Two Styles of Programming: Educational vs. Efficient Temptations of Matrix Languages

Wolfgang M. Hartmann

1 Example of Educational Style

With the advent of matrix languages we find two kinds of formulations of algorithms in statistical software:

educational almost like textbook: very clear and easy to understand; but not efficient in terms of computer resources (time and memory).

efficient fast code designed for large practical problems; but not easy to understand.

In a recent submission to JSS ([1]) we found the following piece of SAS/IML®code which is a beautiful example of the educational programming style:

```
Y=Y-REPEAT(Y[+,]/N,N,1);       /* CORRECT Y BY MEAN           */
X=X-REPEAT(X[+,]/N,N,1);       /* CORRECT X BY MEAN           */
H=X*INV(X‘*X)*X‘;              /* H matrix                    */
YP=Y[+,]/N;                    /* MEANS OF COLUMNS OF Y       */
XPY=X‘*Y;                      /* CROSSPRODUCTS               */
YPY=Y‘*Y;                      /*                             */
XPX=X‘*X;                      /*                             */
SSE= Y‘*(I(N)-H)*Y;            /* Sum sq. of err. for all x’s */
SST= YPY-(N*YP‘*YP);           /* Total sum square of error   */
CONST= det(SSTO);              /* Determinant of SSTO matrix  */
CSAVE=(XPX || XPY) //          /*                             */
      (XPY‘|| YPY);            /* SAVED COPY OF CROSSPRODUCTS */
```

When **YP** is computed, **Y** is already centered at the column mean, which of course means that **YP** is zero and the $ny \times ny$ matrix **SST** is the same as **YPY**. Such statements are clearly only of educational purpose!

As we will show later, this code is easy for understanding but not very efficient for large scale computing. But, first some remarks about the notation. Here, **X** and **Y** are $N \times nx$ and $N \times ny$ matrices (data sets). Since the `inv(X‘*X)` is used, it is implicitly assumed that $N > nx$. An important practical application would be when both parts **X** and **Y** are *tall and narrow* matrices, i.e. $N >> max(nx, ny)$.

For simplicity we count flops (floating point operation) in the old way:

1 flop = 1 multiplication + 1 addition

That means, like in the first edition of [2].

2 Computing the H and SSE Matrices: Educationally

The $N \times N$ matrix **H** is computed for the $ny \times ny$ *error-sum-of-squares* matrix SSE:

```
H = X*INV(X‘*X)*X‘;              /* H matrix                   */
SSE = Y‘*(I(N)-H)*Y;             /* Sum sq. of err. for all x’s */
```

For large N, **H** is huge and a standard matrix language will parse the expression from left to right without respecting the symmetry of the formula. The following steps would be performed:

1. the large **X** is pulled on stack[1].
2. the large **X**’ pulled on stack[2] together with another copy of **X** on stack[3].
3. matrix multiplication is performed between stack [2] and [3] resulting in a small $nx \times nx$ matrix which is stored on stack[4]. Hopefully, stack [2] and [3] are popped from stack after the multiplication and the result is moved from stack [4] to [2]. number of flops: $N * nx^2$
4. the inverse of the small $nx \times nx$ matrix on stack [2] is computed (hopefully it is not singular). Symmetry would not be used (not even known) by SAS/IML®. number of flops: for LU dec: $nx^3/3 + O(nx^2)$, see [2], page 97.
5. matrix multiplication between stack [1] and [2], with large $N \times nx$ result on stack [3]. Hopefully, stack [1] and [2] are popped from stack after the multiplication and the $N \times nx$ result is moved from stack [3] to [1]. number of flops: $N * nx^2$
6. the $nx \times N$ matrix **X**’ is pulled on stack [2].

7. matrix multiplication between stack [1] and [2], with huge $N \times N$ result on stack [3]. Usually, it will not be known that the result is symmetric. Hopefully, stack [1] and [2] are popped from stack after the multiplication and the $N \times N$ result is moved from stack [3] to [1]. number of flops: $N * nx^2$

8. the result on stack [1] is moved to the variable table of **H**.

Computer Resources:

- The total number of flops is: $3 * N * nx^2 + nx^3/3 + O(nx^2)$
- The largest chunk of memory is needed in the last step: two $N \times nx$ matrices and the $N \times N$ result are on stack!

For computing the $ny \times ny$ matrix **SSE** the following steps are performed:

1. the $ny \times N$ matrix is pulled on stack [1].
2. the $N \times N$ identity matrix **I(N)** and the $N \times N$ matrix **H** are pulled on stack [2] and [3].
3. the difference between stack [2] and stack [3] is computed (hopefully by respecting the diagonal shape of the matrix on stack [2]) and stored as $N \times N$ result on stack [4]. Stack [2] and [3] are popped from stack after the operation and the $N \times N$ result is moved from stack [4] to [2]. number of flops: can be neglected.
4. matrix multiplication between stack [1] and [2], with large $ny \times N$ result on stack [3]. Hopefully, stack [1] and [2] are popped from stack after the multiplication and the $ny \times N$ result is moved from stack [3] to [1]. number of flops: $ny * N^2$
5. the $N \times ny$ matrix **Y** is pulled on stack [2].
6. matrix multiplication between stack [1] and [2], with small $ny \times ny$ result on stack [3]. Usually, it will not be known that the result is symmetric. Stack [1] and [2] are popped off after the multiplication and the $ny \times ny$ result is moved from stack [3] to [1]. number of flops: $N * ny^2$
7. the result on stack [1] is moved to the variable table of `SSE`.

Computer Resources:

- The total number of flops is: $ny * N^2 + N * ny^2$

- The largest chunk of memory is needed in step 3: two $N \times N$ matrices are on stack! (SAS/IML®does not find out about symmetry and will always store the full matrix. Until version 6 it will even store an identity matrix as a full square matrix of double words.)

Using the two statements for **SSE** needs the total number of flops:

$$3 * N * nx^2 + nx^3/3 + ny * N^2 + N * ny^2 + O(nx^2) + O(ny^2)$$

3 Computing of the H and SSE Matrices: Efficiently

In truly efficient computing the matrices **X** and **Y** would not be pulled on stack. A data set containing both $(\mathbf{Y}, \mathbf{X})$ would sequentially (rowwise) accessed and the three cross product matrices **YPY**, **XPX**, and **XPY** computed. Computing all three matrices needs only one sequential run through the data set $(\mathbf{Y}, \mathbf{X})$. In practical application a prior sequential run would be used for computing the column mean vector and evtl. some other moments and basic statistics (counting rows with missing values).

Symmetry of **YPY** and **XPX** can be used for saving memory and computation time since only one triangle must be computed.

1. compute the $nx \times nx$ matrix **XPX**, the $ny \times ny$ matrix **YPY**, and the $nx \times ny$ matrix **XPY**. number of flops: $N * (nx^2 + ny^2 + nx * ny)$ (and even less when using the symmetry of **XPX** and **YPY**)
2. compute the $nx \times nx$ lower triangular matrix **L** with Cholesky factorization. number of flops: $nx^3/6 + O(nx^2)$
3. use the triangular shape of **L** for backward solution with $nx \times ny$ right hand sides **XPY**, $nx \times ny$ result **E**. number of flops: $ny * nx^2/2$
4. use symmetry when computing **SSE** as cross product of the $nx \times ny$ matrix **E**. number of flops: $nx * ny^2/2$

Computer Resources:

- The total number of flops is:

  $$N*(nx^2+ny^2+nx*ny)+nx^3/6+ny*nx^2/2+nx*ny^2/2+O(nx^2)+O(ny^2)$$

  That means, by far the most computation time is spent computing the cross product matrices.

- For sequential processing of the rows of **X** and **Y** the algorithm uses only memory of the order nx^2, ny^2, and $nx * ny$ and no memory restrictions are w.r.t. N.

This would be more efficient SAS/IML®code:

```
XPX= X‘*X;                    /* CROSSPRODUCTS               */
YPY= Y‘*Y;                    /*                             */
XPY= X‘*Y;                    /*                             */
L = root(XPX);                /* triang. Cholesky factor L   */
E = inv(L’) * XPY;            /* use backward elimination    */
SST = YPY;                    /* Total sum square of error   */
SSE = YPY - E’ * E;           /* Sum sq. of err. for all x’s */
```

In practical applications it is preferred to use matrix factorization to matrix inversion (LU decomposition for unsymmetric, Cholesky for positive definite symmetric matrices or Bunch-Kaufman for indefinite symmetric matrices). An `inv` statement inside a more complex matrix expression can be very dangerous in practical applications since it does not permit to test immediately if the matrix is singular. In that case the program could have serious problems to recover this situation.

4 Additional Remarks

The educational style is continued by computing some of the matrix operations again for **XPX**, **YPY**, and **XPY**:

```
YP=Y[+,]/N;                   /* MEANS OF COLUMNS OF Y       */
XPY=X‘*Y;                     /* CROSSPRODUCTS               */
YPY=Y‘*Y;                     /*                             */
XPX=X‘*X;                     /*                             */
SSE= Y‘*(I(N)-H)*Y;           /* Sum sq. of err. for all x’s */
SST= YPY-(N*YP‘*YP);          /* Total sum square of error   */
CONST= det(SSTO);             /* Determinant of SSTO matrix  */
CSAVE=(XPX || XPY) //         /*                             */
      (XPY‘|| YPY);           /* SAVED COPY OF CROSSPRODUCTS */
```

Matrix **XPX** was already used as argument of the `inv()` function for **H**. Matrix **YPY** is implicitly used in the formula for SSE: `Y‘*I(N)*Y`.

At the start, the two matrices are centered with respect to column means:

```
Y=Y-REPEAT(Y[+,]/N,N,1);      /* CORRECT Y BY MEAN           */
X=X-REPEAT(X[+,]/N,N,1);      /* CORRECT X BY MEAN           */
```

When dealing with large data sets, the data should not be altered if it is not really necessary. Therefore, when running through the the data set $(\mathbf{Y}, \mathbf{X})$ for computing the cross product matrices, each row is centered by the prior computed mean vector before it is used in the scalar product.

5 Additional Remarks

It should be stressed here that there is nothing to say against the educational stype of programming. Many times the efficient code can be very cumbersome to read for other people interested in the subject. People reading computer programs will very much appreciate the educational programming style. The entire discussion would not be necessary if optimizers in matrix language compilers (interpreters) would be smarter and circumvent the computational problems pictured here.

References

[1] Al-Subaihi, A.A. (2002), "Variable selection in multivariable regression using IML", JSS, 2002.

[2] Golub, G., & Van Loan, C.F. (1989), *Matrix Computations*, John Hopkins University Press, 2nd ed., Baltimore, MD.