

The Classical Jacobi Algorithm

Bill Venables

2015-11-29

Introduction

The Jacobi eigenvalue algorithm

This is a classical algorithm proposed by the mathematician C. G. J. Jacobi in 1846 in connexion with some astronomical computations. See [wikipedia](#) for a detailed description and some historical references.

The method was computationally tedious, and remained dormant until the advent of modern computers in the mid 20th century. Since its re-discovery it has been refined and improved many times, though much faster algorithms have since been devised and implemented.

I first met the Jacobi algorithm as an early Fortran programming exercise I had as a student in 1966. It's simplicity and ingenuity fascinated me then and kindled an interest in numerical computations of this kind that has remained ever since. It was a very good way to learn programming.

Parallel revival

There has been some renewed interest in Jacobi-like methods in recent times, however, since unlike the faster methods for eigensolution computations, it offers the possibility of parallelisation. See, for example, [Zhou and Brent](#) for one possibility, and others in the references therein.

Purpose of this package

This is a **demonstration package** used for teaching purposes. It's main purposes are to provide an example of an intermediate-level programming task where an efficient coding in pure R and one using in C++ using Rcpp are strikingly similar. The task also involves matrix manipulation in *pure* Rcpp, rather than using RcppArmadillo for example, which is of some teaching interest as well.

There are some situations where the C++ function provided, `JacobiCxx`, can be slightly faster than the in-built `eigen` function in the base package, mainly for large numbers of small symmetric matrices. Persons with a fascination for old algorithms might find the comparison with modern versions and alternatives interesting, but generally the functions are **not intended for production use**.

If someone is motivated to take up the challenge of producing a fast parallel Jacobi algorithm coding in R and provide it as a package, there may well be much practical interest (and this package will have served a useful practical purpose, if somewhat vicariously).

Brief synopsis of the algorithm

Let S be a 2×2 symmetric matrix, with entries s_{ij} . It is well known that any symmetric matrix may be diagonalized by an orthogonal similarity transformation. In symbols, for this special case, this implies we need to choose a value for θ for which:

$$H^T S H = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{bmatrix} \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \stackrel{\text{def.}}{=} \Lambda$$

A solution is easily shown to be

$$\theta = \begin{cases} \frac{1}{2} \arctan\left(\frac{2s_{12}}{s_{22}-s_{11}}\right) & \text{if } s_{11} \neq s_{22} \\ \frac{\pi}{4} & \text{if } s_{11} = s_{22} \end{cases}$$

Note that both cases can be accommodated via the R function `atan2`.

In the general case a series of rotation matrices is chosen and applied successively. These have the same form as the 2×2 case, but embedded in an $n \times n$ identity matrix, so the application of any one of them affects two rows and columns *only*. Such *planar rotation matrices* are chosen so that at any stage the off-diagonal element with *maximum* absolute value is annihilated.

Hence if at some stage $|s_{ij}|$, ($i < j$), is maximum, the planar rotation matrix H_{ij} will affect rows and columns i and j only, and will transform s_{ij} to zero, and the process continues.

The process ceases when the $\max_{i < j} |s_{ij}| < \epsilon$, where $\epsilon > 0$ is some small pre-set tolerance.¹

Elements that are annihilated at some stage may become non-zero at later stages, of course, but several properties of the algorithm are guaranteed, namely

- At any stage the sum of squares of the off-diagonal elements is reduced, eventually to zero, and
- The rate of convergence is quadratic, so the algorithm is *relatively* quick.

At the end of the algorithm, the original symmetric matrix S is transformed into the diagonal matrix of eigenvalues, Λ . If eigenvectors are also required then the accumulated product of the planar rotation matrices, starting with the identity, provide a normalized version of them:

$$H = H_{i_p, j_p} \cdots H_{i_3, j_3} H_{i_2, j_2} H_{i_1, j_1} I_n$$

Examples

For a simple example, consider finding the eigenvalues and eigenvectors of a well-known correlation matrix.

```
suppressMessages(library(dplyr))
library(JacobiEigen)
library(stats)

imod <- aov(cbind(Sepal.Length, Sepal.Width, Petal.Length, Petal.Width) ~ Species, iris)
(R <- cor(resid(imod)))
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
Sepal.Length	1.0000000	0.5302358	0.7561642	0.3645064
Sepal.Width	0.5302358	1.0000000	0.3779162	0.4705346
Petal.Length	0.7561642	0.3779162	1.0000000	0.4844589
Petal.Width	0.3645064	0.4705346	0.4844589	1.0000000

```
rEig <- JacobiR(R)
cEig <- Jacobi(R)
identical(rEig, cEig) ## the R and Rcpp implementations are identical
```

```
[1] TRUE
```

```
cEig
```

```
$values
```

```
[1] 2.5037618 0.7251373 0.5824012 0.1886997
```

```
$vectors
```

```
      [,1]      [,2]      [,3]      [,4]
```

¹If only eigenvalues are required, the tolerance can be set somewhat higher than if accurate eigenvectors are required as well.

```
[1,] 0.5423991 -0.4569743 -0.2149752 0.6713892
[2,] 0.4663824 0.4664664 -0.6965582 -0.2823176
[3,] 0.5348347 -0.4534110 0.3139268 -0.6401720
[4,] 0.4497138 0.6066317 0.6083110 0.2443627
```

```
(eEig <- eigen(R))
```

```
$values
```

```
[1] 2.5037618 0.7251373 0.5824012 0.1886997
```

```
$vectors
```

```
      [,1]      [,2]      [,3]      [,4]
[1,] -0.5423991 0.4569743 -0.2149752 0.6713892
[2,] -0.4663824 0.4664664 -0.6965582 -0.2823176
[3,] -0.5348347 0.4534110 0.3139268 -0.6401720
[4,] -0.4497138 -0.6066317 0.6083110 0.2443627
```

```
all.equal(eEig$values, cEig$values) ## eigenvalues are (practically) identical
```

```
[1] TRUE
```

```
crossprod(eEig$vectors, cEig$vectors) %>% ## eigenvectors differ in signs
round(10)
```

```
      [,1] [,2] [,3] [,4]
[1,]  -1    0    0    0
[2,]   0   -1    0    0
[3,]   0    0    1    0
[4,]   0    0    0    1
```

We can now look at some timings.

```
library(microbenchmark)
microbenchmark(JacobiR(R), Jacobi(R), eigen(R))
```

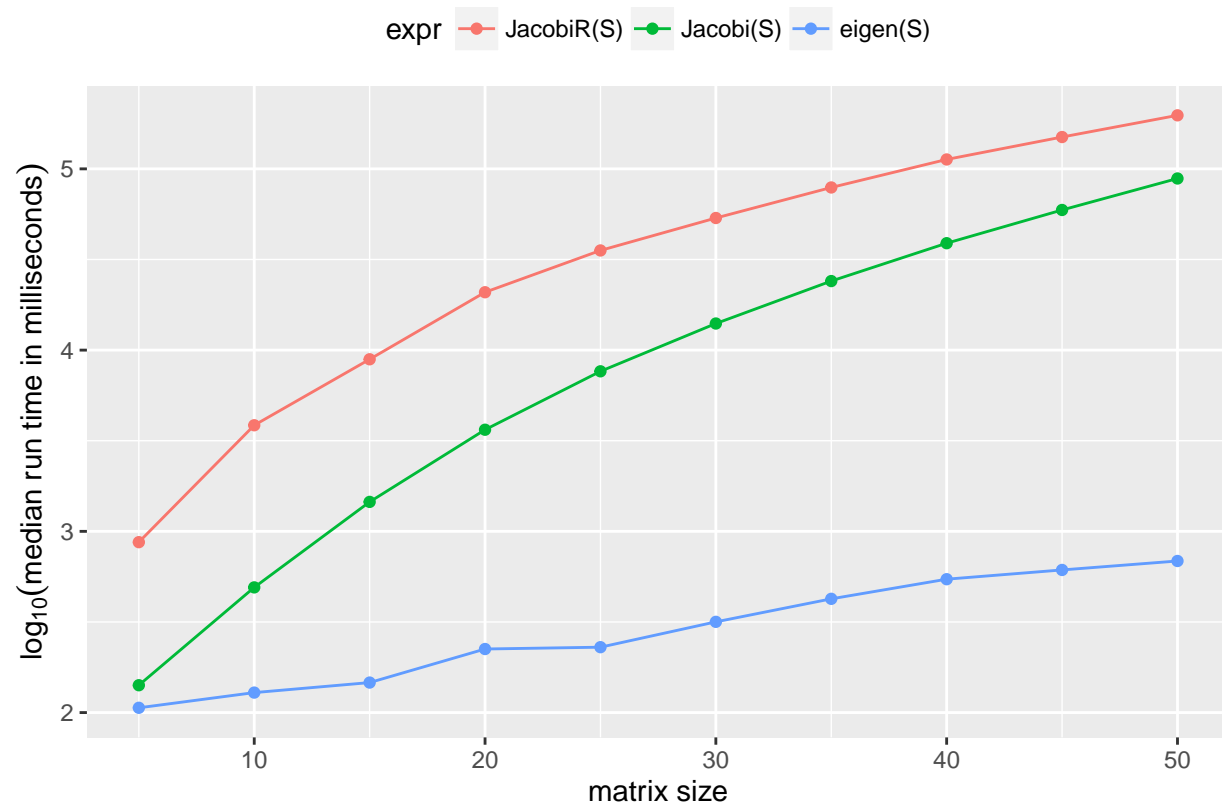
```
Unit: microseconds
```

	expr	min	lq	mean	median	uq	max	neval	cld
	JacobiR(R)	693.496	723.8545	798.7085	771.0470	797.0330	2537.186	100	c
	Jacobi(R)	135.777	144.7300	152.2367	150.5420	158.6445	187.141	100	b
	eigen(R)	87.999	97.7455	103.9259	102.1905	106.8565	216.933	100	a

The apparen advantage of Jacobi rapidly diminishes as the size of the matrix increases:

```
suppressMessages(library(tidyr))
set.seed(1234)
N <- 100
iseq <- seq(5, 50, by = 5)
res <- lapply(iseq, function(n) {
  S <- crossprod(matrix(rnorm(N*n), N, n))/N
  runTime <- microbenchmark(JacobiR(S), Jacobi(S), eigen(S), times = 20)
  c(n = n, with(runTime, tapply(time, expr, median))/1000)
}) %>%
  do.call(rbind, .) %>%
  as.data.frame %>%
  gather(key = expr, value = time, `JacobiR(S)`, `Jacobi(S)`, `eigen(S)`)

suppressMessages(library(ggplot2))
ggplot(res) + aes(x = n, y = log10(time), colour = expr) + geom_line() + geom_point() +
  theme(legend.position = "top") + xlab("matrix size") +
  ylab(expression(log[10]("median run time in milliseconds")))
```



Code

For referece, the R and Rcpp code are listed below.

R

This includes the interface to the Rcpp function.

```
JacobiR <- function(x, symmetric = isTRUE(all.equal(x, t(x))),
                    only.values = FALSE,
                    eps = if(!only.values) .Machine$double.eps else
                        sqrt(.Machine$double.eps)) {
  if(!symmetric)
    stop("only real symmetric matrices are allowed")
  n <- nrow(x)
  H <- if(only.values) NULL else diag(n)
  eps <- max(eps, .Machine$double.eps)

  if(n > 1) {
    lt <- which(lower.tri(x))

    repeat {
      k <- lt[which.max(abs(x[lt]))] ## the matrix element
      j <- floor(1 + (k - 2)/(n + 1)) ## the column
      i <- k - n * (j - 1)           ## the row

      if(abs(x[i, j]) < eps) break

      Si <- x[, i]
      Sj <- x[, j]

      theta <- 0.5*atan2(2*Si[j], Sj[j] - Si[i])
      c <- cos(theta)
      s <- sin(theta)

      x[i, ] <- x[, i] <- c*Si - s*Sj
      x[j, ] <- x[, j] <- s*Si + c*Sj
      x[i,j] <- x[j,i] <- 0
      x[i,i] <- c^2*Si[i] - 2*s*c*Si[j] + s^2*Sj[j]
      x[j,j] <- s^2*Si[i] + 2*s*c*Si[j] + c^2*Sj[j]
      if(!only.values) {
        Hi <- H[, i]
        H[, i] <- c*Hi - s*H[, j]
        H[, j] <- s*Hi + c*H[, j]
      }
    }
  }
  list(values = as.vector(diag(x)), vectors = H)
}

##
## The interface function to the Rcpp code
##
Jacobi <- function(x, symmetric = isTRUE(all.equal(x, t(x))),
                  only.values = FALSE, eps = 0.0) {
  if(!symmetric)
    stop("only real symmetric matrices are allowed")
  .Call('JacobiEigen_JacobiCpp', PACKAGE = 'JacobiEigen', x, only.values, eps)
}
```

Rcpp

We begin with one helper function:

```
#include <Rcpp.h>
using namespace Rcpp;

NumericMatrix Ident(int n) // not exported.
{
    NumericMatrix I(n, n);
    for(int i = 0; i < n; i++) I(i, i) = 1.0;
    return I;
}

// [[Rcpp::export]]
List JacobiCpp(NumericMatrix x, bool only_values = false, double eps = 0.0)
{
    NumericMatrix S(clone(x));
    int nr = S.nrow();
    bool vectors = !only_values;
    NumericMatrix H;

    if(vectors) {
        H = Ident(nr);
    }

    double eps0 = as<double>((as<List>(Environment::base_env())[".Machine"]))["double.eps"]);
    double tol = eps > eps0 ? eps : eps0; // i.e. no lower than .Machine$double.eps
    if(only_values & (eps == 0.0)) tol = sqrt(tol); // a lower accuracy is adequate here.

    while(true) {
        double maxS = 0.0;
        int i=0, j=0;
        for(int row = 1; row < nr; row++) { // find value & position of maximum |off-diagonal|
            for(int col = 0; col < row; col++) {
                double val = fabs(S(row, col));
                if(maxS < val) {
                    maxS = val;
                    i = row;
                    j = col;
                }
            }
        }
        if(maxS <= tol) break;

        NumericVector Si = S(_, i), Sj = S(_, j);

        double theta = 0.5*atan2(2.0*Si(j), Sj(j) - Si(i));
        double s = sin(theta), c = cos(theta);

        S(i, _) = S(_, i) = c*Si - s*Sj;
        S(j, _) = S(_, j) = s*Si + c*Sj;
        S(i, j) = S(j, i) = 0.0;
        S(i, i) = c*c*Si(i) - 2.0*s*c*Si(j) + s*s*Sj(j);
        S(j, j) = s*s*Si(i) + 2.0*s*c*Si(j) + c*c*Sj(j);
    }
}
```

```

    if(vectors) {
        NumericVector Hi = H(_, i);
        H(_, i) = c*Hi - s*H(_, j);
        H(_, j) = s*Hi + c*H(_, j);
    }
}
if(vectors) {
    return List::create(_["values"] = diag(S),
                        _["vectors"] = H);
} else {
    return List::create(_["values"] = diag(S),
                        _["vectors"] = R_NilValue);
}
}

```