Timing Rayleigh Quotient minimization in R

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

July 2012, with update December 2016

Abstract

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

1 The computational task

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

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

2 Timings and speedups

In R , execution times can be measured by the function `system.time`, and in particular the first element of the object this function returns the time taken by the code which is the argument to the function. 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.

Note that the timings here are intended to provide a guide to the relative efficiency of equivalent computations of the same results. There are hardware, operating system, library, and package effects that we will largely ignore. The particular machine used to develop this article is described using the following script.

```
sessionInfo()

## R version 3.3.2 (2016-10-31)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Linux Mint 18
##
## locale:
##  [1] LC_CTYPE=en_CA.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_CA.UTF-8        LC_COLLATE=en_CA.UTF-8
##  [5] LC_MONETARY=en_CA.UTF-8    LC_MESSAGES=en_CA.UTF-8
##  [7] LC_PAPER=en_CA.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_CA.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] knitr_1.15.1
##
## loaded via a namespace (and not attached):
## [1] magrittr_1.5  tools_3.3.2   stringi_1.1.2 digest_0.6.10 stringr_1.1.0
## [6] evaluate_0.10

print(system("inxi")) # special bash script for Linux

## [1] 0
```

The results presented in this article will very likely differ on other machinery and other operating environments. However, the code is included – you are more than welcome to re-run the timings, and I would be delighted to learn of any significant variations from the general findings reported here, especially if the source of such variations can be discovered.

3 Our example matrix

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

Here is the code for generating the Moler matrix.

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

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

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

```
diag(A) <- 1:n
A
}
```

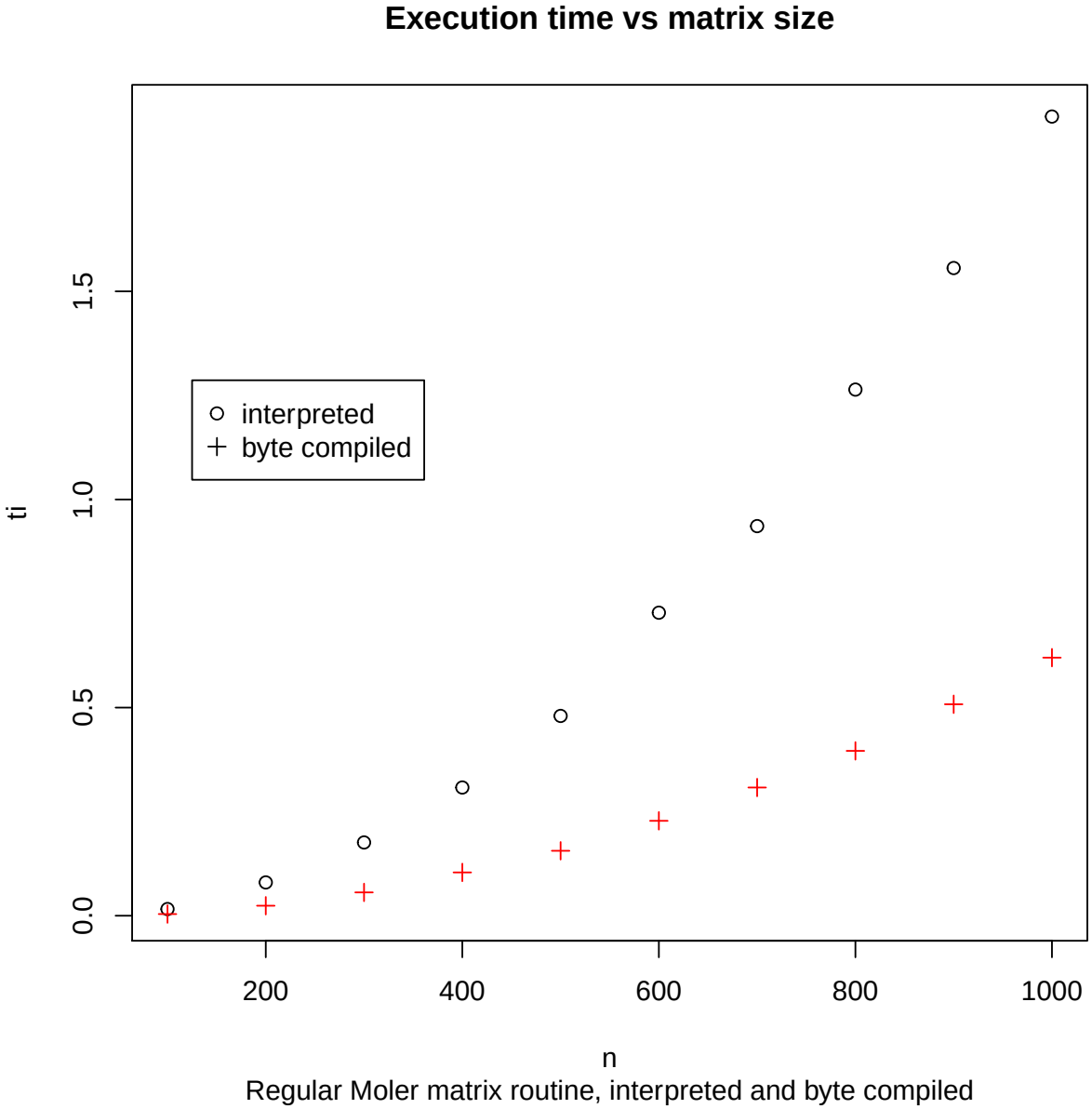
3.1 Time to build the matrix

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

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

```
##       n buildi buildc   osize eigentime bfast bfastc
## 1   100  0.016  0.004   80200     0.004 0.000  0.000
## 2   200  0.080  0.024  320200     0.016 0.004  0.004
## 3   300  0.176  0.056  720200     0.056 0.004  0.004
## 4   400  0.308  0.104 1280200     0.124 0.008  0.008
## 5   500  0.480  0.156 2000200     0.288 0.012  0.008
## 6   600  0.728  0.228 2880200     0.428 0.012  0.012
## 7   700  0.936  0.308 3920200     0.684 0.024  0.020
## 8   800  1.264  0.396 5120200     0.972 0.028  0.020
## 9   900  1.556  0.508 6480200     1.352 0.032  0.036
## 10 1000  1.920  0.620 8000200     1.908 0.040  0.040
## buildi - interpreted build time; buildc - byte compiled build time
## osize - matrix size in bytes; eigentime - all eigensolutions time
## bfast - interpreted vectorized build time; bfastc - same code, byte compiled time
```

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



Regular Moler matrix routine, interpreted and byte compiled

We can also model these timings. If we try to fit a quadratic model in the matrix size, we find almost perfect fits for both interpreted and byte-compiled timings over the cases tried.

```
n2<-n*n
itime<-lm(ti~n+n2)
summary(itime)

##
## Call:
## lm(formula = ti ~ n + n2)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -0.017945 -0.008559 -0.003264  0.003145  0.023691
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
```

```
## (Intercept) -1.460e-02  1.865e-02  -0.783    0.459
## n            8.864e-05  7.788e-05   1.138    0.293
## n2           1.850e-06  6.900e-08  26.813 2.57e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.01585 on 7 degrees of freedom
## Multiple R-squared:  0.9995,Adjusted R-squared:  0.9994
## F-statistic:  7761 on 2 and 7 DF,  p-value: 1.945e-12

ctime<-lm(tc~n+n2)
summary(ctime)

##
## Call:
## lm(formula = tc ~ n + n2)
##
## Residuals:
##        Min         1Q     Median         3Q        Max
## -0.0038545 -0.0010000 -0.0000727  0.0011636  0.0035636
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -4.800e-03  2.897e-03  -1.657   0.1415
## n            2.582e-05  1.210e-05   2.134   0.0703 .
## n2           6.000e-07  1.072e-08  55.971 1.52e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.002463 on 7 degrees of freedom
## Multiple R-squared:  0.9999,Adjusted R-squared:  0.9999
## F-statistic: 3.354e+04 on 2 and 7 DF,  p-value: 1.16e-14

osize<-lm(os~n+n2)
summary(osize)

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

##
## Call:
## lm(formula = os ~ n + n2)
##
## Residuals:
##        Min         1Q     Median         3Q        Max
## -7.707e-10 -1.136e-11  8.319e-11  1.607e-10  1.963e-10
##
## Coefficients:
##               Estimate Std. Error    t value Pr(>|t|)
## (Intercept)  2.000e+02  3.821e-10  5.235e+11   <2e-16 ***
## n           -4.101e-12  1.596e-12 -2.570e+00    0.037 *
## n2           8.000e+00  1.414e-15  5.659e+15   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.248e-10 on 7 degrees of freedom
## Multiple R-squared:      1,Adjusted R-squared:      1
## F-statistic: 3.188e+32 on 2 and 7 DF,  p-value: < 2.2e-16
```

4 Computing the Rayleigh Quotient

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

```
rqdir<-function(x, AA){
  rq<-0.0
  n<-length(x) # assume x, AA conformable
  for (i in 1:n) {
     for (j in 1:n) {
        rq<-rq-x[i]*AA[[i,j]]*x[j] # Note - sign
```

```
    }
  }
  rq
}
```

Somewhat better (as we shall show below) is

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

and better still is

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

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

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

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

5 Matrix-vector products

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

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

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

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

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

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

```
axmolerfast <- function(x, AA=1) {
# A fast and memory-saving version of A%*%x
# For Moler matrix. Note we need a matrix argument to match other functions
n <- length(x)
j <- 1:n
ax <- rep(0, n)
for (i in 1:n) {
term <- x * (pmin(i, j) - 2)
ax[i] <- sum(term[-i])
}
ax <- ax + j*x
ax
}
```

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

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

This is then compiled in a form suitable for R use by the command:

```
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. Normally the compilation is run as a command-line tool, and at first was run in Ubuntu Linux in a directory containing the file `moler.f` but outside this vignette. When I came to check and possibly update this file, I discovered that there were some minor syntax changes in the **knitr** package so that directives for `cache` caused errors and needed to be deleted. Also the dynamic load library files `moler.o` and `moler.so` would not load, apparently because they were compiled using different versions of the Fortran libraries. Recompiling removed this issue. However, we can automate the process to avoid future difficulties as the operating system infrastructure is updated.

```
system("rm moler.so")
system("rm moler.o")
system("R CMD SHLIB moler.f")
cat("Dynamic libraries rebuilt \n")

## Dynamic libraries rebuilt
```

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

```
## Is the mat multiply loaded?  TRUE

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

We can also byte compile each of the routines above

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

Now it is possible to time the different approaches to the matrix-vector product. We only use matrix sizes up to 500 here.

```
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<-100*ni
  x<-runif(n) # generate a vector
  ptable[[ni, 1]]<-n
  AA<-molermat(n)
  tax<-system.time(oax<-replicate(20,ax(x, AA))[,1])[[1]]
  taxc<-system.time(oaxc<-replicate(20,axc(x, AA))[,1])[[1]]
  if (! identical(oax, oaxc)) stop("oaxc NOT correct")
  taxx<-system.time(oaxx<-replicate(20,axx(x, AA))[,1])[[1]]
  if (! identical(oax, oaxx)) stop("oaxx NOT correct")
  taxxc<-system.time(oaxxc<-replicate(20,axxc(x, AA))[,1])[[1]]
  if (! identical(oax, oaxxc)) stop("oaxxc NOT correct")
  taxftn<-system.time(oaxftn<-replicate(20,axftn(x, AA=1))[,1])[[1]]
  if (! identical(oax, oaxftn)) stop("oaxftn NOT correct")
  taxftnc<-system.time(oaxftnc<-replicate(20,axftnc(x, AA=1))[,1])[[1]]
  if (! identical(oax, oaxftnc)) stop("oaxftnc NOT correct")
  taximp<-system.time(oaximp<-replicate(20,aximp(x, AA=1))[,1])[[1]]
  if (! identical(oax, oaximp)) stop("oaximp NOT correct")
  taximpc<-system.time(oaximpc<-replicate(20,aximpc(x, AA=1))[,1])[[1]]
  if (! identical(oax, oaximpc)) stop("oaximpc NOT correct")
  taxmfi<-system.time(oaxmfi<-replicate(20,axmolerfast(x, AA=1))[,1])[[1]]
  if (! identical(oax, oaxmfi)) stop("oaxmfi NOT correct")
  taxmfc<-system.time(oaxmfc<-replicate(20,axmfc(x, AA=1))[,1])[[1]]
  if (! identical(oax, oaxmfc)) stop("oaxmfc NOT correct")
  ptable[[ni, 2]]<-tax
  ptable[[ni, 3]]<-taxc
  ptable[[ni, 4]]<-taxx
  ptable[[ni, 5]]<-taxxc
  ptable[[ni, 6]]<-taxftn
  ptable[[ni, 7]]<-taxftnc
  ptable[[ni, 8]]<-taximp
  ptable[[ni, 9]]<-taximpc
  ptable[[ni, 10]]<-taxmfi
  ptable[[ni, 11]]<-taxmfc
#  cat(n,tax, taxc, taxx, taxxc, taxftn, taxftnc, taximp, taximpc,"\n")
}
axtym<-data.frame(n=ptable[,1], ax=ptable[,2], axc=ptable[,3],
  axx=ptable[,4], axxc=ptable[,5],
  axftn=ptable[,6], axftnc=ptable[,7],
  aximp=ptable[,8], aximpc=ptable[,9], axmfast=ptable[,10],
  amfastc=ptable[,11])
print(axtym)

##       n    ax   axc   axx  axxc axftn axftnc  aximp aximpc axmfast amfastc
```

```
## 1   100 0.000 0.000 0.004 0.000 0.000  0.000  0.328  0.112   0.028   0.024
## 2   200 0.000 0.000 0.004 0.004 0.000  0.004  1.308  0.440   0.064   0.056
## 3   300 0.004 0.004 0.000 0.000 0.004  0.004  2.948  1.032   0.108   0.100
## 4   400 0.008 0.004 0.004 0.004 0.008  0.008  5.196  1.792   0.156   0.148
## 5   500 0.012 0.008 0.008 0.004 0.012  0.012  8.060  2.788   0.212   0.204
## 6   600 0.016 0.012 0.008 0.008 0.012  0.012 11.640  3.976   0.276   0.260
## 7   700 0.024 0.020 0.012 0.012 0.020  0.016 15.784  5.464   0.348   0.328
## 8   800 0.028 0.028 0.016 0.016 0.024  0.020 20.656  7.072   0.420   0.400
## 9   900 0.048 0.048 0.024 0.028 0.028  0.028 26.124  9.016   0.512   0.528
## 10 1000 0.060 0.064 0.036 0.036 0.036  0.036 32.124 11.104   0.596   0.608
```

From the above output, we see that the `crossprod` variant of the matrix-vector product appears to be the fastest. However, we have omitted the time to build the matrix. If we must build the matrix, then we need somehow to include that time. Because the times for the matrix-vector product were so short, we used `replicate` above to run 20 copies of the same calculation, which may give some distortion of the timings. However, we believe the scale of the times is more or less correct. To compare these times to the times for the Fortran or implicit matrix-vector routines, we should add a multiple of the relevant interpreted or compiled build times. Here we have used the times for the rather poor `molermat()` function, but this is simply to illustrate the range of potential timings. Apportioning such "fixed costs" to timings is never a trivial decision. Similarly if, where and how to store large matrices if we do build them, and whether it is worth building them more than once if storage is an issue, are all questions that may need to be addressed if performance becomes important.

```
bmattym <- bmattym[1:10,]
adjtym<-data.frame(n=axtym$n, axx1=axtym$axx+1*bmattym$buildi,
     axxz=axtym$axx+20*bmattym$buildi,
     axxc1=axtym$axxc+1*bmattym$buildc,axxcz=axtym$axxc+20*bmattym$buildc,
     axftn=axtym$axftn, aximp=axtym$aximp, aximpc=axtym$aximpc)
print(adjtym)

##       n  axx1   axxz axxc1  axxcz axftn  aximp aximpc
## 1   100 0.020  0.324 0.004  0.080 0.000  0.328  0.112
## 2   200 0.084  1.604 0.028  0.484 0.000  1.308  0.440
## 3   300 0.176  3.520 0.056  1.120 0.004  2.948  1.032
## 4   400 0.312  6.164 0.108  2.084 0.008  5.196  1.792
## 5   500 0.488  9.608 0.160  3.124 0.012  8.060  2.788
## 6   600 0.736 14.568 0.236  4.568 0.012 11.640  3.976
## 7   700 0.948 18.732 0.320  6.172 0.020 15.784  5.464
## 8   800 1.280 25.296 0.412  7.936 0.024 20.656  7.072
## 9   900 1.580 31.144 0.536 10.188 0.028 26.124  9.016
## 10 1000 1.956 38.436 0.656 12.436 0.036 32.124 11.104
```

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

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

6 RQ computation times

We have in Section 4 above 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 the implicit matrix-vector product `axmolerfast`.

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

```
require(compiler)
rqdirc<-cmpfun(rqdir)
ray1c<-cmpfun(ray1)
ray2c<-cmpfun(ray2)
ray3c<-cmpfun(ray3)
dyn.load("moler.so")
  n<-500
  x<-runif(n) # generate a vector
  AA<-molermat(n)
  tdi<-system.time(rdi<-replicate(20,rqdir(x, AA))[1])[[1]]
  tdc<-system.time(replicate(20,rdc<-rqdirc(x, AA))[1])[[1]]
  cat("Direct algorithm: interpreted=",tdi,"   byte-compiled=",tdc,"\n")

## Direct algorithm: interpreted= 7.368    byte-compiled= 0.8

  t1i<-system.time(replicate(20,r1i<-ray1(x, AA))[1])[[1]]
  t1c<-system.time(replicate(20,r1c<-ray1c(x, AA))[1])[[1]]
  cat("ray1: mat-mult algorithm: interpreted=",t1i,"   byte-compiled=",t1c,"\n")

## ray1: mat-mult algorithm: interpreted= 0.024    byte-compiled= 0.024

  t2i<-system.time(replicate(20,r2i<-ray2(x, AA))[1])[[1]]
  t2c<-system.time(replicate(20,r2c<-ray2c(x, AA))[1])[[1]]
  cat("ray2: crossprod algorithm: interpreted=",t2i,"   byte-compiled=",t2c,"\n")

## ray2: crossprod algorithm: interpreted= 0.008    byte-compiled= 0.008

  t3fi<-system.time(replicate(20,r3i<-ray3(x, AA, ax=axftn))[1])[[1]]
  t3fc<-system.time(replicate(20,r3i<-ray3c(x, AA, ax=axftnc))[1])[[1]]
  cat("ray3: ax Fortran + crossprod: interpreted=",t3fi,"   byte-compiled=",t3fc,"\n")

## ray3: ax Fortran + crossprod: interpreted= 0.012    byte-compiled= 0.008

  t3ri<-system.time(replicate(20,r3i<-ray3(x, AA, ax=axmolerfast))[1])[[1]]
  t3rc<-system.time(replicate(20,r3i<-ray3c(x, AA, ax=axmfc))[1])[[1]]
  cat("ray3: ax fast R implicit + crossprod: interpreted=",t3ri,"   byte-compiled=",t3rc,"\n")

## ray3: ax fast R implicit + crossprod: interpreted= 0.216    byte-compiled= 0.2
```

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

7 Solution by spg

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

```
testsol <- function(A,eval, evec){ # test a trial eigensolution
     cmpval <- length(evec)*1e-5
     tvec <- A %*% evec - eval*evec
     etest <- max(abs(tvec))
     ntest <- abs(1-as.numeric(crossprod(evec)))
     cat("Eigenvalue solution test:", etest ,"   Normtest:", ntest ,"\n")
     if ((ntest > cmpval) || (etest > cmpval)) "FAIL" else "OK"
}
rqt<-function(x, AA){
    rq<-as.numeric(crossprod(x, crossprod(AA,x)))/as.numeric(crossprod(x))
}
proj<-function(x) { x/sqrt(crossprod(x)) }
require(BB)

## Loading required package:  BB
```

```
n<-100
x<-rep(1,n)
AA<-molermat(n)
tevs <- system.time(evs<-eigen(AA))[[1]]
cat("Time to compute full eigensystem = ", tevs, "\n")

## Time to compute full eigensystem =  0.004

tmin<-system.time(amin<-spg(x, fn=rqt, project=proj, control=list(trace=FALSE), AA=AA))[[1]]
#amin
tmax<-system.time(amax<-spg(x, fn=rqt, project=proj, control=list(trace=FALSE), AA=-AA))[[1]]
#amax
evalmax<-evs$values[1]
evecmax<-evs$vectors[,1]
evecmax<-sign(evecmax[1])*evecmax/sqrt(as.numeric(crossprod(evecmax)))
emax<-list(evalmax=evalmax, 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= 5.072545e-08 in time  0.772

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

## Eigenvalue - result from eigen= 5.072498e-08   vector max(abs(diff))= 0.0001255269

print(testsol(AA, amin$value, avecmin))

## Eigenvalue solution test: 0.0003092015    Normtest: 0
## [1] "OK"

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

## maximal eigensolution: Value= 3934.277 in time  0.064

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

## Eigenvalue - result from eigen= -1.928129e-10   vector max(abs(diff))= 5.353476e-08

print(testsol(AA, -amax$value, avecmax))

## Eigenvalue solution test: 0.0002115552    Normtest: 1.110223e-16
## [1] "OK"
```

```
##      n spgrqt spgcrqtcaxc tbldc
## 1   50  0.048       0.048 0.004
## 2  100  0.060       0.060 0.008
## 3  150  0.096       0.096 0.016
## 4  200  0.184       0.188 0.028
## 5  250  0.248       0.264 0.040
## 6  300  0.488       0.516 0.056
## 7  350  0.812       0.860 0.080
## 8  400  1.248       1.276 0.100
## 9  450  1.916       1.944 0.128
## 10 500  2.220       2.320 0.156
```

8 Solution by other optimizers

We can try other optimizers, but we must note that unlike **spg** they do not take account explicitly of the scaling. However, we can build in a transformation, since our function is always the same for all sets of

parameters scaled by the square root of the parameter inner product. The function `nobj` forms the quadratic form that is the numerator of the Rayleigh Quotient using the more efficient `crossprod()` function

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

but we first form

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

to scale the parameters.

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

```
nobj<-function(x, AA=-AA){
   y<-x/sqrt(as.numeric(crossprod(x))) # prescale
   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(optimrx)

## Loading required package:  optimrx

# mset<-c("L-BFGS-B", "BFGS", "CG", "spg", "ucminf", "nlm", "nlminb", "Rvmmin", "Rcgmin")
mset<-"Rcgmin"
nmax<-5
for (ni in 1:nmax){
  n<-20*ni
  x<-runif(n) # generate a vector
#  AA<-molerc(n) # make sure defined
  AA<-molermat(n)
  aall<-opm(x, fn=nobj, gr=ngrobj, method=mset, AA=-AA,
     control=list(starttests=FALSE,  dowarn=FALSE))
  print(summary(aall, order=value, par.select=1:3))
  cat("Above for n=",n," \n")
}

##               p1          p2       p3     value fevals gevals convergence
## Rcgmin -1.297314 0.009341281 1.315929 -140.8991     13      8           0
##        kkt1  kkt2 xtime
## Rcgmin TRUE FALSE 0.004
## Above for n= 20
##                 p1           p2         p3     value fevals gevals
## Rcgmin -0.04639766 7.718822e-05 0.04655189 -602.8685     13      6
##        convergence kkt1  kkt2 xtime
## Rcgmin           0 TRUE FALSE     0
## Above for n= 40
##                 p1           p2         p3     value fevals gevals
## Rcgmin -0.0336468 2.420475e-05 0.03369522 -1389.103     13      7
##        convergence kkt1  kkt2 xtime
## Rcgmin           0 TRUE FALSE     0
## Above for n= 60
##                 p1          p2         p3     value fevals gevals
## Rcgmin -0.0293178 1.17105e-05 0.02934125 -2499.575     13      7
##        convergence kkt1  kkt2 xtime
## Rcgmin           0 TRUE FALSE 0.004
## Above for n= 80
##                 p1           p2         p3     value fevals gevals
## Rcgmin -0.02843895 7.149832e-06 0.02845329 -3934.277     13      7
##        convergence kkt1  kkt2 xtime
## Rcgmin           0 TRUE FALSE     0
## Above for n= 100
```

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

latter (in its unconstrained implementation) to show the times over different matrix sizes.

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

##       n tRcgminu
## 1   50    0.004
## 2  100    0.004
## 3  150    0.008
## 4  200    0.004
## 5  250    0.008
## 6  300    0.020
## 7  350    0.020
## 8  400    0.028
## 9  450    0.040
## 10 500    0.044
```

9 A specialized minimizer - Geradin's method

For comparison, let us try the (Geradin 1971) routine (Appendix 1) as implemented in R by one of us (JN). This is a specialized conjugate gradient minimization routine for eigenvalue problems.

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

## Test geradin with explicit matrix multiplication

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

## Minimal eigensolution

print(ag)

## $x
##  [1] 661073.957 330538.397 165272.745  82643.819  41336.981  20698.722
##  [7]  10409.868   5325.971   2905.070   1936.711
##
## $RQ
## [1] 8.582807e-06
##
## $ipr
## [1] 41
##
## $msg
## [1] "Small gradient -- done"

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

## Geradin time= 0

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

```
## Maximal eigensolution (negative matrix)

print(agn)

## $x
##  [1]  133306672125   -4516928172 -142196879938 -275067864295 -398651235141
##  [6] -508772176341 -601691762048 -674291356801 -724089458284 -749418152832
##
## $RQ
## [1] -31.58981
##
## $ipr
## [1] 36
##
## $msg
## [1] "Small gradient -- done"

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

## Geradin time= 0.004
```

Let us time this routine with different matrix vector approaches.

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

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

## Is the mat multiply loaded?  TRUE

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

require(compiler)

## Loading required package:  compiler

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

# require(microbenchmark)
nmax<-10
gtable<-matrix(NA, nrow=nmax, ncol=6) # to hold results
# loop over sizes
for (ni in 1:nmax){
  n<-50*ni
  x<-runif(n) # generate a vector
  gtable[[ni, 1]]<-n
  AA<-molermat(n)
  BB<-diag(rep(1,n))
  tgax<-system.time(ogax<-geradin(x, ax, bx, AA=-AA, BB=BB, control=list(trace=FALSE)))[[1]]
  gtable[[ni, 2]]<-tgax
  tgaximp<-system.time(ogaximp<-geradin(x, naximp, ident, AA=1, BB=1, control=list(trace=FALSE)))[[1]]
  gtable[[ni, 3]]<-tgaximp
  tgaximpc<-system.time(ogaximpc<-geradin(x, naximpc, ident, AA=1, BB=1, control=list(trace=FALSE)))[[1]]
  gtable[[ni, 4]]<-tgaximpc
  tgaxftn<-system.time(ogaxftn<-geradin(x, naxftn, ident, AA=1, BB=1, control=list(trace=FALSE)))[[1]]
  gtable[[ni, 5]]<-tgaxftn
  tgaxftnc<-system.time(ogaxftnc<-geradin(x, naxftnc, ident, AA=1, BB=1, control=list(trace=FALSE)))[[1]]
```

```
  gtable[[ni, 6]]<-tgaxftnc
#  cat(n,tgax, tgaximp, tgaximpc, tgaxftn, tgaxftnc,"\n")
}

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

```
##       n    ax  aximp aximpc axftn axftnc
## 1    50 0.004  0.148  0.048 0.004  0.000
## 2   100 0.004  0.564  0.192 0.000  0.004
## 3   150 0.004  1.276  0.416 0.004  0.004
## 4   200 0.008  2.252  0.796 0.004  0.004
## 5   250 0.012  3.508  1.160 0.004  0.008
## 6   300 0.012  5.200  1.728 0.008  0.008
## 7   350 0.024  7.376  2.504 0.012  0.012
## 8   400 0.024  9.376  3.160 0.012  0.012
## 9   450 0.032 11.656  3.896 0.016  0.012
## 10 500 0.044 14.932  5.040 0.020  0.016
```

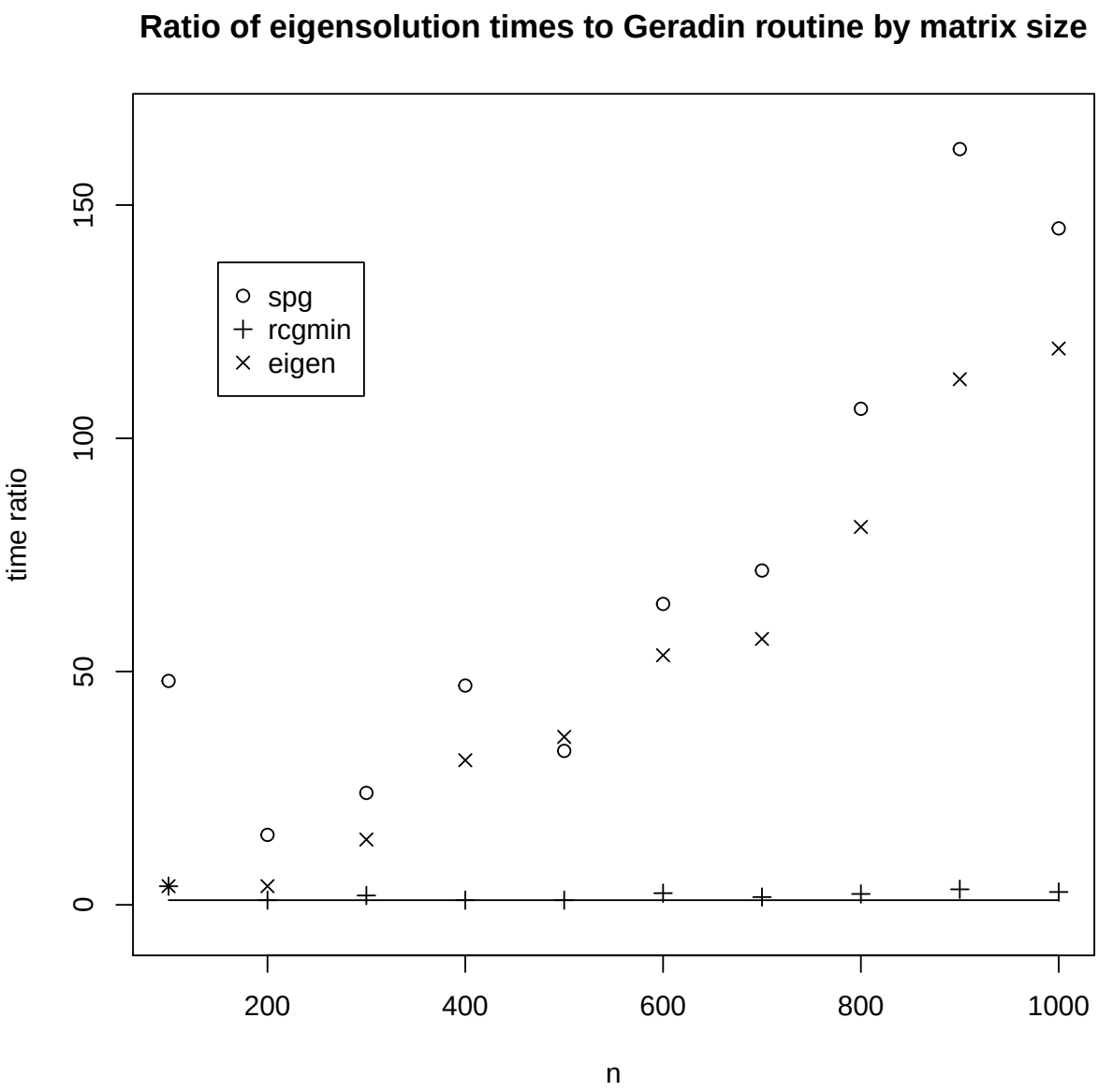
Let us check that the solution for `n = 100` by Geradin is consistent with the answer via `eigen()`.

```
n<-100
x<-runif(n)
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:  -2.145998e-08    max(abs(vector diff))= 4.536945e-07
```

10 Perspective

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



Note that the conjugate gradients method is about as fast as the Geradin specialized method, so the latter may be overkill. The eigenvalue solver is finding all solutions, so it uses more time as the size of the problems increase. However, finding all solutions this way is still faster than finding just one with the `spg()` function. Nevertheless, the latter may still be worth keeping in our toolbox as its projection approach is sometimes much quicker to set up and test. Human time is more valuable than computer time in most situations.

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

```
## Times in seconds
## Build = 2.78  eigen(): 15.944   Rcgminu: 0.676  Geradin: 0.244
## Ratios: build= 11.39344  eigen= 65.34426   Rcgminu= 2.770492
```

11 Conclusions

The Rayleigh Quotient minimization approach to eigensolutions has an intuitive appeal and seemingly offers an interesting optimization test problem, especially if we can make it computationally efficient. To improve time efficiency, we can apply the R byte code compiler, use a Fortran (or other compiled language) subroutine, and choose how we set up our objective functions and gradients. To improve memory use, we can consider using a matrix implicitly.

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

- The R byte code compiler offers a useful gain in speed when our code has statements that access array elements rather than uses them in vectorized form.
- The `crossprod()` function is very efficient.
- Fortran is not very difficult to use for small subroutines that compute a function such as the implicit matrix-vector product, and it allows efficient computations for such operations.
- The `eigen()` routine is a highly effective tool for computing all eigensolutions, even of a large matrix. It does, however, require the explicit full 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. If such a specialized code is unavailable, a general conjugate gradients code can be quite competitive for minimizing the Rayleigh Quotient when the other speed improvements are applied. Both approaches save memory when an implicit matrix-vector product is used.

Acknowledgements

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

References

Geradin, M. (1971). The computational efficiency of a new minimization algorithm for eigenvalue analysis. *J. Sound Vib. 19*, 319–331.

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

Appendix 1: Geradin routine

```
ax<-function(x, AA){
   u<-as.numeric(AA%*%x)
}
bx<-function(x, BB){
   v<-as.numeric(BB%*%x)
}
geradin<-function(x, ax, bx, AA, BB, control=list(trace=TRUE, maxit=1000)){
# Geradin minimize Rayleigh Quotient, Nash CMN Alg 25
# print(control)
  trace<-control$trace
  n<-length(x)
  tol<-n*n*.Machine$double.eps^2
```

```
  offset<-1e+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
}
```