From Wikipedia, the free encyclopedia

Box–Muller transform

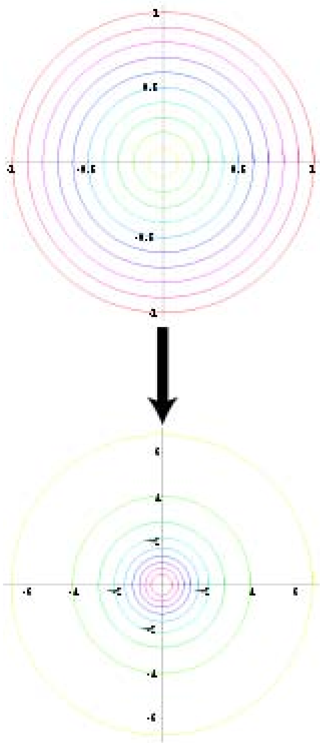


Diagram of the Box Muller transform. The initial circles, uniformly spaced about the origin, are mapped to another set of circles about the origin that are closely spaced near the origin but quickly spread out. The largest circles in the domain map to the smallest circles in the range and vice versa.

A **Box–Muller transform** (by George Edward Pelham Box and Mervin Edgar Muller 1958)[1] is a method of generating pairs of independent standard normally distributed (zero expectation, unit variance) random numbers, given a source of uniformly distributed random numbers.

It is commonly expressed in two forms. The basic form as given by Box and Muller takes two samples from the uniform distribution on the interval (0, 1] and maps them to two normally distributed samples. The polar form takes two samples from a different interval, [−1, +1], and maps them to two normally distributed samples without the use of sine or cosine functions.

One could use the inverse transform sampling method to generate normally-distributed random numbers instead; the Box–Muller transform was developed to be more computationally efficient.[2] The more efficient Ziggurat algorithm can also be used.

Basic form

Suppose U_1 and U_2 are independent random variables that are uniformly distributed in the interval (0, 1]. Let

$$Z_0 = R\cos(\Theta) = \sqrt{-2\ln U_1}\cos(2\pi U_2)$$

and

$$Z_1 = R\sin(\Theta) = \sqrt{-2\ln U_1}\sin(2\pi U_2).$$

Then Z_0 and Z_1 are independent random variables with a normal distribution of standard deviation 1.

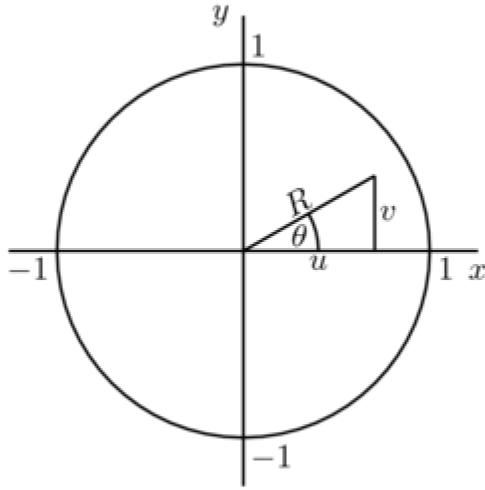
The derivation[3] is based on the fact that, in a two-dimensional Cartesian system where X and Y coordinates are described by two independent and normally distributed random variables, the random variables for $\mathbf{R}^2$ and Θ (shown above) in the corresponding polar coordinates are also independent and can be expressed as

$$R^2 = -2 \cdot \ln U_1$$

and

$$\Theta = 2\pi U_2.$$

Polar form



$$R^2 = u^2 + v^2$$

$$\cos\theta = \frac{u}{R}$$

$$\sin\theta = \frac{v}{R}$$

Two uniformly distributed values, *u* and *v* are used to produce the value $s = R^2$, which is likewise uniformly distributed. The definitions of the sine and cosine are then applied to the basic form of the Box–Muller transform in order to avoid using trigonometric functions.

The polar form is attributed by Devroye[4] to Marsaglia. It is also mentioned without attribution in Carter.[5]

Given *u* and *v*, independent and uniformly distributed in the closed interval [−1, +1], set $s = R = u^2 + v^2$. (Clearly $R=\sqrt{s}$.) If $s = 0$ or $s > 1$, throw *u* and *v* away and try another pair (u, v). Continue until a pair with *s* in the open interval (0, 1) is found. Because *u* and *v* are uniformly distributed and because only points within the unit circle have been admitted, the values of *s* will be uniformly distributed in the open interval (0, 1), too. The latter can be seen by calculating the cumulative distribution function for *s* in the interval (0, 1). This is the area of a circle with radius $\sqrt{s}$ divided by π. From this we find the probability density function to have the constant value 1 on the interval (0, 1). Equally so, the angle θ divided by 2π is uniformly distributed in the open interval (0, 1) and independent of *s*.

We now identify the value of *s* with that of U_1 and $\theta/(2\pi)$ with that of U_2 in the basic form. As shown in the figure, the values of $\cos\theta=\cos 2\pi U_2$ and $\sin\theta=\sin 2\pi U_2$ in the basic form can be replaced with the ratios $\cos\theta=u/R=u/\sqrt{s}$ and $\sin\theta=v/R=v/\sqrt{s}$, respectively. The advantage is that calculating the trigonometric functions directly can be avoided. This is helpful when they are comparatively more expensive than the single division that replaces each one.

Just as the basic form produces two standard normal deviates, so does this alternate calculation.

$$z_0 = \sqrt{-2\ln U_1}\cos(2\pi U_2) = \sqrt{-2\ln s}\left(\frac{u}{\sqrt{s}}\right) = u\cdot\sqrt{\frac{-2\ln s}{s}}$$

and

$$z_1 = \sqrt{-2\ln U_1}\sin(2\pi U_2) = \sqrt{-2\ln s}\left(\frac{v}{\sqrt{s}}\right) = v\cdot\sqrt{\frac{-2\ln s}{s}}.$$

Contrasting the two forms

The polar method differs from the basic method in that it is a type of rejection sampling. It throws away some generated random numbers, but it is typically faster than the basic method because it is simpler to compute (provided that the random number generator is relatively fast) and is more numerically robust.[5] It avoids the use of trigonometric functions, which are comparatively expensive in many computing environments. It throws away $1 - \pi/4 \approx 21.46\%$ of the total input uniformly distributed random number pairs generated, i.e. throws away $4/\pi - 1 \approx 27.32\%$ uniformly distributed random number pairs per Gaussian random number pair generated, requiring $4/\pi \approx 1.2732$ input random numbers per output random number.

The basic form requires three multiplications, one logarithm, one square root, and one trigonometric function for each normal variate.[6]

The polar form requires two multiplications, one logarithm, one square root, and one division for each normal variate. The effect is to replace one multiplication and one trigonometric function with a single division.

From *Numerical Recipes in C*

Normal (Gaussian) Deviates

Transformation methods generalize to more than one dimension. If $x_1, x_2, \ldots$ are random deviates with a *joint* probability distribution $p(x_1, x_2, \ldots)$ $dx_1 dx_2 \ldots$, and if $y_1, y_2, \ldots$ are each functions of all the x's (same number of y's as x's), then the joint probability distribution of the y's is

$$p(y_1, y_2, \ldots)dy_1 dy_2 \ldots = p(x_1, x_2, \ldots)\left|\frac{\partial(x_1, x_2, \ldots)}{\partial(y_1, y_2, \ldots)}\right| dy_1 dy_2 \ldots \qquad (7.2.8)$$

where $|\partial(\quad)/\partial(\quad)|$ is the Jacobian determinant of the x's with respect to the y's (or reciprocal of the Jacobian determinant of the y's with respect to the x's).

An important example of the use of (7.2.8) is the *Box-Muller* method for generating random deviates with a normal (Gaussian) distribution,

$$p(y)dy = \frac{1}{\sqrt{2\pi}} e^{-y^2/2} dy \tag{7.2.9}$$

Consider the transformation between two uniform deviates on (0,1), x_1, x_2, and two quantities y_1, y_2,

$$\begin{aligned} y_1 &= \sqrt{-2 \ln x_1} \cos 2\pi x_2 \\ y_2 &= \sqrt{-2 \ln x_1} \sin 2\pi x_2 \end{aligned} \tag{7.2.10}$$

Equivalently we can write

$$\begin{aligned} x_1 &= \exp\left[-\frac{1}{2}(y_1^2 + y_2^2)\right] \\ x_2 &= \frac{1}{2\pi} \arctan \frac{y_2}{y_1} \end{aligned} \tag{7.2.11}$$

Now the Jacobian determinant can readily be calculated (try it!):

$$\frac{\partial(x_1, x_2)}{\partial(y_1, y_2)} = \begin{vmatrix} \frac{\partial x_1}{\partial y_1} & \frac{\partial x_1}{\partial y_2} \\ \frac{\partial x_2}{\partial y_1} & \frac{\partial x_2}{\partial y_2} \end{vmatrix} = -\left[\frac{1}{\sqrt{2\pi}} e^{-y_1^2/2}\right] \left[\frac{1}{\sqrt{2\pi}} e^{-y_2^2/2}\right] \tag{7.2.12}$$

Since this is the product of a function of y_2 alone and a function of y_1 alone, we see that each y is independently distributed according to the normal distribution (7.2.9).

One further trick is useful in applying (7.2.10). Suppose that, instead of picking uniform deviates x_1 and x_2 in the unit square, we instead pick v_1 and v_2 as the ordinate and abscissa of a random point inside the unit circle around the origin. Then the sum of their squares, $R^2 \equiv v_1^2 + v_2^2$ is a uniform deviate, which can be used for x_1, while the angle that (v_1, v_2) defines with respect to the v_1 axis can serve as the random angle $2\pi x_2$. What's the advantage? It's that the cosine and sine in (7.2.10) can now be written as $v_1/\sqrt{R^2}$ and $v_2/\sqrt{R^2}$, obviating the trigonometric function calls!

We thus have

```
#include <math.h>

float gasdev(long *idum)
Returns a normally distributed deviate with zero mean and unit variance, using ran1(idum)
as the source of uniform deviates.
{
    float ran1(long *idum);
    static int iset=0;
    static float gset;
    float fac,rsq,v1,v2;

    if (*idum < 0) iset=0;                  Reinitialize.
    if (iset == 0) {                        We don't have an extra deviate handy, so
        do {
            v1=2.0*ran1(idum)-1.0;          pick two uniform numbers in the square ex-
            v2=2.0*ran1(idum)-1.0;              tending from -1 to +1 in each direction,
            rsq=v1*v1+v2*v2;                see if they are in the unit circle,
```

```
    } while (rsq >= 1.0 || rsq == 0.0);        and if they are not, try again.
    fac=sqrt(-2.0*log(rsq)/rsq);
    Now make the Box-Muller transformation to get two normal deviates. Return one and
    save the other for next time.
    gset=v1*fac;
    iset=1;                                    Set flag.
    return v2*fac;
  } else {                                     We have an extra deviate handy,
    iset=0;                                    so unset the flag,
    return gset;                               and return it.
  }
}
```

See Devroye [1] and Bratley [2] for many additional algorithms.

Writedata_ols_erf.c

```
/* writedata_ols_erf.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>
#include <math.h>
/* Declare pointer to the output file */
FILE *kp;
FILE *jp;

int main(void){

        int nrow=1000, ncol=25;
        int normaldeviates = 10000;
        double *X, *Y, *BETA, *xnorm1, *xnorm2;
        double sum, xuniform1, xuniform2, xhypot, fact;
        int i = 0;
        int j = 0;

        X = (double *) malloc (nrow*ncol*sizeof(double));
        Y = (double *) malloc (nrow*sizeof(double));
        BETA = (double *) malloc (ncol*sizeof(double));
        xnorm1 = (double *) malloc (normaldeviates*sizeof(double));
        xnorm2 = (double *) malloc (normaldeviates*sizeof(double));

/* Open the output file */
        kp = fopen("data_ols.txt","w");
        jp = fopen("data_normal.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(11);
/* Get Normal Deviates using Box-Muller Method*/
        for(i=0;i<normaldeviates;i++)
        {
                do
                {
                        xuniform1 = 2.0*( (double)rand() /
((double)(RAND_MAX)+1))-1.0;
                        xuniform2 = 2.0*( (double)rand() /
((double)(RAND_MAX)+1))-1.0;
                        xhypot = xuniform1*xuniform1 + xuniform2*xuniform2;
                }while (xhypot >= 1.0 || xhypot == 0.0);
```

```
                fact=sqrt(-2.0*log(xhypot)/xhypot);
                xnorm1[i] = xuniform1*fact;
                xnorm2[i] = xuniform2*fact;
                fprintf(jp,"%10d %12.6f %12.6f\n",i, xnorm1[i],xnorm2[i]);
        }
/* 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];
                }
/*  Add Normal (0, 1) error here*/
                Y[i]=sum + xnorm1[i];
        }

/* 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);
        free(xnorm1);
        free(xnorm2);
        return(0);
}
```

ols_read_svd_general.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 *);
void xrsquare(double *sse, double *tss, int nrow, int ncol, double *Y, double
*X, double *coef);

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, *xxinvdiag;
        double *u, *lambda, *vt;
        int *ipiv;
        double time1, time2, timedif;
        double sse, tss, stderrest=0.0, pearsonrsquare, betastderror;

        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));
        xxinvdiag = (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);
        printf("Singular Values\n");
        for(j=0;j<ncol;j++)
        {
                printf("%d %f\n",j,lambda[j]);
        }
        timedif = ( ((double) clock()) / CLOCKS_PER_SEC) - time2;
        printf("SVD took %12.3f seconds\n", timedif);
        fprintf(jp,"SVD took %12.3f 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]);
/* Save Diagonal of (X'X)-1 */
                        if(i == j)xxinvdiag[i] = 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);
/* Get Sum of Squared Error */
        xrsquare(&sse, &tss, nrow, ncol, Y, X, coef);
        /* Standard Error of the Estimate */
        stderrest = sqrt(sse/(double)(nrow-ncol));
/* Write out Coefficient Vector and Standard Errors */
        fprintf(jp,"\n\nCoefficient Vector = ");
        printf("\n\nCoefficient Vector = ");
        for(i=0;i<ncol;i++)
        {
                betastderror = stderrest*sqrt(xxinvdiag[i]);
                printf("\n%d %12.6f %12.6f", i, coef[i], betastderror);
                fprintf(jp,"\n%d %12.6f %12.6f", i, coef[i], betastderror);
        }
        printf("\n\n");

        fprintf(jp,"SSE = %12.7g\n",sse);
        printf("SSE = %12.7g\n",sse);
        fprintf(jp,"TSS = %12.7g\n",tss);
        printf("TSS = %12.7g\n",tss);
        fprintf(jp,"Standard Error of the Estimate = %12.7g\n",stderrest);
```

```
        printf("Standard Error of the Estimate = %12.7g\n",stderrest);

        pearsonrsquare = 1.0 - sse/tss;
        fprintf(jp,"Pearson R Squared = %12.7g\n",pearsonrsquare);
        printf("Pearson R Squared = %12.7g\n",pearsonrsquare);

        free(X);
        free(Y);
        free(XprimeX);
        free(XXinvX);
        free(coef);
        free(xxinvdiag);
        free(ipiv);
        free(u);
        free(lambda);
        free(vt);
        timedif = ( ((double) clock()) / CLOCKS_PER_SEC) - time1;
        printf("The total elapsed time of the program is %12.3f seconds\n",
timedif);
        fprintf(jp,"\nThe total elapsed time of the program is %12.3f
seconds\n", timedif);

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

}
/*
 *  Pearson R-Square Subroutine -- Computes r-square for simple OLS
 */
void xrsquare(double *sse, double *tss, int nrow, int ncol, double *Y, double
*X, double *coef)
{
        int i, j;
        double sum, sum2, sum3, ymean;
        /* Calculate Y */
        sum2=0.0;
        ymean=0.0;
        for(i=0;i<nrow;i++)
        {
                ymean=ymean+Y[i];
                sum=0.0;
                for(j=0;j<ncol;j++)
                {
                        sum=sum+coef[j]*X[i+j*nrow];
                }
/*  Calculate the SSE here*/
                sum2=sum2+(sum - Y[i])*(sum - Y[i]);
        }
        ymean=ymean/(double)(nrow);
        sum3=0.0;
        for(i=0;i<nrow;i++)
        {
/*  Calculate the TSS here*/
                sum3=sum3+(Y[i]-ymean)*(Y[i]-ymean);
        }
        *sse = sum2;
```

```
        *tss = sum3;
}
/*

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));


        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);
        free(work);
        free(a);
}
```