Lecture 23

Random numbers

1 Introduction

1.1 Motivation

Scientifically an event is considered *random* if it is unpredictable. Classic examples are a coin flip, a die roll and a lottery ticket drawing. For a coin flip we can associate 1 with "heads" and 0 with "tails," and a sequence of coin flips then produces a random sequence of binary digits. These can be taken to describe a random integer. For example, 1001011 can be interpreted as the binary number corresponding to

$$1(2^6)+0(2^5)+0(2^4)+1(2^3)+0(2^2)+1(2^1)+1(2^0)=64+8+2+1=75 \quad (1)$$

For a lottery drawing we can label n tickets with the numbers $0,1,2,\ldots,n-1$. A drawing then produces a random integer i with $0\le i\le n-1$. In these and other ways random events can be associated with corresponding random numbers.

Conversely, if we are able to generate random numbers we can use those to represent random events or phenomena, and this can very useful in engineering analysis and design. Consider the design of a bridge structure. We have no direct control over what or how many vehicles drive across the bridge or what environmental conditions it is subject to. A good way to uncover unforeseen problems with a design is to simulate it being subject to a large number of random conditions, and this is one of the many motivations for developing random number generators. How can we be sure our design will function properly? By the way, if you doubt this is a real problem, read up on the Tacoma Narrows bridge which failed spectacularly due to wind forces alone [1].

1.2 "Truly random" numbers

In classical physics, nothing is fundamentally "random." In Principle the present state of the universe combined with physical laws *exactly determines* the future state of the universe. Classical physics is strictly deterministic. In principle if we knew the positions and velocities of a pair of dice thrown in the air, their physical properties and the properties of the surface on which they will land, we could predict what numbers they will show by solving the equations of motion. However, in practice the behavior of the dice-plus-table system is so complicated that even extremely small changes in the initial conditions or the properties of the dice will cause a significant change in their final state. There are many *chaotic systems* that display this so-called *butterfly effect* [Error: Reference source not found]. For practical purposes these are random events, even in a deterministic universe.

According to quantum mechanics there are truly random phenomena at the atomic scale. For example, it is not possible, even in principle, to predict when a radioactive nucleus will decay. We can only make statements about the probability of its doing so during a certain time interval. For this reason quantum effects are arguably the ultimate way to generate "truly random" numbers, and the fields of quantum random number generation and quantum cryptography are areas of active research and development.

In the past sequences of random numbers, generated by some random physical process, were published as books. In fact the book *A Million Random Digits with 100,000 Normal Deviates*, produced by the Rand Corporation in 1955, is still in print (628 pages of "can't put it down" reading!) as of 2014. It is still the case that if a sequence of truly random numbers is desired they must be obtained from some physical random process. One can buy plug-in cards that sample voltages generated by thermal noise in an electronic circuit (such as described at onerng.info), and the website random.org generates random numbers by sampling radio-frequency noise in the atmosphere.

1.3 Pseudo-random numbers

A properly functioning computer is a strictly deterministic system. Given the same data and instructions it will produce the same output every time. Therefore *it is impossible for a computer to generate truly random numbers*. Instead we have to be content with the generation of *pseudo-random numbers*. A computer program that generates a sequence of such numbers is called a *pseudo-random number generator* (PRNG).

A sequence of numbers is pseudo-random if it "looks like" a random sequence. More rigorously there are various statistical tests available to quantify how "random looking" the output of a PRNG is. Until recently the so-called "diehard tests" [2] were a common software tool for this purpose. More recently the National Institute of Standards and Technology has released a random-number toolkit [3].

Consider the following sequence of numbers

```
0 1 2 3 4 5 6 7
```

There appears to be nothing random about this sequence; it follows an obvious pattern of incrementing by one. Of course it's possible that a random sequence of numbers just happened to form this pattern by chance, but intuitively that's not very likely. On the other hand the sequence

```
1 6 7 4 5 2 3 0
```

which contains the same eight digits does look somewhat random, although it doesn't take long to notice an add-one and subtract-three pattern in the last seven digits. In fact it was generated by a algorithm which is every bit as deterministic as "increment by one," and to which we now turn.

2 Linear congruential generators

The expression

$$y = x \pmod{m} \tag{2}$$

read "y equals x modulo m," means that y is the remainder when x is divided by m. For example

$$3 = 11 \pmod{4} \tag{3}$$

because

$$\frac{11}{4} = \frac{8+3}{4} = 2 \text{ remainder } 3 \tag{4}$$

In Scilab/Matlab we can calculate this using the command

```
y = x-int(x/m)*m
```

which subtracts off an integer number times *n* from *x* leaving the remainder. More directly

```
-->modulo(11,4) //Scilab
 ans  =
    3.

>> mod(11,4) %Matlab
ans =
     3
```

Another way to think of this is that $x \pmod m$ is the least-significant digit of *x* expressed in base-*m*. For example $127 \pmod{10} = 7$, and $13 \pmod 8 = 5$ because $13 = 1 \cdot 8^1 + 5 \cdot 8^0 = 15_8$.

If $x \pmod m = y \pmod m$ we say "*x* is congruent to *y* modulo *m*" and we write

$$x \equiv y \pmod m \tag{5}$$

A *linear congruential generator* (LCG) is a simple method for generating a permutation of the integers $0 \le i \le m-1$ using modular arithmetic. Starting with a *seed* value x_0, with $0 \le x_0 < m$, a LCG generates a sequence $x_1, x_2, \ldots, x_m$ with

$$x_{n+1} = (a\, x_n + c) \pmod m \tag{6}$$

Provided the constants *a* and *c* are properly chosen this will be a permutation of the integers $0 \le i \le m-1$. If *m* is a power of 2, then *c* must be odd and *a* must be one more than a multiple of 4. For example, if $m = 2^3 = 8$, $a = 5, c = 1$ and $x_0 = 0$, then $x_1, x_2, \ldots, x_8$ is the permutation

```
1 6 7 4 5 2 3 0
```

because

$$(0+1) \bmod 8 = 1,\ (5 \cdot 1 + 1) \bmod 8 = 6,\ (5 \cdot 6 + 1) \bmod 8 = 31 \bmod 8 = 7,$$
$$(5 \cdot 7 + 1) \bmod 8 = 36 \bmod 8 = 4,\ (5 \cdot 4 + 1) \bmod 8 = 21 \bmod 8 = 5,$$
$$(5 \cdot 5 + 1) \bmod 8 = 26 \bmod 8 = 2,\ (5 \cdot 2 + 1) \bmod 8 = 11 \bmod 8 = 3$$
$$\text{and } (5 \cdot 3 + 1) \bmod 8 = 16 \bmod 8 = 0$$

Since $x_8 = x_0 = 0$ the permutation will then repeat with $x_9 = x_1 = 1$ and so on. If we used a seed value of $x_0 = 4$ then we would get the sequence

```
5 2 3 0 1 6 7 4
```

which is the same sequence starting from a different digit. A LCG with given *a* and *c* values will always produce the same sequence, and using a different seed value will simply start us off at a different location in the sequence.

A repeating sequence of eight numbers is probably not of much use, but in practice we use a large value of *m*, quite commonly $m = 2^{32}$ corresponding to a "32-bit unsigned integer" which are conveniently implemented in digital hardware. Fig. 1 shows plots of sequences generated with $m = 2^{10} = 1024$ and parameters $a = 5, c = 1$ (left) and $a = 9, c = 1$ (right). Ten samples from the latter sequence

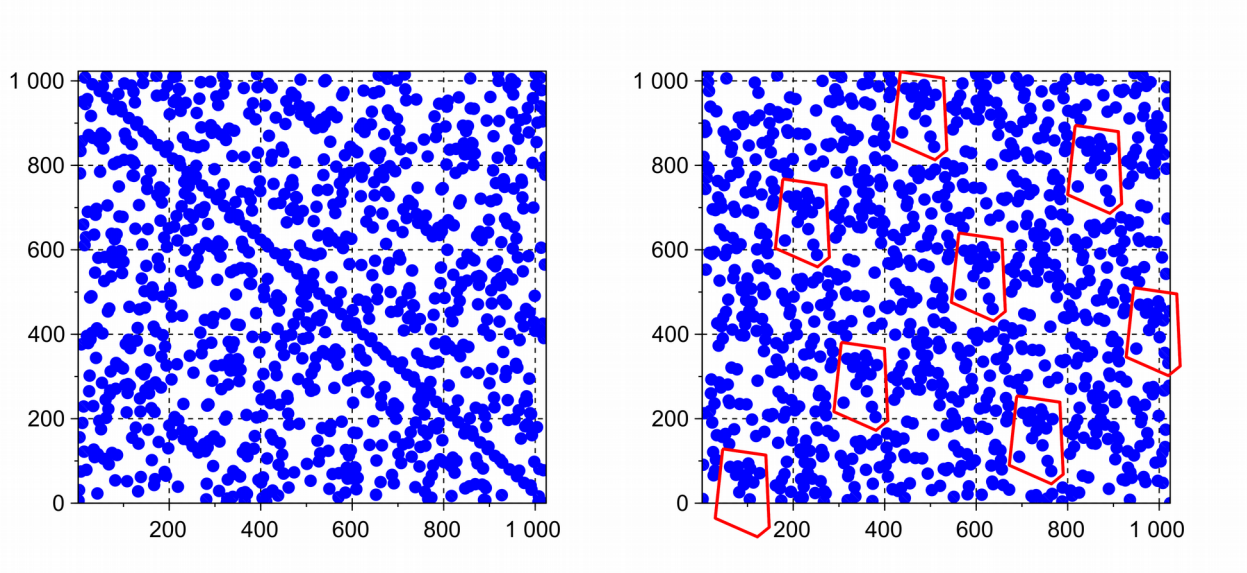

Fig. 1: Output of linear congruential generator (vertical vs. n horizontal) with m=1024, starting with seed value $x_0 = 0$ *. (left) a=5, c=1; (right) a=9,c=1. Repeated patterns are a strong indication that these are not truly random sequences.*

```
532 693 94 847 456 9 82 739 508 477
```

don't display an immediately obvious sequential relationship. In Fig. 1 we plot both sequences in full. In the left graph some of the samples form a very noticeable diagonal line pattern which is a strong indication that the data are almost certainly not truly random. The right plot lacks such a glaring "red flag." However, on closer inspection one can see repeated patterns throughout the image (indicated by polygons). It is extremely unlikely that a truly random sequence of numbers would exhibit such a pattern, and a LCG is unacceptable for a demanding PRNG application such as cryptography.

In engineering applications we usually want random real numbers x instead of random integers i. We can easily generate real numbers $0 \le x < 1$ from integers $0 \le i \le m-1$ by calculating

$$x = \frac{i}{m}$$

Of course there will only be a discrete set of m such real numbers with spacing $1/m$, but if m is large enough this can provide a good approximation to a "real random number generator." In many programming languages (including Scilab) the command

```
x = rand();
```

generates a "random" real number x with $0 \le x < 1$ using this method.

LCGs are simple to implement and for many years were the "standard" PRNGs in many computer programming languages. For simple engineering design and analysis tasks they are often "good enough." A Scilab LCG implementation appears in the appendix. Note that this makes explicit use of 32-bit unsigned integers. Arithmetic performed with this data type is inherently modulo 2^{32} .

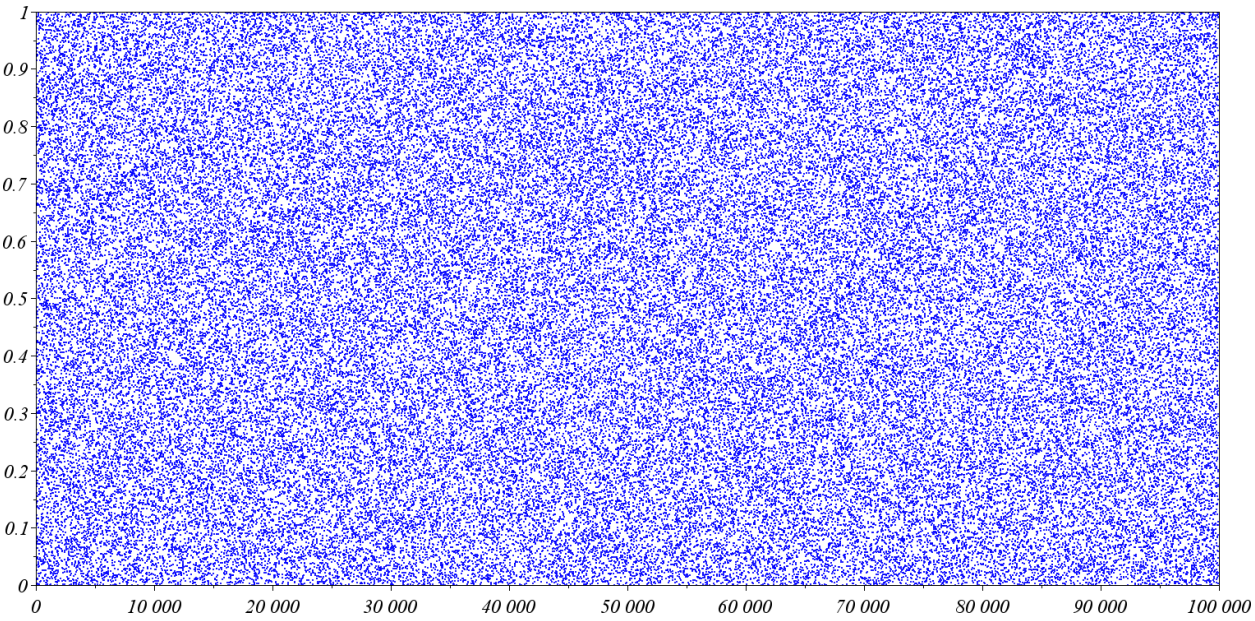

Fig. 2: 100,000 samples of the multiply-with-carry PRNG.

3 More advanced pseudo-random number generators

Various methods have been developed to try and improve on the LCG method. The multiply-with-carry method, roughly speaking, generates two LCG sequences combined with some "bit masking" and "bit shifting" and combines these to produce a pseudo-random output. A Scilab implementation is given in the Appendix and a plot of 100,000 samples is shown in Fig. 2. Currently one of the most commonly used improved algorithm is called the *Mersenne twister* [5], and both Scilab and Matlab have implementations of this. This algorithm is fairly complex. Instead of one or two seed integers, it's initial state consists of an array of 625 integers. Cryptography is arguably the fields with the most demanding PRNG requirements. Not surprisingly, some of the best PRNG algorithms make use of encryption techniques, such as the Advanced Encryption Standard.

4 The `rand` function (Scilab/Matlab)

The function call

```
x = rand();
```

in both Scilab and Matlab generates a random real number $0 \le x < 1$. Subsequent calls return the next number in the pseudo-random sequence. To generate an array with dimensions $m \times n$ use

```
x = rand(m,n);
```

On start up both Scilab and Matlab initialize the PRNG to a specific state. Therefore you will always get the same sequence of numbers. In my current version of Scilab, on start up I will always obtain the initial result

```
-->rand(3,1)
 ans  =

    0.2113249
```

```
    0.7560439
    0.0002211
```

while in my current version of Matlab, on start up I will always obtain the initial result

```
>> rand(3,1)
ans =

    0.8147
    0.9058
    0.1270
```

To start off at a different place in the sequence you can "seed" the PRNG as follows

```
-->rand('seed',i0); //Scilab
>> rand('twister',i0); %Matlab
```

where i_0 is an integer in the range $0 \le i_0 < 2^{32}$. It can actually be useful to generate the same "random" number on separate occasions because it allows interesting simulation results to be repeated. However, sometimes you want pseudo-random numbers that are different each time your open and run a program. One way to get a unique seed each time you run a program is to generate it from the system clock. Recommended ways to do this are

```
-->rand('seed',getdate('s')); //Scilab
>> rand('twister',sum(100*clock)); %Matlab
```

4.1 The grand function (Scilab)

The Scilab `rand()` function is based on a LCG while the Matlab version uses the superior Mersenne twister algorithm. In Scilab the twister algorithm is available in the `grand()` function. This is used as follows

```
-->grand(3,1,'def')
 ans  =

    0.8147237
    0.135477
    0.9057919
```

to produce a 3-by-1 array. The '`def`' string indicates that you want the returned value to be from the default distribution which is uniform over $0 \le x < 1$. To seed this PRNG use the command

```
-->grand('setsd',i0);
```

As with the `rand()` function you can use the system clock as an ever-changing seed

```
-->grand('setsd',getdate('s'));
```

5 The probability density function and histograms

We say that the *probability density function* (pdf) of a random variable x is $f_x(x)$ if the probability that x will take on a value $x_1 \le x \le x_2$ is

$$\text{Prob}\{x_1 \le x \le x_2\} = \int_{x_1}^{x_2} f_x(x)\,dx$$

So far we have used the `rand()` and `grand()` functions to produce x values uniformly distributed over $0 \le x < 1$. The ideal uniform pdf is

$$f_x(x) = \begin{cases} 1 & 0 \le x < 1 \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

Let's test this by generating a large number of random values and then plotting a *histogram* of the data. To generate a histogram we first divide the x interval of interest into a number of subintervals or *bins*. Let's take our bins to be $0 \le x < 0.05$, $0.05 \le x < 0.10$, $0.10 \le x < 0.15$ and so on up to $0.95 \le x < 1.00$. We then count how many x samples fall in each bin. This number divided by the total number of samples is an estimate of $\int f_x(x)\,dx$ over the bin. Dividing by the width of the bin (0.05 in our case) we get an estimate of the average value of $f_x(x)$ for that bin.

Generating a histogram plot in Scilab is as simple as

```
histplot(nbins,x);
```

where `nbins` is the number of equal-sized bins we want in the interval $[x_{min}, x_{max}]$. As the number of samples increases a histogram should give a progressively better estimate of the underlying pdf of the random (or pseudo-random) process. Running the commands

```
x = grand(1e3,1,'def');
histplot(20,x);
```

and

```
x = grand(1e6,1,'def');
histplot(20,x);
```

produced the results shown in Fig. 3. We see that the pdf does approach the ideal uniform

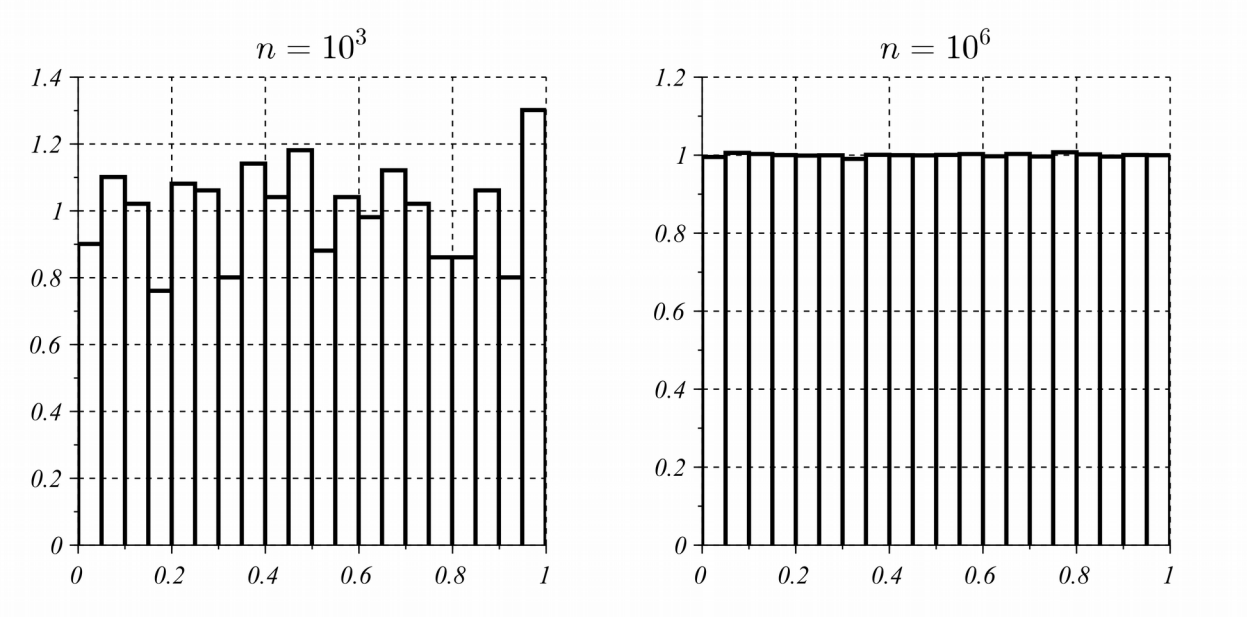

Fig. 3: histograms of `x = grand(1e3,1,'def')` *and* `x = grand(1e6,1,'def');`

distribution as the number of samples gets large enough.

6 Need for random variables with non-uniform pdf

The uniform distribution (7) is only one possible pdf. We often need to generate random numbers with some specific pdf in order to simulate a certain physical process. A few examples are:

6.1 Normal distribution

This is the classic "bell curve" (also called the *Gaussian distribution*)

$$f_y(y)=\frac{1}{2\pi\sigma}e^{-\frac{1}{2}\left(\frac{y-\mu}{\sigma}\right)^2},\ -\infty<y<\infty$$

The normal distribution is specified by two parameters: μ is the *mean value* (average) of y and σ is the *standard deviation*. The *Central Limit Theorem* tells us that any process that is the sum or average of a large number of independent, identically distributed processes will be normally distributed. Since so many natural phenomena have this property, the normal distribution finds wide application. In particular "noise" in measurements is often assumed to be normally distributed.

6.2 Exponential distribution

The power received by a cell phone (or other wireless device) in very "cluttered" environment where the field is scattered many times is described by the exponential distribution

$$f_P(P)=\frac{1}{P_{av}}e^{-P/P_{av}},\ P\geq 0 \tag{8}$$

Here P_{av} is the average received power. To simulate a wireless communication channel we need to be able to generate exponentially distributed random values to model "fading" effects.

6.3 Maxwell-Boltzman distribution

In a gas in thermal equilibrium at temperature T, the probability that a molecule will have velocity v is given by the Maxwell-Boltzmann distribution

$$f_v(v)=\frac{4}{\sqrt{\pi}\,v_p}\left(\frac{v}{v_p}\right)^2 e^{-\left(\frac{v}{v_p}\right)^2},\ v\geq 0 \tag{9}$$

where

$$v_p=\sqrt{\frac{2kT}{m}} \tag{10}$$

is the most likely velocity (the peak of the pdf). Here m is the molecular mass and k is Boltzmann's constant. We need to generate samples from this distribution if we wish to perform molecular dynamics simulations.

7 Generating random variables with an arbitrary pdf

Suppose x is a uniformly distributed random variable of the type we discussed above with pdf

given by (7). Now define a random variable *y* as

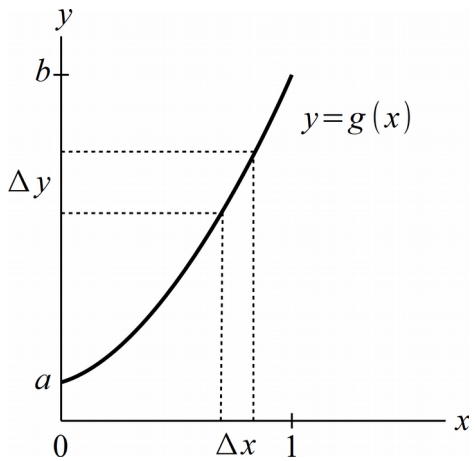

Fig. 4: $f_x(x)\Delta x \approx f_y(y)\Delta y$

$$y = g(x) \tag{11}$$

with

$$a = g(0)\ ,\ b = g(1) \tag{12}$$

This is illustrated in Fig. 4. We want to find the pdf of *y* over the interval $a \le y \le b$. An *x* interval of width Δx will correspond to a *y* interval of width Δy , and *y* will fall in the Δy interval if and only if *x* falls in the Δx interval. Therefore we can equate the probabilities

$$f_x(x)\Delta x \approx f_y(y)\Delta y \tag{13}$$

Since $f_x(x) = 1$ we have the differential relation

$$dx = f_y(y)\,dy \tag{14}$$

Integrating this from 0 to *x* on the left and correspondingly from *a* to *y* on the right, we have

$$x = \int_0^x dx = \int_a^y f_y(y)\,dy = F_y(y)$$

where $F_y(x)$ is called the *cumulative distribution function* (cdf) of the random variable *y*. Inverting this relation

$$x = F_y(y) \tag{15}$$

we get $y = g(x)$.

For example, the exponential distribution (8) has cdf

$$\int_0^y \frac{1}{P_{av}} e^{-P/P_{av}}\,dP = 1 - e^{-y/P_{av}} \tag{16}$$

Solving $x = 1 - e^{-y/P_{av}}$ gives us

$$y = -P_{av}\ln(1-x) \tag{17}$$

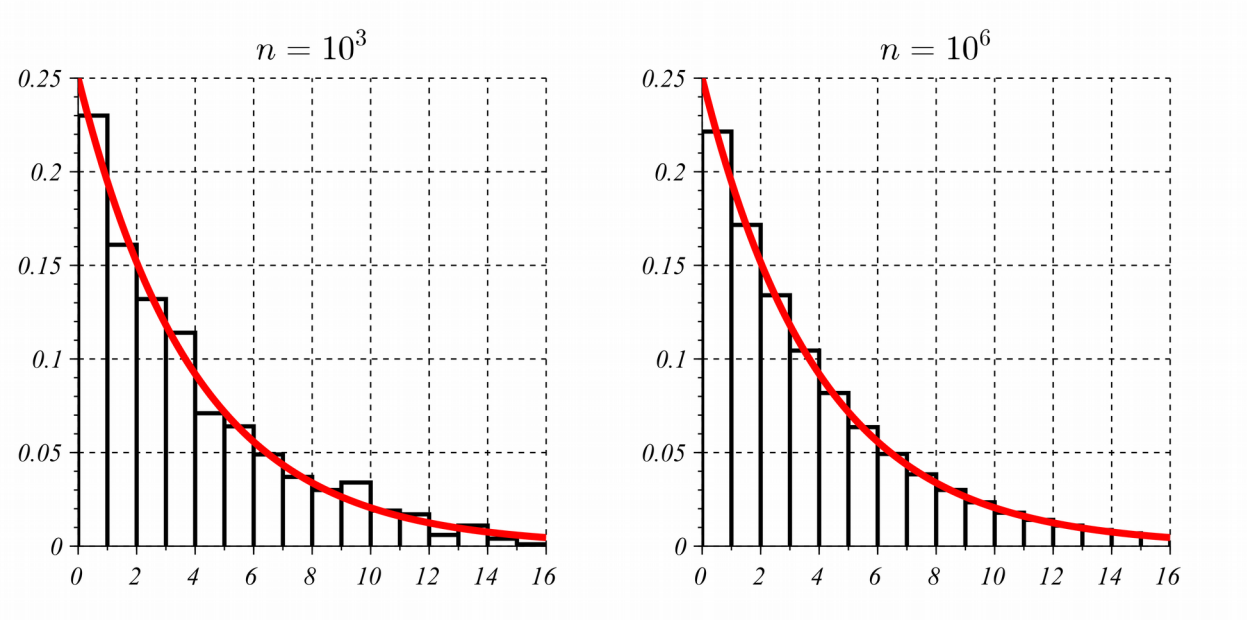

Fig. 5: Histograms of exponentially distributed random values generated by $y=-4\ln(1-x)$ *and ideal pdf.*

In Fig. 5 we show histograms of $y=-4\ln(1-x)$ where x is a uniform random variable. Given enough sample points the histogram approximates the ideal pdf very well.

Unfortunately, for some pdfs it is not possible to calculate the cdf. Arguably the most important example is the normal distribution. The integral

$$F_y(y)=\frac{1}{2\pi\sigma}\int_{-\infty}^{y} e^{-\frac{1}{2}\left(\frac{y-\mu}{\sigma}\right)^2} dy \tag{18}$$

cannot be evaluated in terms of elementary functions. However, it is possible to transform two independent, uniformly distributed random variables x_1, x_2 into two independent normally distributed random variables y_1, y_2 using the *Box-Muller transform*. We first calculate the polar coordinates

$$r=\sqrt{-2\ln x_1}, \theta=2\pi x_2$$

and then y_1, y_2 are the rectangular coordinates

$$y_1=r\cos\theta, \; y_2=r\sin\theta$$

For the values x_1, x_2 we can use two sequential values from a PRNG. Fig. 6 shows histograms obtained by applying the Box-Muller transform to 10^3 (left) and 10^6 (right) values of the `rand()` function and compares these with the ideal normal distribution. With enough samples the agreement is excellent.

The Box-Muller transform produces y values with $\mu=0, \sigma=1$. To change these we perform an additional transform

$$z=\sigma y+\mu \tag{19}$$

Then z has mean μ and standard deviation σ.

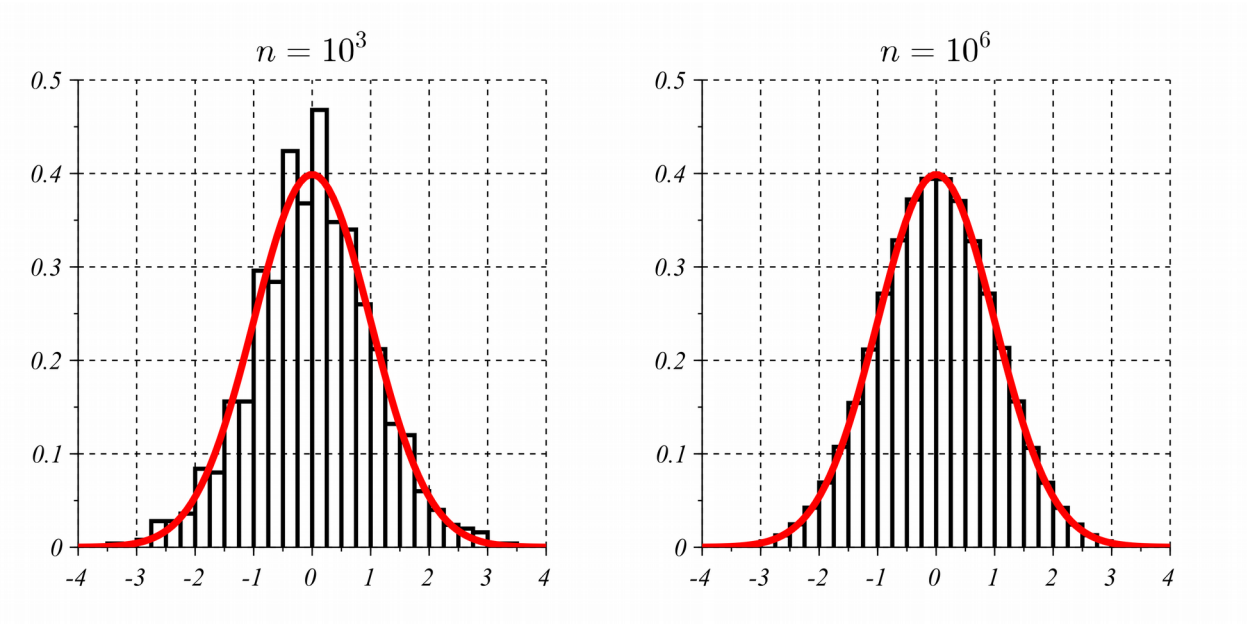

Fig. 6: Histograms of zero-mean, unit-variance, normally distributed random values generated by the Box-Muller transform and ideal pdf.

In Matlab the `randn()` function is similar to the `rand()` function but generates normal random values with $\mu=0, \sigma=1$. In Scilab the grand() function can generate arrays of normal random variables with specified μ, σ when called as follows

```
Y = grand(m, n, 'nor', Av, Sd);
```

For example

```
-->grand(2,3,'nor',2,1)
 ans  =

    1.5434479    3.5310142    1.172681
    0.7914453    1.459521     3.5340618
```

8 References

1. Tacoma Narrows bridge failure: https://www.youtube.com/watch?v=j-zczJXSxnw
2. http://en.wikipedia.org/wiki/Diehard_tests
3. http://csrc.nist.gov/groups/ST/toolkit/random_number.html
4. http://en.wikipedia.org/wiki/Random_number_generation
5. http://en.wikipedia.org/wiki/Mersenne_twister
6. http://en.wikipedia.org/wiki/List_of_probability_distributions

9 Appendix – Scilab code

9.1 Linear congruential generator

```
0001  ////////////////////////////////////////////////////////////////////
0002  // randLCG.sci
0003  // 2014-12-08, Scott Hudson, for pedagocic purposes
0004  // 32-bit linear congruential generator for generating uniformly
0005  // distributed real numbers 0<=x<1.
0006  ////////////////////////////////////////////////////////////////////
0007  global randLCGseed randLCGa randLCGc
0008  randLCGseed = uint32(0);
0009
0010  function x=randLCG(seed)
0011    global randLCGseed randLCGa randLCGc
0012    [nargout,nargin] = argn();
0013    if (nargin==1) //if there is an argument, use it as the seed
0014      randLCGseed = uint32(seed);
0015    end
0016    randLCGseed = uint32(1664525)*randLCGseed+uint32(1013904223);
0017    x = double(randLCGseed)/4294967296.0;
0018  endfunction
```

9.2 Multiply-with-carry method

```
0001  ////////////////////////////////////////////////////////////////////
0002  // randMWC.sci
0003  // 2014-12-08, Scott Hudson, for pedagocic purposes
0004  // Multiply-with-carry algorithm for pseudo-random number generation.
0005  // Retuns uniformly distributed real number 0<x<1.
0006  // Reference: http://en.wikipedia.org/wiki/Random_number_generation
0007  ////////////////////////////////////////////////////////////////////
0008  global randMWCs1 randMWCs2
0009  randMWCs1 = uint32(1);
0010  randMWCs2 = uint32(2);
0011
0012  function x=randMWC(seed1,seed2)
0013    global randMWCs1 randMWCs2
0014    [nargout,nargin] = argn();
0015    if (nargin==2) //if there are arguments, use as seeds
0016      randMWCs1 = uint32(seed1); //should not be zero!
0017      randMWCs2 = uint32(seed2); //should not be zero!
0018    end
0019    s = uint32(2^16);
0020    randMWCs1 = 36969*modulo(randMWCs1,s)+randMWCs1/s;
0021    randMWCs2 = 18000*modulo(randMWCs2,s)+randMWCs2/s;
0022    x = double(randMWCs1*s+randMWCs2)/4294967296.0;
0023  endfunction
```