AMTH142 Lecture 13

Random Numbers

April 20, 2007

All the Scilab functions defined in this Lecture can be found in the file `l13.sci` in the directory for this lecture.

Contents

13.1 Review of Probability

This is just a reminder of few facts about continuous probability distributions covered in MATH102.

13.1.1 Definitions

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

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

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. $\displaystyle\lim_{x\to-\infty} F(x) = 0$
3. $\displaystyle\lim_{x\to\infty} F(x) = 1$
4. $F(x)$ is an increasing function of x.
5. $\displaystyle\frac{dF}{dx} = \rho(x)$

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.

13.1.2 Uniform Density

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

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

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

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

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

13.1.3 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

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

13.2 Random Number Generators

Computer algorithms for generating random numbers are *deterministic* algorithms. Although the sequence of numbers produced by a random number generator appears random, the sequence of numbers is completely predictable and for this reason they are often called **pseudo-random**.

Since computers have only a finite number of different states, the sequence of pseudo-random numbers produced by any deterministic algorithm must necessarily repeat itself. The number of random numbers generated before the sequence repeats is called the *period* of the generator.

A good random number generator should have the following characteristics:

1. *Randomness.* It should pass statistical tests of randomness.

2. *Long Period.* For obvious reasons.

3. *Efficiency.* This is important since simulations often require millions of random numbers.

4. *Repeatability.* It should produce the same sequence of numbers if started in the same state. This allows the repetition and checking of simulations.

Almost all random number generators used in practice produce uniform $[0, 1]$ distributed random numbers and from these random numbers with other distributions can be produced if required.

13.2.1 Algorithms

There are many algorithms for generating pseudo-random numbers.

Linear Congruential Generators

The simplest are the **linear congruential** generators. Starting with the *seed* x_0, these generate a sequence of *integers* by

$$x_k = (ax_{k-1} + c) \bmod M \tag{1}$$

where a, c and M are given integers. All the x_k are integers between 0 and $M - 1$. In order to produce floating point numbers these are divided by M to give a floating point number in the interval $[0, 1)$.

Below is a Scilab function[1] implementing a linear congruential generator. Here `n` is the number of terms generated `a`, `c` and `m` are as in equation (1) and `x0` is the initial value or seed. A vector of `n+1` values including the seed is returned.

```
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
  x = x/m
endfunction
```

The quality of a linear congruential generator depends on the choice of a, b and M, but in any case the period of such a generator is at most M (why?). 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.

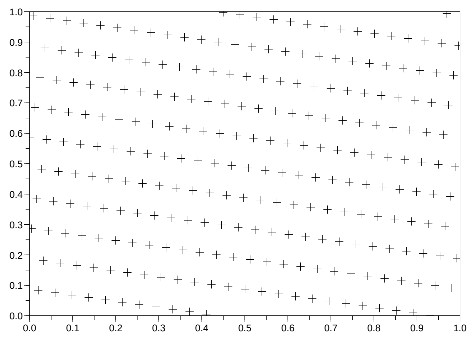
We will run through a full period of the generator:

[1]The version in `l5.sci` has been modified to avoid a problem with rounding error.

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

-->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 !

        column  9 to 10

!   0.2661133    0.1342773 !
```

Now 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.

Fibonacci Generators

Another common type of generator are the **Fibonacci generators**. A typical example generates a sequence by

$$x_k = (x_{k-17} - x_{k-5}) \bmod 1$$

This example uses the previous 17 numbers (and needs 17 seeds to get started). It has a number of advantages compared to congruential generators:

1. The x_k are floating point numbers, so no division is needed.
2. It has a much longer period since the repetition of a single number does not entail repetition of the whole sequence.
3. It can have much better statistical properties.

Combined 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

$$\begin{aligned} t_n =& 45991 t_{n-1} \bmod 2147483647 \\ u_n =& 207707 u_{n-1} \bmod 2147483543 \\ v_n =& 138556 v_{n-1} \bmod 2147483423 \\ w_n =& 49689 w_{n-1} \bmod 2147483323 \\ x_n =& (t_n - u_n + v_n - w_n) \bmod 2147483647 \end{aligned}$$

This generator has a period of about 2^{123}.

Other Random Number Generators

Modern random number generators are based on number theory and have quite sophisticated implementations. One in common use is the **Mersenne twister** which takes a vector of 625 as its seed.

13.2.2 Scilab

Scilab has two random number generators `rand` and `grand`. We will only discuss the simpler generator `rand` which can produce either uniform $[0, 1]$ or normal $\mu = 0, \sigma = 1$ random numbers. Uniform random numbers are the default.

We have already seen how `rand` works:

1. `rand(m, n)` – gives a $m \times n$ matrix of random numbers.

2. `rand(m, n, 'normal')` – gives a $m \times n$ matrix of normally distributed random numbers.

3. `rand(a)` or `rand(a, 'normal')` – for matrix `a` gives a matrix of random numbers the same size as `a`.

4. `rand('seed', 0)` – resets the random number generator to its original state. This is handy if you want to repeat an experiment using the same random numbers.

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`.

```
-->x2 = lcg(10001, 843314861, 453816693, 2^31, 0);
```

We drop off the initial term (which is the seed) and look at first few terms:

```
-->x2 = x2(2:10001);

-->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  =
```

[2]Use the version in `l5.sci`, see previous footnote.

```
         column 1 to 4

!   0.2113249    0.7560439    0.0002211    0.3303271 !

         column 5 to 8

!   0.6653811    0.6283918    0.8497452    0.6857310 !
```

and finally find how many terms differ:

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

    0.
```

13.3 Some Common Distributions

Two simple change of variable tricks are very important.

13.3.1 Uniform Distribution

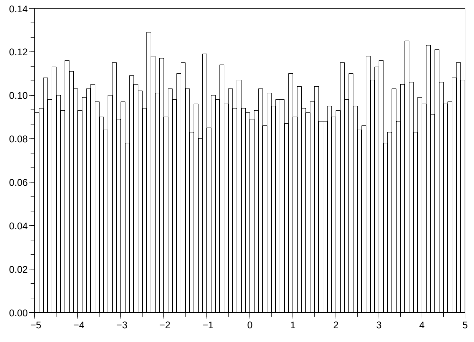
If x is a uniform $[0,1]$ distribution, then the change of variable

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

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

Here is an example generating a uniform $[-5,5]$ distribution:

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

-->y = 10*x - 5;

-->histplot(100,y)
```



13.3.2 Normal Distribution

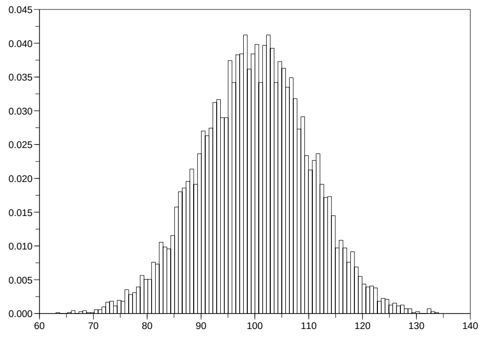
If x has a normal $\mu = 0$, $\sigma = 1$ distribution, then the change of variable

$$y = ax + b$$

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

Here is an example generating a normal $\mu = 100$, $\sigma = 10$ distribution:

```
-->x = rand(1,10000,'normal');

-->y = 10*x + 100;

-->histplot(100,y)
```



13.3.3 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 $1 \leq k \leq n$.

Here is an example generating integers in the range 1 to 10 (inclusive):

```
-->x = rand(1,30);

-->y = floor(10*x+1)
 y  =


         column  1 to 10
```

```
!   9.    7.    9.    3.    4.    9.    9.    1.    2.    4. !

         column 11 to 20

!   9.    2.    4.    2.    7.    6.    3.    6.    10.    8. !

         column 21 to 30

!   4.    7.    5.    4.    8.    2.    7.    1.    10.    1. !
```

13.4 Non-Uniform Random Numbers

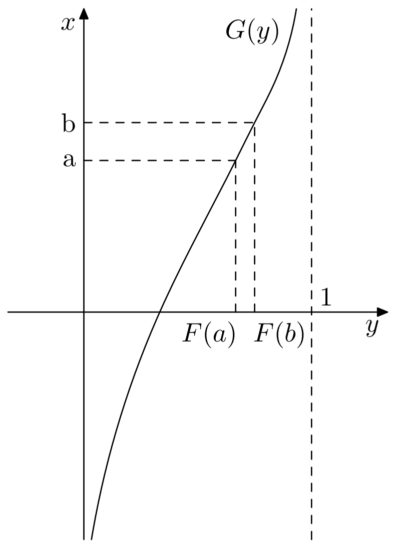
13.4.1 Inverse Transform Method

Consider a probability density $\rho(x)$ and let $F(x)$ be the corresponding distribution function. Then

$$F : \mathbb{R} \to [0,1]$$

and since $F(x)$ is an increasing function it has an inverse function $G(y)$ with

$$G : [0,1] \to \mathbb{R}$$



Let y be uniformly distributed on $[0,1]$ and set

$$x = G(y).$$

Then

$$\text{Prob}(a \le x \le b) = \text{Prob}(F(a) \le y \le F(b)) = F(b) - F(a)$$

Where the last equality follows from the uniformity of y. The statement

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

is equivalent to the statement that x has distribution function $F(x)$ and hence x has density $\rho(x)$. In summary:

> *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 $\rho(x)$.*

Exponential Distribution

This is defined by

$$\rho(x) = \begin{cases} 0 & x < 0 \\ \lambda e^{-\lambda x} & x \ge 0 \end{cases}$$

with distribution function

$$F(x) = \begin{cases} 0 & x < 0 \\ 1 - e^{-\lambda x} & x \ge 0 \end{cases}$$

(Here λ is a parameter).

Setting

$$y = F(x) = 1 - e^{-\lambda x}$$

and solving for x we obtain the inverse function

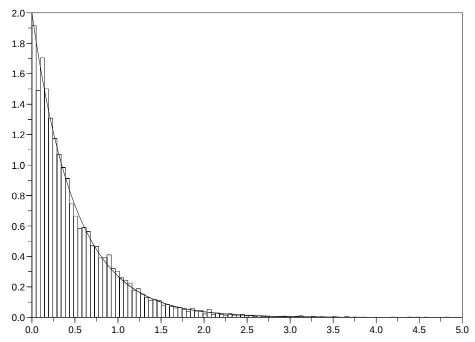
$$x = G(y) = -\frac{\ln(1-y)}{\lambda}.$$

Lets see how it works in Scilab; we will take $\lambda = 2$.

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

-->x = - log(1-y)/2;

-->histplot(100,x)
```

We can compare the histogram to exact density:

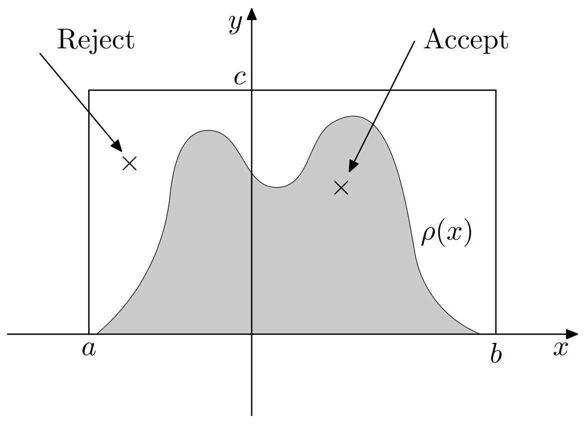
```
-->xx = 0:0.01:5;

-->plot2d(xx, 2*exp(-2*xx))
```



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.

13.4.2 Acceptance/Rejection Method

This method too has its drawbacks. We will need to assume that we are working with a density which is zero outside of a finite interval. Suppose that $\rho(x)$ is zero outside of the interval $[a, b]$ and furthermore that $\rho(x)$ is bounded above by c.

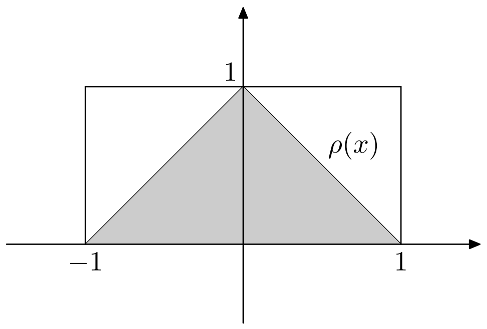


Now generate points (x_i, y_i) with x_i uniformly distributed in $[a, b]$ and y_i uniformly distributed in $[0, c]$. If $y_i \leq \rho(x_i)$ then we accept the value x_i, if $y_i > \rho(x_i)$ then we reject the value x_i. The values x_i which are accepted will have the probability density $\rho(x)$.

An Example

Consider the hat-shaped probability density defined on $[-1,1]$ by

$$\rho(x) = \begin{cases} x+1 & -1 \le x \le 0 \\ 1-x & 0 \le x \le 1 \end{cases}$$



We want to write a Scilab function to generate n random numbers with this density using the acceptance/rejection method. One thing to keep in mind when using the acceptance/rejection is that we do not know before hand how many random numbers we need to generate.

First we need the density function [3]:

```
function y = rhohat(x)
  if (x < 0) then
    y = x + 1
  else
    y = 1 - x
  end
endfunction
```

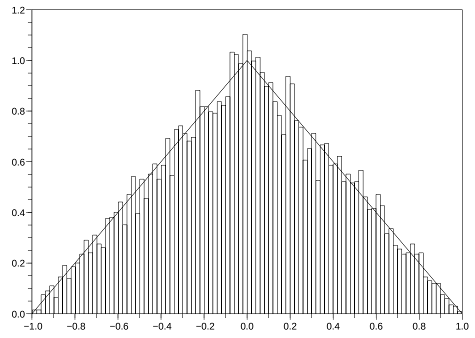
Now the random number generator:

```
function x = randhat(n)
  x = zeros(1,n)
  k = 0               // keep count of numbers generated
  while (k < n)
    xx = -1 + 2*rand(1,1)      // uniform on [-1,1]
    yy = rand(1,1)
    if (yy <= rhohat(xx))      // accept xx
      k = k + 1
      x(k) = xx
    end
  end
endfunction
```

[3]The version of `rhohat` in `l5.sci` has been modified to allow a vector `x` as the argument.

And testing it

```
-->x = randhat(10000);

-->histplot(100, x)

-->xx = -1:0.01:1;

-->plot2d(xx, rhohat(xx))
```



13.4.3 Special Distributions

The two methods discussed above, acceptance/rejection and inverse transform, when applied with some ingenuity can be used to generate random numbers with almost any distribution. However for most distributions there are specific techniques that perform better than these more general techniques. The Scilab function `grand` has facilities for generating random numbers from most of common distributions.

Normal Distribution

Here is a clever method for generate normally distributed random numbers: generate two uniform $[0, 1]$ random numbers x_1 and x_2, then

$$y_1 = \sin(2\pi x_1)\sqrt{-2\ln x_2}$$

and

$$y_2 = \cos(2\pi x_1)\sqrt{-2\ln x_2}$$

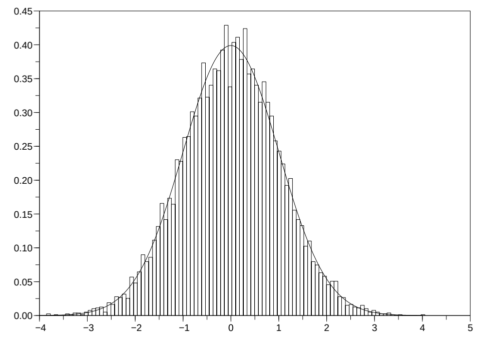
are both normally distributed with $\mu = 0$, $\sigma = 1$.

Here is how it works in Scilab:

```
function x = randnorm(n)  // assumes n even
  m = n/2
  x1 = rand(1,m)
  x2 = rand(1,m)
  y1 = sin(2*%pi*x1).*sqrt(-2*log(x2));
  y2 = cos(2*%pi*x1).*sqrt(-2*log(x2));
  x = [y1 y2]
endfunction

-->x = randnorm(10000);

-->histplot(100,x)
```

We can compare the histogram to exact density:

```
-->xx = -4:0.01:4;

-->yy = exp(-(xx.^2)/2)/sqrt(2*%pi);

-->plot2d(xx, yy)
```


0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−4
−3
−2
−1
0
1
2
3
4
5