Timing Rayleigh Quotient minimization in R

true

2023-9-6

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. It has been updated from a 2012 document to reflect changes in R and its packages that make it awkward to reprocess the original document on newer computers.

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.

Note 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

$$x_{normalized} = x/sqrt(x'x)$$

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.

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
}
```

Time to build the matrix

Let us see how long it takes to build the Moler matrix of different sizes. In 2012 we used the byte-code compiler, but that now seems to be active by default and NOT to give worthwhile improvements. We also include times for the `eigen()` function that computes the full set of eigensolutions very quickly.

```
## Loading required package: microbenchmark
```

```
##      n buildi   osize eigentime bfast
## 1   50   2954   20216      1264  2685
## 2  100   6617   80216      1846   630
## 3  150  16493  180216      3912  1061
## 4  200  27025  320216      6296  1454
## 5  250  33541  500216      7450  1764
## 6  300  42993  720216     10429  2584
## 7  350  53329  980216     14139  2819
## 8  400  65419 1280216     17840  3860
## 9  450  80035 1620216     24538  4722
## 10 500  93734 2000216     29698  4521
```

```
## buildi - interpreted build time
## osize - matrix size in bytes; eigentime - all eigensolutions time
```

```
## bfast - interpreted vectorized build time
```

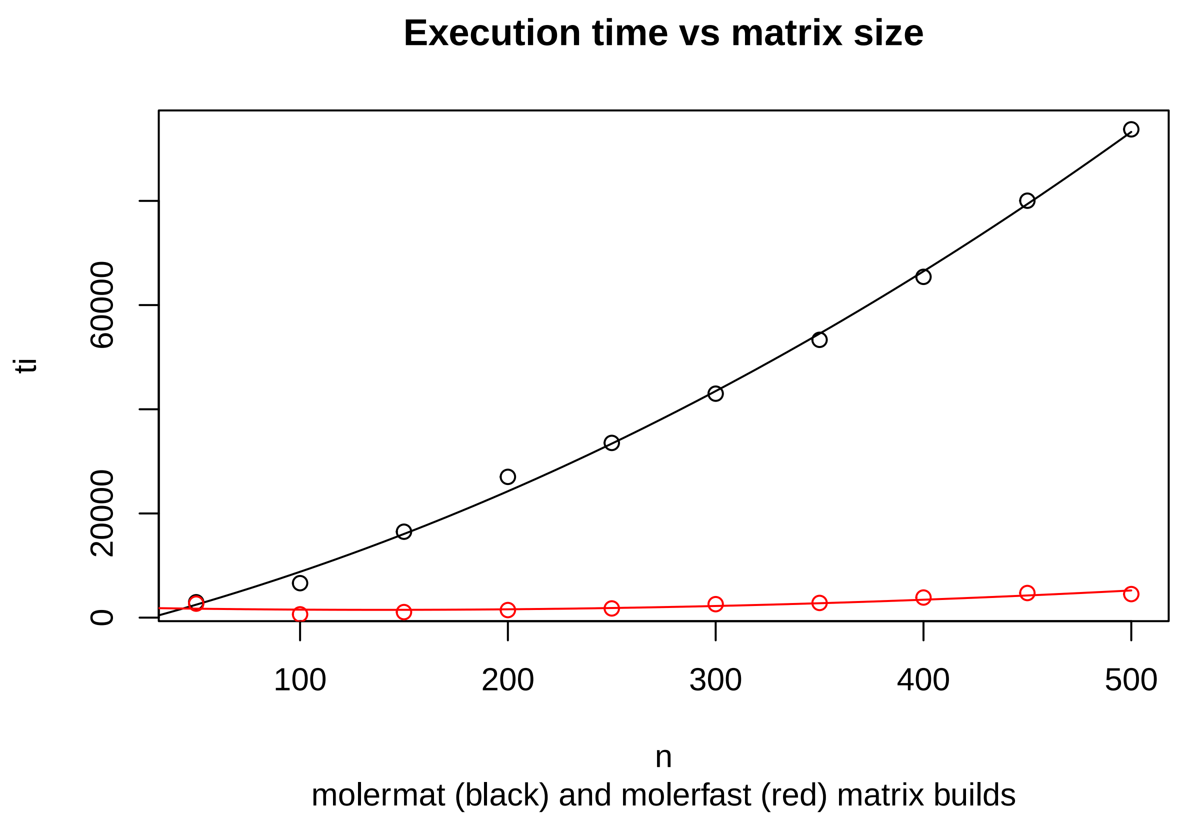
```
## Times converted to milliseconds
```

It does not appear that the compiler has much effect, or else it is being automatically invoked.

We can graph the times. The code, which is not echoed here, also models the times and the object size created as almost perfect quadratic models in `n`.

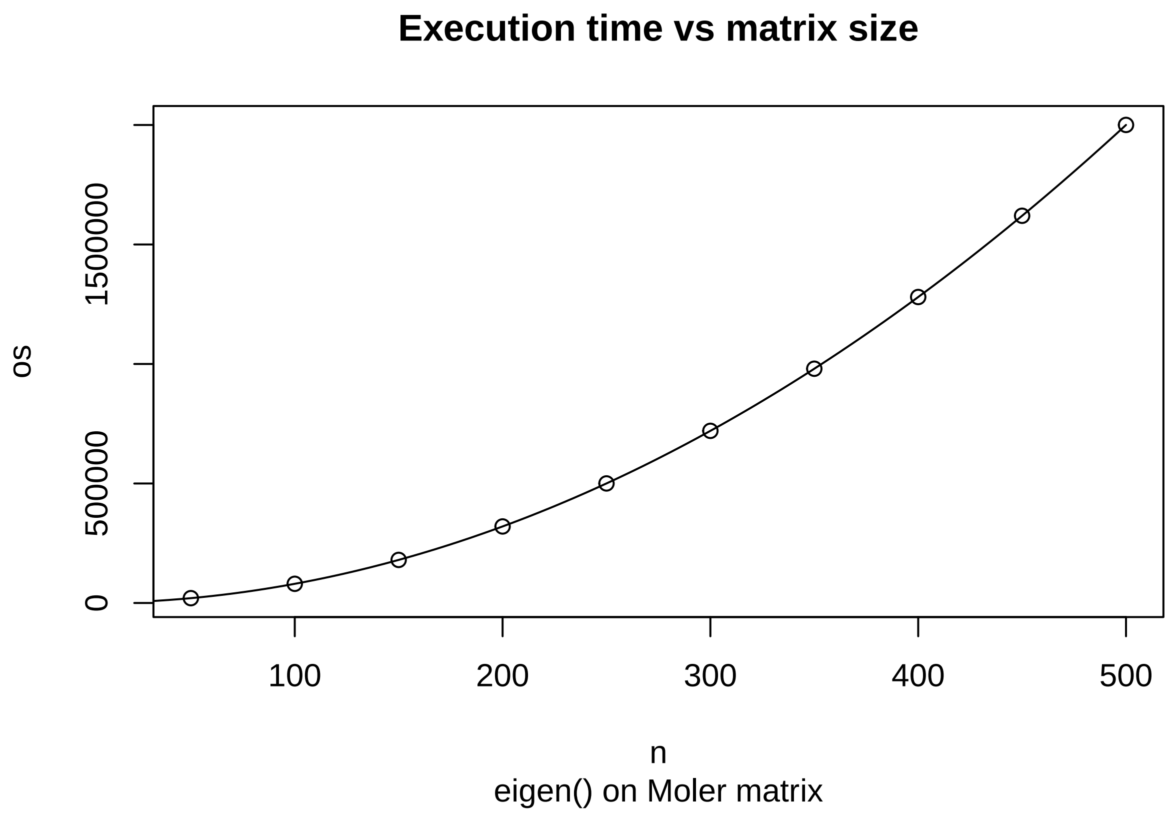
```
##
## Call:
## lm(formula = ti ~ n + n2)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2184.5  -922.0   284.1   483.8  2763.1
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -2.891e+03  1.793e+03  -1.612 0.150894
## n            9.809e+01  1.498e+01   6.550 0.000319 ***
## n2           1.884e-01  2.654e-02   7.098 0.000194 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1524 on 7 degrees of freedom
## Multiple R-squared:  0.9981, Adjusted R-squared:  0.9975
## F-statistic:  1830 on 2 and 7 DF,  p-value: 3.036e-10
```

```
##
## Call:
## lm(formula = tf ~ n + n2)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -905.48 -363.18  -13.97  403.57  967.48
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) 2042.06117  766.62947   2.664   0.0323 *
## n             -7.91120    6.40354  -1.235   0.2565
## n2             0.02850    0.01135   2.512   0.0403 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 651.8 on 7 degrees of freedom
## Multiple R-squared:  0.8356, Adjusted R-squared:  0.7887
## F-statistic: 17.79 on 2 and 7 DF,  p-value: 0.001801
```



```
## Warning in summary.lm(osize): essentially perfect fit: summary may be
## unreliable

##
## Call:
## lm(formula = os ~ n + n2)
##
## Residuals:
##        Min         1Q     Median         3Q        Max
## -2.654e-12 -1.314e-13  3.293e-13  7.262e-13  1.211e-12
##
## Coefficients:
##              Estimate Std. Error   t value Pr(>|t|)
## (Intercept) 2.160e+02  1.617e-12 1.336e+14  < 2e-16 ***
## n           5.127e-13  1.351e-14 3.795e+01 2.29e-09 ***
## n2          8.000e+00  2.394e-17 3.342e+17  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.375e-12 on 7 degrees of freedom
## Multiple R-squared:      1,  Adjusted R-squared:      1
## F-statistic: 1.112e+36 on 2 and 7 DF,  p-value: < 2.2e-16
```



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.

```
rqdir<-function(x, AA){
  rq<-0.0
  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]
     }
  }
  rq
}
```

Somewhat better (as we shall show below) is

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

and (believed) better still is

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

Note that we could implicitly include the minus sign in these routines to allow for finding the maximal eigenvalue by minimizing the Rayleigh Quotient of $-A$. However, such shortcuts often rebound when the implicit negation is overlooked.

If we already have the inner product $ A x$ as vector `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)))
}
```

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' A x$. If `n} is the order of the problem, then for largen`', 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

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=6) # 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<- mean(microbenchmark(oax<-ax(x, AA), times=mbt)$time)*0.001
  taxx<-mean(microbenchmark(oaxx<-axx(x, AA), times=mbt)$time)*0.001
  if (! identical(oax, oaxx)) stop("oaxx NOT correct")
  taxftn<-mean(microbenchmark(oaxftn<-axftn(x, AA=1), times=mbt)$time)*0.001
  if (! identical(oax, oaxftn)) stop("oaxftn NOT correct")
  taximp<-mean(microbenchmark(oaximp<-aximp(x, AA=1), times=mbt)$time)*0.001
  if (! identical(oax, oaximp)) stop("oaximp NOT correct")
  taxmfi<-mean(microbenchmark(oaxmfi<-axmolerfast(x, AA=1), times=mbt)$time)*0.001
  if (! identical(oax, oaxmfi)) stop("oaxmfi NOT correct")
  ptable[[ni, 2]]<-tax
  ptable[[ni, 3]]<-taxx
  ptable[[ni, 4]]<-taxftn
  ptable[[ni, 5]]<-taximp
  ptable[[ni, 6]]<-taxmfi
}

axtym<-data.frame(n=ptable[,1], ax=ptable[,2], axx=ptable[,3],  axftn=ptable[,4],
                  aximp=ptable[,5], axmfast=ptable[,6])
print(axtym)
```

```
##      n       ax      axx     axftn     aximp  axmfast
## 1   50 612.9105 626.3690 1181.9845  3676.759 2714.718
## 2  100  71.0790  15.8895   46.2205  4880.578  637.083
## 3  150  91.0795 224.0115   85.8620  8101.315 1019.492
## 4  200 100.7610  37.1485  140.7080 14150.840 1448.224
## 5  250 103.5340  79.7465  214.0985 21134.297 1835.311
## 6  300 146.5190  72.3760  297.5960 28964.678 2238.751
## 7  350 161.4070  97.2075  398.4990 39023.471 2777.773
## 8  400 164.3340 127.4080  514.0315 51386.129 3312.066
## 9  450 196.4125 160.0440  648.8920 64899.827 3877.553
## 10 500 235.4760 248.5315  797.2560 77996.382 4511.730
```

```
## ax = R matrix * vector  A %*% x
## axx = R crossprod A, x
## axftn = Fortran version of implicit Moler A * x
## aximp = implicit moler A*x in R
## axmfast = A fast and memory-saving version of A %*% x
## Times in milliseconds from microbenchmark
```

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. Apportioning "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.

```
## Times (in millisecs) adjusted for matrix build
```

```
##      n    axbld    axxbld     axftn     aximp
## 1   50  3566.423  3579.881 1181.9845  3676.759
## 2  100  6688.270  6633.081   46.2205  4880.578
## 3  150 16583.658 16716.590   85.8620  8101.315
## 4  200 27125.470 27061.858  140.7080 14150.840
## 5  250 33644.257 33620.470  214.0985 21134.297
## 6  300 43139.153 43065.010  297.5960 28964.678
## 7  350 53490.798 53426.598  398.4990 39023.471
## 8  400 65583.623 65546.697  514.0315 51386.129
## 9  450 80231.123 80194.754  648.8920 64899.827
## 10 500 93969.095 93982.151  797.2560 77996.382
```

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`.

RQ computation times

We have 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.

```
## Direct algorithm:  15448.61
```

```
## ray1: mat-mult algorithm:  167.3047
```

```
## ray2: crossprod algorithm:  167.8529
```

```
## ray3: ax Fortran + crossprod:  713.258
```

```
## ray3: ax fast R implicit + crossprod:  4827.743
```

Here we see that the use of either the matrix multiplication in `ray1` or of crossprodinray2is `very fast, and this is interpreted code. Once again, we note that all timings except those forray3should have some adjustment for the building of the matrix. If storage is an issue, thenray3, 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 withray3`‘ is also useful and quite fast if we need to minimize memory use.

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.

```r
rqt<-function(x, AA){
    rq<-as.numeric(crossprod(x, crossprod(AA,x)))
}
proj <- function(x) {sign(x[1]) * x/sqrt(c(crossprod(x))) } # from ravi
require(BB)
```

```
## Loading required package: BB
```

```r
n<-100
x<-rep(1,n)
AA<-molermat(n)
```

```
evs<-eigen(AA)
tmin<-microbenchmark(amin<-spg(x, fn=rqt, project=proj, control=list(trace=FALSE), AA=AA), times=mbt)$ti
```

```
## Warning in spg(x, fn = rqt, project = proj, control = list(trace = FALSE), :
## convergence tolerance satisified at intial parameter values.

## Warning in spg(x, fn = rqt, project = proj, control = list(trace = FALSE), :
## convergence tolerance satisified at intial parameter values.
```

```
#amin
tmax<-microbenchmark(amax<-spg(x, fn=rqt, project=proj, control=list(trace=FALSE), AA=-AA), times=mbt)$t
```

```
## Warning in spg(x, fn = rqt, project = proj, control = list(trace = FALSE), :
## convergence tolerance satisified at intial parameter values.

## Warning in spg(x, fn = rqt, project = proj, control = list(trace = FALSE), :
## convergence tolerance satisified at intial parameter values.
```

```
#amax
evalmax<-evs$values[1]
evecmax<-evs$vectors[,1]
evecmax<-sign(evecmax[1])*evecmax/sqrt(as.numeric(crossprod(evecmax)))
emax<-list(evalmax=evalmax, evecmax=evecmax)
# save(emax, file="temax.Rdata")
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= 318550 in time  9582008 1845840
```

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

```
## Eigenvalue - result from eigen= 318550   vector max(abs(diff))= 0.7660254
```

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

```
## maximal eigensolution: Value= 318550 in time  1728597 1576352
```

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

```
## Eigenvalue - result from eigen= 314615.7   vector max(abs(diff))= 0.242489
```

```
#print(amax$par)
```

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

```
##      n spgrqt tbld
## 1   50   3051  309
## 2  100  15602  625
## 3  150  36426 1916
## 4  200  70688 2659
## 5  250 122703 2379
## 6  300 217850 2349
## 7  350 295537 2808
## 8  400 427630 3365
## 9  450 595919 3948
## 10 500 800640 4805
```

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 `code{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::opm()`, we have reduced the matrix sizes and numbers.

```r
require(optimx)
```

```
## Loading required package: optimx
```

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

```
## Above for n= 20
## Above for n= 40
## Above for n= 60
## Above for n= 80
## 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 ncg. Below we use the latter 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<-molerfast(n) # define matrix
  tcgu<-microbenchmark(arcgu<-optimr(x, fn=nobj, gr=ngrobj, method="ncg",
         AA=-AA), times=mbt)
  ctable[[ni,1]]<-n
  ctable[[ni,2]]<-mean(tcgu$time)*0.001
}
cgtime<-data.frame(n=ctable[,1], tcgmin=ctable[,2])
print(round(cgtime,0))
```

```
##      n tcgmin
## 1   50    912
## 2  100   1260
## 3  150   2433
## 4  200   2885
## 5  250   3914
## 6  300   8182
## 7  350  10887
## 8  400  16978
## 9  450  16257
## 10 500  16092
```

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<-microbenchmark(ag<-geradin(x, ax, bx, AA=AA, BB=BB,
   control=list(trace=FALSE)), times=mbt)
cat("Minimal eigensolution\n")
```

```
## Minimal eigensolution
```

```
print(ag)
```

```
## $x
##  [1] -771053.330 -385528.319 -192768.296  -96392.834  -48213.996  -24142.258
##  [7]  -12141.703   -6212.024   -3388.371   -2258.911
##
## $RQ
## [1] 8.582807e-06
##
## $ipr
## [1] 52
##
## $msg
## [1] "Small gradient -- done"
```

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

```
## Geradin time= 29815774
```

```
tgn<-microbenchmark(agn<-geradin(x, ax, bx, AA=-AA, BB=BB,
   control=list(trace=FALSE)), times=mbt)
cat("Maximal eigensolution (negative matrix)\n")
```

```
## Maximal eigensolution (negative matrix)
```

```
print(agn)
```

```
## $x
##  [1]  8.316256e+11 -2.776961e+10 -8.862480e+11 -1.714914e+12 -2.485517e+12
##  [6] -3.171754e+12 -3.750958e+12 -4.203891e+12 -4.515523e+12 -4.674265e+12
##
## $RQ
## [1] -31.58981
##
## $ipr
## [1] 32
##
## $msg
## [1] "Small gradient -- done"
```

```
cat("Geradin time=",mean(tgn$time),"\n")
```

```
## Geradin time= 451788
```

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(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<-microbenchmark(ogax<-geradin(x, ax, bx, AA=-AA, BB=BB, control=list(trace=FALSE)), times=mbt)
  gtable[[ni, 2]]<-mean(tgax$time)
  tgaximp<-microbenchmark(ogaximp<-geradin(x, naximp, ident, AA=1, BB=1, control=list(trace=FALSE)), ti
  gtable[[ni, 3]]<-mean(tgaximp$time)
  tgaxftn<-microbenchmark(ogaxftn<-geradin(x, naxftn, ident, AA=1, BB=1, control=list(trace=FALSE)), ti
  gtable[[ni, 4]]<-mean(tgaxftn$time)
}

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

```
##      n         ax      aximp    axftn
## 1   50   759098.5   30061756  2397075
## 2  100  1901770.0  120833844  1539722
## 3  150  2581029.0  263666699  2819041
## 4  200  3243979.0  446778022  3847636
## 5  250  4172532.0  677768515  5649526
## 6  300  5561530.0  795418092  6286728
## 7  350  6255490.5 1080811782  8458502
## 8  400  8752420.5 1820586852 13717728
## 9  450 10919955.0 2356724496 17672469
## 10 500 12976650.5 2827552554 20983414
```

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

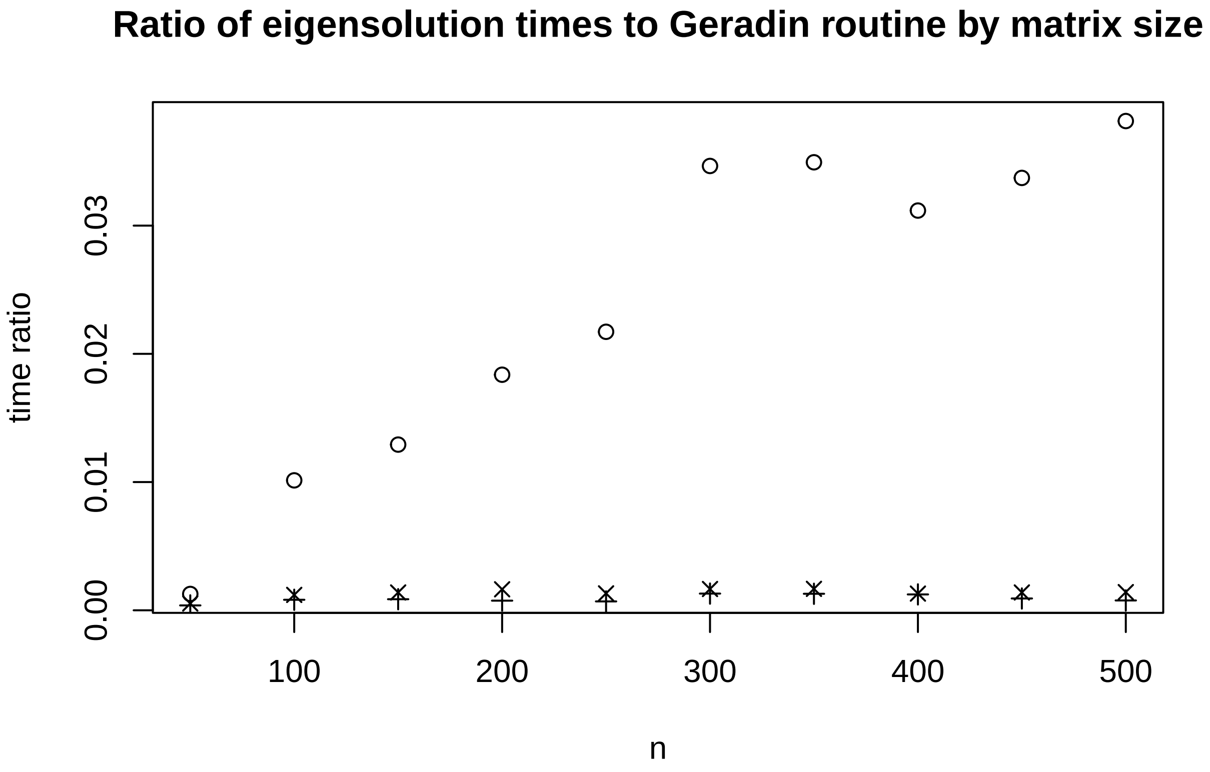
```
n<-100
x<-runif(n)
# emax<-load("temax.Rdata")
evalmax<-emax$evalmax
evecmac<-emax$evecmax
ogaxftn<-geradin(x, naxftn, ident, AA=1, BB=1, control=list(trace=FALSE))
gvec<-ogaxftn$x
gval<- -ogaxftn$RQ
gvec<-sign(gvec[[1]])*gvec/sqrt(as.numeric(crossprod(gvec)))
```

```
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:  -0.0003908111    max(abs(vector diff))= 8.142612e-05
```

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 = 109664668  eigen(): 936686570   Rcgminu: 343720895  Geradin: 332117268

## Ratios: build= 0.3301986 eigen= 2.820349   Rcgminu= 1.034938
```

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.

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+5 # 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
}
```

References

Nash, John C. 1979. *Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation.* Bristol: Adam Hilger.