Timing Rayleigh Quotient minimization in R

by John C. Nash

Abstract This vignette is simply to record the methods and results for timing various Rayleigh Quotient minimizations with R using different functions and different ways of running the computations, in particular trying Fortran subroutines and the R byte compiler. It has been updated from a 2012 document to reflect changes in R and its packages that make it awkward to reprocess the original document on newer computers.

The computational task

The maximal and minimal eigensolutions of a symmetric matrix A are extrema of the Rayleigh Quotient

$$R(x) = (x'Ax)/(x'x)$$

We could also deal with generalized eigenproblems of the form

$$Ax = eBx$$

where B is symmetric and positive definite by using the Rayleigh Quotient (RQ)

$$R_g(x) = (x'Ax)/(x'Bx)$$

In this document, B will always be an identity matrix, but some programs we test assume that it is present.

Note that the objective is scaled by the parameters, in fact by by their sum of squares. Alternatively, we may think of requiring the **normalized** eigensolution, which is given as

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

Timings and speedups

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

There are some ways to speed up R computations.

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

Our example matrix

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

Here is the code for generating the Moler matrix.

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

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

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

Time to build the matrix

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

```
#> Loading required package: microbenchmark

#>      n buildi   osize eigentime bfast
#> 1   50   2869   20216       887   789
#> 2  100   6077   80216      2061   615
#> 3  150  10268  180216      4191  1124
#> 4  200  16498  320216      5611  1358
#> 5  250  25040  500216      8564  1862
#> 6  300  35453  720216     11370  2430
#> 7  350  47422  980216     14715  3029
#> 8  400  60907 1280216     19330  3801
#> 9  450  79044 1620216     25121  4342
#> 10 500  93532 2000216     30489  7914

#> buildi - interpreted build time

#> osize - matrix size in bytes; eigentime - all eigensolutions time

#> bfast - interpreted vectorized build time

#> Times converted to milliseconds
```

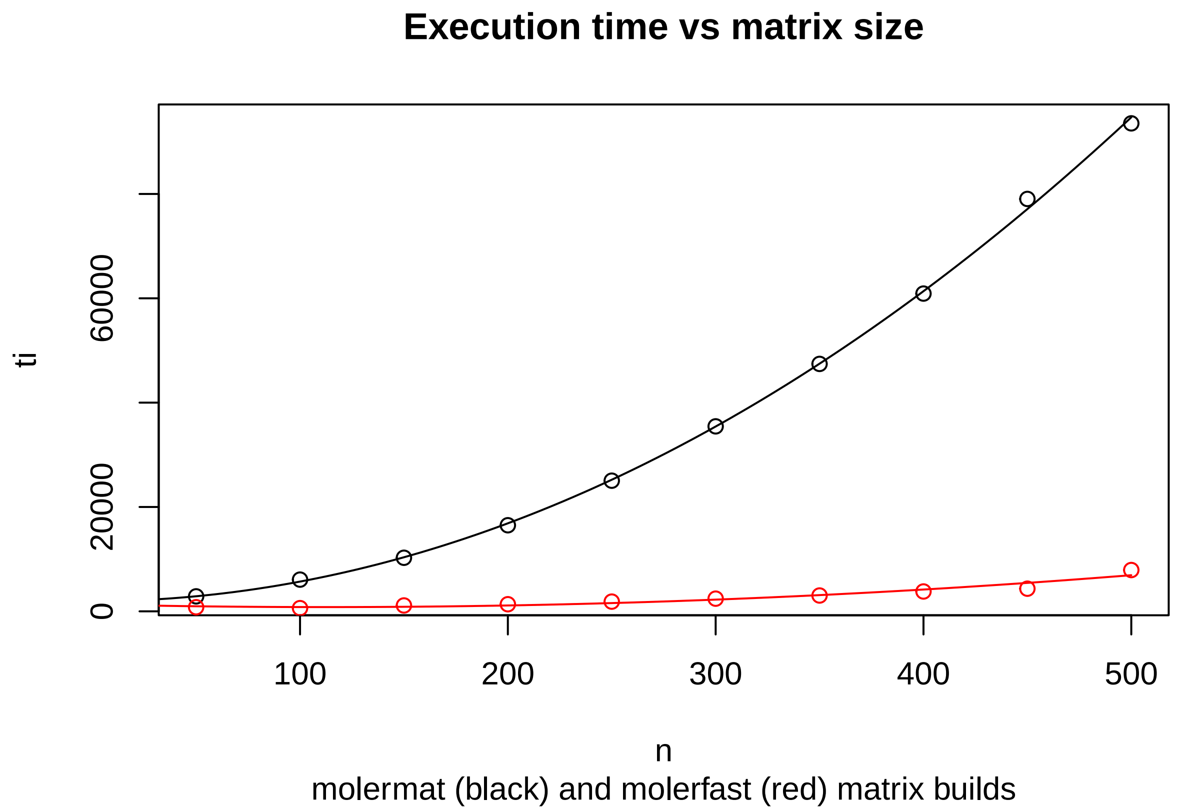
It does not appear that the compiler has much effect, or else it is being automatically invoked.

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

```
#>
#> Call:
#> lm(formula = ti ~ n + n2)
```

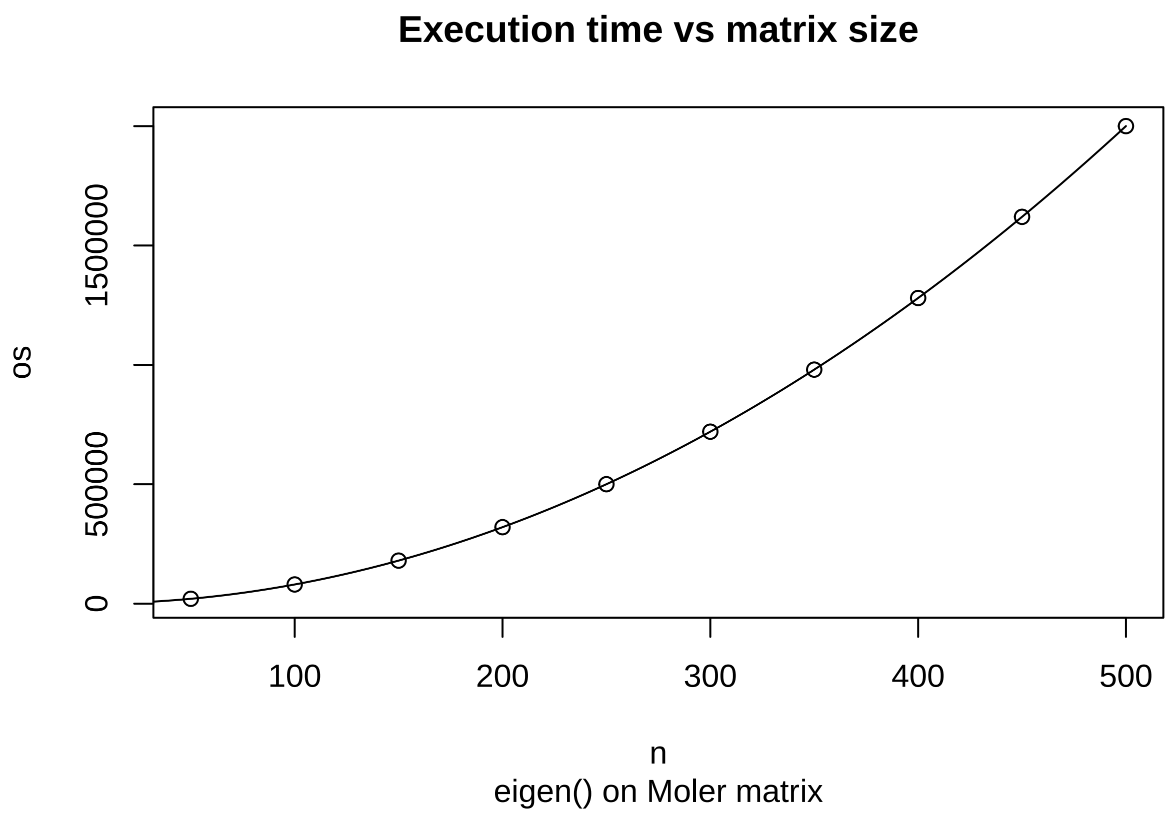
```
#>
#> Residuals:
#>      Min       1Q   Median       3Q      Max
#> -1161.39  -325.49   -72.59    16.32  1937.35
#>
#> Coefficients:
#>              Estimate Std. Error t value Pr(>|t|)
#> (Intercept) 1.919e+03  1.056e+03   1.818    0.112
#> n           8.919e-01  8.818e+00   0.101    0.922
#> n2          3.693e-01  1.562e-02  23.637 6.16e-08 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 897.6 on 7 degrees of freedom
#> Multiple R-squared:  0.9994, Adjusted R-squared:  0.9992
#> F-statistic:  5607 on 2 and 7 DF,  p-value: 6.063e-12

#>
#> Call:
#> lm(formula = tf ~ n + n2)
#>
#> Residuals:
#>      Min       1Q   Median       3Q      Max
#> -1107.88  -197.61    54.16   250.24   988.21
#>
#> Coefficients:
#>               Estimate Std. Error t value Pr(>|t|)
#> (Intercept) 1326.82833  725.00458   1.830  0.10993
#> n             -9.16263    6.05585  -1.513  0.17404
#> n2             0.04072    0.01073   3.795  0.00676 **
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 616.4 on 7 degrees of freedom
#> Multiple R-squared:  0.9399, Adjusted R-squared:  0.9227
#> F-statistic: 54.73 on 2 and 7 DF,  p-value: 5.324e-05
```



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

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



Computing the Rayleigh Quotient

The Rayleigh Quotient requires the quadratic form $x'Ax$ divided by the inner product $x'x$. R lets us form this in several ways.

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

```
  }
  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)))
}
```

1 References

Bibliography

J. C. Nash. *Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation*. Adam Hilger, Bristol, 1979. Second Edition, 1990, Bristol: Institute of Physics Publications. [p1]

John C. Nash
retired professor, University of Ottawa
Telfer School of Management
Ottawa ON Canada K1N 6N5
ORCiD: 0000-0002-2762-8039
profjcnash@gmail.com