Performance Modelling of Computer Systems

Mirco Tribastone

Institut für Informatik
Ludwig-Maximilians-Universität München

Discrete-Event Simulation

Discrete-Event Simulation

Overview

So far, performance models have been analysed through the solution of a mathematical problem (i.e., a system of linear equations, a system of coupled differential equations, etc.).

Simulation is an alternative approach which produces sample traces of the stochastic process under study.

The generation of these traces requires access to some object which is capable of providing random numbers.

When implemented on a computer, pseudo-random generators approximate sequences of random numbers satisfying a desired probability distribution.

Pseudo-random Generators: Java

```
/**
 * Returns a <code>double</code> value with a positive sign, greater
 * than or equal to <code>0.0</code> and less than <code>1.0</code>.
 * Returned values are chosen pseudorandomly with (approximately)
 * uniform distribution from that range.
 *
 * <p>When this method is first called, it creates a single new
 * pseudorandom-number generator, exactly as if by the expression
 * <blockquote><pre>new java.util.Random</pre></blockquote> This
 * new pseudorandom-number generator is used thereafter for all
 * calls to this method and is used nowhere else.
 *
 * <p>This method is properly synchronized to allow correct use by
 * more than one thread. However, if many threads need to generate
 * pseudorandom numbers at a great rate, it may reduce contention
 * for each thread to have its own pseudorandom-number generator.
 *
 * @return  a pseudorandom <code>double</code> greater than or equal
 * to <code>0.0</code> and less than <code>1.0</code>.
 * @see     java.util.Random#nextDouble()
 */
public static double random() {
    if (randomNumberGenerator == null) initRNG();
    return randomNumberGenerator.nextDouble();
}
```

Pseudo-random Generators: Matlab

rand

Uniformly distributed pseudorandom numbers

Syntax

```
r = rand(n)
rand(m,n)
rand([m,n])
rand(m,n,p,...)
rand([m,n,p,...])
rand
rand(size(A))
r = rand(..., 'double')
r = rand(..., 'single')
```

Description

`r = rand(n)` returns an `n`-by-`n` matrix containing pseudorandom values drawn from the standard uniform distribution on the open interval (0,1). `rand(m,n)` or `rand([m,n])` returns an `m`-by-`n` matrix. `rand(m,n,p,...)` or `rand([m,n,p,...])` returns an `m`-by-`n`-by-`p`-by-... array. `rand` returns a scalar. `rand(size(A))` returns an array the same size as **`A`**.

`r = rand(..., 'double')` or `r = rand(..., 'single')` returns an array of uniform values of the specified class.

Discrete-Event Simulation

Discrete-event simulation is a technique in which changes of the state of a system are determined by the occurrence of some event.

Discrete-Event Simulation

Discrete-event simulation is a technique in which changes of the state of a system are determined by the occurrence of some event.

The simulator maintains a global variable, `currentTime`, which gives current simulated time (usually initialised to 0).

Discrete-Event Simulation

Discrete-event simulation is a technique in which changes of the state of a system are determined by the occurrence of some event.

The simulator maintains a global variable, `currentTime`, which gives current simulated time (usually initialised to 0).

An event is an object which has a property called `firingTime` which gives the simulated time when the event is scheduled.

Discrete-Event Simulation

Discrete-event simulation is a technique in which changes of the state of a system are determined by the occurrence of some event.

The simulator maintains a global variable, `currentTime`, which gives current simulated time (usually initialised to 0).

An event is an object which has a property called `firingTime` which gives the simulated time when the event is scheduled.

The simulator keeps a global scheduler which maintains a list of events ordered by increasing firing times (the event with the smallest firing time is at the top of this list).

Discrete-Event Simulation

Discrete-event simulation is a technique in which changes of the state of a system are determined by the occurrence of some event.

The simulator maintains a global variable, `currentTime`, which gives current simulated time (usually initialised to 0).

An event is an object which has a property called `firingTime` which gives the simulated time when the event is scheduled.

The simulator keeps a global scheduler which maintains a list of events ordered by increasing firing times (the event with the smallest firing time is at the top of this list).

The simulation algorithm pops the first event off the list, advances `currentTime` to the event's `firingTime`, and changes the state of the system according to the kind of event being processed.

Discrete-Event Simulation

When an event is processed, the simulation typically updates some data structures which are needed to observe the indices of performance of interest to the modeller.

Discrete-Event Simulation

When an event is processed, the simulation typically updates some data structures which are needed to observe the indices of performance of interest to the modeller.

The processing of an event may in turn generate some other event, which will be then scheduled according to its firing time.

Discrete-Event Simulation

When an event is processed, the simulation typically updates some data structures which are needed to observe the indices of performance of interest to the modeller.

The processing of an event may in turn generate some other event, which will be then scheduled according to its firing time.

The algorithm may terminate when the event list is empty.

Discrete-Event Simulation

When an event is processed, the simulation typically updates some data structures which are needed to observe the indices of performance of interest to the modeller.

The processing of an event may in turn generate some other event, which will be then scheduled according to its firing time.

The algorithm may terminate when the event list is empty.

For ergodic models, the algorithm terminates when a given time horizon is reached (the event list will never be empty).

Discrete-Event Simulation

Example

We wish to study a system in which users arrive with exponential inter-arrival times with mean $1/\lambda$, with $\lambda > 0$. We are interested to know how many users there are in the system after 10 time units.

Discrete-Event Simulation

Example

We wish to study a system in which users arrive with exponential inter-arrival times with mean $1/\lambda$, with $\lambda > 0$. We are interested to know how many users there are in the system after 10 time units.

To simulate this system, we consider the arrival of an user as a discrete event. When this is processed, we record the current time and the total number of users at that time. Then, we schedule a new arrival event for the next user.

Example

We wish to study a system in which users arrive with exponential inter-arrival times with mean $1/\lambda$, with $\lambda > 0$. We are interested to know how many users there are in the system after 10 time units.

To simulate this system, we consider the arrival of an user as a discrete event. When this is processed, we record the current time and the total number of users at that time. Then, we schedule a new arrival event for the next user.

In the object-oriented pseudo-code that we will use, an arrival is a class which is constructed with a parameter denoting the firing time of the event.

Algorithm

```
currentTime = 0
nUsers = 0
Trace = {(currentTime, nUsers)}
new Arrival(currentTime + getRandomExp())
while EventList ≠ ∅ do
   arrival = EventList.pop()
   currentTime = arrival.firingTime
   if currentTime > 10 then
      break
   end if
   nUsers = nUsers + 1
   Trace = Trace ∪ {(currentTime, nUsers)}
   new Arrival(currentTime + getRandomExp())
end while
```

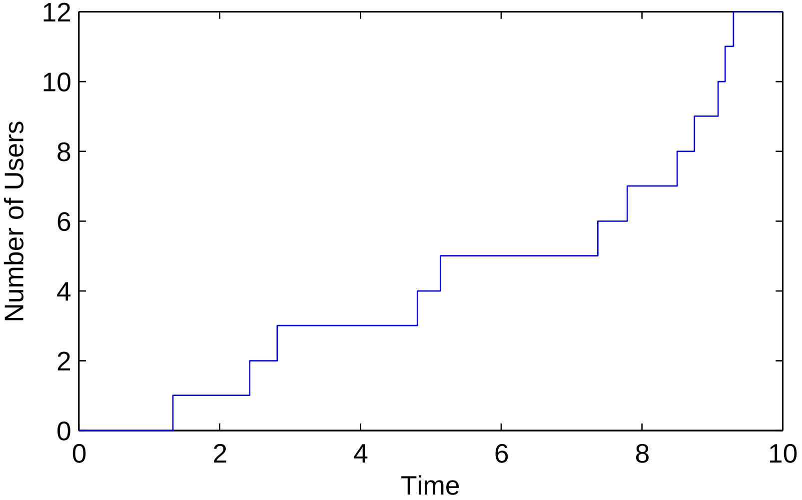
Algorithm

```
currentTime = 0
nUsers = 0
Trace = {(currentTime, nUsers)}
new Arrival(currentTime + getRandomExp())
while EventList ≠ ∅ do
    arrival = EventList.pop()
    currentTime = arrival.firingTime
    if currentTime > 10 then
        break
    end if
    nUsers = nUsers + 1
    Trace = Trace ∪ {(currentTime, nUsers)}
    new Arrival(currentTime + getRandomExp())
end while
```

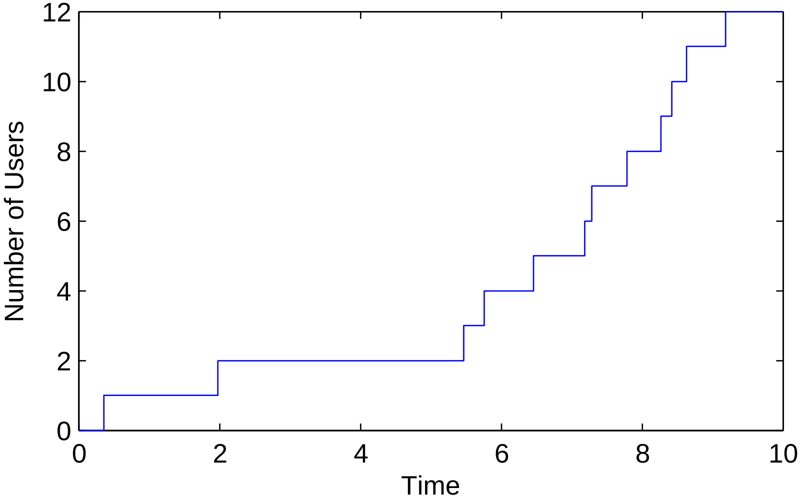
Notes:

- This algorithm may be simplified
- The data structure Trace allows us to plot the sample path obtained

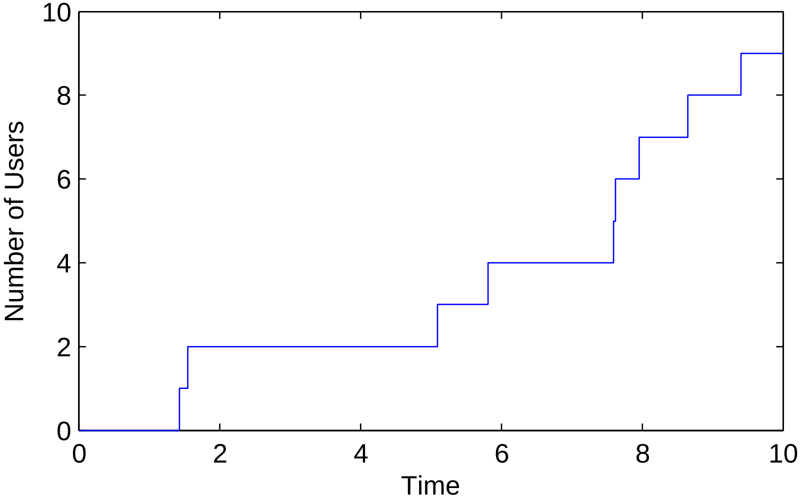
Trace of a Simulation Run



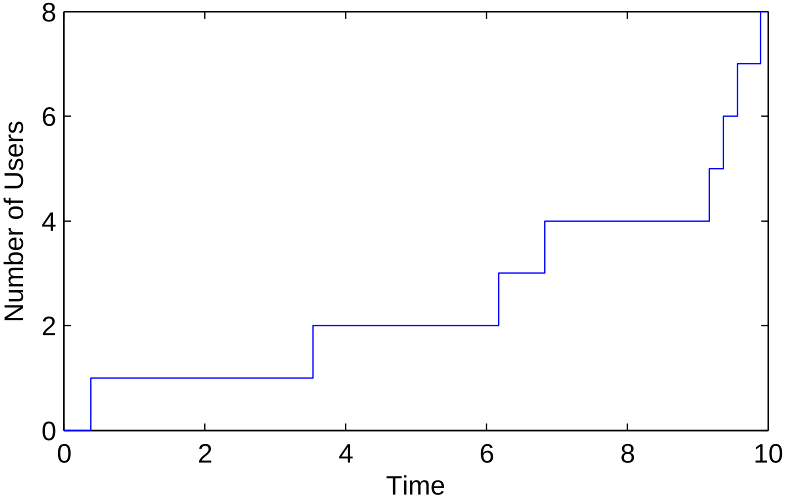
Some Traces



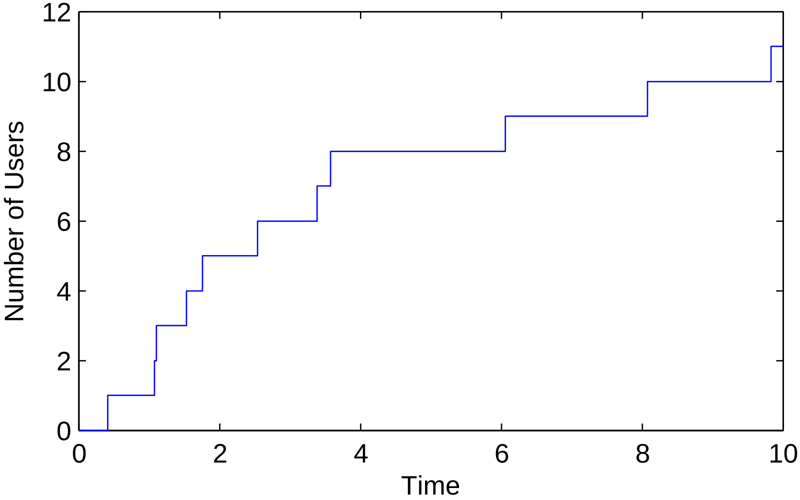
Some Traces



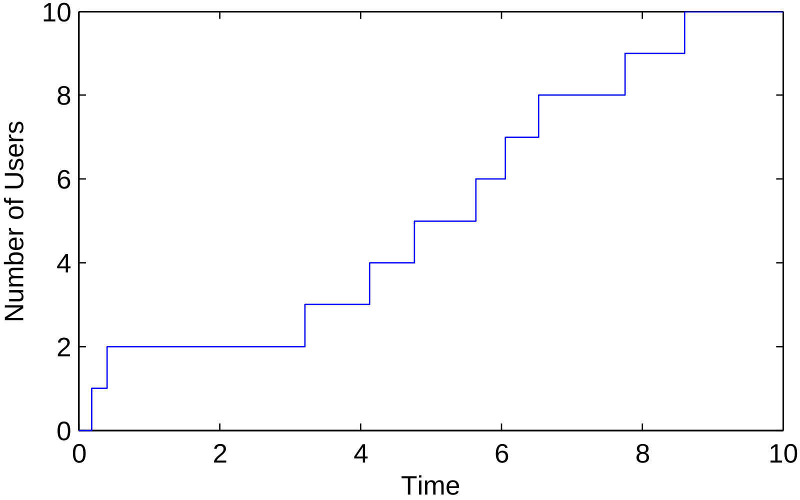
Some Traces



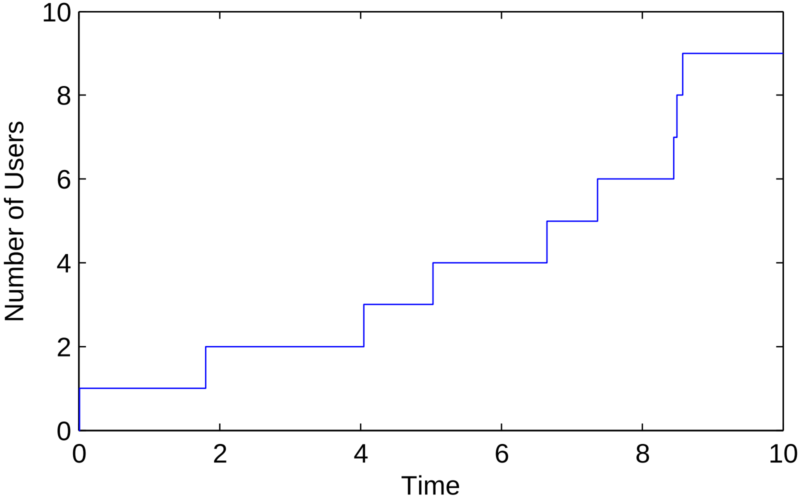
Some Traces



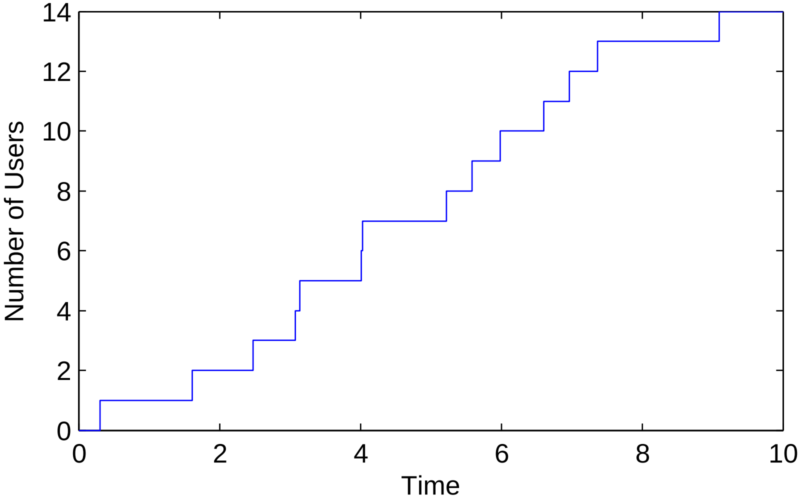
Some Traces



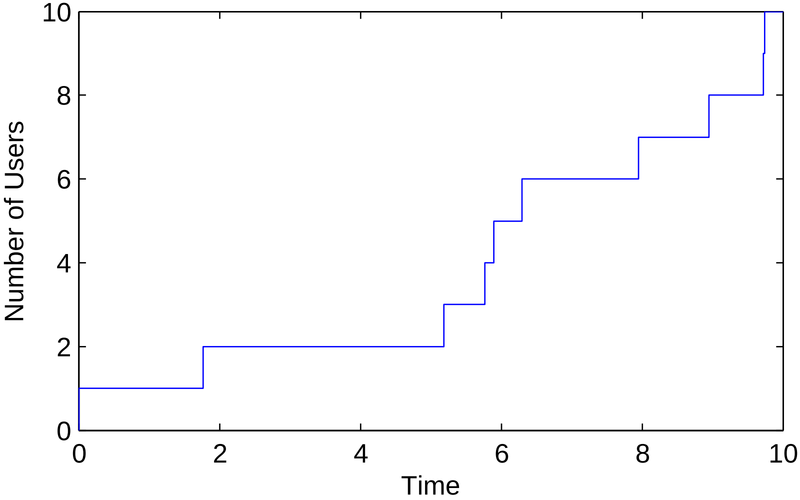
Some Traces



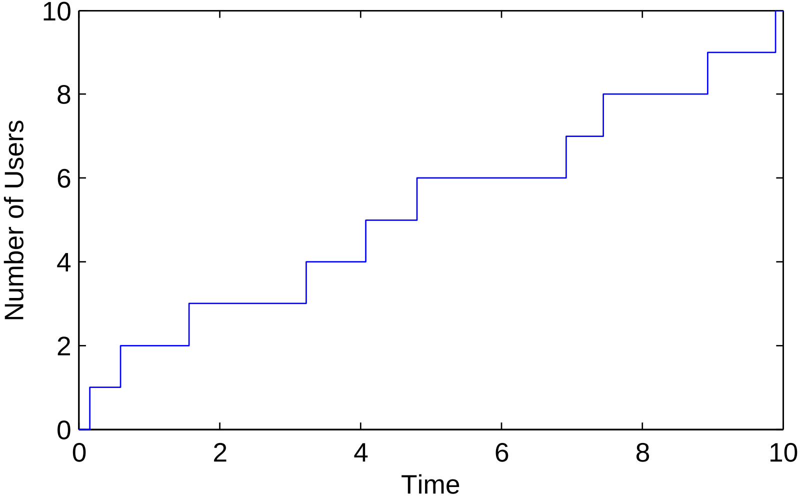
Some Traces



Some Traces



Some Traces

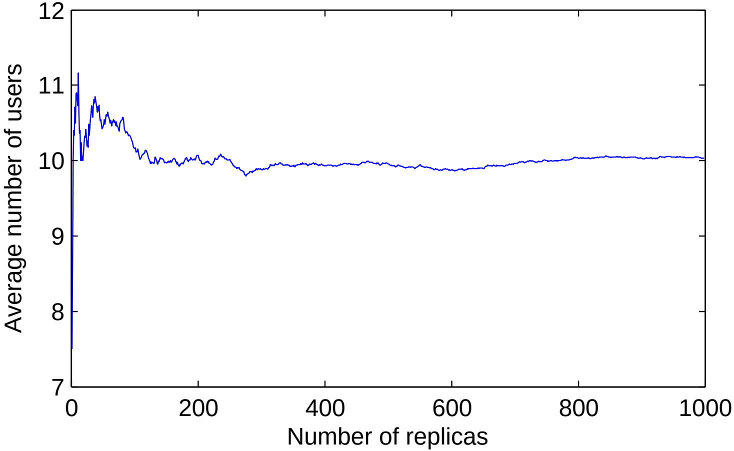


Termination Criteria

Suppose that we are interested to know what is the expected number of users after ten time units: How many traces do we need?

Termination Criteria

Suppose that we are interested to know what is the expected number of users after ten time units: How many traces do we need?



Termination Criteria

For this particular model we know that the exact expected value is 10.

Termination Criteria

For this particular model we know that the exact expected value is 10.

After 1000 replications we have a statistical average of 10.290, hence an absolute relative percentage error

$$\%\ \text{Error} = \left| \frac{10.000 - 10.290}{10.000} \right| \times 100 = 2.9\%$$

Termination Criteria

For this particular model we know that the exact expected value is 10.

After 1000 replications we have a statistical average of 10.290, hence an absolute relative percentage error

$$\%\ \text{Error} = \left| \frac{10.000 - 10.290}{10.000} \right| \times 100 = 2.9\%$$

A termination condition could be based on this error being less than some given threshold. Unfortunately, in general we do not know the exact solution — otherwise we would not simulate the model — therefore termination conditions must be based on some form of statistical analysis of the samples obtained.

Confidence Intervals

We consider a simulation with n runs in which the i-th run gives a random observation x_i (for instance, the total number of users after 10 time units in the previous example). We assume that x_i are independent and identically distributed.

Confidence Intervals

We consider a simulation with n runs in which the i-th run gives a random observation x_i (for instance, the total number of users after 10 time units in the previous example). We assume that x_i are independent and identically distributed.

We wish to know the accuracy of the statistical average $\hat{\mu} = \sum_{i=1}^{n} x_i$.

Confidence Intervals

We consider a simulation with n runs in which the i-th run gives a random observation x_i (for instance, the total number of users after 10 time units in the previous example). We assume that x_i are independent and identically distributed.

We wish to know the accuracy of the statistical average $\hat{\mu} = \sum_{i=1}^{n} x_i$.

A confidence interval for the true mean m is a probability bound of the kind

$$\mathbb{P}(a < m < b) \geq \xi$$

where a and b are obtained from the data and ξ is a given confidence level (typically, $\xi = 0.95$ or $\xi = 0.99$).

Confidence Intervals

For sufficiently large n, an approximate confidence interval for the mean at level $1 - \alpha$ is

$$\hat{\mu} \pm \eta \frac{s_n}{\sqrt{n}},$$

where s_n is the statistical standard deviation

$$s_n^2 = \frac{1}{n} \sum_{i=1}^{n} \left(x_i - \hat{\mu}_n^2 \right)$$

and η is the $(1 - \alpha/2)$ quantile of the normal distribution $N_{0,1}$.

Confidence Intervals

For sufficiently large n, an approximate confidence interval for the mean at level $1 - \alpha$ is

$$\hat{\mu} \pm \eta \frac{s_n}{\sqrt{n}},$$

where s_n is the statistical standard deviation

$$s_n^2 = \frac{1}{n} \sum_{i=1}^{n} \left(x_i - \hat{\mu}_n^2 \right)$$

and η is the $(1 - \alpha/2)$ quantile of the normal distribution $N_{0,1}$.

For a desired α the corresponding value of η, η_α, is available in ready-to-use tables. For instance, for $\eta_{0.05} = 1.960$ (95% confidence interval) and $\eta_{0.01} = 2.576$ (99% confidence interval).

Example

Let us define the percentage error as $\left|\eta \frac{s_n}{\sqrt{n}} / \hat{\mu}\right| \times 100$.

Example

Let us define the percentage error as $\left|\eta \frac{s_n}{\sqrt{n}} / \hat{\mu}\right| \times 100$.

For our Poisson example, we obtained the following results:

Iteration	*Mean*	*Error 95%*	*Error 99%*
1000	10.190	1.882	2.474
5000	9.996	0.871	1.145
10000	9.993	0.619	9.813
15000	10.011	0.506	0.665
20000	10.004	0.438	0.575
25000	10.011	0.391	0.514
30000	9.988	0.358	0.471

Example

Let us define the percentage error as $\left|\eta \frac{s_n}{\sqrt{n}} / \hat{\mu}\right| \times 100$.

For our Poisson example, we obtained the following results:

Iteration	*Mean*	*Error 95%*	*Error 99%*
1000	10.190	1.882	2.474
5000	9.996	0.871	1.145
10000	9.993	0.619	9.813
15000	10.011	0.506	0.665
20000	10.004	0.438	0.575
25000	10.011	0.391	0.514
30000	9.988	0.358	0.471

To halve the error requires a fourfold increase in the number of samples!

Steady-State Simulation

The method of independent replicas is particularly suited to simulating models when transient measures are to be estimated.

Steady-State Simulation

The method of independent replicas is particularly suited to simulating models when transient measures are to be estimated.

The method of batch means instead is preferred when one is interested in steady-state simulation. With the former method, the average is obtained across samples. With the latter the average is a time average.

Steady-State Simulation

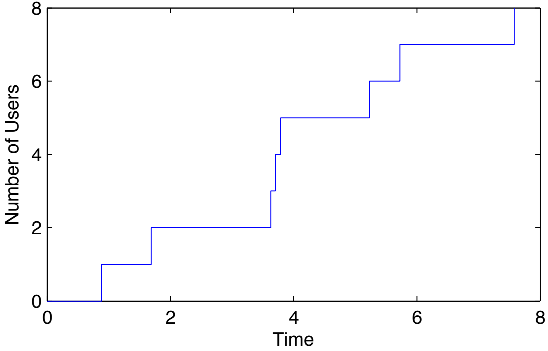
The method of independent replicas is particularly suited to simulating models when transient measures are to be estimated.

The method of batch means instead is preferred when one is interested in steady-state simulation. With the former method, the average is obtained across samples. With the latter the average is a time average.

The simulation consists of a single long run of length T, which tracks the evolution of a performance index, $I(t)$. The average across that index is given by

$$\hat{\mu} = \frac{1}{T} \int_0^T I(t) dt$$

Time Average



$$\hat{\mu} = \frac{1}{10} \times \big[0 \times 0.88 + 1 \times (1.69 - 0.88) + 2 \times (3.63 - 1.69) + 3 \times (3.70 - 3.63) + 4 \times (3.79 - 3.70) + 5 \times (5.23 - 3.79) + 6 \times (5.72 - 5.23) + 7 \times (7.58 - 5.72) + 7 \times (8 - 7.58)\big]$$

Accuracy of Steady-State Simulation

To compute confidence intervals, the single run is divided into b batches of the same length T/b and the following estimates are computed

$$\hat{\mu}_i = \frac{b}{T} \int_{(i-1)T/b}^{iT/b} I(t)dt, \qquad \text{for } i = 1, \ldots b$$

Accuracy of Steady-State Simulation

To compute confidence intervals, the single run is divided into b batches of the same length T/b and the following estimates are computed

$$\hat{\mu}_i = \frac{b}{T} \int_{(i-1)T/b}^{iT/b} I(t)dt, \qquad \text{for } i = 1, \ldots b$$

Then, a confidence interval is computed for the $\hat{\mu}_i$, as usual.

Accuracy of Steady-State Simulation

To compute confidence intervals, the single run is divided into b batches of the same length T/b and the following estimates are computed

$$\hat{\mu}_i = \frac{b}{T} \int_{(i-1)T/b}^{iT/b} I(t)dt, \qquad \text{for } i = 1, \ldots b$$

Then, a confidence interval is computed for the $\hat{\mu}_i$, as usual.

Often, the model is subjected to transient removal, i.e., the statistics are not collected over some initial period of time to exclude those samples which are related to the stochastic process being away from stationary conditions.

Stochastic Simulation for Java

- Stochastic Simulation for Java is a framework for discrete-event simulation for Java
- It is freely available at
  `http://www.iro.umontreal.ca/~simardr/ssj/indexen.html`
- It supports a wide array of random number generators, statistics collectors, a graph visualisation toolkit, etc.

A Queue with SSJ

```java
import umontreal.iro.lecuyer.simevents.*;
import umontreal.iro.lecuyer.rng.*;
import umontreal.iro.lecuyer.randvar.*;
import umontreal.iro.lecuyer.stat.*;
import java.util.LinkedList;

public class QueueEv {

   RandomVariateGen genArr;
   RandomVariateGen genServ;
   LinkedList<Customer> waitList = new LinkedList<Customer> ();
   LinkedList<Customer> servList = new LinkedList<Customer> ();
   Tally custWaits     = new Tally ("Waiting times");
   Accumulate totWait  = new Accumulate ("Size of queue");

   class Customer { double arrivTime, servTime; }

   public QueueEv (double lambda, double mu) {
      genArr = new ExponentialGen (new MRG32k3a(), lambda);
      genServ = new ExponentialGen (new MRG32k3a(), mu);
   }

   public void simulateOneRun (double timeHorizon) {
      Sim.init();
      new EndOfSim().schedule (timeHorizon);
      new Arrival().schedule (genArr.nextDouble());
      Sim.start();
   }
```

A Queue with SSJ

```
class Arrival extends Event {
   public void actions() {
      new Arrival().schedule (genArr.nextDouble()); // Next arri
      Customer cust = new Customer();  // Cust just arrived.
      cust.arrivTime = Sim.time();
      cust.servTime = genServ.nextDouble();
      if (servList.size() > 0) {       // Must join the queue.
         waitList.addLast (cust);
         totWait.update (waitList.size());
      } else {                         // Starts service.
         custWaits.add (0.0);
         servList.addLast (cust);
         new Departure().schedule (cust.servTime);
      }
   }
}
class Departure extends Event {
   public void actions() {
      servList.removeFirst();
      if (waitList.size() > 0) {
         // Starts service for next one in queue.
         Customer cust = waitList.removeFirst();
         totWait.update (waitList.size());
         custWaits.add (Sim.time() - cust.arrivTime);
         servList.addLast (cust);
         new Departure().schedule (cust.servTime);
      }
   }
}
```

A Queue with SSJ

```
    class EndOfSim extends Event {
       public void actions() {
          Sim.stop();
       }
    }

    public static void main (String[] args) {
       QueueEv queue = new QueueEv (1.0, 2.0);
       queue.simulateOneRun (1000.0);
       System.out.println (queue.custWaits.report());
       System.out.println (queue.totWait.report());
    }
}
```

Monte Carlo Simulation

- Discrete-event simulation does not make assumptions on the simulated system.
- In particular, no assumptions are made on the probability distributions.
- Monte Carlo simulation is an algorithm for continuous-time Markov chain, which exploits the properties of the exponential distribution.

Monte Carlo Simulation

In a homogeneous CTMC,

$$q_{ij} = \lim_{\Delta t \to 0} \frac{p_{ij}(t, t + \Delta t)}{\Delta t}, \qquad \text{for all } t.$$

Monte Carlo Simulation

In a homogeneous CTMC,

$$q_{ij} = \lim_{\Delta t \to 0} \frac{p_{ij}(t, t + \Delta t)}{\Delta t}, \qquad \text{for all } t.$$

Given a state i, the holding time is exponentially distributed with rate $\sum_j q_{ij} = -q_{ii}$. Drawing x from a uniform distribution in $(0, 1)$ and computing

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

gives a sample of the holding time in i (see early tutorial).

Monte Carlo Simulation

In a homogeneous CTMC,

$$q_{ij} = \lim_{\Delta t \to 0} \frac{p_{ij}(t, t + \Delta t)}{\Delta t}, \qquad \text{for all } t.$$

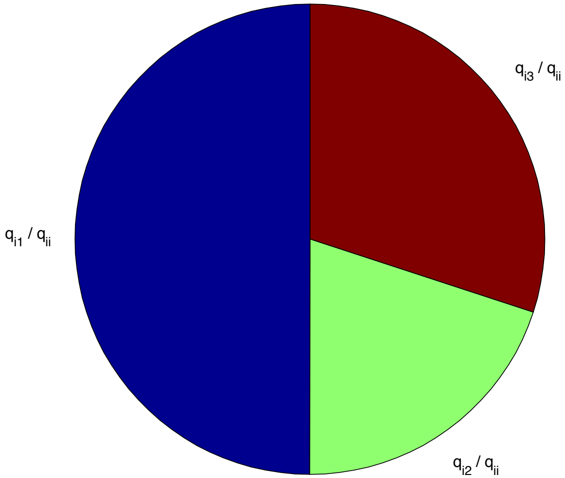
Given a state i, the holding time is exponentially distributed with rate $\sum_j q_{ij} = -q_{ii}$. Drawing x from a uniform distribution in $(0, 1)$ and computing

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

gives a sample of the holding time in i (see early tutorial).

The probability that the transition from i to k happens is given by $q_{ik} / \sum_j q_{ij}$. A sample is obtained by drawing z from a uniform distribution in $(0, 1)$ and choosing the smallest k such that $z > \frac{\sum_{j=1}^{k-1} q_{ij}}{\sum_j q_{ij}}$.

Transition Selection



Monte Carlo Simulation: Algorithm

$t = 0$
$s = s_1$
while termination criteria are not satisfied **do**
  draw x, y in $U(0, 1)$
  $\Delta t = \frac{1}{q_{ss}} \ln(1 - x)$
  chose smallest k such that $y > \frac{\sum_{j=1}^{k-1} q_{sj}}{\sum_j q_{sj}}$
  $t = t + \Delta t$
  $s = k$
  record transition from s to k in Δt units
end while