Writedata_ols.c

```
/* writedata_ols.c -- Writes out a file for ols_read.c.  The X matrix
 * is random data, the BETA vector is pre-set, and Y vector ix X*BETA.
 * The first column is Y, the second is a column of "1"s for the
 * intercept, and the remaing columns are drawn from the rand()
 * function.  User sets number of columns and number of rows below. */
#include <stdlib.h>
#include <stdio.h>
/* Declare pointer to the output file */
FILE *kp;

int main(void){

        int nrow=1000, ncol=25;
        double *X, *Y, *BETA;
        double sum;
        int i = 0;
        int j = 0;

        X = (double *) malloc (nrow*ncol*sizeof(double));
        Y = (double *) malloc (nrow*sizeof(double));
        BETA = (double *) malloc (nrow*sizeof(double));

/* Open the output file */
        kp = fopen("data_ols.txt","w");

        printf("\nnumber of rows = %d  number of columns = %d\n\n",nrow,ncol);

/* Initialize BETA vector -- Note that the Constant term will be
 * affected by the means of the variables when OLS.c is run */
        for(j=0;j<ncol;j++)
        {
                BETA[j] = 1;
        }

        srand(14);
/* Fill X matrix with random numbers*/
        for(i=0;i<nrow*ncol;i++){
                X[i] = ( (double)rand() / ((double)(RAND_MAX)+(double)(1)));
        }
/* Put "1"s in first Column of X -- Columns are stacked in vector X */
        for(i=0;i<nrow;i++)
        {
                X[i] = 1;
        }
/* Calculate Y */
        for(i=0;i<nrow;i++)
        {
                sum=0.0;
                for(j=0;j<ncol;j++)
                {
                        sum=sum+BETA[j]*X[i+j*nrow];
                }
                Y[i]=sum;
```

```
        }

/* For loops for writing out Y and X */
        for(i=0;i<nrow;i++)
        {
                fprintf(kp, "%7.3f",Y[i]);
                for(j=0;j<ncol;j++)
                {
                        fprintf(kp, "%7.3f",X[i+j*nrow]);
                }
/* This is the line feed -- newline at the end of the the jth row */
                fprintf(kp,"\n");
        }
        fclose(kp);
        free(X);
        free(Y);
        free(BETA);
        return(0);
}
```

ols_read.c

```
/*
c:/mingw/bin/gcc -I"c:/program files/R/R-2.9.0/include" -L"C:/Program
Files/R/R-2.9.0/bin" -Wall %1.c -o %1.exe -lRlapack -lRblas

General OLS program.  Reads matrix created by writedata_ols.c

The only parameters the user has to set are the number of rows and
columns -- nrow and ncol below.  The number of columns counts a column
of "1"s used for the intercept term.  All memory is then dynamically
allocated using nrow and ncol.
*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <R_ext/Lapack.h>
#include <R_ext/BLAS.h>
FILE *fp;

int main(){

        int i, j, info, errno;
        char trans = 't', notrans ='n';
        double alpha = 1.0, beta=0.0;
        int nrow=1000, ncol=25;
        int one=1;
        double *X, *Y, *XprimeX, *XXinv, *XXinvX, *coef;
        int *ipiv;

        X = (double *) malloc (nrow*ncol*sizeof(double));
        Y = (double *) malloc (nrow*sizeof(double));
        XprimeX = (double *) malloc (ncol*ncol*sizeof(double));
        XXinv = (double *) malloc (ncol*ncol*sizeof(double));
        XXinvX = (double *) malloc (nrow*ncol*sizeof(double));
        coef = (double *) malloc (ncol*sizeof(double));
        ipiv = (int *) malloc (ncol*sizeof(int));

        printf("\nnumber of rows = %d  number of columns = %d\n\n",nrow,ncol);

        if((fp = fopen("c:/docs_c_summer_course/data_ols.txt","r"))==NULL)
        {
                printf("\nUnable to open file OLS_DATA.TXT: %s\n",
strerror(errno));
                exit(EXIT_FAILURE);
        }
        else {

           printf(" Y and X = \n");
           for(i=0;i<nrow;i++)
           {
                  fscanf(fp,"%lf",&Y[i]);
                  for(j=0;j<ncol;j++)
                  {
                         fscanf(fp,"%lf",&X[i+j*nrow]);
```

```
                }
                printf("%10d %12.6f", i,Y[i]);
                for(j=0;j<ncol;j++)
                {
                        printf("%12.6f",X[i+j*nrow]);
                }
                printf("\n");
           }
        }

        dgemm_(&trans,&notrans,&ncol,&ncol,&nrow,&alpha,X,&nrow,X,&nrow,&beta,
XprimeX,&ncol);
        printf("\n\nX'X = \n");
        for(i=0;i<ncol;i++)
        {
                for(j=0;j<ncol;j++)
                {
                        printf("%12.6f",XprimeX[i+j*ncol]);
                }
                printf("\n");
        }
/* Initialize the Identity matrix */
        for(i=0;i<ncol;i++)
        {
                for(j=0;j<ncol;j++)
                {
                        XXinv[i+j*ncol]=0.0;
                        if(i == j)XXinv[i+j*ncol]=1.0;
                }
        }
        dgesv_(&ncol,&ncol,XprimeX,&ncol,ipiv,XXinv,&ncol,&info);
        printf("\n\n(X'X)-1 = \n");
        for(i=0;i<ncol;i++)
        {
                for(j=0;j<ncol;j++)
                {
                        printf("%12.6f",XXinv[i+j*ncol]);
                }
                printf("\n");
        }

//XXinv is 2x2
//X' is 2x5
//X is 5x2

        dgemm_(&notrans,&trans,&ncol,&nrow,&ncol,&alpha,XXinv,&ncol,X,&nrow,&b
eta,XXinvX,&ncol);

//XXinvX is 2x5
//Y is 5x1

        dgemm_(&notrans,&notrans,&ncol,&one,&nrow,&alpha,XXinvX,&ncol,Y,&nrow,
&beta,coef,&ncol);
        printf("\n\nCoefficient Vector = ");
        for(i=0;i<ncol;i++)
```

```
        {
             printf("\n%d %12.6f", i, coef[i]);
        }
        printf("\n\n");
        free(X);
        free(Y);
        free(XprimeX);
        free(XXinvX);
        free(coef);
        free(ipiv);

        fclose(fp);
        return(0);

}
```

ols_read_svd.c

```
/*
c:/mingw/bin/gcc -I"c:/program files/R/R-2.9.0/include" -L"C:/Program
Files/R/R-2.9.0/bin" -Wall %1.c -o %1.exe -lRlapack -lRblas

General OLS program.  Reads matrix created by writedata_ols.c

The only parameters the user has to set are the number of rows and
columns -- nrow and ncol below.  The number of columns counts a column
of "1"s used for the intercept term.  All memory is then dynamically
allocated using nrow and ncol.

This version does a Singular Value Decomposition on the input matrix
*/
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <R_ext/Lapack.h>
#include <R_ext/BLAS.h>

void xsvd(int kpnp, int kpnq, double *, double *, double *, double *);

FILE *fp;
FILE *jp;

int main(){

        int i, j, info, errno;
        char trans = 't', notrans ='n';
        double alpha = 1.0, beta=0.0;
        int nrow=1000, ncol=25;
        int one=1;
        double *X, *Y, *XprimeX, *XXinv, *XXinvX, *coef;
        double *u, *lambda, *vt;
        int *ipiv;
        double time1, time2, timedif;

        X = (double *) malloc (nrow*ncol*sizeof(double));
        Y = (double *) malloc (nrow*sizeof(double));
        XprimeX = (double *) malloc (ncol*ncol*sizeof(double));
        XXinv = (double *) malloc (ncol*ncol*sizeof(double));
        XXinvX = (double *) malloc (nrow*ncol*sizeof(double));
        coef = (double *) malloc (ncol*sizeof(double));
        ipiv = (int *) malloc (ncol*sizeof(int));
        u      = (double *) malloc (nrow*nrow*sizeof(double));
        lambda = (double *) malloc (ncol*ncol*sizeof(double));
        vt     = (double *) malloc (ncol*ncol*sizeof(double));
/* clock() is part of time.h -- returns the implementation's
 * best approximationto the processor time elapsed since the
 * program was invoke, divide by CLOCKS_PER_SEC to get the time
 * in seconds           */
        time1 = (double) clock();            /* get initial time */
```

```
        time1 = time1 / CLOCKS_PER_SEC;      /*    in seconds    */

        printf("\nnumber of rows = %d  number of columns = %d\n\n",nrow,ncol);

        jp = fopen("c:/docs_c_summer_course/data_ols_svd.txt","w");

        if((fp = fopen("c:/docs_c_summer_course/data_ols.txt","r"))==NULL)
        {
                printf("\nUnable to open file OLS_DATA.TXT: %s\n",
strerror(errno));
                exit(EXIT_FAILURE);
        }
        else {

           fprintf(jp," Y and X = \n");
           for(i=0;i<nrow;i++)
           {
                   fscanf(fp,"%lf",&Y[i]);
                   for(j=0;j<ncol;j++)
                   {
                           fscanf(fp,"%lf",&X[i+j*nrow]);
                   }
                   fprintf(jp,"%10d %12.6f", i,Y[i]);
                   for(j=0;j<ncol;j++)
                   {
                           fprintf(jp,"%12.6f",X[i+j*nrow]);
                   }
                   fprintf(jp,"\n");
           }
        }
/* Call Singular Value Decomposition Routine to look at the colinearity
 * in X */
/*  Clock the SVD Routine*/

        time2 = (double) clock();            /* get initial time */
        time2 = time2 / CLOCKS_PER_SEC;      /*    in seconds    */
        xsvd(nrow,ncol,X,u,lambda,vt);
        timedif = ( ((double) clock()) / CLOCKS_PER_SEC) - time2;
        printf("SVD took %12.8f seconds\n", timedif);
        fprintf(jp,"SVD took %12.8f seconds\n", timedif);

        dgemm_(&trans,&notrans,&ncol,&ncol,&nrow,&alpha,X,&nrow,X,&nrow,&beta,
XprimeX,&ncol);
        fprintf(jp,"\n\nX'X = \n");
        for(i=0;i<ncol;i++)
        {
                for(j=0;j<ncol;j++)
                {
                        fprintf(jp,"%12.6f",XprimeX[i+j*ncol]);
                }
                fprintf(jp,"\n");
        }
/* Initialize the Identity matrix */
        for(i=0;i<ncol;i++)
        {
```

```
                for(j=0;j<ncol;j++)
                {
                        XXinv[i+j*ncol]=0.0;
                        if(i == j)XXinv[i+j*ncol]=1.0;
                }
        }
        dgesv_(&ncol,&ncol,XprimeX,&ncol,ipiv,XXinv,&ncol,&info);
        fprintf(jp,"\n\n(X'X)-1 = \n");
        for(i=0;i<ncol;i++)
        {
                for(j=0;j<ncol;j++)
                {
                        fprintf(jp,"%12.6f",XXinv[i+j*ncol]);
                }
                fprintf(jp,"\n");
        }

//XXinv is 2x2
//X' is 2x5
//X is 5x2

        dgemm_(&notrans,&trans,&ncol,&nrow,&ncol,&alpha,XXinv,&ncol,X,&nrow,&b
eta,XXinvX,&ncol);

//XXinvX is 2x5
//Y is 5x1

        dgemm_(&notrans,&notrans,&ncol,&one,&nrow,&alpha,XXinvX,&ncol,Y,&nrow,
&beta,coef,&ncol);
        fprintf(jp,"\n\nCoefficient Vector = ");
        printf("\n\nCoefficient Vector = ");
        for(i=0;i<ncol;i++)
        {
                printf("\n%d %12.6f", i, coef[i]);
                fprintf(jp,"\n%d %12.6f", i, coef[i]);
        }
        printf("\n\n");
        free(X);
        free(Y);
        free(XprimeX);
        free(XXinvX);
        free(coef);
        free(ipiv);
        free(u);
        free(lambda);
        free(vt);
        timedif = ( ((double) clock()) / CLOCKS_PER_SEC) - time1;
        printf("The total elapsed time of the program is %12.8f seconds\n",
timedif);
        fprintf(jp,"\nThe total elapsed time of the program is %12.8f
seconds\n", timedif);

        fclose(fp);
        fclose(jp);
        return(0);
```

```
}
/*

Singular Value Decomposition Subroutine

*/
void xsvd(int kpnp, int kpnq, double *y, double *u, double *lambda, double
*vt) {
/*
*/

        double *a, *work;
        double sumulv, svd_error_sum, svd_error_sum_2;
        int i, j, jj;
        int  info = 12;
        int  lwork= kpnp*kpnp+kpnq*kpnq;
        int  lda,ldu,ldvt;

        a     = calloc( kpnp*kpnq, sizeof(double));
        work  = calloc( lwork, sizeof(double));

        fprintf(jp,"entering svd...\n");

        lda  = kpnp;
        ldu  = kpnp;
        ldvt = kpnq;
        for (i=0;i<lwork;i++){
                work[i] = 0;
        }

        fprintf(jp,"lwork=%i\n",lwork);

        for (j=0;j<kpnq;j++) {
                for (i=0;i<kpnp;i++) {
                        a[(j*kpnp)+i] = y[(j*kpnp)+i];
                }
        }

        dgesvd_("A","A", &kpnp, &kpnq, a, &lda, lambda,
                  u, &ldu, vt, &ldvt, work, &lwork, &info);

        fprintf(jp,"Info = %i\n",info);
        printf("Info = %i\n",info);
        fprintf(jp,"Singular Values\n");
        printf("Singular Values\n");
        for(jj=0;jj<kpnq;jj++)
        {
                fprintf(jp,"%d %f\n",jj,lambda[jj]);
                printf("%d %f\n",jj,lambda[jj]);
        }
/*
  Do simple check of SVD

*/
```

```
        svd_error_sum=0.0;
        svd_error_sum_2=0.0;
        for (i=0;i<kpnp;i++)
        {
                for (jj=0;jj<kpnq;jj++)
                {
                        sumulv=0.0;
                        for (j=0;j<kpnq;j++)
                        {

        sumulv+=u[(j*kpnp)+i]*lambda[j]*vt[j+(jj*kpnq)];
                        }
                        svd_error_sum+=(y[i+(jj*kpnp)]-sumulv)*(y[i+(jj*kpnp)]-
sumulv);
                        svd_error_sum_2+=fabs(y[i+(jj*kpnp)]-sumulv);
                }
        }
        fprintf(jp,"SVD Error Check = %12.7g
%12.7g\n",svd_error_sum,svd_error_sum_2);
        printf("SVD Error Check = %12.7g
%12.7g\n",svd_error_sum,svd_error_sum_2);
        fprintf(jp,"Leaving svd...\n");
        free(work);
        free(a);
}
```

Subroutine DGESVD from LAPACK Library

```
      SUBROUTINE DGESVD( JOBU, JOBVT, M, N, A, LDA, S, U, LDU, VT, LDVT,
     $                   WORK, LWORK, INFO )
*
*  -- LAPACK driver routine (version 3.2) --
*  -- LAPACK is a software package provided by Univ. of Tennessee,    --
*  -- Univ. of California Berkeley, Univ. of Colorado Denver and NAG Ltd..--
*     November 2006
*
*     .. Scalar Arguments ..
      CHARACTER          JOBU, JOBVT
      INTEGER            INFO, LDA, LDU, LDVT, LWORK, M, N
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   A( LDA, * ), S( * ), U( LDU, * ),
     $                   VT( LDVT, * ), WORK( * )
*     ..
*
*  Purpose
*  =======
*
*  DGESVD computes the singular value decomposition (SVD) of a real
*  M-by-N matrix A, optionally computing the left and/or right singular
*  vectors. The SVD is written
*
*       A = U * SIGMA * transpose(V)
*
*  where SIGMA is an M-by-N matrix which is zero except for its
*  min(m,n) diagonal elements, U is an M-by-M orthogonal matrix, and
*  V is an N-by-N orthogonal matrix.  The diagonal elements of SIGMA
*  are the singular values of A; they are real and non-negative, and
*  are returned in descending order.  The first min(m,n) columns of
*  U and V are the left and right singular vectors of A.
*
*  Note that the routine returns V**T, not V.
*
*  Arguments
*  =========
*
*  JOBU    (input) CHARACTER*1
*          Specifies options for computing all or part of the matrix U:
*          = 'A':  all M columns of U are returned in array U:
*          = 'S':  the first min(m,n) columns of U (the left singular
*                  vectors) are returned in the array U;
*          = 'O':  the first min(m,n) columns of U (the left singular
*                  vectors) are overwritten on the array A;
*          = 'N':  no columns of U (no left singular vectors) are
*                  computed.
*
*  JOBVT   (input) CHARACTER*1
*          Specifies options for computing all or part of the matrix
*          V**T:
*          = 'A':  all N rows of V**T are returned in the array VT;
*          = 'S':  the first min(m,n) rows of V**T (the right singular
```

```
*                 vectors) are returned in the array VT;
*          = 'O':  the first min(m,n) rows of V**T (the right singular
*                  vectors) are overwritten on the array A;
*          = 'N':  no rows of V**T (no right singular vectors) are
*                  computed.
*
*          JOBVT and JOBU cannot both be 'O'.
*
*  M       (input) INTEGER
*          The number of rows of the input matrix A.  M >= 0.
*
*  N       (input) INTEGER
*          The number of columns of the input matrix A.  N >= 0.
*
*  A       (input/output) DOUBLE PRECISION array, dimension (LDA,N)
*          On entry, the M-by-N matrix A.
*          On exit,
*          if JOBU = 'O',  A is overwritten with the first min(m,n)
*                          columns of U (the left singular vectors,
*                          stored columnwise);
*          if JOBVT = 'O', A is overwritten with the first min(m,n)
*                          rows of V**T (the right singular vectors,
*                          stored rowwise);
*          if JOBU .ne. 'O' and JOBVT .ne. 'O', the contents of A
*                          are destroyed.
*
*  LDA     (input) INTEGER
*          The leading dimension of the array A.  LDA >= max(1,M).
*
*  S       (output) DOUBLE PRECISION array, dimension (min(M,N))
*          The singular values of A, sorted so that S(i) >= S(i+1).
*
*  U       (output) DOUBLE PRECISION array, dimension (LDU,UCOL)
*          (LDU,M) if JOBU = 'A' or (LDU,min(M,N)) if JOBU = 'S'.
*          If JOBU = 'A', U contains the M-by-M orthogonal matrix U;
*          if JOBU = 'S', U contains the first min(m,n) columns of U
*          (the left singular vectors, stored columnwise);
*          if JOBU = 'N' or 'O', U is not referenced.
*
*  LDU     (input) INTEGER
*          The leading dimension of the array U.  LDU >= 1; if
*          JOBU = 'S' or 'A', LDU >= M.
*
*  VT      (output) DOUBLE PRECISION array, dimension (LDVT,N)
*          If JOBVT = 'A', VT contains the N-by-N orthogonal matrix
*          V**T;
*          if JOBVT = 'S', VT contains the first min(m,n) rows of
*          V**T (the right singular vectors, stored rowwise);
*          if JOBVT = 'N' or 'O', VT is not referenced.
*
*  LDVT    (input) INTEGER
*          The leading dimension of the array VT.  LDVT >= 1; if
*          JOBVT = 'A', LDVT >= N; if JOBVT = 'S', LDVT >= min(M,N).
*
```

```
*  WORK    (workspace/output) DOUBLE PRECISION array, dimension
(MAX(1,LWORK))
*          On exit, if INFO = 0, WORK(1) returns the optimal LWORK;
*          if INFO > 0, WORK(2:MIN(M,N)) contains the unconverged
*          superdiagonal elements of an upper bidiagonal matrix B
*          whose diagonal is in S (not necessarily sorted). B
*          satisfies A = U * B * VT, so it has the same singular values
*          as A, and singular vectors related by U and VT.
*
*  LWORK   (input) INTEGER
*          The dimension of the array WORK.
*          LWORK >= MAX(1,3*MIN(M,N)+MAX(M,N),5*MIN(M,N)).
*          For good performance, LWORK should generally be larger.
*
*          If LWORK = -1, then a workspace query is assumed; the routine
*          only calculates the optimal size of the WORK array, returns
*          this value as the first entry of the WORK array, and no error
*          message related to LWORK is issued by XERBLA.
*
*  INFO    (output) INTEGER
*          = 0:  successful exit.
*          < 0:  if INFO = -i, the i-th argument had an illegal value.
*          > 0:  if DBDSQR did not converge, INFO specifies how many
*                superdiagonals of an intermediate bidiagonal form B
*                did not converge to zero. See the description of WORK
*                above for details.
*
*  =====================================================================
*
```