Efficient Solution of Large Fixed Effects Problems Using R

Appendix: X^{-1} Using Parallel Cholesky Decomposition

Tom Balmat, Jerome P. Reiter

December 3, 2017

Following is an algorithm to compute the inverse of a symmetric, positive-definite matrix $\boldsymbol{X}$ using its Cholesky decomposition, $\boldsymbol{R}$.[1] [2] A parallelized implementation in C is included along with instructions for configuration, compilation, and execution from within R.

Background

Let $\boldsymbol{R}$ be the $p \times p$ Cholesky decomposition of a symmetric, positive-definite $p \times p$ matrix $\boldsymbol{X}$. Then $\boldsymbol{R}'\boldsymbol{R} = \boldsymbol{X}$. To find $\boldsymbol{X}^{-1}$, we solve $\boldsymbol{X}\boldsymbol{A} = \boldsymbol{I} \Rightarrow \boldsymbol{R}'\boldsymbol{R}\boldsymbol{A} = \boldsymbol{R}'\boldsymbol{B} = \boldsymbol{I}$ for $\boldsymbol{B}$, then $\boldsymbol{R}\boldsymbol{A} = \boldsymbol{B}$ for $\boldsymbol{A}$, giving

$$\boldsymbol{R}'\boldsymbol{B} = \boldsymbol{I} \Rightarrow \begin{bmatrix} r_{11} & 0 & 0 & 0 & \\ r_{12} & r_{22} & 0 & 0 & \cdots \\ r_{13} & r_{23} & r_{33} & 0 & \\ r_{14} & r_{24} & r_{34} & r_{44} & \\ & \vdots & & & \ddots \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} & \\ b_{21} & b_{22} & b_{23} & b_{24} & \cdots \\ b_{31} & b_{32} & b_{33} & b_{34} & \\ b_{41} & b_{42} & b_{43} & b_{44} & \\ & \vdots & & & \ddots \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & \\ 0 & 1 & 0 & 0 & \cdots \\ 0 & 0 & 1 & 0 & \\ 0 & 0 & 0 & 1 & \\ & \vdots & & & \ddots \end{bmatrix}$$

$$\Rightarrow \boldsymbol{B} = \begin{bmatrix} \frac{1}{r_{11}} & 0 & 0 & 0 & \\ -\frac{r_{12}b_{11}}{r_{22}} & \frac{1}{r_{22}} & 0 & 0 & \cdots \\ -\frac{r_{13}b_{11}}{r_{33}} & -\frac{r_{23}b_{22}}{r_{33}} & \frac{1}{r_{33}} & 0 & \\ & \vdots & & \ddots & \\ -\frac{1}{r_{ii}}\sum_{k=1}^{i-1} r_{ki}b_{k1} & -\frac{1}{r_{ii}}\sum_{k=1}^{i-1} r_{ki}b_{k2} & -\frac{1}{r_{ii}}\sum_{k=1}^{i-1} r_{ki}b_{k3} & \cdots & \frac{1}{r_{ii}} \\ & \vdots & & & \ddots \end{bmatrix}.$$

[1] For more on the Cholesky decomposition, see Wikipedia (2017) and Heath (2013)

[2] For more on computing a matrix inverse using the Cholesky decomposition, see Park and Kang (2003) and Krishnamoorthy and Menon (2017)

Note the diagonal of $\boldsymbol{B}, b_{ii} = \frac{1}{r_{ii}}$. It will be seen that these are the sole elements of $\boldsymbol{B}$ required to compute $\boldsymbol{A}$. Note, also, that $\boldsymbol{B}$ is lower triangular. The utility of computing with a Cholesky decomposition arises from the triangular configuration of associated matrices.

Having an expression for $\boldsymbol{B}$, we now compute $\boldsymbol{A}$ from $\boldsymbol{RA} = \boldsymbol{B}$. Let $\boldsymbol{V} \leftarrow \boldsymbol{W}$ be a replacement operation indicating matrix $\boldsymbol{V}$ is replaced, element-wise, by corresponding elements of matrix $\boldsymbol{W}$. To simplify subsequent expressions, by eliminating the need to divide right hand sums by r_{ii} when solving for a_{ij}, let

$$\boldsymbol{R} \leftarrow \boldsymbol{CR} \text{ and } \boldsymbol{B} \leftarrow \boldsymbol{CB}, \text{ where } \boldsymbol{C} = \begin{bmatrix} \frac{1}{r_{11}} & 0 & 0 & \\ 0 & \frac{1}{r_{22}} & 0 & \cdots \\ 0 & 0 & \frac{1}{r_{33}} & \\ & \vdots & & \ddots \end{bmatrix}.$$

(Note that $\boldsymbol{A}$ is invariant, since $\boldsymbol{CRA} = \boldsymbol{CB} \Rightarrow \boldsymbol{C}^{-1}\boldsymbol{CRA} = \boldsymbol{C}^{-1}\boldsymbol{CB} \Rightarrow \boldsymbol{RA} = \boldsymbol{B}$.)

$$\text{Then } \boldsymbol{RA} = \boldsymbol{B} \Rightarrow \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} & \\ 0 & r_{22} & r_{23} & r_{24} & \cdots \\ 0 & 0 & r_{33} & r_{34} & \\ 0 & 0 & 0 & r_{44} & \\ & \vdots & & & \ddots \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & \\ a_{21} & a_{22} & a_{23} & a_{24} & \cdots \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ a_{41} & a_{42} & a_{43} & a_{44} & \\ & \vdots & & & \ddots \end{bmatrix} = \begin{bmatrix} b_{11} & 0 & 0 & 0 & \\ b_{21} & b_{22} & 0 & 0 & \cdots \\ b_{31} & b_{32} & b_{33} & 0 & \\ b_{41} & b_{42} & b_{43} & b_{44} & \\ & \vdots & & & \ddots \end{bmatrix}.$$

Now, for $j > i$, $[\boldsymbol{RA}]_{ij} = a_{ij} + \sum_{k=i+1}^{p} r_{ik}a_{kj} = b_{ij} = 0$

$$\Rightarrow a_{ij} = -\sum_{k=i+1}^{p} r_{ik}a_{kj} \tag{1}$$

a_{ij}, then, is a function of row i of $\boldsymbol{R}$, which is known, and the sub-matrix of $\boldsymbol{A}$ beyond row i and column i, that is, $\boldsymbol{A}_{st}$, such that $s, t > i$. Now, for $i = j$,

$$[\boldsymbol{RA}]_{ij} = [\boldsymbol{RA}]_{ii} = a_{ii} + \sum_{j=i+1}^{p} r_{ij}a_{ji} = b_{ii}$$

$$\Rightarrow a_{ii} = b_{ii} - \sum_{j=i+1}^{p} r_{ij}a_{ji}.$$

Note that, in the previous sum, $j > i$ but, since $\boldsymbol{A}$ is symmetric ($\mathbf{X}$ symmetric $\Rightarrow \boldsymbol{X}^{-1}$ is symmetric), a_{ji} can be substituted with a_{ij}, giving

$$a_{ii} = b_{ii} - \sum j = i + 1^{p} r_{ij}a_{ij}.$$

a_{ii}, then, is a function of b_{ii} and row i of $\boldsymbol{R}$, which are known, and the a_{ij}, computed in equation 1. Considering lower-right sub-matrices,

$$\boldsymbol{RA} = \boldsymbol{B} \Rightarrow \begin{bmatrix} \ddots & & \vdots & \\ & 1 & r_{p-2,p-1} & r_{p-2,p} \\ \cdots & 0 & 1 & r_{p-1,p} \\ & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \ddots & & \vdots & \\ & a_{p-2,p-2} & a_{p-2,p-1} & a_{p-2,p} \\ \cdots & a_{p-1,p-2} & a_{p-1,p-1} & a_{p-1,p} \\ & a_{p,p-2} & a_{p,p-1} & a_{p,p} \end{bmatrix} \begin{bmatrix} \ddots & & \vdots & \\ & b_{p-2,p-2} & 0 & 0 \\ \cdots & b_{p-1,p-2} & b_{p-1,p-1} & 0 \\ & b_{p,p-2} & b_{p,p-1} & b_{p,p} \end{bmatrix}$$

It is clear that $a_{pp} = b_{pp}$. Also, a_{pp} is the initial (lower-right, single element) sub-matrix of $\boldsymbol{A}$. With it we can compute

$$a_{p-1,p} = a_{p,p-1)} - r_{p-1,p} a_{pp}$$

which gives

$$a_{p-1,p-1} = b_{p-1,p-1} - r_{p-1,p}.$$

We now have the lower-right four element sub-matrix of $\boldsymbol{A}$ and with it we can compute

$$a_{p-2,p-1)} a_{p-1,p-2} = -\sum_{k=p-1}^{p} r_{p-2,k} a_{k,p-2},$$

$$a_{p-2,p} = a_{p,p-2} = -\sum_{k=p-1}^{p} r_{p-2,k} a_{k,p}, \text{ and}$$

$$a_{p-2,p-2} = a_{p-2,p-2} = b_{p-2,p-2} - \sum_{k=p-1}^{p} r_{p-2,k} a_{k,p-2},$$

which completes the lower-right nine element sub-matrix of $\boldsymbol{A}$. We continue completing successive lower-right matrices of $\boldsymbol{A}$ until a_{11} is computed, at which time $\boldsymbol{A} = \boldsymbol{X}^{-1}$ is complete.

Algorithm

$\boldsymbol{C} = \text{diag}_{p \times p}\left(\frac{1}{r_{11}}, \frac{1}{r_{22}}, \frac{1}{r_{33}}, \cdots, \frac{1}{r_{pp}}\right)$
$\boldsymbol{A} \leftarrow \boldsymbol{CA}$
for $i = p-1, 1$
    for $j = i+1, p$
        $a_{ij} = -\sum_{j=i+1}^{p} r_{ij} a_{ji}$
        $a_{ji} = a_{ji}$
    $a_{ii} = b_{ii} - \sum_{j=i+1}^{p} r_{ij} a_{ji}$

C Language Implementation

Following is the C language source for creating an R function to compute the inverse of a symmetric, positive-definite matrix $\boldsymbol{X}$ using its Cholesky decomposition. Objects needed for compilation are the R package `Rcpp`,[3] a C compiler, [4] and the `Open-MP` library.[5] Compilation produces file `cholInvDiag.o` (Linux) or `cholInvDiag.dll` (MS Windows) that is loaded into R using `dyn.load()`.[6] The resulting R function is `cholInvDiag()`, which requires a single parameter, $\boldsymbol{X}$, a (symmetric, positive-definite) numeric matrix and produces a numeric vector containing the diagonal entries of the inverse of $\boldsymbol{X}$ (the function presented was developed to compute the variances of parameter estimates from an OLS regression problem; the entire inverse can be returned by declaring the function to be of type `NumericMatrix` and returning the computed matrix $\boldsymbol{A}$).

Source:

```
 1 # diag of Cholesky inverse
 2 cSource <- "
 3 #include <Rcpp.h>
 4 #include <omp.h>
 5 // [[Rcpp::plugins(openmp)]]
 6 // following prefixes Rcpp type with Rcpp::
 7 using namespace Rcpp;
 8 // create array of p vector pointers, each addressing one matrix row
 9 // this compensates for the following issues when the input matrix is large
10 // local arrays (created within function) of size greater than [2047,2047] cause
11 // crash on execution
12 // static arrays of size greater than approximately [10000,10000] cause
13 // compilation errors, assembler messages are returned:  value of x too large
14 // for field of 4 bytes, possibly due to, say, 20,000 X 20,000 =
15 // 400,000,000 * 8 (double) = 3.2 billion, which is greater than unsigned long
16 // capacity (it is assumed that assembler instructions must be generated to
17 // address matrix positions offset from position 0)
18 // declaring a matrix with long long constant, as in static double
19 // R1[20000LL][20000LL]; also produces compiler 'too large' message
20 // it is assumed that the input matrix, X, is square, symmetric, and positive-
21 // definite
22 // [[Rcpp::export]]
23 NumericVector cholInvDiag(NumericMatrix R0) {
24   // use long indices for large addressing, signed since i is decremented to -1
25   long p=R0.ncol();
26   long i, j, k;
27   double s;
28   // create p-length arrays of pointers, each addressing one p-length matrix row
29   // a copy of R0 is placed in the R1 pointers, one row per pointer
30   // A is the computed inverse of the matrix corresponding to R0 (R0'R0=X)
31   // the vector d will contain the parsed diagonal of A and is returned
```

[3] See (Eddelbuettel et al., 2017)
[4] `g++` from `Rtools` (Ripley and Murdoch, 2017) has been tested
[5] Included with `Rtools`
[6] `dyn.load()` is a base R function

```
32  double **R1 = new double *[p];
33  double **A = new double *[p];
34  NumericVector d(p);
35  // create matrix row arrays, one for each row pointer
36  // divide rows of (R) by diagonals (sets diag to 1)
37  // eliminating need of division by R[i,i] in
38  // constructing elements in row i+1
39  for(i=0; i<p; i++) {
40    R1[i] = new double [p];
41    A[i] = new double [p];
42    for(j=0; j<p; j++)
43      R1[i][j]=R0(i,j)/R0(i,i);
44    // diagonals of right hand matrix are squared
45    // reciprocals of original R diagonals
46    // they are the initial values of the inverse
47    A[i][i]=1/R0(i,i)/R0(i,i);
48  }
49  // construct upper rows of the inverse in reverse
50  // order, copy to transpose positions for use by
51  // following row construction
52  // note that rows cannot be constructed in parallel
53  // since prior rows are necessary to construct
54  // subsequent rows
55  //omp_set_num_threads();
56  for(i=p-2; i>=0; i--) {
57    // construct columns in parallel since elements of
58    // rows are referenced
59    #pragma omp parallel for private(j, k, s)
60    for(j=i+1; j<p; j++) {
61      s=0;
62      // following iterative loop is the computationally
63      // intensive segment of this algorithm
64      // note that, although the analytical matrix
65      // equations multiply R[i,k]' by A[k,j], it
66      // is observed that A is symmetric which allows
67      // the equivalent operation R[i,k]' * A[j,k],
68      // which is considerably more efficient as
69      // implemented by the C compiler, since C stores
70      // a matrix in row dominant order, making elements
71      // adjacent in memory with respect to k (note that
72      // each increment of j corresponds to an in memory
73      // distance of p*8, where 8 is the number of bytes
74      // per double floating point element)
75      for(k=i+1; k<p; k++)
76        s+=R1[i][k]*A[j][k];
77      A[i][j]=-s;
78      // copy to transpose position
79      A[j][i]=-s;
80    }
81    // compute diagonal element on current row
82    for(j=i+1; j<p; j++)
83      A[i][i]=A[i][i]-R1[i][j]*A[i][j];
84  }
85  // copy all diagonal elements of computed inverse to result vector
```

```
86   for(i=0; i<p; i++)
87     d(i)=A[i][i];
88   // release memory allocated to dynamic rows
89   for(i=0; i<p; i++) {
90     delete [] R1[i];
91     delete [] A[i];
92   }
93   // release memory allocated to arrays of row pointers
94   delete [] R1;
95   delete [] A;
96   return(d);
97 }"
```

C source code notes:

- Line 5 loads the openmp library for parallel processing.
- Line 23 specifies the type of value to return (a numeric R vector), the name of the function (`cholInvDiag`), and the input parameter type (a numeric R matrix, containing a valid Cholesky decomposition).
- Lines 32 and 33 declare arrays of p vector pointers, one for each row of the input $\boldsymbol{R}$ and one for the computed inverse $\boldsymbol{A}$. Due to compiler limitations and errors during execution, large, single block matrices (`double R[p][p];`) could not be declared.
- Lines 39 through 48 point $\boldsymbol{R}$ rows to corresponding input $\boldsymbol{R0}$ (Cholesky decomposition) rows and initialize matrix $\boldsymbol{R}$ to the inverse of input $\boldsymbol{R0}$ diagonals (this corresponds to the values in matrix $\boldsymbol{B}$ in the above algorithm description). Line 47 computes a_{pp}, the initial lower-right sub-matrix, when $i = p$.
- Lines 56 through 84 implement the algorithm from above. A few special notes:
  - Line 59 (`# pragma omp parallel for`) instructs the compiler to implement the following for loop in parallel. Index j traverses columns $i + 1$ to p for the current row i. A block of j indices, one j for each column, is distributed to each available worker process (core, thread). The clause `private(j, k, s)` instructs the compiler to create one set of local, unshared, variables `j`, `k`, and `s` for each worker process. It is important that these indices and accumulation variables not be modified from a process outside of the one for which they are declared.
  - There are two summations in the algorithm presented above, both with k as the highest frequency iterative index. Since k appears as the first index, it corresponds to rows of $\boldsymbol{R}$. Because C stores a matrix in row major order (columns of a given row are physically stored in contiguous memory), it is advantageous, from a memory retrieval standpoint, to traverse along columns (when the compiler detects the ability to do so, it can implement *loop unrolling*, where multiple columns are retrieved simultaneously, thus improving bus utilization and synchronization with numeric processors). To align loop indices with optimal memory traversal, all operation involving matrix

$\boldsymbol{A}$ are accomplished using the transpose of those presented in the algorithm. Lines 75, 76 and 82, 83 implement high frequency summations from the algorithm, and it is seen that the indices of $\boldsymbol{A}$ are transposed from k, j and j, i to j, k and i, j. Since $\boldsymbol{A}$ is known to be symmetric, results are unaffected.

- Lines 86 and 87 copy the diagonal of $\boldsymbol{A}$ to a numeric R vector to be returned.

Compilation Instructions (from within R)

```
# compile and create .dll
# cacheDir contains .dll and R instructions for loading and creating the function in R
library(Rcpp)
sourceCpp(code=cSource, rebuild=T, showOutput=T, cacheDir=getwd(), cleanupCacheDir=F)
```

Function Execution (from within R)

```
# R is the Cholesky decomposition of a symmetric, positive-definite matrix
varB <- cholInvDiag(R)
```

References

Dirk Eddelbuettel, Romain Francois, JJ Allaire, Kevin Ushey, Qiang Kou, Nathan Russell, Douglas Bates, and John Chambers. R Rcpp Package, 2017. URL `https://cran.r-project.org/web/packages/Rcpp/Rcpp.pdf`.

Michael T. Heath. Parallel Numerical Algorithms, 2013. URL `https://courses.engr.illinois.edu/cs554/fa2013/notes/07_cholesky.pdf`.

Aravindh Krishnamoorthy and Deepak Menon. Matrix Inversion Using Cholesky Decomposition, 2017. URL `https://en.wikipedia.org/wiki/Cholesky_decomposition`.

Jong Tae Park and Chul Kang. A Cholesky Decomposition of the Inverse of a Covariance Matrix. *Journal of Korean Data & Information Science Society*, 14(4):1007–1012, 2003.

Brian Ripley and Duncan Murdoch. Rtools, 2017. URL `https://cran.r-project.org/bin/windows/Rtools/`.

Wikipedia. Cholesky decomposition, 2017. URL `https://pdfs.semanticscholar.org/f229/a57ee5611ca84a8936fdfc29a3f1f19dc1e9.pdf`.