25 Random number generators

(25.1) What is a random number? **In theory,** a random number, or a random sequence of numbers, is one which cannot be produced by a program shorter than the number itself (Kolmogorov complexity).

In practice, we use sequences of numbers which are produced by extremely short programs! By the Kolmogorov criterion, they are certainly *not* random. These should be called *pseudo-random,* but the 'pseudo-' is generally forgotten.

(25.2) Until recent years, the following method, linear congruential random number generator, was used.

$$\boxed{X_{n+1} = (aX_n + c) \bmod m}$$

Here, a, c, X_0, and m are constants, chosen suitably. Typically $m = 2^{31}$. In `stdlib.h`, `RAND_MAX` (which is $m-1$) is given as $2^{31}-1$, over 2 billion.

```
man rand
```

---

it will tell you a little more (probably inaccurate, judging by a look at stdlib.h). The numbers $X_0, X_1, X_2, \ldots$ are returned by successive calls to `rand()` (which remembers the previous number in a `static` variable).

(25.3) **The sequence is always the same.** The main point is that it 'looks' random under various statistical and empirical tests.

(25.4) Obviously some choices of constant are bad. For example, if $a = 0$ then $X_n = c$ for all n. If $a = 1$ then $X_n = (X_0 + nc) \bmod m$, which neither looks random nor is. We may assume that $a \geq 2$.

In this case it is easily shown by induction on n that

$$X_n = (a^n X_0 + \frac{a^n - 1}{a - 1} c) \bmod m.$$

(25.5) Since the numbers are between 0 and $m-1$, eventually we get a repeated value, so for some $n \geq 0$ and $d > 0$, $X_n = X_{n+d}$. The smallest such d is called the *period.*

(25.6) **Using genuinely random seeds.** A simple method is to read the system clock, getting the microsecond part of the time. The chances against getting the same seed twice are about a million to one. **Don't reset the seed** in the middle of a program. Refer to the programming notes.

(25.7) **Producing various random distributions.** Use `man drand48`. Most often, we want a random number between 0 and $b-1$ for some smallish bound b. The obvious method (which I have used in ignorance for many years) is `rand() % b`. The manuals warn us not to do this. The reason is simple. Suppose a linear congruential generator is used. Suppose b divides m. Then one is effectively using the generator

$$X_{n+1} = (aX_n + c) \bmod b$$

whose period is $\leq b$: hardly a good choice.

The recommended method is, for example, to take the double-precision number `b * drand48 ()` and round it down to the nearest integer. If `d` is an `int`, then

```
d = b * drand48();
d = d % b;
```

will accomplish this. This belongs to the programming notes.

(25.8) Linear congruential generators have the property that if m is a power of 2 (which it usually is), then the low-order bits are not 'random'. (This follows from the previous paragraph.)

25.1 Example: estimating π

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

double randval ()
{
  static int first = 1;
  if ( first )
  {
    struct timeval tv;
    gettimeofday ( & tv, NULL );
    srand48 ( tv.tv_usec );
    first = 0;
  }

  return drand48 ();
}

main( int argc, char * argv [] )
{
  int n = atoi ( argv[1] );

  int i;
  double sum = 0;
  for (i=0; i<n; ++i)
  {
    double x = randval();
    double y = randval();
    if ( x*x + y*y  <= 1 )
      sum += 1;
  }
  printf("%d trials, pi is %f\n", n, 4*sum/n );
}
```

```
% a.out 10000
10000 trials, pi is 3.117600
% a.out 10000
10000 trials, pi is 3.147200
% a.out 10000
10000 trials, pi is 3.160400
```

There is a special-purpose technique for generating *normally distributed* random numbers. Of course, the usual random number generator produces random numbers in the range $[0, 1)$ (so 1 never happens). Actually, 0 would break the program.

```
#include <stdio.h>
#include <ctype.h>
#include <math.h>
#include <stdlib.h>
#include <sys/time.h>

// Marsaglia method
// U,V uniform on unit disc
// scale by sqrt((-ln S)/S) where S=U^2+V^2


void two_normally( double * x, double * y )
{
  double v1, v2, s, mul;

  int found = 0;

  while ( ! found )
  {
    v1 = 2 * drand48() - 1;
    v2 = 2 * drand48() - 1;
    s = v1*v1 + v2*v2;
    found = ( s < 1 );
  }

  mul = sqrt ( - 2 * log(s) / s );
  *x = v1 * mul;
  *y = v2 * mul;
}


main ( int argc, char * argv[] )
{
```

```
  struct timeval tv;
  int i,n;

  n = atof ( argv[1] );
  double sum = 0, sumsquares = 0;

  gettimeofday ( &tv, NULL );
  srand48 ( tv.tv_usec );

  double x,y;
  for (i=0; i<n; i += 2)
  {
    two_normally ( &x, &y );
    sum += x;
    sumsquares += x*x;
    printf("%f\n", x);
    if ( i+1 < n )
    {
      sum += y;
      sumsquares += y*y;
      printf("%f\n", y);
    }
  }
  double sample_mean = sum/n;
  double variance = (sumsquares - n * sample_mean * sample_mean)/(n-1);
  double sample_standard_dev = sqrt (variance);
  printf("n %d sample mean %f sample standard deviation %f\n",
        n, sample_mean, sample_standard_dev);

}
gcc -lm normal.c
a.out 4
prompt% a.out 4
-1.250447
-0.114427
-1.878096
-1.941784
n 4 sample mean -1.296189 sample standard deviation 0.847361
% a.out 4   (numbers should be N(0,1)).  Not very close.
-0.713306
-1.578731
1.323076
0.233974
n 4 sample mean -0.183747 sample standard deviation 1.247854
            Closer.
```

How does this work? It's best explained by describing a related method. Here is the Box-Muller version of `two_normally`.

```
// Box-Muller method
// U,V random (0,1)
// X = sqrt(-2 ln U) cos(2 pi V), Y = sqrt(-2 ln U) sin (2 pi V)


void two_normally( double * x, double * y )
{
  double u, v, mul;

  u = drand48();
  v = drand48();

  mul = sqrt ( - 2 * log(u) );
  *x =  mul * cos ( 2 * M_PI * v );
  *y =  mul * sin ( 2 * M_PI * v );
}
```

This works as follows. The multiplier M is a function of U, taking values in $(0, \infty)$, and (as one can check) with probability density function

$$\frac{1}{2}e^{-m/2}$$

But this PDF is χ^2_2. Also, $m = x^2 + y^2$. so $X^2 + Y^2$ has the PDF of χ^2_2. Also, X, Y are independent. It follows that X, Y have the joint PDF of two independent $N(0, 1)$ variables, which is what was wanted.

In the Marsaglia version, $S = U^2 + V^2$ is (since if $|S| \geq 1$ S is discarded) uniformly distributed between 0 and 1. If we write $u = S \cos\theta$ and $v = S \sin\theta$ then

$$X = U\sqrt{-\frac{2\ln S}{S}} = \sqrt{-2\ln S}\cos\theta$$

and similarly Y, so the formula is a variation of the Box-Muller formula.