Statistics 243: *homework 2*

William J. De Meo

November 2, 1997

1 Mean and Variance Algorithms

We construct three algorithms – desk calculator, provisional means, and centered about first observation – for computing the mean and variance of a list of numbers. The C source file `moment.c`, whose listing is given in the Appendix in section 4.2, contains routines which carry out each of these algorithms. The output of the program `illtest.c`, which calls the three routines, appears in section 4.4. As is clear from this output, we first detect relative errors of size 0.1 in the desk calculator algorithm when the coefficient of variation is near 10^{-8}, which is about the square root of machine epsilon. The provisional means algorithm and the centering algorithm appear to do fine until the coefficient of variation is of order machine epsilon.

2 Uniform Random Numbers

The multiplicative congruential generator,

$$X_{n+1} = aX_n \pmod{m}$$

has been shown to work well both theoretically and empirically if the multiplier a and the modulus m are chosen carefully. The modulus 2^{32} and multiplier a such that $a \bmod 8$ is 3 or 5 have been proposed. In order to implement this while avoiding overflow, we can use Schrage's [3] *approximate factorization algorithm.* Factorize m as

$$m = aq + r \tag{1}$$

That is, q is the integer part of m/a while $r = m \bmod a$. If $r < q$ and $0 < X_n < m - 1$, it can be shown that

$$aX_n \bmod m = \begin{cases} a(X_n \bmod q) - r[X_n/q] & \text{if it is } \geq 0 \\ a(X_n \bmod q) - r[X_n/q] + m & \text{otherwise} \end{cases} \tag{2}$$

where $[x]$ represents the largest integer no greater than x. If we put $m = 2^{32}$ and $a = 16805$ (which satisfies the requirement $a \bmod 8 = 5$), then, performing a little paper and pencil algebra, we can see that the values $q = 255576$ and $r = 12614$ do the trick. Nonetheless, in order to implement the algorithm for $m = 2^{32}$, we still need to access the value 2^{32} which will cause overflow on many platforms. The more portable suggestion, described in Park and Miller [2], is called the "Minimal Standard" generator, and is based on the choices,

$$a = 7^5 = 16807 \quad m = 2^{31} - 1 = 2147483647$$

Applying Schrage's approximation to these values gives $q = 127773$ and $r = 2836$. We use these values to implement the version of the multiplicative congruential generator given by (2). The C program and results appear in section 4.5. As we expect, the output shows that the average of the pseudo-random uniform(0,1) random numbers is near $1/2$, and the variance is near $1/12 = 0.08\overline{3}$.

3 Normal Random Numbers

We construct normal random numbers using the polar method. Two random normal(0,1) numbers, x_1, x_2, are constructed from two uniform(0,1) numbers, u_1, u_2 (e.g. the kind described above) using the following steps:

1. generate $u_1, u_2 \sim$ uniform(0,1)
2. $v_j = 2u_j - 1, \;\; j = 1, 2$ so that $v_j \sim$ uniform(-1,1)
3. $s = v_1^2 + v_2^2$
4. if $s \geq 1$ go back to first step and generate new u_1, u_2
5. $x_1 = v_1\sqrt{\frac{-2\log s}{s}}$ and $x_2 = v_2\sqrt{\frac{-2\log s}{s}}$

This results in $x_j \sim$ normal(0,1), $j = 1, 2$. The probability that we will be forced to generate new uniforms (i.e. return to the first step in the algorithm) is the probability that $v_1^2 + v_2^2 < 1$ which, since v_j are uniform(-1,1), is just $\pi/4$. Therefore, using the negative binomial distribution, it is easy to see that the expected number of rejections is $nq/p \approx 0.27n$. When we implement the polar method in the program `rand.c` (section 4.7), we ask the user how many normal random numbers are required, then the program generates twice as many uniform random numbers, from which an attempt is made at getting the required number of normals.

As can be seen from the output in section 4.9, the means and variances (we expect around 0 and 1) look reasonable. Furthermore, we expect only about five percent of the observation to be more than 1.96 in absolute value. This also appears to be the case for our samples.

4 Appendix

4.1 source code for illtest.c

```
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   illtest.c

   purpose:  Test the routines dmoment, pmoment, cmoment in
             file moment.c on illconditioned data .
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "prototypes.h"

#define small 0.0000001
#define BIG 10

main()
{
     double  *relerr, *dave, *dvar, *pave, *pvar, *cave, *cvar, *x;
     double VAR, coef;
     unsigned long i=0,n, count=0;
```

```
int flag=0, j;
FILE *ofp;

dave = dmalloc(1);
dvar = dmalloc(1);
pave = dmalloc(1);
pvar = dmalloc(1);
cave = dmalloc(1);
cvar = dmalloc(1);

printf("\nHow many data values? ");
scanf("%d",&n);
x = dmalloc(n);

printf("\nEnter data values at the > prompt. ");
for(i=0;i<n;i++){
     printf("\n> ");
     scanf("%lf",&x[i]);
}
dmoment(x, n, dave, dvar);
pmoment(x, n, pave, pvar);
cmoment(x, n, cave, cvar);

ofp = fopen("illout", "w");
fprintf(ofp,"\nx = \n");
for(i=0; i<n; i++) fprintf(ofp," %lf ",x[i]);
fprintf(ofp,"\n\n(dave,dvar) = (%lf,%lf)",*dave,*dvar);
fprintf(ofp,"\n\n(pave,pvar) = (%lf,%lf)",*pave,*pvar);
fprintf(ofp,"\n\n(cave,cvar) = (%lf,%lf)\n",*cave,*cvar);

/* Ill-conditioning */

VAR = *dvar; /* the "true" mean and variance */
relerr = dmalloc(3);

while(flag<2)
{
     count++;
     for(i=0;i<n;i++) x[i]+=pow(2,count);

     dmoment(x, n, dave, dvar);
     pmoment(x, n, pave, pvar);
     cmoment(x, n, cave, cvar);

     relerr[0] = (*dvar - VAR)/VAR;
     relerr[1] = (*pvar - VAR)/VAR;
     relerr[2] = (*pvar - VAR)/VAR;
     coef = sqrt(VAR)/(*pave);
```

```
          /* If any relative errors are bigger than small, */
          /* print a warning.  If any are bigger than BIG, exit. */
          for(j=0;j<3;j++) {
               if(relerr[j] > small || -relerr[j] > small)
                  fprintf(ofp,
                    "\nWarning: on iteration %u, relerr[%d] = %g, coef = %g",
                    count,j,relerr[j],coef);
               if(relerr[j] > BIG || -relerr[j] > BIG) {
                  fprintf(ofp,
                    "\nERROR:   on iteration %u, relerr[%d] = %g, coef = %g",
                    count, j,relerr[j],coef);
                  flag++;
               }
          }
     }
     fprintf(ofp,"\n\n%s%u%s%s%lf %s%lf %s%lf %s%lf\n ",
             "On iteration ", count,
             " at least two gross errors:\n",
             "VAR = ", VAR,
             "\ndvar = ", *dvar,
             "\npvar = ", *pvar,
             "\ncvar = ", *cvar);
     fclose(ofp);
}
```

4.2 source code for moment.c

```
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   moment.c

   Three routines for for computing the mean and variance
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/* dmoment: The desk calculator algorithm
   arguments:
              data = a nx1 array of doubles
              n = length of data[]
              ave =(on entry)= the address of *ave (call by reference)
              ave =(on exit)= average of data[]
              var =(on entry)= the address of *var (call by reference)
              var =(on exit)= variance of data[]
              */
void dmoment(double * data, unsigned int n, double *ave, double *var)
{
     unsigned int i;

     *ave=0; *var=0;
```

```
     for(i=0;i<n;i++){
          *ave += data[i];
          *var += data[i]*data[i];
     }
     *ave /= (double) n;
     *var = (*var - (double)n*((*ave)*(*ave)))/(double)(n-1);
}

/* pmoment: The provisional means algorithm
   arguments:
              data = a nx1 array of doubles
              n = length of data[]
              ave =(on exit)= the average of data[]
              var =(on exit)= the variance of data[]
              */
void pmoment(double * data, unsigned int n, double *ave, double *var)
{
     unsigned int i;

     *ave = data[0];
     *var=0;

     for(i=1;i<n;i++){
          *var +=((double)i/(double)(i+1))*(data[i]-*ave)*(data[i]-*ave);
          *ave *= ((double)i/(double)(i+1));
          *ave += data[i]/(double)(i+1);
     }
     *var /= (double)(n-1);
}

/* cmoment: centering around the first observation
   arguments:
              data = a nx1 array of doubles
              ave =(on exit)= the average of data[]
              var =(on exit)= the variance of data[]
              */
void cmoment(double * data, unsigned int n, double *ave, double *var)
{
     unsigned int i;

     *ave=0;
     *var=0;
     for(i=1;i<n;i++){
          *ave += data[i] - data[0];
          *var += (data[i] - data[0])*(data[i] - data[0]);
     }
     *ave /= (double)n;
     *var = (*var - (double)n * ((*ave)*(*ave)))/(double)(n-1);
```

```
     *ave += data[0];
}
```

4.3 makefile for illtest (and its precursor momtest)

```
# Makefile for illtest momtests

HOME = /home/chip
HOMEBASE = $(HOME)/classes/s243/programs/hw2
HOMELIB = $(HOME)/lib
INCL = -I$(HOMELIB)/include
CC = gcc

# Ill-conditioned data test
illtest: illtest.o moment.o $(HOMELIB)/util_lib.a
        $(CC) -o illtest -lm illtest.o moment.o $(HOMELIB)/util_lib.a

illtest.o: illtest.c
        $(CC) $(INCL) -c illtest.c

# Ill-conditioned data test (debugging)
illtest.db: illtest.do moment.do $(HOMELIB)/util_lib.a
        $(CC) -o illtest.db -lm illtest.do moment.do $(HOMELIB)/util_lib.a

illtest.do: illtest.c
        $(CC) $(INCL) -o illtest.do -c -g illtest.c

# (moment.do: is made below)


# Ordinary routine test
momtest: momtest.o moment.o $(HOMELIB)/util_lib.a
        $(CC) -o momtest momtest.o moment.o $(HOMELIB)/util_lib.a

momtest.o: momtest.c
        $(CC) $(INCL) -c momtest.c

moment.o: moment.c
        $(CC) -c moment.c

# Ordinary routine test debugging
momtest.db: momtest.do moment.do $(HOMELIB)/util_lib.a
        $(CC) -o momtest.db momtest.do moment.do $(HOMELIB)/util_lib.a

momtest.do: momtest.c
        $(CC) $(INCL) -o momtest.do -c -g momtest.c

moment.do: moment.c
```

```
	$(CC) -o moment.do -c -g moment.c
```

4.4 output from illtest

The following output was produced upon entering five consective digits (shown in the `x` vector below).

```
x =
 1.000000  2.000000  3.000000  4.000000  5.000000

(dave,dvar) = (3.000000,2.500000)

(pave,pvar) = (3.000000,2.500000)

(cave,cvar) = (3.000000,2.500000)

Warning: on iteration 25, relerr[0] = 0.1, coef = 2.35608e-08
Warning: on iteration 26, relerr[0] = 0.1, coef = 1.17804e-08
Warning: on iteration 27, relerr[0] = -1.5, coef = 5.8902e-09
Warning: on iteration 28, relerr[0] = -1.5, coef = 2.9451e-09
Warning: on iteration 29, relerr[0] = -1.5, coef = 1.47255e-09
Warning: on iteration 30, relerr[0] = -1.4, coef = 7.36275e-10
Warning: on iteration 31, relerr[0] = -1, coef = 3.68138e-10
Warning: on iteration 32, relerr[0] = -1, coef = 1.84069e-10
Warning: on iteration 33, relerr[0] = -1, coef = 9.20344e-11
Warning: on iteration 34, relerr[0] = -1, coef = 4.60172e-11
Warning: on iteration 35, relerr[0] = -1, coef = 2.30086e-11
Warning: on iteration 36, relerr[0] = -1, coef = 1.15043e-11
Warning: on iteration 37, relerr[0] = -1, coef = 5.75215e-12
Warning: on iteration 38, relerr[0] = -1, coef = 2.87607e-12
Warning: on iteration 39, relerr[0] = -1, coef = 1.43804e-12
Warning: on iteration 40, relerr[0] = -1, coef = 7.19019e-13
Warning: on iteration 41, relerr[0] = -1, coef = 3.59509e-13
Warning: on iteration 42, relerr[0] = -1, coef = 1.79755e-13
Warning: on iteration 43, relerr[0] = -1, coef = 8.98773e-14
Warning: on iteration 44, relerr[0] = -1, coef = 4.49387e-14
Warning: on iteration 45, relerr[0] = -1, coef = 2.24693e-14
Warning: on iteration 46, relerr[0] = -1, coef = 1.12347e-14
Warning: on iteration 47, relerr[0] = -1, coef = 5.61733e-15
Warning: on iteration 48, relerr[0] = -1, coef = 2.80867e-15
Warning: on iteration 49, relerr[0] = -1, coef = 1.40433e-15
Warning: on iteration 50, relerr[0] = -1, coef = 7.02167e-16
Warning: on iteration 51, relerr[0] = -9.0072e+14, coef = 3.51083e-16
ERROR:   on iteration 51, relerr[0] = -9.0072e+14, coef = 3.51083e-16
Warning: on iteration 51, relerr[1] = 0.22, coef = 3.51083e-16
Warning: on iteration 51, relerr[2] = 0.22, coef = 3.51083e-16
Warning: on iteration 52, relerr[0] = 7.20576e+15, coef = 1.75542e-16
ERROR:   on iteration 52, relerr[0] = 7.20576e+15, coef = 1.75542e-16
Warning: on iteration 52, relerr[1] = 0.63, coef = 1.75542e-16
Warning: on iteration 52, relerr[2] = 0.63, coef = 1.75542e-16
```

```
On iteration 52 at least two gross errors:
VAR = 2.500000
dvar = 18014398509481984.000000
pvar = 4.075000
cvar = 4.000000
```

4.5 source code for unif.c

```
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  unif.c

  Generate uniform(0,1) random variables
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

#include <math.h>
#include "prototypes.h"

#define A 16807
#define M 2147483647
#define Q 127773
#define R 12614

double unif(long *X);

double unif(long *X)
{
     long k;
     double unif;

     k = (*X)/Q;                    /* [x/q]                      */
     *X = A * (*X - k*Q) - R*k;     /* x - k*q = x mod q          */
     if(*X < 0) *X += M;
     unif = (*X)*((double)1.0/M);  /* make it uniform on (0,1) */
     return(unif);
}

main()
{
     unsigned long n, i;
     double *u, *ave, *var;
     long *X;

     ave = dmalloc(1);
     var = dmalloc(1);
     X = lmalloc(1);
     *X = time('\0');
```

```
     printf("\nHow many uniform random variables? ");
     scanf("%d",&n);
     u = dmalloc(n);

     printf("The unif(0,1) random variables are:\n\n");
     for(i=0;i<n;i++)
     {
          u[i] = unif(X);
          printf("%lf  ",u[i]);
          if((i+1)%6 == 0) printf("\n");
     }
     cmoment(u, n, ave, var);
     printf("\nAverage = %lf, Variance = %lf\n",*ave,*var);
}
```

4.6 output from unif.c

The following output file was produced by invoking the `unif` program three times, each time requesting 100 pseudo-random numbers.

```
The unif(0,1) random variables are:

0.891198  0.293737  0.814619  0.236471  0.346485  0.339607
0.756221  0.751822  0.808802  0.470374  0.539402  0.692317
0.718020  0.707460  0.234207  0.291148  0.308513  0.156884
0.735386  0.579936  0.947248  0.320143  0.622354  0.863478
0.406654  0.599327  0.847265  0.920982  0.869615  0.549835
0.032264  0.254038  0.594280  0.017005  0.804019  0.091891
0.406409  0.488968  0.041630  0.675458  0.367903  0.312026
0.194511  0.129265  0.540316  0.047268  0.421787  0.945138
0.866346  0.605256  0.483753  0.406308  0.788354  0.799639
0.464047  0.202129  0.170173  0.076436  0.659413  0.709490
0.336908  0.385586  0.515348  0.411696  0.348228  0.635149
0.894460  0.114777  0.048596  0.746734  0.301957  0.963401
0.814759  0.589734  0.606227  0.816773  0.444151  0.814875
0.535618  0.092193  0.483548  0.952049  0.008366  0.602685
0.274745  0.609963  0.593704  0.333985  0.258171  0.060376
0.742472  0.666400  0.129082  0.465106  0.007969  0.928584
0.648206  0.353985  0.404499  0.388195

Average = 0.495443, Variance = 0.074287

The unif(0,1) random variables are:

0.891949  0.921305  0.309861  0.812346  0.039243  0.546151
0.113200  0.550127  0.947623  0.635354  0.348007  0.924881
0.401476  0.582573  0.264982  0.540465  0.554694  0.701083
0.054954  0.603580  0.315017  0.466426  0.188175  0.636627
0.741894  0.959982  0.342052  0.846062  0.697293  0.351566
```

```
0.735897  0.168858  0.988652  0.197509  0.518474  0.947082
0.540708  0.636511  0.790586  0.318743  0.093083  0.432440
0.980287  0.609730  0.690539  0.843903  0.419434  0.389202
0.289199  0.541784  0.723724  0.574020  0.518557  0.355256
0.764091  0.027336  0.434970  0.509588  0.609256  0.715255
0.238892  0.040068  0.425652  0.899911  0.741362  0.008386
0.943828  0.849288  0.921455  0.825286  0.522647  0.090202
0.022260  0.119488  0.218426  0.065443  0.892779  0.865183
0.066475  0.234757  0.545922  0.270902  0.024762  0.167835
0.786539  0.294403  0.017080  0.069918  0.100711  0.638742
0.291831  0.779917  0.002764  0.455749  0.736225  0.672166
0.046296  0.090959  0.737707  0.590821

Average = 0.489667, Variance = 0.088923

The unif(0,1) random variables are:

0.892286  0.577408  0.444576  0.963186  0.187000  0.893340
0.294606  0.412780  0.557117  0.427264  0.993391  0.841828
0.542357  0.347577  0.698716  0.273145  0.731618  0.242505
0.758537  0.667052  0.093947  0.961029  0.935320  0.846348
0.506152  0.851337  0.361794  0.642129  0.210016  0.719121
0.217387  0.604308  0.558331  0.825282  0.447318  0.044756
0.206730  0.503358  0.905271  0.820670  0.934329  0.204014
0.855518  0.621555  0.422413  0.464824  0.263866  0.767518
0.614276  0.095455  0.309792  0.658259  0.302246  0.833454
0.803811  0.587230  0.526927  0.019076  0.607356  0.790122
0.525837  0.699197  0.354821  0.453175  0.482786  0.139194
0.425180  0.959356  0.821873  0.161693  0.566057  0.668648
0.918881  0.570908  0.205213  0.004371  0.457523  0.560433
0.147534  0.587341  0.400331  0.339136  0.834900  0.095730
0.922746  0.519107  0.588993  0.154513  0.885645  0.965169
0.517298  0.187958  0.993685  0.786826  0.131531  0.629180
0.583683  0.911382  0.524533  0.790230

Average = 0.551615, Variance = 0.074321
```

4.7 source code for random.c, normal.c, and uniform.c

```
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  random.c

  Generate normal(0,1) random variables
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

#include <math.h>
```

```
#include <stdio.h>
#include "prototypes.h"

double max(double a, double b, double c);
double min(double a, double b, double c);

main()
{
     unsigned long n, m, i,j;
     double *u, *g, *gtemp, *ave, *var, minimum = 100, maximum = -100;
     long *X;
     FILE *ofp;

     ave = dmalloc((unsigned long)1);
     var = dmalloc((unsigned long)1);
     gtemp = dmalloc((unsigned long)2);
     X = lmalloc((unsigned long)1);
     *X = time('\0');

     printf("\nHow many normal random variables? ");
     scanf("%u",&m);
     n = 2*m; /* generate two times as many uniforms */

     ofp = fopen("norm.out", "a");

     u = dmalloc(n);
     for(i=0;i<n;i++)
          u[i] = unif(X);

     cmoment(u, n, ave, var);
     fprintf(ofp,"\n\nUAverage = %lf, UVariance = %lf\n",*ave,*var);

     fprintf(ofp,"\nThe normal(0,1) random variables are:\n\n");
     g = dmalloc(m);
     for(i=0;i<m;i++)
     {
          if(normal(u, gtemp,n)==1)
          {
               g[i] = gtemp[0];
               g[++i] = gtemp[1];
               maximum = max(maximum,g[i-1], g[i]);
               minimum = min(minimum,g[i-1], g[i]);
               fprintf(ofp,"%lf  %lf  ",g[i-1], g[i]);
               if((i+1)%6 == 0) fprintf(ofp,"\n");
          }
          else  /* didn't get enough normals -- need new uniforms */
          {
               i = 0;
```

```
               printf("\n\nGenerating different unif(0,1) variables...\n\n");
               for(j=0;j<n;j++)
                    u[j] = unif(X);
          }
     }
     cmoment(g, m, ave, var);
     fprintf(ofp,"\n\nAverage = %lf, Variance = %lf, min = %lf, max = %lf \n"
             ,*ave,*var, minimum, maximum);
     fclose(ofp);
}

double max(double a, double b, double c)
{
     double max;
     if(a >= b) max = a;
     else max = b;

     if(max >= c) return(max);
     else return(c);
}

double min(double a, double b, double c)
{
     double min;
     if(a <= b) min = a;
     else min = b;

     if(min <= c) return(min);
     else return(c);
}


/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   normal.c

     Routine for constructing normal(0,1) random numbers
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
#include <math.h>
#include "prototypes.h"

int normal(double *u, double *x, unsigned long n)
{
     double s;
     static int i = -2;

     do{
          i += 2;
          s = (2*u[i] - 1)*(2*u[i] - 1) + (2*u[i+1] - 1)*(2*u[i+1] - 1);
          if(i > n-2)
```

```
          {
               i=-2;

               return(0);
          }
     }while(s >= 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);

}

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  uniform.c

  Generate uniform(0,1) random variables
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

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

#define A 16807
#define M 2147483647
#define Q 127773
#define R 12614

double unif(long *X)
{
     long k;
     double unif;

     k = (*X)/Q;                    /* [x/q]                      */
     *X = A * (*X - k*Q) - R*k;     /* x - k*q = x mod q          */
     if(*X < 0) *X += M;
     unif = (*X)*((double)1.0/M);  /* make it uniform on (0,1) */
     return(unif);
}
```

4.8 makefile for rand

```
# Makefile for rand

HOME = /home/chip
HOMEBASE = $(HOME)/classes/s243/programs/hw2
HOMELIB = $(HOME)/lib
INCL = -I$(HOMELIB)/include
```

```
CC = gcc

rand: rand.o normal.o uniform.o moment.o $(HOMELIB)/util_lib.a
        $(CC) -o rand -lm rand.o normal.o uniform.o moment.o $(HOMELIB)/util_lib.a

rand.o: rand.c
        $(CC) $(INCL) -c rand.c

normal.o: normal.c
        $(CC) $(INCL) -c normal.c

uniform.o: uniform.c
        $(CC) $(INCL) -c uniform.c

unif.o: unif.c
        $(CC) $(INCL) -c unif.c

moment.o: moment.c
        $(CC) -c moment.c
```

4.9 output file from rand

Running the `rand` program ten times produced the ten samples below. A sample size of 25 was specified each time.

```
UAverage = 0.456250, UVariance = 0.069789

The normal(0,1) random variables are:

-0.245908  0.420663  -1.339026  -0.224088  2.711854  -1.386384
-0.602924  0.550597  -1.717494  -0.196892  -0.737359  -0.072223
-0.358752  0.774570  0.633406  -0.436363  -0.245075  1.764727
0.176432  -0.960782  -0.308958  -1.033925  -0.576253  -1.469689
-0.032555  -0.653885

Average = -0.196496, Variance = 0.987588, min = -1.717494, max = 2.711854


UAverage = 0.524961, UVariance = 0.083246

The normal(0,1) random variables are:

0.757996  -1.387067  0.139614  -0.974996  -0.298773  0.701019
-0.027365  0.402372  -0.314423  1.978237  0.801608  -0.854846
0.051573  0.909785  0.987436  -1.322042  -0.066928  -0.397302
-0.828935  1.112062  0.230128  0.850592  -0.540374  0.136476
-2.533336  1.666251
```

```
Average = -0.019500, Variance = 0.947179, min = -2.533336, max = 1.978237


UAverage = 0.535653, UVariance = 0.073212

The normal(0,1) random variables are:

1.039667  -0.212589  1.648895  1.182762  -2.099287  1.118224
1.271166  -0.350663  1.373938  -1.715097  -1.730662  2.434986
-1.329970  0.265719  -0.738482  1.443447  -0.200154  2.674323
-0.460446  1.509911  1.414562  -1.026580  0.154165  0.308496
-2.082391  0.012989

Average = 0.235758, Variance = 1.944441, min = -2.099287, max = 2.674323


UAverage = 0.561852, UVariance = 0.083079

The normal(0,1) random variables are:

-1.627013  -1.946721  0.641673  1.599386  -0.050435  0.872663
-1.023078  -0.172720  -0.479716  -2.624229  -1.393816  -0.781216
-1.428000  -0.175682  0.804378  0.006943  -0.579453  0.469167
-1.037056  -1.589434  -1.849918  -0.973552  0.777605  -0.021101
0.015179  0.779816

Average = -0.502646, Variance = 1.102117, min = -2.624229, max = 1.599386


UAverage = 0.471777, UVariance = 0.117916

The normal(0,1) random variables are:

-0.180602  -0.033266  -0.491674  -0.210818  -0.005621  0.935598
-0.750740  -0.150154  -0.651331  -0.605196  -0.568524  0.256189
-0.499665  -0.216110  0.530548  -0.654962  -0.407916  -0.499960
0.056185  -0.187132  -0.081403  0.359179  0.241485  -0.254034
0.882858  -0.725366

Average = -0.127483, Variance = 0.210574, min = -0.750740, max = 0.935598


UAverage = 0.468296, UVariance = 0.078411

The normal(0,1) random variables are:

-0.538085  2.315870  -0.006784  -1.196539  1.234952  -2.092003
0.278209  1.853595  -0.987914  1.145709  0.997270  0.430581
```

```
2.112981  -1.594997  0.162695  0.883180  0.671533  0.501754
-0.508260  -0.821739  -1.440829  0.851804  -1.373836  1.153124
0.367494  -1.014375

Average = 0.175991, Variance = 1.449950, min = -2.092003, max = 2.315870


UAverage = 0.515804, UVariance = 0.097365

The normal(0,1) random variables are:

1.799999  -0.392986  -0.565985  0.775871  -1.454413  2.320997
-0.352454  -0.103099  -2.437839  -0.288213  0.565715  -0.725542
-0.475247  0.863397  -0.197961  0.213789  0.202876  0.665955
0.879525  0.339340  0.554495  -0.809796  0.180958  -0.802160
-0.112587  1.081356

Average = 0.025785, Variance = 0.953816, min = -2.437839, max = 2.320997


UAverage = 0.489400, UVariance = 0.088722

The normal(0,1) random variables are:

-1.164689  2.125476  -1.087806  -1.627695  -0.148151  -0.585310
0.078002  1.434552  -0.490166  0.254162  0.830430  0.382266
0.550786  0.555514  -0.691805  -1.458980  -0.299309  -0.331736
-1.084327  1.043996  -0.185769  1.058329  -1.386962  0.052203
0.452991  -0.702211

Average = -0.068960, Variance = 0.921992, min = -1.627695, max = 2.125476


UAverage = 0.565793, UVariance = 0.078631

The normal(0,1) random variables are:

-0.164288  0.031684  1.785622  0.642314  1.490821  -1.436317
-0.340308  -0.757504  1.372027  1.243159  -3.225931  -0.149886
0.932021  0.349455  -2.775168  0.033131  0.042626  -0.016562
0.354986  2.069033  1.913653  0.436602  0.502835  0.265024
0.438090  -1.048888

Average = 0.201485, Variance = 1.621517, min = -3.225931, max = 2.069033


UAverage = 0.455411, UVariance = 0.081871
```

```
The normal(0,1) random variables are:

-0.177334  -0.208764  0.414832  -0.465136  -0.305812  0.502667
-0.042717  0.714116  -0.293542  2.073023  -0.052186  -1.267175
-0.788828  0.296157  0.881320  0.702763  -0.114213  -0.010150
-0.443612  0.101403  0.648522  -1.565041  1.401154  -0.953936
1.095878  -1.801031

Average = 0.085736, Variance = 0.678109, min = -1.801031, max = 2.073023
```

References

[1] M. Loukides and A. Oram *Programming with GNU Software*. O'Reilly and Associates, Inc., Sebastopol, 1997.

[2] Park, S., and Miller, K. *Communications of the ACM,*, vol. 31, pp. 1192-1201, 1988.

[3] Schrage, L. *ACM Transactions on Mathematical Software,* vol. 5, pp. 132-138, 1979.