As discussed in Chapter 1, `ssq1` and `sis1` are examples of trace-driven discrete-event simulation programs. By definition, a trace-driven simulation relies on input data from an external source to supply recorded realizations of naturally occurring stochastic processes. Total reliance on such external data limits the applicability of a discrete-event simulation program, naturally inhibiting the user's ability to do "what if" studies. Given this limitation, a general discrete-event simulation objective is to develop methods for using a random number generator to convert a trace-driven discrete-event simulation program to a discrete-event simulation program that is not dependent on external data. This chapter provides several examples.

3.1.1 SINGLE-SERVER SERVICE NODE

Relative to the single-server service node model from Chapter 1, two stochastic assumptions are needed to free program `ssq1` from its reliance on external data. One assumption relates to the arrival times, the other assumption relates to the service times. We consider the service times first, using a *Uniform*(a, b) random variate model.

Example 3.1.1 Suppose that time is measured in minutes in a single-server service node model and all that is known about the service time is that it is random with possible values between 1.0 and 2.0. That is, although we know the range of possible values, we are otherwise in such a state of ignorance about the stochastic behavior of this server that we are unwilling to say some service times (between 1.0 and 2.0) are more likely than others. In this case we have modeled service time as a *Uniform*$(1.0, 2.0)$ random variable. Accordingly, random variate service times, say s, can be generated via the assignment

```
s = Uniform(1.0, 2.0);
```

Exponential Random Variates

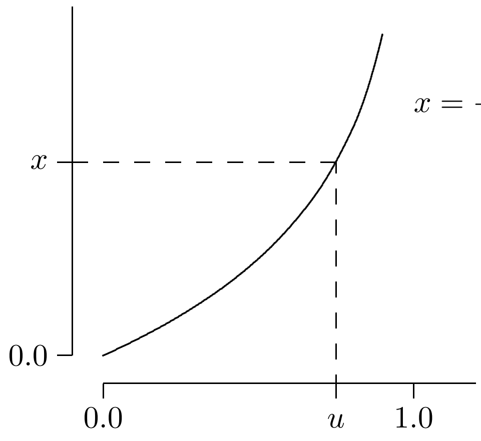
A *Uniform*(a, b) random variable has the property that all values between a and b are equally likely. In most applications this is an unrealistic assumption; instead, some values will be more likely than others. Specifically, there are many discrete-event simulation applications that require a continuous random variate, say x, that can take on any positive value but in such a way that small values of x are more likely than large values.

To generate such a random variate we need a *nonlinear* transformation that maps values of the random number u between 0.0 and 1.0 to values of x between 0.0 and ∞ and does so by "stretching" large values of u much more so than small values. Although a variety of such nonlinear transformation are possible, for example $x = u/(1 - u)$, perhaps the most common is $x = -\mu \ln(1 - u)$ where $\mu > 0$ is a parameter that controls the rate of stretching and $\ln(\cdot)$ is the natural logarithm (base e).* As explained in Chapter 7, this transformation generates what is known as an *Exponential*(μ) random variate.

* The function `log(`x`)` in the ANSI C library `<math.h>` represents the mathematical natural log function $\ln(x)$. In the equation $x = -\mu \ln(1 - u)$ do not confuse the positive, real-valued parameter μ with the *Uniform*$(0, 1)$ random variate u.

It is often appropriate to generate interarrivals as *Exponential*(μ) random variates, as we will do later in Example 3.1.2. In some important theoretical situations, service times are also modeled this way (see Section 8.5). The geometry associated with the *Exponential*(μ) random variate transformation $x = -\mu\ \ln(1-u)$ is illustrated in Figure 3.1.1.

Figure 3.1.1. Exponential variate generation geometry.



By inspection we see that the transformation is monotone increasing and, for any value of the parameter $\mu > 0$, the interval $0 < u < 1$ is mapped one-to-one and onto the interval $0 < x < \infty$. That is,

$$\begin{aligned} 0 < u < 1 &\iff 0 < (1-u) < 1 \\ &\iff -\infty < \ln(1-u) < 0 \\ &\iff 0 < -\mu\ \ln(1-u) < \infty \\ &\iff 0 < x < \infty. \end{aligned}$$

Definition 3.1.1 This ANSI C function generates an *Exponential*(μ) random variate*

```
double Exponential(double μ)                              /* use μ > 0.0 */
{
   return (-μ * log(1.0 - Random()));
}
```

The statistical significance of the parameter μ is that if repeated calls to the function `Exponential`(μ) are used to generate a random variate sample $x_1, x_2, \ldots, x_n$ then, in the limit as $n \to \infty$, the sample mean (average) of this sample will converge to μ. In the same sense, repeated calls to the function `Uniform`(a, b) will produce a sample whose mean converges to $(a+b)/2$ and repeated calls to the function `Equilikely`(a, b) will also produce a sample whose mean converges to $(a+b)/2$.

* The ANSI C standard says that `log(0.0)` may produce "a range error". This possibility is naturally avoided in the function `Exponential` because the largest possible value of `Random` is less than 1.0. See Exercise 3.1.3.

Example 3.1.2 In a single-server service node simulation, to generate a sequence of random variate *arrival* times $a_1, a_2, a_3, \ldots, a_n$ with an average interarrival time that will converge to μ as $n \to \infty$ it is common to generate *Exponential*(μ) *interarrival* times (see Definition 1.2.3) and then (with $a_0 = 0$) create the arrival times by the assignment

a_i `=` a_{i-1} `+ Exponential(`μ`);` $\qquad i = 1, 2, 3, \ldots, n$

As discussed in Chapter 7, this use of an *Exponential*(μ) random variate corresponds naturally to the idea of jobs arriving *at random* with an arrival rate that will converge to $1/\mu$ as $n \to \infty$. Similarly, as in Example 3.1.1, to generate a random variate sequence of service times $s_1, s_2, s_3, \ldots, s_n$ equally likely to lie anywhere between a and b (with $0 \le a < b$) and with an average service time that converges to $(a + b)/2$, the assignment

s_i `= Uniform(`a`, `b`);` $\qquad i = 1, 2, 3, \ldots, n$

can be used. The average service time $(a + b)/2$ corresponds to a service rate of $2/(a + b)$.

Program ssq2

Program `ssq2` is based on the two stochastic modeling assumptions in Example 3.1.2.* Program `ssq2` is an extension of program `ssq1` in that the arrival times and service times are generated randomly (rather than relying on a trace-driven input) and a complete set of first-order statistics $\bar{r}$, $\bar{w}$, $\bar{d}$, $\bar{s}$, $\bar{l}$, $\bar{q}$, $\bar{x}$ is generated. Note that each time the function `GetArrival()` is called the static variable `arrival`, which represents an *arrival* time, is incremented by a call to the function `Exponential(2.0)`, which generates an *interarrival* time.

Because program `ssq2` generates stochastic data as needed there is essentially no restriction on the number of jobs that can be processed. Therefore, the program can be used to study the *steady-state* behavior of a single-server service node. That is, by experimenting with an increasing number of jobs processed, one can investigate whether or not the service node statistics will converge to constant values, *independent* of the choice of the `rng` initial seed and the initial state of the service node. Steady-state behavior — can it be achieved and, if so, how many jobs does it take to do so — is an important issue that will be explored briefly in this chapter and in more detail in Chapter 8.

Program `ssq2` can also be used to study the *transient* behavior of a single-server service node. The idea in this case is to fix the number of jobs processed at some finite value and run (replicate) the program repeatedly with the initial state of the service node fixed, changing *only* the `rng` initial seed from run to run. In this case replication will produce a natural variation in the service node statistics consistent with the fact that for a fixed number of jobs, the service node statistics are *not* independent of the initial seed or the initial state of the service node. Transient behavior, and its relation to steady-state behavior, will be considered in Chapter 8.

* In the jargon of queuing theory (see Section 8.5), program `ssq2` simulates what is known as an $M/G/1$ queue (see Kleinrock, 1975, 1976, or Gross and Harris, 1985).

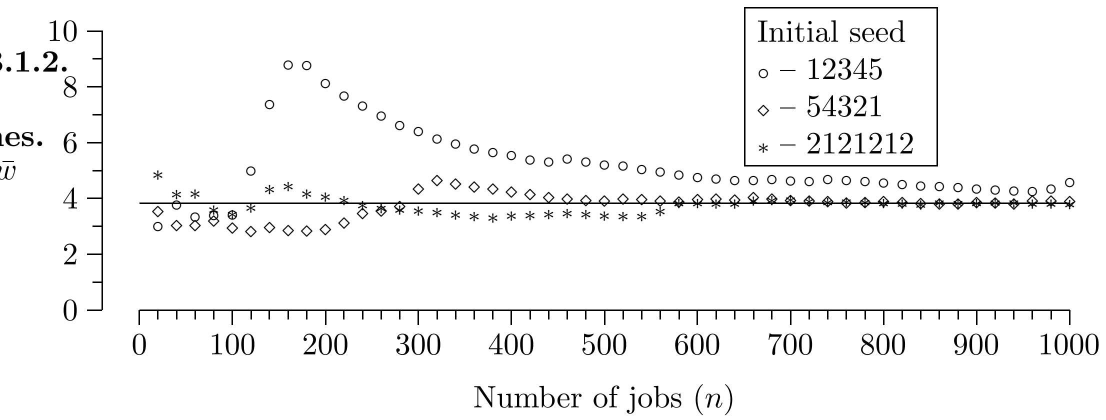
Steady-State Statistics

Example 3.1.3 If the *Exponential*(μ) interarrival time parameter is set to $\mu = 2.0$ so that the steady-state arrival rate is $1/\mu = 0.5$ and if the *Uniform*(a, b) service time parameters are set to $a = 1.0$ and $b = 2.0$ respectively so that the steady-state service rate is $2/(a + b) \cong 0.67$, then to *d.dd* precision the theoretical steady-state statistics generated from an analytic model (exact, not estimates by simulation) program `ssq2` will produce are (see Gross and Harris, 1985)

$\bar{r}$	$\bar{w}$	$\bar{d}$	$\bar{s}$	$\bar{l}$	$\bar{q}$	$\bar{x}$
2.00	3.83	2.33	1.50	1.92	1.17	0.75

Therefore, although the server is only busy 75% of the time ($\bar{x} = 0.75$), "on average" there are approximately two jobs ($\bar{l} = 23/12 \cong 1.92$) in the service node and a job can expect to spend more time ($\bar{d} = 23/6 - 3/2 \cong 2.33$ time units) in the queue than in service ($\bar{s} = 1.50$ time units). As illustrated in Figure 3.1.2 for the average wait, the number of jobs that must be pushed through the service node to achieve these steady-state statistics is large. To produce this figure, program `ssq2` was modified to print the accumulated average wait every 20 jobs. The results are presented for three choices of the `rng` initial seed.* The solid, horizontal line at height $23/6 \cong 3.83$ represents the steady-state value of $\bar{w}$.

Figure 3.1.2. Average wait times.



In Example 3.1.3 convergence of $\bar{w}$ to the steady-state value 3.83 is slow, erratic, and very dependent on the random variate sequence of stochastic arrival and service times, as manifested by the choice of initial seed. Note, for example, the dramatic rise in average wait beginning at about job 100 associated with the `rng` initial seed 12345. You are encouraged to add diagnostic printing and additional statistics gathering to program `ssq2` to better understand what combination of chance events occurred to produce this rise.

The *Uniform*(a, b) service time model in Example 3.1.3 may not be realistic. Service times seldom "cut off" beyond a minimum and maximum value. More detail will be given in subsequent chapters on how to select realistic distributions for input models.

* There is nothing special about these three initial seeds. Do not fall into the common trap of thinking that some `rng` initial seeds are necessarily better than others.

Example 3.1.3 shows that the stochastic character of the arrival times and service times, as manifested by the choice of `rng` initial seed, has a significant effect on the transition-to-steady-state behavior of a single-server service node. This example also illustrates the use of the library `rng` to conduct *controlled* "what if" experiments. Studies like this are the stuff of discrete-event simulation. Additional examples of this kind of experimentation, and the selection of values of μ, a, and b, are presented in later chapters.

Geometric Random Variates

As discussed in Chapter 2, an *Equilikely*(a,b) random variate is the discrete analog of a continuous *Uniform*(a,b) random variate. Consistent with this characterization, one way to generate an *Equilikely*(a,b) random variate is to generate a *Uniform*$(a,\ b+1)$ random variate instead (note that the upper limit is $b+1$, not b) and then convert (cast) the resulting floating-point result to an integer. That is, if a and b are integers with $a < b$ and if x is a *Uniform*$(a,\ b+1)$ random variate then $\lfloor x \rfloor$ is an *Equilikely*(a,b) random variate.

Given the analogy between *Uniform*(a,b) and *Equilikely*(a,b) random variates, it is reasonable to expect that there is a discrete analog to a continuous *Exponential*(μ) random variate; and there is. Specifically, if x is an *Exponential*(μ) random variate, let y be the *discrete* random variate defined by $y = \lfloor x \rfloor$. For a better understanding of this discrete random variate, let $p = \Pr(y \neq 0)$ denote the probability that y is not zero. Since x is generated as $x = -\mu \ln(1-u)$ with u a *Uniform*$(0,1)$ random variate, $y = \lfloor x \rfloor$ will not be zero if and only if $x \geq 1$. Equivalently

$$\begin{aligned} x \geq 1 &\iff -\mu \ln(1-u) \geq 1 \\ &\iff \ln(1-u) \leq -1/\mu \\ &\iff 1-u \leq \exp(-1/\mu) \end{aligned}$$

where $\exp(-1/\mu)$ is $e^{-1/\mu}$, and so $y \neq 0$ if and only if $1-u \leq \exp(-1/\mu)$. Like u, $1-u$ is also a *Uniform*$(0,1)$ random variate. Moreover, for any $0 < p < 1$, the condition $1-u \leq p$ is true with probability p (see Section 7.1). Therefore, $p = \Pr(y \neq 0) = \exp(-1/\mu)$.

If x is an *Exponential*(μ) random variate and if $y = \lfloor x \rfloor$ with $p = \exp(-1/\mu)$, then it is conventional to call y a *Geometric*(p) random variable (see Chapter 6). Moreover, it is conventional to use p rather than $\mu = -1/\ln(p)$ to define y directly by the equation

$$y = \lfloor \ln(1-u)/\ln(p) \rfloor.$$

Definition 3.1.2 This ANSI C function generates a *Geometric*(p) random variate*

```
long Geometric(double p)                      /* use 0.0 < p < 1.0 */
{
   return ((long) (log(1.0 - Random()) / log(p)));
}
```

* Note that `log(0.0)` is avoided in this function because the largest possible value returned by `Random` is less than 1.0.

In addition to its significance as $\Pr(y \neq 0)$, the parameter p is also related to the mean of a *Geometric*(p) sample. Specifically, if repeated calls to the function `Geometric(p)` are used to generate a random variate sample $y_1, y_2, \ldots, y_n$ then, in the limit as $n \to \infty$, the mean of this sample will converge to $p/(1-p)$. Note that if p is close to 0.0 then the mean will be close to 0.0. At the other extreme, if p is close to 1.0 then the mean will be large.

In the following example, a *Geometric*(p) random variate is used as part of a *composite* service time model. In this example the parameter p has been adjusted to make the average service time match that of the *Uniform*$(1.0, 2.0)$ server in program `ssq2`.

Example 3.1.4 Usually one has sufficient information to argue that a *Uniform*(a, b) random variate service time model is not appropriate; instead, a more sophisticated model is justified. Consider a hypothetical server that, as in program `ssq2`, processes a stream of jobs arriving, at random, with a steady-state arrival rate of 0.5 jobs per minute. The service requirement associated with each arriving job has two stochastic components:

- the *number* of service tasks is one plus a *Geometric*(0.9) random variate;
- the *time* (in minutes) per task is, independently for each task, a *Uniform*$(0.1, 0.2)$ random variate.

In this case, program `ssq2` would need to be modified by including the function `Geometric` from Definition 3.1.2 and changing the function `GetService` to the following.

```
double GetService(void)
{
   long   k;
   double sum   = 0.0;
   long   tasks = 1 + Geometric(0.9);

   for (k = 0; k < tasks; k++)
     sum += Uniform(0.1, 0.2);
   return (sum);
}
```

With this modification, the population steady-state statistics from the analytic model to *d.dd* precision that will be produced are:

$\bar{r}$	$\bar{w}$	$\bar{d}$	$\bar{s}$	$\bar{l}$	$\bar{q}$	$\bar{x}$
2.00	5.77	4.27	1.50	2.89	2.14	0.75

Program `ssq2` will produce results that converge to these values for a very long run. When compared with the steady-state results in Example 3.1.3, note that although the arrival rate $1/\bar{r} = 0.50$, the service rate $1/\bar{s} = 0.67$, and the utilization $\bar{x} = 0.75$ are the same, the other four statistics are significantly larger than in Example 3.1.3. This difference illustrates the sensitivity of the performance measures to the service time distribution. This highlights the importance of using an accurate service time model. See Exercise 3.1.5.

3.1.2 SIMPLE INVENTORY SYSTEM

Example 3.1.5 In a simple inventory system simulation, to generate a random variate sequence of demands d_1, d_2, d_3, ... equally likely to have any integer value between a and b inclusive and with an average of $(a + b)/2$ use

d_i = `Equilikely(a, b)` $\qquad i = 1, 2, 3, \ldots$

Recognize, however, that the previous discussion about *Uniform*(a, b) service times as an unrealistic model also applies here in the discrete case. That is, in this application the modeling assumption that all the demands between a and b are equally likely is probably unrealistic; some demands should be more likely than others. As an alternative model we could consider generating the demands as *Geometric*(p) random variates. In this particular case, however, a *Geometric*(p) demand model is probably not very realistic either. In Chapter 6 we will consider alternative models that are more appropriate.

Program sis2

Program `sis2` is based on the *Equilikely*(a, b) stochastic modeling assumption in Example 3.1.5. This program is an extension of program `sis1` in that the demands are generated randomly (rather than relying on a trace-driven input). Consequently, in a manner analogous to that illustrated in Example 3.1.3, program `sis2` can be used to study the transition-to-steady-state behavior of a simple inventory system.

Example 3.1.6 If the *Equilikely*(a, b) demand parameters are set to $(a, b) = (10, 50)$ so that the average demand per time interval is $(a + b)/2 = 30$, then with $(s, S) = (20, 80)$ the (approximate) steady-state statistics program `sis2` will produce are

$\bar{d}$	$\bar{o}$	$\bar{u}$	$\bar{l}^+$	$\bar{l}^-$
30.00	30.00	0.39	42.86	0.26

As illustrated in Figure 3.1.3 (using the same three `rng` initial seeds used in Example 3.1.3) for the average inventory level $\bar{l} = \bar{l}^+ - \bar{l}^-$, at least several hundred time intervals must be simulated to approximate these steady-state statistics. To produce Figure 3.1.3, program `sis2` was modified to print the accumulated value of $\bar{l}$ every 5 time intervals.

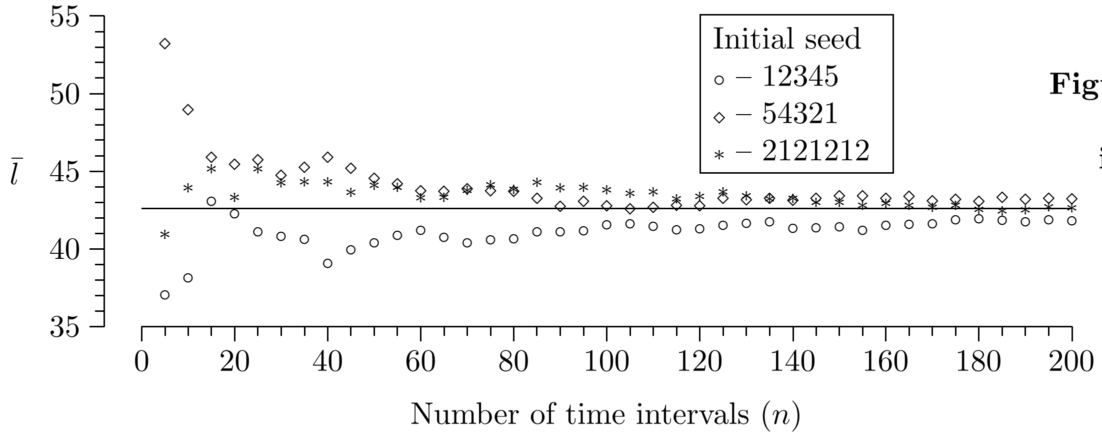
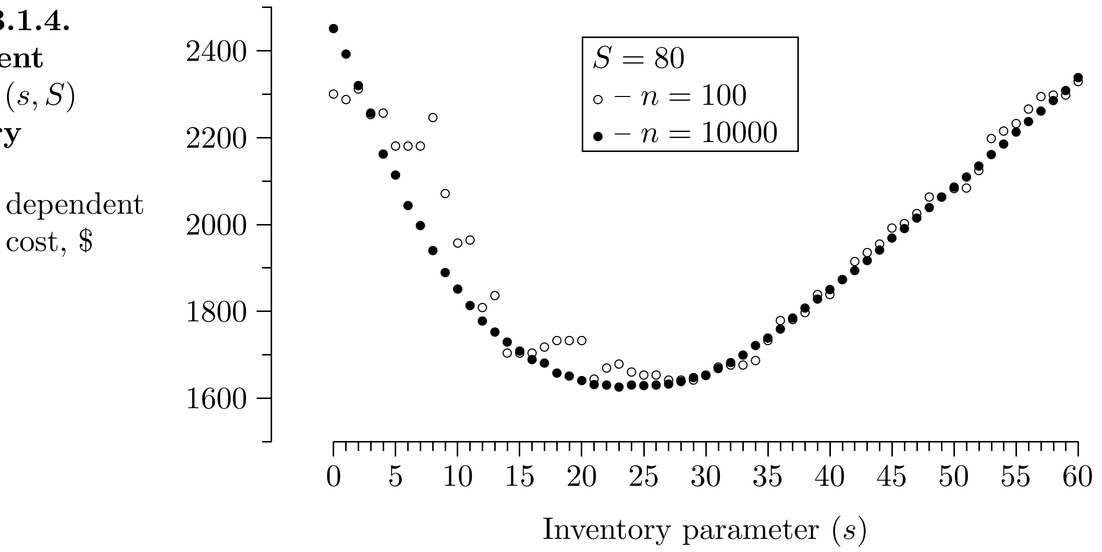


Figure 3.1.3. Average inventory level.

Optimal Steady-State System Performance

Example 3.1.7 As an extension of Example 1.3.7, a modified version of program `sis2` was used to simulate $n = 100$ weeks (about two years) of automobile dealership operation with $S = 80$ and values of s varied from 0 to 60. The cost parameters from Example 1.3.5 were used to determine the variable part of the dealership's average weekly cost of operation. That is, as discussed in Section 1.3, the (large) part of the average weekly cost that is proportional to the average weekly order $\bar{o}$, and therefore is independent of (s, S), was ignored so that only the dependent cost was computed. The results are presented with $\circ$'s representing the $n = 100$ averages. As illustrated in Figure 3.1.4, there is a relatively well-defined minimum with an optimal value of s somewhere between 20 and 30.

Figure 3.1.4. Dependent cost for (s, S) inventory system.



The same initial seed (12345) was used for each value of s. Fixing the initial seed guarantees that *exactly* the same sequence of demands will be processed; therefore, any changes in the resulting system statistics are due to changes in s only. As in Example 3.1.6, the demands were generated as *Equilikely*$(10, 50)$ random variates. To compute steady-state statistics, all the modeling assumptions upon which this simulation is based would have to remain valid for many years. For that reason, steady-state statistics are not very meaningful in this example. Indeed, it is questionable to assume that the demand distribution, cost parameters, and inventory policy will remain constant for even two years. Steady-state performance statistics are fashionable, however, and represent an interesting limiting case. For that reason averages based on $n = 10000$ weeks of operation (approximately 192 years) are presented as $\bullet$'s for comparison. These estimated steady-state averages have the attractive feature that they vary smoothly with s and, because of that, the optimum steady-state value of s is relatively well defined. In a world that (on average) never changes, if $S = 80$ the auto dealer should use an inventory threshold of $s = 23$. This example illustrates the more general problem of optimizing a function that is not known with certainty known as *stochastic optimization.*

3.1.3 STATISTICAL CONSIDERATIONS

The statistical analysis of simulation-generated data is discussed in Chapters 4 and 8. In anticipation of that discussion, we note that Example 3.1.7 illustrates two important ideas: *variance reduction* and *robust estimation.* We consider variance reduction first, particularly as it relates to transient (small n) output statistics.

Variance Reduction

Because the transient statistical output produced by any discrete-event simulation program depends on the sequence of random variates generated as the program executes, the output will always have some inherent uncertainty. That is, as illustrated in Examples 3.1.3 and 3.1.6, the transient statistical output depends on the value of the `rng` initial seed, particularly if the number of jobs or the number of simulated time intervals is small. Therefore, if a variety of initial seeds are used (a policy we certainly advocate) there will be a natural *variance* in any computed transient statistic.

The statistical tools used to quantify this variance/uncertainty will be developed in Chapter 8. For now, the key point to be made is that using *common random numbers*, as in Example 3.1.7 when $n = 100$, is an intuitive approach to reducing the variance in computed system statistics. This example of variance reduction (using common random numbers) is consistent with the time-honored approach to experimental scientific "what if" studies where, if possible, all variables except one are fixed. Fixing the initial seed at 12345 in Example 3.1.7 isolates the variability of the performance measure (dependent cost). This technique is more generally known as *blocking* in statistics.

Robust Estimation

The optimal (minimal cost) estimated steady-state threshold value $s = 23$ in Example 3.1.7 is a robust estimate because other values of s close to 23 yield essentially the same cost. Therefore, the impact of using one of these alternate values of s in place of $s = 23$ would be slight. In a more general sense, it is desirable for an optimal value to be robust to *all* the assumptions upon which the discrete-event simulation model is based. Relative to Example 3.1.7, this means that we would be interested in determining what happens to the optimal value of s when, for example S is varied about 80, or the average demand (per week) is varied about 30, or there are delivery lags. The estimate $s = 23$ is robust only if it remains close to the minimal cost operating policy level when all these assumptions are altered. In general, robust estimators are insensitive to model assumptions.

3.1.4 EXERCISES

Exercise 3.1.1 (a) Modify program `ssq2` to use *Exponential*(1.5) service times. (b) Process a relatively large number of jobs, say 100 000, and determine what changes this produces relative to the statistics in Example 3.1.3? (c) Explain (or conjecture) why some statistics change and others do not.

Exercise 3.1.2 (*a*) Relative to the steady-state statistics in Example 3.1.3 and the statistical equations in Section 1.2, list all of the consistency checks that should be applicable. (*b*) Verify that all of these consistency checks are valid.

Exercise 3.1.3 (*a*) Given that the Lehmer random number generator used in the library `rng` has a modulus of $2^{31} - 1$, what are the largest and smallest possible numerical values (as a function of μ) that the function `Exponential`(μ) can return? (*b*) Comment on this relative to the theoretical expectation that an *Exponential*(μ) random variate can have an arbitrarily large value and a value arbitrarily close to zero.

Exercise 3.1.4 (*a*) Conduct a transition-to-steady-state study like that in Example 3.1.3 except for a service time model that is *Uniform*(1.3, 2.3). Be specific about the number of jobs that seem to be required to produce steady-state statistics. (*b*) Comment.

Exercise 3.1.5 (*a*) Verify that the mean service time in Example 3.1.4 is 1.5. (*b*) Verify that the steady-state statistics in Example 3.1.4 seem to be correct. (*c*) Note that the arrival rate, service rate, and utilization are the same as those in Example 3.1.3, yet all the other statistics are larger than those in Example 3.1.3. Explain (or conjecture) why this is so. Be specific.

Exercise 3.1.6 (*a*) Modify program `sis2` to compute data like that in Example 3.1.7. Use the functions `PutSeed` and `GetSeed` from the library `rng` in such a way that *one* initial seed is supplied by the system clock, printed as part of the program's output and used automatically to generate the same demand sequence for all values of s. (*b*) For $s = 15, 16, \ldots, 35$ create a figure (or table) similar to the one in Example 3.1.7. (*c*) Comment.

Exercise 3.1.7 (*a*) Relative to Example 3.1.5, if instead the random variate sequence of demands are generated as

$$d_i \texttt{ = Equilikely(5, 25) + Equilikely(5, 25)} \qquad i = 1, 2, 3, \ldots$$

then, when compared with those in Example 3.1.6, demonstrate that some of the steady-state statistics will be the same and others will not. (*b*) Explain why this is so.

Exercise 3.1.8[a] Modify program `sis2` to simulate the operation of a simple inventory system *with a delivery lag*. (*a*) Specifically, assume that if an order is placed at time $t = i - 1$ then the order will arrive at the later time $t = i - 1 + \delta_i$ where the delivery lag δ_i is a *Uniform*(0, 1) random variate, independent of the size of the order. (*b*) What are the equations for $\bar{l}_i^{+}$ and $\bar{l}_i^{-}$? (*c*) Using the same parameter values as in Example 3.1.7, determine that value of s for which the average dependent cost is least. Compare this result with that obtained in Example 3.1.7. (*d*) It is important to have the *same* sequence of demands for all values of s, with and without a lag. Why? How did you accomplish this? (*e*) Discuss what you did to convince yourself that the modification is correct. (*f*) For both $n = 100$ and $n = 10000$ produce a table of dependent costs corresponding to $s = 10, 20, 30, 40, 50, 60$.

3.2 Multi-Stream Lehmer Random Number Generators

A typical discrete-event simulation model will have many stochastic components. When this model is implemented at the computational level, the statistical analysis of system performance is often facilitated by having a unique source of randomness for each stochastic component. Although it may seem that the best way to meet this need for multiple sources of randomness is to create multiple random number generators, there is a simpler and better approach — use *one* random number generator to generate multiple "streams" of random numbers using multiple initial seeds as entry points, one for each stochastic system component. Consistent with this approach, in this section we extend the Lehmer random number generation algorithm from Chapter 2 by adding the ability to partition the generator's output sequence into multiple subsequences (streams).

3.2.1 STREAMS

The library `rng` provides a way to partition the random number generator's output into multiple streams by establishing multiple states for the generator, one for each stream. As illustrated by the following example, the function `PutSeed` can be used to set the state of the generator with the current state of the stream before generating a random variate appropriate to the corresponding stochastic component and the function `GetSeed` can be used to retrieve the revised state of the stream after the random variate has been generated.

Example 3.2.1 The program `ssq2` has two stochastic components, the arrival process and the service process, represented by the functions `GetArrival` and `GetService` respectively. To create a different stream of random numbers for each component, it is sufficient to allocate a different Lehmer generator state variable to each function. This is illustrated by modifying `GetService` from its original form in `ssq2`, which is

```
double GetService(void)                                 /* original form */
{
   return (Uniform(1.0, 2.0));
}
```

to the multi-stream form indicated which uses the static variable `x` to represent the current state of the service process stream, initialized to 123456789.

```
double GetService(void)                             /* multi-stream form */
{
   double s;
   static long x = 123456789;      /* use your favorite initial seed */
   PutSeed(x);                     /* set the state of the generator */
   s = Uniform(1.0, 2.0);
   GetSeed(&x);                      /* save the new generator state */
   return (s);
}
```

Example 3.2.2 As in the previous example, the function `GetArrival` should be modified similarly, with a corresponding static variable to represent the current state of the arrival process stream, but initialized to a *different* value. That is, the original form of `GetArrival` in program `ssq2`

```
double GetArrival(void)                               /* original form */
{
   static double arrival = START;
   arrival += Exponential(2.0);
   return (arrival);
}
```

should be modified to something like

```
double GetArrival(void)                           /* multi-stream form */
{
   static double arrival = START;
   static long x = 987654321;     /* use an appropriate initial seed */
   PutSeed(x);                     /* set the state of the generator */
   arrival += Exponential(2.0);
   GetSeed(&x);                      /* save the new generator state */
   return (arrival);
}
```

As in Example 3.2.1, in the multi-stream form the static variable `x` represents the current state of the arrival process stream, initialized in this case to 987654321. Note that there is nothing magic about this initial state (relative to 123456789) and, indeed, it may not even be a particularly good choice — more about that point later in this section.

If `GetService` and `GetArrival` are modified as in Examples 3.2.1 and 3.2.2, then the arrival times will be drawn from one stream of random numbers and the service times will be drawn from another stream. Provided the two streams don't overlap, in this way the arrival process and service process will be uncoupled.* As the following example illustrates, the cost of this uncoupling in terms of execution time is modest.

Example 3.2.3 The parameter `LAST` in program `ssq2` was changed to process 1 000 000 jobs and the execution time to process this many jobs was recorded. (A large number of jobs was used to get an accurate time comparison.) Program `ssq2` was then modified as in Examples 3.2.1 and 3.2.2 and used to process 1 000 000 jobs, with the execution time recorded. The increase in execution time was 20%.

* Also, because the scope of the two stream state variables (both are called `x`) is local to their corresponding functions, the use of `PutSeed` in program `ssq2` to initialize the generator can, and should, be eliminated from `main`.

Jump Multipliers

As illustrated in the previous examples, the library `rng` can be used to support the allocation of a unique stream of random numbers to each stochastic component in a discrete-event simulation program. There is, however, a potential problem with this approach — the assignment of initial seeds. That is, each stream requires a unique initial state that should be chosen to produce *disjoint* streams. But, if multiple initial states are picked at whim there is no convenient way to guarantee that the streams are disjoint; some of the initial states may be just a few calls to `Random` away from one another. With this limitation of the library `rng` in mind, we now turn to the issue of constructing a random number generation library called `rngs` which is a multi-stream version of the library `rng`. We begin by recalling two key points from Section 2.1.

- A Lehmer random number generator is defined by the function
$$g(x) = ax \bmod m,$$
where the modulus m is a large prime integer, the full-period multiplier a is modulus compatible with m, and $x \in \mathcal{X}_m = \{1, 2, \ldots, m-1\}$.
- If $x_0, x_1, x_2, \ldots$ is an infinite sequence in $\mathcal{X}_m$ generated by $g(x) = ax \bmod m$ then each x_i is related to x_0 by the equation
$$x_i = a^i x_0 \bmod m \qquad\qquad i = 1, 2, \ldots$$

The following theorem is the key to creating the library `rngs`. The proof is left as an exercise.

Theorem 3.2.1 Given a Lehmer random number generator defined by $g(x) = ax \bmod m$ and any integer j with $1 < j < m - 1$, the associated *jump function* is
$$g^j(x) = (a^j \bmod m)\, x \bmod m$$
with the *jump multiplier* $a^j \bmod m$. For any $x_0 \in \mathcal{X}_m$, if the function $g(\cdot)$ generates the sequence $x_0, x_1, x_2, \ldots$ then the jump function $g^j(\cdot)$ generates the sequence $x_0, x_j, x_{2j}, \ldots$

Example 3.2.4 If $m = 31$, $a = 3$, and $j = 6$ then the jump multiplier is
$$a^j \bmod m = 3^6 \bmod 31 = 16.$$
Starting with $x_0 = 1$ the function $g(x) = 3x \bmod 31$ generates the sequence
$$\underline{1}, 3, 9, 27, 19, 26, \underline{16}, 17, 20, 29, 25, 13, \underline{8}, 24, 10, 30, 28, 22, \underline{4}, 12, 5, 15, 14, 11, \underline{2}, 6, \ldots$$
while the jump function $g^6(x) = 16x \bmod 31$ generates the sequence of underlined terms
$$1, 16, 8, 4, 2, \ldots$$
That is, the first sequence is $x_0,\ x_1,\ x_2,\ \ldots$ and the second sequence is $x_0,\ x_6,\ x_{12},\ \ldots$

The previous example illustrates that once the jump multiplier $a^j \bmod m$ has been computed — this is a one-time cost — then the jump function $g^j(\cdot)$ provides a mechanism to jump from x_0 to x_j to x_{2j}, etc. If j is properly chosen then the jump function can be used in conjunction with a user supplied initial seed to "plant" additional initial seeds, each separated one from the next by j calls to `Random`. In this way disjoint streams can be automatically created with the initial state of each stream dictated by the choice of just *one* initial state.

Example 3.2.5 There are approximately 2^{31} possible values in the full period of our standard $(a, m) = (48271,\ 2^{31} - 1)$ Lehmer random number generator. Therefore, if we wish to maintain $256 = 2^8$ streams of random numbers (the choice of 256 is largely arbitrary) it is natural to partition the periodic sequence of possible values into 256 disjoint subsequences, each of equal length. This is accomplished by finding the largest value of j less than $2^{31}/2^8 = 2^{23} = 8388608$ such that the associated jump multiplier $48271^j \bmod m$ is modulus-compatible with m. Because this jump multiplier is modulus-compatible, the jump function

$$g^j(x) = (48271^j \bmod m)\, x \bmod m$$

can be implemented using Algorithm 2.2.1. This jump function can then be used in conjunction with one user supplied initial seed to efficiently plant the other 255 additional initial seeds, each separated one from the next by $j \cong 2^{23}$ steps.* By planting the additional seeds this way, the possibility of stream overlap is minimized.

Maximal Modulus-Compatible Jump Multipliers

Definition 3.2.1 Given a Lehmer random number generator with (prime) modulus m, full-period modulus-compatible multiplier a, and a requirement for s disjoint streams as widely separated as possible, the *maximal* jump multiplier is $a^j \bmod m$ where j is the largest integer less than $\lfloor m/s \rfloor$ such that $a^j \bmod m$ is modulus compatible with m.

Example 3.2.6 Consistent with Definition 3.2.1 and with $(a, m) = (48271,\ 2^{31} - 1)$ a table of maximal modulus-compatible jump multipliers can be constructed for 1024, 512, 256, and 128 streams, as illustrated.

# of streams s	$\lfloor m/s \rfloor$	jump size j	jump multiplier $a^j \bmod m$
1024	2097151	2082675	97070
512	4194303	4170283	44857
256	8388607	8367782	22925
128	16777215	16775552	40509

Computation of the corresponding table for $a = 16807$ (the minimal standard multiplier) is left as an exercise.

* Because j is less than $2^{31}/2^8$, the last planted initial seed will be more than j steps from the first.

Library rngs

The library `rngs` is an upward-compatible multi-stream replacement for the library `rng`. The library `rngs` can be used as an alternative to `rng` in any of the programs presented earlier by replacing

```
#include "rng.h"
```

with

```
#include "rngs.h"
```

As configured `rngs` provides for 256 streams, indexed from 0 to 255, with 0 as the default stream. Although the library is designed so that all streams will be initialized to default values if necessary, the recommended way to initialize all streams is by using the function `PlantSeeds`. Only one stream is *active* at any time; the other 255 are *passive.* The function `SelectStream` is used to define the active stream. If the default stream is used exclusively, so that 0 is *always* the active stream, then the library `rngs` is functionally equivalent to the library `rng` in the sense that `rngs` will produce *exactly* the same `Random` output as `rng` (for the same initial seed, of course).

The library `rngs` provides six functions, the first four of which correspond to analogous functions in the library `rng`.

- `double Random(void)` — This is the Lehmer random number generator used throughout this book.
- `void PutSeed(long x)` — This function can be used to set the state of the active stream.
- `void GetSeed(long *x)` — This function can be used to get the state of the active stream.
- `void TestRandom(void)` — This function can be used to test for a correct implementation of the library.
- `void SelectStream(int s)` — This function can be used to define the active stream, i.e., the stream from which the next random number will come. The active stream will remain as the source of future random numbers until another active stream is selected by calling `SelectStream` with a different stream index `s`.
- `void PlantSeeds(long x)` — This function can be used to set the state of all the streams by "planting" a sequence of states (seeds), one per stream, with all states dictated by the state of the default stream. The following convention is used to set the state of the default stream:

    if `x` is positive then `x` is the state;

    if `x` is negative then the state is obtained from the system clock;

    if `x` is 0 then the state is to be supplied interactively.

3.2.2 EXAMPLES

The following examples illustrate how to use the library `rngs` to allocate a separate stream of random numbers to each stochastic component of a discrete-event simulation model. We will see additional illustrations of how to use `rngs` in this and later chapters. From this point on `rngs` will be the basic random number generation library used for *all* the discrete-event simulation programs in this book.

Example 3.2.7 As a superior alternative to the multi-stream generator approach in Examples 3.2.1 and 3.2.2, the functions `GetArrival` and `GetService` in program `ssq2` can be modified to use the library `rngs`, as illustrated

```
double GetArrival(void)
{
   static double arrival = START;
   SelectStream(0);                               /* this line is new */
   arrival += Exponential(2.0);
   return (arrival);
}

double GetService(void)
{
   SelectStream(2);                               /* this line is new */
   return (Uniform(1.0, 2.0));
}
```

The other modification is to include `"rngs.h"` in place of `"rng.h"` and use the function `PlantSeeds(123456789)` in place of `PutSeed(123456789)` to initialize the streams.*

If program `ssq2` is modified consistent with Example 3.2.7, then the arrival process will be *uncoupled* from the service process. That is important because we may want to study what happens to system performance if, for example, the `return` in the function `GetService` is replaced with

```
return (Uniform(0.0, 1.5) + Uniform(0.0, 1.5));
```

Although two calls to `Random` are now required to generate each service time, this new service process "sees" *exactly* the same job arrival sequence as did the old service process. This kind of uncoupling provides a desirable variance reduction technique when discrete-event simulation is used to compare the performance of different systems.

* Note that there is nothing magic about the use of `rngs` stream 0 for the arrival process and stream 2 for the service process — any two different streams can be used. In particular, if even more separation between streams is required then, for example, streams 0 and 10 can be used.

A Single-Server Service Node With Multiple Job Types

A meaningful extension to the single-server service node model is *multiple job types*, each with its own arrival and service process. This model extension is easily accommodated at the conceptual level; each arriving job carries a job type that determines the kind of service provided when the job enters service. Similarly, provided the queue discipline is FIFO the model extension is straightforward at the specification level. Therefore, using program ssq2 as a starting point, we can focus on the model extension at the implementation level. Moreover, we recognize that to facilitate the use of common random numbers, the library rngs can be used with a different stream allocated to each of the stochastic arrival and service processes in the model. The following example is an illustration.

Example 3.2.8 Suppose that there are two job types arriving independently, one with *Exponential*(4.0) interarrivals and *Uniform*(1.0, 3.0) service times and the other with *Exponential*(6.0) interarrivals and *Uniform*(0.0, 4.0) service times. In this case, the arrival process generator in program ssq2 can be modified as

```
double GetArrival(int *j)                       /* j denotes job type */
{
   const  double mean[2]    = {4.0, 6.0};
   static double arrival[2] = {START, START};
   static int    init       = 1;
          double temp;
   if (init) {                        /* initialize the arrival array */
     SelectStream(0);
     arrival[0] += Exponential(mean[0]);
     SelectStream(1);
     arrival[1] += Exponential(mean[1]);
     init       = 0;
   }
   if (arrival[0] <= arrival[1])
     *j = 0;                            /* next arrival is job type 0 */
   else
     *j = 1;                            /* next arrival is job type 1 */
   temp = arrival[*j];                            /* next arrival time */
   SelectStream(*j);                   /* use stream j for job type j */
   arrival[*j] += Exponential(mean[*j]);
   return (temp);
}
```

Note that GetArrival returns the next arrival time *and* the job type as an index with value 0 or 1, as appropriate.

Example 3.2.9 As a continuation of Example 3.2.8, the corresponding service process generator in program `ssq2` can be modified as

```
double GetService(int j)
{
   const double min[2] = {1.0, 0.0};
   const double max[2] = {3.0, 4.0};
   SelectStream(j + 2);          /* use stream j + 2 for job type j */
   return (Uniform(min[j], max[j]));
}
```

Relative to Example 3.2.9, note that the job type index `j` is used in `GetService` to insure that the service time corresponds to the appropriate job type. Also, `rngs` streams 2 and 3 are allocated to job types 0 and 1 respectively. In this way all four simulated stochastic processes are uncoupled. Thus, the random variate model corresponding to any one of these four processes could be changed without altering the generated sequence of random variates corresponding to the other three processes.

Consistency Checks

Beyond the modifications in Examples 3.2.8 and 3.2.9, some job-type-specific statistics gathering needs to be added in `main` to complete the modification of program `ssq2` to accommodate multiple job types. If these modifications are made correctly, with *d.dd* precision the steady-state statistics that will be produced are

$\bar{r}$	$\bar{w}$	$\bar{d}$	$\bar{s}$	$\bar{l}$	$\bar{q}$	$\bar{x}$
2.40	7.92	5.92	2.00	3.30	2.47	0.83

The details are left in Exercise 3.2.4. How do we know these values are correct?

In addition to $\bar{w} = \bar{d} + \bar{s}$ and $\bar{l} = \bar{q} + \bar{x}$, the three following intuitive consistency checks give us increased confidence in these (estimated) steady-state results:

- Both job types have an average service time of 2.0, so that $\bar{s}$ should be 2.00. The corresponding service rate is 0.5.
- The arrival rate of job types 0 and 1 are 1/4 and 1/6 respectively. Intuitively, the net arrival rate should then be $1/4 + 1/6 = 5/12$ which corresponds to $\bar{r} = 12/5 = 2.40$.
- The steady-state utilization should be the ratio of the arrival rate to the service rate, which is $(5/12)/(1/2) = 5/6 \cong 0.83$.

3.2.3 EXERCISES

Exercise 3.2.1 (*a*) Construct the $a = 16807$ version of the table in Example 3.2.6. (*b*) What is the $O(\cdot)$ time complexity of the algorithm you used?

Exercise 3.2.2[a] (a) Prove that if m is prime, $1 \le a \le m - 1$, and $a^* = a^{m-2} \bmod m$ then

$$a^* a \bmod m = 1.$$

Now define

$$g(x) = ax \bmod m \qquad \text{and} \qquad g^*(x) = a^* x \bmod m$$

for all $x \in \mathcal{X}_m = \{1, 2, \ldots, m - 1\}$. ($b$) Prove that the functions $g(\cdot)$ and $g^*(\cdot)$ generate the same sequence of states, except in *opposite* orders. (c) Comment on the implication of this relative to full period multipliers. (d) If $m = 2^{31} - 1$ and $a = 48271$ what is a^*?

Exercise 3.2.3 Modify program `ssq2` as suggested in Example 3.2.7 to create two programs that differ only in the function `GetService`. For one of these programs, use the function as implemented in Example 3.2.7; for the other program, use

```
double GetService(void)
{
   SelectStream(2);                                  /* this line is new */
   return (Uniform(0.0, 1.5) + Uniform(0.0, 1.5));
}
```

(a) For both programs verify that *exactly* the same average interarrival time is produced (print the average with *d.dddddd* precision). Note that the average service time is approximately the same in both cases, as is the utilization, yet the service nodes statistics $\bar{w}$, $\bar{d}$, $\bar{l}$, and $\bar{q}$ are different. (b) Why?

Exercise 3.2.4 Modify program `ssq2` as suggested in Examples 3.2.8 and 3.2.9. (a) What proportion of processed jobs are type 0? (b) What are $\bar{w}$, $\bar{d}$, $\bar{s}$, $\bar{l}$, $\bar{q}$, and $\bar{x}$ for each job type? (c) What did you do to convince yourself that your results are valid? (d) Why are $\bar{w}$, $\bar{d}$, and $\bar{s}$ the same for both job types but $\bar{l}$, $\bar{q}$, and $\bar{x}$ are different?

Exercise 3.2.5 Prove Theorem 3.2.1.

Exercise 3.2.6 Same as Exercise 3.2.3, but using the `GetService` function in Example 3.1.4 instead of the `GetService` function in Exercise 3.2.3.

Exercise 3.2.7 Suppose there are three job types arriving independently to a single-server service node. The interarrival times and service times have the following characterization

job type	interarrival times	service times
0	*Exponential*(4.0)	*Uniform*(0.0, 2.0)
1	*Exponential*(6.0)	*Uniform*(1.0, 2.0)
2	*Exponential*(8.0)	*Uniform*(1.0, 5.0)

(a) What is the proportion of processed jobs for each type? (b) What are $\bar{w}$, $\bar{d}$, $\bar{s}$, $\bar{l}$, $\bar{q}$, and $\bar{x}$ for each job type? (c) What did you do to convince yourself that your results are valid? (Simulate at least 100 000 processed jobs.)

3.3 Discrete-Event Simulation Examples

In this section we will consider three discrete-event system models, each of which is an extension of a model considered previously. The three models are (*i*) a single-server service node *with immediate feedback*, (*ii*) a simple inventory system *with delivery lag*, and (*iii*) a single-server machine shop.

3.3.1 SINGLE-SERVER SERVICE NODE WITH IMMEDIATE FEEDBACK

We begin by considering an important extension of the single-server service node model first introduced in Section 1.2. Consistent with the following definition (based on Definition 1.2.1) the extension is *immediate feedback* — the possibility that the service a job just received was incomplete or otherwise unsatisfactory and, if so, the job feeds back to once again request service.

Definition 3.3.1 A single-server service node *with immediate feedback* consists of a server plus its queue with a feedback mechanism, as illustrated in Figure 3.3.1.

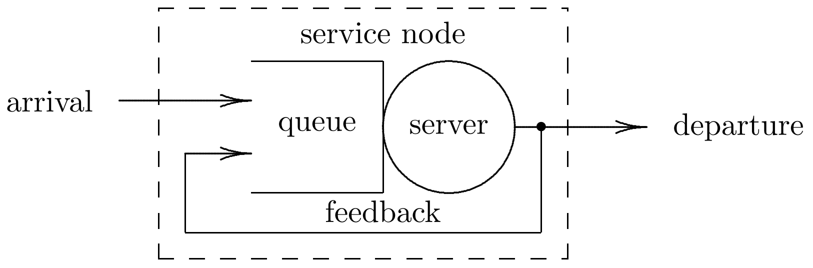


Figure 3.3.1. Single-server service node with immediate feedback.

Jobs arrive at the service node, generally at random, seeking service. When service is provided, the time involved is generally also random. At the completion of service, jobs either depart the service node (forever) or immediately feed back and once again seek service. The service node operates as follows: as each job arrives, if the server is busy then the job enters the queue, else the job immediately enters service; as each job completes service, either a departure or a feedback occurs, generally at random. When a *departure* occurs, if the queue is empty then the server becomes idle, else another job is selected from the queue to immediately enter service. When a *feedback* occurs, if the queue is empty then the job immediately re-enters service, else the job enters the queue, after which one of the jobs in the queue is selected to immediately enter service. At any instant in time, the state of the server will either be busy or idle and the state of the queue will be either empty or not empty. If the server is idle then the queue must be empty; if the queue is not empty then the server must be busy.

Note the distinction between the two events "completion of service" and "departure". If there is no feedback these two events are equivalent; if feedback is possible then it is important to make a distinction. When the distinction is important, the completion-of-service event is more fundamental because at the completion of service, either a departure event or a feedback event then occurs. This kind of "which event comes first" causal reasoning is important at the conceptual model-building level.

Model Considerations

When feedback occurs we assume that the job joins the queue (if any) consistent with the queue discipline. For example, if the queue discipline is FIFO then a fed-back job would receive no priority; it would join the queue at the end, in effect becoming indistinguishable from an arriving job. Of course, other feedback queue disciplines are possible, the most common of which involves assigning a priority to jobs that are fed back. If feedback is possible the default assumption in this book is that a fed-back job will join the queue consistent with the queue discipline and a new service time will be required, independent of any prior service provided. Similarly, the default assumption is that the decision to depart or feed back is random with *feedback probability* β, as illustrated in Figure 3.3.2.

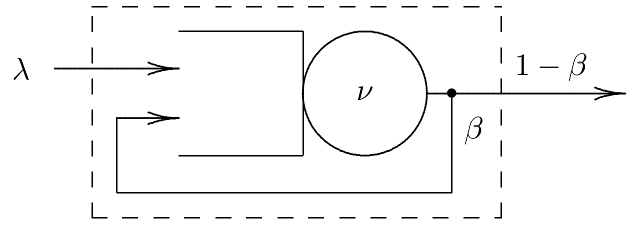


Figure 3.3.2. Single-server service node with feedback probability β.

In addition to β, the other two parameters that characterize the stochastic behavior of a single-server service node with immediate feedback are the *arrival rate* λ and the *service rate* ν. Consistent with Definition 1.2.5, $1/\lambda$ is the average interarrival time and $1/\nu$ is the average service time.*

As each job completes service, it departs the service node with probability $1 - \beta$ or feeds back with probability β. Consistent with this model, feedback is independent of past history and so a job may feed back more than once. Indeed, in theory, a job may feed back arbitrarily many times — see Exercise 3.3.1. Typically β is close to 0.0 indicating that feedback is a rare event. This is not a universal assumption, however, and so a well written discrete-event simulation program should accommodate any probability of feedback in the range $0.0 \leq \beta < 1.0$. At the computational model-building level, feedback can be modeled with a boolean-valued function as illustrated.

```
int GetFeedback(double beta)                  /* use 0.0 <= beta < 1.0 */
{
   SelectStream(2);                 /* use rngs stream 2 for feedback */
   if (Random() < beta)
     return (1);                                 /* feedback occurs */
   else
     return (0);                                 /* no feedback     */
}
```

* The use of the symbol ν to denote the service rate is non-standard. Instead, the usual convention is to use the symbol μ. See Section 8.5 for more discussion of arrival rates, service rates, and our justification for the use of ν in place of μ.

Statistical Considerations

If properly interpreted, the mathematical variables and associated definitions in Section 1.2 remain valid if immediate feedback is possible. The interpretation required is that the index $i = 1, 2, 3, \ldots$ counts jobs that enter the service node; once indexed in this way, a fed-back job is not counted again. Because of this indexing all the job-averaged statistics defined in Section 1.2 remain valid *provided* delay times, wait times, and service times are incremented each time a job is fed back. For example, the average wait is the sum of the waits experienced by all the jobs that enter the service node divided by the number of such jobs; each time a job is fed back it contributes an additional wait to the sum of waits, but it does *not* cause the number of jobs to be increased. Similarly, the time-averaged statistics defined in Section 1.2 also remain valid if feedback is possible.

The key feature of immediate feedback is that jobs from outside the system are merged with jobs from the feedback process. In this way, the (steady-state) request-for-service rate is larger than λ by the positive additive factor $\beta\bar{x}\nu$. As illustrated later in Example 3.3.2, if there is no corresponding increase in service rate this increase in the request-for-service rate will cause job-averaged and time-averaged statistics to increase from their non-feedback values. This increase is intuitive — if you are entering a grocery store and the check-out queues are already long, you certainly do not want to see customers re-entering these queues because they just realized they were short-changed at check-out or forgot to buy a gallon of milk.

Note that indexing by arriving jobs will cause the average service time $\bar{s}$ to increase as the feedback probability increases. In this case do not confuse $\bar{s}$ with the reciprocal of the service rate; $1/\nu$ is the (theoretical) average service time *per service request*, irrespective of whether that request is by an arriving job or by a fed back job.

Algorithm and Data Structure Considerations

Example 3.3.1 Consider the following arrival times, service times, and completion times for the first 9 jobs entering a single-server FIFO service node with immediate feedback. (For simplicity all times are integers.)

job index	:	1	2	3	4	5	·	6	·	7	8	·	9	···
arrival/feedback	:	1	3	4	7	10	**13**	14	**15**	19	24	**26**	30	···
service	:	9	3	2	4	7	5	6	3	4	6	3	7	···
completion	:	10	**13**	**15**	19	**26**	31	37	**40**	44	50	53	**60**	···

The bold-face times correspond to jobs that were fed back. For example, the third job completed service at time 15 and immediately fed back. At the computational level, note that some algorithm and data structure is necessary to insert fed back jobs into the arrival stream. That is, an inspection of the yet-to-be-inserted feedback times (15, 26) reveals that the job fed back at time 15 must be inserted in the arrival stream after job 6 (which arrived at time 14) and before job 7 (which arrived at time 19).

The reader is encouraged to extend the specification model of a single-server service node in Section 1.2 to account for immediate feedback. Then, extend this model at the computational level by starting with program `ssq2` and using a different `rngs` stream for each stochastic process. Example 3.3.1 provides insight into an algorithmic extension and associated data structure that can be used to accomplish this.

Example 3.3.2 Program `ssq2` was modified to account for immediate feedback. Consistent with the stochastic modeling assumptions in Example 3.1.3, the arrival process has *Exponential*(2.0) random variate interarrivals corresponding to a fixed arrival rate of $\lambda = 0.50$, the service process has *Uniform*(1.0, 2.0) random variate service times corresponding to a fixed service rate of $\nu \cong 0.67$ and the feedback probability is $0.0 \leq \beta \leq 1.0$. To illustrate the effect of feedback, the modified program was used to simulate the operation of a single-server service node with nine different values of levels of feedback varied from $\beta = 0.0$ (no feedback) to $\beta = 0.20$. In each case 100 000 arrivals were simulated. Utilization $\bar{x}$ as a function of β is illustrated on the left-hand-side of Figure 3.3.3; the average number in the queue $\bar{q}$ as a function of β is illustrated on the right-hand-side.

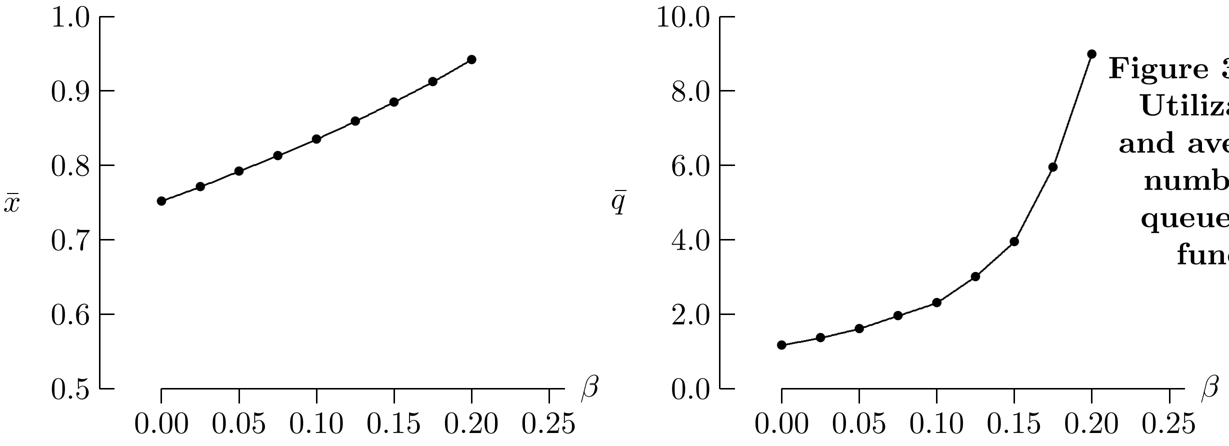


Figure 3.3.3. Utilization and average number in queue as a function of β.

As the probability of feedback increases, the utilization increases from the steady-state value of $\bar{x} = 0.75$ when there is no feedback toward the maximum possible value $\bar{x} = 1.0$. If this $\bar{x}$ versus β figure is extrapolated, it appears that saturation ($\bar{x} = 1.0$) is achieved as $\beta \to 0.25$.

Flow Balance and Saturation

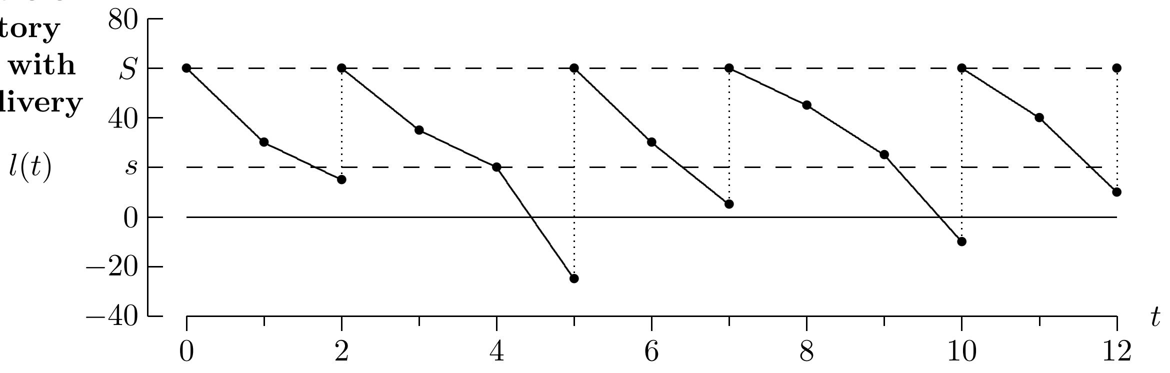
The observation that saturation occurs as β approaches 0.25 is an important consistency check based on steady-state *flow balance* considerations. That is, jobs flow *into* the service node at the average rate of λ. To remain flow balanced jobs must flow *out* of the service node at the same average rate. Because the average rate at which jobs flow out of the service node is $\bar{x}(1-\beta)\nu$, flow balance is achieved when $\lambda = \bar{x}(1-\beta)\nu$. Saturation is achieved when $\bar{x} = 1$; this happens as $\beta \to 1 - \lambda/\nu = 0.25$. Consistent with saturation, in Example 3.3.2 we see that the average number in the queue increases dramatically as β increases, becoming effectively infinite as $\beta \to 0.25$.

3.3.2 SIMPLE INVENTORY SYSTEM WITH DELIVERY LAG

The second discrete-event system model we will consider in this section represents an important extension of the periodic review simple inventory system model first introduced in Section 1.3. The extension is *delivery lag* (or *lead time*) — an inventory replacement order placed with the supplier will not be delivered immediately; instead, there will be a lag between the time an order is placed and the time the order is delivered. Unless stated otherwise, this lag is assumed to be random and independent of the amount ordered.

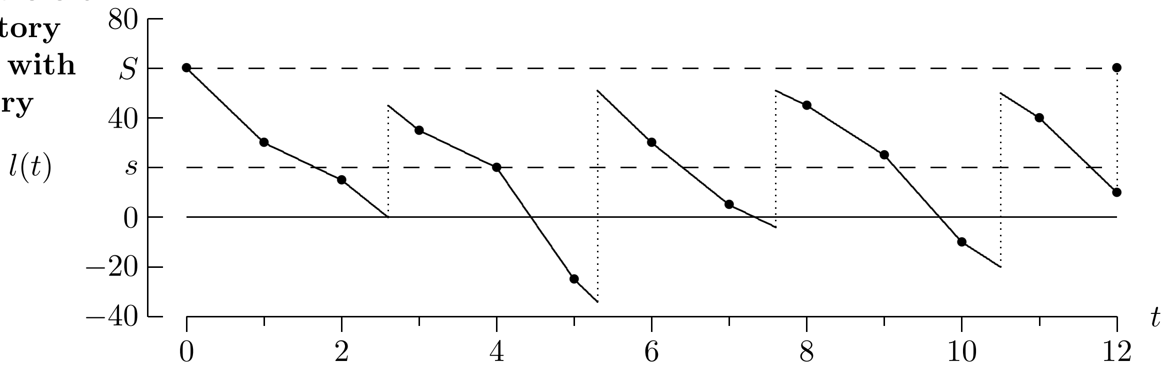
If there are no delivery lags then a typical inventory time history looks like the one in Section 1.3, reproduced in Figure 3.3.4 for convenience, with jump discontinuities possible only at the (integer-valued) times of inventory review.

Figure 3.3.4. Inventory levels with no delivery lags.



If delivery lags are possible, a typical inventory time history would have jump discontinuities at arbitrary times, as illustrated in Figure 3.3.5. (The special-case order at the terminal time $t = 12$ is assumed to be delivered with zero lag.)

Figure 3.3.5. Inventory levels with delivery lags.



Unless stated otherwise, we assume that any order placed at the beginning of a time interval (at times $t = 2, 5, 7,$ and 10 in this case) will be delivered before the end of the time interval. With this assumption there is *no* change in the simple inventory system model at the specification level (see Algorithm 1.3.1). There is, however, a significant change in how the system statistics are computed. For those time intervals in which a delivery lag occurs, the time-averaged holding and shortage integrals in Section 1.3 must be modified.

Statistical Considerations

As in Section 1.3, l_{i-1} denotes the inventory level at the beginning of the i^{th} time interval (at $t = i-1$) and d_i denotes the amount of demand during this interval. Consistent with the model in Section 1.3, the demand rate is assumed to be constant between review times. Given d_i, l_{i-1}, and this assumption, there are two cases to consider.

If $l_{i-1} \geq s$ then, because no order is placed at $t = i - 1$, the inventory decreases at a constant rate throughout the interval with no "jump" in level. The inventory level at the end of the interval is $l_{i-1} - d_i$. In this case, the equations for $\bar{l}_i^+$ and $\bar{l}_i^-$ in Section 1.3 remain valid (with l_{i-1} in place of l'_{i-1}.)

If $l_{i-1} < s$ then an order is placed at $t = i - 1$ which later causes a jump in the inventory level when the order is delivered. In this case the equations for $\bar{l}_i^+$ and $\bar{l}_i^-$ in Section 1.3 must be modified. That is, if $l_{i-1} < s$ then an order for $S - l_{i-1}$ items is placed at $t = i - 1$ and a *delivery lag* $0 < \delta_i < 1$ occurs during which time the inventory level drops at a constant rate to $l_{i-1} - \delta_i d_i$. When the order is delivered at $t = i - 1 + \delta_i$ the inventory level jumps to $S - \delta_i d_i$. During the remainder of the interval the inventory level drops at the same constant rate to its final level $S - d_i$ at $t = i$. All of this is summarized with the observation that, in this case, the inventory-level time history during the i^{th} time interval is defined by one vertical and two parallel lines, as illustrated in Figure 3.3.6.*

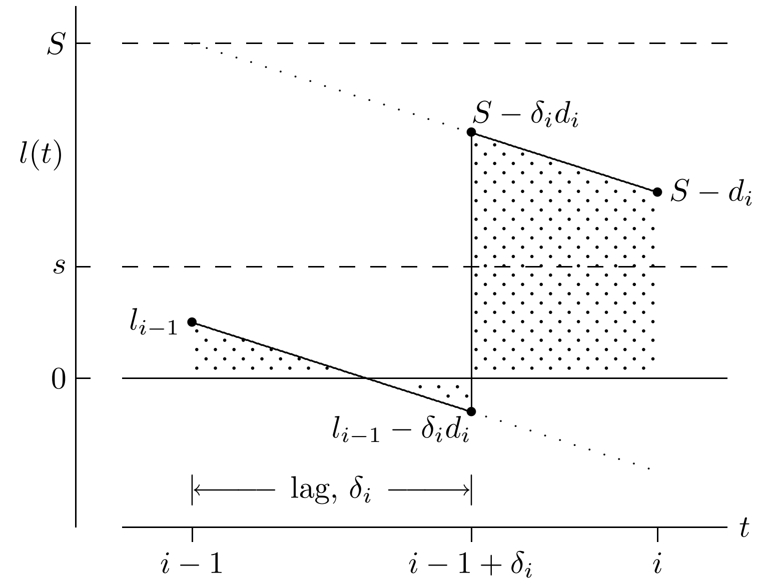


Figure 3.3.6. Inventory level during time interval i when an order is placed.

Depending on the location of the four line-segment endpoints indicated by •'s, with each location measured relative to the line $l(t) = 0$, either triangular or trapezoidal figures will be generated. To determine the time-averaged holding level $\bar{l}_i^+$ and time-averaged shortage level $\bar{l}_i^-$ (see Definition 1.3.3), it is necessary to determine the area of each figure. The details are left as an exercise.

* Note that $\delta_i d_i$ must be integer-valued.

Consistency Checks

When system models are extended, it is fundamentally important to verify that the extended model is consistent with the parent model (the model before extension). This is usually accomplished by setting system parameters to special values. For example, if the feedback probability is set to zero an extended computational model that simulates a single-server service node *with* feedback reduces to a parent computational model of a single-server service node *without* feedback. At the computational level the usual way to make this kind of consistency check is to compare output system statistics and verify that, with the extension removed, the output statistics produced by the extended model agree with the output statistics produced by the parent model. Use of the library `rngs` facilitates this kind of comparison. In addition to these "extension removal" consistency checks, it is also good practice to check for intuitive "small-perturbation" consistency. For example, if the feedback probability is small, but non-zero, the average number in the queue should be slightly larger than its feedback-free value. The following example applies this idea to a simple inventory system model with delivery lag.

Example 3.3.3 For a simple inventory system with delivery lag we adopt the convention that δ_i is defined for all $i = 1, 2, \ldots, n$ with $\delta_i = 0.0$ if and only if there is no order placed at the beginning of the i^{th} time interval (that is, if $l_{i-1} \geq s$). If an order is placed then $0.0 < \delta_i < 1.0$. With this convention the stochastic time evolution of a simple inventory system with delivery lag is driven by the two n-point stochastic sequences $d_1, d_2, \ldots, d_n$ and $\delta_1, \delta_2, \ldots, \delta_n$. The simple inventory system is *lag-free* if and only if $\delta_i = 0.0$ for all $i = 1, 2, \ldots, n$; if $\delta_i > 0.0$ for at least one i then the system is not lag-free. Relative to the five system statistics in Section 1.3, if the inventory parameters (S, s) are fixed then, even if the delivery lags are small, the following points are valid.

- The average order $\bar{o}$, average demand $\bar{d}$, and relative frequency of setups $\bar{u}$ are exactly the same whether the system is lag-free or not.
- Compared to the lag-free value, if the system is not lag-free then the time-averaged holding level $\bar{l}^+$ will decrease.
- Compared to the lag-free value, if the system is not lag-free then the time-averaged shortage level $\bar{l}^-$ will either remain unchanged or it will increase.

At the computational level these three points provide valuable consistency checks for a simple inventory system discrete-event simulation program.

Delivery Lag

If the statistics-gathering logic in program `sis2` is modified to be consistent with the previous discussion, then the resulting program will provide a computational model of a simple inventory system with delivery lag. To complete this modification, a stochastic model of delivery lag is needed. In the absence of information to the contrary, we assume that each delivery lag is an independent *Uniform*$(0, 1)$ random variate.

Example 3.3.4 Program `sis2` was modified to account for *Uniform*$(0, 1)$ random variate delivery lags, independent of the size of the order. As an extension of the automobile dealership example (Example 3.1.7), this modified program was used to study the effect of delivery lag. That is, with $S = 80$ the average weekly cost was computed for a range of inventory threshold values s between 20 and 60. To avoid clutter only steady-state cost estimates (based on $n = 10\,000$ time intervals) are illustrated. For comparison, the corresponding lag-free cost values from Example 3.1.7 are also illustrated in Figure 3.3.7.

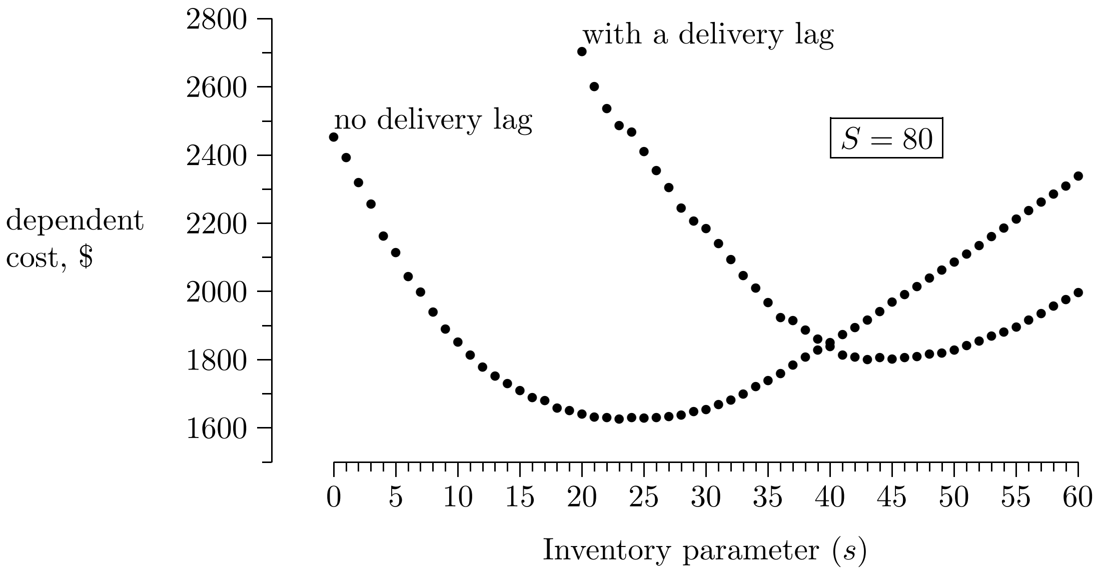


Figure 3.3.7. Effect of delivery lag on dependent cost.

Figure 3.3.7 shows that the effect of delivery lag is profound; the U-shaped cost-versus-s curve is shifted up and to the right. Because of this shift the optimum (minimum cost) value of s is increased by approximately 20 automobiles and the corresponding minimum weekly cost is increased by almost \$200.*

The shift in the U-shaped curve in Example 3.3.4 is consistent with the second and third points in Example 3.3.3. That is, delivery lags cause $\bar{l}^+$ to decrease and $\bar{l}^-$ to increase (or remain the same). Because the holding cost coefficient is $C_{\text{hold}} = \$25$ and the shortage cost coefficient is $C_{\text{short}} = \$700$ (see Example 1.3.5), delivery lags will cause holding costs to decrease a little for all values of s and will cause shortage costs to increase a lot, but only for small values of s. The shift in the U-shaped curve is the result.

Examples 3.3.2 and 3.3.4 present results corresponding to significant extensions of the two canonical system models used throughout this book. The reader is strongly encouraged to work through the details of these extensions at the computational level and reproduce the results in Examples 3.3.2 and 3.3.4.

* Because of the dramatic shift in the optimum value of s from the lag-free value of $s \cong 23$ to the with-lag value of $s \cong 43$, we see that the optimal value of s is *not* robust with respect to model assumptions about the delivery lag.

3.3.3 SINGLE-SERVER MACHINE SHOP

The third discrete-event simulation model considered in this section is a single-server machine shop. This is a simplified version of the multi-server machine shop model used in Section 1.1 to illustrate model building at the conceptual, specification, and computational level (see Example 1.1.1).

A single-server machine shop model is essentially identical to the single-server service node model first introduced in Section 1.2, except for one important difference. The service node model is *open* in the sense that an effectively infinite number of jobs are available to arrive from "outside" the system and, after service is complete, return to the "outside". In contrast, the machine shop model is *closed* because there are a finite number of machines (jobs) that are part of the system — as illustrated in Figure 3.3.8, there is no "outside".

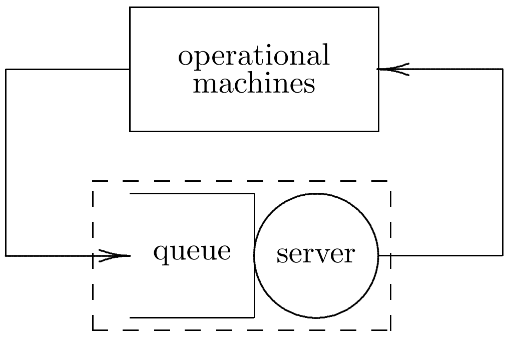


Figure 3.3.8. Single-server machine shop system diagram.

In more detail, there is a finite population of statistically identical machines, all of which are initially in an *operational* state (so the server is initially idle and the queue is empty). Over time these machines fail, independently, at which time they enter a *broken* state and request repair service at the single-server service node.* Once repaired, a machine immediately re-enters an operational state and remains in this state until it fails again. Machines are repaired in the order in which they fail, without interruption. Correspondingly, the queue discipline is FIFO, non-preemptive and conservative. There is no feedback.

To make the single-server machine shop model specific, we assume that the service (repair) time is a *Uniform*(1.0, 2.0) random variate, that there are M machines, and that the time a machine spends in the operational state is an *Exponential*(100.0) random variate. All times are in hours. Based on 100 000 simulated machine failures, we want to estimate the steady-state time-averaged number of operational machines and the server utilization as a function of M.

* Conceptually the machines move along the network arcs indicated, from the operational pool into and out of service and then back to the operational pool. In practice, the machines are usually stationary and the server moves to the machines. The time, if any, for the server to move from one machine to another is part of the service time.

Program ssms

Program `ssms` simulates the single-server machine shop described in this section. This program is similar to program `ssq2`, but with two important differences.

- The library `rngs` is used to provide an independent source of random numbers to both the simulated machine failure process and the machine repair process.
- The failure process is defined by the array `failure` which represents the time of next failure for each of the M machines.

The time-of-next-failure list (array) is not maintained in sorted order and so it must be searched completely each time a machine failure is simulated. The efficiency of this $O(M)$ search could be a problem for large M. Exercise 3.3.7 investigates computational efficiency improvements associated with an alternative algorithm and associated data structure.

Example 3.3.5 Because the time-averaged number in the service node $\bar{l}$ represents the time-averaged number of broken machines, $M - \bar{l}$ represents the time-averaged number of operational machines. Program `ssms` was used to estimate $M - \bar{l}$ for values of M between 20 and 100, as illustrated in Figure 3.3.9.

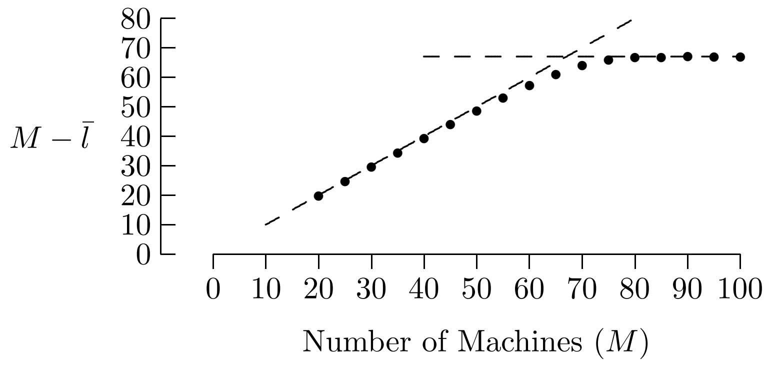


Figure 3.3.9. Time-averaged number of operational machines as a function of the number of machines.

As expected, for small values of M the time-averaged number of operational machines is essentially M. This is consistent with low server utilization and a correspondingly small value of $\bar{l}$. The angled dashed line indicates the ideal situation where all machines are continuously operational (i.e., $\bar{l} = 0$). Also, as expected, for large values of M the time-averaged number of operational machines is essentially constant, independent of M. This is consistent with a saturated server (utilization of 1.0) and a correspondingly large value of $\bar{l}$. The horizontal dashed line indicates that this saturated-server constant value is approximately 67.

Distribution Parameters

The parameters used in the distributions in the models presented in this section, e.g., $\mu = 2.0$ for the average interarrival time and $\beta = 0.20$ for the feedback probability in the single-server service node with feedback, have been drawn from thin air. This has been done in order to focus on the simulation modeling and associated algorithms. Chapter 9 on "Input Modeling" focuses on techniques for estimating realistic parameters from data.

3.3.4 EXERCISES

Exercise 3.3.1 Let β be the probability of feedback and let the integer-valued random variable X be the number of times a job feeds back. (a) For $x = 0, 1, 2, \ldots$ what is $\Pr(X = x)$? (b) How does this relate to the discussion of acceptance/rejection in Section 2.3?

Exercise 3.3.2[a] (a) Relative to Example 3.3.2, based on 1 000 000 arrivals, generate a table of $\bar{x}$ and $\bar{q}$ values for β from 0.00 to 0.24 in steps of 0.02. (b) What data structure did you use and why? (c) Discuss how external arrivals are merged with fed back jobs.

Exercise 3.3.3 For the model of a single-server service node with feedback presented in this section, there is nothing to prevent a fed-back job from colliding with an arriving job. Is this a model deficiency that needs to be fixed and, if so, how would you do it?

Exercise 3.3.4[a] Modify program `ssq2` to account for a *finite* service node capacity. (a) For capacities of 1, 2, 3, 4, 5, and 6 construct a table of the estimated steady-state probability of rejection. (b) Also, construct a similar table if the service time distribution is changed to be *Uniform*(1.0, 3.0). (c) Comment on how the probability of rejection depends on the service process. (d) How did you convince yourself these tables are correct?

Exercise 3.3.5[a] Verify that the results in Example 3.3.4 are correct. Provide a table of values corresponding to the figure in this example.

Exercise 3.3.6 (a) Relative to Example 3.3.5, construct a figure or table illustrating how $\bar{x}$ (utilization) depends on M. (b) If you extrapolate linearly from small values of M, at what value of M will saturation ($\bar{x} = 1$) occur? (c) Can you provide an empirical argument or equation to justify this value?

Exercise 3.3.7[a] In program `ssms` the time-of-next-failure list (array) is not maintained in sorted order and so the list must be searched completely each time another machine failure is simulated. As an alternative, implement an algorithm and associated sorted data structure to determine if a significant improvement in computational efficiency can be obtained. (You may need to simulate a huge number of machine failures to get an accurate estimate of computational efficiency improvement.)

Exercise 3.3.8 (a) Relative to Example 3.3.2, compare a FIFO queue discipline with a priority queue discipline where fed-back jobs go the head of the queue (i.e., re-enter service immediately). (b) Is the following conjecture true or false: although statistics for the fed-back jobs change, system statistics do not change?

Exercise 3.3.9[a] (a) Repeat Exercise 3.3.7 using $M = 120$ machines, with the time a machine spends in the operational state increased to an *Exponential*(200.0) random variate. (b) Use $M = 180$ machines with the time spent in the operational increased accordingly. (c) What does the $O(\cdot)$ computational complexity of your algorithm seem to be?