Statistics 243: *assignment 4*

William J. De Meo

August 8, 2011

1 Sweep

The program `sweep.c` contains the subroutine `sweep()`, which performs the sweep algorithm on a symmetric matrix (actually, the function `colsweep` does all the work).[1] The program is written for a symmetric matrix because that is what we find in applications like regression, and because foresight that the matrix is symmetric affords tremendous savings in storage and computational costs. The `sweep()` routine accesses only the upper triangle of its matrix argument, and overwrites only this portion of the matrix. Therefore, if an entire symmetric matrix is passed to `sweep()`, then on exit the same matrix has the answer in its upper triangle and the original matrix in the lower triangle. The diagonal entries are the negative reciprocals of the original diagonal entries.

Using MATLAB, we show in section 3.1 that, when sweeping all columns of a matrix, we get the negative of the inverse (or close to it). When sweeping the first five columns of the 6 by 6 matrix X^tX, where $X = [X : y]$ is an augmented matrix of covariates and response variable, we arrive at the matrix:

$$\begin{pmatrix} -(X^tX)^{-1} & \hat{\beta} \\ \hat{\beta}^t & y^t(I-P)y \end{pmatrix}$$

where $\hat{\beta} = (X^tX)^{-1}X^ty$ and $P \equiv X(X^tX)^{-1}X^t$ is the projection operator.

2 Non-linear Regression

2.1 Parameter Estimates

Write the non-linear regression model as:[2]

$$y_n = f(x_n, \theta) + z_n$$

where f is the expectation function and x_n is a vector of associated covariates for the nth observation. Further consider the N row vectors x_n, $(n = 1, \dots, N)$ as fixed and concentrate on the relation between the response y_n and the parameters θ. Thus, letting $\eta_n(\theta) = f(x_n, \theta)$, we have

$$Y = \eta(\theta) + Z$$

and we assume

$$E[Z] = 0, \;\; Var(Z) = E[ZZ^t] = \sigma^2 I$$

[1]The algorithm used is similar to that found in [2], p. 87

[2]For further details, consult Bates and Watts [1].

as in the linear model. We refer to $\eta(\theta)$ as the *expectation surface.* When $\theta \in \Theta \subset \mathbf{R}^p$, η is generally a p dimensional curved surface in the N dimensional covariate space.

Having set up the context, we seek the least squares estimate of θ. That is, we want $\hat{\theta} \in \Theta$ such that,

$$\|Y - \eta(\hat{\theta})\|_2 = \min_{\theta \in \Theta} \|Y - \eta(\theta)\|_2$$

This estimate is found by solving a system of non-linear equations, which requires the use of iterative procedures. The subroutine `gn()` applied in section 3.2 employs the familiar Gauss-Newton procedure, which we describe briefly as follows: start with an initial guess θ_0, and write the Taylor approximation (ignoring small order terms) as

$$\eta(\theta) \approx \eta(\theta_0) + J_0(\theta - \theta_0)$$

Now note that this is equivalent to approximating the residuals $z(\theta) = Y - \eta(\theta)$ by

$$z(\theta) \approx (Y - \eta(\theta_0)) - J_0(\theta - \theta_0) \equiv z_0 - J_0\delta$$

Since θ_0 is our known starting value, estimating $\theta - \theta_0$ is equivalent to estimating θ. Thus we seek that $\delta_0 \equiv (\theta - \theta_0)$ which minimizes the residual:

$$\|z(\theta)\|_2 \equiv \|z_0 - J_0\delta_0\|_2 \tag{1}$$

and take as our new estimate $\theta_1 = \theta_0 + \hat{\delta}_0$. Proceeding inductively, the typical step of the algorithm is:

$$\begin{aligned} \theta_{i+1} &= \theta_i + (J_i^t J_i)^{-1} J_i^t \left[Y - \eta(\theta_i)\right] \\ &= \theta_i + \hat{\delta}_i \end{aligned}$$

We have reduced each iteration to the usual least squares problem, to which we could apply the `sweep()` routine from the problem above. However, the QR routine submitted for assignment three has a number of advantages; perhaps most importantly, it warns us when J_i is nearly rank deficient. Therefore, we use the QR routine to compute the successive $\theta_{i+1} = \theta_i + \hat{\delta}_i$.

2.2 A Measure of Uncertainty

It has been said that an estimate unaccompanied by a measure of uncertainty is almost worthless. Thus, the includes estimated standard errors for the $\hat{\theta}$. Since exact expressions are unavailable in the non-linear case, measures of uncertainty for the Gauss-Newton algorithm are based on an *asymptotic* covariance matrix, which is estimated by:

$$\widehat{Cov}(\hat{\theta}) = s^2(J^t J)^{-1}$$

where J is the Jacobian matrix from the discussion above, evaluated at the least squares estimates obtained upon the final iteration, and s^2 is the residual mean square given by

$$s^2 = \|Y - \eta(\hat{\theta})\|^2/(N - p)$$

In the example analyzed in section 3.2, v is the first parameter, so the approximate standard error of $\hat{v}$ is the square root of the (1,1) entry of $\widehat{Cov}(\hat{\theta})$. That is

$$s_v = \sqrt{s^2 e_1^t (J^t J)^{-1} e_1}$$

where $e_1 = (1, 0, \ldots, 0)^t$. These are output by the program `nlstest.c` in section 3.2.

3 Results

3.1 Sweep

In the following interaction with the `sweeptest.c`, which calls the `sweep()` routine, there are two features to notice: 1) the user can specify which, and how many of the columns to sweep, and 2) it doesn't matter what appears in the lower triangle of the matrix (we get the same answers using the full SSCP matrix, as we do when using SSCPU).

The matrix SSCP (called S below) was produced in MATLAB. It is the matrix $X^t X$, where X is a 100×6 matrix of normal (0,1) random numbers.

```
% sweeptest

Enter name of the file containing the spd matrix: SSCP

Enter the dimension: 6

Enter number of columns to be swept: 6

column for sweep 0: 0

column for sweep 1: 1

column for sweep 2: 2

column for sweep 3: 3

column for sweep 4: 4

column for sweep 5: 5

The matrix is:
74.90517        -2.05071        0.68651         7.51039         -5.73764        0.32840
-2.05071        89.96492        1.30913         3.03349         -12.17186       -1.65234
0.68651         1.30913         91.26154        -0.32621        -18.66620       19.21324
7.51039         3.03349         -0.32621        98.56231        2.74726         -10.59074
-5.73764        -12.17186       -18.66620       2.74726         94.53465        -13.85515
0.32840         -1.65234        19.21324        -10.59074       -13.85515       95.10084

The sweep produced:
-0.01354        -0.00047        -0.00008        0.00107         -0.00093        -0.00000
-2.05071        -0.01136        -0.00007        0.00039         -0.00159        -0.00041
0.68651         1.30913         -0.01176        0.00023         -0.00199        0.00188
7.51039         3.03349         -0.32621        -0.01036        0.00026         -0.00108
-5.73764        -12.17186       -18.66620       2.74726         -0.01155        -0.00165
0.32840         -1.65234        19.21324        -10.59074       -13.85515       -0.01052
```

Now we try it with only the upper triangle:

```
% sweeptest

Enter name of the file containing the spd matrix: SSCPU

Enter the dimension: 6

Enter number of columns to be swept: 6

column for sweep 0: 0

column for sweep 1: 1

column for sweep 2: 2

column for sweep 3: 3

column for sweep 4: 4

column for sweep 5: 5

The matrix is:
74.90517        -2.05071        0.68651         7.51039         -5.73764        0.32840
0.00000         89.96492        1.30913         3.03349         -12.17186       -1.65234
0.00000         0.00000         91.26154        -0.32621        -18.66620       19.21324
0.00000         0.00000         0.00000         98.56231        2.74726         -10.59074
0.00000         0.00000         0.00000         0.00000         94.53465        -13.85515
0.00000         0.00000         0.00000         0.00000         0.00000         95.10084

The sweep produced:
-0.01354        -0.00047        -0.00008        0.00107         -0.00093        -0.00000
0.00000         -0.01136        -0.00007        0.00039         -0.00159        -0.00041
0.00000         0.00000         -0.01176        0.00023         -0.00199        0.00188
0.00000         0.00000         0.00000         -0.01036        0.00026         -0.00108
0.00000         0.00000         0.00000         0.00000         -0.01155        -0.00165
0.00000         0.00000         0.00000         0.00000         0.00000         -0.01052
```

To verify that, after sweeping all 6 columns, we have the negative of the inverse, we take the inverse using MATLAB:

```
>> S

S =

   74.9052   -2.0507    0.6865    7.5104   -5.7376    0.3284
   -2.0507   89.9649    1.3091    3.0335  -12.1719   -1.6523
    0.6865    1.3091   91.2615   -0.3262  -18.6662   19.2132
    7.5104    3.0335   -0.3262   98.5623    2.7473  -10.5907
   -5.7376  -12.1719  -18.6662    2.7473   94.5347  -13.8551
```

```
    0.3284   -1.6523   19.2132  -10.5907  -13.8551   95.1008

>> format long

>> inv(S)

ans =

  Columns 1 through 4

   0.01354111561666   0.00046799568625   0.00008525124555  -0.00107611827999
   0.00046799568625   0.01135736256171   0.00007644175507  -0.00038944059053
   0.00008525124555   0.00007644175507   0.01181901254893  -0.00025423412626
  -0.00107611827999  -0.00038944059053  -0.00025423412626   0.01037194437930
   0.00092420563087   0.00157074269915   0.00204583692967  -0.00029704847656
  -0.00004104512204   0.00036574088772  -0.00211701973720   0.00116008917857

  Columns 5 through 6

   0.00092420563087  -0.00004104512204
   0.00157074269915   0.00036574088772
   0.00204583692967  -0.00211701973720
  -0.00029704847656   0.00116008917857
   0.01143124849618   0.00124310603658
   0.00124310603658   0.01125965024054
```

The results are nearly the same. We suspect the difference is due to round-off error, and not the instability of our method of taking inverses. The condition number of S is $\|S\|_2\|S^{-1}\|_2 \approx 1.98$,

```
>> cond(S)

ans =

   1.97781040735287
```

Let's try sweeping only the first 5 columns:

```
% sweeptest

Enter name of the file containing the spd matrix: SSCP

Enter the dimension: 6

Enter number of columns to be swept: 5

column for sweep 0: 0

column for sweep 1: 1

column for sweep 2: 2
column for sweep 3: 3

column for sweep 4: 4

The matrix is:
74.90517        -2.05071        0.68651         7.51039         -5.73764        0.32840
-2.05071        89.96492        1.30913         3.03349         -12.17186       -1.65234
0.68651         1.30913         91.26154        -0.32621        -18.66620       19.21324
7.51039         3.03349         -0.32621        98.56231        2.74726         -10.59074
-5.73764        -12.17186       -18.66620       2.74726         94.53465        -13.85515
0.32840         -1.65234        19.21324        -10.59074       -13.85515       95.10084

The sweep produced:
-0.01335        0.00006         -0.00002        -0.00033        0.00023         0.00438
-2.05071        -0.01112        -0.00000        -0.00003        0.00007         -0.01837
0.68651         1.30913         -0.01096        0.00000         0.00008         0.21053
7.51039         3.03349         -0.32621        -0.01015        0.00001         -0.10745
-5.73764        -12.17186       -18.66620       2.74726         -0.01058        -0.14656
0.32840         -1.65234        19.21324        -10.59074       -13.85515       95.10084
```

That is, our OLS estimate is $\hat{\beta} = (0.00438, -0.01837, 0.21053, -0.10745, -0.14656)$, as compared to that given by MATLAB:

```
>> Y = X(:,p)

>> X = X(:,1:5)


>> [beta,se]=lscov(X,Y,eye(100))

beta =

   0.00364532833282
  -0.03248243772267
   0.18801825029833
  -0.10303065848312
  -0.11040361023850


se =

   0.11251244721632
   0.10298814376501
   0.10333021250112
   0.09790140713222
   0.10275423659730
```

My estimates are slightly different and, at this point, I can't say why. Notice also that the se given by MATLAB is readily computable from our sweep output. MATLAB computes it using se $= \sqrt{\text{diag}(X^tX)^{-1}\text{mse}}$, and each component of this expression appears in the result of our sweep program.

3.2 Non-linear Regression

A description of the first problem on which we test the gn.c program was given as follows:[3]

An experiment was carried out wherein varying doses of Rose Bengal, an enzyme, was used as a catalyst for the production of an protein substrate. The concentration of the substrate was measured spectrophotometrically. The first column of the data represents the dose of the enzyme, and the second column the spectrophotometric response. The equation proposed for the data is

$$\text{response } = \frac{\theta_1 \cdot \text{dose}}{\theta_2 + \text{dose}} \tag{2}$$

where θ_1 and θ_2 are the parameters to estimated. (This equation is known as the Michaelis-Menten model for the velocity of an enzyme catalyzed reaction.)

Such non-linear functions are rather basic and easy to change, so we hard code it into the calling program `nlstest.c`. Now we need to decide on good initial values for $\theta = (\theta_1, \theta_2)$. According to the set up presented in section 2, the function given by (2) corresponds to the expectation surface, $\eta(\theta)$. That is,

$$E_\theta[Y] = \eta(\theta) = \frac{\theta_1 \cdot \text{dose}}{\theta_2 + \text{dose}}$$

We estimate $E_\theta[Y]$ from the data by $\bar{Y} = \sum y_i/N = 23.4$. Looking at a scatter plot of the data, we notice approximately, as dose $\to \infty$, response $\to 30$. Therefore, L'Hopital's rule would suggest an initial estimate of $\theta_1 = 30$. For a crude estimate of θ_2, we compute the sample mean $\sum \text{dose}/N \equiv \bar{d} = 0.228875$ and equate

$$\bar{Y} = \frac{\theta_1 \bar{d}}{\theta_2 + \bar{d}}$$

yielding the estimate $\theta_2 = 0.065$. Therefore, we take as our initial guess, $\theta_0 = (30, 0.065)$. Invoking the program `nlstest` with these initial values produces the following:

```
% nlstest

Enter file name containing the data: prob1.dat


gn() exited with return value 1, on iteration 7

MSE = 0.214164

COEFFICIENT       SE
33.12465          0.37729
0.04606           0.00208
```

[3]Source: BMDP manual, p. 395.

4 Program Listings

4.1 sweep program listings

```
/*********************************************************
 * sweeptest.c  main program for testing sweep subroutine   *
 *                                                          *
 * Created by William J. De Meo                             *
 * on 12/17/97                                              *
 *                                                          *
 *********************************************************/
#include <stdlib.h>
#include <stdio.h>
#include "prototypes.h"
#define MAX_NAME 100

void read_name(char *);

main()
{
     char *filename;
     double *x;
     long i, k, nrow, *index;

     filename = cmalloc(MAX_NAME);

     printf("\nEnter name of the file containing the symmetric matrix: ");
     read_name(filename);
     printf("\nEnter the dimension: ");
     scanf("%u",&nrow);

     printf("\nEnter number of columns to be swept: ");
     scanf("%u",&k);
     index = lmalloc(k);

     for(i=0;i<k;i++)
     {
          printf("\ncolumn for sweep %d: ",i);
          scanf("%u",index+i);
     }

     x = dmalloc(nrow*nrow);
     /* matread(x, nrow, ncol, filename);  */
     matlabread(x, nrow, nrow, filename);
     /*matrix is stored contiguously column-wise */
     printf("\nThe matrix is: \n");
     matprint(x,nrow,nrow);
     /* Test sweep: */
     sweep(nrow,x,k,index);

     printf("\nThe sweep produced: \n");
     matprint(x,nrow,nrow);
}

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

     while ((c = getchar()) != EOF && c != ' ' && c != '\n')
          name[i++] = c;
     name[i] = '\0';
}


/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  sweep.c

  Created on 12/16/97 by William J. De Meo
    Last modified 12/17/97

  Purpose: Perform sweep algorithm of an M dimensional symmetric matrix


  Further Details:  This implementation uses BLAS 1
                    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"

void daxpy_(long *N, double *alpha, double *x, long *INCX, double *Y, long *INCY);
/* Y <- alpha X + Y */

/* Subroutine sweep:  perform a sweep of p columns of A

   Arguments:
              M  (long) dimension of A

              A  (pointer to double)
                 on entry: the M by M symmetric matrix to be swept
                          (lower triangle is never accessed)
                 on exit: upper triangle = the swept matrix
                          (lower triangle is unchanged)
```

```
              p  (long) number of columns to be swept

              index  (pointer to long) a vector of length p containing the
                     indices specifying which columns are to be swept
                     Note: indices can take values in [0, M-1]
*/

void sweep(long M, double *A, long p, long *index)
{
     long i;
     long k;

     for(i=0;i<p;i++)
     {
          k = index[i];
          colsweep(M,A,k);

     }
}

/* Subroutine colsweep: sweep column k of A

   Arguments:
              M  (long) dimension of A

              A  (pointer to double)
                 on entry: the M by M symmetric matrix to be swept
                           (lower triangle is never accessed)
                 on exit: upper triangle = the swept matrix
                           (lower triangle is unchanged)

              k  (long) column to be swept
*/

void colsweep(long M, double *A, long k)
{

     long i,j,length,one = (long)1;
     double alpha;

     /* First convert non-k elements */

     /* rows 0 to k-1 */
     for(i=0;i<k;i++)
     {
          length = k-i;
          /* i < k, i <= j < k  (triangle) */
          /* for columns i:k of rows i:k, do the following daxpy:  */
          /* a(i,i:M) <- alpha*a(k,i:M) + a(i,i:M) */
          alpha = ((double)-1)*A[k*M+i]/A[k*M+k];
          daxpy_(&length,&alpha,A+(k*M+i),&one,A+(i*M+i),&M);
          /* notice: replacing row k col i here ^ with row i col k since sym */

          length = M-k-1;
          /* i < k, j > k  (rectangle) */
          daxpy_(&length,&alpha,A+((k+1)*M+k),&M,A+((k+1)*M+i),&M);

     }

     /* rows k+1 to M */
     for(i=k+1;i<M;i++)
     {
          length = M-i-1;

          /* k < i < j  (triangle) */
          /* for columns k:M of rows k:M, do the following daxpy:  */
          /* a(i,i:M) <- alpha*a(k,i:M) + a(i,i:M) */
          alpha = ((double)-1)*A[i*M+k]/A[k*M+k];
          daxpy_(&length,&alpha,A+(i*M+k),&M,A+(i*M+i),&M);
     }

     /* divide elements 0:k-1 of kth col of A by a_{kk} */
     for(i=0;i<k;i++)   A[k*M+i]/=A[k*M+k];
     /* divide elements k+1:M of the kth row of A by a_{kk} */
     for(j=k+1;j<M;j++)  A[j*M+k]/=A[k*M+k];
     /* Finally, invert diagonal */
     A[k*M+k] = ((double)-1)/A[k*M+k];
}
```

4.2 non-linear regression program listings

```
/*********************************************************************
  nlstest.c  main program for testing gn() subroutine

  Created by William J. De Meo on 12/20/97
       last modified on 12/21/97

  Purpose:  Perform non-linear least squares using the gn()
            implementation of the Gauss Newton algorithm found in
            /accounts/grad/chip/lib/gauss/gn.c
            The following desribes the problem used for the test:

            (Source: BMDP manual, p. 395)
            An experiment was carried out wherein varying doses of
            Rose Bengal, an enzyme, was used as a catalyst for the
            production of an protein substrate.  The concentration
            of the substrate was measured spectrophotometrically.
            The first column of the data represents the dose of the
            enzyme, and the second column the spectrophotometric
            response.  The equation proposed for the data is


                              v * dose
            response  =     ------------ ,
                             (k + dose)

            where v and k are the parameters to estimated.  (This equation
            is known as the Michaelis-Menten model for the velocity of an
            enzyme catalyzed reaction.)

            The data are:

            dose   response
            0.027  12.7
            0.044  16.0
            0.073  20.4
            0.102  22.3
            0.175  26.0
            0.257  28.8
            0.483  29.6
            0.670  31.4
*********************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "prototypes.h"
#define MAX_NAME 100

/* Data specific settings: */
/* number of observations */
#define M 8
/* number of covariates */
#define N 1
/* number of parameters */
#define P 2

#define MAXITER 20

void init(double *);
void eta(long, long, double *, long , double *, double *);
void jac(long, long, double *, long , double *, double *);
void (*f)(long, long, double *, long , double *, double *);
void (*g)(long, long, double *, long , double *, double *);

void gn(long , long, double *X, double *Y, long, double *theta,
        void   (*f)(long , long , double *X, long , double *theta, double *F),
        void   (*g)(long , long , double *X, long , double *theta, double *J),
        double *cov, double *sigma, long iter);

void read_name(char *name);

main()
{
     double *x, *y, *theta, *cov, *sigma, *se;
     long i, j;
     char *filename;

     filename = cmalloc(MAX_NAME);
     printf("\nEnter file name containing the data: ");
     read_name(filename);

     x = dmalloc(M*(N+1));
     y = x+M; /* y is assigned the address of second col of x */
     theta = dmalloc(P);
     se = dmalloc(P);
     cov = dmalloc(P*P);
     sigma = dmalloc((long)1);

     matread(x, M, N+1, filename);

     init(theta);

     f = eta;
     g = jac;

     gn(M,N,x,y,P,theta,f,g,cov,sigma,MAXITER);

     /* compute se's:  */
     for(j=0;j<P;j++)
          se[j] = sqrt((*sigma)*cov[P*j+j]);
     printf("\n\nMSE = %lf\n",*sigma);
     printf("\nCOEFFICIENT \t SE \n");
     for (i = 0; i < P; i++)
          printf("%4.5lf \t %4.5lf\n", theta[i],se[i]);
}

void init(double *theta)
{
     theta[0]=(double)30;
     theta[1]=(double)0.065;
}

void eta(long m, long n, double *x, long p, double *th, double *F)
{
     long i;
     for(i=0;i<m;i++)
          F[i] = (th[0] * x[i])/(th[1] + x[i]);
}

void jac(long m, long n, double *x, long p, double *th, double *J)
{
     long i;

     for(i=0;i<m;i++)
     {
          J[i] = x[i]/(th[1] + x[i]);
          J[M+i] = -th[0]*x[i]/pow((th[1] + x[i]),2);
     }
}

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

     while ((c = getchar()) != EOF && c != ' ' && c != '\n')
          name[i++] = c;
     name[i] = '\0';
}


/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  gn.c

  Created on 12/18/97 by William J. De Meo
    Last modified 12/21/97

  Purpose: Perform one iteration of Gauss-Newton with step halving

  Further Details:  This implementation uses BLAS 2
                    (matrix-vector mult. and rank 1 updates)

  Dependencies:
  subroutines found in:
              /accounts/grad/chip/lib/decomp/QR/House.c
              sunperf, and blas
              the later two are linked with the options:
```

```
              -lsunperf -dalign -lblas
              compilation must be done with the -dalign option
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

#include <math.h>

/* ~~~Prototypes ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

For subroutines in /accounts/grad/chip/lib/decomp/QR/House.c:            */
#include "prototypes.h"

/*
For BLAS subroutines:            */

double dnrm2_(long *N, double *x, long *INC);/* L2 norm of x*/
void dcopy_(long *N, double *X, long *INCX, double *Y, long *INCY); /* y <- x */

/* y <- (alpha)Ax + (beta)y   (or A^t if TRANSA='T') */
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)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);

/* 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 *);

void Exit(int i,long k);
void Free(double*,double*,double*,double*,double*,double*,double*,double*);

/* Tolerance for smallest singular value of J */
#define SINGTOL 1.0e-7
#define DELTOL 1.0e-7
#define STEPTOL 1.0e-4

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  Subroutine gn:

   Arguments:
              M  (long) number of rows of X
```

```
              N  (long) number of columns of X

              X  (pointer to double) the M by N matrix of covariates

              Y  (pointer to double) M by 1 observable vector

              P  (long) number of parameters

              theta (pointer to double) P-vector of parameters
                    on entry: initial guess
                    on exit: improved solution after one iteration

              (*f)(M,N,X,P,theta,F) points to a function which evaluates to the expectation
                                 surface for given covariates X and parameter values theta

              (*g)(M,N,X,P,theta,J) points to a function which evaluates to the analytic
                                 Jacobian of f

              cov   on entry: an arbitrary PxP matrix
                    on exit: the estimated asymptotic covariance matrix

              sigma   on exit: the mse =  |y - f(theta)|^2 / (M-P)

              iter (long) max number of iterations

*/
void gn(long M, long N, double *X, double *Y, long P, double *theta,
        void   (*f)(long M, long N, double *X, long P, double *theta, double *F),
        void   (*g)(long M, long N, double *X, long P, double *theta, double *J),
        double *cov, double *sigma, long iter)
{
     long i,j,k, one=(long)1;
     double *F, *J, *leadu, *E, *z, *delta, newS, oldS, half;
     double *pwork,*ework,test;


     F = dmalloc(M);
     J = dmalloc(M*P);
     z = dmalloc(M);
     E = dmalloc(P*P);
     leadu = dmalloc(P);
     delta = dmalloc(P);

     pwork = dmalloc(P); /* parameter workspace */
     ework = dmalloc(M); /* error workspace */

     for(k=0;k<iter;k++)
```

```
{
     (*f)(M,N,X,P,theta,F);
     /* z <- y - f(theta) */
     for(i=0;i<M;i++) z[i] = Y[i] - F[i];

     oldS = dnrm2_(&M,z,&one); /* L2 norm of error */

     /* Form Jacobian and QR decomposition */
     (*g)(M,N,X,P,theta,J);

     qrpivot(M,P,J,E,leadu);
     /*printf("\nJ[%d] = \n",k+1);
       matprint(J,M,P);*/

     if(fabs(J[M*(P-1)+(P-1)]) < SINGTOL)
     {
          Exit(0,k); /* 0 <= J rank defficient (bad exit) */
          Free(F,J,z,E,leadu,delta,pwork,ework);
          return;
     }

     /* reg for fitting z ~ J.delta */
     reg(M,P,J,leadu,E,z,delta,cov,pwork,ework,sigma);

     /* Don't need errors=pwork or se=ework computed by reg() */
     /* So P vector pwork and M vector ework are used
        for a different purpose below */

     /* cov = inv(J^tJ) for J corresponding to current theta
        so if we exit with codes 1 or 2 below, i.e. before computing a new theta,
        we return the correct covariance matrix */

     test=(double)0;
     for(j=0;j<P;j++)
     {
          if(fabs(delta[j])>test) test = delta[j];
     }
     if(test < DELTOL)
     {
          *sigma = oldS/(M-P);      /* get new sigma before exit */
          Exit(1,k);                  /* 1 <= no change in theta_i */
          Free(F,J,z,E,leadu,delta,pwork,ework);
          return;
     }

     j=0;
     do
     {
```

```
               if((half = pow(.5,j)) < STEPTOL)
               {
                    *sigma = oldS/(M-P);      /* get new sigma before exit */
                    Exit(2,k);  /* 2 <= No improvements from steps */
                    Free(F,J,z,E,leadu,delta,pwork,ework);
                    return;
               }
               j++;
               /* compute new candidate coefficients */
               for(i=0;i<P;i++) pwork[i] = theta[i] + half*delta[i];
               (*f)(M,N,X,P,pwork,F);
               for(i=0;i<M;i++) ework[i] = Y[i] - F[i];        /* compute new error */
               newS = dnrm2_(&M, ework, &one);  /* L2 norm of new error*/
          }while(newS > oldS);

          dcopy_(&P,pwork, &one, theta, &one); /* theta <- pwork */
     }

}/*~~~ end gn() ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  Subroutine reg()
   Arguments:

        M number of rows of X

        N number of columns of X (expect N < M)

        QR the result of applying qrpivot() to X
           (i.e. upper triangle is R, lower trapezoid are u's)

        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^te / (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;
     double a, *Qy, *invR, *EiR, *coef;

     /* BLAS arguments */
     long INC=(long)1;
     double one = (double)1, zero = (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^te */
          for(i=j+1;i<M;i++)
               e[i] +=  a* QR[M*j+i]; /* e <- e + (-2)uu^t e */
     }
/*~~~~~~~~~~~~~~~~~~~~~
  COMPUTE COEFFICIENTS

  /* 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 <- one*inv(R)*invR = one*inv(R)*eye  */
     dtrsm_(&SIDE, &UPLO, &NOTRANS, &DIAG, &N, &N, &one, QR, &M,invR,&N);

     /* compute the E*inv(R) matrix */
```

```
     EiR = dmalloc(N*N);
     /*  EiR <- (one)E*invR + (zero)EiR  */
     dgemm_(&NOTRANS, &NOTRANS, &N, &N, &N, &one, E, &N,
            invR, &N, &zero, EiR, &N);
     free(invR);

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

/********* residuals and mse for the OLS part not necessary *********/
     if(0)
     {

          /*  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  (wrong MSE in nonlinear case. currently embraced by if(0))
           */
          *sigma = ddot_(&M, y, &INC, e, &INC);
          *sigma /= (M - N);

     }
/*** end if(0) ***/

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  COMPUTE COVARIANCE MATRIX and SE's

  cov <- (one)EiR*(EiR)' + (zero)cov  */
     dgemm_(&NOTRANS, &TRANS, &N, &N, &N, &one, EiR, &N,
            EiR, &N, &zero, cov, &N);

/* wrong se's:
   for(j=0;j<N;j++)
   se[j] = sqrt((*sigma)*cov[N*j+j]);
```

```
   */

}/*~~~ end reg() ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/


void Exit(int i, long k)
{
     printf("\n\ngn() exited with return value %d, on iteration %d",i,k+1);
}

void Free(double *a, double *b, double *c, double *d, double *e,
          double *f,double *g, double *h)
{
     free(a);free(b);free(c);free(d);free(e);free(f);free(g);free(h);
}
```

References

[1] D. Bates and D. Watts, *Nonlinear Regression Analysis and its Applications*. John Wiley, New York, 1988.

[2] R. Thisted, *Elements of Statistical Computing*. Chapman and Hall, New York, 1988.