Timing Rayleigh Quotient minimization in R

John C. Nash, Telfer School of Management, University of Ottawa

July 2012

Abstract

This vignette is simply to record the methods and results for timing various Rayleigh Quotient minimizations with R using different functions and different ways of running the computations, in particular trying Fortran subroutines and the R byte compiler.

1 The computational task

The maximal and minimal eigensolutions of a symmetric matrix A are extrema of the Rayleigh Quotient
$R(x) = (x'Ax)/(x'x)$
We could also deal with generalized eigenproblems of the form
$Ax = eBx$
where B is symmetric and positive definite by using the Rayleigh Quotient (RQ)
$R_g(x) = (x'Ax)/(x'Bx)$
In this document, B will always be an identity matrix, but some programs we test assume that it is present.

Noth that the objective is scaled by the parameters, in fact by by their sum of squares. Alternatively, we may think of requiring the **normalized** eigensolution, which is given as
$x_{normalized} = x/sqrt(x'x)$

2 Timings and speedups

In R , execution times can be measured by the function `system.time`, and in particular the third element of the object this function returns. However, various factors influence computing times in a modern computational system, so we generally want to run replications of the times. The R packages `rbenchmark` and `microbenchmark` can be used for this. I have a preference for the latter. However, to keep the time to prepare this vignette with `Sweave` or `knitR` reasonable, many of the timings will be done with only `system.time`.

There are some ways to speed up R computations.

- The code can be modified to use more efficient language structures. We show some of these below, in particular, to use vector operations.
- We can use the R byte code compiler by Luke Tierney, which has been part of the R distribution since version 2.14.
- We can use compiled code in other languages. Here we show how Fortran subroutines can be used.

3 Our example matrix

We will use a matrix called the Moler matrix (Nash 1979, Appendix 1). This is a positive definite symmetric matrix with one small eigenvalue. We will show a couple of examples of computing the small eigenvalue solution, but will mainly perform timings using the maximal eigenvalue solution, which we will find by minimizing the RQ of (-1) times the matrix. (The eigenvalue of this matrix is the negative of the maximal eigenvalue of the original, but the eigenvectors are equivalent to within a scaling factor for non-degenerate eigenvalues.)

Here is the code for generating the Moler matrix.

```
molermat <- function(n) {
    A <- matrix(NA, nrow = n, ncol = n)
    for (i in 1:n) {
        for (j in 1:n) {
            if (i == j)
                A[i, i] <- i else A[i, j] <- min(i, j) - 2
        }
    }
    A
}
```

However, since R is more efficient with vectorized code, the following routine by Ravi Varadhan should do much better.

```
molerfast <- function(n) {
    # A fast version of `molermat'
    A <- matrix(0, nrow = n, ncol = n)
    j <- 1:n
    for (i in 1:n) {
        A[i, 1:i] <- pmin(i, 1:i) - 2
    }
    A <- A + t(A)
    diag(A) <- 1:n
    A
}
```

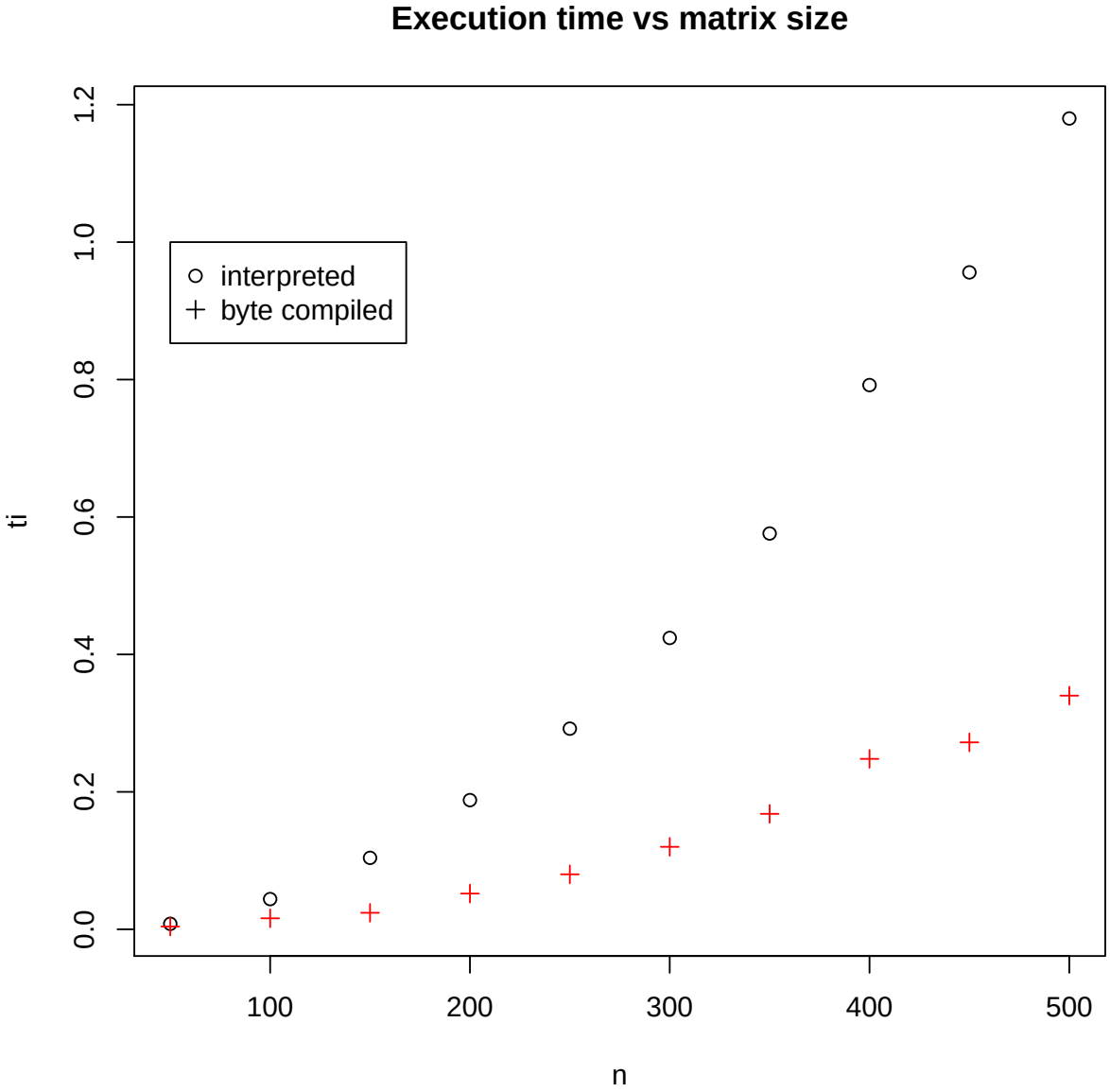
3.1 Time to build the matrix

Let us see how long it takes to build the Moler matrix of different sizes. However, given that it is easy to use the byte-code compiler, we will compare results. We also include times for the `eigen()` function that computes the full set of eigensolutions very quickly.

```
## Loading required package: compiler
```

```
##      n buildi buildc   osize eigentime bfast bfastc
## 1   50  0.008  0.004   20112     0.000 0.000  0.000
## 2  100  0.044  0.016   80112     0.004 0.000  0.000
## 3  150  0.104  0.024  180112     0.012 0.004  0.000
## 4  200  0.188  0.052  320112     0.020 0.004  0.004
## 5  250  0.292  0.080  500112     0.032 0.004  0.004
## 6  300  0.424  0.120  720112     0.056 0.008  0.008
## 7  350  0.576  0.168  980112     0.084 0.008  0.012
## 8  400  0.792  0.248 1280112     0.112 0.012  0.012
## 9  450  0.956  0.272 1620112     0.176 0.016  0.020
## 10 500  1.180  0.340 2000112     0.272 0.020  0.024
## buildi - interpreted build time; buildc - byte compiled build time
## osize - matrix size in bytes; eigentime - all eigensolutions time
## bfast - interpreted vectorized build time; bfastc - same code, byte compiled time
```

We can graph the times, and show a definite advantage for using the byte code compiler. The code, which is not echoed here, also models the times and the object size created as almost perfect quadratic models in `n`. However, the vectorized code is much, much faster, and the byte code compiler does not appear to help.



Regular Moler matrix routine, interpreted and byte compiled

```
##
## Call:
## lm(formula = ti ~ n + n2)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.007473 -0.006355 -0.003179  0.000265  0.030600
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -9.53e-03   1.51e-02   -0.63     0.55
## n            6.31e-05   1.26e-04    0.50     0.63
## n2           4.66e-06   2.23e-07   20.87  1.5e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.0128 on 7 degrees of freedom
```

```
## Multiple R-squared: 0.999,Adjusted R-squared: 0.999
## F-statistic: 4.54e+03 on 2 and 7 DF,  p-value: 1.27e-11
##
##
## Call:
## lm(formula = tc ~ n + n2)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.00830 -0.00572 -0.00316  0.00179  0.02666
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) -2.73e-03   1.36e-02   -0.20  0.84592
## n            9.88e-06   1.13e-04    0.09  0.93291
## n2           1.38e-06   2.01e-07    6.86  0.00024 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.0115 on 7 degrees of freedom
## Multiple R-squared: 0.993,Adjusted R-squared: 0.991
## F-statistic:  480 on 2 and 7 DF,  p-value: 3.23e-08
##
##
## Call:
## lm(formula = os ~ n + n2)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -1.73e-10 -3.30e-11 -2.29e-11  1.34e-11  2.41e-10
##
## Coefficients:
##             Estimate Std. Error  t value Pr(>|t|)
## (Intercept) 1.12e+02   1.39e-10 8.04e+11   <2e-16 ***
## n           0.00e+00   1.16e-12 0.00e+00        1
## n2          8.00e+00   2.06e-15 3.88e+15   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.18e-10 on 7 degrees of freedom
## Multiple R-squared:    1,Adjusted R-squared:    1
## F-statistic: 1.5e+32 on 2 and 7 DF,  p-value: <2e-16
##
```

To verify the relative timings for much larger matrices, we report timings on the same 64 bit 3GHz PC running Ubuntu Linux 10.04 that took 1.184 s to build a matrix of size 500 by the original `molermat` interpreted function.

n	treg	tregc	tfast	tfastc
2500	28.686	8.493	0.496	0.464
5000	116.719	34.086	2.364	2.189

4 Computing the Rayleigh Quotient

The Rayleigh Quotient requires the quadratic form $x'Ax$ divided by the inner product $x'x$. R lets us form this in several ways. Given that we know `for` loops are slow, we will not actually use the direct code (incorporating the minus sign)

```
rqdir <- function(x, AA) {
    rq <- 0
    n <- length(x)  # assume x, AA conformable
    for (i in 1:n) {
        for (j in 1:n) {
            rq <- rq - x[i] * AA[[i, j]] * x[j]  # Note - sign
        }
    }
    rq
}
```

Somewhat better (as we shall show below) is

```
ray1 <- function(x, AA) {
    rq <- -t(x) %*% AA %*% x
}
```

and better still is

```
ray2 <- function(x, AA) {
    rq <- -as.numeric(crossprod(x, crossprod(AA, x)))
}
```

Note that we include the minus sign already in these routines.

If we already have the inner product Ax as `ax` from some other computation, then we can simply use

```
ray3 <- function(x, AA, ax = axftn) {
    # ax is a function to form AA%*%x
    rq <- -as.numeric(crossprod(x, ax(x, AA)))
}
```

5 Matrix-vector products

In generating the RQ, we do not actually need the matrix itself, but simply the inner product with a vector `x`, from which a second inner produce with `x` gives us the quadratic form $x'Ax$. If `n` is the order of the problem, then for large `n`, we avoid storing and manipulating a very large matrix if we use **implicit inner product** formation. We do this with the following code. For future reference, we include the multiplication by an identity.

```
ax <- function(x, AA) {
    u <- as.numeric(AA %*% x)
}

axx <- function(x, AA) {
    u <- as.numeric(crossprod(AA, x))
}
```

Note that second argument, supposedly communicating the matrix which is to be used in the matrix-vector product, is ignored in the following implicit product routine. It is present only to provide a common syntax when we wish to try different routines within other computations.

```
aximp <- function(x, AA = 1) {
    # implicit moler A*x
    n <- length(x)
    y <- rep(0, n)
    for (i in 1:n) {
        tt <- 0
        for (j in 1:n) {
            if (i == j)
                tt <- tt + i * x[i] else tt <- tt + (min(i, j) - 2) * x[j]
        }
        y[i] <- tt
    }
    y
}
ident <- function(x, B = 1) x  # identity
```

However, Ravi Varadhan has suggested the following vectorized code for the implicit matrix-vector product.

```
axmolerfast <- function(x, AA = 1) {
    # A fast and memory-saving version of A%*%x For Moler matrix. Note we need
    # a matrix argument to match other functions
    n <- length(x)
```

```
    j <- 1:n
    ax <- rep(0, n)
    for (i in 1:n) {
        term <- x * (pmin(i, j) - 2)
        ax[i] <- sum(term[-i])
    }
    ax <- ax + j * x
    ax
}
```

We can also use external language routines, for example in Fortran. However, this needs a Fortran **subroutine** which outputs the result as one of the returned components. The subroutine is in file `moler.f`.

```
      subroutine moler(n, x, ax)
      integer n, i, j
      double precision x(n), ax(n), sum
c     return ax = A * x for A = moler matrix
c     A[i,j]=min(i,j)-2 for i<>j, or i for i==j
      do 20 i=1,n
         sum=0.0
         do 10 j=1,n
            if (i.eq.j) then
               sum = sum+i*x(i)
            else
               sum = sum+(min(i,j)-2)*x(j)
            endif
 10      continue
         ax(i)=sum
 20   continue
      return
      end
```

This is then compiled in a form suitable for R use by the command (this is a command-line tool, and was run in Ubuntu Linux in a directory containing the file `moler.f` but outside this vignette):

```
R CMD SHLIB moler.f
```

This creates files `moler.o` and `moler.so`, the latter being the dynamically loadable library we need to bring into our R session.

```
dyn.load("moler.so")
cat("Is the mat multiply loaded? ", is.loaded("moler"), "\n")

## Is the mat multiply loaded?  TRUE


axftn <- function(x, AA = 1) {
    # ignore second argument
    n <- length(x)  # could speed up by having this passed
    vout <- rep(0, n)  # purely for storage
    res <- (.Fortran("moler", n = as.integer(n), x = as.double(x), vout = as.double(vout)))$vout
}
```

We can also byte compile each of the routines above

```
require(compiler)
axc <- cmpfun(ax)
axxc <- cmpfun(axx)
axftnc <- cmpfun(axftn)
aximpc <- cmpfun(aximp)
axmfc <- cmpfun(axmolerfast)
```

Now it is possible to time the different approaches to the matrix-vector product.

```
dyn.load("moler.so")
cat("Is the mat multiply loaded? ", is.loaded("moler"), "\n")

## Is the mat multiply loaded?  TRUE

require(microbenchmark)
nmax <- 10
ptable <- matrix(NA, nrow = nmax, ncol = 11)  # to hold results
# loop over sizes
for (ni in 1:nmax) {
    n <- 50 * ni
    x <- runif(n)  # generate a vector
    ptable[[ni, 1]] <- n
    AA <- molermat(n)
    tax <- system.time(oax <- replicate(100, ax(x, AA))[, 1])[[1]]
    taxc <- system.time(oaxc <- replicate(100, axc(x, AA))[, 1])[[1]]
    if (!identical(oax, oaxc))
        stop("oaxc NOT correct")
    taxx <- system.time(oaxx <- replicate(100, axx(x, AA))[, 1])[[1]]
    if (!identical(oax, oaxx))
        stop("oaxx NOT correct")
    taxxc <- system.time(oaxxc <- replicate(100, axxc(x, AA))[, 1])[[1]]
    if (!identical(oax, oaxxc))
        stop("oaxxc NOT correct")
    taxftn <- system.time(oaxftn <- replicate(100, axftn(x, AA = 1))[, 1])[[1]]
    if (!identical(oax, oaxftn))
        stop("oaxftn NOT correct")
    taxftnc <- system.time(oaxftnc <- replicate(100, axftnc(x, AA = 1))[, 1])[[1]]
    if (!identical(oax, oaxftnc))
        stop("oaxftnc NOT correct")
    taximp <- system.time(oaximp <- replicate(100, aximp(x, AA = 1))[, 1])[[1]]
    if (!identical(oax, oaximp))
        stop("oaximp NOT correct")
    taximpc <- system.time(oaximpc <- replicate(100, aximpc(x, AA = 1))[, 1])[[1]]
    if (!identical(oax, oaximpc))
        stop("oaximpc NOT correct")
    taxmfi <- system.time(oaxmfi <- replicate(100, axmolerfast(x, AA = 1))[,
        1])[[1]]
    if (!identical(oax, oaxmfi))
        stop("oaxmfi NOT correct")
    taxmfc <- system.time(oaxmfc <- replicate(100, axmfc(x, AA = 1))[, 1])[[1]]
    if (!identical(oax, oaxmfc))
        stop("oaxmfc NOT correct")
    ptable[[ni, 2]] <- tax
    ptable[[ni, 3]] <- taxc
    ptable[[ni, 4]] <- taxx
    ptable[[ni, 5]] <- taxxc
    ptable[[ni, 6]] <- taxftn
    ptable[[ni, 7]] <- taxftnc
    ptable[[ni, 8]] <- taximp
    ptable[[ni, 9]] <- taximpc
    ptable[[ni, 10]] <- taxmfi
    ptable[[ni, 11]] <- taxmfc
    # cat(n,tax, taxc, taxx, taxxc, taxftn, taxftnc, taximp, taximpc,'\n')
}
axtym <- data.frame(n = ptable[, 1], ax = ptable[, 2], axc = ptable[,
    3], axx = ptable[, 4], axxc = ptable[, 5], axftn = ptable[, 6], axftnc = ptable[,
    7], aximp = ptable[, 8], aximpc = ptable[, 9], axmfast = ptable[, 10], amfastc = ptable[,
    11])
print(axtym)

##       n    ax   axc   axx  axxc axftn axftnc   aximp aximpc axmfast amfastc
## 1    50 0.004 0.004 0.000 0.004 0.004  0.000   1.016  0.280   0.120   0.128
## 2   100 0.008 0.008 0.000 0.000 0.004  0.004   4.040  1.168   0.272   0.228
## 3   150 0.020 0.020 0.000 0.004 0.008  0.004   9.113  2.584   0.460   0.416
## 4   200 0.032 0.036 0.008 0.004 0.012  0.008  15.993  4.592   0.696   0.576
## 5   250 0.052 0.056 0.008 0.012 0.012  0.012  25.150  7.176   0.908   0.808
## 6   300 0.076 0.080 0.016 0.016 0.020  0.016  36.226 10.252   1.216   1.085
## 7   350 0.108 0.104 0.024 0.020 0.024  0.024  49.583 13.973   1.504   1.344
## 8   400 0.140 0.140 0.028 0.028 0.028  0.032  64.640 18.253   1.860   1.672
## 9   450 0.176 0.176 0.036 0.032 0.040  0.040  82.410 23.605   2.312   2.088
## 10 500 0.220 0.220 0.044 0.044 0.044  0.048 101.603 28.870   2.660   2.456
```

From the above output, we see that the `crossprod` variant of the matrix-vector product appears to be

the fastest. However, we have omitted the time to build the matrix. If we must build the matrix, then we need somehow to include that time. Because the times for the matrix-vector product were so short, we used `replicate` above to run 100 copies of the same calculation, which may give some distortion of the timings. However, we believe the scale of the times is more or less correct. To compare these times to the times for the Fortran or implicit matrix-vector routines, we should add a multiple of the relevant interpreted or compiled build times. Here we have used the times for the rather poor `molermat()` function, but this is simply to illustrate the range of potential timings. Apportioning such "fixed costs" to timings is never a trivial decision. Similarly if, where and how to store large matrices if we do build them, and whether it is worth building them more than once if storage is an issue, are all questions that may need to be addressed if performance becomes important.

```
adjtym <- data.frame(n = axtym$n, axx1 = axtym$axx + 1 * bmattym$buildi,
    axxz = axtym$axx + 100 * bmattym$buildi, axxc1 = axtym$axxc + 1 * bmattym$buildc,
    axxcz = axtym$axxc + 100 * bmattym$buildc, axftn = axtym$axftn, aximp = axtym$aximp,
    aximpc = axtym$aximpc)
print(adjtym)

##      n  axx1   axxz axxc1  axxcz axftn  aximp aximpc
## 1   50 0.008   0.80 0.004  0.400 0.004  1.008  0.292
## 2  100 0.044   4.40 0.016  1.600 0.004  3.973  1.144
## 3  150 0.108  10.40 0.028  2.404 0.004  9.028  2.672
## 4  200 0.196  18.81 0.060  5.208 0.008 16.237  4.652
## 5  250 0.304  29.21 0.088  8.008 0.012 25.109  7.153
## 6  300 0.440  42.42 0.136 12.016 0.020 35.666 10.257
## 7  350 0.596  57.62 0.188 16.820 0.024 48.635 13.925
## 8  400 0.820  79.23 0.276 24.828 0.032 63.536 18.165
## 9  450 0.992  95.64 0.308 27.236 0.036 80.225 23.033
## 10 500 1.224 118.04 0.384 34.044 0.048 99.346 28.546
```

Out of all this, we see that the Fortran implicit matrix-vector product is the overall winner at all values of `n`. Moreover, it does NOT require the creation and storage of the matrix. However, using Fortran does involve rather more work for the user, and for most applications it is likely we could live with the use of either

- the interpreted matrix-product based on `crossprod` and an actual matrix is good enough, especially if a fast matrix build is used and we have plenty of memory, or
- the interpreted or byte-code compiled implicit matrix-vector multiply `axmolerfast`.

6 RQ computation times

We have in 4 set up three versions of a Rayleigh Quotient calculation in addition to the direct form. The third form is set up to use the `axftn` routine that we have already shown is efficient. We could also use this with the implicit matrix-vector product `axmolerfast`.

It seems overkill to show the RQ computation time for all versions and matrices, so we will do the timing simply for a matrix of order 500.

```
require(compiler)
rqdirc <- cmpfun(rqdir)
ray1c <- cmpfun(ray1)
ray2c <- cmpfun(ray2)
ray3c <- cmpfun(ray3)
dyn.load("moler.so")
n <- 500
x <- runif(n)  # generate a vector
AA <- molermat(n)
tdi <- system.time(rdi <- replicate(100, rqdir(x, AA))[1])[[1]]
tdc <- system.time(replicate(100, rdc <- rqdirc(x, AA))[1])[[1]]
cat("Direct algorithm: interpreted=", tdi, "   byte-compiled=", tdc,
    "\n")

## Direct algorithm: interpreted= 78.62    byte-compiled= 18.3
```

```
t1i <- system.time(replicate(100, r1i <- ray1(x, AA))[1])[[1]]
t1c <- system.time(replicate(100, r1c <- ray1c(x, AA))[1])[[1]]
cat("ray1: mat-mult algorithm: interpreted=", t1i, "   byte-compiled=",
    t1c, "\n")

## ray1: mat-mult algorithm: interpreted= 0.232    byte-compiled= 0.224

t2i <- system.time(replicate(100, r2i <- ray2(x, AA))[1])[[1]]
t2c <- system.time(replicate(100, r2c <- ray2c(x, AA))[1])[[1]]
cat("ray2: crossprod algorithm: interpreted=", t2i, "   byte-compiled=",
    t2c, "\n")

## ray2: crossprod algorithm: interpreted= 0.04    byte-compiled= 0.044

t3fi <- system.time(replicate(100, r3i <- ray3(x, AA, ax = axftn))[1])[[1]]
t3fc <- system.time(replicate(100, r3i <- ray3c(x, AA, ax = axftnc))[1])[[1]]
cat("ray3: ax Fortran + crossprod: interpreted=", t3fi, "   byte-compiled=",
    t3fc, "\n")

## ray3: ax Fortran + crossprod: interpreted= 0.048    byte-compiled= 0.044

t3ri <- system.time(replicate(100, r3i <- ray3(x, AA, ax = axmolerfast))[1])[[1]]
t3rc <- system.time(replicate(100, r3i <- ray3c(x, AA, ax = axmfc))[1])[[1]]
cat("ray3: ax fast R implicit + crossprod: interpreted=", t3ri, "   byte-compiled=",
    t3rc, "\n")

## ray3: ax fast R implicit + crossprod: interpreted= 2.66    byte-compiled= 2.432
```

Here we see that the use of the `crossprod` in `ray2` is very fast, and this is interpreted code. Once again, we note that all timings except those for `ray3` should have some adjustment for the building of the matrix. If storage is an issue, then `ray3`, which uses the implicit matrix-vector product in Fortran, is the approach of choice. My own preference would be to use this option if the Fortran matrix-vector product subroutine is already available for the matrix required. I would not, however, generally choose to write the Fortran subroutine for a "new" problem matrix. The fast implicit matrix-vector tool with `ray3` is also useful and quite fast if we need to minimize memory use.

7 Solution by spg

To actually solve the eigensolution problem we will first use the projected gradient method `spg` from `BB`. We repeat the RQ function so that it is clear which routine we are using.

```
rqt <- function(x, AA) {
    rq <- as.numeric(crossprod(x, crossprod(AA, x)))
}
proj <- function(x) {
    x/sqrt(crossprod(x))
}
require(BB)
n <- 100
x <- rep(1, n)
AA <- molermat(n)
evs <- eigen(AA)
tmin <- system.time(amin <- spg(x, fn = rqt, project = proj, control = list(trace = FALSE),
    AA = AA))[[1]]
# amin
tmax <- system.time(amax <- spg(x, fn = rqt, project = proj, control = list(trace = FALSE),
    AA = -AA))[[1]]
# amax
evalmax <- evs$values[1]
evecmax <- evs$vectors[, 1]
evecmax <- sign(evecmax[1]) * evecmax/sqrt(as.numeric(crossprod(evecmax)))
emax <- list(evalmax = evalmax, evecmac = evecmax)
save(emax, "temax.Rdata")

## Error: object 'temax.Rdata' not found

evalmin <- evs$values[n]
```

```
evecmin <- evs$vectors[, n]
evecmin <- sign(evecmin[1]) * evecmin/sqrt(as.numeric(crossprod(evecmin)))
avecmax <- amax$par
avecmin <- amin$par
avecmax <- sign(avecmax[1]) * avecmax/sqrt(as.numeric(crossprod(avecmax)))
avecmin <- sign(avecmin[1]) * avecmin/sqrt(as.numeric(crossprod(avecmin)))
cat("minimal eigensolution: Value=", amin$value, "in time ", tmin,
    "\n")

## minimal eigensolution: Value= 8.386e-08 in time  0.928

cat("Eigenvalue - result from eigen=", amin$value - evalmin, "  vector max(abs(diff))=",
    max(abs(avecmin - evecmin)), "\n\n")

## Eigenvalue - result from eigen= 8.386e-08   vector max(abs(diff))= 0.0001617
##

# print(amin$par)
cat("maximal eigensolution: Value=", -amax$value, "in time ", tmax,
    "\n")

## maximal eigensolution: Value= 3934 in time  0.02

cat("Eigenvalue - result from eigen=", -amax$value - evalmax, "  vector max(abs(diff))=",
    max(abs(avecmax - evecmax)), "\n\n")

## Eigenvalue - result from eigen= -1.054e-06   vector max(abs(diff))= 2.508e-06
##

# print(amax$par)
```

```
##       n spgrqt spgcrqtcaxc tbldc
## 1    50  0.008       0.004 0.004
## 2   100  0.016       0.012 0.012
## 3   150  0.044       0.036 0.032
## 4   200  0.084       0.076 0.056
## 5   250  0.184       0.164 0.120
## 6   300  0.324       0.304 0.124
## 7   350  0.492       0.468 0.164
## 8   400  0.712       0.688 0.216
## 9   450  1.008       0.968 0.276
## 10 500  1.368       1.344 0.340
```

8 Solution by other optimizers

We can try other optimizers, but we must note that unlike `spg` they do not take account of the scaling. However, we can build in a transformation, since our function is always the same for all sets of parameters scaled by the square root of the parameter inner product. The function `nobj` forms the quadratic form that is the numerator of the Rayleigh Quotient using the more efficient `crossprod()` function

`rq<- as.numeric(crossprod(y, crossprod(AA,y)))`

but we first form

`y<-x/sqrt(as.numeric(crossprod(x)))`

to scale the parameters.

Since we are running a number of gradient-based optimizers in the wrapper `optimx`, we have reduced the matrix sizes and numbers.

```
nobj <- function(x, AA = -AA) {
    y <- x/sqrt(as.numeric(crossprod(x)))
    rq <- as.numeric(crossprod(y, crossprod(AA, y)))
}

ngrobj <- function(x, AA = -AA) {
    y <- x/sqrt(as.numeric(crossprod(x)))
    n <- length(x)
```

```
    dd <- sqrt(as.numeric(crossprod(x)))
    T1 <- diag(rep(1, n))/dd
    T2 <- x %o% x/(dd * dd * dd)
    gt <- T1 - T2
    gy <- as.vector(2 * crossprod(AA, y))
    gg <- as.numeric(crossprod(gy, gt))
}
require(optplus)

## Loading required package: optplus
## Loading required package: numDeriv
## Attaching package: 'optplus'
## The following object(s) are masked from 'package:numDeriv':
##
## grback, grcentral, grfwd, grnd

# mset<-c('L-BFGS-B', 'BFGS', 'CG', 'spg', 'ucminf', 'nlm', 'nlminb',
# 'Rvmmin', 'Rcgmin')
mset <- c("L-BFGS-B", "BFGS", "CG", "spg", "ucminf", "nlm", "nlminb",
    "Rvmmin", "Rcgmin")
nmax <- 5
for (ni in 1:nmax) {
    n <- 20 * ni
    x <- runif(n)  # generate a vector
    AA <- molerc(n)  # make sure defined
    aall <- optimx(x, fn = nobj, gr = ngrobj, method = mset, AA = -AA, control = list(starttests = FALSE,
        dowarn = FALSE))
    optansout(aall, NULL)
    cat("Above for n=", n, " \n")
}

## Loading required package: ucminf
## Loading required package: Rcgmin
## Loading required package: Rvmmin
## Attaching package: 'Rvmmin'
## The following object(s) are masked from 'package:optplus':
##
## bmchk
## Loading required package: minqa
## Loading required package: Rcpp
## Loading required package: methods
## Loading required package: dfoptim
## Warning: the condition has length > 1 and only the first element will be
## used
##    method                                          first.5.par fvalues
## 2    BFGS          -3.46318, 0.02941, 3.54568, 7.02968, 10.47589  -140.9
## 6     nlm           -1.95802, 0.01407, 1.98609, 3.94373, 5.87298  -140.9
## 7 nlminb      -0.160846, 0.001158, 0.163149, 0.323970, 0.482456  -140.9
## 1 LBFGSB -0.0864105, 0.0006221, 0.0876527, 0.1740507, 0.2591968  -140.9
## 3     CG      -0.753432, 0.005426, 0.764241, 1.517555, 2.259945  -140.9
## 9 Rcgmin           -1.46692, 0.01056, 1.48797, 2.95467, 4.40010  -140.9
## 4    spg -0.1007390, 0.0007252, 0.1021843, 0.2029076, 0.3021701  -140.9
## 5 ucminf -0.0678448, 0.0004884, 0.0688182, 0.1366524, 0.2035029  -140.9
## 8 Rvmmin          -3.49261, 0.02514, 3.54272, 7.03480, 10.47622  -140.9
##    fns grs hes rs conv KKT1  KKT2     mtilt xtimes  meths
## 2  20  17   0  0    3 TRUE FALSE        NA  0.004   BFGS
## 6 594   0   0  0    3 TRUE FALSE        NA  0.032    nlm
## 7  59  12   0  0    0 TRUE FALSE  0.001362  0.004 nlminb
## 1  47   7   0  0    0 TRUE FALSE  0.001278      0 LBFGSB
## 3  73  68   0  0    3 TRUE FALSE        NA  0.012     CG
## 9  53   8   0  0    0 TRUE FALSE 6.746e-06  0.004 Rcgmin
## 4 209   8   0  0    0 TRUE FALSE 0.0001224  0.016    spg
## 5  70  30   0  0    0 TRUE FALSE 6.566e-05  0.008 ucminf
## 8 102  35   0  0    0 TRUE FALSE  1.42e-06  0.016 Rvmmin
## Above for n= 20
## Warning: the condition has length > 1 and only the first element will be
## used
##    method                                          first.5.par fvalues
## 7 nlminb -5.035e-02, 7.599e-05, 5.053e-02, 1.009e-01, 1.511e-01  -602.9
## 6     nlm -0.1796086, 0.0003386, 0.1802280, 0.3598443, 0.5388450  -602.9
## 2    BFGS -0.1156346, 0.0001954, 0.1160142, 0.2316303, 0.3468605  -602.9
## 1 LBFGSB -3.632e-02, 6.027e-05, 3.644e-02, 7.275e-02, 1.089e-01  -602.9
## 4    spg -4.427e-02, 7.358e-05, 4.442e-02, 8.868e-02, 1.328e-01  -602.9
## 9 Rcgmin      -0.186858, 0.000311, 0.187479, 0.374336, 0.560569  -602.9
## 8 Rvmmin -0.1156204, 0.0001924, 0.1160050, 0.2316244, 0.3468585  -602.9
## 5 ucminf -2.838e-02, 4.723e-05, 2.847e-02, 5.685e-02, 8.513e-02  -602.9
```

```
## 3     CG -4.530e-02, 7.539e-05, 4.545e-02, 9.075e-02, 1.359e-01  -602.9
##   fns grs hes rs conv KKT1  KKT2     mtilt xtimes  meths
## 7  24  12   0  0    3 TRUE FALSE        NA  0.004 nlminb
## 6 499   0   0  0    3 TRUE FALSE        NA  0.032    nlm
## 2  41  14   0  0    3 TRUE FALSE        NA  0.008   BFGS
## 1  87   7   0  0    0 TRUE FALSE  0.001757  0.004 LBFGSB
## 4 408   7   0  0    0 TRUE FALSE 0.0005457  0.028    spg
## 9  94   8   0  0    0 TRUE FALSE 1.367e-05  0.004 Rcgmin
## 8 141  22   0  0    0 TRUE FALSE 3.807e-05  0.016 Rvmmin
## 5 130  50   0  0    0 TRUE FALSE   0.00013  0.016 ucminf
## 3 112  16   0  0    0 TRUE FALSE 9.187e-05  0.004     CG
## Above for n= 40
## Warning: the condition has length > 1 and only the first element will be
## used
##   method                                          first.5.par fvalues
## 6    nlm -0.1809226, 0.0002581, 0.1813833, 0.3622683, 0.5429639   -1389
## 7 nlminb -3.042e-02, 2.195e-05, 3.046e-02, 6.088e-02, 9.125e-02   -1389
## 1 LBFGSB -2.470e-02, 1.788e-05, 2.474e-02, 4.944e-02, 7.411e-02   -1389
## 2   BFGS -0.2504435, 0.0001846, 0.2508152, 0.5012601, 0.7513463   -1389
## 3     CG -5.446e-02, 3.926e-05, 5.454e-02, 1.090e-01, 1.634e-01   -1389
## 9 Rcgmin -3.374e-02, 2.431e-05, 3.379e-02, 6.754e-02, 1.012e-01   -1389
## 4    spg -0.0296869, 0.0000214, 0.0297297, 0.0594166, 0.0890606   -1389
## 5 ucminf -2.150e-02, 1.551e-05, 2.154e-02, 4.304e-02, 6.451e-02   -1389
## 8 Rvmmin -0.2504480, 0.0001806, 0.2508090, 0.5012566, 0.7513428   -1389
##   fns grs hes rs conv KKT1  KKT2     mtilt xtimes  meths
## 6 798   0   0  0    3 TRUE FALSE        NA   0.06    nlm
## 7  28  12   0  0    3 TRUE FALSE        NA  0.008 nlminb
## 1  11   7   0  0    3 TRUE FALSE        NA      0 LBFGSB
## 2  35  14   0  0    3 TRUE FALSE        NA  0.004   BFGS
## 3 162  21   0  0    0 TRUE FALSE 7.532e-05  0.012     CG
## 9 133   7   0  0    0 TRUE FALSE 8.732e-05  0.004 Rcgmin
## 4 609   8   0  0    0 TRUE FALSE 6.281e-05  0.048    spg
## 5 159  39   0  0    0 TRUE FALSE  0.000105  0.016 ucminf
## 8 162  18   0  0    0 TRUE FALSE 6.995e-06  0.016 Rvmmin
## Above for n= 60
## Warning: the condition has length > 1 and only the first element will be
## used
##   method                                          first.5.par fvalues
## 7 nlminb -2.094e-02, 2.151e-05, 2.096e-02, 4.190e-02, 6.283e-02   -2500
## 6    nlm -7.377e-02, 6.245e-05, 7.386e-02, 1.476e-01, 2.214e-01   -2500
## 2   BFGS -2.439e-01, 8.387e-05, 2.441e-01, 4.881e-01, 7.318e-01   -2500
## 1 LBFGSB -1.719e-02, 9.343e-06, 1.721e-02, 3.441e-02, 5.159e-02   -2500
## 3     CG -6.415e-02, 2.566e-05, 6.420e-02, 1.284e-01, 1.925e-01   -2500
## 4    spg -2.108e-02, 8.441e-06, 2.110e-02, 4.217e-02, 6.324e-02   -2500
## 9 Rcgmin -0.0289700, 0.0000116, 0.0289932, 0.0579631, 0.0869099   -2500
## 8 Rvmmin -2.439e-01, 9.747e-05, 2.441e-01, 4.881e-01, 7.318e-01   -2500
## 5 ucminf -1.409e-02, 5.640e-06, 1.410e-02, 2.819e-02, 4.227e-02   -2500
##   fns grs hes rs conv KKT1  KKT2     mtilt xtimes  meths
## 7  18   8   0  0    3 TRUE FALSE        NA  0.008 nlminb
## 6 899   0   0  0    3 TRUE FALSE        NA  0.072    nlm
## 2  23  13   0  0    3 TRUE FALSE        NA  0.008   BFGS
## 1  10   7   0  0    3 TRUE FALSE        NA  0.004 LBFGSB
## 3 263  52   0  0    0 TRUE FALSE 0.0001163  0.036     CG
## 4 809   8   0  0    0 TRUE FALSE 6.586e-05  0.068    spg
## 9 175   8   0  0    0 TRUE FALSE 6.972e-05  0.008 Rcgmin
## 8 204  16   0  0    0 TRUE FALSE  3.06e-05   0.02 Rvmmin
## 5 218  58   0  0    0 TRUE FALSE 0.0001212   0.04 ucminf
## Above for n= 80
## Warning: the condition has length > 1 and only the first element will be
## used
##   method                                          first.5.par fvalues
## 6    nlm -7.566e-02, 2.605e-05, 7.577e-02, 1.514e-01, 2.270e-01   -3934
## 2   BFGS -1.842e-01, 4.922e-05, 1.844e-01, 3.686e-01, 5.528e-01   -3934
## 1 LBFGSB -1.500e-02, 3.820e-06, 1.501e-02, 3.001e-02, 4.501e-02   -3934
## 7 nlminb -1.637e-02, 4.133e-06, 1.638e-02, 3.275e-02, 4.911e-02   -3934
## 3     CG -5.770e-02, 1.467e-05, 5.773e-02, 1.154e-01, 1.731e-01   -3934
## 4    spg -1.755e-02, 4.462e-06, 1.756e-02, 3.510e-02, 5.264e-02   -3934
## 9 Rcgmin -2.821e-02, 7.176e-06, 2.822e-02, 5.643e-02, 8.462e-02   -3934
## 8 Rvmmin -1.843e-01, 4.686e-05, 1.844e-01, 3.686e-01, 5.528e-01   -3934
## 5 ucminf -1.167e-02, 2.965e-06, 1.167e-02, 2.334e-02, 3.500e-02   -3934
##    fns grs hes rs conv KKT1  KKT2     mtilt xtimes  meths
## 6 1218   0   0  0    3 TRUE FALSE        NA  0.112    nlm
## 2   36  14   0  0    3 TRUE FALSE        NA  0.016   BFGS
## 1  208   8   0  0    0 TRUE FALSE  0.002735  0.004 LBFGSB
## 7  230  19   0  0    0 TRUE FALSE  0.001515   0.02 nlminb
## 3  307  54   0  0    0 TRUE FALSE 6.559e-05  0.052     CG
```

```
## 4 1009   8   0  0    0 TRUE FALSE 6.577e-05  0.096    spg
## 9  215   7   0  0    0 TRUE FALSE 5.481e-05  0.008 Rcgmin
## 8  255  21   0  0    0 TRUE FALSE 5.744e-06   0.04 Rvmmin
## 5  240  40   0  0    0 TRUE FALSE 0.0001284  0.036 ucminf
## Above for n= 100
```

The timings for these matrices of order 20 to 100 are likely too short to be very reliable in detail, but do show that the RQ problem using the scaling transformation and with an analytic gradient can be solved very quickly, especially by the limited memory methods such as L-BFGS-B and Rcgmin. Below we use the latter (in its unconstrained implementation) to show the times over different matrix sizes.

```
ctable <- matrix(NA, nrow = 10, ncol = 2)
nmax <- 10
for (ni in 1:nmax) {
    n <- 50 * ni
    x <- runif(n)  # generate a vector
    AA <- molerc(n)  # make sure defined
    tcgu <- system.time(arcgu <- Rcgminu(x, fn = nobj, gr = ngrobj, AA = -AA))[[1]]
    ctable[[ni, 1]] <- n
    ctable[[ni, 2]] <- tcgu
}
cgtime <- data.frame(n = ctable[, 1], tRcgminu = ctable[, 2])
print(cgtime)

##      n tRcgminu
## 1   50    0.004
## 2  100    0.008
## 3  150    0.008
## 4  200    0.016
## 5  250    0.020
## 6  300    0.036
## 7  350    0.064
## 8  400    0.108
## 9  450    0.096
## 10 500    0.188
```

9 A specialized minimizer - Geradin's method

For comparison, let us try the Geradin routine (Appendix 1) as implemented in R by one of us (JN).

```
cat("Test geradin with explicit matrix multiplication\n")

## Test geradin with explicit matrix multiplication

n <- 10
AA <- molermat(n)
BB = diag(rep(1, n))
x <- runif(n)
tg <- system.time(ag <- geradin(x, ax, bx, AA = AA, BB = BB, control = list(trace = FALSE)))[[1]]
cat("Minimal eigensolution\n")

## Minimal eigensolution

print(ag)

## $x
##  [1] -88447285 -44223832 -22112390 -11057192  -5530613  -2769352  -1392770
##  [8]   -712579   -388679   -259119
##
## $RQ
## [1] 8.583e-06
##
## $ipr
## [1] 48
##
## $msg
## [1] "Small gradient -- done"
##
```

```
cat("Geradin time=", tg, "\n")

## Geradin time= 0

tgn <- system.time(agn <- geradin(x, ax, bx, AA = -AA, BB = BB, control = list(trace = FALSE)))[[1]]
cat("Maximal eigensolution (negative matrix)\n")

## Maximal eigensolution (negative matrix)

print(agn)

## $x
##  [1] -6.805e+10  2.309e+09  7.259e+10  1.404e+11  2.035e+11  2.597e+11
##  [7]  3.071e+11  3.442e+11  3.696e+11  3.826e+11
##
## $RQ
## [1] -31.59
##
## $ipr
## [1] 36
##
## $msg
## [1] "Small gradient -- done"
##

cat("Geradin time=", tgn, "\n")

## Geradin time= 0.004
```

Let us time this routine with different matrix vector approaches.

```
naximp <- function(x, A = 1) {
    # implicit moler A*x
    n <- length(x)
    y <- rep(0, n)
    for (i in 1:n) {
        tt <- 0
        for (j in 1:n) {
            if (i == j)
                tt <- tt + i * x[i] else tt <- tt + (min(i, j) - 2) * x[j]
        }
        y[i] <- -tt  # include negative sign
    }
    y
}

dyn.load("moler.so")
cat("Is the mat multiply loaded? ", is.loaded("moler"), "\n")

## Is the mat multiply loaded?  TRUE


naxftn <- function(x, A) {
    # ignore second argument
    n <- length(x)  # could speed up by having this passed
    vout <- rep(0, n)  # purely for storage
    res <- (-1) * (.Fortran("moler", n = as.integer(n), x = as.double(x), vout = as.double(vout)))$vout
}

require(compiler)
naxftnc <- cmpfun(naxftn)
naximpc <- cmpfun(naximp)

require(microbenchmark)
nmax <- 10
gtable <- matrix(NA, nrow = nmax, ncol = 6)  # to hold results
# loop over sizes
for (ni in 1:nmax) {
    n <- 50 * ni
    x <- runif(n)  # generate a vector
    gtable[[ni, 1]] <- n
    AA <- molermat(n)
    BB <- diag(rep(1, n))
```

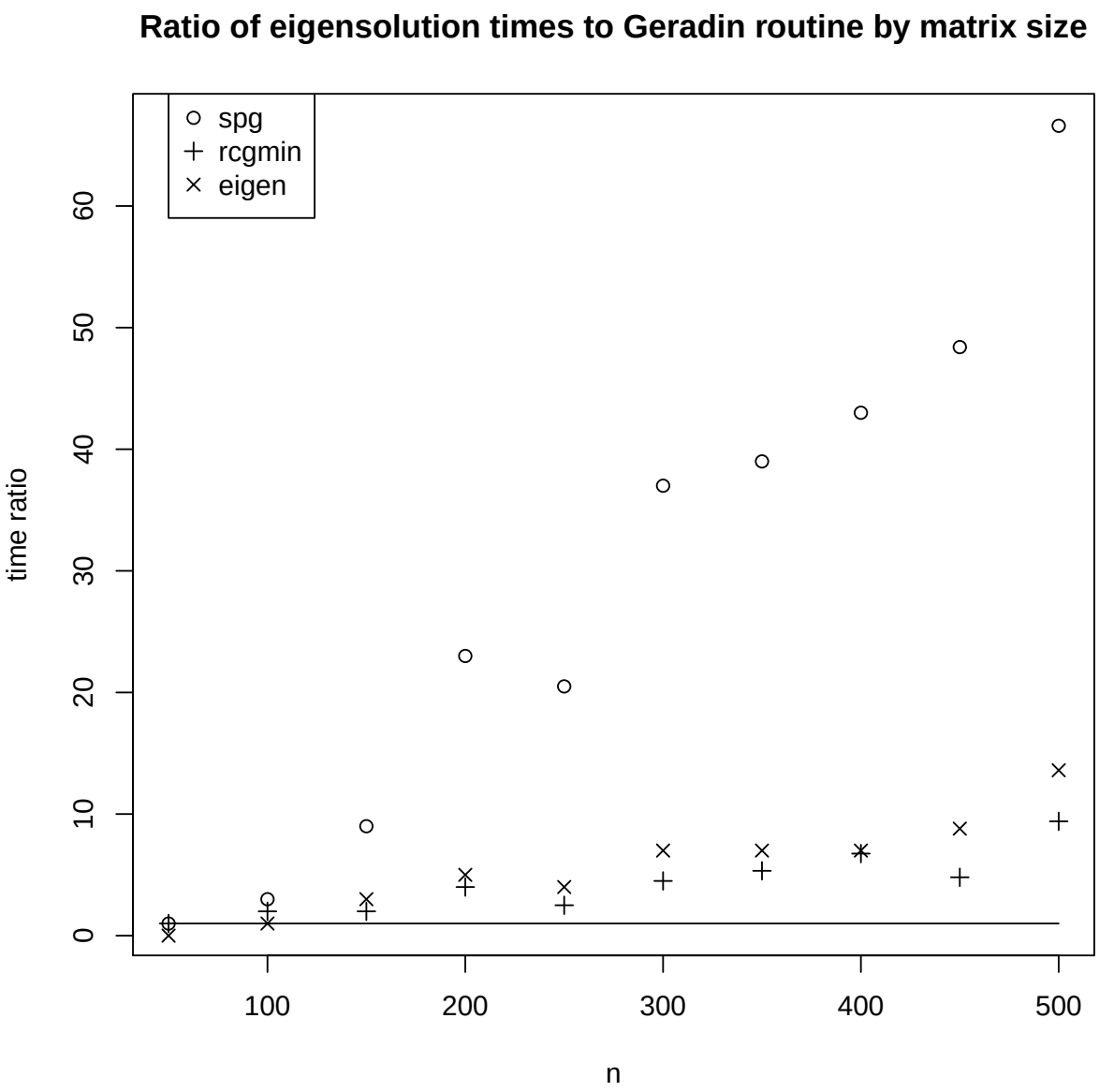
```
    tgax <- system.time(ogax <- geradin(x, ax, bx, AA = -AA, BB = BB, control = list(trace = FALSE)))[[1]]
    gtable[[ni, 2]] <- tgax
    tgaximp <- system.time(ogaximp <- geradin(x, naximp, ident, AA = 1, BB = 1,
        control = list(trace = FALSE)))[[1]]
    gtable[[ni, 3]] <- tgaximp
    tgaximpc <- system.time(ogaximpc <- geradin(x, naximpc, ident, AA = 1, BB = 1,
        control = list(trace = FALSE)))[[1]]
    gtable[[ni, 4]] <- tgaximpc
    tgaxftn <- system.time(ogaxftn <- geradin(x, naxftn, ident, AA = 1, BB = 1,
        control = list(trace = FALSE)))[[1]]
    gtable[[ni, 5]] <- tgaxftn
    tgaxftnc <- system.time(ogaxftnc <- geradin(x, naxftnc, ident, AA = 1, BB = 1,
        control = list(trace = FALSE)))[[1]]
    gtable[[ni, 6]] <- tgaxftnc
    # cat(n,tgax, tgaximp, tgaximpc, tgaxftn, tgaxftnc,'\n')
}

gtym <- data.frame(n = gtable[, 1], ax = gtable[, 2], aximp = gtable[,
    3], aximpc = gtable[, 4], axftn = gtable[, 5], axftnc = gtable[, 6])
print(gtym)

##       n    ax  aximp aximpc axftn axftnc
## 1   50 0.004  0.316  0.092 0.004  0.004
## 2  100 0.008  1.368  0.392 0.004  0.004
## 3  150 0.016  2.924  0.824 0.004  0.004
## 4  200 0.028  5.504  1.549 0.008  0.004
## 5  250 0.040  8.980  2.568 0.008  0.008
## 6  300 0.056 11.857  3.372 0.008  0.008
## 7  350 0.068 15.541  4.441 0.012  0.012
## 8  400 0.108 23.525  6.737 0.016  0.012
## 9  450 0.136 30.057  8.696 0.020  0.020
## 10 500 0.152 34.582  9.769 0.016  0.020
```

Let us check that the solution for `n = 100` by Geradin is consistent with the answer via `eigen()`.

```
n <- 100
x <- runif(n)
emax <- load("temax.Rdata")

## Warning: cannot open compressed file 'temax.Rdata', probable reason 'No
## such file or directory'
## Error: cannot open the connection

evalmax <- emax$evalmax
evecmac <- emax$evecmax
ogaxftn <- geradin(x, naxftn, ident, AA = 1, BB = 1, control = list(trace = FALSE))

## Error: Fortran symbol name "moler" not in load table

gvec <- ogaxftn$x
gval <- -ogaxftn$RQ
gvec <- sign(gvec[[1]]) * gvec/sqrt(as.numeric(crossprod(gvec)))
diff <- gvec - evecmax
cat("Geradin diff eigenval from eigen result: ", gval - evalmax,
    "   max(abs(vector diff))=", max(abs(diff)), "\n")

## Geradin diff eigenval from eigen result:  96782    max(abs(vector diff))= 0.1236
```

10 Perspective

We can compare the different approaches by looking at the ratio of the best solution time for each method (compiled or interpreted, with best choice of function) to the time for the Geradin approach for the different matrix sizes. In this we will ignore the fact that some approaches do not build the matrix.



To check the value of the Geradin approach, let us use a much larger problem, with `n=2000`.

```
## Times in seconds
## Build = 5.672  eigen(): 11.64   Rcgminu: 2.004  Geradin: 0.264
## Ratios: build= 21.48 eigen= 44.11   Rcgminu= 7.591
```

11 Conclusions

The Rayleigh Quotient minimization approach to eigensolutions has an intuitive appeal and seemingly offers an interesting optimization test problem, especially if we can make it computationally efficient. To improve time efficiency, we can apply the R byte code compiler, use a Fortran (or other compiled language) subroutine,

and choose how we set up our objective functions and gradients. To improve memory use, we can consider using a matrix implicitly.

From the tests in this vignette, here is what we may say about these attempts, which we caution are based on a relatively small sample of tests:

- The R byte code compiler offers a useful gain in speed when our code has statements that access array elements rather than uses them in vectorized form.
- The `crossprod()` function is very efficient.
- Fortran is not very difficult to use for small subroutines that compute a function such as the implicit matrix-vector product, and it allows efficient computations for such operations.
- The `eigen()` routine is a highly effective tool for computing all eigensolutions, even of a large matrix. It is only worth computing a single solution when the matrix is very large, in which case a specialized method such as that of Geradin makes sense and offers significant savings, especially when combined with the Fortran implicit matrix-product routine.

Acknowledgements

This vignette originated due to a problem suggested by Gabor Grothendieck. Ravi Varadhan has provided inciteful comments and some vectorized functions which greatly altered some of the observations.

References

Nash, J. C. (1979). *Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation*. Bristol: Adam Hilger. Second Edition, 1990, Bristol: Institute of Physics Publications.

Appendix 1: Geradin routine

```
ax <- function(x, AA) {
    u <- as.numeric(AA %*% x)
}
bx <- function(x, BB) {
    v <- as.numeric(BB %*% x)
}
geradin <- function(x, ax, bx, AA, BB, control = list(trace = TRUE,
    maxit = 1000)) {
    # Geradin minimize Rayleigh Quotient, Nash CMN Alg 25 print(control)
    trace <- control$trace
    n <- length(x)
    tol <- n * n * .Machine$double.eps^2
    offset <- 1e+05  # equality check offset
    if (trace)
        cat("geradin.R, using tol=", tol, "\n")
    ipr <- 0  # counter for matrix mults
    pa <- .Machine$double.xmax
    R <- pa
    msg <- "no msg"
    # step 1 - main loop
    keepgoing <- TRUE
    while (keepgoing) {
        avec <- ax(x, AA)
        bvec <- bx(x, BB)
        ipr <- ipr + 1
        xax <- as.numeric(crossprod(x, avec))
        xbx <- as.numeric(crossprod(x, bvec))
        if (xbx <= tol) {
            keepgoing <- FALSE  # not really needed
            msg <- "avoid division by 0 as xbx too small"
```

```
        break
    }
    p0 <- xax/xbx
    if (p0 > pa) {
        keepgoing <- FALSE  # not really needed
        msg <- "Rayleigh Quotient increased in step"
        break
    }
    pa <- p0
    g <- 2 * (avec - p0 * bvec)/xbx
    gg <- as.numeric(crossprod(g))  # step 6
    if (trace)
        cat("Before loop: RQ=", p0, " after ", ipr, " products, gg=", gg,
            "\n")
    if (gg < tol) {
        # step 7
        keepgoing <- FALSE  # not really needed
        msg <- "Small gradient - done"
        break
    }
    t <- -g  # step 8
    for (itn in 1:n) {
        # major loop step 9
        y <- ax(t, AA)
        z <- bx(t, BB)
        ipr <- ipr + 1  # step 10
        tat <- as.numeric(crossprod(t, y))  # step 11
        xat <- as.numeric(crossprod(x, y))
        xbt <- as.numeric(crossprod(x, z))
        tbt <- as.numeric(crossprod(t, z))
        u <- tat * xbt - xat * tbt
        v <- tat * xbx - xax * tbt
        w <- xat * xbx - xax * xbt
        d <- v * v - 4 * u * w
        if (d < 0)
            stop("Geradin: imaginary roots not possible")  # step 13
        d <- sqrt(d)  # step 14
        if (v > 0)
            k <- -2 * w/(v + d) else k <- 0.5 * (d - v)/u
        xlast <- x  # NOT as in CNM - can be avoided with loop
        avec <- avec + k * y
        bvec <- bvec + k * z  # step 15, update
        x <- x + k * t
        xax <- xax + as.numeric(crossprod(x, avec))
        xbx <- xbx + as.numeric(crossprod(x, bvec))
        if (xbx < tol)
            stop("Geradin: xbx has become too small")
        chcount <- n - length(which((xlast + offset) == (x + offset)))
        if (trace)
            cat("Number of changed components = ", chcount, "\n")
        pn <- xax/xbx  # step 17 different order
        if (chcount == 0) {
            keepgoing <- FALSE  # not really needed
            msg <- "Unchanged parameters - done"
            break
        }
        if (pn >= p0) {
            if (trace)
              cat("RQ not reduced, restart\n")
            break  # out of itn loop, not while loop (TEST!)
        }
        p0 <- pn  # step 19
        g <- 2 * (avec - pn * bvec)/xbx
        gg <- as.numeric(crossprod(g))
        if (trace)
            cat("Itn", itn, " RQ=", p0, " after ", ipr, " products, gg=",
              gg, "\n")
        if (gg < tol) {
            # step 20
            if (trace)
              cat("Small gradient in iteration, restart\n")
            break  # out of itn loop, not while loop (TEST!)
        }
        xbt <- as.numeric(crossprod(x, z))  # step 21
        w <- y - pn * z  # step 22
        tabt <- as.numeric(crossprod(t, w))
```

```
            beta <- as.numeric(crossprod(g, (w - xbt * g)))
            beta <- beta/tabt  # step 23
            t <- beta * t - g
        }  # end loop on itn - step 24
    }  # end main loop - step 25
    ans <- list(x = x, RQ = p0, ipr = ipr, msg = msg)  # step 26
}
```