smallest eigenvalue of S.

The two eigenvalues of S are

$$
\begin{aligned}
\lambda_{\max}(S) &= \frac{1}{2}\{(p+r) + \sqrt{(p-r)^2 + 4q^2}\},\\
\lambda_{\min}(S) &= \frac{1}{2}\{(p+r) - \sqrt{(p-r)^2 + 4q^2}\}.
\end{aligned}
$$

There are several cases to consider, depending on the precise structure of S.

- If $q \neq 0$ the two eigenvalues of S are different, and $\lambda_{\min}(S) < \min(p,r)$. One choice for the corresponding normalized eigenvector has elements

$$u = \frac{q}{\sqrt{q^2 + (\lambda_{\min}(S) - p)^2}}, \tag{8}$$

$$v = \frac{\lambda_{\min}(S) - p}{\sqrt{q^2 + (\lambda_{\min}(S) - p)^2}}, \tag{9}$$

  For this choice both u and v are non-zero, with $v < 0$. The eigenvector is unique, except for sign changes, which do not influence the loss function value in (7). So we always choose $v < 0$. From (4) and (5) we see that we can choose the solution

$$K = \begin{bmatrix} \frac{1}{2}\sqrt{2(1+u)} & -\frac{1}{2}\sqrt{2(1-u)} \\ \frac{1}{2}\sqrt{2(1-u)} & \frac{1}{2}\sqrt{2(1+u)} \end{bmatrix}, \tag{10}$$

  which has $f(u,v) = \lambda_{\min}(S)$.

- If $q = 0$ and $p > r$ then $u = 0$ and $v = \pm 1$. Thus $f(u, v) = \lambda_{\min}(S) = r$, and it does not matter which sign we choose. We set
$$K = \begin{bmatrix} \frac{1}{2}\sqrt{2} & -\frac{1}{2}\sqrt{2} \\ \frac{1}{2}\sqrt{2} & \frac{1}{2}\sqrt{2} \end{bmatrix}.$$
- If $q = 0$ and $p < r$ then $v = 0$ and $u = \pm 1$. We choose $u = 1$ in (10) , which makes K equal to the identity, and which means we skip this rotation.
- If $q = 0$ and $p = r$ then $f(u, v) = p + r$ for any choice of u and v. Again we set K equal to the identity.

3 Illustration

Suppose we have three 2×2 symmetric matrices

```
##      [,1] [,2]
## [1,] +1   -1
## [2,] -1   +1
```

```
##      [,1] [,2]
## [1,] +2   +0
## [2,] +0   +0
```

```
##      [,1] [,2]
## [1,] +1   -2
## [2,] -2   +0
```

The sum of squares of the off-diagonal elements is 5.

For 2×2 matrices we only have to compute a single plane rotation, there is no cycling. So let's perform the necessary calculations.

The matrix S is

```
##      [,1]  [,2]
## [1,] +5.00 -1.00
## [2,] -1.00 +1.25
```

with smallest eigenvalue 1 and with corresponding eigenvector -0.242535625, -0.9701425001. Thus $u = \cos(2\theta)$ is -0.242535625.

```
## [1] 1
```

```
## [1] 1
```

From u we compute $\frac{1}{2}\sqrt{2(1+u)}$ and $\frac{1}{2}\sqrt{2(1-u)}$, which are respectively 0.6154122094 and 0.788205438. And the rotation matrix is

```
##      [,1]      [,2]
## [1,] +0.615412 -0.788205
## [2,] +0.788205 +0.615412
```

The rotated matrices are

```
##      [,1]      [,2]
## [1,] +0.029857 +0.242536
## [2,] +0.242536 +1.970143
```

```
##      [,1]      [,2]
## [1,] +0.757464 -0.970143
## [2,] -0.970143 +1.242536
```

```
##      [,1]      [,2]
## [1,] -1.561553 +0.000000
## [2,] +0.000000 +2.561553
```

with a sum of squares of off-diagonal elements of 1 and a sum of squares of diagonal elements of 15.

4 Examples

4.1 Three Two by Two Matrices

This is the same example we used as an illustration earlier.

```
a <- c(1,-1,1,2,0,0,1,-2,0)
h <- sjacobi (a, 2, 3)
```

After 2 cycles (where the second one merely concludes there is convergence) we find the rotation

```
##                [,1]         [,2]
## [1,]  0.7882054380 0.6154122094
## [2,] -0.6154122094 0.7882054380
```

The sum of squares of the diagona elements function increased from 7 to 15. The rotated matrices are

```
##      [,1]      [,2]
## [1,] +1.970143 -0.242536
## [2,] -0.242536 +0.029857
```

```
##      [,1]      [,2]
## [1,] +1.242536 +0.970143
## [2,] +0.970143 +0.757464
```

```
##      [,1]      [,2]
## [1,] +2.561553 -0.000000
## [2,] -0.000000 -1.561553
```

The results are the same as those computed "by hand".

4.2 One Matrix

The algorithm can obviously also be applied if $m = 1$, in which case it becomes the cyclic version of the Jacobi method. Minimum loss should be zero.

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    1    2    3    4    5    6    7    8    9    10
##  [2,]    2   11   12   13   14   15   16   17   18    19
##  [3,]    3   12   20   21   22   23   24   25   26    27
##  [4,]    4   13   21   28   29   30   31   32   33    34
##  [5,]    5   14   22   29   35   36   37   38   39    40
##  [6,]    6   15   23   30   36   41   42   43   44    45
##  [7,]    7   16   24   31   37   42   46   47   48    49
##  [8,]    8   17   25   32   38   43   47   50   51    52
##  [9,]    9   18   26   33   39   44   48   51   53    54
## [10,]   10   19   27   34   40   45   49   52   54    55
```

```r
h <- sjacobi (a, 10, 1)
```

In 26 cycles we reduce the sum of squares of the off-diagonal elements from 84636 to 0.0000000003. The diagonal of the rotated matrix is

```
##  [1]    1.0699214091   12.1639813624  314.7797170547    2.1774756456
##  [5]    2.8050481734    0.5991942823    6.6137980129   -1.8824366513
##  [9]    0.1409608363    1.5323398746
```

which can be compared with the eigenvalues computed by R

```
##  [1] 314.7797170547  12.1639813624   6.6137980129   2.8050481734
##  [5]   2.1774756456   1.5323398746   1.0699214091   0.5991942823
##  [9]   0.1409608363  -1.8824366513
```

Clearly the eigenvalues computed by `sjacobi` and by `eigen` are the same, except for order. And so will be the eigenvectors.

4.3 General Case

For our last example we construct 4 commuting matrices of order 4. Minimum loss should be zero, again.

```r
set.seed(12345)
c1 <- crossprod (matrix(rnorm(40), 10, 4))
ee <- eigen(c1)$vectors
c2 <- tcrossprod (ee %*% diag(rnorm(4)), ee)
c3 <- tcrossprod (ee %*% diag(rnorm(4)), ee)
c4 <- tcrossprod (ee %*% diag(rnorm(4)), ee)
```

```
sc <- sum(c1^2) + sum (c2^2) + sum (c3^2) + sum (c4^2)
a <- c(c1[outer(1:4,1:4,">=")],c2[outer(1:4,1:4,">=")],c3[outer(1:4,1:4,">=")],c4[outer
h <- sjacobi (a, 4, 4)
```

After 4 iterations we have reduced loss from 227.4632340211 to 0.0000000000. Close enough to zero.

5 Code

5.1 R code

5.1.1 sjacobi.R

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

sjacobi <-
  function (a,
            n,
            m,
            eps = 1e-15,
            itmax = 1000,
            vectors = TRUE,
            verbose = FALSE) {
    h <- .C(
      "sjacobi",
      n = as.integer (n),
      m = as.integer (m),
      a = as.double (a),
      k = as.double (rep(0, n * n)),
      fstart = as.double (0),
      ffinal = as.double (0),
      itel = as.integer (0),
      itmax = as.integer(itmax),
      eps = as.double(eps),
      verbose = as.integer(verbose),
      vectors = as.integer(vectors)
      )
    return(list(
      astart = a,
      afinal = h$a,
      k = h$k,
      fstart = h$fstart,
      ffinal = h$ffinal,
      itel = h$itel
    ))
  }

triangle2matrix <- function (x) {
  m <- length (x)
  n <- round ((sqrt (1 + 8 * m) - 1) / 2)
  h <-
    .C("trimat", as.integer (n), as.double (x), as.double (rep (0, n * n)))
  return (matrix(h[[3]], n, n))
}

matrix2triangle <- function (x) {
  n <- dim(x)[1]
  m <- n * (n + 1) / 2
  h <-
    .C("mattri", as.integer (n), as.double (x), as.double (rep (0, m)))
  return (h[[3]])
}

trianglePrint <- 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))
}

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

5.2 C Code

5.2.1 sjacobi.h

```
#ifndef SJACOBI_H
#define SJACOBI_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 UINDEX(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 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 int UINDEX(const int i, const int j, const int k, const int n,
                         const int m) {
  return ((k - 1) * n * (n + 1) / 2) + ((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; }

void sjacobi(const int *, const int *, double *, double *, double *, double *,
             int *, const int *, const double *, const bool *, const bool *);

void primat(const int *, const int *, const int *, const int *, const double *);
void pritru(const int *, const int *, const int *, const double *);
void prisru(const int *, const int *, const int *, const int *, const double *);
void trimat(const int *, const double *, double *);
void mattri(const int *, const double *, double *);

#endif /* SJACOBI_H */
```

5.2.2 sjacobi.c

```c
#include "sjacobi.h"

/*
int main(void) {
  int n = 10, m = 6, it = 0, itmax = 100, w = 10, p = 6;
  bool verbose = true, vectors = true;
  double a[330], evec[100], oldi[10], oldj[10], fini = 0, f = 0, eps = 1e-15;
  for (int i = 1; i <= 330; i++)
    a[VINDEX(i)] = (double)i;
  (void)sjacobi(&n, &m, a, evec, &fini, &f, &it, oldi, oldj, &itmax, &eps,
                &verbose, &vectors);
  (void)primat(&n, &n, &w, &p, evec);
  (void)prisru(&n, &m, &w, &p, a);
}
*/

/*
int main(void) {
  int n = 2, m = 3, it = 0, itmax = 100;
  bool verbose = true, vectors = true;
  double a[9] = {1.0, -1.0, 1.0, 2.0, 0.0, 0.0, 1.0, -2.0, 0.0};
  double evec[4] = {0.0, 0.0, 0.0, 0.0};
  double oldi[2] = {0.0, 0.0}, oldj[2] = {0.0, 0.0}, fini = 0, f = 0,
         eps = 1e-15;
  (void)sjacobi(&n, &m, a, evec, &fini, &f, &it, oldi, oldj, &itmax, &eps,
                &verbose, &vectors);
}
*/

/*
int main(void) {
  int n = 10, m = 1, it = 0, itmax = 100, w = 10, p = 6;
  bool verbose = true, vectors = true;
  double a[55], oldi[10], evec[100], oldj[10], fini = 0, f = 0, eps = 1e-16;
  for (int i = 1; i <= 55; i++) {
    a[VINDEX(i)] = (double)i;
  }
  (void)sjacobi(&n, &m, a, evec, &fini, &f, &it, oldi, oldj, &itmax, &eps,
                &verbose, &vectors);
  (void)pritru(&n, &w, &p, a);
  (void)primat(&n, &n, &w, &p, evec);
}
```

```
*/

void sjacobi(const int *nn, const int *mm, double *a, double *evec,
             double *fini, double *f, int *it, const int *itmax,
             const double *eps, const bool *verbose, const bool *vectors) {
  int n = *nn, m = *mm, itel = 1;
  double d = 0.0, cost = 0.0, sint = 0.0, u = 0.0, p = 0.0, q = 0.0, r = 0.0,
         piil = 0.0, pijl = 0.0, lbd = 0.0, dd = 0.0, pp = 0.0, dp = 0.0;
  double fold = 0.0, fnew = 0.0, oldi = 0.0, oldj = 0.0;
  if (*vectors) {
    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 k = 1; k <= m; k++) {
    for (int i = 1; i <= n; i++) {
      fold += SQUARE(a[UINDEX(i, i, k, n, m)]);
    }
  }
  *fini = fold;
  while (true) {
    for (int j = 1; j <= n - 1; j++) {
      for (int i = j + 1; i <= n; i++) {
        dd = 0.0, pp = 0.0, dp = 0.0;
        for (int k = 1; k <= m; k++) {
          p = a[UINDEX(i, j, k, n, m)];
          q = a[UINDEX(i, i, k, n, m)];
          r = a[UINDEX(j, j, k, n, m)];
          d = (q - r) / 2.0;
          dd += SQUARE(d);
          pp += SQUARE(p);
          dp += p * d;
        }
        lbd = ((dd + pp) - sqrt(SQUARE(dd - pp) + 4.0 * SQUARE(dp))) / 2.0;
        u = dp / sqrt(SQUARE(dp) + SQUARE(lbd - pp));
        if ((fabs(dp) < 1e-15) && (pp <= dd)) {
          continue;
        }
        cost = sqrt((1 + u) / 2);
        sint = -sqrt((1 - u) / 2);
        if (*vectors) {
          for (int l = 1; l <= n; l++) {
```

```
          piil = evec[MINDEX(l, i, n)];
          pijl = evec[MINDEX(l, j, n)];
          evec[MINDEX(l, i, n)] = cost * piil - sint * pijl;
          evec[MINDEX(l, j, n)] = sint * piil + cost * pijl;
        }
      }
      for (int k = 1; k <= m; k++) {
        p = a[UINDEX(i, j, k, n, m)];
        q = a[UINDEX(i, i, k, n, m)];
        r = a[UINDEX(j, j, k, n, m)];
        for (int l = 1; l <= n; l++) {
          if ((l == i) || (l == j))
            continue;
          int il = IMIN(i, l);
          int li = IMAX(i, l);
          int jl = IMIN(j, l);
          int lj = IMAX(j, l);
          oldi = a[UINDEX(li, il, k, n, m)];
          oldj = a[UINDEX(lj, jl, k, n, m)];
          a[UINDEX(li, il, k, n, m)] = cost * oldi - sint * oldj;
          a[UINDEX(lj, jl, k, n, m)] = sint * oldi + cost * oldj;
        }
        a[UINDEX(i, i, k, n, m)] =
            SQUARE(cost) * q + SQUARE(sint) * r - 2 * cost * sint * p;
        a[UINDEX(j, j, k, n, m)] =
            SQUARE(sint) * q + SQUARE(cost) * r + 2 * cost * sint * p;
        a[UINDEX(i, j, k, n, m)] =
            cost * sint * (q - r) + (SQUARE(cost) - SQUARE(sint)) * p;
      }
    }
  }
  fnew = 0.0;
  for (int k = 1; k <= m; k++) {
    for (int i = 1; i <= n; i++) {
      fnew += SQUARE(a[UINDEX(i, i, k, n, m)]);
    }
  }
  if (*verbose == true) {
    printf("itel %4d fold %15.10f fnew %15.10f\n", itel, fold, fnew);
  }
  if (((fnew - fold) < *eps) || (itel == *itmax))
    break;
  fold = fnew;
  itel++;
  }
  *f = fnew;
  *it = itel;
  return;
}

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;
}

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;
}

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

void trimat(const int *n, const double *x, double *y) {
```

```c
  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;
}

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;
}
```

References

Bunse-Gerstner, A., R. Byers, and V. Mehrmann. 1993. "Numerical Methods for Simultaneous Diagonalization." *SIAM Journal Matrix Analysis and Applications* 14 (4): 927–49.

Cardoso, J.-F., and A. Souloumiac. 1996. "Jacobi Anges for Simultaneous Diagonalization." *SIAM Journal Matrix Analysis and Applications* 17 (1): 161–64.

Clarkson, D.B. 1988. "Remark AS R74: A Least Squares Version of Algorithm AS 211: The F-G Diagonalization Algorithm." *Applied Statistics* 37 (2): 317–21.

De Leeuw, J. 2017. "Jacobi Eigen in R/C with Lower Triangular Column-wise Compact Storage." http://deleeuwpdx.net/pubfolders/jacobi/jacobi.pdf.

De Leeuw, J., and D.B. Ferrari. 2008. "Using Jacobi Plane Rotations in R." Preprint Series 556. Los Angeles, CA: UCLA Department of Statistics. http://deleeuwpdx.net/janspubs/2008/reports/deleeuw_R_08a.pdf.

De Leeuw, J., and S. Pruzansky. 1978. "A New Computational Method to Fit the Weighted Euclidean Distance Model." *Psychometrika* 43: 479–90. http://deleeuwpdx.net/janspubs/1978/articles/deleeuw_pruzansky_A_78.pdf.

R Core Team. 2018. *R: A Language and Environment for Statistical Computing.* Vienna, Austria: R Foundation for Statistical Computing. {https://www.R-project.org/}.