Jacobi Eigen in R and C with Lower Triangular Column-wise Compact Storage

Jan de Leeuw

Version 01, July 11, 2017

Abstract

The Jacobi method for computing eigenvalues and eigenvectors of a symmetric matrix is implemented in C using column-wise compact storage of the lower triangle. The complied C code can be loaded into R using the .C() interface. We compare the C implementation with an earlier version in pure R, and with the built-in eigen function in R.

Contents

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

1 Introduction

In multivariate analysis we often deal with symmetric (and positive semi-definite) matrices. Computer code, both in R or in C, often does not take symmetry into account when storing such objects. Thus a symmetric matrix is stored in memory like any other matrix, with the obvious redundancy all all off-diagonal elements are stired twice. This may be smart from a computational and programming point of view, because all operations on matrices are available without change. But I, for one, have always had a gnawing feeling of unease, because of this blemish of redundancy.

There are, of course, ways to store a symmetric matrix more efficiently. We can store either the upper or lower triangle, and we can store either one row-wise or column-wise. In this paper we have chosen to use the column-wise lower triangle, and we apply the classical iterative Jacobi method to compute eigenvalues and eignevectors to this stored triangle.

2 Implementation

The basic computations are in C. This is done mostly for speed, but also because I like to think of R as a wrapper for compiled C code. The fact that our C code is going to be called from R makes it natural to store our matrices column-wise.

The C code has a number of idiosyncracies. I use small static inline functions, defined and declared in the header file, to compute locations in memory (i.e. to map row and column matrix indices to locations in the triangle, stored column-wise as vector). The C code is written in such a way that it can easily be used on systems where R is not available. There are no R includes and no R functions called. But on the other hand the code follows most of the conventions of the .C() interface in R – i.e. all functions return void, and all arguments are passed as pointers. There is also no I/O and dynamic memory allocation, because I generally prefer to delegate that to the calling system, i.e. our case to R.

The implementation of the Jacobi method does not have any bells and whistles. It cycles through all off-diagonal elements in order. It does not incorporate some well-known ways to use parallelism and sparseness. We have closely followed an earlier implementation in pure R (De Leeuw and Ferrari (2008)). But all in all, I needed something like this for a new R/C implement of smacof which uses the same approach to storing symmetric matrices. There is no attempt to compete with Lapack in terms of speed or robustness. Of course jacobi() has a parameter which determines precision, and this may be useful if only an approximate diagonalization is needed.

3 Example

We first try eigen(), jacobi(), and the R implementation jacobiR() from (**???**) to verify the output is the same. All three routines are applied to a Hilbert matrix of order 4.

```
h <- 1/(outer(1:4,1:4,"+") - 1)
g <- matrix2triangle (h)
eigen(h)
```

```
## eigen() decomposition
## $values
## [1] 1.500214280e+00 1.691412202e-01 6.738273606e-03 9.670230402e-05
##
## $vectors
##               [,1]          [,2]          [,3]           [,4]
## [1,] 0.7926082912  0.5820756995 -0.1791862905 -0.02919332316
```

```
## [2,] 0.4519231209 -0.3705021851  0.7419177906  0.32871205576
## [3,] 0.3224163986 -0.5095786345 -0.1002281369 -0.79141114583
## [4,] 0.2521611697 -0.5140482722 -0.6382825282  0.51455275000
```

```r
jacobi(g)
```

```
## $eval
## [1] 1.500214280e+00 1.691412202e-01 6.738273606e-03 9.670230402e-05
##
## $evec
##                [,1]          [,2]          [,3]           [,4]
## [1,] 0.7926082911 -0.5820756995  0.1791862906 -0.02919332318
## [2,] 0.4519231210  0.3705021851 -0.7419177906  0.32871205578
## [3,] 0.3224163986  0.5095786345  0.1002281370 -0.79141114581
## [4,] 0.2521611696  0.5140482722  0.6382825282  0.51455275003
```

```r
jacobiR(h)
```

```
## $values
## [1] 1.500214280e+00 1.691412202e-01 6.738273606e-03 9.670230402e-05
##
## $vectors
##                [,1]          [,2]          [,3]           [,4]
## [1,] 0.7926082913  0.5820756996  0.1791862916  0.02919331096
## [2,] 0.4519231201 -0.3705021850 -0.7419178134 -0.32871200562
## [3,] 0.3224163988 -0.5095786345  0.1002281901  0.79141113902
## [4,] 0.2521611704 -0.5140482722  0.6382824931 -0.51455279320
```

We use a Hilbert matrix of order 100 to compare the running speed of eigen(), eigen() with symmetric=TRUE, jacobi(), and jacobiR(). For this example, and for the chosen level of precision, eigen() without symmetric is takes about 1.5 times the time that eigen() with symmetric does, jacobi() takes four to five times as long as eigen(), and jacobiR() takes at least 100 times the time of jacobiC().

```r
h <- 1/(outer(1:100,1:100,"+") - 1)
g <- matrix2triangle (h)
microbenchmark (eigen(h), eigen(h, symmetric = TRUE), jacobi(g), jacobiR(h))
```

```
## Unit: microseconds
##                        expr        min          lq         mean
##                    eigen(h)    997.528   1096.7200   1558.17462
##  eigen(h, symmetric = TRUE)    680.777    713.2980    914.11742
##                   jacobi(g)   3459.693   3615.8415   3834.97385
##                  jacobiR(h) 428195.412 447114.0620 464897.94215
##       median          uq        max neval cld
##    1188.6970   1331.7410   5108.599   100   a
##     752.0980    870.7415   4945.969   100   a
```

```
##    3708.8975   3885.5455   6871.676   100   a
##  459867.4815 482148.1105 515549.009   100    b
```

4 Appendix: Code

4.1 R code

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

jacobi <- function (a,
                    itmax = 10,
                    eps = 1e-6,
                    verbose = FALSE) {
  m <- length (a)
  n <- round ((sqrt (1 + 8 * m) - 1) / 2)
  r <- -((1:n) ^ 2) / 2 + (2 * n + 3) * (1:n) / 2 - n
  h <-
    .C(
      "jacobiC",
      as.integer(n),
      eval = as.double (a),
      evec = as.double (rep(0, n * n)),
      as.double (rep(0, n)),
      as.double (rep(0, n)),
      as.integer(itmax),
      as.double(eps)
    )
  return (list (eval = h$eval[r],
                evec = matrix(h$evec, n, n)))
}


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

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(precisi
}

jacobiR <-
  function(a,
           eps1 = 1e-10,
           eps2 = 1e-6,
           itmax = 100,
           vectors = TRUE,
           verbose = FALSE) {
    n <- nrow(a)
    k <- diag(n)
    itel <- 1
    mx <- 0
    saa <- sum(a ^ 2)
    repeat {
      for (i in 1:(n - 1))
        for (j in (i + 1):n) {
          aij <- a[i, j]
          bij <- abs(aij)
          if (bij < eps1)
            next()
          mx <- max(mx, bij)
          am <- (a[i, i] - a[j, j]) / 2
          u <- c(aij, -am)
          u <- u / sqrt(sum(u ^ 2))
          c <- sqrt((1 + u[2]) / 2)
          s <- sign(u[1]) * sqrt((1 - u[2]) / 2)
          ss <- s ^ 2
          cc <- c ^ 2
          sc <- s * c
          ai <- a[i, ]
```

```r
        aj <- a[j, ]
        aii <- a[i, i]
        ajj <- a[j, j]
        a[i, ] <- a[, i] <- c * ai - s * aj
        a[j, ] <- a[, j] <- s * ai + c * aj
        a[i, j] <- a[j, i] <- 0
        a[i, i] <- aii * cc + ajj * ss - 2 * sc * aij
        a[j, j] <- ajj * cc + aii * ss + 2 * sc * aij
        if (vectors) {
          ki <- k[, i]
          kj <- k[, j]
          k[, i] <- c * ki - s * kj
          k[, j] <- s * ki + c * kj
        }
      }
    ff <- sqrt(saa - sum(diag(a) ^ 2))
    if (verbose)
      cat(
        "Iteration ",
        formatC(itel, digits = 4),
        "maxel ",
        formatC(mx, width = 10),
        "loss ",
        formatC(ff, width = 10),
        "\n"
      )
    if ((mx < eps1) || (ff < eps2) || (itel == itmax))
      break()
    itel <- itel + 1
    mx <- 0
  }
  d <- diag(a)
  o <- order(d, decreasing = TRUE)
  if (vectors)
    return(list(values = d[o], vectors = k[, o]))
  else
    return(values = d[o])
}
```

4.2 C code

4.2.1 jacobi.h

```c
#ifndef JACOBI_H
#define JACOBI_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 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; }

#endif /* JACOBI_H */
```

4.2.2 jacobi.c

```
#include "jacobi.h"

void jacobiC(const int *nn, double *a, double *evec, double *oldi, double *oldj,
             int *itmax, 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;
}

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 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;
}

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

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.