CHAPTER 1

MODELS

The modeling approach in this book is based on the use of a general-purpose programming language for model implementation at the computational level. The alternative approach is to use a special-purpose simulation language; for a survey of several such languages, see Appendix A.

Sections

1.1. Introduction 2
1.2. A Single-Server Queue (program `ssq1`) 12
1.3. A Simple Inventory System (program `sis1`) 26

This chapter presents an introduction to discrete-event simulation, with an emphasis on model building. The focus in Section 1.1 is on the multiple steps required to construct a discrete-event simulation model. By design, the discussion in this section is at a high level of generality, with few details. In contrast, two specific discrete-event system models are presented in Sections 1.2 and 1.3, with a significant amount of detail. A single-server queue model is presented in Section 1.2 and a simple inventory system model is presented in Section 1.3. Both of these models are of fundamental importance because they serve as a basis for a significant amount of material in later chapters.

Although the material in this chapter also can be found in other modeling and simulation texts, there is a relatively novel emphasis on model building at the conceptual, specification and computational levels. Moreover, in Sections 1.2 and 1.3 there is a significant amount of notation, terminology, and computational philosophy which extends to subsequent chapters. For these reasons, this chapter is important to any reader of the book, even those already familiar with the rudiments of discrete-event simulation.

1.1 Introduction

This book is the basis for a first course on *discrete-event simulation.* That is, the book provides [illegible] By definition, the nature of discrete-event simulation is that [illegible]. [illegible]. Consistent with that observation, the [illegible].

1.1.1 MODEL CHARACTERIZATION

Briefly, [illegible] with the special discrete-event property that the [illegible] (see Definition 1.1.1). But what does that mean?

A system model is *deterministic* or *stochastic*. A deterministic system model has no stochastic (random) components. For example, provided the conveyor belt and machine never fail, a model of a constant velocity conveyor belt feeding parts to a machine with a constant service time is deterministic. At some level of detail, however, all systems have some stochastic components; machines fail, people are not robots, service requests occur at random, etc. An attractive feature of discrete-event simulation is that stochastic components can be accommodated, usually without a dramatic increase in the complexity of the system model at the computational level.

A system model is *static* or *dynamic*. A static system model is one in which time is not a significant variable. For example, if three million people play the state lottery this week, what is the probability that there will be at least one winner? A simulation program written to answer this question should be based on a static model; when during the week these three million people place their bets is not significant. If, however, we are interested in the probability of no winners in the next four weeks, then this model needs to be dynamic. That is, experience has revealed that each week there are no winners, the number of players in the following week increases (because the pot grows). When this happens, a dynamic system model must be used because the probability of at least one winner will increase as the number of players increases. The passage of time always plays a significant role in dynamic models.

A dynamic system model is *continuous* or *discrete*. Most of the traditional dynamic systems studied in classical mechanics have state variables that evolve continuously. A particle moving in a gravitational field, an oscillating pendulum, or a block sliding on an inclined plane are examples. In each of these cases the motion is characterized by one or more differential equations which model the continuous time evolution of the system. In contrast, the kinds of queuing, machine repair and inventory systems studied in this book are discrete because the state of the system is a *piecewise-constant* function of time. For example, the number of jobs in a queuing system is a natural state variable that only changes value at those discrete times when a job arrives (to be served) or departs (after being served).

The characterization of a system model can be summarized by a tree diagram that starts at the system model root and steps left or right at each of the three levels, as illustrated in Figure 1.1.1.

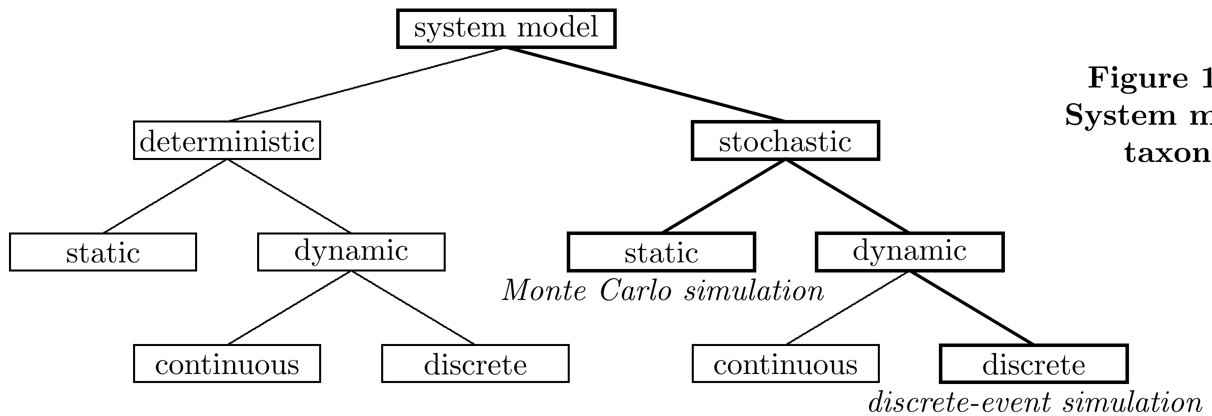


Figure 1.1.1. System model taxonomy.

As summarized by Definition 1.1.1, the system model characterized by the right-most branch of this tree is of primary interest in this book.

Definition 1.1.1 A *discrete-event simulation model* is defined by three attributes:

- *stochastic* — at least some of the system state variables are random;
- *dynamic* — the time evolution of the system state variables is important;
- *discrete-event* — significant changes in the system state variables are associated with events that occur at discrete time instances only.

One of the other five branches of the system model tree is of significant, but secondary, interest in this book. A *Monte Carlo simulation model* is stochastic and static — at least some of the system state variables are random, but the time evolution (if any) of the system state variables is not important. Accordingly, the issue of whether time flows continuously or discretely is not relevant.

Because of space constraints, the remaining four branches of the system model tree are not considered. That is, there is no material about deterministic systems, static or dynamic, or about stochastic dynamic systems that evolve continuously in time.

1.1.2 MODEL DEVELOPMENT

It is naive to think that the process of developing a discrete-event simulation model can be reduced to a simple sequential algorithm. As an instructional device, however, it is useful to consider two algorithms that outline, at a high level, how to develop a discrete-event simulation model (Algorithm 1.1.1) and then conduct a discrete-event simulation study (Algorithm 1.1.2).

Algorithm 1.1.1 If done well, a typical discrete-event simulation model will be developed consistent with the following six steps. Steps (2) through (6) are typically iterated, perhaps many times, until a (hopefully) valid computational model, a computer program, has been developed.

(1) Determine the *goals* and *objectives* of the analysis once a system of interest has been identified. These goals and objectives are often phrased as simple Boolean decisions (e.g., should an additional queuing network service node be added) or numeric decisions (e.g., how many parallel servers are necessary to provide satisfactory performance in a multi-server queuing system). Without specific goals and objectives, the remaining steps lack meaning.

(2) Build a *conceptual* model of the system based on (1). What are the *state variables*, how are they interrelated and to what extent are they dynamic? How comprehensive should the model be? Which state variables are important; which have such a negligible effect that they can be ignored? This is an intellectually challenging but rewarding activity that should not be avoided just because it is hard to do.

(3) Convert the conceptual model into a *specification* model. If this step is done well, the remaining steps are made much easier. If instead this step is done poorly (or not at all) the remaining steps are probably a waste of time. This step typically involves collecting and statistically analyzing data to provide the input models that drive the simulation. In the absence of such data, the input models must be constructed in an ad hoc manner using stochastic models believed to be representative.

(4) Turn the specification model into a *computational* model, a computer program. At this point, a fundamental choice must be made — to use a general-purpose programming language or a special-purpose simulation language. For some this is a religious issue not subject to rational debate.

(5) *Verify*. As with all computer programs, the computational model should be consistent with the specification model — did we implement the computational model correctly? This verification step is not the same as the next step.

(6) *Validate*. Is the computational model consistent with the system being analyzed — did we build the right model? Because the purpose of simulation is insight, some (including the authors) would argue that the *act* of developing the discrete-event simulation model — steps (2), (3), and (4) — is frequently as important as the tangible *product*. However, given the blind faith many people place in any computer generated output the validity of a discrete-event simulation model is always fundamentally important. One popular non-statistical, Turing-like technique for model validation is to place actual system output alongside similarly formatted output from the computational model. This output is then examined by an expert familiar with the system. Model validation is indicated if the expert is not able to determine which is the model output and which is the real thing. Interactive computer graphics (animation) can be very valuable during the verification and validation steps.

Example 1.1.1 The following *machine shop* model helps illustrate the six steps in Algorithm 1.1.1. A new machine shop has 150 identical machines; each operates continuously, 8 hours per day, 250 days per year until failure. Each machine operates independently of all the others. As machines fail they are repaired, in the order in which they fail, by a service technician. As soon as a failed machine is repaired, it is put back into operation. Each machine produces a net income of $20 per hour of operation. All service technicians are hired at once, for 2 years, at the beginning of the 2-year period with an annual salary expense of $52,000. Because of vacations, each service technician only works 230 8-hour days per year. By agreement, vacations are coordinated to maximize the number of service technicians on duty each day. How many service technicians should be hired?

(1) The objective seems clear — to find the number of service technicians for which the profit is maximized. One extreme solution is to hire one technician for each machine; this produces a huge service technician overhead but maximizes income by minimizing the amount of machine down-time. The other extreme solution is to hire just one technician; this minimizes overhead at the potential expense of large down-times and associated loss of income. In this case, neither extreme is close to optimal for typical failure and repair times.

(2) A reasonable conceptual model for this system can be expressed in terms of the state of each machine (failed or operational) and each service technician (busy or idle). These state variables provide a high-level description of the system at any time.

(3) To develop a specification model, more information is needed. Machine failures are random events; what is known (or can be assumed) about the time between failures for these machines? The time to repair a machine is also random; what, for example, is the distribution of the repair time? In addition, to develop the associated specification model some systematic method must be devised to simulate the time evolution of the system state variables.

(4) The computational model will likely include a simulation clock data structure to keep track of the current simulation time, a queue of failed machines and a queue of available service technicians. Also, to characterize the performance of the system, there will be statistics gathering data structures and associated procedures. The primary statistic of interest here is the total profit associated with the machine shop.

(5) The computational model must be verified, usually by extensive testing. Verification is a software engineering activity made easier if the model is developed in a contemporary programming environment.

(6) The validation step is used to see if the verified computational model is a reasonable approximation of the machine shop. If the machine shop is already operational, the basis for comparison is clear. If, however, the machine shop is not yet operational, validation is based primarily on consistency checks. If the number of technicians is increased, does the time-averaged number of failed machines go down; if the average service time is increased, does the time-averaged number of failed machines go up?

System Diagrams

Particularly at the conceptual level, the process of model development can be facilitated by drawing system diagrams. Indeed, when asked to explain a system, our experience is that, instinctively, many people begin by drawing a system diagram. For example, consider this system diagram of the machine shop model in Example 1.1.1.

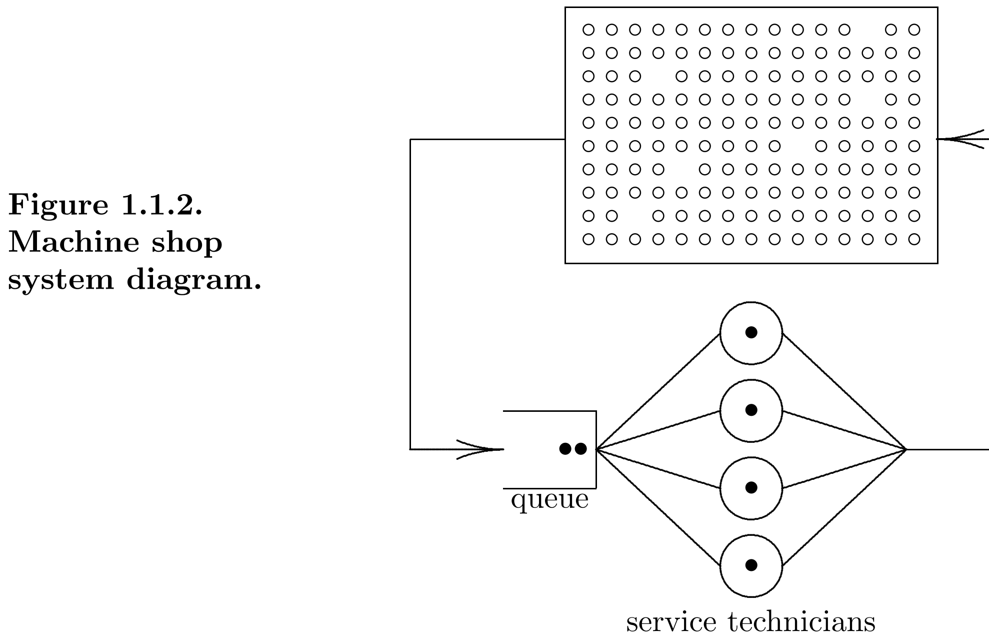


Figure 1.1.2. Machine shop system diagram.

The box at the top of Figure 1.1.2 represents the pool of machines. The composite object at the bottom of the figure represents the four service technicians and an associated single queue. Operational machines are denoted with a $\circ$ and broken machines with a $\bullet$. Conceptually, as machines break they change their state from operational ($\circ$) to broken ($\bullet$) and move along the arc on the left from the box at the top of the figure to the queue at the bottom of the figure. From the queue, a broken machine begins to be repaired as a service technician becomes available. As each broken machine is repaired, its state is changed to operational and the machine moves along the arc on the right, back to the pool of operational machines.*

As time evolves, there is a continual counter-clockwise circulation of machines from the pool at the top of Figure 1.1.2 to the service technicians at the bottom of the figure, and then back again. At the "snapshot" instant illustrated, there are six broken machines; four of these are being repaired and the other two are waiting in the queue for a service technician to become available.

* The movement of the machines to the servers is conceptual, as is the queue. In practice, the servers would move to the machines and there would not be a physical queue of broken machines.

In general, the application of Algorithm 1.1.1 should be guided by the following observations.

- Throughout the development process, the operative principle should always be to make every discrete-event simulation model as simple as possible, but never simpler. The goal is to capture only the relevant characteristics of the system. The dual temptations of (1) ignoring relevant characteristics or (2) including characteristics that are extraneous to the goals of the model, should be avoided.
- The actual development of a complex discrete-event simulation model will not be as sequential as Algorithm 1.1.1 suggests, particularly if the development is a team activity in which case some steps will surely be worked in parallel. The different characteristics of each step should always be kept clearly in mind avoiding, for example, the natural temptation to merge steps (5) and (6).
- There is an unfortunate tendency on the part of many to largely skip over steps (1), (2), and (3), jumping rapidly to step (4). Skipping these first three steps is an approach to discrete-event simulation virtually certain to produce large, inefficient, unstructured computational models that cannot be validated. Discrete-event simulation models should *not* be developed by those who like to think a little and then program a lot.

1.1.3 SIMULATION STUDIES

Algorithm 1.1.2 Following the successful application of Algorithm 1.1.1, use of the resulting computational model (computer program) involves the following steps.

(7) Design the simulation experiments. This is not as easy as it may seem. If there are a significant number of system parameters, each with several possible values of interest, then the combinatoric possibilities to be studied make this step a real challenge.

(8) Make production runs. The runs should be made systematically, with the value of all initial conditions and input parameters recorded along with the corresponding statistical output.

(9) Analyze the simulation results. The analysis of the simulation output is statistical in nature because discrete-event simulation models have stochastic (random) components. The most common statistical analysis tools (means, standard deviations, percentiles, histograms, correlations, etc.) will be developed in later chapters.

(10) Make decisions. Hopefully the results of step (9) will lead to decisions that result in actions taken. If so, the extent to which the computational model correctly predicted the outcome of these actions is always of great interest, particularly if the model is to be further refined in the future.

(11) Document the results. If you really did gain insight, summarize it in terms of specific observations and conjectures. If not, why did you fail? Good documentation facilitates the development (or avoidance) of subsequent similar system models.

Example 1.1.2 As a continuation of Example 1.1.1, consider the application of Algorithm 1.1.2 to a verified and validated machine shop model.

(7) Since the objective of the model is to determine the optimal number of service technicians to hire to maximize profit, the number of technicians is the primary system parameter to be varied from one simulation run to the next. Other issues also contribute to the design of the simulation experiments. What are the initial conditions for the model (e.g., are all machines initially operational)? For a fixed number of service technicians, how many replications are required to reduce the natural sampling variability in the output statistics to an acceptable level?

(8) If many production runs are made, management of the output results becomes an issue. A discrete-event simulation study can produce *a lot* of output files which consume large amounts of disk space if not properly managed. Avoid the temptation to archive "raw data" (e.g., a detailed time history of simulated machine failures). If this kind of data is needed in the future, it can always be reproduced. Indeed, the ability to reproduce previous results *exactly* is an important feature which distinguishes discrete-event simulation from other, more traditional, experimental sciences.

(9) The statistical analysis of simulation output often is more difficult than classical statistical analysis, where observations are assumed to be *independent*. In particular, time-sequenced simulation-generated observations are often correlated with one another, making the analysis of such data a challenge. If the current number of failed machines is observed each hour, for example, consecutive observations will be found to be significantly positively correlated. A statistical analysis of these observations based on the (false) assumption of independence may produce erroneous conclusions.

(10) For this example, a graphical display of profit versus the number of service technicians yields both the optimal number of technicians and a measure of how sensitive the profit is to variations about this optimal number. In this way a policy decision can be made. Provided this decision does not violate any external constraints, such as labor union rules, the policy should be implemented.

(11) Documentation of the machine shop model would include a system diagram, explanations of assumptions made about machine failure rates and service repair rates, a description of the specification model, software for the computational model, tables and figures of output, and a description of the output analysis.

Insight

An important benefit of developing and using a discrete-event simulation model is that valuable insight is acquired. As conceptual models are formulated, computational models developed and output data analyzed, subtle system features and component interactions may be discovered that would not have been noticed otherwise. The systematic application of Algorithms 1.1.1 and 1.1.2 can result in better actions taken due to insight gained by an increased understanding of how the system operates.

1.1.4 PROGRAMMING LANGUAGES

There is a continuing debate in discrete-event simulation — to use a general-purpose programming language or a (special-purpose) simulation programming language. For example, two standard discrete-event simulation textbooks provide the following contradictory advice. Bratley, Fox, and Schrage (1987, page 219) state "... for any important large-scale real application we would write the programs in a standard general-purpose language, and avoid all the simulation languages we know." In contrast, Law and Kelton (2000, page 204) state "... we believe, in general, that a modeler would be prudent to give serious consideration to the use of a simulation package."

General-purpose languages are more flexible and familiar; simulation languages allow modelers to build computational models quickly. There is no easy way to resolve this debate in general. However, for the specific purpose of this book — learning the principles and techniques of discrete-event simulation — the debate is easier to resolve. Learning discrete-event simulation methodology is facilitated by using a familiar, general-purpose programming language, a philosophy that has dictated the style and content of this book.

General-Purpose Languages

Because discrete-event simulation is a specific instance of scientific computing, any *general-purpose* programming language suitable for scientific computing is similarly suitable for discrete-event simulation. Therefore, a history of the use of general-purpose programming languages in discrete-event simulation is really a history of general-purpose programming languages in scientific computing. Although this history is extensive, we will try to summarized it in a few paragraphs.

For many years FORTRAN was the primary general-purpose programming language used in discrete-event simulation. In retrospect, this was natural and appropriate because there was no well-accepted alternative. By the early 80's things began to change dramatically. Several general-purpose programming languages created in the 70's, primarily C and Pascal, were as good as or superior to FORTRAN in most respects and they began to gain acceptance in many applications, including discrete-event simulation, where FORTRAN was once dominant. Because of its structure and relative simplicity, Pascal became the de facto first programming language in many computer science departments; because of its flexibility and power, the use of C became common among professional programmers.

Personal computers became popular in the early 80's, followed soon thereafter by increasingly more powerful workstations. Concurrent with this development, it became clear that networked workstations or, to a lesser extent, stand-alone personal computers, were ideal discrete-event simulation engines. The popularity of workstation networks then helped to guarantee that C would become the general-purpose language of choice for discrete-event simulation. That is, the usual workstation network was Unix-based, an environment in which C was the natural general-purpose programming language of choice. The use of C in discrete-event simulation became wide-spread by the early 90's when C became standardized and C++, an object-oriented extension of C, gained popularity.

In addition to C, C++, FORTRAN, and Pascal, other general-purpose programming languages are occasionally used in discrete-event simulation. Of these, Ada, Java, and (modern, compiled) BASIC are probably the most common. This diversity is not surprising because every general-purpose programming language has its advocates, some quite vocal, and no matter what the language there is likely to be an advocate to argue that it is ideal for discrete-event simulation. We leave that debate for another forum, however, confident that our use of ANSI C in this book is appropriate.

Simulation Languages

Simulation languages have built-in features that provide many of the tools needed to write a discrete-event simulation program. Because of this, simulation languages support rapid prototyping and have the potential to decrease programming time significantly. Moreover, animation is a particularly important feature now built into most of these simulation languages. This is important because animation can increase the acceptance of discrete-event simulation as a legitimate problem-solving technique. By using animation, dynamic graphical images can be created that enhance verification, validation, and the development of insight. The most popular discrete-event simulation languages historically are GPSS, SIMAN, SLAM II, and SIMSCRIPT II.5. Because of our emphasis in the book on the use of general-purpose languages, any additional discussion of simulation languages is deferred to Appendix A.

Because it is not discussed in Appendix A, for historical reasons it is appropriate here to mention the simulation language Simula. This language was developed in the 60's as an object-oriented ALGOL extension. Despite its object orientation and several other novel (for the time) features, it never achieved much popularity, except in Europe. Still, like other premature-but-good ideas, the impact of Simula has proven to be profound, including serving as the inspiration for the creation of C++.

1.1.5 ORGANIZATION AND TERMINOLOGY

We conclude this first section with some brief comments about the organization of the book and the sometimes ambiguous use of the words *simulation*, *simulate*, and *model*.

Organization

The material in this book could have been organized in several ways. Perhaps the most natural sequence would be to follow, in order, the steps in Algorithms 1.1.1 and 1.1.2, devoting a chapter to each step. However, that sequence is not followed. Instead, the material is organized in a manner consistent with the experimental nature of discrete-event simulation. That is, we begin to model, simulate, and analyze simple-but-representative systems as soon as possible (indeed, in the next section). Whenever possible, new concepts are first introduced in an informal way that encourages experimental self-discovery, with a more formal treatment of the concepts deferred to later chapters. This organization has proven to be successful in the classroom.

Terminology

The words "model" and "simulation" or "simulate" are commonly used interchangeably in the discrete-event simulation literature, both as a noun and as a verb. For pedagogical reasons this word interchangeability is unfortunate because, as indicated previously, a "model" (the noun) exists at three levels of abstraction: conceptual, specification, and computational. At the computational level, a system model is a computer program; this computer program is what most people mean when they talk about a system simulation. In this context a simulation and a computational system model are equivalent. It is uncommon, however, to use the noun "simulation" as a synonym for the system model at either the conceptual or specification level. Similarly, "to model" (the verb) implies activity at three levels, but "to simulate" is usually a computational activity only.

When appropriate we will try to be careful with these words, generally using simulation or simulate in reference to a computational activity only. This is consistent with common usage of the word *simulation* to characterize not only the computational model (computer program) but also the computational process of using the discrete-event simulation model to generate output statistical data and thereby analyze system performance. In those cases when there is no real need to be fussy about terminology, we will yield to tradition and use the word simulation or simulate even though the word model may be more appropriate.

1.1.6 EXERCISES

Exercise 1.1.1 There are six leaf nodes in the system model tree in Figure 1.1.1. For each leaf node, describe a specific example of a corresponding physical system.

Exercise 1.1.2 The distinction between model *verification* and model *validation* is not always clear in practice. Generally, in the sense of Algorithm 1.1.1, the ultimate objective is a valid discrete-event simulation model. If you were told that "this discrete-event simulation model had been verified but it is not known if the model is valid" how would you interpret that statement?

Exercise 1.1.3 The *state* of a system is important, but difficult to define in a general context. (a) Locate at least five contemporary textbooks that discuss system modeling and, for each, research and comment on the extent to which the technical term "state" is defined. If possible, avoid example-based definitions or definitions based on a specific system. (b) How would you define the state of a system?

Exercise 1.1.4 (a) Use an Internet search engine to identify at least 10 different simulation languages that support discrete-event simulation. (Note that the '-' in discrete-event is not a universal convention.) Provide a URL, phone number, or mailing address for each and, if it is a commercial product, a price. (b) If you tried multiple search engines, which produced the most meaningful hits?

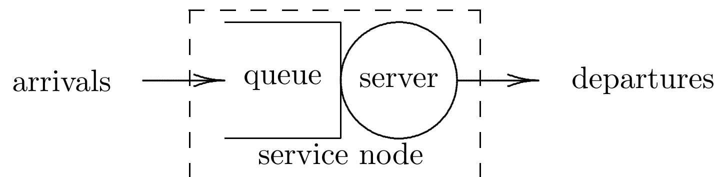
1.2 A Single-Server Queue

In this section we will construct a *trace-driven* discrete-event simulation model (i.e., a model driven by external data) of a single-server service node. We begin the construction at the conceptual level.

1.2.1 CONCEPTUAL MODEL

Definition 1.2.1 A *single-server service node* consists of a *server* plus its *queue.**

Figure 1.2.1. Single-server service node system diagram.



Jobs (customers) arrive at the service node at random points in time seeking service. When service is provided, the service time involved is also random. At the completion of service, jobs depart. The service node operates as follows: as each (new) job arrives, if the server is busy then the job enters the queue, else the job immediately enters service; as each (old) job departs, if the queue is empty then the server becomes idle, else a job is selected from the queue to immediately enter service. At any 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, the queue must be empty; if the queue is not empty then the server must be busy.

Example 1.2.1 If there is just one service technician, the machine shop model presented in Examples 1.1.1 and 1.1.2 is a single-server service node model. That is, the "jobs" are the machines to be repaired and the "server" is the service technician. (In this case, whether the jobs move to the server or the server moves to the jobs is not an important distinction because the repair time is the primary source of delay.)

Definition 1.2.2 Control of the queue is determined by the *queue discipline* — the algorithm used when a job is selected from the queue to enter service. The standard algorithms are:

- FIFO — first in, first out (the traditional computer science queue data structure);
- LIFO — last in, first out (the traditional computer science stack data structure);
- SIRO — service in random order;
- Priority — typically, shortest job first (SJF) or equivalently, in job-shop terminology, shortest processing time (SPT).

The maximum possible number of jobs in the service node is the *capacity.* The capacity can be either *finite* or *infinite.* If the capacity is finite then jobs that arrive and find the service node full will be *rejected* (unable to enter the service node).

---

* The term "service node" is used in anticipation of extending this model, in later chapters, to a *network* of service nodes.

Certainly the most common queue discipline is FIFO (also known as FCFS — first come, first served). If the queue discipline is FIFO, then the order of arrival to the service node and the order of departure from the service node are the same; there is no passing. In particular, upon arrival a job will enter the queue if and only if the *previous* job has not yet departed the service node. This is an important observation that can be used to simplify the simulation of a FIFO single-server service node. If the queue discipline is not FIFO then, for at least some jobs, the order of departure will differ from the order of arrival. In this book, the *default* assumptions are that the queue discipline is FIFO and the service node capacity is infinite, unless otherwise specified. Discrete-event simulation allows these assumptions to be easily altered for more realistic modeling.

There are two important additional default assumptions implicit in Definition 1.2.1. First, service is *non-preemptive* — once initiated, service on a job will be continued until completion. That is, a job in service cannot be preempted by another job arriving later. Preemption is commonly used with priority queue disciplines to prevent a job with a large service time requirement from producing excessive delays for small jobs arriving soon after service on the large job has begun. Second, service is *conservative* — the server will never remain idle if there is one or more jobs in the service node. If the queue discipline is not FIFO *and* if the next arrival time is known in advance then, even though one or more jobs are in the service node, it may be desirable for a non-conservative server to remain idle until the next job arrives. This is particularly true in non-preemptive job scheduling applications if a job in the service node has a much larger service requirement than the next job scheduled to arrive.

1.2.2 SPECIFICATION MODEL

The following variables, illustrated in Figure 1.2.2, provide the basis for moving from a conceptual model to a specification model. At their arrival to the service node, jobs are indexed by $i = 1, 2, 3, \ldots$ For each job there are six associated time variables.

- The *arrival time* of job i is a_i.
- The *delay* of job i in the queue is $d_i \geq 0$.
- The time that job i begins service is $b_i = a_i + d_i$.
- The *service time* of job i is $s_i > 0$.
- The *wait* of job i in the service node (queue and service) is $w_i = d_i + s_i$.
- The time that job i completes service (the *departure time*) is $c_i = a_i + w_i$.

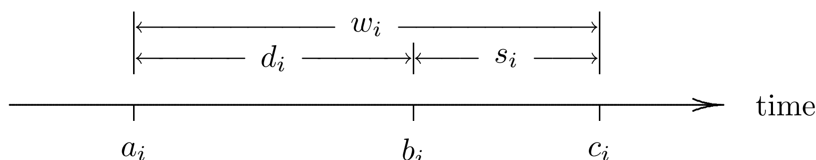


Figure 1.2.2. Six variables associated with job i.

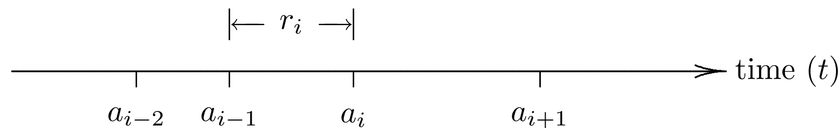
The term "wait" can be confusing; w_i represents the total time job i spends in the service node, *not* just the time spent in the queue. The time spent in the queue (if any) is the delay d_i. In many computer science applications the term *response time* is used. To some authors this means *wait*, to others it means *delay.* Because of this ambiguity, we will generally avoid using the term "response time" choosing instead to consistently use the terminology specified previously. Similarly, we avoid the use of the common terms *sojourn time*, *flow time*, or *system time*, in place of wait.

Arrivals

As a convention, if the service node capacity is finite then rejected jobs (if any) are not indexed. That is, although rejected jobs may be counted for statistical purposes (for example, to estimate the probability of rejection), the index $i = 1, 2, 3, \ldots$ is restricted to only those jobs that actually enter the service node.

Rather than specify the *arrival* times a_1, a_2, ... explicitly, in some discrete-event simulation applications it is preferable to specify the *interarrival* times r_1, r_2, ..., thereby defining the arrival times implicitly, as shown in Figure 1.2.3 and defined in Definition 1.2.3.

Figure 1.2.3. Relationship between arrival and interarrival times.



Definition 1.2.3 The *interarrival time* between jobs $i-1$ and i is $r_i = a_i - a_{i-1}$. That is, $a_i = a_{i-1} + r_i$ and so (by induction), with $a_0 = 0$ the arrival times are*

$$a_i = r_1 + r_2 + \cdots + r_i \qquad i = 1, 2, 3, \ldots$$

(We assume that $r_i > 0$ for all i, thereby eliminating the possibility of *bulk* arrivals. That is, jobs are assumed to arrive one at a time.)

Algorithmic Question

The following algorithmic question is fundamental. Given a knowledge of the arrival times a_1, a_2, ... (or, equivalently, the interarrival times r_1, r_2, ...), the associated service times s_1, s_2, ..., and the queue discipline, how can the delay times d_1, d_2, ... be computed?

As discussed in later chapters, for some queue disciplines this question is more difficult to answer than for others. If the queue discipline is FIFO, however, then the answer is particularly simple. That is, as demonstrated next, if the queue discipline is FIFO then there is a simple algorithm for computing d_i (as well as b_i, w_i, and c_i) for all i.

---

* All arrival times are referenced to the virtual arrival time a_0. Unless explicitly stated otherwise, in this chapter and elsewhere we assume that elapsed time is measured in such a way that $a_0 = 0$.

Two Cases

If the queue discipline is FIFO then the delay d_i of job $i = 1, 2, 3, \ldots$ is determined by when the job's arrival time a_i occurs relative to the departure time c_{i-1} of the previous job. There are two cases to consider.

- **Case I.** If $a_i < c_{i-1}$, i.e., if job i arrives before job $i-1$ departs then, as illustrated, job i will experience a delay of $d_i = c_{i-1} - a_i$. Job $i-1$'s history is displayed above the time axis and job i's history is displayed below the time axis in Figures 1.2.4 and 1.2.5.

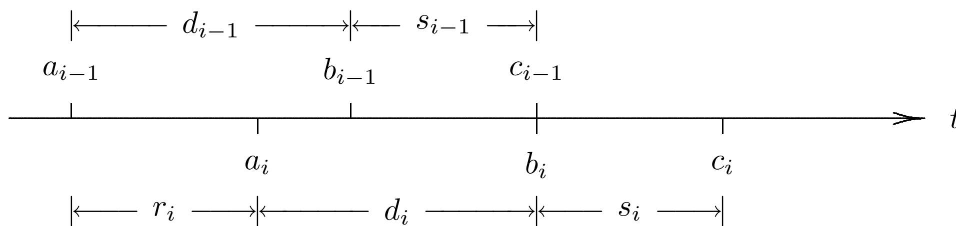


Figure 1.2.4. Job i arrives before job $i-1$ departs.

- **Case II.** If instead $a_i \geq c_{i-1}$, i.e., if job i arrives after (or just as) job $i-1$ departs then, as illustrated, job i will experience no delay so that $d_i = 0$.

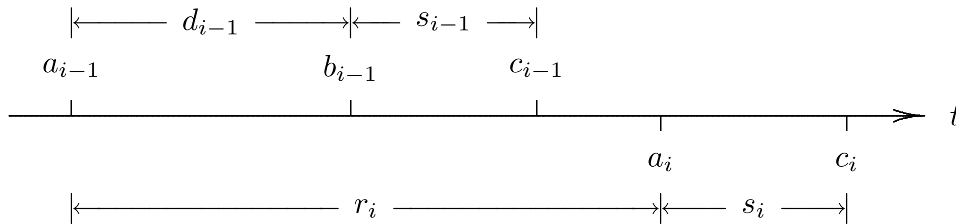


Figure 1.2.5. Job i arrives after job $i-1$ departs.

Algorithm

The key point in algorithm development is that if the queue discipline is FIFO then the truth of the expression $a_i < c_{i-1}$ determines whether or not job i will experience a delay. Based on this logic, the computation of the delays is summarized by Algorithm 1.2.1. This algorithm, like all those presented in this book, is written in a C-like pseudo-code that is easily translated into other general-purpose programming languages.

Although it is not an explicit part of Algorithm 1.2.1, an equation can be written for the delay that depends on the interarrival and service times only. That is

$$\begin{aligned} c_{i-1} - a_i &= (a_{i-1} + d_{i-1} + s_{i-1}) - a_i \\ &= d_{i-1} + s_{i-1} - (a_i - a_{i-1}) \\ &= d_{i-1} + s_{i-1} - r_i. \end{aligned}$$

If $d_0 = s_0 = 0$ then $d_1, d_2, d_3, \ldots$ are defined by the nonlinear equation

$$d_i = \max\{\, 0,\, d_{i-1} + s_{i-1} - r_i \,\} \qquad i = 1, 2, 3, \ldots$$

This equation is commonly used in theoretical studies to analyze the stochastic behavior of a FIFO service node.

Algorithm 1.2.1 If the arrival times a_1, a_2, ... and service times s_1, s_2, ... are known and if the server is initially idle, then this algorithm computes the delays $d_1, d_2, \ldots$ in a single-server FIFO service node with infinite capacity.

```
c0 = 0.0;                                        /* assumes that a0 = 0.0 */
i = 0;
while ( more jobs to process ) {
   i++;
   ai = GetArrival();
   if (ai < ci-1)
     di = ci-1 - ai;                         /* calculate delay for job i */
   else
     di = 0.0;                                   /* job i has no delay */
   si = GetService();
   ci = ai + di + si;               /* calculate departure time for job i */
}
n = i;
return d1, d2, ..., dn;
```

The GetArrival and GetService procedures read the next arrival and service time from a file. (An algorithm that does not rely on the FIFO assumption is presented in Chapter 5.)

Example 1.2.2 If Algorithm 1.2.1 is used to process $n = 10$ jobs according to the input indicated below (for simplicity the a_i's and s_i's are integer time units, e.g., seconds, minutes, etc.) then the output is the sequence of delays are calculated as:

	i	:	1	2	3	4	5	6	7	8	9	10
read from file	a_i	:	15	47	71	111	123	152	166	226	310	320
from algorithm	d_i	:	0	11	23	17	35	44	70	41	0	26
read from file	s_i	:	43	36	34	30	38	40	31	29	36	30

For future reference, note that the last job arrived at time $a_n = 320$ and departed at time $c_n = a_n + d_n + s_n = 320 + 26 + 30 = 376$.

As discussed in more detail later in this section, it is a straight-forward programming exercise to produce a computational model of a single-server FIFO service node with infinite capacity using Algorithm 1.2.1. The ANSI C program ssq1 is an example. Three features of this program are noteworthy. (*i*) Because of its reliance on previously recorded arrival and service time data read from an external file, ssq1 is a so-called *trace-driven* discrete-event simulation program. (*ii*) Because the queue discipline is FIFO, program ssq1 does not need to use a queue data structure. (*iii*) Rather than produce a sequence of delays as output, program ssq1 computes four averages instead: the average interarrival time, service time, delay, and wait. These four job-averaged statistics and three corresponding time-averaged statistics are discussed next.

1.2.3 OUTPUT STATISTICS

One basic issue that must be resolved when constructing a discrete-event simulation model is the question of what statistics should be generated. The purpose of simulation is insight and we gain insight about the performance of a system by looking at meaningful statistics. Of course, a decision about what statistics are most meaningful is dependent upon your perspective. For example, from a job's (customer's) perspective the most important statistic might be the average delay or the 95th percentile of the delay — in either case, the smaller the better. On the other hand, particularly if the server is an expensive resource whose justification is based on an anticipated heavy workload, from management's perspective the server's utilization (the proportion of busy time, see Definition 1.2.7) is most important — the larger the better.

Job-Averaged Statistics

Definition 1.2.4 For the first n jobs, the *average interarrival time* and the *average service time* are, respectively*

$$\bar{r} = \frac{1}{n}\sum_{i=1}^{n} r_i = \frac{a_n}{n} \qquad \text{and} \qquad \bar{s} = \frac{1}{n}\sum_{i=1}^{n} s_i.$$

The reciprocal of the average interarrival time, $1/\bar{r}$, is the *arrival rate*; the reciprocal of the average service time, $1/\bar{s}$, is the *service rate*.

Example 1.2.3 For the $n = 10$ jobs in Example 1.2.2, $\bar{r} = a_n/n = 320/10 = 32.0$ and $\bar{s} = 34.7$. If time in this example is measured in seconds, then the average interarrival time is 32.0 seconds per job and the average service time is 34.7 seconds per job. The corresponding arrival rate is $1/\bar{r} = 1/32.0 \cong 0.031$ jobs per second; the service rate is $1/\bar{s} = 1/34.7 \cong 0.029$ jobs per second. In this particular example, the server is not quite able to process jobs at the rate they arrive on average.

Definition 1.2.5 For the first n jobs, the *average delay* in the queue and the *average wait* in the service node are, respectively

$$\bar{d} = \frac{1}{n}\sum_{i=1}^{n} d_i \qquad \text{and} \qquad \bar{w} = \frac{1}{n}\sum_{i=1}^{n} w_i.$$

Recall that $w_i = d_i + s_i$ for all i. Therefore, the average time spent in the service node will be the sum of the average times spent in the queue and in service. That is

$$\bar{w} = \frac{1}{n}\sum_{i=1}^{n} w_i = \frac{1}{n}\sum_{i=1}^{n}(d_i + s_i) = \frac{1}{n}\sum_{i=1}^{n} d_i + \frac{1}{n}\sum_{i=1}^{n} s_i = \bar{d} + \bar{s}.$$

The point here is that it is sufficient to compute any two of the statistics $\bar{w}$, $\bar{d}$, $\bar{s}$. The third statistic can then be computed from the other two, if appropriate.

* The equation $\bar{r} = a_n/n$ follows from Definition 1.2.3 and the assumption that $a_0 = 0$.

Example 1.2.4 For the data in Example 1.2.2, $\bar{d} = 26.7$ and $\bar{s} = 34.7$. Therefore, the average wait in the node is $\bar{w} = 26.7 + 34.7 = 61.4$. (See also Example 1.2.6.)

In subsequent chapters we will construct increasingly more complex discrete-event simulation models. Because it is never easy to verify and validate a complex model, it is desirable to be able to apply as many *consistency checks* to the output data as possible. For example, although program `ssq1` is certainly not a complex discrete-event simulation model, it is desirable in this program to accumulate $\bar{w}$, $\bar{d}$, and $\bar{s}$ *independently.* Then, from the program output the equation $\bar{w} = \bar{d} + \bar{s}$ can be used as a consistency check.

Time-Averaged Statistics

The three statistics $\bar{w}$, $\bar{d}$ and $\bar{s}$ are *job-averaged* statistics — the data is averaged over all jobs. Job averages are easy to understand; they are just traditional arithmetic averages. We now turn to another type of statistic that is equally meaningful, *time-averaged.* Time-averaged statistics may be less familiar, however, because they are defined by an area under a curve, i.e., by integration instead of summation.

Time-averaged statistics for a single-server service node are defined in terms of three additional variables. At any time $t > 0$:

- $l(t) = 0, 1, 2, \ldots$ is the number of jobs in the service node at time t;
- $q(t) = 0, 1, 2, \ldots$ is the number of jobs in the queue at time t;
- $x(t) = 0, 1$ is the number of jobs in service at time t.

By definition, $l(t) = q(t) + x(t)$ for any $t > 0$.

The three functions $l(\cdot)$, $q(\cdot)$, and $x(\cdot)$ are *piecewise constant.* That is, for example, a display of $l(t)$ versus t will consist of a sequence of constant segments with unit height step discontinuities as illustrated in Figure 1.2.6. (This figure corresponds to the data in Example 1.2.2. The dashed line represents the time-averaged number in the node — see Example 1.2.6.)

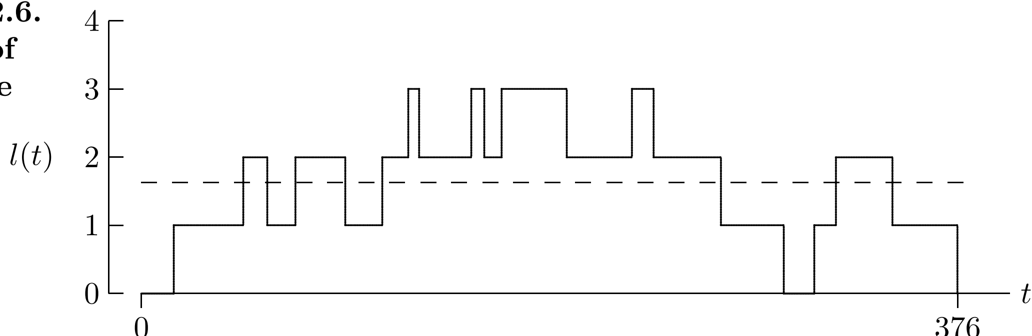


Figure 1.2.6. Number of jobs in the service node.

The step discontinuities are positive at the arrival times and negative at the departure times. The corresponding figures for $q(\cdot)$ and $x(\cdot)$ can be deduced from the fact that $q(t) = 0$ and $x(t) = 0$ if and only if $l(t) = 0$, otherwise $q(t) = l(t) - 1$ and $x(t) = 1$.

Definition 1.2.6 Over the time interval $(0, \tau)$ the *time-averaged number in the node* is

$$\bar{l} = \frac{1}{\tau} \int_0^\tau l(t)\, dt.$$

Similarly, the *time-averaged number in the queue* and the *time-averaged number in service* are

$$\bar{q} = \frac{1}{\tau} \int_0^\tau q(t)\, dt \qquad \text{and} \qquad \bar{x} = \frac{1}{\tau} \int_0^\tau x(t)\, dt.$$

Because $l(t) = q(t) + x(t)$ for all $t > 0$ it follows that

$$\bar{l} = \bar{q} + \bar{x}.$$

Example 1.2.5 For the data in Example 1.2.2 (with $\tau = c_{10} = 376$) the three time-averaged statistics are $\bar{l} = 1.633$, $\bar{q} = 0.710$, and $\bar{x} = 0.923$. These values can be determined by calculating the areas associated with the integrals given in Definition 1.2.6 or by exploiting a mathematical relationship between the job-averaged statistics $\bar{w}$, $\bar{d}$, and $\bar{s}$ and the time-averaged statistics $\bar{l}$, $\bar{q}$, and $\bar{x}$, as illustrated subsequently in Example 1.2.6.

The equation $\bar{l} = \bar{q} + \bar{x}$ is the time-averaged analog of the job-averaged equation $\bar{w} = \bar{d} + \bar{s}$. As we will see in later chapters, time-averaged statistics have the following important characterizations.

- If we were to observe (sample) the number in the service node, for example, at many different times chosen *at random* between 0 and τ and then calculate the arithmetic average of all these observations, the result should be close to $\bar{l}$.
- Similarly, the arithmetic average of many random observations of the number in the queue should be close to $\bar{q}$ and the arithmetic average of many random observations of the number in service (0 or 1) should be close to $\bar{x}$.
- $\bar{x}$ must lie in the closed interval $[0, 1]$.

Definition 1.2.7 The time-averaged number in service $\bar{x}$ is also known as the server *utilization*. The reason for this terminology is that $\bar{x}$ represents the proportion of time that the server is busy.

Equivalently, if one particular time is picked *at random* between 0 and τ then $\bar{x}$ is the probability that the server is busy at that time. If $\bar{x}$ is close to 1.0 then the server is busy most of the time and correspondingly large values of $\bar{l}$ and $\bar{q}$ will be produced. On the other hand, if the utilization is close to 0.0 then the server is idle most of the time and the values of $\bar{l}$ and $\bar{q}$ will be small. The case study, presented later, is an illustration.

Little's Equations

One important issue remains — how are job averages and time averages related? Specifically, how are $\bar{w}$, $\bar{d}$, and $\bar{s}$ related to $\bar{l}$, $\bar{q}$, and $\bar{x}$?

In the particular case of an infinite capacity FIFO service node that begins and ends in an idle state, the following theorem provides an answer to this question. See Exercise 1.2.7 for a generalization of this theorem to any queue discipline.

Theorem 1.2.1 (Little, 1961) If the queue discipline is FIFO, the service node capacity is infinite, and the server is idle both initially (at $t = 0$) and immediately after the departure of the nth job (at $t = c_n$) then

$$\int_0^{c_n} l(t)\,dt = \sum_{i=1}^{n} w_i \qquad \text{and} \qquad \int_0^{c_n} q(t)\,dt = \sum_{i=1}^{n} d_i \qquad \text{and} \qquad \int_0^{c_n} x(t)\,dt = \sum_{i=1}^{n} s_i.$$

Proof For each job $i = 1, 2, \ldots$, define an *indicator function* $\psi_i(t)$ that is 1 when the ith job is in the service node and is 0 otherwise

$$\psi_i(t) = \begin{cases} 1 & a_i < t < c_i \\ 0 & \text{otherwise.} \end{cases}$$

Then

$$l(t) = \sum_{i=1}^{n} \psi_i(t) \qquad\qquad 0 < t < c_n$$

and so

$$\int_0^{c_n} l(t)\,dt = \int_0^{c_n} \sum_{i=1}^{n} \psi_i(t)\,dt = \sum_{i=1}^{n} \int_0^{c_n} \psi_i(t)\,dt = \sum_{i=1}^{n} (c_i - a_i) = \sum_{i=1}^{n} w_i.$$

The other two equations can be derived in a similar way.

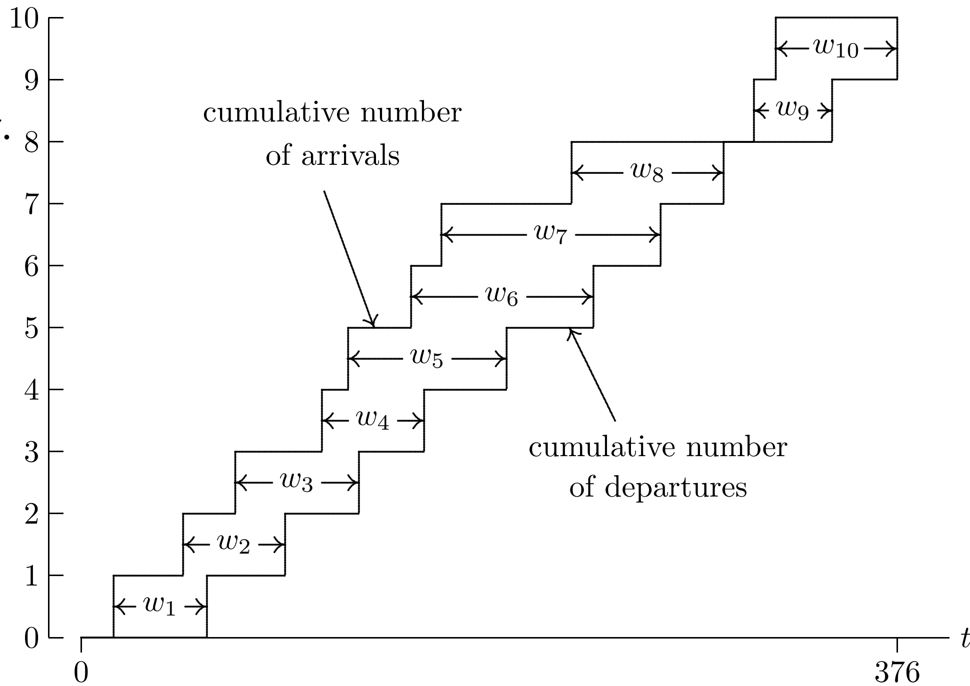


Figure 1.2.7. Little's first equation.

Example 1.2.6 Figure 1.2.7 illustrates Little's first equation for the data in Example 1.2.2. The top step function denotes the cumulative number of arrivals to the service node and the bottom step function denotes the cumulative number of departures from the service node. The vertical distance between the two step-functions at any time t is $l(t)$, which was plotted in Figure 1.2.6. The wait times are indicated as the horizontal distances between the risers. In this figure, it is easy to see that

$$\int_0^{376} l(t)\,dt = \sum_{i=1}^{10} w_i = 614.$$

Little's equations provide a valuable link between the job-averaged statistics $\bar{w}$, $\bar{d}$, and $\bar{s}$ and the time-averaged statistics $\bar{l}$, $\bar{q}$, and $\bar{x}$. In Definition 1.2.6 let $\tau = c_n$. Then from Theorem 1.2.1 we have

$$c_n\bar{l} = \int_0^{c_n} l(t)\,dt = \sum_{i=1}^{n} w_i = n\bar{w}$$

so that $\bar{l} = (n/c_n)\bar{w}$. Similarly, $c_n\bar{q} = n\bar{d}$ and $c_n\bar{x} = n\bar{s}$. Therefore

$$\bar{l} = \left(\frac{n}{c_n}\right)\bar{w} \qquad \text{and} \qquad \bar{q} = \left(\frac{n}{c_n}\right)\bar{d} \qquad \text{and} \qquad \bar{x} = \left(\frac{n}{c_n}\right)\bar{s}$$

which explains how $\bar{w}$, $\bar{d}$, and $\bar{s}$ are related to $\bar{l}$, $\bar{q}$, and $\bar{x}$. These important equations relate to *steady-state* statistics and *Little's equations* — for more detail, see Chapter 8.

Example 1.2.7 For the data in Example 1.2.2 the last ($n = 10$) job departs at $c_n = 376$. From Example 1.2.4, $\bar{w} = 61.4$ and therefore $\bar{l} = (10/376)\,61.4 \cong 1.633$. Similarly, the time-averaged number in the queue and in service are $\bar{q} = (10/376)\,26.7 \cong 0.710$ and $\bar{x} = (10/376)\,34.7 \cong 0.923$.

1.2.4 COMPUTATIONAL MODEL

As discussed previously, by using Algorithm 1.2.1 in conjunction with some statistics gathering logic it is a straight-forward programming exercise to produce a computational model of a single-server FIFO service node with infinite capacity. The ANSI C program `ssq1` is an example. Like all of the software presented in this book, this program is designed with readability and extendibility considerations.

Program ssq1

Program `ssq1` reads arrival and service time data from the disk file `ssq1.dat`. This is a text file that consists of arrival times $a_1, a_2, \ldots, a_n$ and service times $s_1, s_2, \ldots, s_n$ for $n = 1000$ jobs in the format

$$\begin{array}{cc} a_1 & s_1 \\ a_2 & s_2 \\ \vdots & \vdots \\ a_n & s_n \end{array}$$

In Chapter 3 we will free this trace-driven program from its reliance on external data by using randomly generated arrival and service times instead.

Because the queue discipline is FIFO, program `ssq1` does not need to use a queue data structure. In Chapter 5 we will consider non-FIFO queue disciplines and some corresponding priority queue data structures that can be used at the computational model level.

Program `ssq1` computes the average interarrival time $\bar{r}$, the average service time $\bar{s}$, the average delay $\bar{d}$, and the average wait $\bar{w}$. In Exercise 1.2.2 you are asked to modify this program so that it will also compute the time-averaged statistics $\bar{l}$, $\bar{q}$, and $\bar{x}$.

Example 1.2.8 For the datafile `ssq1.dat` the observed arrival rate $1/\bar{r} \cong 0.10$ is significantly less than the observed service rate $1/\bar{s} \cong 0.14$. If you modify `ssq1` to compute $\bar{l}$, $\bar{q}$, and $\bar{x}$ you will find that $1 - \bar{x} \cong 0.28$, and so the server is idle 28% of the time. Despite this significant idle time, enough jobs are delayed so that the average number in the queue is nearly 2.0.

Traffic Intensity

The ratio of the arrival rate to the service rate is commonly called the *traffic intensity*. From the equations in Definition 1.2.4 and Theorem 1.2.1 it follows that the observed traffic intensity is the ratio of the observed arrival rate to the observed service rate

$$\frac{1/\bar{r}}{1/\bar{s}} = \frac{\bar{s}}{\bar{r}} = \frac{\bar{s}}{a_n/n} = \left(\frac{c_n}{a_n}\right)\bar{x}.$$

Therefore, provided the ratio c_n/a_n is close to 1.0, the traffic intensity and utilization will be nearly equal. In particular, if the traffic intensity is less than 1.0 and n is large, then it is reasonable to expect that the ratio $c_n/a_n = 1 + w_n/a_n$ will be close to 1.0. We will return to this question in later chapters. For now, we close with an example illustrating how relatively sensitive the service node statistics $\bar{l}$, $\bar{q}$, $\bar{w}$, and $\bar{d}$ are to changes in the utilization and how nonlinear this dependence can be.

Case Study

Sven & Larry's Ice Cream Shoppe is a thriving business that can be modeled as a single-server queue. The owners are considering adding additional flavors and cone options, but are concerned about the resultant increase in service times on queue length. They decide to use a trace-driven simulation to assess the impact of the longer service times associated with the additional flavors and cone options.

The file `ssq1.dat` represents 1000 customer interactions at Sven & Larry's. The file consists of arrival times of groups of customers and the group's corresponding service times. The service times vary significantly because of the number of customers in each group.

The dependence of the average queue length $\bar{q}$ on the utilization $\bar{x}$ is illustrated in Figure 1.2.8. This figure was created by systematically increasing or decreasing each service time in the datafile `ssq1.dat` by a common multiplicative factor, thereby causing both $\bar{x}$ and $\bar{q}$ to change correspondingly. The $(\bar{x}, \bar{q}) \cong (0.72, 1.88)$ point circled corresponds to the data in `ssq1.dat` while the point immediately to its right, for example, corresponds to the same data with each service time multiplied by 1.05. The next point to the right corresponds to each service time multiplied by 1.10. From this figure, we see that even a modest increase in service times will produce a significant increase in the average queue length. A nonlinear relationship between $\bar{x}$ and $\bar{q}$ is particularly pronounced for utilizations near $\bar{x} = 1$. A 15% increase in the service times from their current values will result in a 109% increase in the time-averaged number of customers in queue, whereas a 30% increase in the service times from their current values will result in a 518% increase in the time-averaged number of customers in queue.

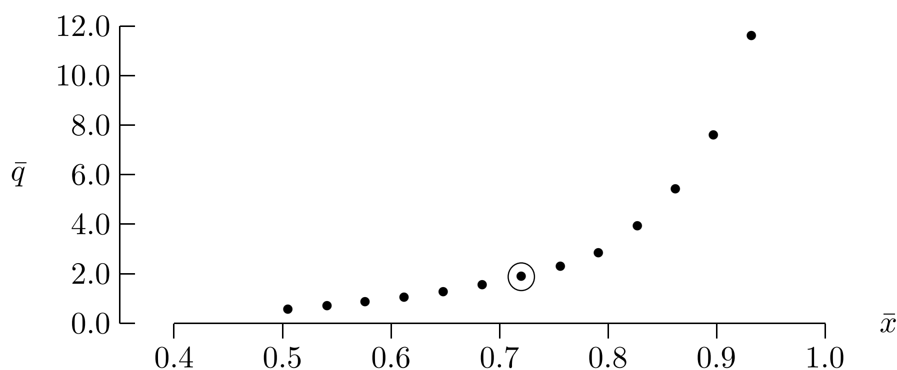


Figure 1.2.8. Average queue length vs. utilization.

Sven & Larry need to assess the impact of the increased service time associated with new flavors and cones on their operation. If the service times increase by only a modest amount, say 5% or 10% above the current times, the average queue length will grow modestly. The new flavors and cone options may, however, also increase the arrival rate — potentially exacerbating the problem with long lines. If queues grow to the point where the owners believe that customers are taking their ice cream business elsewhere, they should consider hiring a second server. A separate analysis would be necessary to determine the probability that an arriving group of customers will balk (never enter the queue) or renege (depart from the queue after entering) as a function of the queue's length.

Graphics Considerations

Figure 1.2.8 presents "raw" simulation output data. That is, each • represents a computed $(\bar{x}, \bar{q})$ point. Because there is nothing inherently discrete about either $\bar{x}$ or $\bar{q}$, many additional points could have been computed and displayed to produce an (essentially) continuous $\bar{q}$ versus $\bar{x}$ curve. In this case, however, additional computations seem redundant; few would question the validity of the smooth curve produced by connecting the •'s with lines, as illustrated in Figure 1.2.9. The nonlinear dependence of $\bar{q}$ on $\bar{x}$ is evident, particularly as $\bar{x}$ approaches 1.0 and the corresponding increase in $\bar{q}$ becomes dramatic.

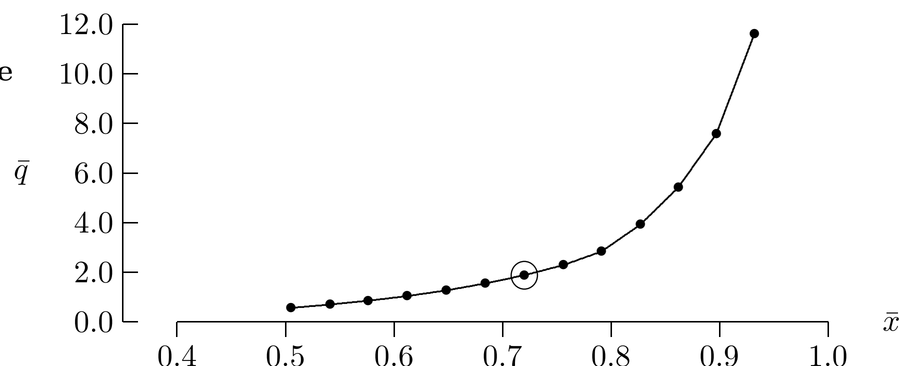


Figure 1.2.9. Average queue length vs. utilization with linear interpolation.

Perhaps because we were taught to do this as children, there is a natural tendency to *always* "connect the dots" (interpolate) when presenting a discrete set of experimental data. (The three most common interpolating functions are linear, quadratic, and spline functions.) Before taking such artistic liberties, however, consider the following guidelines.

- If the data has essentially no uncertainty and if the resulting interpolating curve is smooth, then there is little danger in connecting the dots — provided the original dots are left in the figure to remind the reader that some artistic license was used.
- If the data has essentially no uncertainty but the resulting interpolating curve is not smooth then more dots need to be generated to achieve a graphics scale at which smooth interpolation is reasonable.
- If the dots correspond to uncertain (noisy) data then interpolation is not justified; instead, either approximation should be used in place of interpolation, or the temptation to superimpose a continuous curve should be resisted completely.

These guidelines presume that the data is not inherently discrete. If the data is inherently discrete then it is illogical and potentially confusing to superimpose a continuous (interpolating or approximating) curve. Example 1.3.7 in the next section is an illustration of data that is inherently discrete.*

1.2.5 EXERCISES

Exercise 1.2.1[a] How would you use the table in Example 1.2.2 to construct the associated $l(t)$ versus t figure? That is, construct an algorithm that will compute *in order* the interlaced arrival and departure times that define the points at which $l(t)$ changes. (Avoid storing $a_1, a_2, \ldots, a_n$ and $c_1, c_2, \ldots, c_n$ as two arrays, linked lists or external disk files and then merging the two into one due to memory and CPU considerations for large n.)

---

* Those interested in an excellent discussion and illustration of graphics considerations are encouraged to read the classic *The Visual Display of Quantitative Information* (Tufte, 2001). The author discusses clarity of presentation through uncluttered graphics that maximize information transmission with minimal ink. The accurate display of simulation output will be stressed throughout this text.

Exercise 1.2.2 (a) Modify program `ssq1` to output the additional statistics $\bar{l}$, $\bar{q}$, and $\bar{x}$. (b) Similar to the case study, use this program to compute a table of $\bar{l}$, $\bar{q}$, and $\bar{x}$ for traffic intensities of 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, and 1.2. (c) Comment on how $\bar{l}$, $\bar{q}$, and $\bar{x}$ depend on the traffic intensity. (d) Relative to the case study, if it is decided that $\bar{q}$ greater than 5.0 is not acceptable, what systematic increase in service times would be acceptable? Use *d.dd* precision.

Exercise 1.2.3 (a) Modify program `ssq1` by adding the capability to compute the maximum delay, the number of jobs in the service node at a specified time (known at compile time) and the proportion of jobs delayed. (b) What was the maximum delay experienced? (c) How many jobs were in the service node at $t = 400$ and how does the computation of this number relate to the proof of Theorem 1.2.1? (d) What proportion of jobs were delayed and how does this proportion relate to the utilization?

Exercise 1.2.4 Complete the proof of Theorem 1.2.1.

Exercise 1.2.5 If the traffic intensity is less than 1.0, use Theorem 1.2.1 to argue why for large n you would expect to find that $\bar{l} \cong \lambda\,\bar{w}$, $\bar{q} \cong \lambda\,\bar{d}$, and $\bar{x} \cong \lambda\,\bar{s}$, where the observed arrival rate is $\lambda = 1/\bar{r}$.

Exercise 1.2.6 The text file `ac.dat` consists of the arrival times $a_1, a_2, \ldots, a_n$ and the departure times $c_1, c_2, \ldots, c_n$ for $n = 500$ jobs in the format

$$\begin{matrix} a_1 & c_1 \\ a_2 & c_2 \\ \vdots & \vdots \\ a_n & c_n \end{matrix}$$

(a) If these times are for an initially idle single-server FIFO service node with infinite capacity, calculate the average service time, the server's utilization and the traffic intensity. (b) Be explicit: for $i = 1, 2, \ldots, n$ how does s_i relate to a_{i-1}, a_i, c_{i-1}, and c_i?

Exercise 1.2.7[a] State and prove a theorem analogous to Theorem 1.2.1 but valid for *any* queue discipline. *Hint*: in place of c_n use $\tau_n = \max\{c_1, c_2, \ldots, c_n\}$. For a conservative server prove that τ_n is *independent* of the queue discipline.

Exercise 1.2.8 (a) Similar to Exercise 1.2.2, modify program `ssq1` to output the additional statistics $\bar{l}$, $\bar{q}$, and $\bar{x}$. (b) By using the arrival times in the file `ssq1.dat` and an appropriate *constant* service time in place of the service times in the file `ssq1.dat`, use the modified program to compute a table of $\bar{l}$, $\bar{q}$, and $\bar{x}$ for traffic intensities of 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, and 1.2. (c) Comment on how $\bar{l}$, $\bar{q}$, and $\bar{x}$ depend on the traffic intensity.

Exercise 1.2.9[a] (a) Work Exercises 1.2.2 and 1.2.8. (b) Compare the two tables produced and explain (or conjecture) why the two tables are different. Be specific.

The inputs to program `ssq1`, the arrival times and the service times, can have any positive real value — they are *continuous* variables. In some models, however, the input variables are inherently *discrete*. That is the case with the (trace-driven) discrete-event simulation model of a simple inventory system constructed in this section. As in the previous section, we begin with a conceptual model then move to a specification model and, finally, to a computational model.

1.3.1 CONCEPTUAL MODEL

Definition 1.3.1 An *inventory system* consists of a facility that distributes items from its current inventory to its customers in response to a customer demand that is typically random, as illustrated in Figure 1.3.1. Moreover, the demand is integer-valued (discrete) because customers do not want a portion of an item.* Because there is a *holding cost* associated with items in inventory, it is undesirable for the inventory level to be too high. On the other hand, if the inventory level is too low, the facility is in danger of incurring a *shortage cost* whenever a demand occurs that cannot be met.

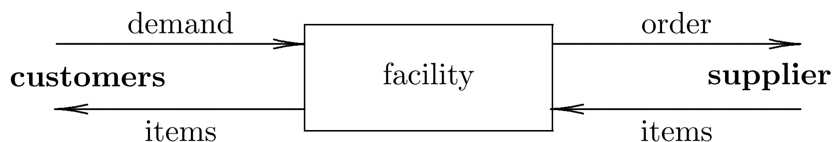


Figure 1.3.1. Simple inventory system diagram.

As a policy, the inventory level is reviewed periodically and new items are then (and only then) ordered from a supplier, if necessary.** When items are ordered, the facility incurs an *ordering cost* that is the sum of a fixed *setup cost* independent of the amount ordered plus an *item cost* proportional to the number of items ordered. This *periodic inventory review policy* is defined by two parameters, conventionally denoted s and S.

- s is the *minimum* inventory level — if at the time of review the current inventory level is below the threshold s then an order will be placed with the supplier to replenish the inventory. If the current inventory level is at or above s then no order will be placed.
- S is the *maximum* inventory level — when an order is placed, the amount ordered is the number of items required to bring the inventory back up to the level S.
- The (s, S) parameters are constant in time with $0 \leq s < S$.

* Some inventory systems distribute "items" that are not inherently discrete, for example, a service station that sells gasoline. With minor modifications, the model developed in this section is applicable to these inventory systems as well.

** An alternate to the periodic inventory review policy is a *transaction reporting inventory policy*. With this policy, inventory review occurs after *each* demand instance. Because inventory review occurs more frequently, significantly more labor may be required to implement a transaction reporting inventory policy. (The scanners at a grocery store, however, require no extra labor). The transaction reporting policy has the desirable property that, for the same value of s, it is less likely for the inventory system to experience a shortage.

A discrete-event simulation model can be used to compute the cost of operating the facility. In some cases, the values of s and S are fixed; if so, the cost of operating the facility is also fixed. In other cases, if at least one of the (s, S) values (usually s) is not fixed, the cost of operating the facility can be modified and it is natural to search for values of (s, S) for which the cost of operating the facility is minimized.

To complete the conceptual model of this simple (one type of item) inventory system we make three additional assumptions. (a) *Back ordering* (backlogging) is possible — the inventory level can become negative in order to model customer demands not immediately satisfied. (b) There is no *delivery lag* — an order placed with the supplier will be delivered immediately. Usually this is an unrealistic assumption; it will be removed in Chapter 3. (c) The *initial* inventory level is S.

Example 1.3.5, presented later in this section, describes an automobile dealership as an example of an inventory system with back ordering and no delivery lag. In this example the periodic inventory review occurs each week. The value of S is fixed, the value of s is not.

1.3.2 SPECIFICATION MODEL

The following variables provide the basis for a specification model of a simple inventory system. Time begins at $t = 0$ and is measured in a coordinate system in which the inventory review times are $t = 0, 1, 2, 3, \ldots$ with the convention that the ith time interval begins at time $t = i - 1$ and ends at $t = i$.

- The inventory level at the *beginning* of the ith time interval is an integer l_{i-1}.
- The amount ordered (if any) at time $t = i - 1$ is an integer $o_{i-1} \geq 0$.
- The demand quantity *during* the ith time interval is an integer $d_i \geq 0$.

Because we have assumed that back ordering is possible, if the demand during the ith time interval is greater than the inventory level at the beginning of the interval (plus the amount ordered, if any) then the inventory level at the end of the interval will be negative.

The inventory level is reviewed at $t = i - 1$. If l_{i-1} is greater than or equal to s then no items are ordered so that $o_{i-1} = 0$. If instead l_{i-1} is less than s then $o_{i-1} = S - l_{i-1}$ items are ordered to replenish inventory. In this case, because we have assumed there is no delivery lag, ordered items are delivered immediately (at $t = i - 1$) thereby restoring inventory to the level S. In either case, the inventory level at the end of the ith time interval is diminished by d_i. Therefore, as summarized by Algorithm 1.3.1, with $l_0 = S$ the inventory orders $o_0, o_1, o_2, \ldots$ and corresponding inventory levels $l_1, l_2, \ldots$ are defined by

$$o_{i-1} = \begin{cases} 0 & l_{i-1} \geq s \\ S - l_{i-1} & l_{i-1} < s \end{cases} \qquad \text{and} \qquad l_i = l_{i-1} + o_{i-1} - d_i.$$

Note that $l_0 = S > s$ and so o_0 must be zero; accordingly, only $o_1, o_2, \ldots$ are of interest.

Algorithm 1.3.1 If the demands $d_1, d_2, \ldots$ are known then this algorithm computes the discrete time evolution of the inventory level for a simple (s, S) inventory system with back ordering and no delivery lag.

```
l0 = S;                              /* the initial inventory level is S */
i = 0;
while ( more demand to process ) {
   i++;
   if (l(i-1) < s)
     o(i-1) = S - l(i-1);
   else
     o(i-1) = 0;
   d(i) = GetDemand();
   l(i) = l(i-1) + o(i-1) - d(i);
}
n = i;
o(n) = S - l(n);
l(n) = S;                            /* the terminal inventory level is S */
return l1, l2, ..., ln and o1, o2, ..., on;
```

Example 1.3.1 Let $(s, S) = (20, 60)$ and apply Algorithm 1.3.1 to process $n = 12$ time intervals of operation with the input demand schedule:

	i	:	1	2	3	4	5	6	7	8	9	10	11	12
input	d_i	:	30	15	25	15	45	30	25	15	20	35	20	30

As illustrated in Figure 1.3.2, the time evolution of the inventory level typically features several intervals of decline, followed by an increase when an order is placed (indicated by the vertical dotted line) and, because there is no delivery lag, is immediately delivered.

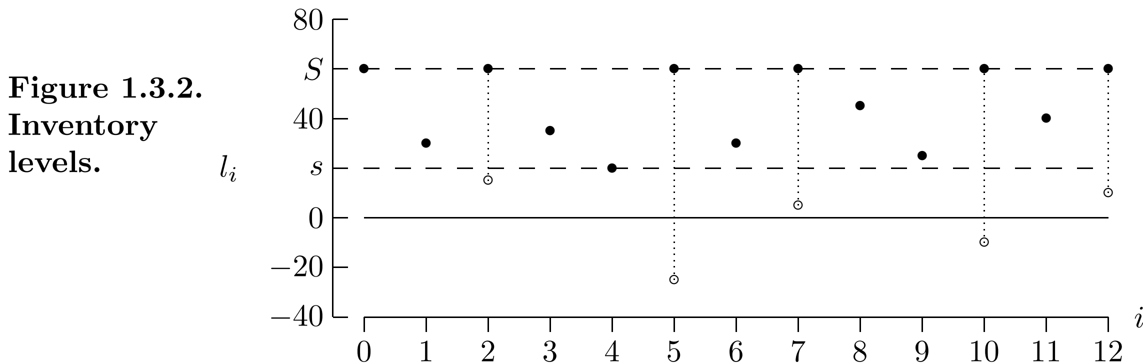


Figure 1.3.2. Inventory levels.

At the end of the last interval (at $t = n = 12$) an order for $o_n = 50$ inventory units was placed. The immediate delivery of this order restores the inventory level at the end of the simulation to the initial inventory level S, as shown in Figure 1.3.2.

1.3.3 OUTPUT STATISTICS

As with the development of program `ssq1` in the previous section, we must address the issue of what statistics should be computed to measure the performance of a simple inventory system. As always, our objective is to analyze these statistics and, by so doing, better understand how the system operates.

Definition 1.3.2 The *average demand* and *average order* are, respectively

$$\bar{d} = \frac{1}{n}\sum_{i=1}^{n} d_i \qquad \text{and} \qquad \bar{o} = \frac{1}{n}\sum_{i=1}^{n} o_i.$$

Example 1.3.2 For the data in Example 1.3.1, $\bar{d} = \bar{o} = 305/12 \cong 25.42$ items per time interval. As explained next, these two averages *must* be equal.

The terminal condition in Algorithm 1.3.1 is that at the end of the nth time interval an order is placed to return the inventory to its initial level. Because of this terminal condition, independent of the value of s and S, the average demand $\bar{d}$ and the average order $\bar{o}$ must be equal. That is, over the course of the simulated period of operation, all demand is satisfied (although not immediately when back ordering occurs). Therefore, if the inventory level is the same at the beginning and end of the simulation then the average "flow" of items into the facility from the supplier, $\bar{o}$, must have been equal to the average "flow" of items out of the facility to the customers, $\bar{d}$. With respect to the flow of items into and out of the facility, the inventory system is said to be *flow balanced.*

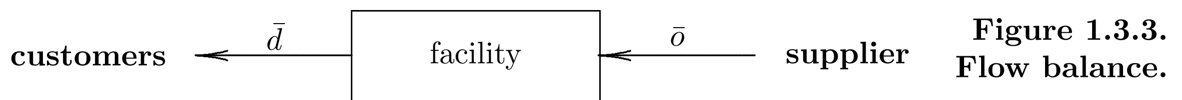


Figure 1.3.3. Flow balance.

Average Inventory Level

The holding cost and shortage cost are proportional to time-averaged inventory levels. To compute these averages it is necessary to know the inventory level for *all* t, not just at the inventory review times. Therefore, we *assume* that the *demand rate* is constant between review times so that the continuous time evolution of the inventory level is *piecewise linear* as illustrated in Figure 1.3.4.

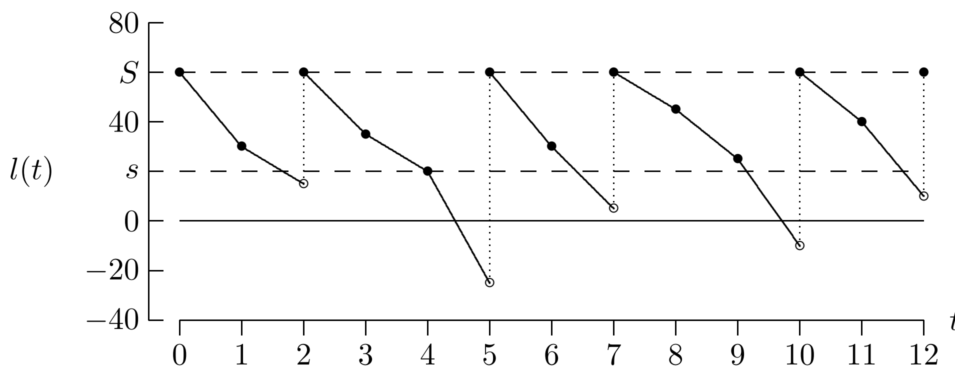


Figure 1.3.4. Piecewise linear inventory levels.

Definition 1.3.3 If the demand rate is constant between review times, then at any time t in the ith time interval the inventory level is $l(t) = l'_{i-1} - (t - i + 1)d_i$, as illustrated in Figure 1.3.5.

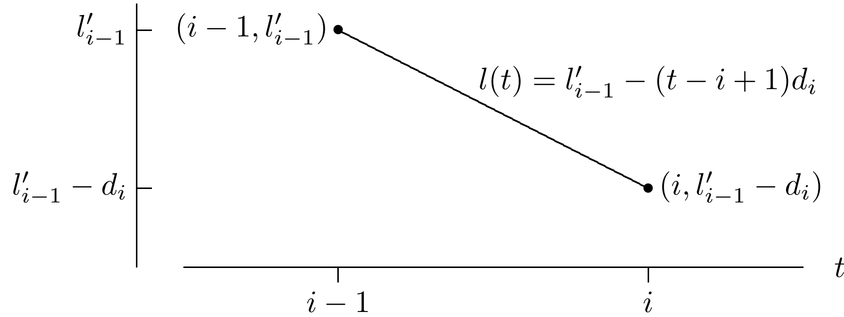


Figure 1.3.5. Linear inventory level in time interval i.

In this figure and related figures and equations elsewhere in this section, $l'_{i-1} = l_{i-1} + o_{i-1}$ represents the inventory level *after* inventory review. Accordingly, $l'_{i-1} \geq s$ for all i. (For the figure in Example 1.3.1, the $\circ$'s and $\bullet$'s represent l_{i-1} and l'_{i-1} respectively).

The equation for $l(t)$ is the basis for calculating the time-averaged inventory level for the ith time interval.* There are two cases to consider. If $l(t)$ remains non-negative over the ith time interval then there is only a time-averaged *holding level* integral to evaluate:

$$\bar{l}_i^+ = \int_{i-1}^{i} l(t)\, dt.$$

If instead $l(t)$ becomes negative at some time τ interior to the ith interval then, in addition to a time-averaged holding level integral, there is also a time-averaged *shortage level* integral to evaluate. In this case the two integrals are

$$\bar{l}_i^+ = \int_{i-1}^{\tau} l(t)\, dt \qquad \text{and} \qquad \bar{l}_i^- = -\int_{\tau}^{i} l(t)\, dt.$$

---

* Because the inventory level at any time is an *integer*, the figure in Definition 1.3.3 is technically incorrect. Instead, rounding to an integer value should be used to produce the inventory level time history illustrated in Figure 1.3.6. ($\lfloor z \rfloor$ is the floor function; $\lfloor z + 0.5 \rfloor$ is z rounded to the nearest integer.)

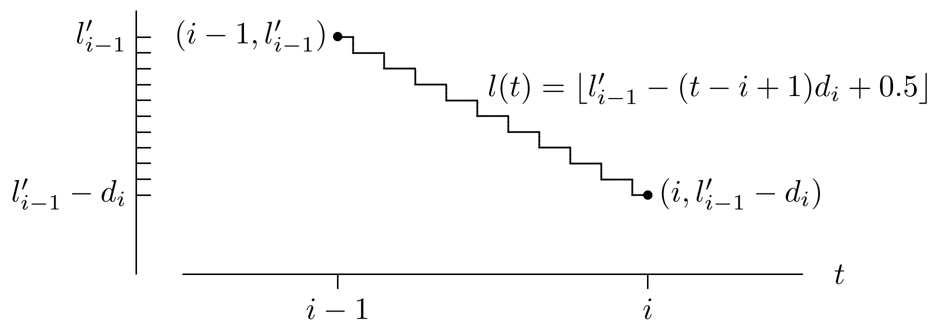


Figure 1.3.6. Piecewise constant inventory level in time interval i.

It can be shown, however, that rounding has *no* effect on the value of $\bar{l}_i^+$ and $\bar{l}_i^-$.

No Back Ordering

The inventory level $l(t)$ remains non-negative throughout the ith time interval if and only if the inventory level at the end of this interval is non-negative, as in Figure 1.3.7.

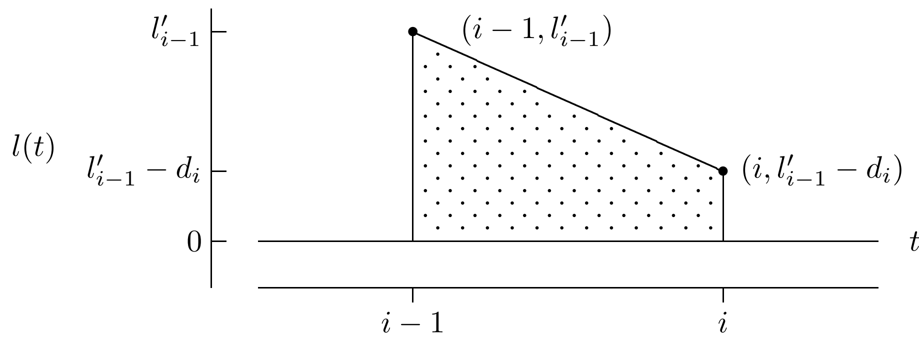


Figure 1.3.7. Inventory level in time interval i with no backordering.

Therefore, there is no shortage during the ith time interval if and only if $d_i \le l'_{i-1}$. In this case the time-averaged holding level integral for the ith time interval can be evaluated as the area of a trapezoid so that

$$\bar{l}_i^+ = \int_{i-1}^{i} l(t)\,dt = \frac{l'_{i-1} + (l'_{i-1} - d_i)}{2} = l'_{i-1} - \frac{1}{2}d_i \qquad \text{and} \qquad \bar{l}_i^- = 0.$$

With Back Ordering

The inventory level becomes negative at some point τ in the ith time interval if and only if $d_i > l'_{i-1}$, as illustrated in Figure 1.3.8.

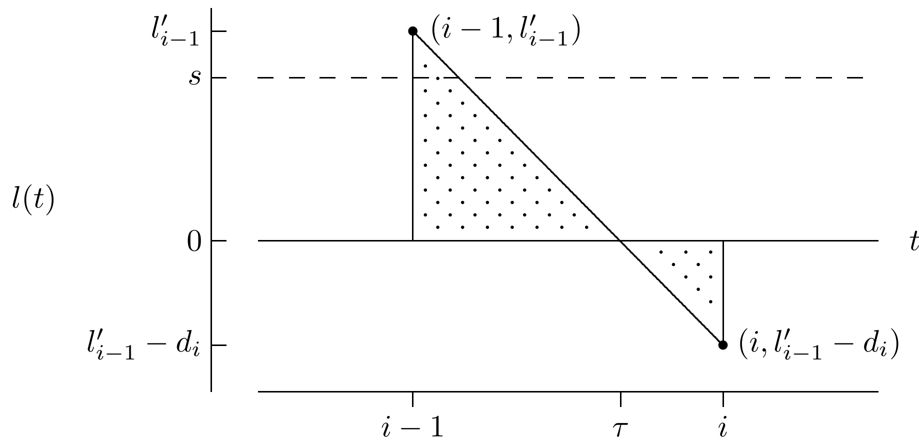


Figure 1.3.8. Inventory level in time interval i with backordering.

By using similar triangles, it can be shown that $\tau = i - 1 + (l'_{i-1}/d_i)$. In this case, the time-averaged holding level integral and shortage level integral for the ith time interval can be evaluated as the area of a triangle so that

$$\bar{l}_i^+ = \int_{i-1}^{\tau} l(t)\,dt = \cdots = \frac{(l'_{i-1})^2}{2d_i} \qquad \text{and} \qquad \bar{l}_i^- = -\int_{\tau}^{i} l(t)\,dt = \cdots = \frac{(d_i - l'_{i-1})^2}{2d_i}.$$

The time-averaged holding level and shortage level for each time interval can be summed over all intervals with the resulting sums divided by the number of intervals. Consistent with Definition 1.3.4, the result represents the average number of items "held" and "short" respectively, with the average taken over all time intervals.

Definition 1.3.4 The *time-averaged holding level* and the *time-averaged shortage level* are, respectively

$$\bar{l}^+ = \frac{1}{n}\sum_{i=1}^{n} \bar{l}_i^+ \qquad \text{and} \qquad \bar{l}^- = \frac{1}{n}\sum_{i=1}^{n} \bar{l}_i^- .$$

It is potentially confusing to define the time-averaged shortage level as a *positive* number, as we have done in Definition 1.3.3. In particular, it would be a mistake to compute the time-averaged inventory level as the sum of $\bar{l}^+$ and $\bar{l}^-$. Instead, the *time-averaged inventory level* is the difference

$$\bar{l} = \frac{1}{n}\int_0^n l(t)\,dt = \bar{l}^+ - \bar{l}^- .$$

The proof of this result is left as an exercise.

Example 1.3.3 For the data in Example 1.3.1, $\bar{l}^+ = 31.74$ and $\bar{l}^- = 0.70$. Therefore, over the 12 time intervals, the average number of items held was 31.74, the average number of items short was 0.70, and the average inventory level was 31.04.

1.3.4 COMPUTATIONAL MODEL

Algorithm 1.3.1 is the basis for program `sis1` presented at the end of this section — a trace-driven computational model of a simple inventory system.

Program sis1

Program `sis1` computes five statistics: $\bar{d}$, $\bar{o}$, $\bar{l}^+$, $\bar{l}^-$ and the order frequency $\bar{u}$, which is

$$\bar{u} = \frac{\text{number of orders}}{n}.$$

Because the simulated system is flow balanced, $\bar{o} = \bar{d}$ and so it would be sufficient for program `sis1` to compute just one of these two statistics. The independent computation of both $\bar{o}$ and $\bar{d}$ is desirable, however, because it provides an important consistency check for a (flow balanced) simple inventory system.

Example 1.3.4 Program `sis1` reads input data corresponding to $n = 100$ time intervals from the file `sis1.dat`. With the inventory policy parameter values $(s, S) = (20, 80)$ the results (with *dd.dd* precision) are

$$\bar{o} = \bar{d} = 29.29 \qquad \bar{u} = 0.39 \qquad \bar{l}^+ = 42.40 \qquad \bar{l}^- = 0.25.$$

As with program `ssq1`, in Chapter 3 we will free program `sis1` from its reliance on external data by using randomly generated demand data instead.

1.3.5 OPERATING COST

Definition 1.3.5 In conjunction with the four statistics $\bar{o}$, $\bar{u}$, $\bar{l}^+$ and $\bar{l}^-$, a facility's cost of operation is determined by four constants:

- c_{item} — the (unit) cost of a new item;
- c_{setup} — the setup cost associated with placing an order;
- c_{hold} — the cost to hold one item for one time interval;
- c_{short} — the cost of being short one item for one time interval.

Case Study

Consider a hypothetical automobile dealership that uses a weekly periodic inventory review policy. The facility is the dealer's showroom, service area and surrounding storage lot and the items that flow into and out of the facility are new cars. The supplier is the manufacturer of the cars and the customers are people convinced by clever advertising that their lives will be improved significantly if they purchase a new car from this dealer.

Example 1.3.5 Suppose space in the facility is limited to a maximum of, say $S = 80$, cars. (This is a small dealership.) Every Monday morning the dealer's inventory of cars is reviewed and if the inventory level at that time falls below a threshold, say $s = 20$, then enough new cars are ordered from the supplier to restock the inventory to level S.*

- The (unit) cost to the dealer for each new car ordered is $c_{\text{item}} = \$8000$.
- The setup cost associated with deciding what cars to order (color, model, options, etc.) and arranging for additional bank financing (this is not a rich automobile dealer) is $c_{\text{setup}} = \$1000$, independent of the number ordered.
- The holding cost (interest charges primarily) to the dealer, per week, to have a car sit unsold in his facility is $c_{\text{hold}} = \$25$.
- The shortage cost to the dealer, per week, to not have a car in inventory is hard to determine because, in our model, we have assumed that *all* demand will ultimately be satisfied. Therefore, any customer who wants to buy a new car, even if none are available, will agree to wait until next Monday when new cars arrive. Thus the shortage cost to the dealer is primarily in goodwill. Our dealer realizes, however, that in this situation customers may buy from another dealer and so, when a shortage occurs, he sweetens his deals by agreeing to pay "shorted" customers \$100 cash *per day* when they come back on Monday to pick up their new car. This means that the cost of being short one car for one week is $c_{\text{short}} = \$700$.

---

* There will be some, perhaps quite significant, delivery lag but that is ignored, for now, in our model. In effect, we are assuming that this dealer is located adjacent to the supplier and that the supplier responds immediately to each order.

Definition 1.3.6 A simple inventory system's average operating costs *per time interval* are defined as follows:

- item cost: $c_{\text{item}} \cdot \bar{o}$;
- setup cost: $c_{\text{setup}} \cdot \bar{u}$;
- holding cost: $c_{\text{hold}} \cdot \bar{l}^{+}$;
- shortage cost: $c_{\text{short}} \cdot \bar{l}^{-}$.

The average total cost of operation per time interval is the sum of these four costs. This sum multiplied by the number of time intervals is the total cost of operation.

Example 1.3.6 From the statistics in Example 1.3.4 and the constants in Example 1.3.5, for our auto dealership the average costs are:

- the item cost is $\$8000 \cdot 29.29 = \$234,320$;
- the setup cost is $\$1000 \cdot 0.39 = \$390$;
- the holding cost is $\$25 \cdot 42.40 = \$1060$;
- the shortage cost is $\$700 \cdot 0.25 = \$175$.

Each of these costs is a per week average.

Cost Minimization

Although the inventory system statistic of primary interest is the average total cost per time interval, it is important to know the four components of this total cost. By varying the value of s (and possibly S) it seems reasonable to expect that an optimal (minimal average cost) periodic inventory review policy can be determined for which these components are properly balanced.

In a search for optimal (s, S) values, because $\bar{o} = \bar{d}$ and $\bar{d}$ depends only on the demand sequence, it is important to note that the item cost is *independent* of (s, S). Therefore, the only cost that can be controlled by adjusting the inventory policy parameters is the sum of the average setup, holding, and shortage costs. In Example 1.3.7, this sum is called the average *dependent cost*. For reference, in Example 1.3.6 the average dependent cost is $\$390 + \$1060 + \$175 = \$1625$ per week.

If S and the demand sequence is fixed, and if s is systematically increased, say from 0 to some large value less than S, then we expect to see the following.

- Generally, the average setup cost and holding cost will increase with increasing s.
- Generally, the average shortage cost will decrease with increasing s.
- Generally, the average total cost will have a 'U' shape indicating the presence of one (or more) optimal value(s) of s.

Example 1.3.7 is an illustration.

Example 1.3.7 With S fixed at 80, a modified version of program `sis1` was used to study how the total cost relates to the value of s. That is, the cost constants in Example 1.3.5 were used to compute the average setup, holding, shortage, and dependent cost for a range of s values from 1 to 40. As illustrated in Figure 1.3.9, the minimum average dependent cost is approximately \$1550 at $s = 22$.

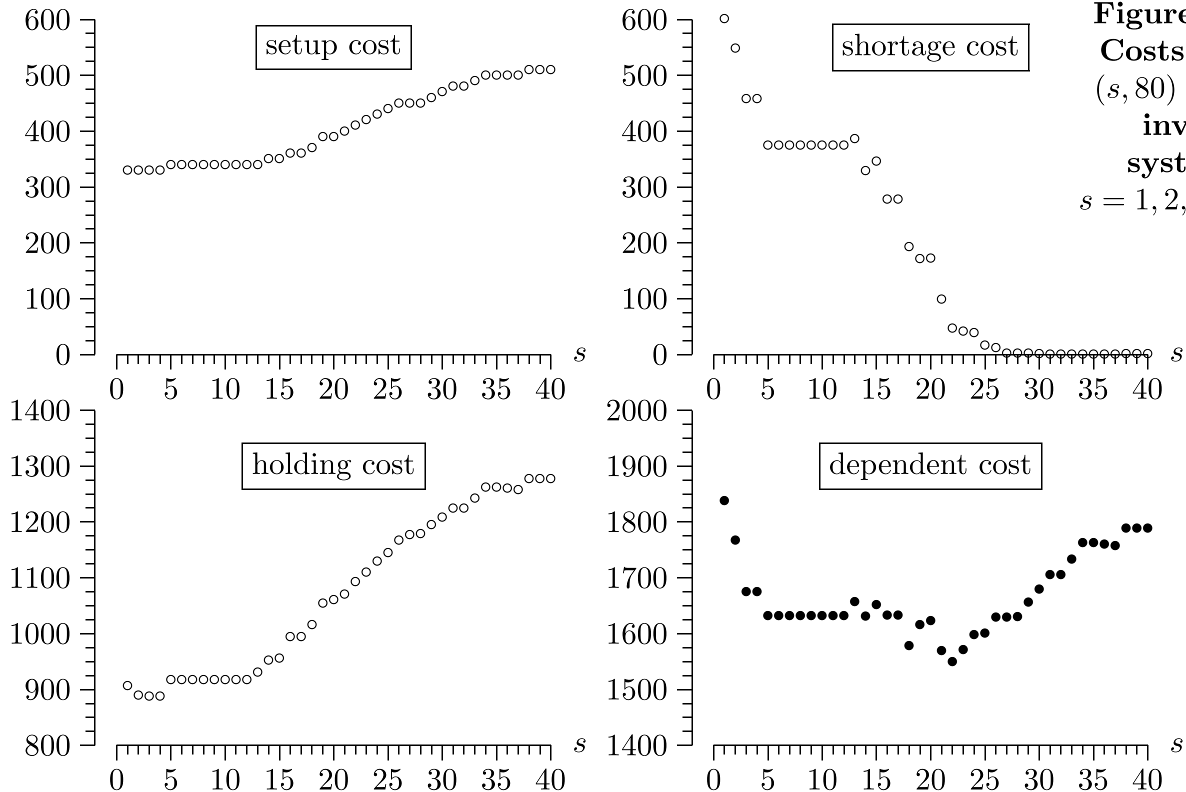


Figure 1.3.9. Costs for an $(s, 80)$ simple inventory system for $s = 1, 2, \ldots, 40$.

As in the case study concerning the ice cream shop, the "raw" simulation output data is presented. In this case, however, because the parameter s is inherently integer-valued (and so there is no "missing" output data at, say, $s = 22.5$) no interpolating or approximating curve is superimposed. [For a more general treatment of issues surrounding simulation optimization, see Andradóttir (1998).]

1.3.6 EXERCISES

Exercise 1.3.1 Verify that the results in Example 1.3.1 and the averages in Examples 1.3.2 and 1.3.3 are correct.

Exercise 1.3.2 (a) Using the cost constants in Example 1.3.5, modify program `sis1` to compute all four components of the total average cost per week. (b) These four costs may differ somewhat from the numbers in Example 1.3.6. Why? (c) By constructing a graph like that in Example 1.3.7, explain the trade-offs involved in concluding that $s = 22$ is the optimum value (when $S = 80$). (d) Comment on how well-defined this optimum is.

Exercise 1.3.3 Suppose that the inventory level $l(t)$ has a constant rate of change over the time interval $a \le t \le b$ and both $l(a)$ and $l(b)$ are integers. (*a*) Prove that

$$\int_a^b l(t)\,dt = \int_a^b \lfloor l(t) + 0.5 \rfloor\,dt = \frac{1}{2}(b-a)\big(l(a) + l(b)\big).$$

(*b*) What is the value of this integral if $l(t)$ is truncated rather than rounded (i.e., if the 0.5 is omitted in the second integral)?

Exercise 1.3.4 (*a*) Construct a table or figure similar to Figure 1.3.7 but for $S = 100$ and $S = 60$. (*b*) How does the minimum cost value of s seem to depend on S? (See Exercise 1.3.2.)

Exercise 1.3.5 Provided there is no delivery lag, prove that if $d_i \le s$ for $i = 1, 2, \ldots, n$, then $\bar{l}^- = 0$.

Exercise 1.3.6 (*a*) Provided there is no delivery lag, prove that if $S - s < d_i \le S$ for $i = 1, 2, \ldots, n$ then $\bar{l}^+ = S - \bar{d}/2$. (*b*) What is the value of $\bar{l}^-$ and $\bar{u}$ in this case?

Exercise 1.3.7 Use Definitions 1.3.3 and 1.3.4 to prove that the average inventory level equation

$$\bar{l} = \frac{1}{n}\int_0^n l(t)\,dt = \bar{l}^+ - \bar{l}^-$$

is correct. *Hint*: use the $(\cdot)^+$ and $(\cdot)^-$ functions defined for any integer (or real number) x as

$$x^+ = \frac{|x| + x}{2} \qquad \text{and} \qquad x^- = \frac{|x| - x}{2}$$

and recognize that $x = x^+ - x^-$.

Exercise 1.3.8 (*a*) Modify program `sis1` so that the demands are first read into a circular array, then read out of that array, as needed, during program execution. (*b*) By experimenting with different starting locations for reading the demands from the circular array, explore how sensitive the program's statistical output is to the order in which the demands occur.

Exercise 1.3.9[a] (*a*) Consider a variant of Exercise 1.3.8, where you use a conventional (non-circular) array and randomly *shuffle* the demands within this array before the demands are then read out of the array, as needed, during program execution. (*b*) Repeat for at least 10 different random shuffles and explore how sensitive the program's statistical output is to the order in which the demands occur.