1 Cholesky Decomposition

Theorem: 1 A matrix $A$ is symmetric positive definite (spd) if and only if there is a unique lower triangular nonsingular matrix $L$, with positive diagonal entries, such that $A = LL^t$.

$A = LL^t$ is called the Cholesky decomposition of $A$, and $L$ is called the Cholesky factor of $A$. A simple algorithm for computing the Cholesky factor of an arbitrary spd matrix is found in the subroutine cholesky() in the Appendix. If the subroutine breaks down, the matrix passed to it was not spd.

2 Simulation

We construct a program for simulating groups of uncorrelated, as well as correlated data, and consider the test statistic for equality of means. It is observed that, when the data are correlated, this test statistic is far from the theoretical values assumed by an F-distribution.

An attempt was made to use the unif() and normal() subroutines from assignment 2 to generate normal (0,1) random numbers. Although these routines worked well for assignment 2, their use was abandoned after obtaining bad data values. A NAG routine was used instead to produce pseudo iid normal (0,1) random numbers. We now describe the technique used to generate correlated random numbers from the uncorrelated ones.

Suppose $X$ has mean $\theta$ variance $\Sigma$. Then consider $L$, which has

$$E(LX) = L\theta \quad \text{and} \quad V(LX) = L\Sigma L^t$$

Similarly, suppose $X$ has mean 0 and variance $I$. Then $\tilde{X} = LX$ has variance $\Sigma = LL^t$ So we can created correlated random variables with variance covariance structure $\Sigma$ by starting with uncorrelated random variables in $X$, taking the Cholesky decomposition $\Sigma = LL^t$, and then applying $L$ to $X$.

Section 5.2 of the appendix lists the programs which carry out the foregoing. There is also listed the output from a sample run.

3 Gram-Schmidt orthogonalization (and other QR decompositions)

Our task is to write and test a function for the Gram-Schmidt orthogonalization (GS) of an arbitrary $n \times p$ matrix. Being arbitrary, we cannot assume that the matrix is well-conditioned\(^2\), and it might seem desirable

\(^1\)A proof can be found in [1]

\(^2\)We call a problem well-conditioned (ill-conditioned) if it has a small (large) condition number. In the present context, the condition number of the matrix $X$ is $\|X\|_2\|X^{-1}\|_2$, and in general, the condition number can be defined heuristically as the inverse of the “distance” to the nearest singular problem. Consult, e.g., [1] for further details.
for our program to produce a warning when the data are ill-conditioned. However, we hold that this is unnecessary for the following reasons: In practice, all matrices are of full rank due to round-off error, and there exist algorithms that guarantee a numerically stable orthogonalization. Therefore, given any matrix, a good QR decomposition routine should proceed to numerically stable results. On the other hand, one might argue that, when using such algorithms to compute parameter estimates for least squares, as we do in the following problem, suppressing warnings of ill-conditioned data is misleading. This is a weak argument for a number of reasons. If the user of least squares estimates believes that the columns of $X$ (covariates in the statistical setting) might be nearly linearly dependent, he should consider an SVD algorithm instead of GS. If he is too poor for SVD, then perhaps a rank revealing QR decomposition would be within the computing budget. The QR algorithm of this paper is backward stable\(^3\) and, using column pivoting, it is in some sense rank revealing in that small values on the diagonal of the resulting $R$ matrix indicate the degree of ill-conditioning.

Any QR decomposition that uses a Gram-Schmidt type algorithm is unstable and, when the data are ill-conditioned, the resulting $Q$ matrix may be far from orthogonal. Two QR decomposition procedures guarantee backward stability; one involves Givens rotations, and the other, Householder reflections. In this paper, we use the latter. Briefly, a Householder reflection (or transformation) $P = I - 2uu^T$ applied to $x$, is a reflection of $x$ through the plane perpendicular to $u$. It is easy to verify that $P$ is a symmetric orthogonal matrix. Why is this useful for QR? Given any vector $x$, we can find an Householder transformation $P$ such that $Px$ is a multiple of $e_1$, where $(e_1, e_2, \ldots)$ is the standard basis. Therefore, we orthogonalize the columns of any matrix $X$ by applying a sequence of Householder transformations $P_i$, where $P_i$ annihilates elements $x(i+1,i), x(i+2,i), \ldots$ of $X$. This results in an upper triangular\(^4\) matrix

\[
\tilde{R} \equiv P_p \cdots P_1 X
\]

Suppose $n > p$, and let $Q_1$ be the first $p$ columns of the $n \times n$ orthogonal matrix $P_1 \cdots P_p \equiv (Q_1 Q_2)$, and $R$ the first $p$ rows of the $n \times p$ matrix

\[
\tilde{R} \equiv \begin{pmatrix} R \\ O \end{pmatrix}
\]

Then,

\[
X = P_1 \cdots P_p \tilde{R} = Q_1 R
\]

is a QR decomposition of $X$.

We could take each column of $X$ in order, and perform the required Householder reflection, but pivoting can be used to make the algorithm more stable. Pivoting also has the nice side effect of producing an automatic estimate of the smallest singular value.\(^5\) Pivoting is introduced by simply selecting, of the remaining columns of $X$, the one with the largest norm as the next one to be reflected. To keep track of which vectors are where, we note that the column pivot is equivalent to right multiplication of $X$ by a permutation of the identity. Since a permutation of the identity has its own transpose for an inverse, transforming back to the original ordering is easy.

To construct the $Q$ and $R$ matrices explicitly ($Q$ is rarely required), we only need the Householder reflector vectors, $u$, and the upper triangle (trapezoid) of $\tilde{R}$. This is all that is returned (via the argument list) by the routine $\text{qr}$. The routine $\text{qr\_pivot}$ requires the user to pass an additional matrix, $E$, which will equal the permutation matrix on exit (since it is essential that we know what permutation of our matrix was really decomposed).

The core of the algorithm just described is performed by the functions $\text{qr}()$, $\text{qr\_pivot}()$, and $\text{House}()$, each of which appears in the appendix. In their current state, these functions use only Level 2 BLAS.

---

\(^3\)Informally, we say an algorithm is \textit{backward stable} if it produces the exact answer to a slightly wrong problem.

\(^4\)or, more accurately, upper trapezoidal when $n \neq p$

\(^5\)The smallest singular value provides useful information about ill-conditioning. It is the norm of the smallest perturbation that can lower the rank of $X$. 
However, it is possible to restructure the QR algorithm to take advantage of the Level 3 BLAS (as is done in the LAPACK routine sgeqrf).

## 4 Regression Using Householder Reflections

Given a vector $Y$ (the response variable) and a matrix $X = [1 \ x_1 \ldots \ x_p]$ (of intercept and covariates), we often make the following simplifying assumptions:

1. $Y = X\beta + \epsilon$
2. $\epsilon \sim N(0, \sigma^2 I)$

We estimate the coefficients $\beta$ with the usual OLS estimate given by the normal equations:

$$\hat{\beta} = (X^t X)^{-1} X^t Y$$

(2)

where $(X^t X)^{-1}$ is a generalized inverse of $X^t X$. The matrix $X^t X$ has the same rank as $X$, so if the our covariates are not *exactly* linearly dependent, then $(X^t X)^{-1} = (X^t X)^{-1}$.

If the data are known to be well-conditioned, the fastest method of solving for $\hat{\beta}$ proceeds by simply taking the Cholesky decomposition and solving the normal equations (equations (2) above). Briefly, we would proceed as follows:

1. Do a Cholesky decomposition of $X^t X = LL^t$.
2. Let $\theta = L^t \hat{\beta}$. So that $L\theta = X^t y$.
3. Solve the l.t. system for $\theta$.
4. Solve the l.t. system $L\hat{\beta} = \theta$ for $\hat{\beta}$.

However, that method is not very stable and can fail when the covariate matrix $X$ is nearly rank deficient. Therefore, we will use the more stable QR algorithm with pivoting. From equation (1), we then have $XE = Q_1 R$. Therefore, $X^t X = E(R^t R)^t$, and, using the fact that the permutation matrix $E$ is orthogonal, (2) becomes

$$\hat{\beta} = ER^{-1}Q_1^t Y$$

(3)

The residuals are then

$$e \equiv Y - X\hat{\beta}$$

$$= Y - (Q_1 R E^t)(ER^{-1}Q_1^t Y)$$

$$= (I - Q_1 Q_1^t)Y$$

Recalling that $(Q_1, Q_2)$ is an $n \times n$ orthogonal matrix, we see that $(I - Q_1 Q_1^t) = Q_2 Q_2^t$. Whence,

$$e = Q_2 Q_2^t Y$$

(4)

Notice that (3) and (4) are obtained from transformations of $Y$, namely $X$’s Householder transformations

$$P_p \cdots P_1 Y = \begin{pmatrix} Q_1^t Y \\ Q_2^t Y \end{pmatrix}$$

Our routine `qrpivot` returns the arguments $E$ and $R$, but not $(Q_1, Q_2)$. That is because construction of $(Q_1, Q_2)$ is unnecessary. Instead, recall that $P_j = I - 2u_j u_j^t$, so applying $(Q_1, Q_2)^t = P_p \cdots P_1$ to $Y$ is
equivalent to the following algorithm:

\[
\begin{align*}
&\text{for } j = 1 \text{ to } p \\
&\quad a = -2\sum_{i=j}^{n} u_{ij}Y_i \\
&\quad \text{for } i = j \text{ to } n \\
&\quad \quad Y_i = Y_i + au_{ij} \\
&\quad \text{end for} \\
&\text{end for}
\end{align*}
\]

For the regression problem, like many others, the \( u \) vectors are enough, and that is why the \( u \)'s, and not the \( Q \)'s, are returned by the QR routines.

The program \texttt{regress.c}, calls the functions \texttt{qrpivot()} and \texttt{reg()} to solve the least squares problem. The subroutine \texttt{reg} applies the orthogonal transformation \((Q_1 Q_2)^t\) to \( Y \) using the algorithm above, and applies \( ER^{-1} \) to the first \( p \) elements \((Q_1 Y)\) to find the solution \( \hat{\beta} \). Then, again using the algorithm above but with \( j \) descending from \( p \) to 1, \texttt{reg} applies \( Q_2 \) to the last \( n - p \) elements of \((Q_1 Q_2)^t Y\) by first annihilating its first \( p \) elements. More precisely,

\[
P_1 \cdots P_p \begin{pmatrix} O \\ Q_2^t Y \end{pmatrix} = (Q_1 Q_2) \begin{pmatrix} O \\ Q_2^t Y \end{pmatrix} = Q_2 Q_2^t Y
\]

By (4), this is the vector of residuals.

Finally, \texttt{regress.c} computes standard errors of the estimates. Note that equation (3) implies

\[
Var(\hat{\beta}) = Var(ER^{-1} Q_1^t Y) = (ER^{-1} Q_1^t) Var(Y) (ER^{-1} Q_1^t)^t = \sigma^2 ER^{-1} R^{-1} E^t
\]

which agrees with the usual formula under the i.i.d. normal assumption, \( \sigma^2 (X^t X)^{-1} \). So, to get standard error estimates, we only need the matrix \( ER^{-1} \) (which was derived when computing \( \hat{\beta} \)), and an estimate of \( \sigma^2 \). To get an estimate of \( \sigma^2 \) we note that the sum of squared residuals is

\[
\|Y - X \hat{\beta}\|^2 = Y^t (I - X (X^t X)^{-1} X^t) Y = Y^t (I - Q_1 Q_1^t) Y = Y^t e
\]

So we take as our estimate of \( \sigma^2 \) the mean squared error, \( Y^t e / (n - p) \).

A listing of the program \texttt{regress.c} and a test run using some wire bond strength data, appear in Appendix section 5.4.
5 Appendix

5.1 Cholesky Decomposition

A few lines of MATLAB can be used to produce a random spd matrix and compute the Cholesky factor:

```matlab
>> n=5;
>> A=randn(n);
>> % Generate a random nxn spd matrix:
>> A = A'*A;
>> % Write the matrix A to datafile:
>> fid = fopen('datafile','w');
>> fprintf(fid,'%f
',A);
>> fclose(fid);
>> % Perform Cholesky decomposition:
>> R = chol(A);
>> R'
```

```
ans =

1.5120  0  0  0  0
-0.1738  2.4651  0  0  0
-0.5637  0.5136  2.8050  0  0
-0.5603 -1.2035 -0.0434  2.1631  0
 0.2969  1.7911 -0.6201 -1.0762  0.2292
```

We test out our program `cholesky.c` on the same matrix, now stored in `datafile`.

```plaintext
% cholesky

Enter file name containing the spd matrix: datafile

Enter its dimension: 5

The Cholesky factor is:

```
L =
1.51199
-0.17384  2.46512
-0.56367  0.51358  2.80495
-0.56033 -1.20354 -0.04342  2.16306
 0.29689  1.79109 -0.62008 -1.07619  0.22917
```

The results are identical.

The listing for the main calling program `cholesky.c` is as follows:

```plaintext
/************************************************************
* cholesky.c main program for testing Cholesky             *
* decomposition routine cholesky()                         *
*                                                        *
* Created by William J. De Meo                            *
* on 11/29/97                                              *
***********************************************************/
```
```c
#include <stdlib.h>
#include <stdio.h>
#include "prototypes.h"
#define MAX_NAME 100

void cholesky(long N, double *A, double *diag);
void read_name(char *);

main()
{
  char *filename;
  double *A, *diag;
  long i, j, dim;

  filename = cmalloc(MAX_NAME);

  printf("Enter file name containing the spd matrix: ");
  read_name(filename);
  printf("Enter its dimension: ");
  scanf("%d", &dim);
  A = dmalloc(dim*dim);
  diag = dmalloc(dim);

  matlabread(A, dim, dim, filename);
  /*matrix is stored contiguously column-wise */

  cholesky(dim, A, diag);

  printf("The Cholesky factor is: \nL = \n");
  for (i = 0; i < dim; i++)
  {
    for (j = 0; j < i; j++)
      printf("%4.5lf \t", A[dim * j + i]);
    printf("%4.5lf", diag[i]);
    printf("\n");
  }
}

void read_name(char *name)
{
  int c, i = 0;

  while ((c = getchar()) != EOF && c != ' ' && c != '\n')
    name[i++] = c;
```
The listing for the subroutine *cholesky*() is as follows:

```c
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
cholesky.c
Created on 11/29/97 by William J. De Meo
Purpose: Cholesky decomposition of an n-by-n spd matrix
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
#include "prototypes.h"
#include <math.h>

/* Subroutine cholesky:

Arguments:

N  dimension of A

A
    on entry: the N by N matrix to be decomposed
    on exit: upper triangle is still A
       lower sub-triangle is the sub-triangle
       of the Cholesky factor L

diag
    on entry: an arbitrary vector of length N
    on exit: the diagonal of the Cholesky factor L

*/

void cholesky(long N, double *A, double *diag)
{
    long i,j,k;
    for(j=0;j<N;j++)
        diag[j] = A[N*j+j];
    for(j=0;j<N;j++)
    {
        for(k=0;k<j;k++)
            diag[j] -= A[N*k+j]*A[N*k+j];
        diag[j] = sqrt(diag[j]);
        for(i=j+1;i<N;i++)
        {
            for(k=0;k<i;k++)
            A[N*j+i] /= diag[j];
        }
    }
```
5.2 Simulation

A trial run of the program anova produced the following:

```
% anova

How many groups? 5
How many in group 1? 6
How many in group 2? 6
How many in group 3? 6
How many in group 4? 6
How many in group 5? 6
How many simulated F’s for this group structure? 10000
```

**UNCORRELATED DATA**

```
90th percentile: theoretical = 2.184240, observed = 2.196409
95th percentile: theoretical = 2.758710, observed = 2.797849
99th percentile: theoretical = 4.177420, observed = 4.273534
```

**CORRELATED DATA**

```
90th percentile: theoretical = 2.184240, observed = 14.493500
95th percentile: theoretical = 2.758710, observed = 18.310077
99th percentile: theoretical = 4.177420, observed = 29.474791
```

The output clearly indicates that when the data are correlated, the statistic no longer follows an F-distribution.

The program listing for `anova.c`, including the subroutines `F()` and `normal()` (even though I was forced to abandon `normal()`) appears below:

```c
/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * anova.c
 */
```
Statistics 243

Created by William J. De Meo
on 11/30/97

Purpose: Simulating F-statistics for one way ANOVA
using correlated and uncorrelated data

```
#include <math.h>
#include <stdio.h>
#include "prototypes.h"
/* NAG prototypes */
double g01fdf_(double *p, double *DF1, double *DF2, int *ifail);
void m01caf_(double *RV, long *M1, long *M2, char *ORDER, int *ifail);
void g05fdf_(double *mean,double *sd,long *n,double *g);
/* BLAS prototypes */
/* C <- (alpha)AB + (beta)C */
void dgemm_(char *TRANSA, char *TRANSB, long *M, long *N, long *K, double *alpha,
double *A, long *LDA, double *B, long *LDB, double *beta, double *C, long *LDC);
void F(long N, long *n, long k, double *AVE, double *ave, double *var, double *eff);
extern long I = (long)0;
main()
{
  char ORDER = 'A'; /* F stats will be sorted in ascending order */
  int ifail = 0;
  long k, i,j,p,q,r, N=(long)0, M = (long)12, numF,f, one = (long)1;
  double minimum=(double)1000, maximum=(double)-1000,P1,P2,P3,unit=(double)1,zero=(double)0;
  long *X, *n;
  /* BLAS arguments */
  double alpha = (double)1, beta = (double)0;
  char NOTRANS = 'N';
  AVE = dmalloc((long)1);
  VAR = dmalloc((long)1);
  gtemp = dmalloc((long)2);
  X = lmalloc((long)1);
  *X = time("\0");

  printf("\nHow many groups? ");
  scanf("%d",&k);
```
n = lmalloc(k); /* n vector stores number in each group */
ave = dmalloc(k);
var = dmalloc(k);

for(i=0;i<k;i++)
{
    printf("How many in group %d? ", i+1);
    scanf("%d",n+i);
    N+=n[i];
}

M = 2*N; /* unifs fail about 27% of the time, so take twice as many */
g = dmalloc(N+1); /* normals will be stored in g (N+1 in case N odd)*
u = dmalloc(M);

printf("How many simulated F's for this group structure? ");
scanf("%d",&numF);

eff = dmalloc(3*numF); /* matrix of F-stats with df's */
pureff = dmalloc(numF); /* vector of F-stats (without df's) */

for(f=0;f<numF;f++)
{
    I = (long)0;
    for(r=0;r<M;r++)
        u[r] = unif(X);

    i=(long)0;
p=(long)0;

    for(j=0;j<k;j++)
    {
        /*Forced to scrap the following broken random number generating code */
        /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
        if(0)
        {
            while(i<n[j])
            {
                if(normal(u, M, gtemp)==1)
                {
                    g[p+1] = gtemp[0]; i++;
g[p+1] = gtemp[1]; i++;
                }
                else /* didn't get enough normals -- need new uniforms */
                {
                    printf("Generating different unif(0,1) variables...\n\n");
                    I = 0;
                    for(r=0;r<M;r++)
                        u[r] = unif(X);
if(i==n[j])/* i.e. even number in group */
   i=(long)0;
else i=(long)1; /* i.e. odd number in group */
/*then give last random number to the next group */

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/* NAG routine puts normal(0,1) random numbers in g */
g05fdf_(&zero,&unit,n+j,g+p);
/* compute group averages and variances */
cmoment(g+p, n[j], ave+j, var+j);
/*printf("\nave[%d] = %lf, var[%d] = %lf\n\n",j+1,ave[j],j+1,var[j]);*/
p+=n[j];

cmoment(g,N,AVE,VAR); /* Get overall average */
/* finally compute the associated F-statistic */
F(N,n,k,AVE,ave,var,eff+3*f);

/* strip degrees of freedom from eff (so we can sort it) */
for(j=0;j<numF;j++)
   pureff[j] = eff[j*3];
/* sort the pureff vector of F statistics */
m01caf_(pureff, &one, &numF, &ORDER, &ifail);
printf("\n\nUNCORRELATED DATA\n");
printf("-------------------\n");
P1 = (double).9;
P1 = g01fdf_(&P1,eff+1,eff+2,&ifail);
printf("\n90th percetile: theoretical = %lf, observed = %lf\n",P1,pureff[(long)(.9*numF)]);
P2 = (double).95;
P2 = g01fdf_(&P2,eff+1,eff+2,&ifail);
printf("\n95th percetile: theoretical = %lf, observed = %lf\n",P2,pureff[(long)(.95*numF)]);
P3 = (double).99;
P3 = g01fdf_(&P3,eff+1,eff+2,&ifail);
printf("\n99th percetile: theoretical = %lf, observed = %lf\n",P3,pureff[(long)(.99*numF)]);

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 Correlated data
 WARNING: This part only written for non-ragged arrays
 i.e. all k group sizes must be equal*/
/* Test to see that all group sizes equal */
for (j=0; j<k; j++)
  if (n[0] != n[j])
  {
    printf("\n\nGroups of different sizes\n");
    printf("\nCan't perform correlation simulations.\n");
    exit(0);
  }

/* now just refer to group size as *n */

sig = dmalloc(*n * *n);
diag = dmalloc(*n);
work = dmalloc(N);

for (f=0; f<numF; f++)
{
  /* NAG routine puts normal(0,1) random numbers in g */
  g05fdf_(&zero, &unit, &N, g);

  for (j=0; j<*n; j++)
    for (i=j; i<*n; i++) /* only need lower triangle of symmetric matrix */
      sig[*n * j + i] = pow(0.7, (i-j));

  for (i=0; i<*n; i++)
    for (j=i+1; j<*n; j++)
      sig[*n * j + i] = 0;

  cholesky(*n, sig, diag);
  for (j=0; j<*n; j++) sig[*n * j + j] = diag[j];
  free(diag);

  /*work <- sig*g */
  dgemm_(&NOTRANS, &NOTRANS, n, &k, n, &alpha, sig, n, g, n, &beta, work, n);
  for (j=0; j<N; j++) g[j] = work[j];

  /* compute group averages and variances */
  p=(long)0;
  for (j=0; j<k; j++)
  {
    cmoment(g+p, n[j], ave+j, var+j);
    p+=n[j];
  }
  /*printf("\nave[%d] = %lf, var[%d] = %lf\n\n",j+1,ave[j],j+1,var[j]);*/
  cmoment(g, N, AVE, VAR); /* Get overall average */

  /* finally compute the associated F-statistic */
  F(N, n, k, AVE, ave, var, eff+3*f);
Statistics 243

} /* strip degrees of freedom from eff (so we can sort it) */
for(j=0;j<numF;j++)
    pureff[j] = eff[j*3];

/* sort the pureff vector of F statistics */
m01caf_(pureff, &one, &numF, &ORDER, &ifail);
printf("\n\nCORRELATED DATA\n");
printf("\n90th percetile: theoretical = %lf, observed = %lf\n",P1,pureff[(long)(.9*numF)]);
printf("\n95th percetile: theoretical = %lf, observed = %lf\n",P2,pureff[(long)(.95*numF)]);
printf("\n99th percetile: theoretical = %lf, observed = %lf\n",P3,pureff[(long)(.99*numF)]);
}

/* subroutine F() ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Purpose: Compute test statistic for equality of means
Arguments:
    N  total number of observations
    n  a vector of length k where the ith element
        contains the number of observations in the ith group
    k  the number of groups
    AVE average of all observations
    ave a vector of length k where the ith element
        contains the average of the ith group of observations
    var a vector of length k where the ith element
        contains the empirical variance (mse) of the ith group of observations
    F  on entry: a vector of length 3
        on exit: first element is the F-statistic
        second element is k-1, third element is N-k
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

void F(long N, long *n, long k, double *AVE, double *ave, double *var, double *F)
{
    long i,j;
    double den=(double)0,num=(double)0;

for(i=0;i<k;i++)
{
    num += n[i]*(ave[i]-*AVE)*(ave[i]-*AVE)/F[1];
    den += (n[i]-1)*var[i]/F[2];
}
F[0] = num/den;

#include <math.h>

int normal(double *u, long n, double *x)
{
    static int numcount = (long)0, dencount = (long)0;
    int count = (long)0;
    double s;
    extern long I;
    dencount++;
    do{
        /* The following lines display the proportion of times we are rejecting
        s, which we expect to be around .27
        if(count>0)
        {
        */

numcount++;  
printf("\n\n s = %lf, prop = %lf\n\n",
    s, (double)numcount/(double)dencount);
}
*/
if(I > n-2)
    return(0); /* set external I back to 0 in main() */
s = (2*u[I] - 1)*(2*u[I] - 1) + (2*u[I+1] - 1)*(2*u[I+1] - 1);
I += 2;
count++;
}while(s >= (double)1);

x[0] = (2*u[I] - 1)*sqrt(-2*log(s)/s);
x[1] = (2*u[I+1] - 1)*sqrt(-2*log(s)/s);

return(1);
}

5.3 QR Decomposition

Using the MATLAB program tests.m (listed below), we produce a data file called datafile containing a random 4 × 3 matrix with condition number 100. The matrix is stored column-wise by MATLAB, so datafile contains the following:

-55.201723
-40.707641
42.683791
16.618428
-35.914110
-24.880338
28.457431
15.289218
20.047960
19.983140
-6.967239
75.587311

That is, the matrix to be decomposed is:

\[ A = \begin{pmatrix} -55.201723 & -35.914110 & 20.047960 \\ -40.707641 & -24.880338 & 19.983140 \\ 42.683791 & 28.457431 & -6.967239 \\ 16.618428 & 15.289218 & 75.587311 \end{pmatrix} \]

We can use the program QR (which calls the subroutine qr()) to decompose A, as the following output demonstrates:

% QR
Enter file name containing the matrix: datafile

Enter the number of rows: 4

Enter the number of columns: 3

The orthogonalization produced:
82.47679 54.12546 -11.65654
-0.27012 -4.78269 -77.59750
0.28324 0.06872 20.14904
0.11027 0.55963 0.62361

with leading u’s:
-0.91359 0.82589 -0.78174

Running the program QRpiv (which calls the subroutine qrpivot()) on the same matrix produces:

% QRpiv

Enter file name containing the matrix: datafile

Enter the number of rows: 4

Enter the number of columns: 3

The orthogonalization produced:
82.47679 -11.65654 54.12546
-0.27012 -80.17079 -4.62918
0.28324 0.02372 -1.20202
0.11027 0.65823 -0.59810

With permutation matrix:
1.00000 0.00000 0.00000
0.00000 0.00000 1.00000
0.00000 1.00000 0.00000

and leading u’s:
-0.91359 0.75244 0.80142

We can check that these results are accurate using MATLAB (which also performs QR using Householder reflections). The QR portion of the MATLAB program tests.m produces the following:

>> % Perform QR without pivoting:
>> [Q,R] = qr(A);
>> R

R =

82.4768 54.1255 -11.6565
Statistics 243

0  -4.7827  -77.5975
0   0     20.1490
0   0     0

>> % Perform QR with pivoting:
>> [Qpiv, Rpiv, E] = qr(A);
>> Rpiv

Rpiv =

82.4768   -11.6565   54.1255
   0  -80.1708   -4.6292
   0     0    -1.2020
   0     0       0

>> E

E =

 1    0    0
 0    0    1
 0    1    0

We see that the only difference in the results is that our subroutines are storing the u’s in the lower part of the R matrix.

The MATLAB program listing for tests.m is as follows:

% MATLAB code tests.m
% Created by William J. De Meo
% on 11/28/97
% Purpose: Perform QR decomposition (with and without pivoting)
% on a random matrix of user specified dimension and
% condition number

% Inputs:
% m, n = numbers of rows, columns in test matrices
% m should be at least n
% cnd = condition number of test matrices to generate
% (ratio of largest to smallest singular value)
% cnd should be at least 1

% Outputs:
% datafile = A file containing the matrix tested
% (so that we can run QR and QRpiv on the same matrix)
% R = the R from the QR decomposition of A
% Rpiv, E = the R and permutation matrix E from QR decomposition
% of A with pivoting
%
% Generate random matrix A, starting with the SVD of a random matrix
A=randn(m,n);
[u,s,v]=svd(A);
% Let singular values range from 1 to cnd, with
% uniformly distributed logarithms
sd = [1, cnd, exp(rand(1,n-2)*log(cnd))];
s = diag(sd);
A=u(:,1:n)*s*v';
%
% Write the matrix A to datafile:
fid = fopen('datafile','w');
fprintf(fid,'%f
',A);
fclose(fid);

% Perform QR without pivoting:
[Q,R] = qr(A);
R
% Perform QR with pivoting:
[Qpiv, Rpiv, E] = qr(A);
Rpiv
E
%
% END tests.m

The following is the program listing for QRpiv.c.6

/*****************************
* QRpiv.c main program for testing QR *
* Orthogonalization routine qrpivot() *
* Created by William J. De Meo *
* on 11/23/97 *
* Note: differences between QRpiv.c and QR.c are marked *
* with the comment "QRpiv.c"
*******************************/

#include <stdlib.h>
#include <stdio.h>
#include "prototypes.h"
define MAX_NAME 100

6QR.c is not listed as it is almost identical to QRpiv.c. The only difference being that it calls qr() instead of qrpivot(). The differing lines are marked in QRpiv.c with the comment /* QRpiv */.
/ * QRpiv.c */
void qrpivot(long M, long N, double *A, double *E, double *leadu);

void read_name(char *);

main()
{
    char *filename;
    double *x, *leadu, *E;
    long i, j, nrow, ncol, mindim;

    filename = cmalloc(MAX_NAME);

    printf("Enter file name containing the matrix: ");
    read_name(filename);
    printf("Enter the number of rows: ");
    scanf("%u",&nrow);
    printf("Enter the number of columns: ");
    scanf("%u",&ncol);
    mindim = lmin(nrow,ncol); /* mindim is the smaller dimension */

    x = dmalloc(nrow*ncol);
    leadu = dmalloc(mindim);
    E = dmalloc(ncol*ncol);

    matlabread(x, nrow, ncol, filename);
    /*matrix is stored contiguously column-wise */

    /* Test qrpivot: */
    qrpivot(nrow,ncol,x,E,leadu); /* QRpiv.c */

    printf("The orthogonalization produced: \n");
    matprint(x,nrow,ncol);
    printf("With permutation matrix: \n"); /* QRpiv.c */
    matprint(E,ncol,ncol);
    printf("and leading u's:\n");
    for(i=0;i<mindim;i++)
        printf("%4.5lf \t", leadu[i]);
    printf("\n");
}

void read_name(char *name)
{
    int c, i = 0;

    while ((c = getchar()) != EOF && c != ' ' && c != '\n')
        name[i++] = c;
    name[i] = '\0';
The functions \texttt{qr()}, \texttt{qrpivot()}, and \texttt{House()} are found in \texttt{House.c}, which is listed below:

```c
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
House.c
Created on 11/12/97 by William J. De Meo
Last modified: 11/28/97
Purpose: QR decomposition of an m-by-n matrix using 
Householder reflections
Further Details: This implementation uses BLAS 2  
(matrix-vector mult. and rank 1 updates)
Dependencies: Requires subroutines found in the libraries:
sunperf, and blas
the later two are linked with the options:
-lsunperf -dalign -lblas
compilation must be done with the -dalign option
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
#include "prototypes.h"
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
BLAS Subroutine prototypes
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
double dnrm2_(long *N, double *x, long *INC);/* L2 norm of x*/
double dcopy_(long *N, double *X, long *INCX, double *Y, long *INCY); /* y <- x */
double dgemv_(char *TRANSA, long *M, long *N, double *alpha, double *A, long *LDA,
double *x, long *INCX, double *beta, double *y, long *INCY);
/* y <- (alpha)Ax + (beta)y (or A^t if TRANSA='T') */
void dger_(long *M,long *N,double *alpha,double *x,long *INCX,double *y,long *INCY,
double *A,long *LDA);/* Rank 1 update A <- (alpha)xy^t + A */
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/* Subroutine qr:*/
Arguments:
M (long) number of rows of A
N (long) number of columns of A
A (pointer to double)
on entry: the M by N matrix to be decomposed
on exit: upper-right-triangle = R
column i of lower trapezoid = a(i+1:M,i) = u(1:m,i)
where P(i) = I - 2 u(1:m,i)u^t(1:m,i) is the ith
```
Householder transformation (of dimension \((M-i)x(M-i)\))

\textbf{leadu} (pointer to double)

\begin{itemize}
  \item on entry: an arbitrary vector of length \(\min(M-1,N)\)
  \item on exit: the leading entries of the Householder vectors \(u(i)\)
    \begin{itemize}
      \item i.e. \(u(i) = (\text{leadu}(i), a(i+1:M,i))\) i=1,...,\(\min(M-1,N)\)
    \end{itemize}
\end{itemize}

Note that \(Q\) is obtained from augmenting the Householder transformations to be of proper dimensions, and then multiplying:

If \(P'(i)\) denotes augmented \(P(i)\),

\[ Q = P'(1) P'(2) P'(3) \ldots = \]

\[
\begin{vmatrix}
  1 & 0 & 0 & \ldots \\
  P(1) & 1 & 0 & \ldots \\
  0 & P(2) & 1 & \ldots \\
  0 & 0 & P(3) & \ldots \\
\end{vmatrix}
\]

But this is left to the calling function and is not performed in \texttt{qr()}.

```c
void qr(long M, long N, double *A, double *leadu)
{
    char T = 'T';
    long i, j, nrow, ncol, mindim;
    double unit, zero, alpha=(double)-2;
    double *u, *y; /* used for temporary work space */
    long INC=1; /* INC is used to represent storage spacing between elements */
    unit = (double)1; zero = (double)0;

    u = dmalloc(M); /* work space */
    y = dmalloc(N);

    mindim = lmin(M-1,N);

    for(i=0;i<mindim;i++)
    {
        nrow=M-i;
        ncol=N-i;

        House(nrow, A+(M*i+i), u);

        /* y <- (A^t)u (is working)*/
        dgemv_(&T, &nrow, &ncol, &unit, A+(M*i)+i,
               &M, u, &INC, &zero, y, &INC);

        /* Rank 1 update: A <- A + (-2)uy^t i.e. A - 2uu^tA */
    }
}
```
dger_(&nrow,&ncol, &alpha, u, &INC, y, &INC, A+(M*i)+i, &M);
leadu[i] = u[0];

/* store u(2:nrow) in A(i+1:M,i) */
nrow--;
dcopy_(&nrow, u+1, &INC, A+(M*i)+i+1,&INC);
}
free(u); free(y);

/* Subroutine qrpivot: */

Arguments: same as qr() with one exception:

E (pointer to double)
on entry: an arbitrary N by N matrix
on exit: the permutation matrix
The final decomposition is AE = QR
*

void qrpivot(long M, long N, double *A, double *E, double *leadu)
{
char T = 'T';
long i,j, nrow, ncol, mindim, perm=0;
double unit, zero, alpha=(double)-2, maxnorm, norm;
double *u, *y; /* used for temporary work space */
long INC=1; /* INC is used to represent storage
spacing between elements */
unit = (double)1; zero = (double)0;
u = dmalloc(M); /* work space */
y = dmalloc(N);
/* Start permutation matrix as the identity */
for(i=0;i<N;i++){
    E[N*i+i] = (double)1;
}
mindim = lmin(M-1,N);

for(i=0;i<mindim;i++)
{
    nrow=M-i;
    ncol=N-i;

    /* column pivot */
    maxnorm=0;
    for(j=1;j<N;j++)
    {

norm = dnrm2_(&nrow,A+(M*j+i),&INC);
if(norm>maxnorm)
{
    perm=j;
    maxnorm=norm;
}
}

if(perm>i)
{ /* If the i'th column was not the largest in norm, permute cols
of A, and note it by swapping cols of pivot matrix*/
dswap_(&M,A+(M*i),&INC,A+(M*perm),&INC);
dswap_(&N,E+(N*i),&INC,E+(N*perm),&INC);
}

House(nrow, A+(M*i+i), u);

/* y <- (A^t)u */
dgemv_(&T, &nrow, &ncol, &unit, A+(M*i)+i,
     &M,u,&INC,&zero,y,&INC);

/* Rank 1 update: A <- A + (-2)uy^t i.e. A = 2uu^tA */
dger_(&nrow,&ncol, &alpha, u, &INC, y, &INC, A+(M+i)+i, &M);
leadu[i] = u[0];

/* store u(2:nrow) in A(i+1:M,i) */
nrow--;
dcopy_(&nrow, u+1, &INC, A+(M*i)+i+1,&INC);
}
free(u); free(y);

void House(long N, double *a, double *u)
{
    double sign = (double)1, norm;
    long inc=1;
    long i;

    if(a[0] <= (double)0) sign = -1;

    /* u <- A(i:M,i) */
dcopy_(&N, a, &inc, u, &inc);

    norm = dnrm2_(&N, a, &inc); /* L2 norm of a = A(i:M,i) */
    u[0] += sign*norm;
    norm = dnrm2_(&N, u, &inc); /* L2 norm of new u */
for(i=0;i<N;i++) u[i] /= norm;

} /* consider skipping this normalization and putting the norms in the coefficient:
   alpha = 2/(unorm * unorm) */

5.4 Regression Using Householder Reflections

The file datafile contains 30 observations of 5 variables related to wire strength (plus a column of 1’s for the intercept). The data were taken from a Stat 215a lab and can be found in /saruman/class/data/s215/fall97/lab3.data. Using the regress program, we can estimates the OLS coefficients and compute the statistics of interest as follows:

% regress

Enter file name containing the matrix [X,y] (with intercept column included if desired): datafile

Enter the number of observations: 30

Enter the number of parameters (including intercept): 5

Rough estimate of smallest singular value of X:
R(5,5) = 0.393750

MSE = 26.604747

<table>
<thead>
<tr>
<th>COEFFICIENT</th>
<th>SE</th>
</tr>
</thead>
<tbody>
<tr>
<td>-37.47667</td>
<td>13.09964</td>
</tr>
<tr>
<td>0.21167</td>
<td>0.21057</td>
</tr>
<tr>
<td>0.49833</td>
<td>0.07019</td>
</tr>
<tr>
<td>0.12967</td>
<td>0.04211</td>
</tr>
<tr>
<td>0.25833</td>
<td>0.21057</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OBS</th>
<th>RESIDUAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.86000</td>
</tr>
<tr>
<td>2</td>
<td>-1.15667</td>
</tr>
<tr>
<td>3</td>
<td>-0.49000</td>
</tr>
<tr>
<td>4</td>
<td>-2.70667</td>
</tr>
<tr>
<td>5</td>
<td>6.77667</td>
</tr>
<tr>
<td>6</td>
<td>1.56000</td>
</tr>
<tr>
<td>7</td>
<td>2.02667</td>
</tr>
<tr>
<td>8</td>
<td>-1.109000</td>
</tr>
<tr>
<td>9</td>
<td>-1.32333</td>
</tr>
<tr>
<td>10</td>
<td>-6.64000</td>
</tr>
<tr>
<td>11</td>
<td>-4.27333</td>
</tr>
<tr>
<td>12</td>
<td>7.11000</td>
</tr>
</tbody>
</table>
The upper bound on the smallest singular value given by $R(5,5) = 0.393750$ indicates that $X$ is at most a distance of $0.393750$ from the nearest rank deficient matrix. Perhaps one of our variables is nearly constant and we don’t need an intercept. The \texttt{regress} program handles this, requiring only that we modify the input file. The new data file, \texttt{newdata}, doesn’t contain a column of 1’s. The new results are as follows:

\begin{verbatim}

% regress

Enter file name containing the matrix [X,y] (with intercept column included if desired): newdata

Enter the number of observations: 30

Enter the number of parameters (including intercept): 4

Rough estimate of smallest singular value of X:
$R(4,4) = 25.868210$

MSE = 33.956565

\begin{tabular}{ll}
COEFFICIENT & SE \\
-0.12727 & 0.19667 \\
0.41763 & 0.07261 \\
0.05220 & 0.03644 \\
0.06466 & 0.22527 \\
\end{tabular}

\begin{tabular}{ll}
OBS & RESIDUAL \\
1 & -5.14401 \\
2 & -3.77132 \\
3 & -4.07304 \\
\end{tabular}

\end{verbatim}
That looks better. A quick check can be made using one line of MATLAB:

```matlab
>> [B,se] = lscov(X,Y,eye(30))
```

```
B =

-0.1273
  0.4176
  0.0522
  0.0647

se =

  0.1967
  0.0726
  0.0364
  0.2253
```

The results are identical (not a coincidence – MATLAB least squares functions also use Householder QR with pivoting).

The following is the program listing for `regress.c`: 

```c
```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "prototypes.h"
#define MAX_NAME 100

void dcopy_(long *N, double *X, long *INCX, double *Y, long *INCY); /* y <- x */
double ddot_(long *N, double *X, long *INCX, double *Y, long *INCY); /* returns x * y */
void dtrsv_(char *UPLO, char *TRANSA, char *DIAG, long *N, double *A,
long *LDA, double *Y, long *INCY); /* y <- inv(A)*y */
void dgemv_(char *TRANSA, long *M, long *N, double *alpha, double *A, long *LDA,
double *x, long *INCX, double *beta, double *y, long *INCY);/* C <- (alpha)Ax + (beta)C */
void dgemm_(char *TRANSA, char *TRANSB, long *M, long *N, long *K, double *alpha,
double *A, long *LDA, double *B, long *LDB, double *beta, double *C, long *LDC); /* B <- alpha*inv(A)*B */
void dtrsm_(char *SIDE, char *UPLO, char *TRANSA, char *DIAG, long *M, long *N,
double *alpha, double *A, long *LDA, double *B, long *LDB);

void reg(long M, long N, double *QR, double *leadu, double *E,
double *y, double *B, double *cov, double *se, double *e, double *sigma);

void read_name(char *);

main()
{
    char *filename;
    long i, j, nrow, ncol, mindim;

    filename = cmalloc(MAX_NAME);

    /* matrix must be of the form [X,y] where first column of X is a vector of 1's if an intercept term is desired */
printf("\n\s\n\s","Enter file name containing the matrix [X,y] ",
"(with intercept column included if desired): ");
read_name(filename);
printf("\n\Enter the number of observations: ");
scanf("%u",&nrow);
printf("\nEnter the number of parameters (including intercept): ");
scanf("%u",&ncol);
if(nrow <= ncol)
    printf("\n\nWARNING: \#obs = &d <= &d = \#parameters\n\n",nrow,ncol);
mindim = lmin(nrow,ncol); /* mindim is the smaller dimension */
x = dmalloc(nrow*(ncol+1));
y = x+(nrow*ncol); /* y is assigned the address of last col of x */
leadu = dmalloc(mindim);
E = dmalloc(ncol*ncol);
B = dmalloc(ncol);
cov = dmalloc(ncol*ncol);
se = dmalloc(ncol);
e = dmalloc(nrow);
sigma = dmalloc((long)1);
matread(x, nrow, ncol+1, filename);
/*matlabread(x, nrow, ncol, filename); */
/*matrix is stored contiguously column-wise */
qrpivot(nrow,ncol,x,E,leadu); /* only send first ncol columns of x */
printf("\nRough estimate of smallest singular value of X: ");
printf("\nR(%d,%d) = %lf",ncol,ncol,x[nrow*(ncol-1)+(ncol-1)]);
reg(nrow,ncol,x,leadu,E,y,B,cov,se,e,sigma);
printf("\n\nMSE = %lf\n",*sigma);
printf("\nCOEFFICIENT \t SE \n");
for (i = 0; i < ncol; i++)
    printf("%4.5lf \t %4.5lf\n", B[i],se[i]);
printf("\nOBS \t RESIDUAL\n");
for (i = 0; i < nrow; i++)
    printf("%d \t %4.5lf\n", i+1,e[i]);
}

/* Subroutine reg() */
Arguments:

M number of rows of X

N number of columns of X (expect N < M)
QR the matrix resulting from applying qrpivot() to X

leadu
  on entry: the vector of leading u's resulting from qrpivot()
  on exit: the vector of coefficient estimates B, where y = XB

E the permutation matrix resulting from qrpivot()

y a vector (length M) of "observables" (the rhs in XB = y)

B on entry: an arbitrary length N vector
  on exit: the coefficient estimates

cov on entry: an arbitrary NxN matrix
  on exit: the covariance matrix

se on entry: an arbitrary length N vector
  on exit: the s.e.'s of the coefficient estimates

e on entry: an arbitrary length M vector
  on exit: the vector of residuals: e = y - XB

sigma on exit: the mse = y^t e / (M-N)

*/

void reg(long M, long N, double *QR, double *leadu, double *E,
         double *y, double *B, double *cov, double *se, double *e, double *sigma)
{
  long i,j,mindim;

  /* BLAS arguments */
  long INC=(long)1;
  double alpha = (double)1, beta = (double)0;
  char UPLO, NOTRANS, TRANS, DIAG, SIDE;
  UPLO='U'; NOTRANS = 'N'; TRANS = 'T'; DIAG = 'N'; SIDE='L';

  dcopy_(&M, y, &INC, e, &INC); /* e <- y */
  mindim = lmin(M-1,N); /* expect mindim = N */

  /* Apply P(n)...P(1) to e to get e <- (Q_1 Q_2)^t Y*/
  for(j=0;j<mindim;j++)
    { a = leadu[j]*e[j]; /* initialize a = u(1)e(1) */
      for(i=j+1;i<M;i++)
        a += QR[M*j+i]*e[i]; /* a = u^t e */
a *= (double)(-2);
e[j] += a * leadu[j];    /* e(1) <- e(1) - 2 u(1)u^t e */
for(i=j+1;i<N;i++)
    e[i] += a* QR[M*j+i]; /* e <- e + (-2)uu^t e */

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
COMPUTE COEFFICIENTS
BLAS 3 method: (currently used method) */

    /* compute inv(R) */
invR = dmalloc(N*N); /* workspace */
for(j=0;j<N;j++) /* begin with identity matrix */
{
    for(i=0;i<N;i++)
        invR[N*j+i]=(double)0;
    invR[N*j+j]=(double)1;
}
    /* invR <- alpha*inv(R)*invR = alpha*inv(R)*eye */
dtrsm_(&SIDE, &UPLO, &NOTRANS, &DIAG, &N, &N, &alpha, QR, &M,invR,&N);

    /* compute the E*inv(R) matrix */
EiR = dmalloc(N*N);
    /* EiR <- (alpha)E*invR + (beta)EiR */
dgemm_(&NOTRANS, &NOTRANS, &N, &N, &N, &alpha, E, &N,
        invR, &N, &beta, EiR, &N);
free(invR);

    /* compute the coefficients */
dgemv_(&NOTRANS, &N, &N, &alpha, EiR, &N, e, &INC, &beta, B, &INC);
    /* B <- (alpha)EiR*e + (beta)B (beta = 0) only references 1st N elements of e */

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
COMPUTE COEFFICIENTS
Alternative method (BLAS 2): */

if(0) /* not currently used */
{
    coef = dmalloc(N); /* workspace */
dcopy_(&N, e, &INC, coef, &INC); /* coef <- e(1:N) */
dtrsv_(&UPLO, &NOTRANS, &DIAG, &N, QR, &M, coef, &INC); /* coef <- Inv(R)*coef */
dgemv_(&NOTRANS, &N, &N, &alpha, E, &N, coef, &INC, &beta, B, &INC);
    /* B <- (alpha)E*coef + (beta)B (beta = 0) */
free(coef);
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
COMPUTE RESIDUALS
*/
for(i=0;i<N;i++) e[i]=(double)0; /* annihilate first N elements of e */
/* Apply P(1)...P(n) to e to get e <- Q2 Q2^t Y*/
for(j=(mindim-1);j>=0;j--)
{
    a = leadu[j]*e[j]; /* initialize a = u(1)e(1) */
    for(i=j+1;i<M;i++)
        a += QR[M*j+i]*e[i]; /* a = u^t e */
    a *= (double)(-2);
    e[j] += a * leadu[j]; /* e(1) <- e(1) - 2 u(1)u^te */
    for(i=j+1;i<M;i++)
        e[i] += a* QR[M*j+i]; /* e <- e + (-2)uu^t e */
}
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
COMPUTE MSE
*/
*sigma = ddot_(&M, y, &INC, e, &INC);
*sigma /= (M - N);
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
COMPUTE COVARIANCE MATRIX and SE's
*/
cov <- (alpha)EiR*(EiR)' + (beta)cov */
dgemm_(&NOTRANS, &TRANS, &N, &N, &N, &alpha, EiR, &N, &beta, cov, &N);
for(j=0;j<N;j++)
    se[j] = sqrt((*sigma)*cov[N*j+j]);
}
void read_name(char *name)
{
    int c, i = 0;
    while ((c = getchar()) != EOD & c != ' ' & c != '
')
        name[i++] = c;
    name[i] = '\0';
}
References

