

# Performance Evaluation and Applications

 POLITECNICO DI MILANO

## Random Numbers Generation

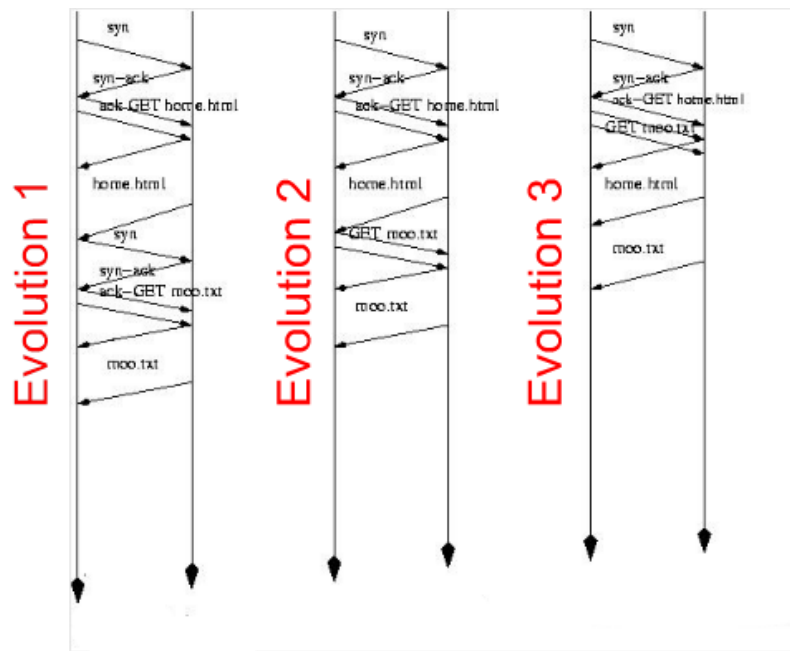
POLITECNICO DI MILANO



## Motivating example

Probability distributions allows to describe workload of a system in terms of inter-arrival and service times.

If we can generate numbers belonging to those specific distributions, we can create meaningful synthetic traces to study the evolutions of the performances of the system. But how?





## Pseudo random numbers

Most of mathematical packages and programming languages have primitives for generating numbers uniformly distributed in the zero-one interval.

Several techniques exist to generate random number distributed according to a specific probability distribution, starting from a uniform distribution in the 0...1 range.

We will then start focusing on algorithms that generate *pseudo random numbers* in the interval  $[0,1]$ .



## Pseudo random numbers

Since the behavior of a computer program is deterministic, random numbers are emulated by generating sequences of values that seems to be completely unrelated one-another.

The sequences however are deterministic and there is a point after which they repeat in the exact same order.

This is called the *period* of the generator.

0,285883864	0,062340012
0,249337778	0,243622939
0,957365572	0,552485985
0,150898169	0,429201756
0,000580347	0,16186627
0,062340012	0,697200783
0,243622939	0,34699925
0,552485985	0,449236644
0,429201756	0,087206649
0,16186627	0,089211615
0,697200783	0,438385352



## Pseudo random numbers

Sequences are identified by their initial value, which is called the *seed*.

When the same seed is used, the sequence is reproduced identically.

In order to produce arbitrary sequences, the seed is set to an always-changing value at the beginning of the simulation. This value usually corresponds to the system time.

0,062340012  
0,243622939  
0,552485985  
0,429201756  
0,16186627  
0,697200783  
0,34699925  
0,449236644  
0,087206649  
0,089211615  
0,438385352




0,063471556  
0,967716225  
0,495128158  
0,321749792  
0,068872067  
0,023403256  
0,41885377  
0,463690338  
0,531417012  
0,801000259  
0,421339817



## Pseudo random numbers

The ability of setting a specific seed allows studies to repeat identical sequences in different runs.

This feature is especially useful when debugging models where some problematic event may occur only very rarely.

0,062340012		0,063471556
0,243622939		0,967716225
0,552485985		0,495128158
0,429201756		0,321749792
0,16186627		0,068872067
0,697200783		0,023403256
0,34699925		0,41885377
0,449236644		0,463690338
0,087206649		0,531417012
0,089211615		0,801000259
0,438385352		0,421339817



## Linear Congruential Generator

Among the many algorithms that have been defined to generate random numbers in the 0...1 range, the simplest, yet effective one, is the *Linear Congruential Generator*.

Let us call  $X_n$  the  $n$ -th number of the sequence, with  $X_0$  corresponding to the seed. The algorithm computes  $X_n$  from  $X_{n-1}$  in the following way:

$$X_n = \text{mod}(a \cdot X_{n-1} + c, m)$$

Here  $\text{mod}(y, x)$  is the *remainder* of the division between  $y$  and  $x$ , so that there exists an integer  $k$  such that:

$$y = x \cdot k + \text{mod}(y, x) \\ \text{with } 0 \leq \text{mod}(y, x) < x$$

Where  $a$ ,  $c$  and  $m$  are three constants that identify the sequence.

In particular,  $m$  corresponds to the period of the sequence.



# Linear Congruential Generator

The algorithm generates numbers in the range  $[0, m-1]$ .

To obtain a sufficiently dense approximation of a uniform distribution in the  $[0, 1)$  or  $[0, 1]$  range, the value is divided by either  $m$  or  $m-1$ , depending on whether the limit value  $1$  should be included or not in the generation process.

The most common implementations exclude the upper bound from the generation.

$$X_n = \text{mod}(a \cdot X_{n-1} + c, m)$$

$$u_n = \frac{X_n}{m}$$





# Linear Congruential Generator

The choices of the parameters is crucial for the algorithm.

*Wikipedia* reports the values used in the most known libraries:

Source	modulus $m$	multiplier $a$	increment $c$
<a href="#">Numerical Recipes</a>	$2^{32}$	1664525	1013904223
<a href="#">Borland C/C++</a>	$2^{32}$	22695477	1
<a href="#">glibc</a> (used by <a href="#">GCC</a> ) <sup>[15]</sup>	$2^{31}$	1103515245	12345
<a href="#">ANSI C</a> : <a href="#">Watcom</a> , <a href="#">Digital Mars</a> , <a href="#">CodeWarrior</a> , <a href="#">IBM VisualAge C/C++</a> <sup>[16]</sup> <a href="#">C90</a> , <a href="#">C99</a> , <a href="#">C11</a> : Suggestion in the <a href="#">ISO/IEC 9899</a> , <sup>[17]</sup> <a href="#">C18</a>	$2^{31}$	1103515245	12345
<a href="#">Borland Delphi</a> , <a href="#">Virtual Pascal</a>	$2^{32}$	134775813	1
<a href="#">Turbo Pascal</a>	$2^{32}$	134775813 (8088405 <sub>16</sub> )	1
<a href="#">Microsoft Visual/Quick C/C++</a>	$2^{32}$	214013 (343FD <sub>16</sub> )	2531011 (269EC3 <sub>16</sub> )
<a href="#">Microsoft Visual Basic</a> (6 and earlier) <sup>[18]</sup>	$2^{24}$	1140671485 (43FD43FD <sub>16</sub> )	12820163 (C39EC3 <sub>16</sub> )
RtlUniform from <a href="#">Native API</a> <sup>[19]</sup>	$2^{31} - 1$	2147483629 (7FFFFFFD <sub>16</sub> )	2147483587 (7FFFFFFC3 <sub>16</sub> )
<a href="#">Apple CarbonLib</a> , <a href="#">C++11's</a> <code>minstd_rand0</code> <sup>[20]</sup>	$2^{31} - 1$	16807	0
<a href="#">C++11's</a> <code>minstd_rand</code> <sup>[20]</sup>	$2^{31} - 1$	48271	0



## Random variables generation

For finite discrete random variables, the uniform value  $u_i \sim \text{Unif}(0, 1)$  can be used in a simple algorithm to select the sampled value:

$$\Omega = \{a_1, \dots, a_N\} \qquad \sum_{j=1}^N p(a_j) = 1$$

```
for k=1:N
    if  $\sum_{j=1}^{k-1} p(a_j) \leq u_1 < \sum_{j=1}^k p(a_j)$ 
        return  $a_k$ 
    end
end
```

Usually, an array  $C(k)$  with  $N$  elements is constructed:

$$C(k) = \sum_{j=1}^k p(a_j)$$

The simplified procedure on the right can then be used  $\rightarrow$

```
for k=1:N
    if  $u_i < C(k)$ 
        return  $a_k$ 
    end
end
```

Since elements are ordered, if  $N$  is large, an algorithm with logarithmic complexity can be used. Its creation is left as an exercise.



## Random variables generation

The particular case of an integer uniform discrete distribution, between two extremes  $a$  and  $b$ , can be generated with a simple expression:

$$a_i = a + \lfloor u_i \cdot (b - a + 1) \rfloor$$

For completeness, the above expressions has a flaw if  $u_i$  also includes the upper bound, in the extremely rare event that exactly 1 is generated (which would occur once per period  $m$ ).

In this case, a more robust and precise expression is the following:

$$a_i = \min(a + \lfloor u_i \cdot (b - a + 1) \rfloor, b)$$



## Inverse Transform Sampling

The *Inverse transform sampling algorithm* allows generating samples according to any distributions, starting from a set of random numbers uniformly distributed between  $[0, 1]$ .

It is based on the fact that for every CDF, the values on the *y-axis* are uniformly distributed in the  $[0, 1]$  range.

If  $F_X(x)$  is the CDF of the considered distribution, then  $X$  can be computed from a number  $u$  uniformly distributed between  $[0, 1]$  in the following way:

$$u_i \sim X_{Unif<0,1>} \quad x_i = F_X^{-1}(u_i) \sim X$$

This expression is an equation where, given a random number  $u_i$ , it looks for a solution  $x_i$  such that:

$$F_X(x_i) = u_i$$



## Random variables generation with Inverse Transform

If  $v_i$  is  $Unif<0,1>$ , then, also  $u_i = 1-v_i$  is  $Unif<0,1>$  and we have:

$$u_i = F_{Unif<0,1>}(t) = \frac{t - a}{b - a}$$

$$a + (b - a)u_i \sim X_{Unif<0,1>}$$

$$v_i = F_{Exp<\lambda>}(t) = 1 - e^{-\lambda t}$$

$$\begin{aligned} 1-v_i &= e^{-\lambda t} \\ u_i &= e^{-\lambda t} \end{aligned}$$

$$-\frac{\ln(u_i)}{\lambda} \sim X_{Exp<\lambda>}$$

$$v_i = F_{Weibull<\lambda,k>}(t) = 1 - e^{-\left(\frac{t}{\lambda}\right)^k}$$

$$u_i = e^{-\left(\frac{t}{\lambda}\right)^k}$$

$$\lambda \sqrt[k]{-\ln(u_i)} \sim X_{Weibull<\lambda,k>}$$

$$v_i = F_{Pareto<\alpha,m>}(t) = 1 - \left(\frac{m}{t}\right)^\alpha$$

$$u_i = \left(\frac{m}{t}\right)^\alpha$$

$$\frac{m}{\sqrt[\alpha]{u_i}} \sim X_{Pareto<\alpha,m>}$$



## Random variables generation

Although the *Inverse transform sampling algorithm* is theoretically always applicable, in many cases it might not be the best solution.

In particular, if the inverse of the CDF, function  $F_X^{-1}(u)$ , does not have an analytical closed form expression, or it is particularly complex to compute, the *Inverse transform sampling algorithm* can be particularly inefficient.

$$u_i \sim X_{Unif<0,1>} \quad x_i = F_X^{-1}(u_i) \sim X$$



## Random variables generation: Erlang

In many cases we can determine procedures for generating samples in a more efficient way, by taking into account the properties of the considered distributions.

An Erlang distribution with  $k$  stages of rate  $\lambda$ , can be computed by summing up  $k$  samples generated from an exponential distribution with parameter  $\lambda$ .

$$\begin{aligned} X_{Erlang<\lambda,k>} &= \sum_{i=1}^k X_{Exp<\lambda>} = -\frac{\sum_{i=1}^k \ln(u_i)}{\lambda} \\ &= -\frac{\ln(\prod_{i=1}^k u_i)}{\lambda} \end{aligned}$$



## Random variables generation

The *HypoExponential* distribution, can be generated by summing up  $n$  samples from  $n$  exponential distributions, each one characterized by its rate  $\lambda_i$ .

$$\begin{aligned} X_{Hypo-Exp<\lambda_1, \dots, \lambda_n>} &= X_{Exp<\lambda_1>} + \dots + X_{Exp<\lambda_n>} \\ &= -\frac{\ln(u_1)}{\lambda_1} - \dots - \frac{\ln(u_n)}{\lambda_n} \end{aligned}$$





## Random variables generation

Hyper exponential can be generated by first using a discrete random variable, characterized by probabilities  $p_1 \dots p_n$ , to select a stage  $i$ . Then, the corresponding exponential distribution of parameter  $\lambda_i$  can be used to determine the final sample.

This technique always requires two samples,  $u_1$  and  $u_2$ , regardless of the number of stages  $n$ .

$$X_{Hyper-exp<\lambda_i, p_i>} = \begin{cases} X_{Exp<\lambda_1>} & p_1 \\ \dots & \dots \\ X_{Exp<\lambda_n>} & p_n \end{cases}$$

```
for k=1:n
    if  $\sum_{j=1}^{k-1} p_j < u_1 \leq \sum_{j=1}^k p_j$ 
        return  $-\frac{\ln(u_2)}{\lambda_k}$ 
    end
end
```

with  $C(k) = \sum_{j=1}^k p(a_j)$

```
for k=1:N
    if  $u_1 \leq C(k)$ 
        return  $-\frac{\ln(u_2)}{\lambda_k}$ 
    end
end
```



## Random variables generation

A similar procedure can be used also for the *HyperErlang*, combining both a discrete distribution to select the branch, and the technique seen for the *Erlang* distribution to determine the final sample from the selected stage.

$$X_{Hyper-Erlang<\lambda_i, k_i, p_i>} = \begin{cases} X_{Erlang<\lambda_1, k_1>} & p_1 \\ \dots & \dots \\ X_{Erlang<\lambda_n, k_n>} & p_n \end{cases}$$



## Normal random variables generation

The *Box-Muller* method can be used to generate *two independent samples* of a *Standard Normal* distribution starting from two random variables  $u$  and  $v$  in the range  $[0,1]$ .

$$u_i \sim X_{Unif<0,1>} \quad v_i \sim X_{Unif<0,1>}$$

$$x_i = \sin(2\pi v_i) \sqrt{-2 \ln(u_i)}$$

$$y_i = \cos(2\pi v_i) \sqrt{-2 \ln(u_i)}$$

$$x_i \sim X_{N<0,1>} \quad y_i \sim X_{N<0,1>}$$



## Truncated normal

Generation of samples from a *Truncated Normal Distribution* can be done with a simple rejection algorithm:

```
repeat
     $x_i = \text{GenNormalRandomSample}(\mu, \sigma^2)$ 
until  $x_i \geq 0$ 
```

In this way, however, the number of samples required before finding the desired one is also random, and it depends on the number of negative values that are discarded before finding a positive one.



## Log Normal and Gamma

Random numbers from a *Log Normal Distribution* can be generated by simply applying its definition, starting from a normally distributed random value (generated for example with the Box-Muller technique).

$$x_i = \exp(\text{GenNormalRandomSample}(\mu, \sigma^2))$$

Generation of random numbers from the *Gamma* distribution instead require a rather complex procedures, which should be chosen appropriately, according to the shape parameter. For this reason we will consider the generation of samples from the *Gamma* distribution outside the scope of this course.



## Random numbers required

Each distribution, can then be generated with a different number of samples in the  $[0,1)$  intervals:

Distribution	N. samples	Meaning
Discrete ( $k$ values)	1	Selection threshold
Deterministic	0	Just return the value
Uniform	1	Inverse transform
Exponential	1	Inverse transform
Hyper-exponential ( $k$ branches)	2	Select the branch, then exponential
Hypo-exponential ( $k$ stages)	$k$	One exponential per stage
Erlang $k$	$k$	One exponential per stage
Hyper-Erlang ( $k$ branches)	$1 + k_i$	Select the branch, then Erlang
Pareto	1	Inverse transform
Weibull	1	Inverse transform



# Analysis of Motivating Example

As seen, starting from a given distribution, we can generate samples using the corresponding technique.

