Performance of Computer Communication Systems: A Model-Based Approach.
Boudewijn R. Haverkort

ISBNs: 0-471-97228-2 (Hardback); 0-470-84192-3 (Electronic)

Part V

Simulation

Performance of Computer Communication Systems: A Model-Based Approach.
Boudewijn R. Haverkort

ISBNs: 0-471-97228-2 (Hardback); 0-470-84192-3 (Electronic)

Chapter 18

Simulation: methodology and statistics

In the previous chapters we have addressed models that can be solved by analytical or numerical means. Although the class of addressed models has been very wide, there are still models that cannot be solved adequately with the presented techniques. These models, however, can still be analysed using simulation. With simulation there are no fundamental restrictions towards what models can be solved. Practical restrictions do exist since the amount of computer time or memory required for running a simulation can be prohibitively large.

In this chapter we concentrate on the general set-up of simulations as well as on the statistical aspects of simulation studies. To compare the concept of simulation with analytical and numerical techniques we discuss the application of simulation for the computation of an integral in Section 18.1. Various forms of simulation are then classified in Section 18.2. Implementation aspects for so-called discrete event simulations are discussed in Section 18.3. In order to execute simulation programs, realisations of random variables have to be generated. This is an important task that deserves special attention since a wrong or biased number generation scheme can severely corrupt the outcome of a simulation. Random number generation is therefore considered in section 18.4. The gathering of measurements from the simulation and their processing is finally discussed in Section 18.5.

18.1 The idea of simulation

Consider the following mathematical problem. One has to obtain the (unknown) area α under the curve $y = f(x) = x^2$, from $x = 0$ to $x = 1$. Let $\tilde{a}$ denote the result of the

calculation we perform to obtain this value. Since $f(x)$ is a simple quadratic term, this problem can easily be solved *analytically*:

$$\tilde{a} = \int_0^1 x^2 dx = \left(\frac{1}{3}x^3\right)_{x=0}^{x=1} = \frac{1}{3}. \tag{18.1}$$

Clearly, in this case, the calculated value $\tilde{a}$ is exactly the same as the real value α.

Making the problem somewhat more complicated, we can pose the same question when $f(x) = x^{\sin x}$. Now, we cannot solve the problem analytically any more (as far as we have consulted integration tables). We can, however, resort to a numerical technique such as the trapezoid rule. We then have to split the interval $[0, 1]$ into n consecutive intervals $[x_0, x_1]$, $[x_1, x_2], \cdots$, $[x_{n-1}, x_n]$ so that the area under the curve can be approximated as:

$$\tilde{a} = \frac{1}{2}\sum_{i=1}^{n}(x_i - x_{i-1})(f(x_i) + f(x_{i-1})). \tag{18.2}$$

By making the intervals sufficiently small $\tilde{a}$ will approximate a with any level of desired accuracy. This is an example of a *numerical* solution technique.

Surprisingly, we can also obtain a reasonable estimate $\tilde{a}$ for α by means of *stochastic simulation*. Studying $f(x) = x^{\sin x}$ on the interval $[0, 1]$, we see that $0 \leq f(x) \leq 1$. Taking two random samples x_i and y_i from the uniform distribution on $[0, 1]$, can be interpreted as picking a random point in the unit-square $\{(x, y)|0 \leq x \leq 1,\ 0 \leq y \leq 1\}$. Repeating this N times, the variables $n_i = \mathbf{1}\{y_i \leq f(x_i)\}$ indicate whether the i-point lies below $f(x)$, or not. Then, the value

$$\tilde{a} = \frac{1}{N}\sum_{i=1}^{N} n_i, \tag{18.3}$$

estimates the area a.

In trying to obtain α by means of a so-called *Monte Carlo simulation* we should keep track of the accuracy of the obtained results. Since $\tilde{a}$ is obtained as a function of a number of realisations of random variables, $\tilde{a}$ is itself a realisation of a random variable (which we denote as $\tilde{A}$). The random variable $\tilde{A}$ is often called the *estimator*, whereas the realisation $\tilde{a}$ is called the *estimate*. The random variable $\tilde{A}$ should be defined such that it obeys a number of properties, otherwise the estimate $\tilde{a}$ cannot be guaranteed to be accurate:

- $\tilde{A}$ should be *unbiased*, meaning that $E[\tilde{A}] = a$;
- $\tilde{A}$ should be *consistent*, meaning that the more samples we take, the more accurate the estimate $\tilde{a}$ becomes.

We will come back to these properties in Section 18.5. From the simulation we can compute an estimate for the variance of the estimator $\tilde{A}$ as follows:

$$\tilde{\sigma}^2 = \frac{1}{N(N-1)} \sum_{i=1}^{N} (n_i - \tilde{a})^2 . \tag{18.4}$$

Note that this estimator should not be confused with the esimator for the variance of a single sample, which is N times larger; see also Section 18.5.2 and [231]. Now we can apply *Chebyshev's inequality*, which states that for any $\beta > 0$

$$\Pr\{|\tilde{A} - \tilde{a}| \geq \beta\} \leq \frac{\tilde{\sigma}^2}{\beta^2}. \tag{18.5}$$

In words, it states that the probability that $\tilde{A}$ deviates more than β from the estimated value $\tilde{a}$, is at most equal to the quotient of $\tilde{\sigma}^2$ and β. The smaller the allowed deviation is, the weaker the bound on the probability. Rewriting (18.5) by setting $\delta = 1 - \tilde{\sigma}^2/\beta^2$ and $\tilde{\sigma} = \sqrt{\tilde{\sigma}^2}$, we obtain

$$\Pr\{|\tilde{A} - \tilde{a}| \leq \frac{\tilde{\sigma}}{\sqrt{1-\delta}}\} \geq \delta. \tag{18.6}$$

This equation tells us that $\tilde{A}$ deviates at most $\tilde{\sigma}/\sqrt{1-\delta}$ from $\tilde{a}$, with a probability of at least δ. In this expression, we would like δ to be relatively large, e.g., 0.99. Then, $\sqrt{1-\delta} = 0.1$, so that $\Pr\{|\tilde{A} - \tilde{a}| \leq 10\tilde{\sigma}\} \geq 0.99$. In order to let this inequality have high significance, we must make sure that the term "$10\tilde{\sigma}$" is small. This can be accomplished by making many observations.

It is important to note that when there is an analytical solution available for a particular problem, this analytical solution typically gives far more insight than the numerical answers obtained from a simulation. Individual simulation results only give information about a particular solution to a problem, and not at all over the range of possible solutions, nor do they give insight into the sensitivity of the solution to changes in one or more of the model parameters.

18.2 Classifying simulations

In this section we will classify simulations according to two criteria: their state space and their time evolution. Note that we used the same classification criteria when discussing stochastic processes in Chapter 3.

In *continuous-event simulations*, systems are studied in which the state continuously changes with time. Typically, these systems are physical processes that can be described by

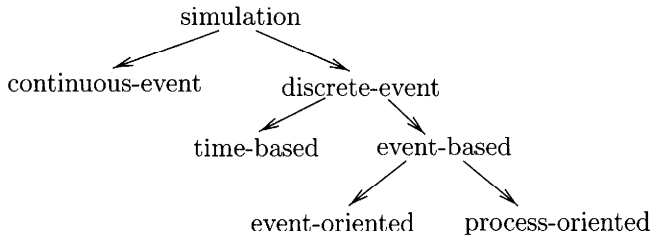

Figure 18.1: Classifying simulations

systems of differential equations with boundary conditions. The numerical solution of such a system of differential equations is sometimes called a simulation. In physical systems, time is a continuous parameter, although one can also observe systems at predefined time instances only, yielding a discrete time parameter. We do not further address continuous-state simulations, as we did not consider continuous-state stochastic processes.

More appropriate for our aims are *discrete-event simulations* (DES). In discrete-event simulations the state changes take place at discrete points in time. Again we can either take time as a continuous or as a discrete parameter. Depending on the application at hand, one of the two can be more or less suitable. In the discussions to follow we will assume that we deal with time as a continuous parameter.

In Figure 18.1 we show the discussed classification, together with some sub-classifications that follow below.

18.3 Implementation of discrete-event simulations

Before going into implementation details of discrete-event simulations, we first define some terminology in Section 18.3.1. We then present time-based simulations in Section 18.3.2 and event-based simulations in Section 18.3.3. We finally discuss implementation strategies for event-based discrete-event simulations in Section 18.3.4.

18.3.1 Terminology

The *simulation time* or *simulated time* of a simulation is the value of the parameter "time" that is used in the simulation program, which corresponds to the value of the time that would have been valid in the real system. The *run time* is the time it takes to execute a simulation program. Difference is often made between *wall-clock time* and *process time*;

the former includes any operating system overhead, whereas the latter includes only the required CPU, and possibly I/O time, for the simulation process.

In a discrete-event system, the state will change over time. The cause of a state variable change is called an *event*. Very often the state changes themselves are also called events. Since we consider simulations in which events take place one-by-one, that is, discrete in time, we speak of discrete-event simulations. In fact, it is because events in a discrete-event system happen one-by-one that discrete-event simulations are so much easier to handle than simulations of continuous-events systems. In discrete-event simulations we "jump" from event to event and it is the ordering of events and their relative timing we are interested in, because this exactly describes the performance of the simulated system. In a simulation program we will therefore mimic all the events. By keeping track of all these events and their timing, we are able to derive measures such as the average inter-event time or the average time between specific pairs of events. These then form the basis for the computation of performance estimates.

18.3.2 Time-based simulation

In a *time-based simulation* (also often called *synchronous simulation*) the main control loop of the simulation controls the time progress in constant steps. At the beginning of this control loop the time t is increased by a step Δt to $t + \Delta t$, with Δt small. Then it is checked whether any events have happened in the time interval $[t, t + \Delta t]$. If so, these events will be executed, that is, the state will be changed according to these events, before the next cycle of the loop starts. It is assumed that the ordering of the events within the interval $[t, t + \Delta t]$ is not of importance and that these events are independent. The number of events that happened in the interval $[t, t + \Delta t]$ may change from time to time. When t rises above some maximum, the simulation stops. In Figure 18.2 a diagram of the actions to be performed in a time-based simulation is given.

Time-based simulation is easy to implement. The implementation closely resembles the implementation of numerical methods for solving differential equations. However, there are some drawbacks associated with this method as well. Both the assumption that the ordering of events within an interval $[t, t + \Delta t]$ is not important and the assumption that these events are independent require that Δt be sufficiently small, in order to minimize the probability of occurrence of mutually dependent events. For this reason, we normally have to take Δt so small that the resulting simulation becomes very inefficient. Many very short time-steps will have to be performed without any event occurring at all. For these reasons time-based simulations are not often employed.

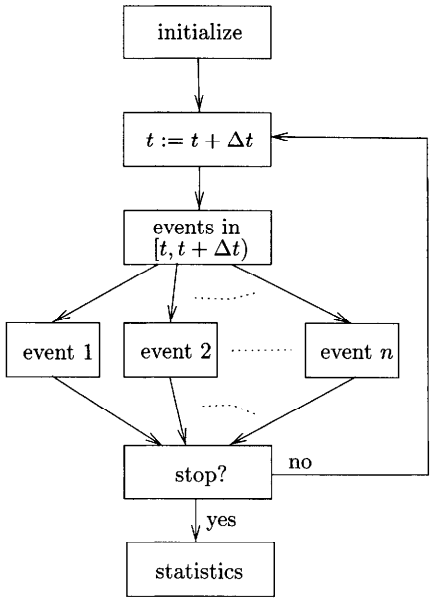

Figure 18.2: Diagram of the actions to be taken in a time-based simulation

Example 18.1. A time-based M|M|1 simulation program.
As an example, we present the framework of a time-based simulation program for an M|M|1 queue with arrival rate λ and service rate μ. In this program, we use two state variables: $N_s \in \{0, 1\}$ denoting the number of jobs in service, and $N_q \in \mathbb{N}$ denoting the number of jobs queued. Notice that there is a slight redundancy in these two variables since $N_q > 0 \Rightarrow N_s = 1$. The aim of the simulation program is to generate a list of time-instances and the state variables at these instances. The variable Δ is assumed to be sufficiently small. Furthermore, we have to make use of a function `draw(`p`)`, which evaluates to `true` with probability p and to `false` with probability $1 - p$; see also Section 18.4.

The resulting program is presented in Figure 18.3. After the initialisation (lines 1–3), the main program loop starts. First, the time is updated (line 6). If during the period $[t, t + \Delta t)$ an arrival has taken place, which happens with probability $\lambda \cdot \Delta t$ in a Poisson process with rate λ, we have to increase the number of jobs in the queue (line 7). Then, we check whether there is a job in service. If not, the just arrived job enters the server (lines 13–14). If there is already a job in service, we verify whether its service has ended in the last interval (line 9). If so, the counter N_s is set to 0 (line 12), unless there is another job waiting to be served (line 10); in that case a job is taken out of the queue and the server

```
1.  input(λ, μ, t_max)
2.  t := 0
3.  N_s := 0; N_q := 0
4.  while t < t_max
5.  do
6.      t := t + Δt
7.      if draw(λ · Δt) then N_q := N_q + 1
8.      if N_s = 1
9.      then if draw(μ · Δt)
10.          then if N_q > 0
11.               then N_q := N_q − 1
12.               else N_s := 0
13.     if N_s = 0 and N_q > 0
14.     then N_s := 1; N_q := N_q − 1
15.     writeln(t, N_q, N_s)
16. od
```

Figure 18.3: Pseudo-code for a time-based M|M|1 simulation

remains occupied (N_s does not need to be changed). □

18.3.3 Event-based simulation

In time-based simulations the time steps were of fixed length, but the number of events per time step varied. In *event-based simulations* (also often called *asynchronous simulation*) it is just the other way around. We then deal with time steps of varying length such that there is always exactly one event in every time step. So, the simulation is controlled by the occurrence of "next events". This is very efficient since the time steps are now just long enough to optimally proceed with the simulation and just short enough to exclude the possibility of having more than one event per time step, thus circumventing the problems of handling dependent events in one time step.

Whenever an event occurs this causes new events to occur in the future. Consider for instance the arrival of a job at a queue. This event causes the system state to change, but will also cause at least one event in the future, namely the event that the job is taken

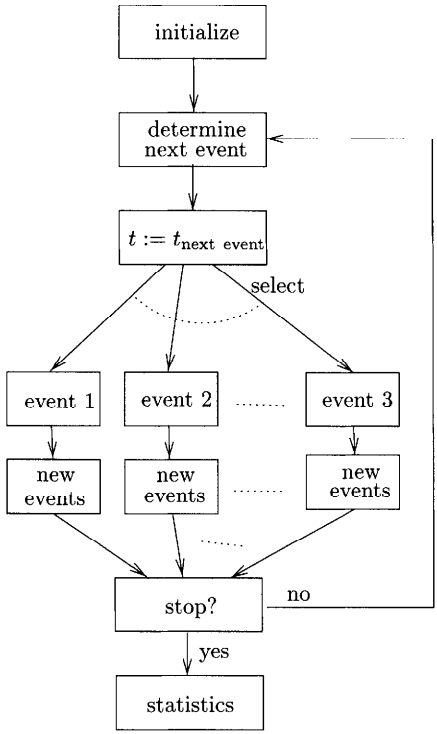

Figure 18.4: Diagram of the actions to be taken in a event-based simulation

into service. All future events are generally gathered in an *ordered event list.* The head of this list contains the next event to occur and its occurrence time. The tail of this list contains the future events, in their occurrence order. Whenever the first event is simulated (processed), it is taken from the list and the simulation time is updated accordingly. In the simulation of this event, new events may be created. These new events are inserted in the event list at the appropriate places. After that, the new head of the event list is processed. In Figure 18.4 we show a diagram of the actions to be performed in such a simulation.

Most of the discrete-event simulations performed in the field of computer and communication performance evaluation are of the event-based type. The only limitation to event-based simulation is that one must be able to compute the time instances at which future events take place. This is not always possible, e.g., if very complicated delay distributions are used, or if the system is a continuous-variable dynamic system. In those cases, time-based simulations may be preferred. Also when simulating at a very fine time-

```
1.  input(λ, μ, t_max)
2.  t := 0
3.  N_s := 0; N_q := 0
4.  while t < t_max
5.  do
6.      if N_s = 1
7.      then narr := negexp(λ)
8.           ndep := negexp(μ)
9.           if ndep < narr
10.          then t := t + ndep
11.              if N_q > 0
12.              then N_q := N_q − 1
13.              else N_s := 0
13.          else t := t + narr
14.              N_q := N_q + 1
15.     else narr := negexp(λ)
16.          t := t+ narr
17.          N_s := 1
18.     writeln( t, N_q, N_s)
19. od
```

Figure 18.5: Pseudo-code for an event-based M|M|1 simulation

granularity, time-based simulations are often used, e.g., when simulating the execution of microprocessor instructions. In such cases, the time-steps will resemble the processor clock-cycles and the microprocessor should have been designed such that dependent events within a clock-cycle do not exist. We will only address event-based simulations from now on.

Example 18.2. An event-based M|M|1 simulation program.
We now present the framework of an event-based simulation program for the M|M|1 queue we addressed before. We again use two state variables: $N_s \in \{0, 1\}$ denoting the number of jobs in service, and $N_q \in I\!N$ denoting the number of jobs queued. We furthermore use two variables that represent the possible next events: `narr` denotes the time of the next arrival and `ndep` denotes the time of the next departure. Since there are at most two possible

next events, we can store them in just two variables (instead of in a list). The aim of the simulation program is to generate a list of events times, and the state variables at these instances. We have to make use of a function `negexp(`λ`)` which generates a realisation of a random variable with negative exponential distribution with rate λ; see also Section 18.4.

The resulting program is presented in Figure 18.5. After the initialisation (lines 1–3), the main program loop starts. Using the variable N_s, it is decided what the possible next events are (line 6). If there is no job being processed, the only possible next event is an arrival: the time until this next event is generated, the simulation time is updated accordingly and the state variable N_s increased by 1 (lines 15–17). If there is a job being processed, then two possible next events exist. The times for these two events are computed (lines 7–8) and the one which occurs first is performed (decision in line 9). If the departure takes place first, the simulation time is adjusted accordingly, and if there are jobs queued, one of them is taken into service. Otherwise, the queue and server remain empty (lines 10–13). If the arrival takes place first, the time is updated accordingly and the queue is enlarged by 1 (lines 13–14). □

18.3.4 Implementation strategies

Having chosen the event-based approach towards simulation, there exist different implementation forms. The implementation can either be *event-oriented* or *process-oriented.*

With the event-oriented implementation there is a procedure P_i defined for every type of event i that can occur. In the simulator an event list is defined. After initialisation of this event list the main control loop starts, consisting of the following steps. The first event to happen is taken from the list. The simulation time is incremented to the value at which this (current) event occurred. Then, if this event is of type i, the procedure P_i is invoked. In this procedure the simulated system state is changed according to the occurrence of event i, and new events are generated and inserted in the event list at the appropriate places. After procedure P_i terminates, some statistics may be collected, and the main control loop is continued. Typically employed stopping criteria include the simulated time, the number of processed events, the amount of used processing time, or the width of the confidence intervals that are computed for the measures of interest (see Section 18.5).

In an event-oriented implementation, the management of the events is explicitly visible. In a process-oriented implementation, on the other hand, a process is associated with every event-type. These processes exchange information to communicate state changes to one another, e.g., via explicit message passing or via shared variables. The simulated system

operation can be seen as an execution path of the communicating event-processes. The scheduling of the events in the simulation is done implicitly in the scheduling of the event-processes. The latter can be done by the operating system or by the language run-time system. A prerequisite for this approach is that language elements for parallel programming are provided.

Both implementation strategies are used extensively. For the event-oriented implementation normal programming languages such as Pascal or C are used. For the process-oriented implementation, Simula'67 has been used widely for a long period; however, currently the use of C^{++} in combination with public domain simulation classes is most common.

Instead of explicitly coding a simulation, there are many software packages available (both public domain and commercial) that serve to construct and execute simulations in an efficient way. Internally, these packages employ one of the two methods discussed above; however, to the user they represent themselves in a more application-oriented way, thus hiding most of the details of the actual simulation (see also Section 1.5 on the GMTF). A number of commercial software packages, using graphical interfaces, for the simulation of computer-communication systems have recently been discussed by Law and McComas [176]; with these tools, the simulations are described as block-diagrams representing the system components and their interactions. A different approach is taken with (graphical) simulation tools based on queueing networks and stochastic Petri nets. With such tools, the formalisms we have discussed for analytical and numerical performance evaluations are extended so that they can be interpreted as simulation specifications. The tools then automatically transform these specifications to executable simulation programs and present the results of these simulations in a tabular or graphical format. Of course, restrictions that apply for the analytic and numerical solutions of these models do not apply any more when the simulative solution is used. For more information, we refer to the literature, e.g., [125].

18.4 Random number generation

In order to simulate performance models of computer-communication systems using a computer program we have to be able to generate random numbers from certain probability distributions, as we have already seen in the examples in the previous section. Random number generation (RNG) is a difficult but important task; when the generated random numbers do not conform to the required distribution, the results obtained from the simu-

lation should at least be regarded with suspicion.

To start with, *true* random numbers cannot be generated with a deterministic algorithm. This means that when using computers for RNG, we have to be satisfied with *pseudo-random numbers*. To generate pseudo-random numbers from a given distribution, we proceed in three steps. We first generate a series of pseudo-random numbers on a finite subset of $\mathbb{N}$, normally $\{0, \cdots, m-1\}$, $m \in \mathbb{N}$. This is discussed in Section 18.4.1. From this pseudo-random series, we compute (pseudo) uniformly distributed random numbers. To verify whether these pseudo-random numbers can be regarded as true random numbers we have to employ a number of statistical tests. These are discussed in Section 18.4.2. Using the uniform distributed random variables, various methods exist to compute non-uniform pseudo-random variables. These are discussed in Section 18.4.3.

18.4.1 Generating pseudo-random numbers

The generation of sequences of pseudo-random numbers is a challenging task. Although many methods exist for generating such sequences, we restrict ourselves here to the so-called *linear* and *additive congruential methods*, since these methods are relatively easy to implement and most commonly used. An RNG can be classified as good when:

- successive pseudo-random numbers can be computed with little cost;
- the generated sequence appears as truly random, i.e., successive pseudo-random numbers are independent from and uncorrelated with one another and conform to the desired distribution;
- its period (the time after which it repeats itself) is very long.

Below, we will present two RNGs and comment on the degree of fulfillment of these properties.

The basic idea of linear congruential RNGs is simple. Starting with a value z_0, the so-called *seed*, z_{i+1} is computed from z_i as follows:

$$z_{i+1} = (az_i + c) \text{ modulo } m. \tag{18.7}$$

With the right choice of parameters a, c, and m, this algorithm will generate m different values, after which it starts anew. The number m is called the *cycle length*. Since the next value of the series only depends on the current value, the cycle starts anew whenever a value reappears. The linear congruential RNG will generate a cycle of length m if the following three conditions hold:

- the values m and c are relative primes, i.e., their greatest common divisor is 1;
- all prime factors of m should divide $a - 1$;
- if 4 divides m, then 4 should also divide $a - 1$.

These conditions only state something about the cycle length; they do not imply that the resulting cycle appears as truly random.

Example 18.3. Linear congruential method.
Consider the case when $m = 16$, $c = 7$, and $a = 5$. We can easily check the conditions above. Starting with $z_0 = 0$, we obtain $z_1 = (5 \times 0 + 7)$ modulo $16 = 7$. Continuing in this way we obtain: $0, 7, 10, 9, 4, 11, 14, \cdots$. □

The main problem with linear congruential methods is that the cycles are relatively short, hence, there is too much repetition, too little randomness. This problem is avoided by using additive congruential methods. With these methods, the i-value z_i is derived from the k previous values $(z_{i-1}, \cdots, z_{i-k})$ in the following way:

$$z_i = \left(\sum_{j=1}^{k} a_j z_{i-j} \right) \text{ modulo } m. \tag{18.8}$$

The starting values z_0 through z_{k-1} are generally derived by a linear congruential method, or by assuming $z_l = 0$, for $l < 0$. With an appropriate selection of the factors a_j cycles of length $m^k - 1$ are obtained.

Example 18.4. Additive congruential method.
Choosing the value $k = 7$ and setting the coefficients $a_1 = a_7 = 1$ and $a_2 = \cdots = a_6 = 0$, we can extend the previous example. As starting sequence we take the first 7 terms computed before: $0, 7, 10, 9, 4, 11, 14$. The next value would then be $(14 + 0)$ modulo $16 = 14$. Continuing in this way we obtain: $0, 7, 10, 9, 4, 11, 14, 14, 5, 15, 8, 12, 7, 5, \cdots$. Observe that when a number reappears, this does not mean that the cycle restarts. For this example, the cycle length is limited by $16^7 - 1 = 268435455$. □

Finally, it is advisable to use a different RNG for each random number sequence to be used in the simulation, otherwise undesired dependencies between random variables can be introduced. Also, a proper choice of the seed is of importance. There are good RNGs that do not function properly, or not optimally, with wrongly chosen seeds. To be able to reproduce simulation experiments, it is necessary to control the seed selection process; taking a random number as seed is therefore not a good idea.

18.4.2 Testing pseudo-uniformly distributed random numbers

With the methods of Section 18.4.1 we are able to generate pseudo-random sequences. Since the largest number that is obtained is $m-1$, we can simply divide the successive z_i values by $m-1$ to obtain a sequence of values $u_i = z_i/(m-1)$. It is then assumed that these values are pseudo-uniformly distributed.

Before we proceed to compute random numbers obeying other distributions, it is now time to verify whether the generated sequence of pseudo-uniform random numbers can indeed be viewed as a realisation sequence of the uniform distribution.

Testing the uniform distribution with the χ^2-test

We apply the χ^2-test to decide whether a sequence of n random numbers $x_1, \cdots, x_n$ obeys the uniform distribution on $[0,1]$. For this purpose, we divide the interval $[0,1]$ in k intervals $I_i = [(i-1)/k, i/k]$, that is, I_i is the i-th interval of length $1/k$ in $[0,1]$ starting from the left, $i = 1, \cdots, k$. We now compute the number n_i of generated random numbers in the i-th interval:

$$n_i = |\{x_j | x_j \in I_i, j = 1, \cdots, n\}|. \tag{18.9}$$

We would expect all values n_i to be close to n/k. We now define as quality criterion for the RNG the relative squared difference of the values n_i and their expectation [289]:

$$d = \frac{\sum_{i=1}^{k} \left(n_i - \frac{n}{k}\right)^2}{n/k}. \tag{18.10}$$

The value d is a realisation of a stochastic variable D which has approximately a χ^2-distribution with $k-1$ degrees of freedom. The hypothesis that the generated numbers do come from the uniform distribution on $[0,1]$ cannot be rejected with probability α, if d is smaller than the critical value for $\chi^2_{\alpha,k-1}$, according to Table 18.1.

A few remarks are in order here. For the χ^2-test to be valuable, we should have a large number of intervals k, and the number of random numbers in each interval should not be too small. Typically, one would require $k \geq 10$ and $n_i \geq 5$.

The χ^2-test employs a discretisation to test the generated pseudo-random sequence. Alternatively, one could use the Kolmogorov-Smirnov test to directly test the real numbers generated. As quality measure, this test uses the maximum difference between the desired CDF and the observed CDF; for more details, see e.g., [137, 145].

k	$\alpha = 0.9$	$\alpha = 0.95$	$\alpha = 0.99$
2	4.605	5.991	9.210
3	6.253	7.817	11.356
4	7.779	9.488	13.277
5	9.236	11.071	15.086
6	10.645	12.592	16.812
7	12.017	14.067	18.475
8	13.362	15.507	20.090
9	14.684	16.919	21.666
10	15.987	18.307	23.209
15	22.307	24.996	30.578
20	28.412	31.410	37.566
25	34.382	37.653	44.314
30	40.256	43.773	50.892
40	51.805	55.759	63.691
50	63.17	67.505	76.154
60	74.40	79.082	88.379
70	85.53	90.531	100.425
80	96.58	101.879	112.329
90	107.6	113.145	124.116
100	118.5	124.342	135.807
$k > 100$	$\frac{1}{2}(h + 1.28)^2$	$\frac{1}{2}(h + 1.64)^2$	$\frac{1}{2}(h + 2.33)^2$

Table 18.1: Critical values $x_{k,\alpha}$ for the χ^2-distribution with k degrees of freedom and confidence level α such that $\Pr\{D \leq x_{k,\alpha}\} = \alpha$ (with $h = \sqrt{2k-1}$)

Testing the correlation structure

In order to test whether successive random numbers can be considered independent from one another, we have to study the correlation between the successive pseudo-random numbers. Let the random numbers $x_1, \cdots, x_n$ be generated uniform numbers on $[0, 1]$. The *auto-correlation coefficient with lag* $k \geq 1$ is then estimated by:

$$C_k = \frac{1}{n-k} \sum_{i=1}^{n-k} \left(X_i - \tfrac{1}{2}\right)\left(X_{i+k} - \tfrac{1}{2}\right). \tag{18.11}$$

Since C_k is the sum of a large number of identically distributed random variables, it has a Normal distribution, here with mean 0 and variance $(144(n-k))^{-1}$. Therefore, the random

α	0.9	0.95	0.99
z	1.645	1.960	2.576

Table 18.2: Critical values z for the $N(0,1)$-distribution and confidence level α such that $\Pr\{|Z| \leq z\} = \alpha$

variable $A_k = 12C_k\sqrt{n-k}$ is $N(0,1)$-distributed. Hence, we can determine the value z such that

$$\Pr\{C_k \leq z\} = \Pr\{|A_k| \leq z/12\sqrt{n-k}\} = \alpha, \tag{18.12}$$

by using Table 18.2. Thus, for a chosen confidence level α, the autocorrelation coefficient at lag k will lie in the interval $[c_k - z/12\sqrt{n-k}, c_k + z/12\sqrt{n-k}]$, where c_k is a realisation of C_k. For a proper uniform RNG, the auto-correlation coefficients should be very close to 0, that is, the computed confidence intervals should contain 0.

18.4.3 Generation of non-uniformly distributed random numbers

There are various techniques to use uniform random numbers to obtain differently distributed random numbers that obey other distributions. We present some of these techniques below.

The inversion method

Consider the distribution function $F_Y(y)$ of some stochastic variable Y. Let Z be a random variable defined as a function of the random variable Y, and let us choose as function a very special one, namely the distribution function of Y: i.e., $Z = F_Y(Y)$. The distribution function of Z, i.e., $F_Z(z)$, has the following form:

$$F_Z(z) = \Pr\{Z \leq z\} = \Pr\{F_Y(Y) \leq z\}. \tag{18.13}$$

Now, assuming that F_Y can be inverted, we can equate the latter probability with $\Pr\{Y \leq F_Y^{-1}(z)\}$, for $0 \leq z \leq 1$. But, since $F_Y(y) = \Pr\{Y \leq y\}$ we find, after having substituted $F_Y^{-1}(z)$ for y:

$$F_Z(z) = F_Y(F_Y^{-1}(z)) = z, \quad \text{for } 0 \leq z \leq 1. \tag{18.14}$$

In conclusion, we find that Z is distributed uniformly on $[0,1]$. To generate random numbers with a distribution $F_Y(y)$ we now proceed as follows. We generate a uniformly distributed random number z and apply the inverse function to yield $y = F_Y^{-1}(z)$. The

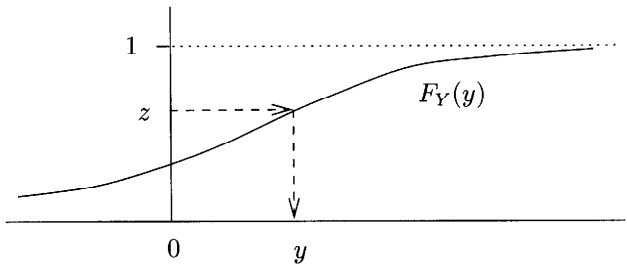

Figure 18.6: Deriving a continuous random variable from a uniformly distributed random variable

realisations y are then distributed according to distribution function F_Y. In Figure 18.6 we visualise the inversion approach.

Example 18.5. Negative exponential random numbers.
To generate random numbers from the exponential distribution $F_Y(y) = 1 - e^{-\lambda y}$, $y > 0$, we proceed as follows. We solve $z = F_Y(y)$ for y to find: $y = -\ln(1-z)/\lambda$. Thus, we can generate uniformly distributed numbers z, and apply the just derived equation to obtain exponentially distributed random numbers y. To save one arithmetic operation, we can change the term $1 - z$ to z, since if z is uniformly distributed, then $1 - z$ is so as well. □

Example 18.6. Erlang-k random numbers.
To generate random numbers from the Erlang-k distribution, we generate k random numbers, distributed according to a negative exponential distribution, and simply add these. In order to avoid having to take k logarithms, we can consider the following. Let $u_1, \cdots, u_k$ be k uniformly distributed random numbers, and let $x_i = -\ln(u_i)/\lambda$ be the corresponding k negative exponential distributed random numbers (λ is the rate per phase). We now compute the Erlang-k distributed number z as follows:

$$z = \sum_{i=1}^{k} x_i = -\frac{1}{\lambda} \sum_{i=1}^{k} \ln(u_i) = -\frac{1}{\lambda} \ln\left(\prod_{i=1}^{k} u_i\right). \tag{18.15}$$

In conclusion, we simply have to multiply the k terms, and have to take only one natural logarithm. Since RNGs are invoked very often during a simulation, such efficiency improvements are very important. □

Example 18.7. Hyper-exponential random numbers.
The hyper-exponential distribution can be interpreted as a choice between n negative expo-

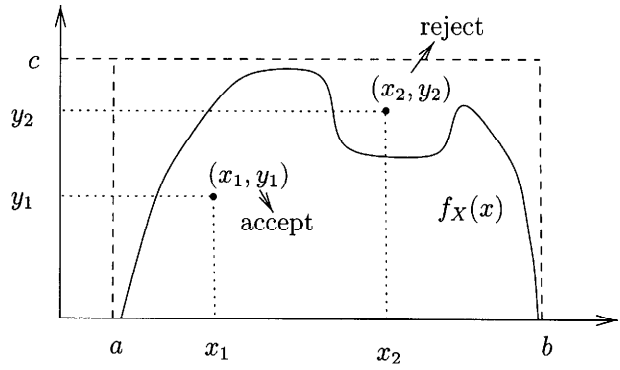

Figure 18.7: Rejection method for generating random variables with density $fX(x)$

nential distributions, each with rate λ_i. We therefore first generate a random integer i from the set $\{1, \cdots, n\}$; this is the selection phase. We then generate a negative exponentially distributed random number with rate λ_i. □

The rejection method

For obtaining random variables for which the inverse distribution function cannot be easily obtained, we can use the *rejection method*, provided we know the density function $f_X(x)$. Furthermore, the density function must have a finite domain, say $[a, b]$, as well as a finite image on $[a, b]$, say $[0, c]$. If there are values $x \notin [a, b]$ for which $f_X(x) > 0$, then the rejection method only provides approximate random numbers. In Figure 18.7 we show a density function fulfilling the requirements. We proceed as follows. We generate two uniformly distributed numbers u_1 and u_2 on $[0, 1]$ and derive the random numbers $x = a + (b - a)u_1$ and $y = cu_2$. The tuple (x, y) is a randomly selected point in the rectangle $[a, b] \times [0, c]$. Now, whenever $y < f_X(x)$, that is, whenever the point (x, y) lies below the density $f_X(x)$, we accept x. Successive values for x then obey the density $f_X(x)$. Whenever $y \geq f_X(x)$ we repeat the procedure until we encounter a tuple for which the condition holds. This procedure is fairly efficient when the area under the density $f_X(x)$ is close to $c(b - a)$. In that case we have a relatively high probability that a sample point lies under the density, so that we do not need many sample points.

The proof of the rejection method is fairly simple. Consider two random variables: X is distributed uniformly on $[a, b]$, and Y is distributed uniformly on $[0, c]$. We then proceed to compute the following conditional probability, which exactly equals the probability

distribution function for an accepted value x according to the rejection scheme:

$$\Pr\{x \leq X \leq x + dx | Y \leq f_X(X)\} = \frac{\Pr\{x \leq X \leq x + dx, Y \leq f_X(x)\}}{\Pr\{Y \leq f_X(X)\}}$$

$$= \left(\frac{dx}{b-a}\right)\left(\frac{f_X(x)}{c}\right)\left(\frac{1}{(b-a)c}\right)^{-1} = f_X(x)dx.$$

We see that the conditional probability reduces to the required probability density.

Normally distributed random numbers

For some random variables, the distribution function cannot be explicitly computed or inverted, nor does the density function have a finite domain. For these cases we have to come up with even other methods to generate random numbers.

As a most interesting example of these, we address the normal distribution. We apply the central limit theorem to compute normally distributed random numbers as follows. We first generate n independent and identically distributed random numbers $x_1, \cdots, x_n$, which can be seen as realisations of the random variables $X_1, \cdots, X_n$, which are all distributed as the random variable X with mean $E[X]$ and variance $\mathrm{var}[X]$. We can then define the random variable $S_n = X_1 + \cdots + X_n$. The central limit theorem then states that the random variable

$$N = \frac{S_n - nE[X]}{\sqrt{n\mathrm{var}[X]}}$$

approaches a normally distributed random variable with mean 0 and variance 1, i.e., an $N(0,1)$ distribution.

Now, by choosing the uniform distribution on $[0,1]$ for X (X has mean $E[X] = 1/2$ and $\mathrm{var}[X] = 1/12$) and taking $n = 12$ samples, $S_{12} = X_1 + \cdots + X_{12}$, so that

$$N = \frac{S_n - nE[X]}{\sqrt{n\mathrm{var}[X]}} = \frac{S_n - 6}{\sqrt{12\frac{1}{12}}} = S_{12} - 6 \tag{18.16}$$

approaches a $N(0,1)$-distributed random variable. An advantage of using N is that it is very efficient to compute. Of course, taking larger values for n increases the accuracy of the generated random numbers.

18.5 Statistical evaluation of simulation results

A discrete-event simulation is performed to obtain quantitative insight into the operation of the modelled system. When executing a simulation (program), relevant events can be

time-stamped. All the resulting samples (or observations) can be written to a *trace* or *log file*. In Section 18.5.1 we discuss how we can obtain single samples from a simulation execution. Although a complete log file contains all the available information, it is generally of little practical use. Therefore, the simulation log is "condensed" to a format that is more suitable for human interpretation. This step is performed using statistical techniques and is discussed in Section 18.5.2.

18.5.1 Obtaining measurements

We address the issues of obtaining individual samples and the removal of invalid samples (the initial transient) below.

Sampling individual events

We distinguish two types of measures that can be obtained from a simulation: user-oriented measures and system-oriented measures. *User-oriented measures* are typically obtained by monitoring specific users of the system under study, i.e., by monitoring individual jobs. An example of a user-oriented measure is the job residence time in some part of the modelled system. When the i-th job enters that system part, a time-stamp $t_i^{(a)}$ ("a" for arrival) is taken. When the job leaves the system part a time-stamp $t_i^{(d)}$ ("d" for departure) is taken. The difference $t_i = t_i^{(d)} - t_i^{(a)}$ is an realisation of the job residence time. By summing over all simulated jobs, denoted as n, we finally obtain an estimate for the mean job residence time as:

$$\tilde{r} = \frac{1}{n}\sum_{i=1}^{n}(t_i^{(d)} - t_i^{(a)}) = \frac{1}{n}\left(\sum_{i=1}^{n} t_i^{(d)} - \sum_{i=1}^{n} t_i^{(a)}\right). \tag{18.17}$$

Notice that during the simulation, we do not have to store all the individual time-stamp values, since in the end we only need the difference of their sums.

For the derivation of *system-oriented measures* no individual jobs should be monitored, but the system state itself. A typical example is the case where the measure of interest is the long-run probability that a finite buffer is fully occupied. Upon every state change in the model the system state of interest is checked for this condition. If this condition becomes true, a time-stamp $t_i^{(f)}$ is taken ("f" for full). The next time-stamp, denoted $t_i^{(n)}$ ("n" for not full) is then taken when the buffer is not fully occupied anymore. The difference $t_i^{(n)} - t_i^{(f)}$ is a realisation of random variable that could be called the "buffer-full period". The sum of all these periods, divided by the total simulation time then estimates

the long-run buffer full probability, i.e.,

$$\tilde{b} = \frac{1}{T}\sum_{i=1}^{n}(t_i^{(n)} - t_i^{(f)}) = \frac{1}{T}\left(\sum_{i=1}^{n} t_i^{(n)} - \sum_{i=1}^{n} t_i^{(f)}\right), \tag{18.18}$$

where we assume that during the simulation of length T we have experienced n buffer full periods. Notice that the simulation time T is predetermined for system-oriented measures, whereas the number of samples n is prespecified for user-oriented measures, in order to obtain unbiased estimators.

Initial transient removal

With most simulations, we try to obtain steady-state performance measures. However, when starting the simulation, the modelled system state will generally be very non-typical, e.g., in queueing network simulations one may choose all queues to be initially empty. Hence, the observations made during the first period of the simulation will be non-typical as well, thus influencing the simulation results in an inappropriate way. Therefore, we should ignore the measurements taken during this so-called *(initial) transient period.* The main question then is: how long should we ignore the measurements, or, in other words, how long should we take this initial transient? This question is not at all easy to answer; Pawlikowski discusses 11 "rules" to recognise the initial transient period [231]. Below, we briefly discuss a number of simple guidelines.

The first guideline is to simulate so long that the effect of the initial transient period becomes negligible. Of course, this is not an efficient method, nor does it provide any evidence. This method becomes slightly better when combined with a smart initial state in the simulation, e.g., based on an analytic queueing network model of a simplified version of the simulation model. In doing so, the period in which queues have to built up towards their mean occupation becomes smaller. But this method does not provide evidence for a particular choice.

The slightly better guideline is given by the *truncation method*, which removes the first $l < n$ samples from a sequence of samples $x_1, \cdots, x_n$, where l is the smallest value such that:

$$\min\{x_{l+1}, \cdots, x_n\} \neq x_{l+1} \neq \max\{x_{l+1}, \cdots, x_n\}. \tag{18.19}$$

In words: the samples $x_1, \cdots, x_l$ are removed when the $(l+1)$-th sample is no longer the maximum, nor the minimum of the remaining samples. This means that the samples that follow x_{l+1} have values both smaller and larger than x_{l+1}, thus indicating that an oscillation around a stationary situation has started. This method corresponds to rule R1 in [231] and

has been shown to overestimate the length of the initial transient period when simulating systems under low utilisation, and to underestimate it for high utilisation.

A better but still simple guideline is the following, based on an estimation of the variance. Consider a sequence of n samples. The sample mean m is computed as $(\sum_{i=1}^{n} x_i)/n$. Then, we split the n observations into k groups or batches, such that $k = \lfloor n/l \rfloor$. We start with batch size $l = 2$ and increase it stepwisely, thereby computing k accordingly, until the sample variance starts to decrease, as follows. We compute the batch means as

$$m_i = \frac{1}{l} \sum_{j=1}^{l} x_{(i-1)l+j}, \quad i = 1, \cdots, k, \tag{18.20}$$

and the sample variance as:

$$\sigma^2 = \frac{1}{k-1} \sum_{i=1}^{k} (m_i - m)^2 . \tag{18.21}$$

By increasing the batch size l, more and more samples of the initial transient period will become part of the first batch. If l is small, many batches will contain samples from the initial transient period, thus making σ^2 larger, also for increasing l. However, if l becomes so large that the first batch contains almost exclusively the samples that can be considered part of the initial transient period, only m_1 will significantly differ from m, thus making σ^2 smaller. The batch size l for which σ^2 starts to decrease monotonously equals the number of samples that should not be considered any further (see also [145]).

18.5.2 Mean values and confidence intervals

Suppose we are executing a simulation to estimate the mean of the random variable X with (unknown) $E[X] = a$. In doing so, a simulation is used to generate n samples x_i, $i = 1, \cdots, n$, each of which can be seen as a realisation of a stochastic variable X_i. The simulation has been constructed such that the stochastic variables X_i are all distributed as the random variable X. Furthermore, to compute confidence intervals, we require the X_i to be independent of each other.

Below, we discuss how to estimate the mean value of X and present means to compute confidence intervals for the obtained estimate. We also comment on ways to handle independence in the measurements.

Mean values

To estimate $E[X]$, we define a new stochastic variable, $\tilde{X}$, which is called *an estimator* of X. Whenever $E[\tilde{X}] = a$, the estimator $\tilde{X}$ is called *unbiased*. Whenever $\Pr\{|\tilde{X} - a| < \epsilon\} \to 0$

when $n \to \infty$, the estimator $\tilde{X}$ is called *consistent*. The latter condition translates itself to the requirement that $\text{var}[\tilde{X}] \to 0$, whenever the number of samples $n \to \infty$. Clearly, both unbiasedness and consistency are desirable properties of estimators. Whenever the observations $X_1, \cdots, X_n$ are independent, then

$$\tilde{X} = \frac{1}{n} \sum_{i=1}^{n} X_i \tag{18.22}$$

is an unbiased and consistent estimator for $E[X]$ since $E[\tilde{X}] = E[X]$ (unbiasedness) and $\text{var}[\tilde{X}] = \text{var}[X]/n$, so that $\text{var}[\tilde{X}] \to 0$, when $n \to \infty$ (consistence).

Although the estimator $\tilde{X}$ seems to be fine, the requirement that the random variables X_i should be independent causes us problems (independence is required for consistency, not for unbiasedness). Indeed, successive samples taken from a simulation are generally not independent. For instance, suppose that the random variables X_i signify samples of job response times in a particular queue. Whenever X_i is large (or small), X_{i+1} will most probably be large (or small) as well, so that successive samples are dependent. There are a number of ways to cope with the dependence between successive samples; these will be discussed below.

Guaranteeing independence

There are a number of ways to obtain almost independent observations from simulations, as they are required to compute confidence intervals. We discuss the three most-widely used methods below.

With the method of *independent replicas* the simulation execution is replicated n times, each time with a different seed for the RNG. In the i-th simulation run, the samples $x_{i,1}, \cdots, x_{i,m}$ are taken. Although these individual samples are not independent, the sample means $x_i = (\sum_{j=1}^{m} x_{i,j})/m$, $i = 1, \cdots, n$, are considered to be independent from one another. These n mean values are therefore considered as the samples from which the overall mean value and confidence interval (see below) are estimated. The advantage of this method lies in the fact that the used samples are really independent, provided the RNGs deliver independent streams. A disadvantage is that the simulation has to be executed from start several times. This implies that also the transient behaviour at the beginning of every simulation has to performed and removed multiple times.

A method which circumvents the latter problem is the *batch means method*. It requires only a single simulation run from which the samples $x_1, \cdots, x_{n \cdot m}$ are split into n batches

of size m each. Within every batch the samples are averaged as follows:

$$y_i = \frac{1}{m} \sum_{j=1}^{m} x_{(i-1)m+j}. \tag{18.23}$$

The samples $y_1, \cdots, y_n$ are assumed to be independent and used to compute the overall average and confidence intervals. The advantage of this method is that only a single simulation run is performed for which the initial transient has to be removed only once. A disadvantage of this method is the fact that the batches are not totally independent, hence, this method is only approximate. In practice, however, the batch-means method is most often used.

The fact that successive batches are not totally independent is overcome with the so-called *regenerative method*. With this method a single simulation execution is split into several batches as well; however, the splitting is done at so-called regeneration points in the simulation. Regeneration points are defined such that the behaviour before such a point is totally independent from the behaviour after it. A good example of a regeneration point is the moment the queue gets empty in the simulation of an M|G|1 queue. As before, the batch averages are taken as the samples to compute the overall mean and the confidence intervals; however, since the number of samples per batch is not constant, more complex, so-called ratio-estimators, must be used.

The advantage of the regeneration method is that the employed samples to compute the confidence intervals are really independent. The problems with it are the more complex estimators and the fact that regeneration points might occur only very rarely, thus yielding long simulation run times. True regeneration points that are visited frequently during a simulation are rare; the state in which the modelled system empties is often considered a regeneration point, but this is only correct when the times until the next events are drawn from memoryless distributions. To illustrate this, in an M|M|1 simulation, every arrival and departure point is a regeneration point. In an M|G|1 simulation, all departure instances and the first arrival instance are regeneration points. In contrast, in an G|G|1 simulation, only the first arrival instance is a regeneration point.

Confidence intervals

Since the random variables X_i are assumed to be independent and identically distributed, the estimator $\tilde{X}$ as defined in (18.22) will, according to the central limit theorem, approximately have a Normal distribution with mean a and variance σ^2/n (we denote var$[X]$ as

σ^2). This implies that the random variable

$$Z' = \frac{\tilde{X} - a}{\sigma/\sqrt{n}} \tag{18.24}$$

is $N(0,1)$-distributed. However, since we do not know the variance of the random variable X, we have to estimate it as well. An unbiased estimator for σ^2, known as the sample variance, is given as:

$$S^2 = \frac{1}{n-1} \sum_{i=1}^{n} \left(X_i - \tilde{X}\right)^2, \tag{18.25}$$

The stochastic variable

$$Z = \frac{\tilde{X} - a}{\tilde{S}/\sqrt{n}} \tag{18.26}$$

then has a Student- or t-distribution with $n-1$ degrees of freedom (a t_{n-1}-distribution). Notice that a and $\tilde{\sigma}^2$ can easily be computed when $\sum_i x_i$ and $\sum_i x_i^2$ are known:

$$a = \frac{\sum_{i=1}^{n} x_i}{n}, \quad \text{and} \quad \tilde{\sigma}^2 = \frac{\sum_{i=1}^{n} x_i^2}{n-1} - \frac{\left(\sum_{i=1}^{n} x_i\right)}{n(n-1)}, \tag{18.27}$$

so that during the simulation, only two real numbers have to be maintained per measure.

The Student distribution with three or more degrees of freedom is a symmetric bell-shaped distribution, similar in form to the Normal distribution (see Figure 18.8). For $n \to \infty$, the t_n-distribution approaches an $N(0,1)$ distribution. By using a standard table for it, such as given in Table 18.3, we can find the value $z > 0$ such that $\Pr\{|Z| \leq z\} = \beta$; z is called the *double-sided critical value* for the t_n-distribution, given β. The last row in Table 18.3 corresponds to the case of having an unbounded number of degrees of freedom, that is, these critical values follow from the $N(0,1)$ distribution. Using these double-sided critical values, we can write:

$$\Pr\{|Z| \leq z\} = \Pr\left\{\frac{|\tilde{X} - a|}{\sigma/\sqrt{n}} \leq z\right\} = \Pr\{|\tilde{X} - a| \leq z\sigma/\sqrt{n}\} = \beta, \tag{18.28}$$

which states that the probability that the estimator $\tilde{X}$ deviates less than $z\sigma/\sqrt{n}$ from the mean a is β. Stated differently, the probability that X lies in the so-called *confidence interval* $[a - z\sigma/\sqrt{n}, a + z\sigma/\sqrt{n}]$ is β. The probability β is called the confidence level. As can be observed, to make the confidence interval a factor l smaller, l^2 times more observations are required, so that the simulation needs to be l^2 times as long.

Note that many statistical tables present *single-sided critical values*, i.e., those values z' for which the probability $\Pr\{Z \leq z'\} = \beta'$. Due to the symmetrical nature of the t_n-distribution, for $n \geq 3$, we have $z' = z$ if $\beta' = (1+\beta)/2$.

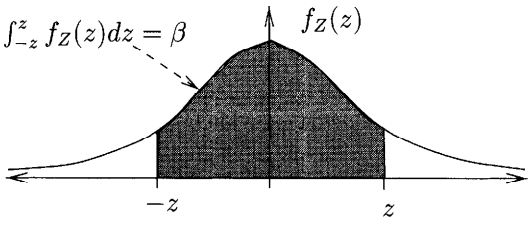

Figure 18.8: The bell-shaped Student-distribution with $n \geq 3$ degrees of freedom

We finally note that when using regenerative simulation, due to the use of ratio-estimators, the computation of confidence intervals becomes more complicated.

Example 18.8. Confidence interval determination.
From five mutually independent simulation runs we obtain the following five samples: $(x_1, \cdots, x_5) = (0.108, 0.112, 0.111, 0.115, 0.098)$. The sample mean $a = (\sum_{i=1}^{5} x_i)/5 = 0.1088$. The sample variance $\sigma^2 = \sum_{i=1}^{5}(x_i - a)^2/(5-1) = 0.0000427$. We assume a confidence level of $\beta = 0.9$. In Table 18.3, we find the corresponding double-sided critical value for the t_4-distribution with 90% confidence level $z = 2.132$. We thus obtain that:

$$\Pr\{|Z| \leq 2.132\} = \Pr\{|\tilde{X} - m| \leq 2.132\sigma/\sqrt{5}\},$$

from which we derive that $\Pr\{|\tilde{X} - m| \leq 0.00623\} = 0.90$. Thus, we know that:

$$\tilde{X} \in [0.1026, 0.1150] \text{ with } 90\% \text{ confidence.}$$

Notice that in practical situations, we should always aim to have at least ten degrees of freedom, i.e., $n \geq 10$. □

18.6 Further reading

More information on the statistical techniques to evaluate simulation results can be found in the performance evaluation textbooks by Jain [145], Mitrani [202], Trivedi [280] and in the survey by Pawlikowski [231]. The former two books also discuss the simulation setup and random number generation. More detailed information on the statistical techniques to be used in the evaluation of simulation results can be found in [90, 157, 175, 285] and statistics textbooks [137, 289]. Random number generation is treated extensively by Knuth

n	$\alpha = 90\%$	$\alpha = 95\%$	$\alpha = 99\%$	n	$\alpha = 90\%$	$\alpha = 95\%$	$\alpha = 99\%$
3	2.353	3.182	5.841	18	1.734	2.101	2.878
4	2.132	2.776	4.604	20	1.725	2.086	2.845
5	2.015	2.571	4.032	22	1.717	2.074	2.819
6	1.943	2.447	3.707	24	1.711	2.064	2.797
7	1.895	2.365	3.499	26	1.706	2.056	2.779
8	1.860	2.306	3.355	28	1.701	2.048	2.763
9	1.833	2.262	3.250	30	1.697	2.042	2.750
10	1.812	2.228	3.169	60	1.671	2.000	2.660
12	1.782	2.179	3.055	90	1.662	1.987	2.632
14	1.761	2.145	2.977	120	1.658	1.980	2.617
16	1.746	2.120	2.921	∞	1.645	1.960	2.576

Table 18.3: Double-sided critical values z of the Student distribution for n degrees of freedom and confidence level α such that $\Pr\{|Z| \leq z\} = \alpha$

[162], l'Ecuyer [79, 80] and others [56, 229]. Some recent tests of random number generators can be found in [179].

Research in simulation techniques is still continuing. In particular, when the interest is in obtaining measures that are to be associated with events that occur rarely, e.g., packet loss probabilities in communication systems or complete system failures in fault-tolerant computer systems, simulations tend to be very long. Special techniques to deal with these so-called *rare-event simulations* are being developed, see [99, 130, 222]. Pawlikowski gives an overview of the problems (and solutions) of the simulation of queueing processes [231]. Also techniques to speed-up simulations using parallel and distributed computer systems are receiving increased attention; for an overview of these techniques see [102]. Chiola and Ferscha discuss the special case of parallel simulations of SPNs [48].

18.7 Exercises

18.1. Random number generation.
Use a linear congruential method with $m = 18$, $c = 17$, $a = 7$ and seed $z_0 = 3$ to generate random numbers in $[0, 17]$.

Now make use of an additive congruential method to generate random numbers, using $m = 18$ and $a_0 = \cdots = a_{17} = 1$, thereby using as seeds the values computed with the linear

congruential method.

18.2. Uniform number generation.
Use the additive congruential method of the previous exercise to obtain 1000 uniformly-distributed random numbers on $[0, 1]$. Use the χ^2-test to validate this RNG. Use the auto-correlation test to validate the independence of successive random numbers $k = 1, 2, 3, 4$ generated with this RNG.

18.3. Uniform number generation.
Validate the following two RNGs, recently proposed by Fishman and Moore [91]:

- $z_n = 48271 z_{n-1}$ modulo $2^{32} - 1$;
- $z_n = 696211 z_{n-1}$ modulo $2^{32} - 1$.

18.4. How fair is a coin.
We try to compute the probability p with which the tossing of a coin yield "heads". We therefore toss the coin n times. How large should we take n such that with confidence level $\alpha = 0.9$ the width of the confidence interval is only 0.1 times the mean np. What happens (with n) if p tends to 0, that is, when tossing "heads" becomes a rare event?

18.5. Confidence interval construction.
As a result of a batch-means simulation we have obtained following batch means:

$$2.45 \quad 2.55 \quad 2.39 \quad 2.41 \quad 2.49 \quad 2.67 \quad 2.38 \quad 2.44 \quad 2.47$$

Compute confidence intervals, with $\alpha = 0.9, 0.95, 0.99$. How does the width of the confidence intervals change when the confidence level gets closer to 1? How many batches do you expect to be necessary to decrease the width of the confidence interval by a factor l.

18.6. Simulating a G|M|1 and a G|G|1 queue.
We consider the event-based simulation of the M|M|1 queue, as sketched in Section 18.3. How should we adapt the given program to simulate queues of M|G|1 and of G|G|1 type. Hint: do not "redraw" random number for non-exponential distributions, unless the associated events have really been executed.

18.7. Simulating a CTMC.
We are given a finite CTMC on state space $\mathcal{I}$, with generator matrix $\mathbf{Q}$ and initial probability distribution $\underline{p}(0)$. We are interested in the long-run proportion of time that the state

of the simulated CTMC is in some (given) subset $\mathcal{J}$ from $\mathcal{I}$. We start simulating at $t_s = 0$ until some value t_e; we only start collecting samples after the initial transient period of length t_i is over.

1. What is a useful estimator for the measure of interest?
2. Outline a simulation program for CTMC simulation, thereby using properties of CTMCs.

18.8. Simulating SPNs.
Given is an SPN of the class we have discussed in Chapter 14, however, without any restrictions on the timings of the transitions. Furthermore, we assume that a transition which is disabled due to the firing of another transition will resume its activity once it becomes enabled again. Thus, the time until the firing of a transition is not resampled every time it becomes enabled, but only when it becomes first enabled (after having fired or after simulation starts). Present the outline of a simulation program for SPNs obeying the sketched sampling strategy. What are the advantages of simulating SPNs instead of numerically solving them? And what are the disadvantages?