Statistics 580

Matrix Computations

Introduction

Preliminaries:

- `References used in this introduction:`

1. Golub and van Loan (1984a,1991b) Matrix Computations
2. Thisted (1988) Elements of Statistical Computing
3. Gentle (1998) Numerical Linear Algebra for Applications in Statistics
4. Dongarra et al. (1978) Linpack User's Guide
5. Anderson et al. (1992) Lapack User's Guide

- `Notation:`

A:	an $m \times n$ matrix
$\mathbf{a}$:	$m \times 1$ vector (printed boldface)
b_j, $b(j)$:	elements of vectors
a_{ij}, $A(i,j)$:	elements of matrices
$A(i)$, $A(,j)$:	columns or rows of matrices
A^T, $\mathbf{a}^T$, A', $\mathbf{a}'$:	transpose of vectors and matrices
$\mathbf{e}_k$:	k^{th} elementary vector $(0,\ldots,1,0,\ldots,0)^T$
I, I_n:	$n \times n$ Identity matrix
L, R, T:	triangular matrices

- `Background in Matrix Algebra:`

Some knowledge is assumed. But read § 1.1–1.3, § 2.1–2.2 of Reference Text 1a, e.g., vector norms

p-norm: $\parallel \mathbf{x} \parallel_p = (|x_1|^p + \cdots + |x_n|^p)^{1/p}$

2-norm: $\parallel \mathbf{x} \parallel_2$ denoted simply by $\parallel \mathbf{x} \parallel$

1-norm: $\parallel \mathbf{x} \parallel_1 = |x_1| + \cdots + |x_n|$

α-norm $= \max_i |x_i|$

- `Matrix norms`

`Definition:` The matrix p-norm is defined as:

$$\parallel A \parallel_p = \sup_{\mathbf{x} \neq \mathbf{0}} \frac{\parallel A_{\mathbf{x}} \parallel_p}{\parallel \mathbf{x} \parallel_p}$$

The 2-norm $\| A \|_2$ is invariant with respect to orthogonal transformations i.e.

$$\| A \|_2 = \| PAQ \|_2$$

where P and Q are orthogonal matrices.

$$\begin{aligned} \| A \|_2 &= (\text{largest eigenvalue of } A^T A)^{\frac{1}{2}} \\ \| A \|_1 &= \max_j \sum_{i=1}^{n} |a_{ij}| \\ \| A \|_\infty &= \max_i \sum_{j=1}^{n} |a_{ij}| \end{aligned}$$

- `Definition of a flop (floating point (FP) operation)`

  Consider the generic computation (in pseudocode): sum = sum + a(i,k)*b(k,j)

  `old definition`: 1 FP addition + 1 FP multiply + a little subscripting

  `current definition`: 1 FP multiply + some subscripting

Matrix Algorithms

- `Example`: Compute $C = AB$ where A and B are upper triangular

  Algebraic computation:

$$\begin{aligned} c_{ij} &= \sum_{k=i}^{j} a_{ij} b_{kj},\ 1 \le i \le j \le n \\ &= 0,\ 1 \le j \le i \le n \end{aligned}$$

  Algorithm in pseudocode:

```
for  $i = n : 1$
   for $j = n : i$
      $A(i,j) \longleftarrow \sum_{k=i}^{j} A(i,k)B(k,j)$
   end
end
```

- `Features of the above computational algorithm:`

  1. Exploits structure; does not use standard matrix multiply.
  2. AB overwrites A.
  3. The summation $\sum_{k+i}^{j} A(i,k)B(k,j)$ is an inner product, so it is preferable if a standard procedure is used for its computation.
  4. Algorithm requires $n^3/6$ flops! (Count only multiplies since we count one add for each multiply).

- `Computation of the flops required for the above computation.`

$$\begin{aligned}\text{No. of Multiplies} &= \sum_{i=n}^{1}\sum_{j=n}^{i}(j-i+1) = \sum_{i=n}^{1}\sum_{k=1}^{n-i+1} k \\ &= \sum_{i=n}^{1}\sum_{k=1}^{n-i} k + \underbrace{\sum_{i=1}^{n}(n-i+1)}_{0(n^2)}\end{aligned}$$

where, we use the known results:

$\sum_{p=1}^{q} p = \frac{q(q+1)}{2}$

$\sum_{p=1}^{q} p^2 = \frac{1}{3}q^3 + \frac{1}{2}q^2 + \frac{1}{6}q$

We need only compute the 'Order of Magnitude' or O( ), i.e.,

$$\begin{aligned}\sum_{i=n}^{1}\sum_{k=1}^{n-i} k &= \sum_{i=n}^{1}\frac{(n-i)(n-i+1)}{2} \\ &= \sum_{i=n}^{1}\frac{(n-i)^2}{2} + O(n^2) \\ &= \frac{1}{2}\sum_{i=1}^{n} i^2 + O(n^2) \\ &= O(n^3/6)\end{aligned}$$

- `Matrix storage`

There is some consideration memory storage in matrix algorithms. Above pseudocode requires $2n^2$ storage locations as presented. However, $n \times n$ upper triangular matrix T can be stored as a $n(n+1)/2$ 1-dimensional array. For example, the matrix

$$\begin{bmatrix} \downarrow & \downarrow & \downarrow & \cdots & \downarrow \\ t_{11} & t_{12} & t_{13} & \cdots & t_{1n} \\ & t_{22} & \cdots & & t_{2n} \\ & & \ddots & & \vdots \\ & & & & t_{nn} \end{bmatrix}$$

can be stored in a 1-dimensional array as $t = (t_{11}, t_{12}, t_{22}, \ldots t_{1n}, \ldots, t_{nn})$. This halves the memory requirement at the expense of more cumbersome addressing t_{ij} is found in $t(i + j(j-1)/2)$ in the array t for $i \leq j$. This method is also used to store symmetric matrices and is called the symmetric storage mode. Other storage schemes such as triangular mode, band mode, and sparse storage mode are also used in several packages as well as in BLAS.

Basic Linear Algebra Operations

There are several basic computations involving vectors and matrices that commonly occur across many statistical applications. Computing the dot product of two vectors, for example, is a computation needed in such varied tasks as fitting a multiple regression model to data or the maximization of a likelihood function. The sets of routines called "basic linear algebra subprograms" (BLAS) implement many standard operations for vectors and matrices.

The level 1 BLAS or BLAS-1, the original set of the BLAS, are for vector operations. They were defined by Lawson et al. (1979). Matrix operations, such as multiplying two matrices were built using the BLAS-1. Later, a set of the BLAS, called level 2 or the BLAS-2, for operations involving a matrix and a vector, was defined by Dongarra et al. (1988), a set called the level 3 BLAS or the BLAS-3, for operations involving two matrices, was defined by Dongarra et al. (1990).

When work was being done on the BLAS-1 in the 1970s, those routines were incorporated into a set of Fortran routines for solutions of linear systems, called LINPACK (Dongarra et al., 1979). As work progressed on the BLAS-2 and BLAS-3 in the 1980s and later, a unified set of Fortran routines for both eigenvalue problems and solutions of linear systems was developed, called LAPACK (Anderson et al., 1995).

What follows below are some examples of algorithms used in common operations defined in the three BLAS packages.

- `saxpy operation` – a basic vector operation used in LINPACK (BLAS level-1)

  Defined algebraically as $\mathbf{z} = \alpha\mathbf{x} + \mathbf{y}$ it is an O(n) operartion.

  In pseudocode:

```
function: saxpy $(\alpha, \mathbf{x}, \mathbf{y})$
    $n \leftarrow$ length $(\mathbf{x})$
    for $i = 1 : n$
        $z(i) \leftarrow \alpha\, x(i) + y(i)$
    end
    return $(\mathbf{z})$
end saxpy.
```

- `dot product operation` – a basic vector operation used in LINPACK (BLAS level-1)

  Defined algebraically as $c = \mathbf{x}^T\mathbf{y}$ it is an O(n) operartion.

  In pseudocode:

```
function: sdot $(\mathbf{x}, \mathbf{y})$
    $c \leftarrow 0; n \leftarrow$ length$(\mathbf{x})$
    for $i = 1 : n$
        $c \leftarrow c + x(i)y(i)$
    end
    return(c)
end sdot
```

- `matrix-vector multiply`

  Defined as

$$\mathbf{z} = A\mathbf{x} \quad \text{where} \quad A \,\epsilon\, \Re^{n\times n}, \quad \mathbf{z} \,\epsilon\, \Re^{n}$$

  computed algebraically elementwise, using

$$z_i = \sum_{j=1}^{n} a_{ij}\, z_j$$

  This is the direct method of computation i.e., basically each element of $\mathbf{z}$ is a result of a dot product operation.

  In pseudocode:

```
function:  matvec.ij $(A, \mathbf{x})$
    $m \leftarrow \text{rows}(A); n \leftarrow \text{cols}(A)$
    $\mathbf{z} \leftarrow \mathbf{0}$
    for $i = 1 : m$
        for $j = 1 : n$
            $z(i) \leftarrow z(i) + A(i,j)x(j)$
        end
    end
    return $(\mathbf{z})$
end matvec.ij
```

  A pseudocode using the function `sdot` substitutes the following function call for the inner “for” loop:

$$z(i) \leftarrow \text{sdot}(A(i,:), \mathbf{x})$$

  Using C, assuming that the arrays used have been properly allocated, the above could be coded as:

```
s = 0;
for (i=0; i<m; i++){
    for (j=0; j<n; j++){
       s+= a[i,j]*y[j];
    }
z[i] = s;
}
```

  An alternative algorithm for the same operation results if we regard $\mathbf{z} = A\mathbf{x}$ as a linear combination of columns of A:

$$\mathbf{z} = \sum_{j=1}^{n} x_j\, \mathbf{a}_j \text{ where } A = (\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n).$$

Now the computation is expressed as n `saxpy` operations, as follows:

```
function: matvec.ji $(A, \mathbf{x})$
    $m \leftarrow \text{rows}(A); n \leftarrow \text{cols}(A)$
    $\mathbf{z} \leftarrow \mathbf{0}$
    for $j = 1 : n$
        for $i = 1 : m$
            $z(i) \leftarrow z(i) + x(j)A(i, j)$
        end
    end
    return($\mathbf{z}$)
end matvec.ji
```

Use function `saxpy` to replace the inner loop by:

$$\mathbf{z} \leftarrow \text{saxpy}(x(j), A(:, j), \mathbf{z}).$$

It is important to note that the difference between these two algorithms is the order of the two loops. Because of how arrays are stored in languages (i.e. their data structures), algorithms that access arrays by column turn out to be preferable.
From the above it is clear that $\mathbf{z}$ is formed by a sequence of n saxpy operations. Taken together, these could be used to define a new operation called `gaxpy`, i.e., a "generalized saxpy," operation.

- `gaxpy operation:` BLAS level-2 routine

  Defined algebraically as: $\mathbf{z} = A\mathbf{x} + \mathbf{y}$, it is an $\mathrm{O}(mn)$ operation.

  In pseudocode:

```
function gaxpy $(A, \mathbf{x}, \mathbf{y})$
    $n \leftarrow \text{cols}(A); \mathbf{z} \leftarrow \mathbf{y}$
    for $j = 1 : n$
        $\mathbf{z} \leftarrow \mathbf{z} + x(j)A(:, j)$
    end
    return($\mathbf{z}$)
end gaxpy
```

Notice that $\mathbf{z}$ is a running vector sum of a sequence of saxpy's. Another useful saxpy-based matrix operation is:

- `matrix-matrix multiply`

  `dot product version:` $C = AB$ using $c_{ij} = \sum_{k=1}^{r} a_{ik}\, b_{kj}$

  In pseudocode:

```
function: matmat.ijk $(A, B)$
    $m \leftarrow$ rows$(A)$; $r \leftarrow$ cols$(A)$; $n \leftarrow$ cols$(B)$
    $C \leftarrow 0$
    for $i = 1 : n$
        for $j = 1 : n$
            for $k = 1 : r$
                $C(i,j) \leftarrow C(i,j) + A(i,k)B(k,j)$
            end
        end
    end
    return$(C)$
end matmat.ijk
```

  Replace the k-loop by

$$C(i,j) \leftarrow \text{sdot}(A(i,:), B(:,j))$$

  using the level-1 operation `sdot`.

- `gaxpy version of matrix-matrix multiply:`

  As in the case of $A\mathbf{x}$, we can consider each column of C in $C = AB$ as a linear combination of columns of A:

$$\mathbf{c}_j = \sum_{k=1}^{r} b_{kj}\, \mathbf{a}_k$$

  where $C = (\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_n)$, $A = (\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_r)$ so the algorithm is simply:

```
$C \leftarrow 0$
for $j = 1 : n$
        $C(:,j) \leftarrow$ gaxpy$(A, B(:,j), C(:,j)$
end
```

  using the level-2 operation gaxpy.

Condition Number

We have had a brief discussion of accuracy and numerical stability of an accuracy and numerical stability of an algorithm. While for a given set of data, one algorithm may not be stable, another may give very accurate results.

The term “condition” is used to denote a characteristic of input data relative to a computational problem. The condition measures the relative change in the computed results due to a relative change in the input data.

Numerical Algorithms produce output from input i.e., output $= f(\text{input})$. We can then define the `condition number` κ as

$$\text{relative error in output} = \frac{|f(\text{input}+\delta) - \text{output}|}{\text{output}} \leq \kappa \cdot \frac{|\delta|}{\text{input}}$$

Small condition numbers are obviously better.

`Example:`

$$A = \begin{bmatrix} 5 & 7 & 6 & 5 \\ 7 & 10 & 8 & 7 \\ 6 & 8 & 9.99 & 9 \\ 5 & 7 & 9 & 10 \end{bmatrix} \qquad A^{-1} = \begin{bmatrix} 68 & -41 & -17 & 10 \\ -41 & 25 & 10 & -6 \\ -17 & 10 & 5 & -3 \\ 10 & -6 & -3 & 2 \end{bmatrix}$$

$$A^{-1} = \begin{bmatrix} 71.04 & -42.79 & -17.89 & 10.54 \\ -42.79 & 26.05 & 10.53 & -6.32 \\ -17.89 & 10.53 & 5.26 & -3.16 \\ 10.54 & -6.32 & -3.16 & 2.09 \end{bmatrix}$$

`Note:` 0.1% change in data results in $\frac{|10-10{\cdot}54|}{10} \times 100 = 5.4\%$change in result

We need to define a matrix norm to measure relative sizes of elements; here the Euclidean 2-norm will be used for most purposes.

`Definition` If A is a square matrix, then its condition number with respect to the matrix 2-norm $\| \cdot \|$ is

$$\kappa(A) = \| A \| \cdot \| A^{-1} \| .$$

with respect to the 2-norm, $\kappa = \ell_1/\ell_p$ i.e., ratio of largest to smallest singular values of A. If A is $n \times p$, $n \geq p$ we still use the same ratio as κ. The smallest value of κ is 1; occurs when A is orthogonal; if A is exactly collinear then $\kappa = \alpha$. Thus κ measures the closeness of A to rank deficiency.

The condition number κ plays a key role in least squares computations. Consider the solution to the problem

$$A\mathbf{x} = \mathbf{b}$$

A result (Theorem 2.5–1 of Reference 1a or 2) shows that introducing relative changes in the inputs $(A, \mathbf{b})$ of size δ can cause relative changes in the solution by a multiple of $\kappa(X)\delta$. Thus, roughly speaking, if the input data are good to t decimal places the solution to the linear system are good to $t - \log_{10}(\kappa(X))$. When $\kappa(X)$ is large, then the linear system is said to be `ill-conditioned`.

Level 1 BLAS

`dim scalar vector   vector   scalars              5-element array`		prefixes
`SUBROUTINE xROTG (                                A, B, C, S )`	Generate plane rotation	S, D
`SUBROUTINE xROTMG(                        D1, D2, A, B,        PARAM )`	Generate modified plane rotation	S, D
`SUBROUTINE xROT  ( N,         X, INCX, Y, INCY,               C, S )`	Apply plane rotation	S, D
`SUBROUTINE xROTM ( N,         X, INCX, Y, INCY,                PARAM )`	Apply modified plane rotation	S, D
`SUBROUTINE xSWAP ( N,         X, INCX, Y, INCY )`	$x \leftrightarrow y$	S, D, C, Z
`SUBROUTINE xSCAL ( N,  ALPHA, X, INCX )`	$x \leftarrow \alpha x$	S, D, C, Z, CS, ZD
`SUBROUTINE xCOPY ( N,         X, INCX, Y, INCY )`	$y \leftarrow x$	S, D, C, Z
`SUBROUTINE xAXPY ( N,  ALPHA, X, INCX, Y, INCY )`	$y \leftarrow \alpha x + y$	S, D, C, Z
`FUNCTION   xDOT  ( N,         X, INCX, Y, INCY )`	$dot \leftarrow x^T y$	S, D, DS
`FUNCTION   xDOTU ( N,         X, INCX, Y, INCY )`	$dot \leftarrow x^T y$	C, Z
`FUNCTION   xDOTC ( N,         X, INCX, Y, INCY )`	$dot \leftarrow x^H y$	C, Z
`FUNCTION   xxDOT ( N,         X, INCX, Y, INCY )`	$dot \leftarrow \alpha + x^T y$	SDS
`FUNCTION   xNRM2 ( N,         X, INCX )`	$nrm2 \leftarrow \|\|x\|\|_2$	S, D, SC, DZ
`FUNCTION   xASUM ( N,         X, INCX )`	$asum \leftarrow \|\|re(x)\|\|_1 + \|\|im(x)\|\|_1$	S, D, SC, DZ
`FUNCTION   IxAMAX( N,         X, INCX )`	$amax \leftarrow 1^{st} k \ni \|re(x_k)\| + \|im(x_k)\| = max(\|re(x_i)\| + \|im(x_i)\|)$	S, D, C, Z

Level 2 BLAS

`     options            dim   b-width scalar matrix  vector   scalar vector`		
`xGEMV (        TRANS,      M, N,         ALPHA, A, LDA, X, INCX, BETA,  Y, INCY )`	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
`xGBMV (        TRANS,      M, N, KL, KU, ALPHA, A, LDA, X, INCX, BETA,  Y, INCY )`	$y \leftarrow \alpha Ax + \beta y, y \leftarrow \alpha A^T x + \beta y, y \leftarrow \alpha A^H x + \beta y, A - m \times n$	S, D, C, Z
`xHEMV ( UPLO,                 N,         ALPHA, A, LDA, X, INCX, BETA,  Y, INCY )`	$y \leftarrow \alpha Ax + \beta y$	C, Z
`xHBMV ( UPLO,                 N, K,      ALPHA, A, LDA, X, INCX, BETA,  Y, INCY )`	$y \leftarrow \alpha Ax + \beta y$	C, Z
`xHPMV ( UPLO,                 N,         ALPHA, AP,     X, INCX, BETA,  Y, INCY )`	$y \leftarrow \alpha Ax + \beta y$	C, Z
`xSYMV ( UPLO,                 N,         ALPHA, A, LDA, X, INCX, BETA,  Y, INCY )`	$y \leftarrow \alpha Ax + \beta y$	S, D
`xSBMV ( UPLO,                 N, K,      ALPHA, A, LDA, X, INCX, BETA,  Y, INCY )`	$y \leftarrow \alpha Ax + \beta y$	S, D
`xSPMV ( UPLO,                 N,         ALPHA, AP,     X, INCX, BETA,  Y, INCY )`	$y \leftarrow \alpha Ax + \beta y$	S, D
`xTRMV ( UPLO, TRANS, DIAG,    N,                A, LDA, X, INCX )`	$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
`xTBMV ( UPLO, TRANS, DIAG,    N, K,             A, LDA, X, INCX )`	$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
`xTPMV ( UPLO, TRANS, DIAG,    N,                AP,     X, INCX )`	$x \leftarrow Ax, x \leftarrow A^T x, x \leftarrow A^H x$	S, D, C, Z
`xTRSV ( UPLO, TRANS, DIAG,    N,                A, LDA, X, INCX )`	$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$	S, D, C, Z
`xTBSV ( UPLO, TRANS, DIAG,    N, K,             A, LDA, X, INCX )`	$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$	S, D, C, Z
`xTPSV ( UPLO, TRANS, DIAG,    N,                AP,     X, INCX )`	$x \leftarrow A^{-1}x, x \leftarrow A^{-T}x, x \leftarrow A^{-H}x$	S, D, C, Z
`     options            dim   scalar vector   vector   matrix`		
`xGER  (                    M, N, ALPHA, X, INCX, Y, INCY, A, LDA )`	$A \leftarrow \alpha xy^T + A, A - m \times n$	S, D
`xGERU (                    M, N, ALPHA, X, INCX, Y, INCY, A, LDA )`	$A \leftarrow \alpha xy^T + A, A - m \times n$	C, Z
`xGERC (                    M, N, ALPHA, X, INCX, Y, INCY, A, LDA )`	$A \leftarrow \alpha xy^H + A, A - m \times n$	C, Z
`xHER  ( UPLO,                 N, ALPHA, X, INCX,          A, LDA )`	$A \leftarrow \alpha xx^H + A$	C, Z
`xHPR  ( UPLO,                 N, ALPHA, X, INCX,          AP )`	$A \leftarrow \alpha xx^H + A$	C, Z
`xHER2 ( UPLO,                 N, ALPHA, X, INCX, Y, INCY, A, LDA )`	$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	C, Z
`xHPR2 ( UPLO,                 N, ALPHA, X, INCX, Y, INCY, AP )`	$A \leftarrow \alpha xy^H + y(\alpha x)^H + A$	C, Z
`xSYR  ( UPLO,                 N, ALPHA, X, INCX,          A, LDA )`	$A \leftarrow \alpha xx^T + A$	S, D
`xSPR  ( UPLO,                 N, ALPHA, X, INCX,          AP )`	$A \leftarrow \alpha xx^T + A$	S, D
`xSYR2 ( UPLO,                 N, ALPHA, X, INCX, Y, INCY, A, LDA )`	$A \leftarrow \alpha xy^T + \alpha yx^T + A$	S, D
`xSPR2 ( UPLO,                 N, ALPHA, X, INCX, Y, INCY, AP )`	$A \leftarrow \alpha xy^T + \alpha yx^T + A$	S, D

Level 3 BLAS

`     options                          dim      scalar matrix  matrix  scalar matrix`		
`xGEMM (             TRANSA, TRANSB,      M, N, K, ALPHA, A, LDA, B, LDB, BETA,  C, LDC )`	$C \leftarrow \alpha op(A)op(B) + \beta C, op(X) = X, X^T, X^H, C - m \times n$	S, D, C, Z
`xSYMM ( SIDE, UPLO,                      M, N,    ALPHA, A, LDA, B, LDB, BETA,  C, LDC )`	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^T$	S, D, C, Z
`xHEMM ( SIDE, UPLO,                      M, N,    ALPHA, A, LDA, B, LDB, BETA,  C, LDC )`	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C - m \times n, A = A^H$	C, Z
`xSYRK (       UPLO, TRANS,                  N, K, ALPHA, A, LDA,         BETA,  C, LDC )`	$C \leftarrow \alpha AA^T + \beta C, C \leftarrow \alpha A^T A + \beta C, C - n \times n$	S, D, C, Z
`xHERK (       UPLO, TRANS,                  N, K, ALPHA, A, LDA,         BETA,  C, LDC )`	$C \leftarrow \alpha AA^H + \beta C, C \leftarrow \alpha A^H A + \beta C, C - n \times n$	C, Z
`xSYR2K(       UPLO, TRANS,                  N, K, ALPHA, A, LDA, B, LDB, BETA,  C, LDC )`	$C \leftarrow \alpha AB^T + \bar{\alpha} BA^T + \beta C, C \leftarrow \alpha A^T B + \bar{\alpha} B^T A + \beta C, C - n \times n$	S, D, C, Z
`xHER2K(       UPLO, TRANS,                  N, K, ALPHA, A, LDA, B, LDB, BETA,  C, LDC )`	$C \leftarrow \alpha AB^H + \bar{\alpha} BA^H + \beta C, C \leftarrow \alpha A^H B + \bar{\alpha} B^H A + \beta C, C - n \times n$	C, Z
`xTRMM ( SIDE, UPLO, TRANSA,        DIAG, M, N,    ALPHA, A, LDA, B, LDB )`	$B \leftarrow \alpha op(A)B, B \leftarrow \alpha Bop(A), op(A) = A, A^T, A^H, B - m \times n$	S, D, C, Z
`xTRSM ( SIDE, UPLO, TRANSA,        DIAG, M, N,    ALPHA, A, LDA, B, LDB )`	$B \leftarrow \alpha op(A^{-1})B, B \leftarrow \alpha Bop(A^{-1}), op(A) = A, A^T, A^H, B - m \times n$	S, D, C, Z

Meaning of prefixes

S - REAL	C - COMPLEX
D - DOUBLE PRECISION	Z - COMPLEX*16 (this may not be supported by all machines)

For the Level 2 BLAS a set of extended-precision routines with the prefixes ES, ED, EC, EZ may also be available.

Level 1 BLAS
In addition to the listed routines there are two further extended-precision dot product routines DQDOTI and DQDOTA.

Level 2 and Level 3 BLAS
Matrix types:

GE - GEneral	GB - General Band	
SY - SYmmetric	SB - Sym. Band	SP - Sum. Packed
HE - HErmitian	HB - Herm. Band	HP - Herm. Packed
TR - TRiangular	TB - Triang. Band	TP - Triang. Packed

Level 2 and Level 3 BLAS Options
Dummy options arguments are declared as CHARACTER*1 and may be passed as character strings.

TRANx	= '**N**o transpose', '**T**ranspose', '**C**onjugate transpose' (X, X^T, X^H)
UPLO	= '**U**pper triangular', '**L**ower triangular'
DIAG	= '**N**on-unit triangular', '**U**nit triangular'
SIDE	= '**L**eft', '**R**ight' (A or op(A) on the left, or A or op(A) on the right)

For real matrices, TRANSx = 'T' and TRANSx = 'C' have the same meaning.
For Hermitian matrices, TRANSx = 'T' is not allowed.
For complex symmetric matrices, TRANSx = 'H' is not allowed.

References

C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Trans. on Math. Soft.* 5 (1979) 308-325

J.J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson, "An Extended Set of Fortran Basic Linear Algebra Subprograms," *ACM Trans. on Math. Soft.* 14,1 (1988) 1-32

J.J. Dongarra, I. Duff, J. DuCroz, and S. Hammarling, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. on Math. Soft.* (1989)

Obtaining the Software via `netlib@ornl.gov`

To receive a copy of the single-precision software, type in a mail message:

```
send sblas from blas
send sblas2 from blas
send sblas3 from blas
```

To receive a copy of the double-precision software, type in a mail message:

```
send dblas from blas
send dblas2 from blas
send dblas3 from blas
```

To receive a copy of the complex single-precision software, type in a mail message:

```
send cblas from blas
send cblas2 from blas
send cblas3 from blas
```

To receive a copy of the complex double-precision software, type in a mail message:

```
send zblas from blas
send zblas2 from blas
send zblas3 from blas
```

Send comments and questions to `lapack@cs.utk.edu` .

Basic Linear Algebra Subprograms

A Quick Reference Guide

University of Tennessee
Oak Ridge National Laboratory
Numerical Algorithms Group Ltd.

May 11, 1997

Using BLAS/LAPACK routines in C programs

LAPACK is a linear algebra package written in Fortran77 that provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. LAPACK also provides the associated matrix factorizations (e.g., LU, Cholesky, QR, SVD) and also related computations such as solving linear systems and estimating condition numbers. Dense and banded matrices are handled, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices, in both single and double precision.

LAPACK is available from NETLIB at *www.netlib.org/lapack* which is also a good reference for using the many routines. It is a very-reputed package, and is also used by statistical packages such as `R` and scientific software libraries such as NAG and GSL. However, perhaps because numerical algorithms were written in Fortran, it is a Fortran 77 package and it needs to be invoked appropriately. By re-arranging C data arrays in the *column major order* as required by Fortran matrix data structure, it is possible to call Fortran, and in particular LAPACK routines, from C.

When compiling C code that includes calls to Fortran functions from LAPACK using `gcc`, the flag `-llapack` is added during compilation. The Basic Linear Algebra Subroutines (BLAS-1, BLAS-2, and BLAS-3) routines discussed earlier are also available as a Fortran 77 package and if calls to BLAS functions are included in the C program, the linker flag `-lblas` is needed in addition to the flag `-llapack`.

Calling Fortran Routines from C

A major difference between C and Fortran lies in the way arguments are passed to a subroutine or a function. In Fortran, the argument is passed to the routine and the routine can read and change this value. This mechanism is called *call by reference.* In C a copy of the value of the argument is passed to the function and not the argument itself, the subroutine can't change the value of the argument. This mechanism is called *call by value.* To achieve the effect of call-by-reference in C, we pass pointers to objects to functions, rather than the values of the objects.

Another difference between C and Fortran lies in how matrices are stored in memory. Fortran stores matrices by column (column major order), rather than by row, which is what C does. Therefore, we need to rearrange any matrix in C before passing it to Fortran. Any resulting matrix output from the Fortran routine should therefore be necessarily converted back. It is simpler to pass matrices as *one-dimensional arrays* to the BLAS/LAPACK Fortran routines. If a matrix is stored as a 2-dimensional array in C, just access the successive rows and store them consecutively in a 1-dimensional C array, and pass this array as the argument in the calls to the Fortran function where matrix argument is needed.

In addition, for the C compiler to recognize that a function being invoked is a Fortran subroutine, an underscore ("`_`") is appended to the end of the subroutine name. For instance, the LAPACK function `dpotri` in Fortran should be called in C using the name `dpotri_`. These features are illustrated in example that follows.

An Example: Matrix Inversion using Cholesky factorization

A list of LAPACK functions are avilable from the Quick Reference Guide found at the end of this note. Once the name of the function is found use the `man` command under Linux to get the manual page describing how this function is to be called in Fortran. For example enter, `man dpotri` to get the manual page for the LAPACK subroutine `dpotri`.

Note that there are single-precision as well as double-precision versions of the routines in LAPACK, only the use of double-precision functions are described here. (In addition, there are complex double-precision and complex single-precision routines available.) At least three LAPACK routines perform matrix inversion. Here, only one – the inversion of a symmetric positive-definite matrix, using `dpotri` is discussed.

The Fortran subroutine `SUBROUTINE DPOTRI( UPLO, N, A, LDA, INFO )` is a facility for inversion of a symmetric positive-definite matrix A. The manual page for this subroutine states that the subroutine `DPOTRI` "computes the inverse of a real symmetric positive definite matrix A using the Cholesky factorization $A = U'U$ or $A = LL'$ as computed by `DPOTRF`." The description of the LAPACK subroutine `SUBROUTINE DPOTRF(UPLO, N, A, LDA, INFO)` indicates that it "computes the Cholesky factorization of a real symmetric positive definite matrix A." Thus, in order to invert a symmetric postive-definite matrix, the subroutine `DPOTRF` is first called followed by `DPOTRI`.

As per the man page, the arguments for `DPOTRF` are as follows:

UPLO The first argument is a `char` of length 1, with a value of `U` indicating that the upper triangular portion of the matrix `A` is used, and the strictly lower triangular part of `A` is not referenced, while `L` indicates that the lower triangle is being used, and the strictly upper part of `A` is not referenced.

N The next argument `N` is the order of the matrix.

A It is followed by `A`, which is a double precision array. On input, if `UPLO` is indeed `U` this matrix `A` contains the upper triangular part of the matrix `A`. If, on the other hand, `UPLO` is `L`, the lower triangular part of `A` contains the lower triangle of `A`. Here the properly arranged 1-dimensional array is passed.

LDA The fourth argument pertains to the leading dimension of the array. (see below for a description of this parameter.)

INFO As output the integer `INFO` which returns zero on successful exit. If `INFO` returns a negative value, then it indicates that the argument indicated by `-INFO` has an illegal value, while if `INFO` returns a positive value, then the leading minor of order given by the returned value of `INFO` is not positive definite, and the factorization could not be completed.

So, what is the leading dimension of an array? This is a Fortran feature associated with arrays. When a 2-dimensional array `A` is declared in Fortran with the following parameters:

```
number of rows:    m
number of columns: n
leading dimension: p     [ p >= m ]
```

it reserves memory for storing a $p \times n$ array, of which only the top m rows are actually used. If the array entries (scanned column-wise) are to be regarded as stored in a 1-dimensional array `a()`, then:

```
A(i,j) = a(i + p*j)
```

For all practical purposes, the leading dimesion is taken to be the same as the number of rows of the matrix, i.e., $p = m$.

The output from `DPOTRF` is input into `DPOTRI` for which the arguments are exactly the same. So, a call for matrix inversion using the above LAPACK subroutines needs to first call `DPOTRF`, followed by a call to `DPOTRI`. Note also that the arguments to the Fortran routine are passed as C pointers. The first call to `dpotrf_` provides the Cholesky decomposition, if `INFO` returns a value of zero. If this happens, the sum of the squares of the diagonals of L are multiplied successively to obtain the determinant. Further, a call to `dpotri_` provides the inverse, which is then read back in column-wise and stored in the input array `A`. In the example all of the above steps are carried out in the C function `pdsinv.c`, that returns the inverse of `A` as well as the value of the determinant.

Calling the C function `pdsinv` is then similar to calling any C function, and is illustrated in the main program `chol_inv.c`. The utility functions `indata.c` and `printa.c` are used to handle the input and printing of the data matrix and the resulting matrix.

In the following pages the C main program (available in the file `chol_inv.c`), the C function `pdsinv` (available in the file `pdsinv.c`), and the utility C functions `indata` and `printa` (available in the file `util.c`) are listed. The annotations in the programs summarize the computations carried out in each segment.

```
/* Main program to call the function pdsinv.c to invert a symmetric
   positive definite matrix. The data matrix is read using the function
   indata.c and it is printed using the function printa.c */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void indata(double [][10], int*, int *, FILE *);
void printa(double [][10] , int,  int , FILE *);
void pdsinv(int , double [][10], double *);

int main(void)
{
  FILE *fin,*fout;
  int m, n;
  double a[10][10], determinant;
  if((fin=fopen("chol.dat","r"))==NULL)
    {
      fprintf(stderr,"Cannot open readfile \n");
      exit(1);
    }

  /* input data matrix */

  indata(a, &m, &n, fin);

  if((fout=fopen("chol.out","w"))==NULL)
    {
      fprintf(stderr,"Cannot open writefile \n");
      exit(1);
    }

  /* print data matrix */

  fprintf(fout ,"** Input Matrix: \n");
  printa(a, m, n, fout);

  /* call  driver routine to perform inversion */

  pdsinv(n, a, &determinant);

  /* print determinant and inverse */

  fprintf(fout ,"** Determinant: %g \n \n", determinant);

  fprintf(fout ,"** Inverse: \n");
  printa(a, m, n,fout);

  return 0;
}
```

```
/* pdsinv is a driver function for calling the LAPACK functions
dpotrf_ and dpotri_ for Cholesky factorization and inversion*/

#include <stdio.h>
#include <math.h>

void  dpotrf_(char *,int *, double *, int *, int *);
void  dpotri_(char *,int *, double *, int *, int *);

void pdsinv(int size, double (*a)[10], double *determinant)
{
  int i, j,INFO,N,LDA ;
  char uplo='L';
  double at[100];

  /* store matrix in array in column major form */

  for (i=0; i<size; i++)
    {
      for(j=0; j<size; j++) at[i*size+j]=a[j][i];
    }

  /* call dpotrf for Cholesky facorization */

  N=size;
  LDA=size;

  dpotrf_ (&uplo, &N, at, &LDA, &INFO);

  /* use lower triangle to compute determinant */

  if (INFO==0) {
    *determinant=1.0;
    for (i=0;i<N;i++) {
      *determinant *=at[i+i*N]*at[i+i*N];
    }

    /* use lower triangle to compute inverse using dpotri */

    dpotri_ (&uplo, &N, at, &LDA, &INFO);
    if (INFO!=0) {
      printf("Problem in pdsinv: dpotri error %d\n",INFO);
    }

    /* restore matrix in row major form */

    for (i=0; i<size; i++) {
      for(j=i; j<size; j++) {
      a[j][i]=at[i*size+j];
      a[i][j]=at[i*size+j];
      }
    }
  }
  else {
    printf("Problem in pdsinv: dpotrf error %d\n",INFO);
  }
}
```

```
/* Functions indata.c and printa.c available in the file util.c */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void indata(double (*a)[10], int *m, int *n, FILE *fin)
{
  int dim,i,j;
  dim=fscanf(fin,"%d %d",m,n);
  if(dim!=2)
    {
      fprintf(stderr,"First line of data must have dimensions \n");
      exit(1);
    }
  for(i=0;i<*m;i++)
    {
      for(j=0;j<*n;j++)
      {
        if(fscanf(fin,"%lf",&a[i][j])!=1)
          {
            fprintf(stderr,"Data incomplete\n");
            exit(1);
          }
      }
    }
  fclose(fin);
}

void printa(double (*a)[10], int m, int n, FILE *fout)
{
  int i,j,k;
  for(i=0;i<m;i++)
    {
      k=0;
      for(j=0;j<n;j++)
      {
        fprintf(fout,"%g ",a[i][j]);
        k++;
      }
      if(k%n==0)
      {
        fprintf(fout,"\n");
      }
    }
  fprintf(fout,"\n");
}
```

```
/* The main program reads a data file containing columns of x variable
values and a lst column with y variable values. The BLAS FORTRAN
routines dgemm and  dgemv are used to form the X'X matrix and X'y
vector. It calls the C function solver to solve X'Xb=X'y and retrn the
vector b, and calls the C function anova to compute the ANOVA table.*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void indata(double [][10], int*, int *, FILE *);
void printa(double [][10] , int,  int , FILE *);
void solver(int, double *, double *);
void anova(int, int, double *, double *, double *, FILE *);

void  dgemm_(char *, char *, int *, int *, int *, double *,double *,
                     int *,double *, int *, double *, double *,int *);
void dcopy_(int *, double *,int*, double *,int *);
void  dgemv_(char *, int *, int *, double *, double *, int *, double *,
                                   int *, double *, double *,  int *);

int main(void)
{
  FILE *fin,*fout;
  int mm, nn, n, p,i, j, lda ,incy;
  char transa='T', transb='N';
  double a[100][10], xt[1000], xpx[1000], y[100], xpy[10];
  double alpha,beta, xpy2[10];

  /* input data matrix */

  if((fin=fopen("reg.dat","r"))==NULL)
    {
      fprintf(stderr,"Cannot open readfile \n");
      exit(1);
    }
  indata(a, &mm, &nn, fin);
  p=nn-1;
  n=mm;

  if((fout=fopen("reg.out","w"))==NULL)
    {
      fprintf(stderr,"Cannot open writefile \n");
      exit(1);
    }
  fprintf(fout ,"** Input data: \n");
  printa(a, mm, nn, fout);

 /* arrange X for FORTRAN input (column major order) */

  for (i=0; i<p; i++)
    {
      for(j=0; j<n; j++) xt[i*n+j]=a[j][i];
    }
/* form X'X  and print */

  alpha=1.0;
  beta=0.0;
  lda=n;
  dgemm_(&transa, &transb, &p, &p, &n, &alpha, xt, &n, xt, &lda, &beta,
                   xpx, &p);

  fprintf(fout ,"** X'X matrix: \n");
  for(i=0;i<p;i++)
    {
      for(j=0;j<p;j++)
      {
      fprintf(fout,"%8.2f ",xpx[i*p+j]);
      }
      fprintf(fout," \n \n");
    }

  /* get y from input data and print it */

  for (i=0; i<n; i++)
    y[i]=a[i][p];

  /* form X'y and print;  save X'y for later use */

  alpha=1.0;
  incy=1;
  dgemv_(&transa, &n, &p, &alpha, xt, &n, y, &incy, &beta, xpy, &incy);

  fprintf(fout ,"** X'y vector: \n");
  for (i=0; i<p; i++)
    fprintf(fout ," %f ",y[i]);
  fprintf(fout ," \n \n");

  dcopy_(&p, xpy, &incy, xpy2, &incy);

  /* solve normal equations X'Xb=X'y for b and print */

  solver(p, xpx, xpy);

  fprintf(fout ,"** Parameter Estimates: \n");
  for (i=0; i<p; i++)
    fprintf(fout ," %f ",xpy[i]);
  fprintf(fout ," \n \n");

  /* compute ANOVA SS  and print table */

  anova(n, p, y, xpy, xpy2, fout);
  return 0;
}
```

```
/* The C function solver solves the system X'Xb=X'y. It compute the
Chelesky factor L using the LAPACK Fortran subroutine dpotrf and uses
it in dpotrs to compute the solution*/

#include <stdio.h>
#include <math.h>

void  dpotrf_(char *,int *, double *, int *, int *);
void  dpotrs_(char *,int *, int *, double *, int *,  double *, int *,
int *);

void solver(int p,double *xpx, double *xpy)
{
  int INFO, n, lda ,nrhs;
  char uplo='L';;

  n=p;
  lda=p;

  dpotrf_ (&uplo, &n, xpx, &lda, &INFO);

  if (INFO!=0)  {
    printf("Problem in SOLVER: dpotrf error %d\n",INFO);
  }

  else{
    nrhs=1;

    dpotrs_ (&uplo, &p, &nrhs, xpx, &lda, xpy, &lda, &INFO);

    if (INFO!=0)  {
    printf("Problem in SOLVER: dpotrs error %d\n",INFO);
    }
  }
}
```

Matrix Decompositions

Orthogonal Transformations

Given an $n \times 1$ vector $\mathbf{y}$ and an $n \times p$ matrix X, consider the least squares problem. That is, consider the problem of determining a $p \times 1$ vector $\boldsymbol{\beta}$ such that

$$\min_{\boldsymbol{\beta}} \| \mathbf{y} - X\boldsymbol{\beta} \|^2 \tag{1}$$

where $\| \cdot \|$ indicates the 2-norm (Euclidean vector norm). An $n \times n$ orthogonal matrix Q can be used to transform the regression model

$$\mathbf{y} = X\boldsymbol{\beta} + \boldsymbol{\epsilon} \qquad \boldsymbol{\epsilon} \sim (0, \sigma^2 I) \tag{2}$$

to

$$Q^T\mathbf{y} = Q^T X\boldsymbol{\beta} + Q^T\boldsymbol{\epsilon}$$

or

$$\mathbf{y}^* = X^*\boldsymbol{\beta} + \boldsymbol{\epsilon}^* \qquad \boldsymbol{\epsilon}^* \sim (0, \sigma^2 I) \tag{3}$$

The solutions of the least squares problem associated with model (2) and model (3) are equivalent. The computational task is to select an orthogonal transformation Q such that the least squares problem is easier to solve as well as the resulting algorithm is more stable than simply solving the normal equations.

One possible approach is to construct Q such that $Q^T X$ is zero below its diagonal. That is, (assuming $n \geq p$) $Q^T X$ can be written as the block upper triangular matrix

$$Q^T X = \begin{bmatrix} R \\ 0 \end{bmatrix} \tag{4}$$

where R is a $p \times p$ upper triangular matrix. Since orthogonal transformations preserve length, i.e., $\|Q^T\mathbf{z}\| = \|z\|$ we can write the quantity to be minimized in (1) as

$$\begin{aligned} \|\mathbf{y} - X\boldsymbol{\beta}\|^2 &= \|Q^T(\mathbf{y} - X\boldsymbol{\beta})\|^2 \\ &= \|\mathbf{y}_1^* - R\boldsymbol{\beta}\|^2 + \|\mathbf{y}_2^*\|^2 \end{aligned} \tag{5}$$

where $\mathbf{y}_1^*$ is $p \times 1$ and $\mathbf{y}_2^*$ $(n-p) \times 1$ vectors respectively s.t.

$$Q^T\mathbf{y} = (Q_1,\ Q_2)^T\mathbf{y} = \begin{bmatrix} \mathbf{y}_1^* \\ \mathbf{y}_2^* \end{bmatrix} .$$

Thus the choice of $\boldsymbol{\beta}$ that minimizes (5) clearly also minimizes (1). The problem

$$\min_{\boldsymbol{\beta}} \|\mathbf{y}_1^* - R\boldsymbol{\beta}\|^2 \tag{6}$$

is a very easy problem since R is an upper triangular matrix. If rank$(X) = p$ the solution is

$$\hat{\boldsymbol{\beta}} = R^{-1} Q^T\mathbf{y} = R^{-1}\mathbf{y}_1^* \tag{7}$$

which is easily computed directly by back-substitution. The residual sum of squares s^2 for model (2) is given by $\|Q_2^T \mathbf{y}\|^2 = \|\mathbf{y}_2^*\|^2 = \mathbf{y}_2^{*T}\,\mathbf{y}_2^*$. This orthogonal transformation can be computed by several numerical algorithms, Householder reflections: Givens rotations, and modified Gram-Schmidt. The Householder transformation will be described later in this note.

QR Decomposition

From (4) we see that X can be expressed in the form

$$\begin{aligned} X &= Q \begin{bmatrix} R \\ 0 \end{bmatrix} = [Q_1,\ Q_2] \begin{bmatrix} R \\ 0 \end{bmatrix} \\ &= Q_1 R \end{aligned} \tag{8}$$

where Q is an $n \times n$ orthogonal matrix and R is $p \times p$ upper triangular. This decomposition is called the QR factorization of X. If X is of full column rank then the above factorization shows that columns of Q_1 form an *orthonormal basis* for the column space of X (i.e., range(X)). Thus, computation of the QR factorization is one way to compute an orthonormal basis for a set of vectors. The matrix

$$P_X = Q_1 Q_1^T \tag{9}$$

is the orthogonal projection onto the column space of X, and the matrix

$$P_X^{\perp} = Q_2\, Q_2^T \tag{10}$$

is the projection onto the orthogonal complement of X. One important characteristic of this decomposition is that the QR factorization of the $n \times k$ matrix X_1 where X is partitioned as $(X_1,\ X_2)$ can be had at no additional cost after the full QR factorization has been computed. This is useful in regression computations where partial or reduced models may have to be fitted in addition to the full model. Suppose that R is partitioned as:

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix} \tag{11}$$

where R_{11} is $k \times k$, then $Q^T X_1 = \begin{bmatrix} R_{11} \\ 0 \end{bmatrix}$, so that if Q_1 is partitioned conformally as $Q_1 = (Q_{11},\ Q_{12})$, $X_1 = Q_{11}\, R_{11}$ is the factorization needed.

Note that the least squares problem associated with X_1 is

$$\min_{\boldsymbol{\beta}_1} \|\mathbf{y} - X_1 \boldsymbol{\beta}_1\|^2 \tag{12}$$

the solution of which is

$$\hat{\boldsymbol{\beta}}_1 = R_{11}^{-1}\, Q_{11}^T\, \mathbf{y} = R_{11}^{-1} \mathbf{z}_1 \tag{13}$$

where $\mathbf{z} = Q_1^T\mathbf{y} = \mathbf{y}_1^* = \begin{bmatrix} \mathbf{z}_1 \\ \mathbf{z}_2 \end{bmatrix}$. To obtain the residual sum of squares for the reduced model write $\|\mathbf{y} - X_1\boldsymbol{\beta}_1\|^2$ as

$$\begin{aligned} \|\mathbf{y} - X_1\boldsymbol{\beta}_1\|^2 &= \|Q^T(\mathbf{y} - X_1\boldsymbol{\beta}_1)\|^2 \\ &= \|Q_1^T\mathbf{y} - R_{11}\boldsymbol{\beta}_1\|^2 + \|Q_2^T\mathbf{y}\|^2 \\ &= \|Q_{11}^T\mathbf{y} - R_{11}\boldsymbol{\beta}_1\|^2 + \|Q_{12}^T\mathbf{y}\|^2 + \|Q_2^T\mathbf{y}\|^2 \,. \end{aligned} \tag{14}$$

Thus the residual sum of squares is given by

$$\|\mathbf{z}_2\|^2 + \|\mathbf{y}_2^*\|^2 \,. \tag{15}$$

Thus the full QR factorization enables one to solve least squares problems associated with subsets of columns in the order they appear in the full X matrix.

By computing the QR decomposition of the augmented matrix $\tilde{X} = (X,\ \mathbf{y})$, the quantities need to solve the regression problem are obtained at once because, then the triangular part has the form

$$\begin{bmatrix} R & \mathbf{z} \\ 0 & s \end{bmatrix}$$

where $s^2 = RSS$ and thus, $\hat{\boldsymbol{\beta}} = R^{-1}\mathbf{z}$. If residuals and predicted values are also needed then Q must be available for use. The predicted values are given by $Q_1\mathbf{z}$. The covariance matrix of $\hat{\boldsymbol{\beta}}$ is computed by the formula

$$s^2\, R^{-1}(R')^{-1}$$

where R^{-1} is easily computed using back-substitution to solve the system of linear equations

$$RW = I_p$$

to give $W = R^{-1}$. If X has full rank p then the Moore-Penrose generalized inverse is given by

$$X^\dagger = R^{-1}\, Q_1^T \,.$$

Quantities needed for fitting reduced models are easily obtained by examining formulas (13) and (15).

Householder Transformations

The matrix Q required in the regression problem will actually be built by combining a series of elementary transformations, i.e.,

$$Q^T = H_p\, H_{p-1} \ldots H_2\, H_1$$

such that

$$Q^T X = \begin{bmatrix} R \\ 0 \end{bmatrix} ,$$

where R is upper triangular. Each matrix H_t is a Householder transformation of the form

$$H_t = I_n - \frac{1}{\rho_t}\mathbf{u}_t\,\mathbf{u}_t^T$$

where $\mathbf{u}_t$ and ρ_t are chosen in such a way that H_t acts on the $n \times 1$ vector $\mathbf{x}$ by *rotating* it to make all but the first t components of the resulting vector $H_t\,\mathbf{x}$ equal to zero. Moreover, it can be shown that these can be chosen to preserve the first $(t-1)$ components of $\mathbf{x}$, as well. That is, $\mathbf{u}_t$ and ρ_t can be chosen such that

$$H_t\,\mathbf{x} = (x_1, x_2, \ldots, x_{t-1}, \,\tau\,, 0, \ldots, 0)^T\,.$$

where $\tau = \pm\,\|\mathbf{x}\|$.

As an example, we derive the vector $\mathbf{u}_1$ that defines the transformation H_1 such that $H_1\mathbf{x} = \tau\,\mathbf{e}_1$, where $\mathbf{e}_1$ is the unit vector $(1, 0, \ldots, 0)^T$. For any $\mathbf{x} \in \Re^n$ we have

$$H_1\,\mathbf{x} = \left(I_n - \frac{1}{\rho_1}\mathbf{u}_1\mathbf{u}_1^T\right)\,\mathbf{x} = \mathbf{x} - \frac{1}{\rho_1}\left(\mathbf{u}_1^T\mathbf{x}\right)\,\mathbf{u}_1\,.$$

For $H_1\mathbf{x}$ to be a multiple of $\mathbf{e}_1$, $\mathbf{u}_1$ must belong to the subspace spanned by $\{\mathbf{x},\, \mathbf{e}_1\}$, implying that $\mathbf{u}_1$ must be of the form

$$\mathbf{u}_1 = \mathbf{x} + \tau\,\mathbf{e}_1\,.$$

Thus $\mathbf{u}_1^T\mathbf{x} = \mathbf{x}^T\mathbf{x} + \tau x_1$ where x_1 is the first component of $\mathbf{x}$ and therefore

$$H_1\mathbf{x} = \mathbf{x} - \frac{1}{\rho_1}(\mathbf{x}'\mathbf{x} + \tau x_1)(\mathbf{x} + \tau\mathbf{e}_1).$$

Setting $\rho_1 = \mathbf{x}'\mathbf{x} + \tau x_1$ cancels out $\mathbf{x}$, leaving

$$H_1\mathbf{x} = -\tau\mathbf{e}_1\,.$$

Since H_1 must be orthogonal $\|H_1\mathbf{x}\| = \|\mathbf{x}\|$, implying that $\tau = \pm\,\|\mathbf{x}\|$. Thus

$$\rho_1 = \tau^2 + \tau\,x_1 = \pm\,\|\mathbf{x}\|u_1$$

where u_1 is first component of $\mathbf{u}$ and $H_1\mathbf{x} = \mp\,\|\mathbf{x}\|\mathbf{e}_1$.

Example If $\mathbf{x} = (3, 1, 5, 1)^T$, $\mathbf{u}_1 = (9, 1, 5, 1)^T$ and $\rho_1 = 54$, it follows that

$$H_1 = I - \frac{1}{54}\,\mathbf{u}_1\mathbf{u}_1^T = \frac{1}{54}\begin{bmatrix} -27 & -9 & -45 & -9 \\ -9 & 53 & -5 & -1 \\ -45 & -5 & 29 & -5 \\ -9 & -1 & -5 & 53 \end{bmatrix}$$

and $H_1\mathbf{x} = (-6, 0, 0, 0)^T$. Algebraically, the choice of the sign of τ is immaterial since either choice produces the desired transformation. However, *numerically* the choice of the sign is critical. The sign of τ is chosen to be the same as that of x_1, since otherwise a subtraction will take place in the formation of u_1 leading to possible cancellation problems.

Computing Considerations of the QR Decomposition

It is critical to exploit structure when computing Householder reflections. Failure to do this by treating the transformation Q as a product of p matrices would be very expensive in terms of computation time as well as storage space. As we proceed from right to left of the product $H_p\, H_{p-1}\, \ldots\, H_1$, each H_t is applied to a set of column vectors resulting from previous computations. For an arbitrary vector $\mathbf{w}$,

$$\begin{aligned} H_t\,\mathbf{w} &= \mathbf{w} - \frac{1}{\rho_t}\,(\mathbf{u}_t\,\mathbf{u}_t^T)\mathbf{w} \\ &= \mathbf{w} - \frac{1}{\rho_t}(\mathbf{u}_t^T\mathbf{w})\mathbf{u}_t \\ &= \mathbf{w} - c\,\mathbf{u}_t \end{aligned}$$

where $c = (\mathbf{u}_t^T\mathbf{w})/\rho_t = 2(\mathbf{u}_t^T\mathbf{w})/(\mathbf{u}_t^T\mathbf{u}_t)$. The first $(t-1)$ elements of $H_t\mathbf{w}$ are the same as those of $\mathbf{w}$; the rest are formed by the operation $w_j - cu_j$ where c is a constant formed once during the computation of $H_t\mathbf{w}$.

In practice, each application of H_t is considered as performing a first step (as defined in H_1 above) on the $(n-t+1)\times(p-t+1)$ submatrix on lower right corner of X, beginning with $t=1$ and going through to $t=p$. Given an $n\times p$ matrix X with $n \geq p$, the following algorithm constructs $H_1, H_2, \ldots, H_p$ such that

$$Q^T X = \begin{bmatrix} R \\ 0 \end{bmatrix}$$

where R is upper triangular. The transformation Q is stored in *factored form*, i.e., the $\mathbf{u}_t$ and ρ_t that correspond to each H_t are stored for $t = 1, \ldots, p$. Since $\mathbf{u}_t$ is of the form

$$\mathbf{u}_t = (\underbrace{0, \ldots, 0}_{t-1}, u_t, \ldots, u_p)^T,$$

non-zero portions of each $\mathbf{u}_t$ can be stored in the lower-triangular portion of X during the transformation. The ρ_t's are stored in a separate vector as well as the diagonal elements of R. The off-diagonal elements of R are stored above the diagonal in X. We give the following pseudocode for the Householder factorizations. Note that in this algorithm, X is an $n\times p$ matrix, b and r are $p\times 1$ vectors, $\mathbf{x}$ and $\mathbf{y}$ are internal (length-adjusting) vectors and τ and ρ scalars.

Algorithm QR1:

```
for $t = 1 : p$
        $\mathbf{x} \leftarrow X(t : n, t)$
        $\tau \leftarrow \mathbf{sign}\,(x_1)\|\mathbf{x}\|$
        $X(t, t) \leftarrow X(t, t) + \tau$
        $\rho \leftarrow X(t, t) * \tau$
        for $j = t + 1 : p$
                $\mathbf{x} \leftarrow X(t : n, t)$
                $\mathbf{y} \leftarrow X(t : n, j)$
                $c \leftarrow \mathbf{ddot}(\mathbf{x}, \mathbf{y})$
                $X(t : n, j) \leftarrow \mathbf{daxpy}(-\mathbf{c}/\rho, \mathbf{x}, \mathbf{y})$
        end
        $b(t) \leftarrow \rho$
        $r(t) \leftarrow -\tau$
end
```

Results from an application of Algorithm QR1 to a 6×5 matrix X can be represented as follows:

$$X = \begin{bmatrix} u_1^{(1)} & r_{12} & r_{13} & r_{14} & r_{15} \\ u_2^{(1)} & u_2^{(2)} & r_{23} & r_{24} & r_{25} \\ u_3^{(1)} & u_3^{(2)} & u_3^{(3)} & r_{34} & r_{35} \\ u_4^{(1)} & u_4^{(2)} & u_4^{(3)} & u_4^{(4)} & r_{45} \\ u_5^{(1)} & u_5^{(2)} & u_5^{(3)} & u_5^{(4)} & u_5^{(5)} \\ u_6^{(1)} & u_6^{(2)} & u_6^{(3)} & u_6^{(4)} & u_6^{(5)} \end{bmatrix}, \ \mathbf{r} = \begin{bmatrix} r_{11} \\ r_{22} \\ r_{33} \\ r_{44} \\ r_{55} \end{bmatrix}, \ \mathbf{b} = \begin{bmatrix} \rho_1 \\ \rho_2 \\ \rho_3 \\ \rho_4 \\ \rho_5 \end{bmatrix}$$

We can use the stored $\mathbf{u}$ vectors to reconstruct the Q matrix using the fact that $Q^T = H_p H_{p-1}, \ldots, H_1$. A pseudocode for doing this is the following:

Algorithm QR2: (*To reconstruct Q*)

```
$Q \leftarrow I(n,\ n)$
for  $t = p : 1$
        $\mathbf{u} \leftarrow X(t : n, t)$
        for  $j = t : n$
                $\mathbf{y} \leftarrow Q(t : n, j)$
                $c \leftarrow \mathbf{ddot}(\mathbf{u}, \mathbf{y})$
                $Q(t : n, j) \leftarrow \mathbf{daxpy}(-c/b(t), \mathbf{u}, \mathbf{y})$
        end
end
```

To solve the regression problem all that is needed is to append the dependent variable $n \times 1$ vector $\mathbf{y}$ to the right of X and compute the QR decomposition of $\tilde{X} = (X,\ \mathbf{y})$, as outlined earlier. The parameter estimates $\hat{\boldsymbol{\beta}} = R^{-1}\mathbf{z}$ are obtained by *back substitution.* In the following algorithm it is assumed that $\tilde{X}$ is an $n \times p$ matrix; thus $\hat{\boldsymbol{\beta}}$ is a $(p - 1) \times 1$ vector.

Algorithm QR3: *(Back Substitution)*

```
beta$(p-1) = X(p-1,p)/r(p-1)$
for $i = p-2 : 1$
        beta$(i) = (X(i,p) - \sum_{j=i+1}^{p-1} X(i,j) * \text{beta}(j))/r(i)$
end
```

The sequential sums of squares of fitting each parameter to the regression model are given by $[X(i,p)]^2, i = 1,\ldots,p-1$ respectively, and $s^2 = [r(p)]^2$.

In the discussion so far, we have assumed that X is of full column rank p. However, if X has rank k which is less than p, then there is a permutation matrix P such that if $XP = (X_1,\ X_2)$ where the k columns of X_1 are linearly independent. Hence the triangular part R of the QR decomposition of XP has the form

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix}$$

where R_{11} and R_{12} are unique. Since, in practice P is not known, column interchanges of X are made so that when the next transformation matrix H_t is constructed the column of X which has the largest norm is used. When this norm falls below a preset tolerance value, the rank of X is taken to be the number of columns so far pivoted. Since repeated computations of norms is expensive, alternative methods have been devised. One such method incorporated in the LINPACK routine is to compute the norms of the columns of X once initially, and then update these after each transformation instead of recomputing them. (See pp. 9.16–9.17 of LINPACK User's Guide for details).

Gaussian Elimination and the LU Decompositon

Orthogonal transformations are not the only means for introducing zeros below the diagonal of a matrix. For example, if A is a 4×4 matrix and writing $A^{(0)} = A$, it is seen assuming $a_{11} \neq 0$, that

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ -a_{21}/a_{11} & 1 & 0 & 0 \\ -a_{31}/a_{11} & 0 & 1 & 0 \\ -a_{41}/a_{11} & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_{11}^{(0)} & a_{12}^{(0)} & a_{13}^{(0)} & a_{14}^{(0)} \\ a_{21}^{(0)} & a_{22}^{(0)} & a_{23}^{(0)} & a_{24}^{(0)} \\ a_{31}^{(0)} & a_{32}^{(0)} & a_{33}^{(0)} & a_{34}^{(0)} \\ a_{41}^{(0)} & a_{42}^{(0)} & a_{43}^{(0)} & a_{44}^{(0)} \end{pmatrix} = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & a_{14}^{(1)} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} & a_{24}^{(1)} \\ 0 & a_{32}^{(1)} & a_{33}^{(1)} & a_{34}^{(1)} \\ 0 & a_{42}^{(1)} & a_{43}^{(1)} & a_{44}^{(1)} \end{pmatrix}$$

i.e.,

$$M_1 A^{(0)} = A^{(1)}.$$

Multiplying by M_1 effectively eliminates the elements in the first column of A, with the exception of the first element, by adding multiples of the first row to the other rows. It is important to note that the elements of the first row did not change. Next, perform the second step:

$$M_2 A^{(1)} = A^{(2)}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -a_{32}^{(1)}/a_{22}^{(1)} & 1 & 0 \\ 0 & -a_{42}^{(1)}/a_{22}^{(1)} & 0 & 1 \end{pmatrix} \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & a_{14}^{(1)} \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} & a_{24}^{(1)} \\ 0 & a_{32}^{(1)} & a_{33}^{(1)} & a_{34}^{(1)} \\ 0 & a_{42}^{(1)} & a_{43}^{(1)} & a_{44}^{(1)} \end{pmatrix} = \begin{pmatrix} a_{11}^{(2)} & a_{12}^{(2)} & a_{13}^{(2)} & a_{14}^{(2)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & a_{24}^{(2)} \\ 0 & 0 & a_{33}^{(2)} & a_{34}^{(2)} \\ 0 & 0 & a_{43}^{(2)} & a_{44}^{(2)} \end{pmatrix}$$

In general, for an $n \times n$ matrix A, M_k is of the form

$$M_k = I_n - \boldsymbol{\alpha}^{(k)}\, \mathbf{e}_k^T$$

where the column vector $\boldsymbol{\alpha}^{(k)}$ of multipliers is of the form $\boldsymbol{\alpha}^{(k)} = (0, \ldots, 0, \alpha_{k+1}, \ldots, \alpha_n)$ and $\alpha_i = a_{ik}^{(k-1)} \Big/ a_{kk}^{(k-1)}$ for $i = k+1, \ldots, n$. The matrix M_k is called an *elementary transformation.*

Continuing the above process, a sequence of matrices $A^{(k)}$ are produced, the last of which $A^{(n-1)}$ is upper triangular. It is very important to note that we need $a_{kk}^{(k-1)} \neq 0$ for every $k = 1, \ldots, n-1$. The a_{kk} are referred to as the *pivots* and their relative magnitude play an important role in the accuracy of computational algorithms using Gaussian elimination.

Suppose that our interest is in solving the linear system $A\mathbf{x} = \mathbf{b}$ and suppose that elementary transformations are performed such that

$$M_{n-1} \cdots M_2 M_1 A = MA = U$$

is upper triangular. The problem of solving $A\mathbf{x} = \mathbf{b}$ is now equivalent to solving

$$U\mathbf{x} = M\mathbf{b}$$

which can be solved easily via back substitution, since U is upper triangular.

Example: If

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 11 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

and

$$M_1 = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -3 & 0 & 1 \end{bmatrix}, \; M_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{bmatrix},$$

then $(M_1 M_2 A)\mathbf{x} = M\mathbf{b}$ results in the equation

$$\begin{bmatrix} 1 & 4 & 7 \\ 0 & -3 & -6 \\ 0 & 0 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix},$$

which is solved using *back substitution* giving $\mathbf{x} = (-1/3,\, 1/3,\, 0)^T$. □

Noting that $M_k = I_n - \boldsymbol{\alpha}^{(k)}\mathbf{e}_k^T$ and therefore its inverse is given by $M_k^{-1} = I_n + \boldsymbol{\alpha}^{(k)}\mathbf{e}_k^T$, we have that

$$\begin{aligned} A &= M_1^{-1} M_2^{-1} \ldots M_{n-1}^{-1} U \\ &= LU \end{aligned}$$

where $L = M_1^{-1} \dots M_{n-1}^{-1} = M^{-1}$. It is clear that L is a unit lower triangular matrix because each M_k^{-1} is unit lower triangular. This factorization is called the *LU factorization* of A.

Example:

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 11 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 4 & 7 \\ 0 & -3 & -6 \\ 0 & 0 & 2 \end{bmatrix}$$

□

The LU factorization does not always exist since if a zero pivot is encountered the corresponding M_k is not defined. The following theorem gives conditions for its existence.

Theorem:

Let $A_k,\ k = 1, \dots, n-1$ denote the $k \times k$ leading submatrices of an $n \times n$ matrix A. Then A has an LU factorization if $det(A_k) \neq 0$ for $k = 1, \dots, n-1$. If the LU factorization exists and A is nonsingular, then the LU factorization is unique and $\det(A) = u_{11}, \dots, u_{nn}$. □

In practice, entries in A can be overwritten with corresponding entries of L and U as they are produced during the course of the factorizations. Thus a 4×4 matrix will look like

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ \ell_{21} & u_{22} & u_{23} & u_{24} \\ \ell_{31} & \ell_{32} & u_{33} & u_{34} \\ \ell_{41} & \ell_{42} & \ell_{43} & \ell_{44} \end{bmatrix}$$

The LU factorization can be used to solve the linear system $A\mathbf{x} = \mathbf{b}$ by first solving the triangular system $L\mathbf{y} = \mathbf{b}$ for $\mathbf{y}$ using *forward substitution*, and then solving the triangular system $U\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$ using *back substitution.*

Example:

Using this representation, the matrix A in the previous example is factored as

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & -3 & -6 \\ 3 & 2 & 2 \end{bmatrix}$$

We then first solve $L\mathbf{y} = \mathbf{b}$ for $\mathbf{y}$ using forward substitution:

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

giving $\mathbf{y} = (1,\ -1,\ 0)^T$ and then solve $U\mathbf{x} = \mathbf{y}$ by back substitution as before, giving $\mathbf{x} = (-1/3,\ 1/3,\ 0)^T$.

Computational Aspects of the LU Factorization

Pseudocode for a version of the factorization which is the classical formulation of *Gaussian elimination* is provided below.

Algorithm LU1:

```
for $k = 1 : n-1$
    If $A(k,k) = 0$ then return
    Else
    $\mathbf{w} \leftarrow A(k, k+1:n)$
    for $i = k+1 : n$
        $\delta \leftarrow A(i,k)/A(k,k)$
        $A(i,k) \leftarrow \delta$
        for$j = k+1 : n$
            $A(i,j) \leftarrow A(i,j) - \delta\, w(j)$
        end
    end
end
```

An error analysis of this algorithm shows that the algorithm must be modified to interchange rows to avoid using relatively small pivots. The following example illustrates the difficulty.

$$A = \begin{bmatrix} .0001 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 10,000 & 1 \end{bmatrix} \begin{bmatrix} .0001 & 1 \\ 0 & -9999 \end{bmatrix} = LU$$

Using the permutation matrix

$$P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

leads to a more stable factorization:

$$PA = \begin{bmatrix} 1 & 1 \\ .0001 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ .0001 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & .9999 \end{bmatrix} = LU$$

A permutation matrix is just an identity matrix with rows permuted, e.g.,

$$P = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

In practice, an $n \times 1$ vector which stores the column indices of the nonzero values in each row is used to represent an $n \times n$ permutation matrix; in the above example this index vector is $p = (4\ 1\ 3\ 2\ )$. Note that P is orthogonal so that $P^{-1} = P^T$. A method prescribed for determining the order of pivots in Gaussian elimination is known as a *pivotal strategy.* One such strategy we shall discuss is called *partial pivoting.* In partial pivoting each column is searched for its largest (in magnitude) entry and the current row swapped with the row containing the largest element. The complete method attempts to find permutation $P_1, \ldots, P_{n-1}$ and Gauss transformations $M_1, \ldots, M_{n-1}$ such that

$$M_{n-1}P_{n-1} \cdots M_1 P_1 A = U$$

is upper triangular.

The Cholesky Factorization

If the $n \times n$ matrix A is symmetric positive definite, then there exists a unique upper triangular matrix R with positive diagonal elements such that

$$A = R^T R \ .$$

The Cholesky factor R is sometimes called the square root of A and the factorization, the square root factorization. The factorization can also be presented as $A = LL^T$, but we shall adopt the first definition in order to be consistent with the notation used in LINPACK. The algorithm for computing the Cholesky factorization is derived through a straightforward application of mathematical induction and it is informative to examine this derivation induction.

The induction begins with $k = 1$. The factorization is

$$A^{(1)} = R^{(1)} R^{(1)}$$

with $A^{(1)}$ being the 1×1 matrix A_{11} . This gives $R^{(1)} = R_{11} = \sqrt{A_{11}}$. Then the factorization is assumed to exist in the $(k-1)^{th}$ step, i.e., the factorization of the $(k-1) \times (k-1)$ matrix $A^{(k-1)}$ is

$$A^{(k-1)} = (R^{(k-1)})^T \ R^{(k-1)}$$

where $R^{(k-1)}$ is $k-1 \times k-1$ upper triangular. The k^{th} step is now constructed to factorize the $k \times k$ matrix $A^{(k)}$ by finding a new column to update $R^{(k-1)}$:

$$A^{(k)} = (R^{(k)})^T R^{(k)}$$

$$\begin{pmatrix} A^{(k-1)} & \mathbf{a}^{(k)} \\ (\mathbf{a}^{(k)})^T & a_{kk} \end{pmatrix} = \begin{pmatrix} R^{(k-1)} & \mathbf{r}^{(k)} \\ \mathbf{0}^T & r_{kk} \end{pmatrix}^T \begin{pmatrix} R^{(k-1)} & \mathbf{r}^{(k)} \\ \mathbf{0}^T & r_{kk} \end{pmatrix}$$

Assuming $R^{(k-1)}$ is known from the $(k-1)^{st}$ step, the above equality gives 2 equations to solve to obtain $\mathbf{r}^{(k)}$ and r_{kk} :

$$\begin{aligned} a_{kk} &= (\mathbf{r}^{(k)})^T \mathbf{r}^{(k)} + r_{kk}^2 \\ \mathbf{a}^{(k)} &= (R^{(k-1)})^T \mathbf{r}^{(k)} \end{aligned}$$

As a result, step k consists of using forward substitution to solve a lower triangular system for $\mathbf{r}^{(k)}$ and then obtaining r_{kk} by computing the square root of $(a_{kk} - (\mathbf{r}^{(k)})^T \mathbf{r}(k))$. Whether this quantity is positive serves as a test of the positive definiteness of $A^{(k)}$. If these two equations are satisfied and the factorization exists at the $(k-1)^{st}$ step, then the factorization exists at the k^{th} step, and by induction the algorithm is proved.

Algorithm CHOL:

The following is the algorithm given in LINPACK and is a *column version* of the Cholesky decomposition. This assumes that R does not overwrite A.

```
for $j = 1 : n$
    for $k = 1 : j - 1$
        $R(k, j) \leftarrow \left( A(k, j) - \sum_{i=1}^{k-1} R(i, k) R(i, j) \right) / R(k, k)$
    end
    $R(j, j) \leftarrow \left( A(j, j) - \sum_{k=1}^{j-1} R(k, j)^2 \right)^{1/2}$
end
```

To solve the linear system $A\mathbf{x} = \mathbf{b}$ where A is a symmetric positive definite matrix , factorize $A = R^T R$ and first solve the lower triangular system

$$R^T \mathbf{y} = \mathbf{b}$$

using forward substitution, and then solve the upper triangular system

$$R\mathbf{x} = \mathbf{y}$$

by back substitution.

The determinant of A is computed using

$$\det(A) = \Pi_{k=1}^{n} R(k, k)^2$$

and the inverse using

$$A^{-1} = R^{-1} (R^{-1})^T$$

The Cholesky decomposition can be applied directly to the $X'X$ matrix to solve the regression problem. However, if $\mathbf{y}$ is appended to the X matrix to form $\tilde{X} = (X,\ \mathbf{y})$, then the method of solution is straightforward. Apply the Cholesky decomposition to

$$S = \tilde{X}^T \tilde{X} = \begin{bmatrix} X'X & X^T\mathbf{y} \\ \mathbf{y}^T X & \mathbf{y}^T\mathbf{y} \end{bmatrix}$$

to obtain the upper triangular matrix

$$\begin{bmatrix} T & \mathbf{t} \\ & s \end{bmatrix} .$$

Using this system, the necessary regression computations can be carried out as follows: compute $\hat{\boldsymbol{\beta}} = T^{-1}\mathbf{t}$ by back substitution, regression sum of squares by $\mathbf{t}^T\mathbf{t}$, $SSE = s^2$ and $(X'X)^{-1} = T^{-1}(T^{-1})^T$.

The QR decomposition and the Cholesky decomposition are closely related. From the QR decompostion

$$Q^T X = \begin{bmatrix} R \\ 0 \end{bmatrix} \qquad (16)$$

and the fact that Q is orthogonal, it follows that

$$X^T X = R^T R.$$

Thus, if R is chosen to have positive diagonal elements, it is a Cholesky factor of $X^T X$. If X has linearly independent columns, $X^T X$ is positive definite and R is unique, along with $Q_1 = XR^{-1}$. The matrix R_{11} is the Cholesky factor of $X_1^T X_1$, which is the leading principal submatrix of order k of $X^T X$.

Finally, an accounting of the work in Cholesky factorization is of interest. The factoring of an $n \times n$ matrix A into $R^T R$ requires back-solving a triangular system of size (k-1) at each step k, (k-1) multiplies and a square root for each diagonal element. Thus the total is $n^3/6 + O(n^2)$ flops plus n square roots which are each $O(n^2)$. Since the LU factorization is $n^3/3 + O(n^2)$, the reduction in work in Cholesky is by half as might be expected due to symmetry.