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 and which show that timing profiles of R commands have changed in the interim.

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.

```
##     n   osize buildi buildir eigentime eigentimr bfast bfastr
## 1 100   80216   3820     962      2144       343   869    966
## 2 200  320216  14330    3504      6125      1059  1545    304
## 3 300  720216  31083    4655     11626      1086  2571    649
## 4 400 1280216  53936    4735     20066      1421  4468   5201
## 5 500 2000216  84616    6753     32048      6664  4883    885
```

```
## osize - matrix size in bytes
```

```
## eigentime - all eigensolutions time
```

```
## buildi - interpreted build time, range
```

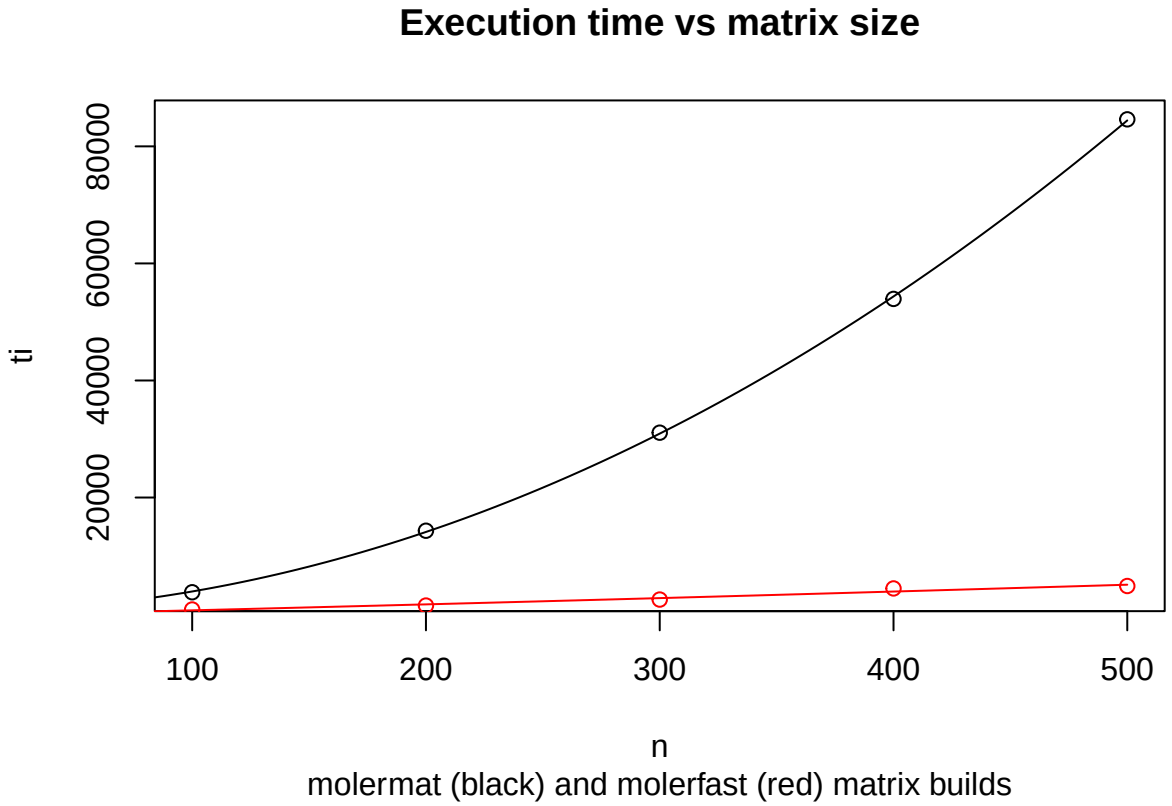
```
## bfast - interpreted vectorized build time
```

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

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`.

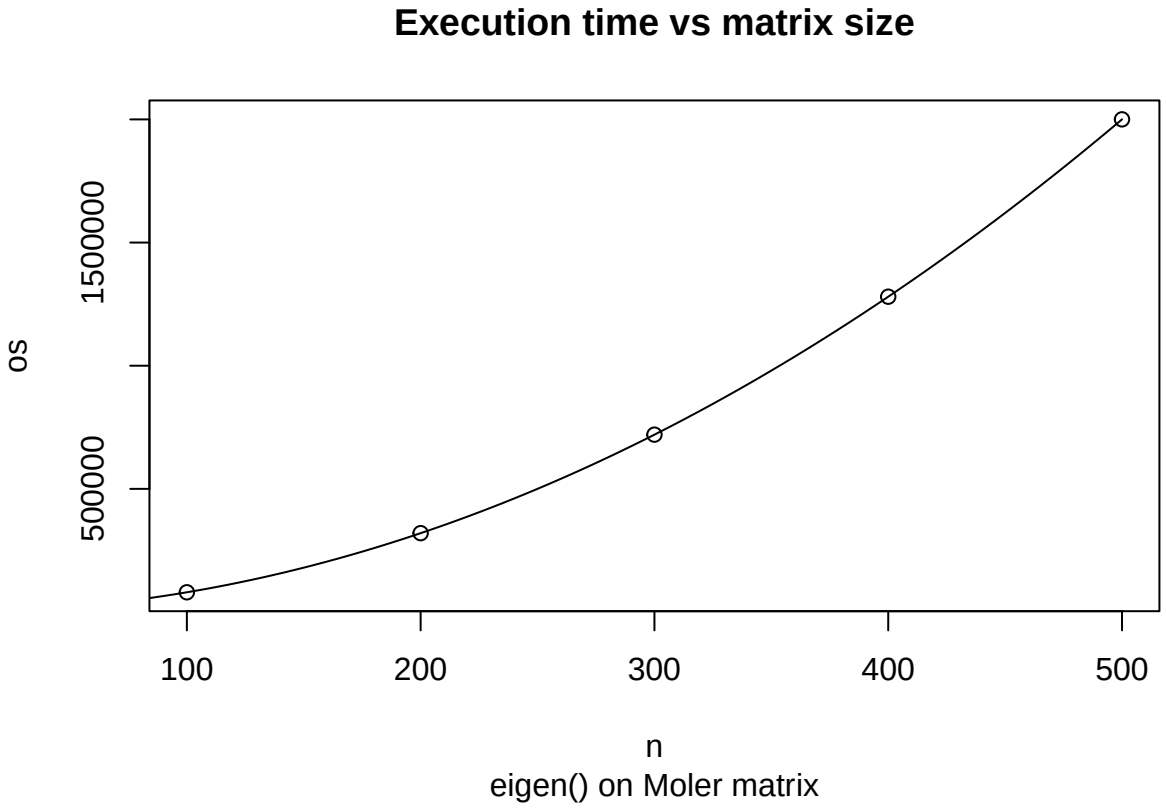
```
##
## Call:
## lm(formula = ti ~ n + n2)
##
## Residuals:
##      1      2      3      4      5
## -131.7  209.9  160.3 -423.7  185.1
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 417.60000  831.85397   0.502 0.665476
## n             2.16943    6.33928   0.342 0.764802
## n2            0.33171    0.01037  32.001 0.000975 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 387.9 on 2 degrees of freedom
## Multiple R-squared:  0.9999, Adjusted R-squared:  0.9999
## F-statistic: 1.397e+04 on 2 and 2 DF,  p-value: 7.159e-05
```

```
##
## Call:
## lm(formula = tf ~ n + n2)
##
## Residuals:
##      1      2      3      4      5
##  142.1 -202.2 -246.3  530.6 -224.3
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -2.436e+02  1.021e+03  -0.238    0.834
## n            9.455e+00  7.784e+00   1.215    0.348
## n2           2.493e-03  1.273e-02   0.196    0.863
##
## Residual standard error: 476.2 on 2 degrees of freedom
## Multiple R-squared:  0.9636, Adjusted R-squared:  0.9272
## F-statistic: 26.46 on 2 and 2 DF,  p-value: 0.03642
```




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

##
## Call:
## lm(formula = os ~ n + n2)
##
## Residuals:
##          1          2          3          4          5
##  4.765e-13  5.479e-12 -1.930e-11  2.025e-11 -6.908e-12
##
## Coefficients:
##               Estimate Std. Error    t value Pr(>|t|)
## (Intercept)  2.160e+02  4.448e-11  4.856e+12   <2e-16 ***
## n           -7.363e-13  3.390e-13 -2.172e+00    0.162
## n2           8.000e+00  5.543e-16  1.443e+16   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.074e-11 on 2 degrees of freedom
## Multiple R-squared:      1,  Adjusted R-squared:      1
## F-statistic: 2.782e+33 on 2 and 2 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<-5
ptable<-matrix(NA, nrow=nmax, ncol=11) # to hold results
# loop over sizes
for (ni in 1:nmax){
```

```
  n<-100*ni
  x<-runif(n) # generate a vector
  ptable[[ni, 1]]<-n
  AA<-molermat(n)
  tax<- microbenchmark(oax<-ax(x, AA), times=mbt)$time
  taxx<-microbenchmark(oaxx<-axx(x, AA), times=mbt)$time
  if (! identical(oax, oaxx)) stop("oaxx NOT correct")
  taxftn<-microbenchmark(oaxftn<-axftn(x, AA=1), times=mbt)$time
  if (! identical(oax, oaxftn)) stop("oaxftn NOT correct")
  taximp<-microbenchmark(oaximp<-aximp(x, AA=1), times=mbt)$time
  if (! identical(oax, oaximp)) stop("oaximp NOT correct")
  taxmfi<-microbenchmark(oaxmfi<-axmolerfast(x, AA=1), times=mbt)$time
  if (! identical(oax, oaxmfi)) stop("oaxmfi NOT correct")
  ptable[[ni, 2]]<-msect(tax); ptable[[ni,3]]<-msecr(tax)
  ptable[[ni, 4]]<-msect(taxx); ptable[[ni, 5]]<-msecr(taxx)
  ptable[[ni, 6]]<-msect(taxftn); ptable[[ni, 7]]<-msecr(taxftn)
  ptable[[ni, 8]]<-msect(taximp); ptable[[ni,9]]<-msecr(taximp)
  ptable[[ni, 10]]<-msect(taxmfi); ptable[[ni,11]]<-msecr(taxmfi)
}

axtym<-data.frame(n=ptable[,1], ax=ptable[,2], sd_ax=ptable[,3],  axx=ptable[,4],
                  sd_axx=ptable[,5],  axftn=ptable[,6], sd_axftn=ptable[,7],
                  aximp=ptable[,8], sd_aximp=ptable[,9],
                  axmfast=ptable[,10], sd_axmfast=ptable[,11])
print(axtym)
```

```
##     n  ax sd_ax axx sd_axx axftn sd_axftn aximp sd_aximp axmfast sd_axmfast
## 1 100  89   369 233    794   206      831  3898     1270     951        986
## 2 200  35    32  30     22   134        4 13648     1010    1719        228
## 3 300  67    34 295   1175   291        5 30949     1438    2776        924
## 4 400 452  1220  92     19   509        4 55195     1115    3759        969
## 5 500 403  1103 211    252   789        5 86189     1934    5265       1124
```

```
## 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
## 1 100  3909   4053   206  3898
## 2 200 14365  14360   134 13648
## 3 300 31150  31378   291 30949
## 4 400 54388  54028   509 55195
## 5 500 85019  84827   789 86189
```

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`.

```
##   direct matmul crossprod ftncross impcross
## 1    672     37        37      109      871
## 2   2528     32        27      136     1569
## 3   5625     57        53      293     2516
## 4  10072     94        93      501     3627
## 5  15553    242       142      697     4839
```

```
## direct - looped a*x
```

```
## matmul - A %% x
```

```
## crossprod - A * X via crossprod
```

```
## ftncross - Fortran + crossprod
```

```
## impcross - A * x fast R implicit matrix + crossprod
```

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.

```
# spgRQ.R
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
}

rqfast<-function(x){
```

```
  rq<-as.numeric(t(x) %*% axmolerfast(x))
  rq
}
rqneg<-function(x) { -rqfast(x)}
proj <- function(x) {sign(x[1]) * x/sqrt(c(crossprod(x))) } # from ravi
# Note that the c() is needed in denominator to avoid error msgs
require(BB)
n<-100
x<-rep(1,n)
x<-x/as.numeric(sqrt(crossprod(x)))
AA<-molerfast(n)
teig<-microbenchmark(evs<-eigen(AA), times=mbt)$time
cat("eigen time =", msect(teig),"sd=",msecr(teig),"\n")
```

```
## eigen time = 2533 sd= 1622
```

```
tmin<-microbenchmark(amin<-spg(x, fn=rqfast, project=proj,
                               control=list(trace=FALSE)), times=mbt)$time
tmax<-microbenchmark(amax<-spg(x, fn=rqneg, project=proj,
                               control=list(trace=FALSE)), times=mbt)$time
evalmax<-evs$values[1]
evecmax<-evs$vectors[,1]
evecmax<-sign(evecmax[1])*evecmax/sqrt(as.numeric(crossprod(evecmax))) # normalize
emax<-list(evalmax=evalmax, evecmax=evecmax)
evalmin<-evs$values[n]
evecmin<-evs$vectors[,n]
evecmin<-sign(evecmin[1])*evecmin/sqrt(as.numeric(crossprod(evecmin)))
emin<-list(evalmin=evalmin, evecmin=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 ",
      msect(tmin),"sd=",msecr(tmin),"\n")
```

```
## minimal eigensolution: Value= 5.939165e-08 in time  27369586 sd= 963227
```

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

```
## Eigenvalue - result from eigen= 5.93916e-08   vector max(abs(diff))= 0.000135496
```

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

```
## maximal eigensolution: Value= 3934.277 in time  484406 sd= 8166
```

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

```
## Eigenvalue - result from eigen= -3.761099e-06   vector max(abs(diff))= 4.747616e-06
```

```
nmax<-5
stable<-matrix(NA, nrow=nmax, ncol=4) # to hold results
# =========== works to here, but spg is slower than eigen
# loop over sizes
```

```
for (ni in 1:nmax){
  n<-50*ni
  x<-runif(n) # generate a vector
  AA<-molerfast(n) # make sure defined
  stable[[ni, 1]]<-n
  tbld<-microbenchmark(AA<-molerfast(n), times=mbt)
  tspg<-microbenchmark(aspg<-spg(x, fn=rqneg, project=proj,
                                 control=list(trace=FALSE)), times=mbt)
  teig<-microbenchmark(aseig<-eigen(AA), times=mbt)
  stable[[ni, 2]]<-msect(tspg$time)
  stable[[ni, 3]]<-msect(tbld$time)
  stable[[ni, 4]]<-msect(teig$time)
}
spgtym<-data.frame(n=stable[,1], spgrqt=stable[,2], tbld=stable[,3], teig=stable[,4])
print(round(spgtym,0))
```

```
##     n  spgrqt tbld teig
## 1  50  182066  294  791
## 2 100  760851 1131 2302
## 3 150 1816301 1807 4012
## 4 200 3372781 2139 6229
## 5 250 5504575 2519 8515
```

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.

```
require(optimx)
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))
}
```

```
mset<-c("L-BFGS-B", "BFGS", "ncg", "spg", "ucminf", "nlm", "nlminb", "nvm")
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(trace=0,starttests=FALSE, dowarn=FALSE, kkt=FALSE))
  # optansout(aall, NULL)
  summary(aall, order=value, )
}
```

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<-5
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    511
## 2   100   1449
## 3   150   4257
## 4   200   3125
## 5   250   4861
## 6    NA     NA
## 7    NA     NA
## 8    NA     NA
## 9    NA     NA
## 10   NA     NA
```

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<-molerfast(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] 386618.971 193310.315  96657.231  48332.971  24175.300  12105.330
##  [7]   6088.052   3114.812   1698.986   1132.655
##
## $RQ
## [1] 8.582807e-06
##
## $ipr
## [1] 44
##
## $msg
## [1] "Rayleigh Quotient increased in step"
```

```
cat("Geradin time=",msect(tg$time),"sd=",msecr(tg$time),"\n")
```

```
## Geradin time= 2670 sd= 9979
```

```
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] -228929753806    7726992696  244104578112  472239647405  684417090221
##  [6]  873471850096 1033005754381 1157633064483 1243135437891 1286614945412
##
## $RQ
## [1] -31.58981
##
## $ipr
## [1] 35
##
## $msg
## [1] "Small gradient -- done"
```

```
cat("Geradin time=",msect(tgn$time),"sd=",msecr(tgn$time),"\n")
```

```
## Geradin time= 462 sd= 8
```

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
   # NEED TO EXPLAIN -1 below
   res<-(-1)*(.Fortran("moler", n=as.integer(n), x=as.double(x), vout=as.double(vout)))$vout
}

require(microbenchmark)
nmax<-5
gtable<-matrix(NA, nrow=nmax, ncol=4) # to hold results
# loop over sizes
for (ni in 1:nmax){
  n<-100*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]]<-msect(tgax$time)
  tgaximp<-microbenchmark(ogaximp<-geradin(x, naximp, ident, AA=1, BB=1, control=list(trace=FALSE)), ti
  gtable[[ni, 3]]<-msect(tgaximp$time)
  tgaxftn<-microbenchmark(ogaxftn<-geradin(x, naxftn, ident, AA=1, BB=1, control=list(trace=FALSE)), ti
  gtable[[ni, 4]]<-msect(tgaxftn$time)
}

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

```
##     n    ax   aximp axftn
## 1 100  4153  119401  1731
## 2 200  6710  476684  4247
## 3 300  5581  925870  7342
## 4 400 10485 1722521 13136
## 5 500 14804 2980109 21301
```

Let us check that the eigenvalue approximations by the Geradin are is consistent with the answers via `eigen()`.

```
for (n in c(100, 200, 300, 400, 500) ) {
x<-runif(n)
evalmax<-emax$evalmax
# evecmax<-emax$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 eigenvalue - eigen result: ",gval-evalmax,"   max(abs(vector diff))=",
      max(abs(diff)), "\n")
}
```

```
## Geradin eigenvalue - eigen result:  -3.157309e-05    max(abs(vector diff))= 1.959107e-05
## Geradin eigenvalue - eigen result:  12036.95    max(abs(vector diff))= 0.07695634
## Geradin eigenvalue - eigen result:  32179.59    max(abs(vector diff))= 0.1039409
## Geradin eigenvalue - eigen result:  60427.94    max(abs(vector diff))= 0.1166165
## Geradin eigenvalue - eigen result:  96781.98    max(abs(vector diff))= 0.1236483
```

Fortran version of Geradin code

Above we performed the Rayleigh Quotient calculations in Fortran. However, the Geradin code was part of Nash (1979), and in particular A25 in the Fortran version of Nashlib (Nash (1980), see also https://github.com/pcolsen/Nash-Compact-Numerical-Methods/tree/main/fortran). We can use this to get an approximation to the All-Fortran timing for the Geradin code, and note that the time to get the minimal eigensolution is longer than that for the maximal one.

A modified version of the Fortran particularized to the Moler matrix follows (file 'a25moler.f):

```
C&&& A25
C  TEST ALG 25 USING GRID (5 POINT)
C  J.C. NASH   JULY 1978, APRIL 1989
      LOGICAL IFR
      INTEGER N,M,NOUT,NIN,KPR,LIMIT,I
      EXTERNAL APR,BPR
C     REAL EPS,P0,X(N),S(N),T(N),U(N),V(N),W(N),Y(N),RNORM
      COMMON /GSZ/ M,IFR,R(1600)
      DOUBLE PRECISION EPS,P0,RNORM,VNORM,RNV, SCALM
      DOUBLE PRECISION S(1600),T(1600),U(1600),V(1600),W(1600),
     X   X(1600),Y(1600)
C  I/O CHANNELS
      NIN=5
      NOUT=6
   1  READ(NIN,900)N, SCALM
 900  FORMAT(I6, F10.0)
      LIMIT=1000*N
      WRITE(NOUT,950)N,LIMIT
 950  FORMAT(' MOLER MATRIX ORDER',I5,'  LIMIT=',I7)
      IF(N.LE.0)STOP
      IF(SCALM.LT.0.0) WRITE(NOUT,953)
 953  FORMAT(' Finding maximal eigenvalue')
      IFR=.FALSE.
C APPROX
      EPS=16.0**(-14)
      KPR=LIMIT
      RNORM=1.0/SQRT(FLOAT(N))
      DO 10 I=1,N
        X(I)=RNORM
  10  CONTINUE
      write(NOUT,*) " About to call A25RQM"
      CALL A25RQM(N,X,EPS,KPR,S,T,U,V,W,Y,P0,NOUT,APR,BPR)
```

```
CCC        A25RQM(N,X,EPS,KPR,Y,Z,T,G,A,B,P0,IPR,APR,BPR)
      P0=P0*SCALM
      WRITE(NOUT,951)KPR,P0
 951  FORMAT(' RETURNED AFTER',I4,' PRODUCTS WITH EV=',1PE16.8)
C     TO GET TIMING STOP HERE
      STOP
      END
      SUBROUTINE BPR(N,X,V)
C  J.C. NASH   JULY 1978, APRIL 1989
C  UNITM MATRIX * X  INTO V
      INTEGER N,I
      DOUBLE PRECISION X(N),V(N)
      DO 100 I=1,N
        V(I)=X(I)
 100  CONTINUE
      RETURN
      END
      SUBROUTINE APR(N,X,V)
      integer n, i, j
      double precision x(n), V(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
         V(i)=sum
C         V(i)=-sum
C use negative to get largest ev
 20   continue
      return
      end
      SUBROUTINE A25RQM(N,X,EPS,KPR,Y,Z,T,G,A,B,P0,IPR,APR,BPR)
C  STEP 0
      INTEGER N,LP,IPR,ITN,I,LIM,COUNT
      DOUBLE PRECISION X(N),T(N),G(N),Y(N),Z(N),PN,A(N),B(N)
      DOUBLE PRECISION EPS,TOL,P0,PA,XAX,XBX,XAT,XBT,TAT,TBT,W,K,
     X   D,V,GG,BETA,TABT,U
C  ALGORITHM 25 RAYLEIGH QUOTIENT MINIMIZATION BY CONJUGATE GRADIENTS
C  J.C. NASH   JULY 1978, FEBRUARY 1980, APRIL 1989
C    N   =  ORDER OF PROBLEM
C    X   =  INITIAL (APPROXIMATE?) EIGENVECTOR
C   EPS  =  MACHINE PRECISION
C&&& for Microsoft test replace with actual names
C   APR,BPR  ARE NAMES OF SUBROUTINES WHICH FORM THE PRODUCTS
C          V= A*X    VIA   CALL APR(N,X,V)
C          T= B*X    VIA   CALL BPR(N,X,T)
C   KPR  =  LIMIT ON THE NUMBER OF PRODUCTS (INPUT) (TAKES ROLE OF IPR)
C        =  PRODUCTS USED (OUTPUT)
```

```
C  Y,Z,T,G,A,B RE WORKING VECTORS IN AT LEAST N ELEMENTS
C   P0   =  APPROXIMATE EIGENVALUE (OUTPUT)
C   IPR  =  PRINT CHANNEL   PRINTING IF IPR.GT.0
C  IBM VALUE   -  APPROX. LARGEST NUMBER REPRESENTABLE.
C&&&       PA=R1MACH(2)
      write(6, 960) N
 960  FORMAT(' In A25RQM Geradin Rayleigh Quotient Min for N=', I5)
      PA=1E+35
      LIM=KPR
      KPR=0
      TOL=N*N*EPS*EPS
C  STEP 1
  10  KPR=KPR+1
      IF(KPR.GT.LIM)RETURN
C  FIND LIMIT IN ORIGINAL PROGRAMS
      CALL APR(N,X,A)
      CALL BPR(N,X,B)
C  STEP 2
      XAX=0.0
      XBX=0.0
      DO 25 I=1,N
        XAX=XAX+X(I)*A(I)
        XBX=XBX+X(I)*B(I)
  25  CONTINUE
C  STEP 3
      IF(XBX.LT.TOL)STOP
C  STEP 4
      P0=XAX/XBX
      IF(P0.GE.PA)RETURN
      IF(IPR.GT.0)WRITE(IPR,963)KPR,P0
 963  FORMAT( 1H ,I4,' PRODUCTS, EST. EIGENVALUE=',1PE16.8)
C  STEP 5
      PA=P0
C  STEP 6
      GG=0.0
      DO 65 I=1,N
        G(I)=2.0*(A(I)-P0*B(I))/XBX
        GG=GG+G(I)**2
  65  CONTINUE
C  STEP 7
      IF(IPR.GT.0)WRITE(IPR,964)GG
 964  FORMAT(' GRADIENT NORM SQUARED=',1PE16.8)
      IF(GG.LT.TOL)RETURN
C  STEP 8
      DO 85 I=1,N
        T(I)=-G(I)
  85  CONTINUE
C  STEP 9
      DO 240 ITN=1,N
C  STEP 10
        KPR=KPR+1
        IF(KPR.GT.LIM)RETURN
        CALL APR(N,T,Y)
        CALL BPR(N,T,Z)
```

```
C  STEP 11
        TAT=0.0
        TBT=0.0
        XAT=0.0
        XBT=0.0
        DO 115 I=1,N
        TAT=TAT+T(I)*Y(I)
         XAT=XAT+X(I)*Y(I)
          TBT=TBT+T(I)*Z(I)
         XBT=XBT+X(I)*Z(I)
 115    CONTINUE
C  STEP 12
        U=TAT*XBT-XAT*TBT
        V=TAT*XBX-XAX*TBT
        W=XAT*XBX-XAX*XBT
        D=V*V-4.0*U*W
C  STEP 13
        IF(D.LT.0)STOP
C  MAY NOT WISH TO STOP
C  STEP 14
        D=SQRT(D)
        IF(V.GT.0.0)GOTO 145
        K=0.5*(D-V)/U
        GOTO 150
 145    K=-2.0*W/(D+V)
 150    COUNT=0
C  STEP 15
        XAX=0.0
        XBX=0.0
        DO 155 I=1,N
          A(I)=A(I)+K*Y(I)
          B(I)=B(I)+K*Z(I)
          W=X(I)
          X(I)=W+K*T(I)
          IF(W.EQ.X(I))COUNT=COUNT+1
          XAX=XAX+X(I)*A(I)
          XBX=XBX+X(I)*B(I)
 155    CONTINUE
C  STEP 16
        IF(XBX.LT.TOL)STOP
        PN=XAX/XBX
C  STEP 17
        IF(COUNT.LT.N)GOTO 180
        IF(ITN.EQ.1)RETURN
        GOTO 10
C  STEP 18
 180    IF(PN.LT.P0)GOTO 190
        IF(ITN.EQ.1)RETURN
        GOTO 10
C  STEP 19
 190    P0=PN
        GG=0.0
        DO 195 I=1,N
        G(I)=2.0*(A(I)-PN*B(I))/XBX
```

```
        GG=GG+G(I)**2
 195    CONTINUE
C  STEP 20
        IF(GG.LT.TOL)GOTO 10
C  STEP 21
        XBT=0.0
        DO 215 I=1,N
         XBT=XBT+X(I)*Z(I)
 215    CONTINUE
C  STEP 22
        TABT=0.0
        BETA=0.0
        DO 225 I=1,N
          W=Y(I)-PN*Z(I)
          TABT=TABT+T(I)*W
          BETA=BETA+G(I)*(W-G(I)*XBT)
 225    CONTINUE
C  STEP 23
        BETA=BETA/TABT
        DO 235 I=1,N
          T(I)=BETA*T(I)-G(I)
 235    CONTINUE
C  STEP 24
 240  CONTINUE
C  STEP 25
      GOTO 10
C  NO STEP 26  -  HAVE USED RETURN INSTEAD
      END
```

This can be easily compiled on most Linux systems. Here we do so via R.

```
system("gfortran ./a25moler.f")
```

The program takes as input a single line with an integer for the size of the Moler matrix to use (maximum 1600 as per the program declarations) and a number that is either 1.0 for the minimal eigensolution or -1.0 for the maximal one. Though it is slightly clumsy, we have created files for matrices of size 100, 200, 300, 400 and 500.

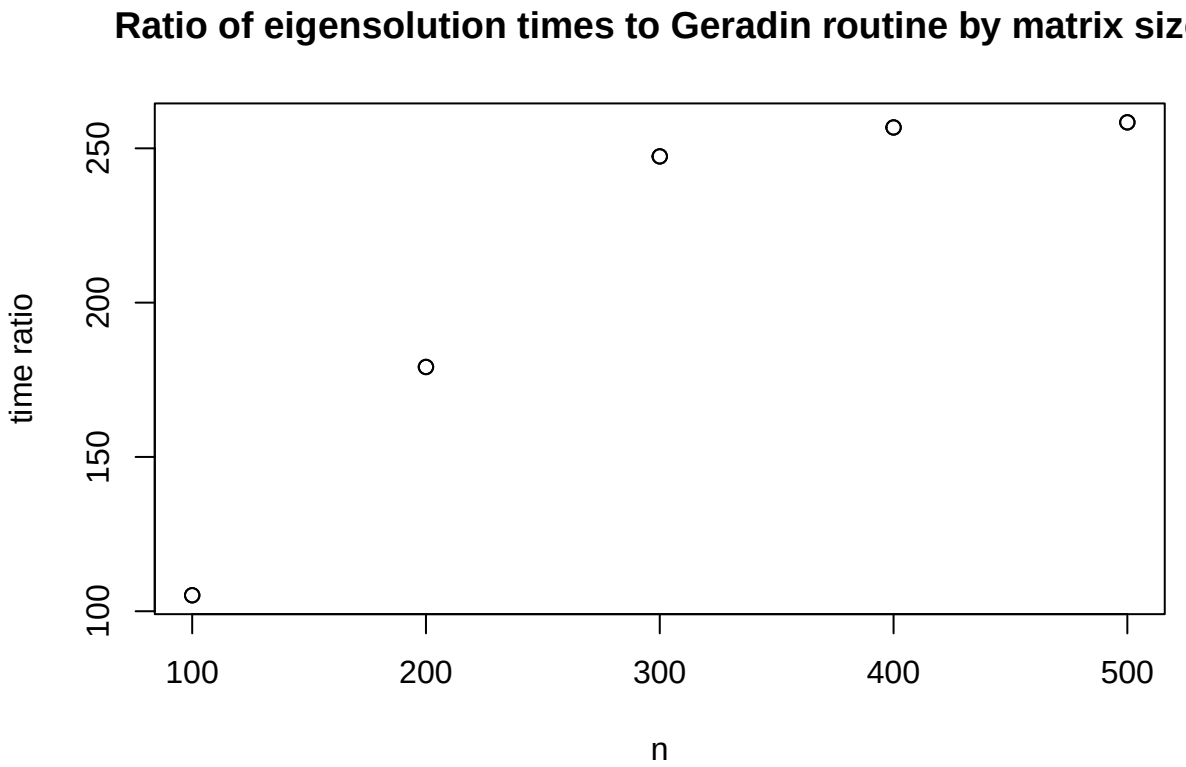
Let us compare timings for R's `eigen()` with the Fortran Geradin program for these cases.

```
## Geradin fortran version a25moler.f
## eigen(): n=100 build time= 611   eigen time= 2009
## eigen: Minimal Eigenvalue = 5.04378e-14   RQ= -6.463237e-17
## A25RQM N= 100  matvec ops= 198  Min Est. EV= 2.586285e-18   Gradient= 1.515018e-30  time= 7
## A25RQM N= 100  matvec ops= 13  Max Est. EV= 3934.277   Gradient= 2.441749e-22  time= 1
## eigen: Maximal Eigenvalue = 3934.277   RQ= 3934.277
## eigen(): n=200 build time= 2244   eigen time= 5788
## eigen: Minimal Eigenvalue = -2.491097e-14   RQ= 8.728006e-17
## A25RQM N= 200  matvec ops= 412  Min Est. EV= 2.411243e-19   Gradient= 4.764326e-30  time= 41
## A25RQM N= 200  matvec ops= 9  Max Est. EV= 15971.22   Gradient= 4.973105e-11  time= 2
## eigen: Maximal Eigenvalue = 15971.22   RQ= 15971.22
```

```
## eigen(): n=300 build time= 3406   eigen time= 11192
## eigen: Minimal Eigenvalue = -1.716203e-13   RQ= -1.255081e-16
## A25RQM N= 300  matvec ops= 598  Min Est. EV= -5.995543e-20   Gradient= 1.377315e-29  time= 126
## A25RQM N= 300  matvec ops= 11  Max Est. EV= 36113.87   Gradient= 2.340489e-13  time= 4
## eigen: Maximal Eigenvalue = 36113.87   RQ= 36113.87
## eigen(): n=400 build time= 5484   eigen time= 20162
## eigen: Minimal Eigenvalue = -7.201366e-13   RQ= 1.550574e-17
## A25RQM N= 400  matvec ops= 553  Min Est. EV= -1.216296e-19   Gradient= 1.912164e-29  time= 204
## A25RQM N= 400  matvec ops= 12  Max Est. EV= 64362.22   Gradient= 3.036479e-19  time= 6
## eigen: Maximal Eigenvalue = 64362.22   RQ= 64362.22
## eigen(): n=500 build time= 7859   eigen time= 32373
## eigen: Minimal Eigenvalue = 1.949665e-12   RQ= -9.8516e-18
## A25RQM N= 500  matvec ops= 733  Min Est. EV= -3.794744e-18   Gradient= 4.428853e-29  time= 418
## A25RQM N= 500  matvec ops= 12  Max Est. EV= 100716.3   Gradient= 2.030114e-15  time= 8
## eigen: Maximal Eigenvalue = 100716.3   RQ= 100716.3
```

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 = 77927  eigen(): 1253718   Rcgminu: 411255  Geradin: 334346
## Ratios: build= 0.2330729 eigen= 3.749762   Rcgminu= 1.230028
```

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.

———. 1980. "NASHLIB: Algorithms for Compact Numerical Methods, now available in FORTRAN."