

# Comparing Least Squares Calculations

Douglas Bates  
R Development Core Team  
Douglas.Bates@R-project.org

December 28, 2006

## Abstract

Many statistics methods require one or more least squares problems to be solved. There are several ways to perform this calculation, using objects from the base R system and using objects in the classes defined in the `Matrix` package.

We compare the speed of some of these methods on a very small example and on a example for which the model matrix is large and sparse.

## 1 Linear least squares calculations

Many statistical techniques require least squares solutions

$$\hat{\beta} = \arg \min_{\beta} \|\mathbf{y} - \mathbf{X}\beta\|^2 \quad (1)$$

where  $\mathbf{X}$  is an  $n \times p$  model matrix ( $p \leq n$ ),  $\mathbf{y}$  is  $n$ -dimensional and  $\beta$  is  $p$  dimensional. Most statistics texts state that the solution to (1) is

$$\hat{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} \quad (2)$$

when  $\mathbf{X}$  has full column rank (i.e. the columns of  $\mathbf{X}$  are linearly independent) and all too frequently it is calculated in exactly this way.

### 1.1 A small example

As an example, let's create a model matrix, `mm`, and corresponding response vector, `y`, for a simple linear regression model using the `Formaldehyde` data.

```
> data(Formaldehyde)
> str(Formaldehyde)
```

```
'data.frame':      6 obs. of  2 variables:
 $ carb   : num  0.1 0.3 0.5 0.6 0.7 0.9
 $ optden : num  0.086 0.269 0.446 0.538 0.626 0.782
```

```
> (m <- cbind(1, Formaldehyde$carb))
```

```
      [,1] [,2]
[1,]    1 0.1
[2,]    1 0.3
[3,]    1 0.5
[4,]    1 0.6
[5,]    1 0.7
[6,]    1 0.9
```

```
> (yo <- Formaldehyde$optden)
```

```
[1] 0.086 0.269 0.446 0.538 0.626 0.782
```

Using `t` to evaluate the transpose, `solve` to take an inverse, and the `%%` operator for matrix multiplication, we can translate 2 into the S language as

```
> solve(t(m) %% m) %% t(m) %% yo
```

```
      [,1]
[1,] 0.005085714
[2,] 0.876285714
```

On modern computers this calculation is performed so quickly that it cannot be timed accurately in R <sup>1</sup>

```
> system.time(solve(t(m) %% m) %% t(m) %% yo)
```

```
[1] 0 0 0 0 0
```

and it provides essentially the same results as the standard `lm.fit` function that is called by `lm`.

```
> dput(c(solve(t(m) %% m) %% t(m) %% yo))
```

```
c(0.00508571428571428, 0.876285714285715)
```

```
> dput(unname(lm.fit(m, yo)$coefficients))
```

```
c(0.00508571428571408, 0.876285714285715)
```

## 1.2 A large example

For a large, ill-conditioned least squares problem, such as that described in Koenker and Ng (2003), the literal translation of (2) does not perform well.

---

<sup>1</sup>From R version 2.2.0, `system.time()` has default argument `gcFirst = TRUE` which is assumed and relevant for all subsequent timings

```

> library(Matrix)
> data(KNex, package = "Matrix")
> y <- KNex$y
> mm <- as(KNex$mm, "matrix")
> dim(mm)

[1] 1850  712

> system.time(naive.sol <- solve(t(mm) %*% mm) %*% t(mm) %*%
+   y)

[1] 1.637 0.050 1.687 0.000 0.000

```

Because the calculation of a “cross-product” matrix, such as  $\mathbf{X}^\top \mathbf{X}$  or  $\mathbf{X}^\top \mathbf{y}$ , is a common operation in statistics, the `crossprod` function has been provided to do this efficiently. In the single argument form `crossprod(mm)` calculates  $\mathbf{X}^\top \mathbf{X}$ , taking advantage of the symmetry of the product. That is, instead of calculating the  $712^2 = 506944$  elements of  $\mathbf{X}^\top \mathbf{X}$  separately, it only calculates the  $(712 \cdot 713)/2 = 253828$  elements in the upper triangle and replicates them in the lower triangle. Furthermore, there is no need to calculate the inverse of a matrix explicitly when solving a linear system of equations. When the two argument form of the `solve` function is used the linear system

$$(\mathbf{X}^\top \mathbf{X}) \hat{\boldsymbol{\beta}} = \mathbf{X}^\top \mathbf{y} \quad (3)$$

is solved directly.

Combining these optimizations we obtain

```

> system.time(cpod.sol <- solve(crossprod(mm), crossprod(mm,
+   y)))

[1] 1.586 0.011 1.598 0.000 0.000

> all.equal(naive.sol, cpod.sol)

[1] TRUE

```

On this computer (2.0 GHz Pentium-4, 1 GB Memory, Goto’s BLAS) the `crossprod` form of the calculation is about four times as fast as the naive calculation. In fact, the entire `crossprod` solution is faster than simply calculating  $\mathbf{X}^\top \mathbf{X}$  the naive way.

```

> system.time(t(mm) %*% mm)

[1] 0.086 0.011 0.098 0.000 0.000

```

### 1.3 Least squares calculations with Matrix classes

The `crossprod` function applied to a single matrix takes advantage of symmetry when calculating the product but does not retain the information that the product is symmetric (and positive semidefinite). As a result the solution of (3) is performed using general linear system solver based on an LU decomposition when it would be faster, and more stable numerically, to use a Cholesky decomposition. The Cholesky decomposition could be used but it is rather awkward

```
> system.time(ch <- chol(crossprod(mm)))
[1] 1.632 0.004 1.636 0.000 0.000

> system.time(chol.sol <- backsolve(ch, forwardsolve(ch, crossprod(mm,
+      y), upper = TRUE, trans = TRUE)))
[1] 0.018 0.003 0.021 0.000 0.000

> all.equal(chol.sol, naive.sol)
[1] TRUE
```

The `Matrix` package uses the S4 class system (Chambers, 1998) to retain information on the structure of matrices from the intermediate calculations. A general matrix in dense storage, created by the `Matrix` function, has class `"dgeMatrix"` but its cross-product has class `"dpoMatrix"`. The `solve` methods for the `"dpoMatrix"` class use the Cholesky decomposition.

```
> mm <- as(KNex$mm, "dgeMatrix")
> class(crossprod(mm))
[1] "dpoMatrix"
attr(,"package")
[1] "Matrix"

> system.time(Mat.sol <- solve(crossprod(mm), crossprod(mm,
+      y)))
[1] 1.63 0.01 1.64 0.00 0.00

> all.equal(naive.sol, unname(as(Mat.sol, "matrix")))
[1] TRUE
```

Furthermore, any method that calculates a decomposition or factorization stores the resulting factorization with the original object so that it can be reused without recalculation.

```
> xpx <- crossprod(mm)
> xpy <- crossprod(mm, y)
> system.time(solve(xpx, xpy))
```

```
[1] 0.112 0.006 0.118 0.000 0.000
```

```
> system.time(solve(xpx, xpy))
```

```
[1] 0.002 0.000 0.002 0.000 0.000
```

The model matrix `mm` is sparse; that is, most of the elements of `mm` are zero. The `Matrix` package incorporates special methods for sparse matrices, which produce the fastest results of all.

```
> mm <- KNex$mm
```

```
> class(mm)
```

```
[1] "dgCMatrix"
```

```
attr(,"package")
```

```
[1] "Matrix"
```

```
> system.time(sparse.sol <- solve(crossprod(mm), crossprod(mm,
+ y)))
```

```
[1] 0.007 0.000 0.007 0.000 0.000
```

```
> all.equal(naive.sol, as(sparse.sol, "matrix"))
```

```
[1] "Attributes: < Length mismatch: comparison on first 1 components >"
```

As with other classes in the `Matrix` package, the `dsCMatrix` retains any factorization that has been calculated although, in this case, the decomposition is so fast that it is difficult to determine the difference in the solution times.

```
> xpx <- crossprod(mm)
```

```
> xpy <- crossprod(mm, y)
```

```
> system.time(solve(xpx, xpy))
```

```
[1] 0.002 0.000 0.002 0.000 0.000
```

```
> system.time(solve(xpx, xpy))
```

```
[1] 0 0 0 0 0
```

## References

John M. Chambers. *Programming with Data*. Springer, New York, 1998. ISBN 0-387-98503-4.

Roger Koenker and Pin Ng. SparseM: A sparse matrix package for R. *J. of Statistical Software*, 8(6), 2003.