2WB05 Simulation
Lecture 5: Random-number generators

Marko Boon
http://www.win.tue.nl/courses/2WB05

December 6, 2012
Random-number generators

It is important to be able to efficiently generate independent random variables from the uniform distribution on $(0, 1)$, since:

- Random variables from all other distributions can be obtained by transforming uniform random variables;
- Simulations require many random numbers.
Random-number generators

A ‘good’ random-number generator should satisfy the following properties:

- **Uniformity**: The numbers generated appear to be distributed uniformly on \((0, 1)\);
- **Independence**: The numbers generated show no correlation with each other;
- **Replication**: The numbers should be replicable (e.g., for debugging or comparison of different systems).
- **Cycle length**: It should take long before numbers start to repeat;
- **Speed**: The generator should be fast;
- **Memory usage**: The generator should not require a lot of storage.
A ‘good’ random-number generator should satisfy the following properties:

- **Uniformity**: The numbers generated appear to be distributed uniformly on \((0, 1)\);
- **Independence**: The numbers generated show no correlation with each other;
- **Replication**: The numbers should be replicable (e.g., for debugging or comparison of different systems).
- **Cycle length**: It should take long before numbers start to repeat;
- **Speed**: The generator should be fast;
- **Memory usage**: The generator should not require a lot of storage.

- **Cryptographically secure**
Most random-number generators are of the form:

Start with $z_0$ (seed)
For $n = 1, 2, \ldots$ generate

$$z_n = f(z_{n-1})$$

and

$$u_n = g(z_n)$$

$f$ is the pseudo-random generator
$g$ is the output function

$\{u_0, u_1, \ldots\}$ is the sequence of uniform random numbers on the interval $(0, 1)$. 
Random-number generators

Midsquare method
Start with a 4-digit number $z_0$ (seed)
Square it to obtain 8-digits (if necessary, append zeros to the left)
Take the *middle 4 digits* to obtain the next 4-digit number $z_1$; then square $z_1$ and take the middle 4-digits again and so on.
We get uniform random number by placing the decimal point at the left of each $z_i$ (i.e., divide by 10000).
Random-number generators

Midsquare method

Examples

- For $z_0 = 1234$ we get $0.1234, 0.5227, 0.3215, 0.3362, 0.3030, 0.1809, 0.2724, 0.4201, 0.6484, 0.0422, 0.1780, 0.1684, 0.8361, 0.8561, 0.2907, ...$

- For $z_0 = 2345$ we get $0.2345, 0.4990, 0.9001, 0.0180, 0.0324, 0.1049, 0.1004, 0.0080, 0.0064, 0.0040, ...$ Two successive zeros behind the decimal will never disappear.

- For $z_0 = 2100$ we get $0.2100, 0.4100, 0.8100, 0.6100, 0.2100, 0.4100, ...$ Already after four numbers the sequence starts to repeat itself.

Clearly, random-number generators involve a lot more than doing ‘something strange’ to a number to obtain the next.
Linear congruential generators

Most random-number generators in use today are linear congruential generators. They produce a sequence of integers between 0 and \( m - 1 \) according to

\[ z_n = (az_{n-1} + c) \mod m, \quad n = 1, 2, \ldots \]

\( a \) is the multiplier, \( c \) the increment and \( m \) the modulus.

To obtain uniform random numbers on \((0, 1)\) we take

\[ u_n = z_n / m \]

A good choice of \( a, c \) and \( m \) is very important.
Linear congruential generators

A linear congruential generator has full period (cycle length is $m$) if and only if the following conditions hold:

- The only positive integer that exactly divides both $m$ and $c$ is 1;
- If $q$ is a prime number that divides $m$, then $q$ divides $a - 1$;
- If 4 divides $m$, then 4 divides $a - 1$. 
Random-number generators

Linear congruential generators

Examples:

• For \((a, c, m) = (1, 5, 13)\) and \(z_0 = 1\) we get the sequence 1, 6, 11, 3, 8, 0, 5, 10, 2, 7, 12, 4, 9, 1, ... which has full period (of 13).

• For \((a, c, m) = (2, 5, 13)\) and \(z_0 = 1\) we get the sequence 1, 7, 6, 4, 0, 5, 2, 9, 10, 12, 3, 11, 1, ... which has a period of 12. If we take \(z_0 = 8\), we get the sequence 8, 8, 8, ... (period of 1).
Random-number generators

Multiplicative congruential generators
These generators produce a sequence of integers between 0 and $m - 1$ according to

$$z_n = a z_{n-1} \mod m, \quad n = 1, 2, \ldots$$

So they are linear congruential generators with $c = 0$.

They cannot have full period, but it is possible to obtain period $m - 1$ (so each integer 1, ..., $m - 1$ is obtained exactly once in each cycle) if $a$ and $m$ are chosen carefully. For example, as $a = 630360016$ and $m = 2^{31} - 1$. 
Additive congruential generators

These generators produce integers according to

\[ z_n = (z_{n-1} + z_{n-k}) \mod m, \quad n = 1, 2, \ldots \]

where \( k \geq 2 \). Uniform random numbers can again be obtained from

\[ u_n = z_n / m \]

These generators can have a long period up to \( m^k \).

Disadvantage:
Consider the case \( k = 2 \) (the Fibonacci generator). If we take three consecutive numbers \( u_{n-2}, u_{n-1} \) and \( u_n \), then it will never happen that

\[ u_{n-2} < u_n < u_{n-1} \quad \text{or} \quad u_{n-1} < u_n < u_{n-2} \]

whereas for true uniform variables both of these orderings occur with probability 1/6.
Random-number generators

- Linear (or mixed) congruential generators
- Multiplicative congruential generators
- Additive congruential generators
- ...

Question 1: How **secure** are pseudorandom numbers?
Question 2: How **random** are pseudorandom numbers?
How (cryptographically) secure are pseudorandom numbers?

Desirable properties:

- given only a number produced by the generator, it is impossible to predict previous and future numbers;
- the numbers produced contain no known biases;
- the generator has a large period;
- the generator can seed itself at any position within that period with equal probability.

For example, when using the generator to produce a session ID on a web server: we don’t want user \( n \) to predict user \( n + 1 \)’s session ID.
How (cryptographically) secure are pseudorandom numbers?

Desirable properties:
- given only a number produced by the generator, it is impossible to predict previous and future numbers;
- the numbers produced contain no known biases;
- the generator has a large period;
- the generator can seed itself at any position within that period with equal probability.

For example, when using the generator to produce a session ID on a web server: we don’t want user $n$ to predict user $n + 1$’s session ID.

In Java, use `java.security.SecureRandom`.

Disadvantage: 20-30 times slower than `java.util.Random`. 
Choosing the initial seed

We need to find entropy or “genuine randomness” as the seed starting point for generating other random numbers. Some typical sources that in principle we may be able to access in software include:

- time between key presses,
- positions of mouse movements,
- amount of free memory,
- CPU temperature.

In practice: Operating Systems collect information and help to generate the random seed using

- time (wall-clock and since boot),
- performance and CPU counter data,
- timings of context switches and other software interrupts.
Choosing the initial seed

Reliable method: `java.security.SecureRandom` has a method called `generateSeed(int nrOfBytes)`

`java.util.Random` is a Linear Congruential Generator using a 48-bit seed. (Meaning that \( m = 2^{48} \), the other parameters are chosen such that the generator has maximum period.)

Old versions of this class used `System.currentTimeMillis()` as default random seed. Disadvantage: two instances created in the same millisecond generate the same sequence of pseudo-random numbers.
How random are pseudorandom numbers?

The random numbers generated by LCGs have the following undesirable property: randomness depends on the bit position!

In more detail: lower bits will be “less random” than the upper bits. For this reason, `Random.nextInt()` uses the top 32 bits of the 48-bit random seed.
Testing random-number generators

How random are pseudorandom numbers?

bit 1

bit 32
How random are pseudorandom numbers?

Try to test two main properties:

- Uniformity;
- Independence.
Uniformity or goodness-of-fit tests

Let $X_1, \ldots, X_n$ be $n$ observations. A goodness-of-fit test can be used to test the hypothesis:

$H_0$: The $X_i$’s are i.i.d. random variables with distribution function $F$.

Two goodness-of-fit tests:

- Kolmogorov-Smirnov test
- Chi-Square test
Kolmogorov-Smirnov test

Let \( F_n(x) \) be the empirical distribution function, so

\[
F_n(x) = \frac{\text{number of } X_i's \leq x}{n}
\]

Then

\[
D_n = \sup_x |F_n(x) - F(x)|
\]

has the Kolmogorov-Smirnov (K-S) distribution. Now we reject \( H_0 \) if

\[
D_n > d_{n,1-\alpha}
\]

where \( d_{n,1-\alpha} \) is the \( 1 - \alpha \) quantile of the K-S distribution.

Here \( \alpha \) is the significance level of the test: The probability of rejecting \( H_0 \) given that \( H_0 \) is true.
For $n \geq 100$, 

$$d_{n,0.95} \approx 1.3581/\sqrt{n}$$

In case of the uniform distribution we have 

$$F(x) = x, \quad 0 \leq x \leq 1.$$
Chi-Square test

Divide the range of $F$ into $k$ adjacent intervals

$$(a_0, a_1], (a_1, a_2], \ldots, (a_{k-1}, a_k]$$

Let

$$N_j = \text{number of } X_i \text{'s in } [a_{j-1}, a_j)$$

and let $p_j$ be the probability of an outcome in $(a_{j-1}, a_j]$, so

$$p_j = F(a_j) - F(a_{j-1})$$

Then the test statistic is

$$\chi^2 = \sum_{j=1}^{k} \frac{(N_j - np_j)^2}{np_j}$$

If $H_0$ is true, then $np_j$ is the expected number of the $n X_i$'s that fall in the $j$-th interval, and so we expect $\chi^2$ to be small.
Testing random-number generators

If $H_0$ is true, then the distribution of $\chi^2$ converges to a chi-square distribution with $k - 1$ degrees of freedom as $n \to \infty$.

The chi-square distribution with $k - 1$ degrees of freedom is the same as the Gamma distribution with parameters $(k - 1)/2$ and 2.

Hence, we reject $H_0$ if

$$\chi^2 > \chi^2_{k-1,1-\alpha}$$

where $\chi^2_{k-1,1-\alpha}$ is the $1 - \alpha$ quantile of the chi-square distribution with $k - 1$ degrees of freedom.
Chi-square test for $U(0, 1)$ random variables

We divide $(0, 1)$ into $k$ subintervals of equal length and generate $U_1, \ldots, U_n$; it is recommended to choose $k \geq 100$ and $n/k \geq 5$. Let $N_j$ be the number of the $n$ $U_i$’s in the $j$-th subinterval.

Then

$$
\chi^2 = \frac{k}{n} \sum_{j=1}^{k} \left( N_j - \frac{n}{k} \right)^2
$$
Example:
Consider the linear congruential generator

$$z_n = a z_{n-1} \mod m$$

with $a = 630360016$, $m = 2^{31} - 1$ and seed

$$z_0 = 1973272912$$

Generating $n = 2^{15} = 32768$ random numbers $U_i$ and dividing $(0, 1)$ in $k = 2^{12} = 4096$ subintervals yields

$$\chi^2 = 4141.0$$

Since

$$\chi_{4095.09}^2 \approx 4211.4$$

we do not reject $H_0$ at level $\alpha = 0.1$. 
Serial test
This is a 2-dimensional version of the chi-square test to test *independence* between successive observations.

We generate $U_1, \ldots, U_{2n}$; if the $U_i$’s are really i.i.d. $U(0, 1)$, then the nonoverlapping pairs

$$(U_1, U_2), (U_3, U_4), \ldots, (U_{2n-1}, U_{2n})$$

are i.i.d. random vectors uniformly distributed in the square $(0, 1)^2$.

- Divide the square $(0, 1)^2$ into $k^2$ subsquares;
- Count how many outcomes fall in each subsquare;
- Apply a chi-square test to these data.

This test can be generalized to higher dimensions.

Testing random-number generators
Testing random-number generators

Permutation test
Look at $n$ successive $d$-tuples of outcomes

$$(U_0, \ldots, U_{d-1}), (U_d, \ldots, U_{2d-1}),$$

$$\ldots, (U_{(n-1)d}, \ldots, U_{nd-1});$$

Among the $d$-tuples there are $d!$ possible orderings and these orderings are equally likely.

- Determine the frequencies of the different orderings among the $n d$-tuples;
- Apply a chi-square test to these data.
Testing random-number generators

Runs-up test
Divide the sequence $U_0, U_1, \ldots$ in blocks, where each block is a subsequence of *increasing* numbers followed by a number that is *smaller* than its predecessor.

Example: The realization 1,3,8,6,2,0,7,9,5 can be divided in the blocks (1,3,8), (2,0), (7,9,5).

A block consisting of $j + 1$ numbers is called a *run-up of length* $j$. It holds that

$$P(\text{run-up of length } j) = \frac{1}{j!} - \frac{1}{(j + 1)!}$$

- Generate $n$ run-ups;
- Count the number of run-ups of length 0, 1, 2, \ldots, $k - 1$ and $\geq k$;
- Apply a chi-square test to these data.
Correlation test

Generate $U_0, U_1, \ldots, U_n$ and compute an estimate for the (serial) correlation

$$
\hat{\rho}_1 = \frac{\sum_{i=1}^{n} (U_i - \bar{U}(n))(U_{i+1} - \bar{U}(n))}{\sum_{i=1}^{n} (U_i - \bar{U}(n))^2}
$$

where $U_{n+1} = U_1$ and $\bar{U}(n)$ the sample mean.

If the $U_i$'s are really i.i.d. $U(0,1)$, then $\hat{\rho}_1$ should be close to zero. Hence we reject $H_0$ is $\hat{\rho}_1$ is too large.

If $H_0$ is true, then for large $n$,

$$
P\left(-\frac{2}{\sqrt{n}} \leq \hat{\rho}_1 \leq \frac{2}{\sqrt{n}}\right) \approx 0.95
$$

So we reject $H_0$ at the 5% level if

$$
\hat{\rho}_1 \notin (-\frac{2}{\sqrt{n}}, \frac{2}{\sqrt{n}})
$$
If you still have problems with `java.util.Random` you can use the code given in the *Numerical recipes* (combining two XORShift generators with an LCG and a multiply with carry generator).

```java
private long u;
private long v = 4101842887655102017L;
private long w = 1;

public long nextLong() {
    u = u * 286293355777941757L + 7046029254386353087L;
    v ^= v >>> 17;
    v ^= v << 31;
    v ^= v >>> 8;
    w = 4294957665L * (w & 0xffffffff) + (w >>> 32);
    long x = u ^ (u << 21);
    x ^= x >>> 35;
    x ^= x << 4;
    long ret = (x + v) ^ w;
    return ret;
}
```

High quality generator