

# 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 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 `molerma'
  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	osize	buildi	builddir	eigentime	eigentimr	bfast	bfastr
## 1	50	20216	1173	855	510	247	512	1045
## 2	100	80216	3345	582	1559	63	677	46
## 3	150	180216	7324	710	4354	190	1023	45
## 4	200	320216	12994	872	9007	718	1446	53
## 5	250	500216	20180	553	15885	510	2151	262
## 6	300	720216	29251	682	26235	912	2657	685
## 7	350	980216	39864	1390	40224	711	4601	7208
## 8	400	1280216	51991	958	58801	1053	5140	7171
## 9	450	1620216	67892	6795	82032	1092	7362	9582
## 10	500	2000216	81840	2055	110934	1028	6888	7099

## osize - matrix size in bytes

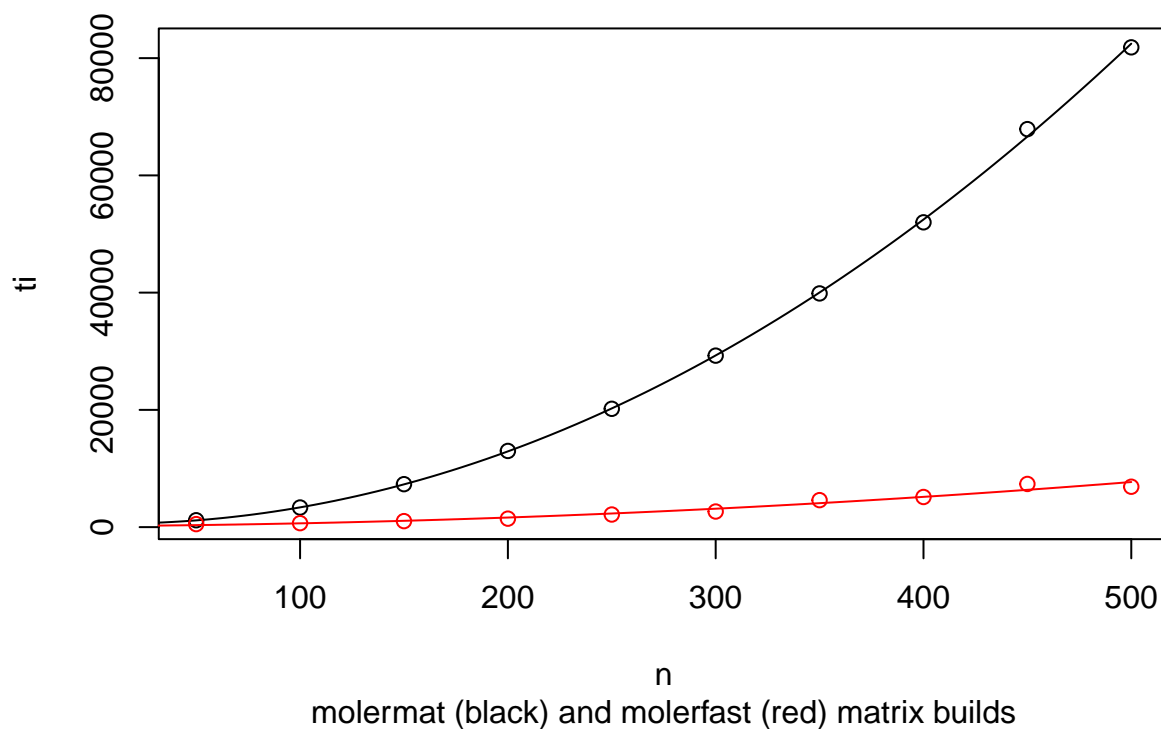
```
## eigentime - all eigensolutions time
## buildi - interpreted build time, range
## bfast - interpreted vectorized build time
## Times converted to milliseconds
```

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
## -629.39 -139.57  -30.15   29.11 1274.38
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 635.38333 672.693364   0.945   0.376
## n           -6.740318   5.618902  -1.200   0.269
## n2             0.340817   0.009956  34.231 4.71e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 571.9 on 7 degrees of freedom
## Multiple R-squared:  0.9997, Adjusted R-squared:  0.9996
## F-statistic: 1.088e+04 on 2 and 7 DF, p-value: 5.963e-13
##
## Call:
## lm(formula = tf ~ n + n2)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -802.38 -185.60  -35.85   136.25 1001.59
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 166.36667 666.21131   0.250   0.8100
## n             2.21318   5.56476   0.398   0.7027
## n2            0.02567   0.00986   2.603   0.0353 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 566.4 on 7 degrees of freedom
## Multiple R-squared:  0.9622, Adjusted R-squared:  0.9514
## F-statistic: 89.12 on 2 and 7 DF, p-value: 1.049e-05
```

## Execution time vs matrix size



```
## 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
```

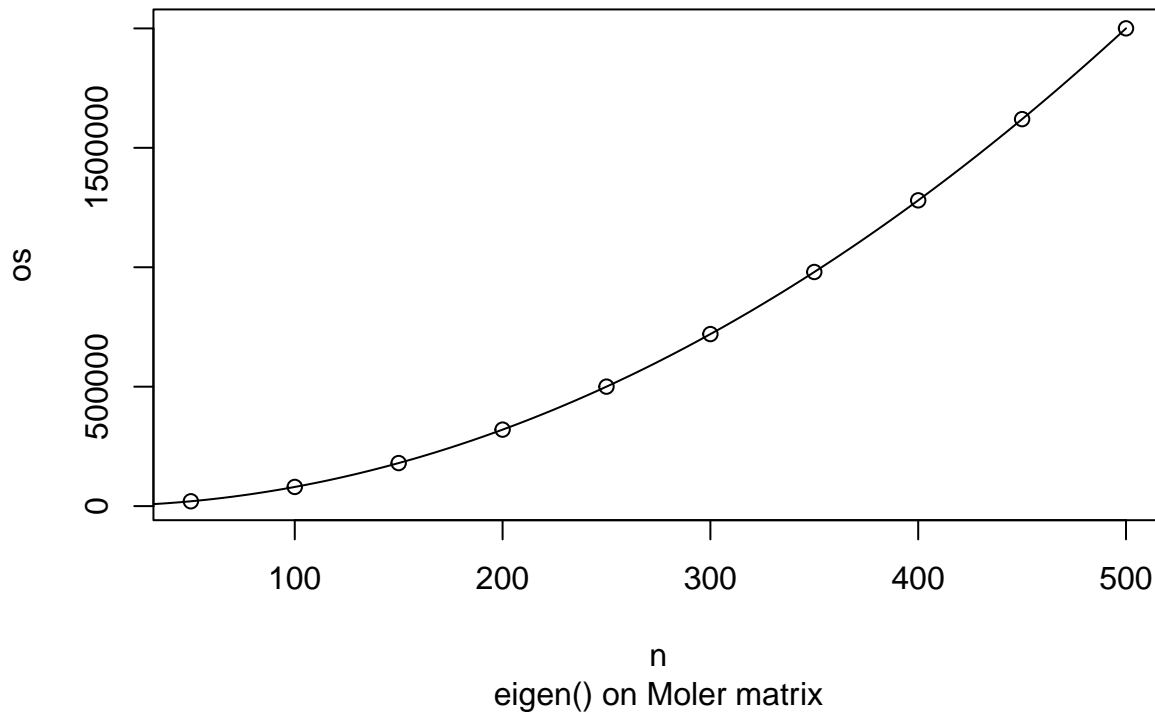
```
##
```

```
## 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
```

## Execution time vs matrix size



## 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  $x^T A x$  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 product with `x` gives us the quadratic form  $x^T A x$ . 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

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<-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   50  71  335  62   281   135     615  1071   1242    511     981
## 2  100  12   23  10     3    31      5  3262    529    642      25
## 3  150  24   59  20     3    62      6  7249    716   1085     79
## 4  200  39   76  36     6   107      4 12928    879   1579    148
## 5  250  35   10  59    19  166     11 19862    803   2250   1058
## 6  300  48   15  80    15  234      4 29597    7263  2649    901
## 7  350  65   24 106    10  314      4 38524    934   3296    921
## 8  400  82   25 139    12  411      6 50403   1188   4057   1054
## 9  450 104   32 180    30  519      9 64972   7008   4793   1358
## 10 500 124   26 222    43  638      4 80379   7400   5408   1104

```

```

## 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  1244  1235   135  1071
## 2  100 3357  3355    31  3262

```



```
## 3 150 7348 7344 62 7249
## 4 200 13033 13030 107 12928
## 5 250 20215 20239 166 19862
## 6 300 29299 29331 234 29597
## 7 350 39929 39970 314 38524
## 8 400 52073 52130 411 50403
## 9 450 67996 68072 519 64972
## 10 500 81964 82062 638 80379
```

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: 17514 sd= 469
## ray1: mat-mult algorithm: 240 sd= 160
## ray2: crossprod algorithm: 238 sd= 169
## ray3: ax Fortran + crossprod: 682.6669
## ray3: ax fast R implicit + crossprod: 5694 sd= 1538
```

Here we see that the use of either the matrix multiplication in `ray1` or of `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.

## 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 'molerfast'
  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 = 2367 sd= 574

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)))
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 26505073 sd= 168549

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 488215 sd= 8766

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(aspq<-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)
}
spgty<-data.frame(n=stable[,1], spgrqt=stable[,2], tbld=stable[,3], teig=stable[,4])
print(round(spgty,0))

##      n spgrqt tbld teig
## 1  50 184521  279  549
## 2 100 778303  604 2108
## 3 150 1848876 1754 4068
## 4 200 3385922 2120 6290
## 5 250 5524652 2524 8118

```

## 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::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", "nbg", "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 nbg. 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="nbg",
    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= 3176 sd= 11952

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= 472 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<-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]]<-msect(tgax$time)
  tgaximp<-microbenchmark(ogaximp<-geradin(x, naximp, ident, AA=1, BB=1, control=list(trace=FALSE)), times=mbt)
  gtable[[ni, 3]]<-msect(tgaximp$time)
  tgaxftn<-microbenchmark(ogaxftn<-geradin(x, naxftn, ident, AA=1, BB=1, control=list(trace=FALSE)), times=mbt)
  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   50    647    27284    937
## 2  100   1413   89498   1265
## 3  150   2415  238622   2491
## 4  200   3191  425920   3841
## 5  250   5330  725304   6043
## 6  300   5483  980984   7649
## 7  350   7553 1334017  10333
## 8  400   9340 1761336  13320
## 9  450  11479 2292495  16842
## 10 500  14803 2820204  20842

```

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

```
n<-100
x<-runif(n)
evalmax<-emax$evalmax
evecmac<-emax$evecmac
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:  -6.494235e-06    max(abs(vector diff))= 7.995635e-06
```

### Fortran version of Geradin code

Above we performed the Rayleigh Quotient calculations in Fortran. However, the Geradin code was part of (?), 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,PO,X(N),S(N),T(N),U(N),V(N),W(N),Y(N),RNORM
      COMMON /GSZ/ M,IFR,R(1600)
      DOUBLE PRECISION EPS,PO,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,PO,NOUT,APR,BPR)
CCC      A25RQM(N,X,EPS,KPR,Y,Z,T,G,A,B,PO,IPR,APR,BPR)
        PO=PO*SCALM
        WRITE(NOUT,951)KPR,PO
    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,PO,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,PO,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   PO   =  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
      PO=XAX/XBX
      IF(PO.GE.PA)RETURN
      IF(IPR.GT.0)WRITE(IPR,963)KPR,PO
963  FORMAT( 1H ,I4,' PRODUCTS, EST. EIGENVALUE=',1PE16.8)
C STEP 5
      PA=PO
C STEP 6
      GG=0.0
      DO 65 I=1,N
          G(I)=2.0*(A(I)-PO*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.

```
cat("Geradin fortran version a25moler.f")
```

```
## Geradin fortran version a25moler.f
```

```

tmin100<-msect(microbenchmark(system("./a.out <n100min.txt > out100min.txt"), times=mbt)$time)
tmax100<-msect(microbenchmark(system("./a.out <n100max.txt > out100max.txt"), times=mbt)$time)
tbld100<-msect(microbenchmark(AA<-molerfast(100), times=mbt)$time)
teig100<-msect(microbenchmark(a100<-eigen(AA), times=mbt)$time)
cat("eigen(): n=100 build time=",tbld100," eigen time=",teig100,"\n")

```

```
## eigen(): n=100 build time= 600 eigen time= 2035
```

```

vecmin<-a100$vectors[,100]
vecmax<-a100$vectors[,1]
cat("eigen: Minimal Eigenvalue =",a100$values[100]," RQ=", ray1(vecmin, AA),"\n")

```

```

## eigen: Minimal Eigenvalue = 5.04378e-14   RQ= -6.463237e-17
cat("Geradin-fortran - minimal eigenvalue time=",tmin100,"\n")

## Geradin-fortran - minimal eigenvalue time= 6301
system("cat out100min.txt")
cat("eigen: Maximal Eigenvalue =",a100$values[1]," RQ=", ray1(vecmax, AA),"\n")

## eigen: Maximal Eigenvalue = 3934.277   RQ= 3934.277
cat("Geradin-fortran - maximal eigenvalue time=",tmin100,"\n")

## Geradin-fortran - maximal eigenvalue time= 6301
system("cat out100max.txt")

tmin200<-msect(microbenchmark(system("./a.out <n200min.txt > out200min.txt"), times=mbt)$time)
tmax200<-msect(microbenchmark(system("./a.out <n200max.txt > out200max.txt"), times=mbt)$time)
tbld200<-msect(microbenchmark(AA<-molerfast(200), times=mbt)$time)
teig200<-msect(microbenchmark(a200<-eigen(AA), times=mbt)$time)
cat("eigen(): n=200 build time=",tbld200," eigen time=",teig200,"\n")

## eigen(): n=200 build time= 1373   eigen time= 6436
vecmin<-a200$vectors[,200]
vecmax<-a200$vectors[,1]
cat("eigen: Minimal Eigenvalue =",a200$values[200]," RQ=", ray1(vecmin, AA),"\n")

## eigen: Minimal Eigenvalue = -2.491097e-14   RQ= 8.728006e-17
cat("Geradin-fortran - minimal eigenvalue time=",tmin200,"\n")

## Geradin-fortran - minimal eigenvalue time= 41284
system("cat out200min.txt")
cat("eigen: Maximal Eigenvalue =",a200$values[1]," RQ=", ray1(vecmax, AA),"\n")

## eigen: Maximal Eigenvalue = 15971.22   RQ= 15971.22
cat("Geradin-fortran - maximal eigenvalue time=",tmin200,"\n")

## Geradin-fortran - maximal eigenvalue time= 41284
system("cat out200max.txt")

tmin300<-msect(microbenchmark(system("./a.out <n300min.txt > out300min.txt"), times=mbt)$time)
tmax300<-msect(microbenchmark(system("./a.out <n300max.txt > out300max.txt"), times=mbt)$time)
tbld300<-msect(microbenchmark(AA<-molerfast(300), times=mbt)$time)
teig300<-msect(microbenchmark(a300<-eigen(AA), times=mbt)$time)
cat("eigen(): n=300 build time=",tbld300," eigen time=",teig300,"\n")

## eigen(): n=300 build time= 2429   eigen time= 11323
vecmin<-a300$vectors[,300]
vecmax<-a300$vectors[,1]
cat("eigen: Minimal Eigenvalue =",a300$values[300]," RQ=", ray1(vecmin, AA),"\n")

## eigen: Minimal Eigenvalue = -1.716203e-13   RQ= -1.255081e-16
cat("Geradin-fortran - minimal eigenvalue time=",tmin300,"\n")

```

```

## Geradin-fortran - minimal eigenvalue time= 126895
system("cat out300min.txt")
cat("eigen: Maximal Eigenvalue =",a300$values[1]," RQ=", ray1(vecmax, AA),"\n")

## eigen: Maximal Eigenvalue = 36113.87 RQ= 36113.87
cat("Geradin-fortran - maximal eigenvalue time=",tmin300,"\n")

## Geradin-fortran - maximal eigenvalue time= 126895
system("cat out300max.txt")

tmin400<-msect(microbenchmark(system("./a.out <n400min.txt > out400min.txt"), times=mbt)$time)
tmax400<-msect(microbenchmark(system("./a.out <n400max.txt > out400max.txt"), times=mbt)$time)
tbld400<-msect(microbenchmark(AA<-molerfast(400), times=mbt)$time)
teig400<-msect(microbenchmark(a400<-eigen(AA), times=mbt)$time)
cat("eigen(): n=400 build time=",tbld400," eigen time=",teig400,"\n")

## eigen(): n=400 build time= 4539 eigen time= 20391
vecmin<-a400$vectors[,400]
vecmax<-a400$vectors[,1]
cat("eigen: Minimal Eigenvalue =",a400$values[400]," RQ=", ray1(vecmin, AA),"\n")

## eigen: Minimal Eigenvalue = -7.201366e-13 RQ= 1.550574e-17
cat("Geradin-fortran - minimal eigenvalue time=",tmin400,"\n")

## Geradin-fortran - minimal eigenvalue time= 203937
system("cat out400min.txt")
cat("eigen: Maximal Eigenvalue =",a400$values[1]," RQ=", ray1(vecmax, AA),"\n")

## eigen: Maximal Eigenvalue = 64362.22 RQ= 64362.22
cat("Geradin-fortran - maximal eigenvalue time=",tmin400,"\n")

## Geradin-fortran - maximal eigenvalue time= 203937
system("cat out400max.txt")

tmin500<-msect(microbenchmark(system("./a.out <n500min.txt > out500min.txt"), times=mbt)$time)
tmax500<-msect(microbenchmark(system("./a.out <n500max.txt > out500max.txt"), times=mbt)$time)
tbld500<-msect(microbenchmark(AA<-molerfast(500), times=mbt)$time)
teig500<-msect(microbenchmark(a500<-eigen(AA), times=mbt)$time)
cat("eigen(): n=500 build time=",tbld500," eigen time=",teig500,"\n")

## eigen(): n=500 build time= 6949 eigen time= 34716
vecmin<-a500$vectors[,500]
vecmax<-a500$vectors[,1]
cat("eigen: Minimal Eigenvalue =",a500$values[500]," RQ=", ray1(vecmin, AA),"\n")

## eigen: Minimal Eigenvalue = 1.949665e-12 RQ= -9.8516e-18
cat("Geradin-fortran - minimal eigenvalue time=",tmin500,"\n")

## Geradin-fortran - minimal eigenvalue time= 416530
system("cat out500min.txt")
cat("eigen: Maximal Eigenvalue =",a500$values[1]," RQ=", ray1(vecmax, AA),"\n")

```

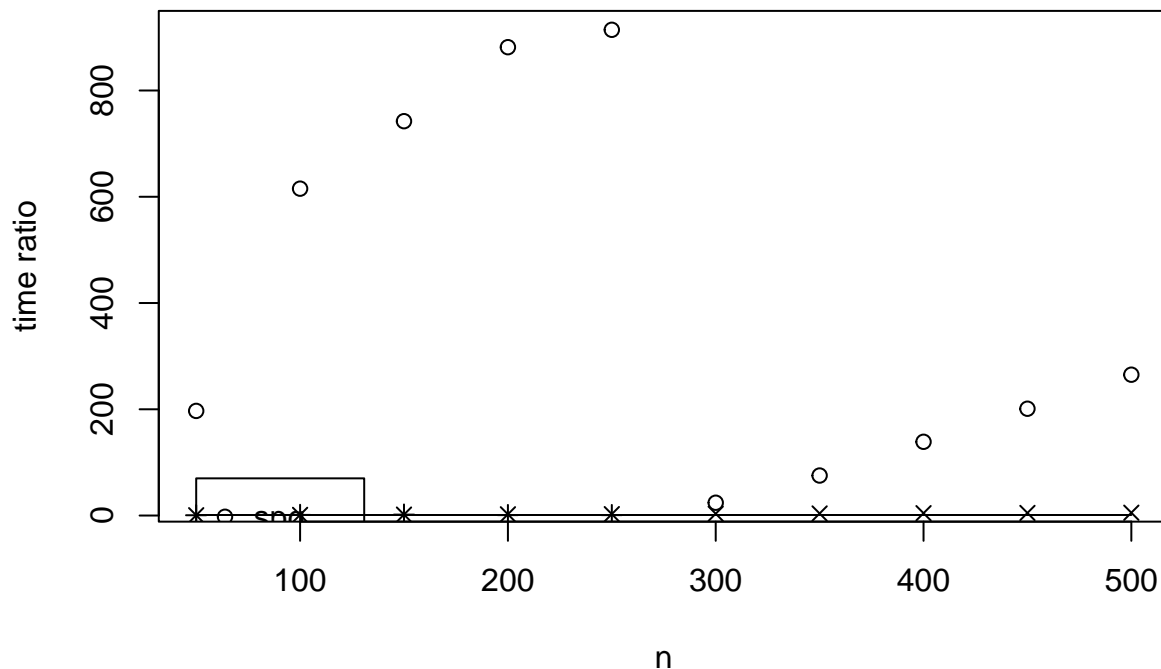
```
## eigen: Maximal Eigenvalue = 100716.3   RQ= 100716.3
cat("Geradin-fortran - maximal eigenvalue time=",tmin500,"\\n")

## Geradin-fortran - maximal eigenvalue time= 416530
system("cat out500max.txt")
```

## 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.

### Ratio of eigensolution times to Geradin routine by matrix size



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

```
## Times in seconds
## Build = 75719   eigen(): 856613   Rcgminu: 344915   Geradin: 317873
## Ratios: build= 0.2382052 eigen= 2.694828   Rcgminu= 1.085072
```

## 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.”