

Random Numbers

February 21, 2006

The Scilab functions used in this lecture are in the file `l1.sci` in the directory for the lecture.

Contents

1	Review of Probability Theory	2
1.1	Continuous Distributions	2
1.2	Discrete Distributions	3
2	Random Numbers	4
2.1	Uniform Random Numbers	4
2.2	Linear Congruential Generators	5
2.3	Combining Random Number Generators	6
2.4	Change of Variable	7
2.5	Non-Uniform Random Numbers	8
3	Random Numbers in Scilab	8
3.1	<code>rand</code>	8
3.2	<code>grand</code>	10

1 Review of Probability Theory

I have presented here a few basic definitions, mean and variance for instance, and some common probability distributions. I don't assume much knowledge of probability; besides the definitions below an intuitive understanding should be OK. I certainly don't assume any technical proficiency in the subject.

1.1 Continuous Distributions

Continuous probability distributions are described by **probability densities**, i.e. real valued functions $\rho(x)$ with the properties:

1. $\rho(x) \geq 0$.
2. $\int_{-\infty}^{\infty} \rho(x) dx = 1$.
3. $\text{Prob}(a \leq x \leq b) = \int_a^b \rho(x) dx$.

The **distribution function** associated with the density $\rho(x)$ is defined by

$$F(x) = \int_{-\infty}^x \rho(t) dt.$$

It has the properties:

1. $F(s) = \text{Prob}(x \leq s)$.
2. $\lim_{x \rightarrow -\infty} F(x) = 0$
3. $\lim_{x \rightarrow \infty} F(x) = 1$
4. $F(x)$ is an increasing function of x .
5. $\frac{dF}{dx} = \rho(x)$

It follows from 1 that

$$\text{Prob}(a \leq x \leq b) = F(b) - F(a).$$

The **mean** and **variance** of a density $\rho(x)$ are defined by

$$\mu = \text{Mean}(x) = \int_{-\infty}^{\infty} x \rho(x) dx$$

and

$$\sigma^2 = \text{Var}(x) = \int_{-\infty}^{\infty} (x - \mu)^2 \rho(x) dx$$

The **standard deviation**, σ , is the square root of the variance.

Uniform Density

The **uniform density** on the interval $[a, b]$ is defined by

$$\rho(x) = \begin{cases} 0 & x < a \\ \frac{1}{b-a} & a \leq x \leq b \\ 0 & x > b \end{cases}$$

An important special case is the uniform density on $[0, 1]$

$$\rho(x) = \begin{cases} 0 & x < 0 \\ 1 & 0 \leq x \leq 1 \\ 0 & x > 1 \end{cases}$$

This density has mean $\mu = 1/2$ and variance $\sigma^2 = 1/12$.

Normal Density

The **normal density** with mean μ and standard deviation σ is defined by

$$\rho(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The case $\mu = 0$, $\sigma = 1$ is especially important

$$\rho(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

Exponential Density

The **exponential density** with parameter λ is

$$\rho(x) = \lambda e^{-\lambda x}, \quad 0 \leq x < \infty, \quad \lambda > 0.$$

It has mean λ and variance λ^2 .

1.2 Discrete Distributions

Discrete probability distributions are defined on finite sets or more generally on any discrete set X , for example a subset the integers. A probability $p(x)$ is assigned to each $x \in X$ with the properties:

1. $p(x) \geq 0$.
2. $\sum_{x \in X} p(x) = 1$.

It follows from 2 that for any set $A \subset X$

$$\text{Prob}(x \in A) = \sum_{x \in A} p(x).$$

By analogy with continuous case, the mean and variance of a discrete distribution $p(x)$ are

$$\mu = \sum_{x \in X} xp(x)$$

and

$$\sigma^2 = \sum_{x \in X} (x - \mu)^2 p(x).$$

As a general principle, statements about continuous and discrete distributions are equivalent with integrals in the continuous case being equivalent to sums in the discrete case.

Binomial Distribution

The **binomial distribution** is the probability of obtaining k successes in n independent trials when the probability of success in each individual trial is p . It is given by

$$p(k) = \binom{n}{k} p^k (1-p)^{n-k}, \quad 0 \leq k \leq n,$$

where the **binomial coefficients** are

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

It has mean np and variance $np(1-p)$.

Poisson Distribution

The **Poisson distribution** is the limiting case of the binomial distribution in the limit $n \rightarrow \infty$ and $p \rightarrow 0$ but with the mean $np = \lambda$ remaining finite. This results in

$$p(k) = \frac{\lambda^k}{k!} e^{-\lambda}$$

with mean λ and variance λ .

2 Random Numbers

2.1 Uniform Random Numbers

Most random number generators produce uniformly distributed random numbers and numbers with other distributions being produced from the uniform numbers. Most commonly, random number generators produce integers in some range $[0, N]$. To get uniform floating point numbers the integer values produced by the generator are divided by $N + 1$ to give floating point numbers in the interval $[0, 1)$. A few generators, e.g. Fibonacci generators, produce floating point numbers directly.

Just about all random number generators generate a sequence of numbers based on some sort of **recurrence relation**

$$x_n = f(x_{n-1}, x_{n-2}, \dots, x_{n-k}). \quad (1)$$

As mentioned above, most commonly the values x_n are integers in some fixed range $[0, N]$.

A sequence generated by a recurrence relation (1) must eventually repeat itself. The length of the sequence generated before repetition is called the **period** of the generator. The set of initial values used to start the generator is called the **seed**.

2.2 Linear Congruential Generators

These are probably the simplest (and fastest) random number generators. Starting with the *seed* x_0 , a sequence of *integers* is generated by

$$x_n = (ax_{n-1} + c) \bmod M \quad (2)$$

where a , c and M are given integers. Sometimes a is called the *multiplier* and M the *modulus*. All the x_n are integers between 0 and $M - 1$. Division by M gives $[0, 1]$ floating point numbers.

Here is a Scilab implementation¹ of a general linear congruential generator. Here n is number of terms generated, a , c , and m are as in equation (2) and x_0 is the initial value or seed. A vector of $n+1$ values including the seed is returned. These values are integers in the range 0 to $m-1$. Division by m will give floating point numbers in the interval $[0, 1)$.

```
function x = lcg(n,a,c,m,x0)
  x = zeros(1, n+1)
  x(1) = x0
  for i = 2:n+1
    x(i) = pmodulo(a*x(i-1)+c, m)
  end
endfunction
```

The quality of a linear congruential generator depends on the choice of a , c and M , but in any case the period of such a generator is at most M . As we will see later, Scilab's basic `rand` generator is a linear congruential generator.

The following example illustrates a common problem with some random number generators. We will take the linear congruential generator with $a = 1203$, $c = 0$, $m = 2048$. With initial value $x_0 = 1$, this generator has period 512.

We will run through a full period of the generator:

```
-->ii = lcg(511, 1203, 0, 2048, 1);
```

Dividing by 2048 gives us $[0, 1]$ floating point numbers:

```
-->xx = ii/2048;
```

```
-->xx(1:10)
```

```
ans =
```

```
column 1 to 4
```

```
! 0.0004883 0.5874023 0.6450195 0.9584961 !
```

```
column 5 to 8
```

```
! 0.0708008 0.1733398 0.5278320 0.9819336 !
```

¹The version in `l1.sci` has been modified to avoid a problem with rounding error.

```
column 9 to 10
```

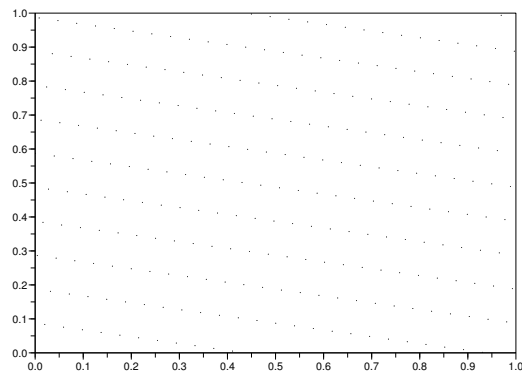
```
! 0.2661133 0.1342773 !
```

Now we take successive pairs of values as the x and y coordinates of a point in the plane and plot the results:

```
-->x = xx(1:2:511);
```

```
-->y = xx(2:2:512);
```

```
-->plot2d(x,y,style = -1)
```



This “latticing” is a common problem with random number generators. Of course, we have used a very simple generator for which the problem is obvious, for other generators the problem may occur in higher dimensions. That is, taking successive n -tuples of values and plotting them (hypothetically at least) in n -dimensional space the points may tend to lie on a lattice of lower dimensional subspaces.

2.3 Combining Random Number Generators

By combining a few simple random number generators we can obtain a generator with better properties. One generator used by Scilab combines four linear congruential generators

$$t_n = 45991t_{n-1} \bmod 2147483647$$

$$u_n = 207707u_{n-1} \bmod 2147483543$$

$$v_n = 138556v_{n-1} \bmod 2147483423$$

$$w_n = 49689w_{n-1} \bmod 2147483323$$

$$x_n = (t_n - u_n + v_n - w_n) \bmod 2147483647$$

Here is a Scilab implementation. n is the number of terms generated and $x0$ is the seed, in this case a vector of four values.

```

function x = clcg(n,x0)
    x = zeros(1,n)
    t = x0(1)
    u = x0(2)
    v = x0(3)
    w = x0(4)
    for i = 1:n
        t = pmodulo( 45991*t, 2147483647)
        u = pmodulo(207707*u, 2147483543)
        v = pmodulo(138556*v, 2147483423)
        w = pmodulo( 49689*w, 2147483323)
        x(i) = pmodulo(t - u + v - w, 2147483647)
    end
endfunction

```

Dividing by $2147483647 = 2^{31} - 1$ gives floating point numbers. This generator has a period of about 2^{123} .

2.4 Change of Variable

Some change of variable tricks are very handy.

Uniform Distribution

If x is a uniform $[0, 1]$ distribution, then the change of variable

$$y = (b - a)x + a$$

gives a uniform $[a, b]$ distribution.

Normal Distribution

If x has a normal $\mu = 0, \sigma = 1$ distribution, then the change of variable

$$y = ax + b$$

gives a normal distribution with $\mu = b$ and $\sigma = a$.

Uniform Discrete Distributions

It is quite common to want to generate uniformly distributed random integers. Typically they will be in the range $0 \leq k \leq n$ or $1 \leq k \leq n$. These can be obtained from uniform $[0, 1]$ random floating numbers x by

1. `floor((n+1)*x)` gives integers in the range $0 \leq k \leq n$.
2. `floor(n*x+1)` gives integers in the range $1 \leq k \leq n$.

2.5 Non-Uniform Random Numbers

Inverse Transform Method

Given a probability density $\rho(x)$ with distribution function $F(x)$, let $G(y)$ be the inverse function of $F(x)$. Then if we take y to be uniformly distributed on $[0, 1]$, $x = G(y)$ has probability density $\rho(x)$.

The exponential density has distribution function

$$F(x) = 1 - e^{-\lambda x}, \quad x \geq 0$$

and solving for x we obtain the inverse function

$$x = G(y) = -\frac{\ln(1-y)}{\lambda}.$$

This if y is uniform $[0, 1]$, then

$$x = -\frac{\ln(1-y)}{\lambda}$$

will have an exponential density.

One drawback of the inverse transform method is that we need to know the inverse of the distribution function. For most distributions there is no explicit formula for this inverse and we must resort to approximations or seek another method.

Acceptance/Rejection Method

This was covered in AMTH142.

3 Random Numbers in Scilab

3.1 rand

You should be familiar with `rand`:

```
rand(m,n)
```

generates a $m \times n$ matrix of uniform $[0, 1]$ random numbers.

```
rand(m,n,'normal')
```

generates a $m \times n$ matrix of normal, mean 0, standard deviation 1, random numbers. The current “seed” of `rand` is given by

```
rand('seed')
```

and

```
rand('seed',r)
```

sets the seed to `r`. This exhausts the capabilities of `rand`.

As mentioned earlier, `rand` is a linear congruential generator:

$$x_n = (ax_{n-1} + c) \bmod M$$

with $a = 843314861$, $c = 453816693$, and $M = 2^{31}$.

Let us check this. First we set the seed of `rand` to 0 (which is the seed on the first call to `rand`) and generate 10000 terms.

```
-->rand("seed",0)
```

```
-->x1 = rand(1,10000);
```

Now generate 10000 terms of lcg^2 with parameters above and seed 0.

```
-->y2 = lcg(10001, 843314861, 453816693, 2^31, 0);
```

We drop off the initial term and divide by $M = 2^{31}$ to get floating point numbers:

```
-->x2 = y2(2:10001)/(2^31);
```

We can look at first few terms:

```
-->x1(1:8)
```

```
ans =
```

```
column 1 to 4
```

```
! 0.2113249 0.7560439 0.0002211 0.3303271 !
```

```
column 5 to 8
```

```
! 0.6653811 0.6283918 0.8497452 0.6857310 !
```

```
-->x2(1:8)
```

```
ans =
```

```
column 1 to 4
```

```
! 0.2113249 0.7560439 0.0002211 0.3303271 !
```

```
column 5 to 8
```

```
! 0.6653811 0.6283918 0.8497452 0.6857310 !
```

Compare them

```
-->x1(1:8) - x2(1:8)
```

```
ans =
```

```
! 0. 0. 0. 0. 0. 0. 0. 0. !
```

and finally find how many terms differ:

```
-->sum(x1 ~= x2)
```

```
ans =
```

```
0.
```

²Use the version in `l1.sci`, see previous footnote.

3.2 grand

The random number generator **grand** extends the capabilities of **rand** in a number of ways:

1. Generates random numbers from various probability distributions.
2. Allows choice of random number generator.
3. Can produce independent streams of random numbers.

We will discuss each of these briefly, see **help grand** for more details.

Distributions

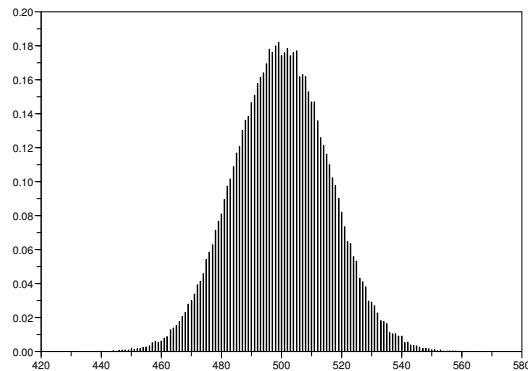
grand can generate all the probability distributions from §1 and a number of other distributions.

Uniform $[0, 1]$	<code>grand(m,n,'def')</code>
Uniform $[a, b]$	<code>grand(m,n,'unf',a,b)</code>
Discrete Uniform	<code>grand(m,n,'uin',n1,n2)</code>
Normal	<code>grand(m,n,'nor',m,s)</code>
Exponential	<code>grand(m,n,'exp',k)</code>
Binomial	<code>grand(m,n,'bin',n,p)</code>
Poisson	<code>grand(m,n,'poi',k)</code>
Raw Integer	<code>grand(m,n,'lgi')</code>

Here is an example with the binomial distribution:

```
-->bb = grand(1,100000,'bin',1000,0.5);
```

```
-->histplot(1000,bb)
```



It is no accident that this looks like a normal distribution, a topic we will return to later.

Random Permutations

```
grand(n,'prm',xs)
```

generates n random permutations of the *column* vector xs . Here is an example generating 3 random permutations of the vector $1:10$:

```
-->grand(3,'prm',(1:10)')
ans =

!  4.    2.    10. !
!  9.    3.    9.  !
!  5.    7.    5.  !
!  6.    8.    6.  !
!  7.    1.    3.  !
!  1.    4.    4.  !
!  2.   10.    1.  !
! 10.    9.    8.  !
!  8.    5.    7.  !
!  3.    6.    2.  !
```

Choice of Generator

`grand` allows the choice of a few different random numbers generators. Of interest to user are:

<code>urand</code>	Linear Congruential (used by <code>rand</code>)
<code>clcg4</code>	Combined Linear Congruential
<code>mt</code>	Mersenne Twister (default)

The default Mersenne twister generator is far too complex to describe here. For example it takes a vector of 625 integers as a seed. We will show that `clcg4` is the same as the combined generator we described earlier.

First we find the current generator and then set it to `clcg4`:

```
-->grand("getgen")
ans =

mt

-->grand("setgen", "clcg4")
ans =

clcg4
```

Now to get its seed which we need for our `clcg`:

```
-->sd = grand("getsd")
sd =

!  11111111. !
!  22222222. !
!  33333333. !
!  44444444. !
```

Now we can compare the raw integer output of the two generators

```
-->x1 = grand(1,10000,'lgi');
```

```
-->x2 = clcg(10000,sd);
```

```
-->sum(x1 ~= x2)
```

```
ans =
```

```
0.
```

Independent Streams

By changing the seed of any generator we can obtain different sequences of random numbers. However there is no guarantee that the sequences won't overlap. For some applications we might want non-overlapping sequences, for instance in (a) independent repetitions of a Monte-Carlo simulation or (b) parallel computations.

Scilab offers independent non-overlapping sequences with `clcg4`. Once the generator is set `clcg4` you can choose between 101 virtual random number generators, numbered 0 to 100. Here is an example of how it is done:

```
-->grand("setgen", "clcg4")
```

```
ans =
```

```
clcg4
```

```
-->grand("getcgn")
```

```
ans =
```

```
0.
```

```
-->x0 = grand(1,1000,"def");
```

```
-->grand("setcgn",23)
```

```
ans =
```

```
23.
```

```
-->x23 = grand(1,1000,"def");
```