II. DISCRETE VENT SIMULATION

2.1. Introduction

Simulation modelling has been used in a wide range of physical and social sciences and engineering fields, ranging from nuclear fusion to economic forecast to space shuttle design. For different types of situations and systems, different types of models are used. In classifying simulations, there are important distinctions among the types of models that are being simulated, and among the types of program structures that are used to carry out the simulation.

2.2. Deterministic Simulation Models

Simulation models may be either deterministic or stochastic (meaning probabilistic). In a **deterministic** simulation, all of the events and relationships among the variables are governed entirely by a combination of known, but possibly complicated, rules.

Deterministic simulations are often used to study the behaviour of physical systems. As a simple example, consider the problem of determining the time required to pay off a $100 loan if you pay $10 per month and the interest rate is 10 percent per year, compounded monthly. In this case, the answer (namely 11 months) can be found very easily, by writing a simple program to ‘‘simulate’’ the history of the loan:

```
const rate := .10 / 12  % interest rate per month, expressed as a fraction
const payment := 10     % payment per month
var loan : real := 100  % current value of the loan (initially $100)
var month :=0           % current month

put "end of month:" : 15, "loan balance:" : 15, "payment due:" : 10
put month : 10, loan : 15 : 2

loop
    month += 1
    loan *= (1 + rate)
    const thisPayment := min (payment, loan)
    loan -= thisPayment
    put month : 10, loan : 15 : 2, thisPayment : 15 : 2
    exit when loan <= 0.0
end loop
```

By running the program, we see that the answer is 11 months:

```
end of month:  loan balance:  payment due:
         0         100.00
         1          90.83          10.00
         2          81.59          10.00
         3          72.27          10.00
         4          62.87          10.00
         5          53.40          10.00
         6          43.84          10.00
         7          34.21          10.00
         8          24.49          10.00
         9          14.70          10.00
        10           4.82          10.00
        11           0.00           4.86
```

In this case, it was not really necessary to resort to simulation, since standard mathematical techniques for dealing with series can be used to obtain a general solution to the number of loan periods, P, required assuming the periodic payment is a fraction D of the initial loan, and the interest per period is R:

$$P = \frac{\ln(D) - \ln(D - R)}{\ln(1 + R)},$$

which in this case tells us the answer is 10.48 months.

The advantage of simulation is that you can still answer the question even if the model is too complicated to solve analytically. For example, consider the trajectory of a baseball under the combined influence of gravity and aerodynamic drag. Once the ball leaves the pitcher’s fingertips, with a given orientation (i.e., the direction that the seams are facing), speed, direction and spin (i.e., the axis of rotation and rotational speed), we can in principle determine its flight path to the catcher by assuming that gravity causes it to fall towards the ground with a constant rate of acceleration, and that at the same time aerodynamic drag causes it to slow down its rate of spin, reduce its speed, and change its direction of flight, at rates that depend on its current orientation, speed and spin. We could use a deterministic simulation to estimate the flight path of the ball by, in effect, ‘‘flying’’ the ball. Periodically, we take a snapshot of the current state of the ball (i.e., its position, speed, spin, etc.), and then calculate all the forces acting on the ball in that state. We then let the ball move ahead a short distance under the combined influence of those forces, assuming that they remain unchanged until we take the next snapshot, recalculate all the forces, and so on. In general, the assumption of unchanging forces causes an error in the resulting solution, so we must limit the distance between successive recalculations of the forces to be small enough for the error to be acceptable. Nevertheless, when the forces acting on the ball are complicated, it can be much simpler to find the flight path from such a deterministic simulation than it would be to solve all the equations to obtain an analytic solution!

2.3. Stochastic Simulation Models

In a **stochastic** simulation, ‘‘random variables’’ are included in the model to represent the influence of factors that are unpredictable, unknown, or beyond the scope of the model we use in the simulation. Throughout the rest of this chapter, we will be discussing stochastic simulation models.

In many applications, such as a model of the tellers in a bank, it makes sense to incorporate random variables into the model. In the case of a bank, we might wish to assume that there is a stream of anonymous customers coming in the door at unpredictable times, rather than explicitly modelling the behaviour of each of their actual customers to determine when they plan to go to the bank.

It is worth noting here that it is well known in statistics that when we combine the actions of a large population of more-or-less *independently* operating entities (customers, etc.) the resulting behaviour appears to have been randomly produced, and that the patterns of activity followed by the individual entities within that population are unimportant. For example, all of the telephone systems designed in the last 60 years are based on the (quite justified) assumption that the number of calls made in fixed length time intervals obeys the Poisson distribution. Thus the generation and use of random variables is an important topic in simulation, and we will discuss it in a separate section below.

2.4. Static Versus Dynamic Simulation Models

Another dimension along which simulation models can be classified is that of time. If the model is used to simulate the operation of a system over a period of time, it is **dynamic**. The baseball example above uses dynamic simulation. On the other hand, if no time is involved in a model, it is **static**. Many gambling situations (e.g., dice, roulette) can be simulated to determine the odds of winning or losing. Since only the number of bets made, rather than the duration of gambling, matters, static simulation models are appropriate for them.

Monte Carlo Simulation (named after a famous casino town[1] in Europe) refers to the type of simulation in which a static, approximate, and stochastic model is used for a deterministic system. This is in contrast to the baseball example at the beginning of this chapter, where both the system and the model are dynamic and deterministic. Let us now look at an example of Monte Carlo simulation. Consider estimating the value of π by finding the approximate area of a circle with a unit radius. The first quadrant of the circle is enclosed by a unit square (See Figure 1). If pairs of uniformly distributed pseudo-random values for the x and y coordinates in [0, 1] are generated, the probability of the corresponding points falling in the quarter circle is simply the ratio of the area of the quarter circle to that of the unit square:

[1] Interestingly, John Von Neumann, a pioneer in computer design and use, used Monte Carlo as the code name for his military project during the Second World War.

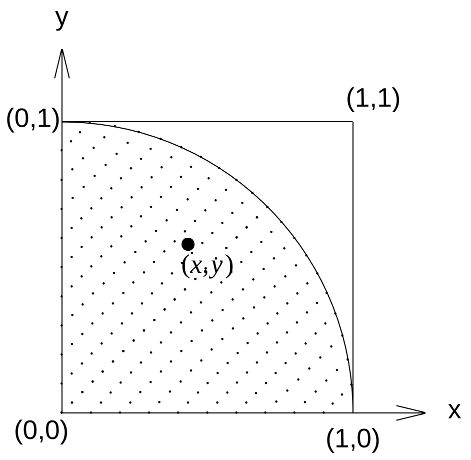

Figure 1. Monte Carlo Integration of a Quarter Circle.

$$\frac{\textit{number of points falling in the quarter circle}}{\textit{total number of points generated}} = \frac{\pi}{4}$$

The simple Turing program below simulates this situation. Some sample output follows. A good approximation can be achieved by using a large number of points. The convergence is quite slow, though. In the next chapter, we will discuss a faster way through numerical integration.

```
randseed(1, 1)% could have used interactive input to initialize, but this is simpler
randseed(2, 2)

var x, y: real
var repeatCount, inCount := 0

put "repetition count: " ..
get repeatCount

for i: 1 .. repeatCount
   randnext(x, 1)
   randnext(y, 1)
   if x*x + y*y < 1 then
inCount += 1
   end if
end for

put "Approximate value of pi (based on ", repeatCount, " samples): ", 4 * (inCount / repeatCount)
```

Repetition count	Approximate value of π
100	3.24
1,000	3.16
10,000	3.1608
100,000	3.14508
1,000,000	3.14098

The above program is an example of *Monte Carlo integration*, by which definite integral of arbitrary but finite functions can be estimated. Consider the following integral:

$$\int_a^b f(x)dx$$

Such an integral can be partitioned into segments above or below the horizontal axis. Without loss of generality, let us assume $f(x) \geq 0$, $a \leq x \leq b$. We can then bound the curve with a rectangle with borders of $x = a$, $x = b$, $y = 0$, and $y = y_{\max}$, where $y_{\max}$ is the maximum value of $f(x)$ in the interval [a, b]. By generating random points uniformly distributed over this rectangular area, and deciding whether they fall above or below the curve $f(x)$, we can estimate the integral.

2.5. Program Structures for Dynamic Simulation

In the remainder of chapter, dynamic stochastic simulation models will be emphasized. A dynamic simulation program that is written in a general purpose programming language (such as Turing) must:

i) keep track of ‘‘simulated’’ time,

ii) schedule events at ‘‘simulated’’ times in the future, and

iii) cause appropriate changes in state to occur when ‘‘simulated’’ time reaches the time at which one or more events take place.

The structure of a simulation program may be either time-driven or event-driven, depending on the nature of the basic loop of the program. In **time-driven** models (see Figure 2a), each time through the basic loop, simulated time is advanced by some ‘‘small’’ (in relation to the rates of change of the variables in the program) fixed amount, and then each possible event type is checked to see which, if any, events have occurred at that point in simulated time. In **event-driven** models (see Figure 2b), events of various types are scheduled at future points in simulated time. The basic program loop determines when the next scheduled event should occur, as the minimum of the scheduled times of each possible event. Simulated time is then advanced to exactly that event time, and the corresponding event handling routine is executed to reflect the change of state that occurs as a result of that event.

```
initialize
time := startingTime
loop
    exit when time >= endingTime
    % Collect statistics and record measurements.
    % Call event routines for each type of event
    % that has occurred by this time.
    time += timeIncrement
end loop
```

a. Time Driven Structure

```
initialize
TimeToStop := false
loop
    exit when TimeToStop
    % remove entry for next event from event list
    time := NextEventTime
    % Collect statistics and record measurements.
    % Call event routine for next event
    % (end of simulation event sets TimeToStop).
    % Insert next occurrence of this event type
    % in event list.
end loop
```

b. Event Driven Structure

Figure 2. Structure of the basic program loop.

The choice of whether a time- or event-driven program structure should be used depends on the nature of the model. In general, an event driven structure means that the basic program loop gets executed fewer times (since an event is guaranteed to take place every time through the loop). However, there is overhead involved in managing the event list, so that the ‘‘cost’’ (i.e., execution time) of going around the loop once is, in general, substantially higher than it would be for a time-driven structure. Thus, a time-driven structure is generally better for models in which it is possible to find a time increment that is both small enough to distinguish among the times at which events occur that cannot be treated as simultaneous, and also large enough for at least one event to occur at a significant proportion of the times through the basic program loop. If such a compromise is not possible (because events occur at irregular intervals), an event-driven structure is preferable.

2.5.1. Time-Driven Dynamic Stochastic Simulation

Some dynamic systems have events occurring at regular intervals, so the time-driven structure discussed in Section 1 is more suitable than the event-driven structure (See Figure 2). Consider a ‘‘drunken’’ flea being placed at the center of a 20×20 plate (See Figure 3). The flea will make a sequence of random jumps, each being in one of the four directions, east, west, north, or south, and with a unit distance. How many jumps will it take for the flea to fall off the plate? There is apparently no definite answer to this question, because the process is stochastic in nature. If we assume that the flea makes its jumps at regular intervals, the appropriate type of model for this system would be time-driven, dynamic, and stochastic. We can use the following Turing program to find out the possible numbers of jumps.

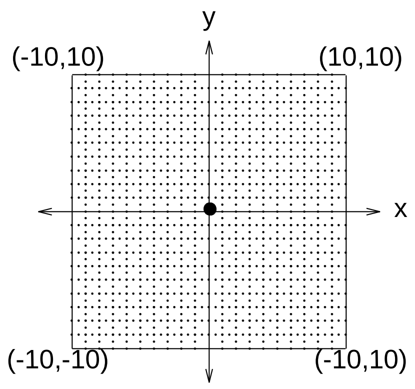

Figure 3. Drunken Flea on a Plate.

```
var x, y, step, direction: int := 0

randomize
loop
   exit when abs(x) > 10 or abs(y) > 10
   randint(direction, 1, 4)
   case direction of
         label 1: x += 1
         label 2: y += 1
         label 3: x −= 1
         label 4: y −= 1
   end case
   step += 1
end loop
put "Flea jumped off plate in ", step, " steps"
```

By running this program, you will find that the number of jumps varies (depending on the random seed). You can repeat the experiment a number of times, and compute the mean and standard deviation of the number of jumps.

2.5.2. Event-Driven Dynamic Stochastic Simulation

If an event-driven structure is used, the program must manage an *event list* in some manner. If the number of future events scheduled is large, then the determination of the next event to occur is most efficiently accomplished by maintaining a linked list of all scheduled events, ordered by the times at which they are scheduled to occur. Thus, each time through the basic program loop, the event description for the next event is (easily) removed from the front of the list and acted upon. However, whenever a future event needs to be scheduled, an event description must be inserted at an arbitrary point in the linked list. A way to improve the efficiency of insertion would be to use a **heap** data structure instead of an ordered linked list to store the events.

If the number of scheduled events remains small at all times, however, it is more efficient to simply set aside one variable for each type of event, and use it to hold the next time at which an event of that type is scheduled to occur. (If at some point no event of that type is scheduled to occur, then the value of the corresponding variable should be set to ‘‘infinity’’, i.e., to any value that is greater than the ending time for the simulation.) The basic loop of the program then determines the next scheduled event by scanning all the variables holding times of future events and selecting the one with the smallest value.

2.6. Constructing a Simulation Model

2.6.1. Identification of Components

Our first task is to identify those system components that should be included in the model. This choice depends not only on the real system, but also on the aspects of its behaviour that we intend to investigate. The only complete model of a system is the system itself. Any other model includes assumptions or simplifications. Some components of a system may be left out of the system model *if* their absence is not expected to alter the aspects of behaviour that we wish to observe. For example, returning to our bank teller model, we need not keep track of the activities of individual customers in order to determine the optimum number of tellers. However, we might have to track the activities of individual customers in our model to investigate customer satisfaction, assuming that customers can ‘‘balk’’ (i.e., leave without service) if they arrive when the waiting lines are too long, go about some other business for a few minutes, and then return to the bank. It is also important to keep track of individuals when the total population is small. For example, if we were to imagine the (highly unrealistic!) case that only two people had accounts in our bank, then we could not simply ‘‘toss a coin’’ to decide whether the next incoming customer is the businessman (with a long list of things for the teller to do) or the student (who merely wants to withdraw enough money to go to a movie). Otherwise, we may reach a state in which there are two businessmen (or two students) in the bank! (Although this example may sound far fetched, **sampling without replacement** can be very important, for example when selecting cards from a deck of playing cards.)

With the goal of formulating a simulation model in mind, we now proceed to list each of the necessary concepts. These concepts will be illustrated by applying them to the problem of simulating an automobile service station. The purpose of the simulation is to determine how many pumps the service station should keep running in order to maximize its profit. We assume that the problem specification includes the following information:

i) the times between successive arrivals of customers, expressed as a probability distribution,

ii) the distribution of the number of litres of gasoline needed by customers,

iii) the distribution of how long service at the pump takes as a function of the number of litres needed,

iv) the probability that an arriving customer will *balk* as a function of the number of cars already waiting at the service station and the number of litres of gasoline he needs,

v) the profit per litre of gasoline sold, and

vi) the cost per day of running a pump (including an attendant’s salary, etc.).

2.6.1.1. Entities

Customers, resources, service facilities, materials, service personnel, etc., are *entities*. Each type of entity has a set of relevant *attributes*. In our service station example, the entities are *cars*, with the attributes ‘arrival time’ and ‘number of litres needed’, and *pumps*, with the attribute ‘time to the completion of service to the current customer’.

A Turing **record** is well-suited for representing an entity and its associated attributes. A group of similar entities may be represented as either an **array** or a **collection** of records. An array is best suited for representing a group of ‘‘static’’ entities (i.e., those that are present throughout the simulation run, such as the pumps), while a collection is more appropriate for ‘‘dynamic’’ entities (i.e., those that enter the system, spend some time there, and then depart, such as the autos).

2.6.1.2. Events

Events are occurrences that alter the system state. Here the events are the *arrival* of a customer at the station, the *start-of-service* to a customer at a pump, and the *completion-of-service* to a customer at a pump. The first *arrival* event must be scheduled in the initialization routine; the remaining arrivals are handled by letting each invocation of the arrival routine schedule the next arrival event. The scheduling of a *start-of-service* event takes place either in the arrival routine, if there are no other cars waiting and a pump is available, or in the completion-of-service routine, otherwise. Each time the start-of-service routine is invoked, the *completion-of-service* at that pump is scheduled.

Besides the types of events identified in the real system, there are two other **pseudo-events** that should be included in every simulation program. *End-of-simulation* halts the simulation after a certain amount of simulated time has elapsed, and initiates the final output of the statistics and measurements gathered in the run. End-of-simulation should be scheduled during the initialization of an event-driven simulation; for time-driven simulations, it is determined by the upper bound of the basic program loop. A *progress-report* event allows a summary of statistics and measurements to be printed after specified intervals of simulated time have elapsed. These progress-report summaries can be used to check the validity of the program during its development, and also to show whether or not the system is settling down to some sort of ‘‘equilibrium’’ (i.e., stable) behaviour.

2.6.1.3. Groupings

Similar entities are often *grouped* in meaningful ways. Sometimes an *ordering* of the entities within a group is relevant. In the service station example, the groupings are *available* pumps (that are not currently serving any auto), *busy* pumps (ordered by the service completion time for the customer in service), and *autos* awaiting service (ordered by time of arrival). In a Turing program, such groupings can be represented as linked lists of records, as long as a suitable link field is included in the records for that entity type.

2.6.1.4. Relationships

Pairs of non-similar entities may be *related*. For example, a busy pump is related to the auto that is being served. Relationships can be indicated by including in the record for one of the entities in a pair a link to the other entity. In some cases, it may be desirable for each of the entities in the pair to have a link to the other entity. For example, the relationship between a busy pump and the auto being served can be represented by a link from the pump record to the corresponding auto record.

2.6.1.5. Stochastic variables

In a stochastic simulation, certain attributes of entities or events must be chosen ‘‘at random’’ from some probability distribution. In the service station example, these include the customer interarrival times (i.e., the times between successive arrivals), the number of litres needed, the service time (based on the number of litres needed), and whether the customer decides to balk (based on the length of the waiting queue and the number of litres needed).

In a Turing program, a value is calculated for each instance of a stochastic variable by generating a ‘‘pseudo-random’’ (i.e., random-appearing, but actually algorithmically computed) number using one of the pre-defined pseudo-random number generators in Turing (based on built-in functions *rand*, *randint*, and *randnext*), and then transforming it as necessary to match the specified probability distribution for the stochastic variable. For example, *randint* **(***n***,** *10***,** *60***)** could be used to set *n* to the number of litres needed by the next incoming auto, assuming that any integer value between 10 litres and 60 litres is equally likely to occur. See the section on **pseudo-random number generation** below for more details.

2.6.1.6. Strategies

Typically, a simulation experiment consists of comparing several alternative approaches to running the system to find out which one(s) maximize some measure of system ‘‘performance’’. In the case of the service station, the strategies consist of keeping different numbers of pumps in service.

All code that is not common to all of the strategies should be localized to facilitate changing strategies. Some strategies, such as the number of gasoline pumps, involve merely picking a number, so that changing the value of a single variable suffices to alter the strategy. In other cases, changing strategy may involve taking some actions in substantially different ways, such as replacing separate queues for each teller in the bank with a common queue feeding customers to all of the tellers. Each particular strategy might then be represented by a separate subprogram, and changing the strategy could done by invoking different subprograms to carry out a certain action.

2.6.1.7. Measurements

The activity of the system will be reflected in *measurements* associated with entities, events, groups, or relationships. Such measurements are used in specifying the performance of the system. If the measurement is associated with an entity, it should be recorded in a field contained within the record for that entity. For measurements associated with events, groupings or relationships, additional variables or records must be declared to record the measured quantities.

In our example system, we might wish to measure the numbers of customers served, litres of gasoline sold, customers who balk, and litres of gasoline *not* sold to balking customers, the total waiting time (from which we can calculate average waiting time), the total service time (to get pump utilization), and the total time that the waiting queue is empty.

The number of customers served and litres of gasoline sold are found by setting counters to zero in the initialization routine, and thereafter at each completion of service incrementing the number of customers served by one and the number of litres sold by the amount of this sale. The number of customers who balk and the number litres lost because of balking are handled similarly, except that the updating takes place at customer arrivals. The total waiting time is the sum over all customers of the differences between his start of service and his arrival time, and may be recorded in one of two ways. First, the start-of-service routine could simply add the difference between the current value of simulated time and the arrival time for the auto to the total. Second, the arrival routine could *subtract* the arrival time for each auto from the total waiting time, and the start-of-service routine could *add* the starting time for each auto's service to the total. The total service time and total time that the waiting queue is empty are handled similarly. No matter how the measurements are recorded, it is important to make sure that the boundary conditions due to customers left in the system when end-of-simulation is reached are handled properly.

2.6.2. Outlines for the Event Routines

For each type of event, we require an event routine with the following basic structure:

i) Collect statistics and record measurements.

ii) Carry out the actions associated with the event (i.e., change the state of the system).

iii) Schedule future events if necessary.

There is much freedom in choosing where to collect a particular statistic or to schedule the next event of a particular type. However, it is *very important* to make sure that each desired measurement is recorded *somewhere*, and that each type of event is scheduled *somewhere*.

2.6.2.1. Arrival Routine

```
Generate the number of litres needed.
if customer balks when planning to buy L litres because of the queue size then
    Update the number of balking customers
    Add L litres to the total number of litres needed by balking customers
else
    Create a new auto record
    Record the arrival time and number of litres in the auto record
    if available pump list is not empty then
        Schedule start-of-service immediately
    else
    Link the auto record to the end of the waiting queue
    end if
end if
Generate the next interarrival time
Schedule the next arrival at the current time plus the next interarrival time
```

2.6.2.2. Start-of-Service-Routine

```
Match the auto to an available pump
Add auto's waiting time to total wait time
Generate service time based on the number of litres needed
Add service time to total service time
Schedule the departure of the auto
Link the pump record into the departure list
```

2.6.2.3. Departure Routine

```
Remove the first pump from the departure list
Add 1 to the number of autos served
Add the number of litres needed to the total number of litres
Free the auto record
Place the pump on the available pump list
if the waiting queue is not empty then
    Schedule start-of-service immediately
end if
```

2.7. Pseudo-Random Number Generation

2.7.1. Probability and Random Variables

Probability is used to express our confidence in the outcome of some random event as a real number between 0 and 1. An outcome that is impossible has probability 0; one that is inevitable has probability 1. Sometimes, the probability of an outcome is calculated by recording the outcome for a very large number of occurrences of that random event (the more the better), and then taking the ratio of the number of events in which the given outcome occurred to the total number of events. For example, we could try to estimate the probability that the outcome of tossing a coin is ‘‘heads’’ by tossing some coin 100 times, and then dividing the number of times that it came up ‘‘heads’’ by 100. Chances are that our estimate will be close to 0.5, the value that gamblers have ‘‘known’’ to be correct for centuries!

It is also possible to determine the probability of an event in a non-experimental way, by listing all possible non-overlapping outcomes for the experiment and then using some insight to assign a probability to each outcome. For example, we can show that the probability of getting ‘‘heads’’ twice during three coin tosses should be 3/8 from the following argument. First, we list all eight possible outcomes for the experiment (TTT, TTH, THT, THH, HTT, HTH, HHT, HHH). Next, we assign equal probability to each outcome (i.e., 1/8), because a ‘‘fair’’ coin should come up heads half the time, and the outcome of one coin toss should have no influence on another. (In general, when we have no other information to go on, each possible outcome for an experiment is assigned the same probability.) And finally, we observe that the desired event (namely two H's) includes three of these outcomes, so that its probability should be the sum of the probabilities for those outcomes, namely 3/8.

A **random variable** is a variable, say X, whose value, x, depends on the outcome of some random event. For example, we can define a random variable that takes on the value of 1 whenever ‘‘heads’’ occurs and 0 otherwise. Our estimate of the probability of ‘‘heads’’ can then be found as the average value of this random variable over our 100 tosses. A random variable can be characterized by the set of possible values that it can take on, together with the probability that it takes on each of those possible values, say x, which can be expressed as a **probability density function**, $f_X(x)$.

2.7.1.1. Discrete Random Variables

Random variables can be either discrete or continuous. A discrete random variable can take on only a finite or countable number of different values $v_1, v_2, \cdots, v_N$. In this case, $0 \le f_X(v_j) \le 1$, and we have

$$\sum_{j=1}^{N} f_X(v_j) = 1$$

The outcome of a coin toss, discussed above, provides a simple example of a discrete random variable. In fact, this is just a special case of the **discrete uniform** distribution, where each of the N possible values occurs with probability $1/N$. Another common example of a discrete uniform distribution is the outcome of rolling a die, where each of the values 1, 2, 3, 4, 5, and 6 occurs with probability $1/6$. The **binomial** distribution represents the number of times that a given outcome comes up during n experiments, assuming that the given outcome occurs with probability p at each experiment. The probability of exactly k occurrences, $0 \le k \le n$, is given by

$$f_X(k) = \binom{n}{k} p^k (1-p)^{n-k}$$

(The term $\binom{n}{k} \equiv n!/[k!(n-k)!]$ in the equation above represents the number of distinct ways that the k outcomes can come up during the n experiments.) Thus, another way to determine the probability of getting exactly two ''heads'' in three coin tosses would be to substitute $n=3$, $k=2$ and $p=1/2$ in the formula above to get $3/8$. The **Poisson** distribution represents the number of times that a random event (such as the arrival of an auto to our service station) occurs over a time interval of length t, assuming that the events occur at a uniform *rate* of r events per unit time. The probability of exactly k occurrences, $k \ge 0$, is given by

$$f_X(k) = \frac{(r\,t)^k}{k!} \exp\{-r\,t\}$$

So if autos arrive at the rate of 10 per hour, the probability of having 5 arrivals in a 30 minute period would be

$$\frac{(10\times 1/2)^5}{5!} \exp\{-10\times 1/2\} \approx 0.176$$

2.7.1.2. Continuous Random Variables

A continuous random variable can take on any value within some segment of the real line, say ranging from a to b. Since the probability of hitting a particular real number within this interval is essentially zero (because there are so many of them!), we must restrict ourselves to finding the probability of the event that the random variable falls within some interval (x_1, x_2), $a \le x_1 < x_2 \le b$. (The probability of this event can still be non-zero, because any interval contains an infinite number of real numbers.) For this reason, we define the probability density function so that its *integral* from x_1 to x_2 gives the probability of that event, namely

$$\int_{x_1}^{x_2} f_X(z)dz$$

Obviously, we must have $\int_a^b f_X(z)dz = 1$.

In the **continuous uniform** distribution over the interval (a, b), the probability that the random variable falls within a sub-interval of (a, b) is proportional to the length of that sub-interval, from which we can see that $f_X(x) = 1/(b-a)$, $a \le x \le b$. By far the most important example of a continuous uniform distribution is for the interval (0, 1), because, as we shall see below, it is always possible to *transform* a random variable that is uniformly distributed over (0, 1) into another random variable with any other distribution. For this reason, most programming languages (including Turing) include built-in **pseudo-random number generators** that produce a sequence of values that are uniformly distributed over (0, 1) and appear to be randomly generated.

Other important continuous distributions include the exponential distribution and the normal distribution. It can be shown that the **exponential** distribution represents the time between two consecutive events, when the events occur at a uniform rate. If that rate is r per unit time, the distribution is given by

$$f_X(t) = r \exp\{-r\,t\}$$

for all $t \ge 0$. Note the relation between the Poisson and exponential distributions: if random events are occurring at a uniform rate over time, then the times between consecutive events will be exponential and the number of events that fall within a given interval will be Poisson. The **normal** distribution represents the well-known ''bell-shaped'' curve given by

$$f_X(x) = \frac{1}{\sqrt{2\pi}\,\sigma} \exp\{-(x-\mu)^2/2\sigma^2\}$$

where μ represents the mean and σ the standard deviation of the distribution. The normal distribution is important in statistics because the sum of a large number of random variables drawn from any distribution tends to obey a normal distribution, no matter what is the distribution of the individual random variables. (The *serviceTime* procedure in Section 4 is based on this property.)

2.7.2. Generating Uniform Pseudo-Random Numbers

Since the result of executing any computer program is in general both predictable (if we look carefully at the code before execution) and repeatable (if we run the same program again), the idea that a computer can be used to generate a sequence of *random* numbers seems to be a contradiction. There is no contradiction, however, because the sequence of numbers that is actually generated by a computer is entirely predictable once the algorithm is known. Such algorithmically generated sequences are called **pseudo-random** sequences because they *appear* to be random in the sense that a good pseudo-random number sequence can pass most statistical tests designed to check that their distribution is the same as the intended distribution. On the other hand, to call them ''random'' numbers is no worse than it is to label floating point numbers as ''real'' numbers in a programming language.

It is worth noting that the availability of pseudo-random (rather than truly random) numbers is actually an *advantage* in the design of simulation experiments, because it means that our ''random'' numbers are *reproducible*. Thus, to compare two strategies for operating our service station, we can run two experiments with different numbers of pumps but with exactly the same sequence of customers. Were we to attempt the same comparison using an actual service station, we would have to try the two alternatives one after the other, with a different sequence of customers (e.g., those that came on Tuesday instead of those that came on Monday), making it difficult to claim that a change in our profits was due to the change in strategy rather than the differences in traffic patterns between Mondays and Tuesdays.

In general, most pseudo-random number generators produce uniform values. (This does not lead to a loss of generality because, as we shall see in the next section, uniform values can be transformed into other distributions.) Since only finite accuracy is possible on a real computer, we cannot generate continuous random variables. However, it is possible to generate pseudo-random integers, x_k, uniformly between 0 and some very large number (say m), which we use directly as a discrete uniform distribution, or transform into the fraction x_k/m to be used as an approximation to a uniform continuous distribution between 0 and 1.

The most popular method of generating uniform pseudo-random integers is as follows. First, an initial value x_0, the **seed**, is chosen. Thereafter, each number, x_k, is used to generate the next number in the sequence, x_{k+1}, from the relation

$$x_{k+1} := (a * x_k + c) \ mod \ m$$

The only difficult part is choosing the constants a, c and m. These must be chosen both to ensure that the *period* of the random number generator (i.e., the number of distinct values before the sequence repeats) is large, and also to ensure that consecutive numbers appear to be independent. Note that the period can be at most m, because there are only m distinct values between 0 and $m-1$. A more complete discussion is beyond the scope of this course. For more information, see *The Art of Computer Programming,* Volume 2, by D. E. Knuth, Addison-Wesley.

In most programming languages, one or more pre-defined subprograms are provided that return either an integer between 0 and m, or a ''real'' number between 0 and 1. If the seed is initialized to the same value when the program starts executing, then the sequence is repeatable. However, if the seed is set to a different value, a different sequence will be generated.

In Turing, the ''standard'' random number generator can be called using

$$rand\ (r)$$

to set r to a uniform value between 0 and 1 (limits not included), or

$$randint\ (k,\ a,\ b)$$

to set k to one of the integers a, $a+1, \ldots, b$ with equal probability. The seed is automatically initialized to the same value when the program starts executing. However, the seed can be changed at any time to a value that is both unpredictable (to the programmer) and non-repeatable, using

$$randomize$$

In addition, Turing has 10 more uniform (0, 1) random number generators available using

$$\textit{randnext}\ (r,\ j)$$

to set r to a new value taken from the jth random number generator, $1 \le j \le 10$. The seeds for each of these 10 random number generators may be independently reset, for example using

$$\textit{randseed}\ (s,\ j)$$

to reset the seed for the jth random number generator to the integer value s. Note that *randnext* allows us to use a *separate* random number stream to generate each of the random variables in our simulator (customer inter-arrival times, service times, etc.), so it is easier to assess the effects of changing strategies. For example, if separate random number streams are used, it is clear that by changing the number of pumps, we do not accidentally change the customer arrival times or service requirements. Otherwise, were we to use a single random number stream to generate *all* random variables in our simulation, we might end up using the kth random number to generate a service time in one case and an arrival time in the other.

2.7.3. Generating Pseudo-Random Numbers with Other Distributions

At this point, we must introduce the **cumulative distribution function**, $F_X(x)$, which represents the probability that the value of the random variable X is less than or equal to x. (Note the difference from the probability *density* function that we have been using up to now, which represents the probability that the value of X is *equal* to x.) Whenever X is less than x, it is also less than all values greater than x, so that $F_X(x)$ is non-decreasing.

For discrete random variables (where X can only take the values $v_1, \cdots, v_N$, and $v_1 < v_2 < \cdots < v_N$), we can find $F_X(x)$, assuming $v_k \le x < v_{k+1}$, as

$$F_X(x) = \sum_{j=1}^{k} f_X(v_j)$$

For continuous random variables, we have

$$F_X(x) = \int_{-\infty}^{x} f_X(z)dz$$

Figure 4 shows graphs of $F_X(x)$ as a function of x for both discrete and continuous random variables that are uniformly distributed between 1 and 4, inclusive.

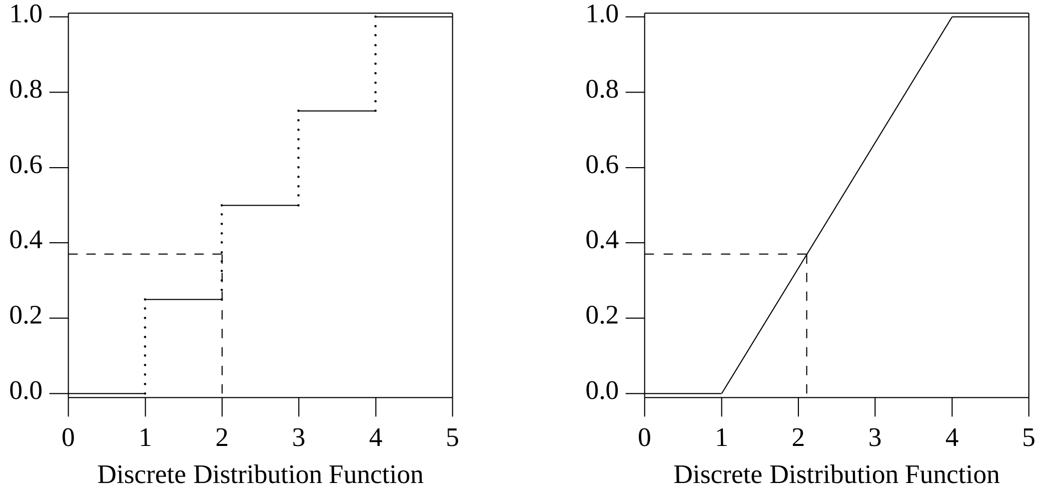

Figure 4. Probability Distribution Functions for Uniform [1, 4] Random Variables

We are now ready to explain how F_X can be ''inverted'' in a way that allows us to transform pseudo-random numbers that have a uniform distribution over the interval (0, 1), which we now know how to generate, into pseudo-random numbers with other distributions. First, we observe that no matter what value the random variable, X, takes on, the corresponding value of F_X is always between 0 and 1, which also ''happens'' to be the range of a continuous uniform (0, 1) random variable. Next, with reference to Figure 4 we choose any value, y, between 0 and

1, for example y=0.4. For any y, a horizontal line through y always intersects the $F_X(x)$ curve at a *unique* point, which for this example is x=2 in the discrete case and x=2.2 in the continuous case. (If F_X has a horizontal component at that point, for example at y=0.25, then we take the *left-most* intersection point.) >From the definition of the cumulative distribution function, it should be clear that at this intersection, we have determined the value of x such that the random variable X takes on values no greater than x a fraction y of the time. But it is also true that a continuous uniform (0, 1) random variable takes on values no greater than y a fraction y of the time. Thus, we can generate a sequence of values $x_1, x_2, \cdots$ for the random variable X having cumulative distribution function F_X by generating a sequence of values $y_1, y_2, \cdots$ having a continuous uniform (0, 1) distribution and then solving the equation $y_k = F_X(x_k)$ for x_k, i.e.,

$$x_k = F_X^{-1}(y_k)$$

where we use F_X^{-1} to indicate the inverse function to F_X. In general, it is not easy to invert F_X (for example, F_X for the normal distribution cannot be inverted), but in many cases we can write down an explicit expression for F_X^{-1}. In the examples below we will explain how to generate pseudo-random variables with several commonly used distributions in a programming language such as Turing.

For any **discrete** distribution, where X takes on the value v_j with probability p_j, $j=1, 2, \ldots, N$, we have in general that x_k can be found from y_k using a loop:

```
rand (y)
j := 1
loop
    exit when y <= p (j)
    y -= p (j)
    j += 1
end loop
x := v (j)
```

In the special case of a **discrete uniform** distribution over the interval $[a, b]$ (such as Figure 4a, for example), we can invert F_X to obtain

$$F_X^{-1}(y) = a + floor\,[\,y\,(b-a+1)\,]$$

In the case of the **Poisson** distribution, there is again no simple way to express F_X^{-1}, but since the value of X is potentially unbounded, it is more common to exploit the relation between the exponential and Poisson distributions to yield the following algorithm

```
x := 0
interval := t
loop
    exponential (y, n)        % y set to an exponential value at rate n, as described below
    exit when y > interval    % i.e., if next arrival occurs outside interval
    x += 1                    % this arrival is within the interval
    interval -= y             % so we can test for more arrivals in remaining interval
end loop
```

As for **continuous** distributions, we have that pseudo-random variables having a **continuous uniform** distribution over (a, b) can be found from

$$F_X^{-1}(y) = a + y \times (b-a)$$

To generate a value from the **exponential** distribution with rate r, we proceed as follows. First, we must integrate the probability density function that was given in Section 3.1 to find $F_X(x)$:

$$F_X(x) = \int_0^x r \exp\{-r\,t\}\,dt$$

for all $x>0$. Therefore, to invert F_X, we must solve $y = 1 - \exp\{-r\,x\}$ in terms of x, i.e., $x = -1/r \ln\{1-y\}$. But if y is uniformly distributed over (0, 1), then so is $1-y$, so we can replace $1-y$ by y to obtain:

$$x_k = (-1/r)\ln(y_k)$$

2.8. A Complete Example Simulation Program

We have now outlined the general approach to designing a stochastic simulation program, and illustrated each step by referring to an automobile service station problem. It is now time to show you what a complete simulation program for modelling this system might look like in Turing. Thus, the remainder of this section is a complete Turing program for modelling the service station problem. Notice the correspondence between the concepts mentioned above (entities, relationships, event routines, generation of stochastic variables, etc.) and the various components of the program. In the final section of this chapter, we will refer to the output generated by this program when we consider the problem of how to interpret the output of a stochastic simulation.

```
const profit := .025 % dollars per litre
const cost := 20 % the incremental cost in dollars to operate one pump for a day

var ReportInterval, endingTime : real
get ReportInterval, endingTime

var simulationTime : real := 0 % in seconds since the start of the simulation
var TotalArrivals, customersServed, balkingCustomers : int := 0
var TotalLitresSold, TotalLitresMissed, TotalWaitingTime, TotalServiceTime : real := 0

%----------------------------------------------------------------------------
%       Entities
%----------------------------------------------------------------------------

var car : collection of
   record
         arrivalTime : real
         litresNeeded : real
         next : pointer to car
   end record

var numPumps : int
get numPumps
put "This simulation run uses ", numPumps, " pumps " ..

var pump : array 1 .. numPumps of
   record
         completionTime : real
         next : int
         carInService : pointer to car
   end record

% Link all pumps into the available pumps list.

var availablePump : int := 1
for j : 1 .. numPumps – 1
   pump (j).next := j + 1
end for
pump (numPumps).next := 0

%----------------------------------------------------------------------------
%       Event list.
%----------------------------------------------------------------------------
```

```
% To avoid managing a separate list of busy pumps, sorted by departure times, we do the
% following "trick":  event types will be represented as integer constants, where values
% 1 .. numPumps represent a departure from the respective pump, 0 represents an arrival,
% -1 a progress report, and -2 the end of the simulation.  (We don't reserve a number for
% the start of service events, because all are triggered by service completions or arrivals)

const pervasive arrival := 0
const pervasive progressReport := – 1
const pervasive allDone := – 2

module event
   export (insert, getNext)

   var eventList : collection of
         record
            what : int
            whatTime : real
            next : pointer to eventList
         end record

   var firstEvent := nil (eventList)

   % Now create insert/delete utilities for manipulating the eventList

   procedure insert (whatType : int, time : real)
         var e : pointer to eventList
         new eventList, e
         eventList (e).what := whatType
         eventList (e).whatTime := time
         if firstEvent = nil (eventList) or time < eventList (firstEvent).whatTime then
            eventList (e).next := firstEvent
            firstEvent := e
         else
            var behind := firstEvent
            var ahead := eventList (firstEvent).next
            loop
                  exit when ahead = nil (eventList) or eventList (ahead).whatTime > time
                  behind := ahead
                  ahead := eventList (ahead).next
            end loop
            eventList (behind).next := e
            eventList (e).next := ahead
         end if
   end insert

   procedure getNext (var what : int, var whatTime : real)
         pre firstEvent not= nil (eventList)
         what := eventList (firstEvent).what
         whatTime := eventList (firstEvent).whatTime
         const restOfList := eventList (firstEvent).next
         free eventList, firstEvent
         firstEvent := restOfList
   end getNext

end event
```

```
%---------------------------------------------------------------------------
%       Queue for waiting cars
%---------------------------------------------------------------------------

module carQueue
   import (simulationTime, var car)
   export (insert, getNext, queueSize, emptyTime)

   var firstWaitingCar, lastWaitingCar : pointer to car := nil (car)
   var hiddenQueueSize : int := 0
   var totalEmptyQueueTime : real := 0

   function queueSize : int % This function is only necessary because of the Turing proof rules
         result hiddenQueueSize
   end queueSize

   function emptyTime : real
         if hiddenQueueSize > 0 then
            result totalEmptyQueueTime
         else
            result totalEmptyQueueTime + simulationTime
         end if
   end emptyTime

   procedure insert (newestCar : pointer to car)
         pre car (newestCar).next = nil (car)
         if lastWaitingCar = nil (car) then
            assert hiddenQueueSize = 0
            firstWaitingCar := newestCar
            totalEmptyQueueTime += simulationTime
         else
            assert hiddenQueueSize > 0
            car (lastWaitingCar).next := newestCar
         end if
         lastWaitingCar := newestCar
         hiddenQueueSize += 1
   end insert

   procedure getNext (var nextCar : pointer to car)
         pre hiddenQueueSize > 0 and firstWaitingCar not= nil (car)
         hiddenQueueSize –= 1
         nextCar := firstWaitingCar
         firstWaitingCar := car (firstWaitingCar).next
         if firstWaitingCar = nil (car) then
            % empty queue: update the pointer to the end of queue too!
            assert lastWaitingCar = nextCar
            lastWaitingCar := nil (car)
            totalEmptyQueueTime –= simulationTime
         end if
         car (nextCar).next := nil (car) % so we can’t have a dangling pointer
   end getNext

end carQueue
```

```
%---------------------------------------------------------------------------
%       Generation of Stochastic Variables
%---------------------------------------------------------------------------

module random
   export (interarrivalTime, numLitres, DoesCarBalk, serviceTime)

   % Since Turing provides us with 10 separate pseudo-random number streams, and
   % our program contains four stochastic variables, will reserve a different
   % stream for each stochastic variable as follows:
   const arrivalStream := 1 % auto arrival times
   const litreStream := 2 % number of litres needed
   const balkingStream := 3 % balking probability
   const serviceStream := 4 % service times

   put "and the following random number seeds:"
   var nextSeed : int
   for j : 1 .. 4
          get nextSeed
          put nextSeed : 10 ..
          randseed (nextSeed, j)
   end for
   put ""

   procedure interarrivalTime (var thisArrivalTime : real)

          % The interarrival time of the next car is exponentially distributed

          const MEAN := 50 % seconds
          var r : real
          randnext (r, arrivalStream)
          thisArrivalTime := – MEAN ∗ ln (r)
   end interarrivalTime

   procedure numLitres (var n : real)

          % The number of litres required by a car is uniform between 10 and 60

          randnext (n, litreStream)
          n := 10 + n ∗ 50
   end numLitres

   procedure DoesCarBalk (litres : real, queueLength : int, var yes : boolean)

          % The probability that a car leaves without buying gas (i.e., balks)
          % grows larger as the queue length gets larger, and grows smaller
          % when the car requires a greater number of litres of gas, such that:
          % (1) there is no balking if the queueLength is zero, and
          % (2) otherwise, the probability of *not* balking is
          %        (40 + litres) / (25 * (3 + queueLength))

          var r : real
          randnext (r, balkingStream)
          yes := queueLength > 0 and r > (40 + litres) / (25 ∗ (3 + queueLength))
   end DoesCarBalk
   procedure serviceTime (litres : real, var howLong : real)

          % Service times will have a near Normal distribution, where the mean is 150
          % seconds plus 1/2 second per litre, and the standard deviation is 30 seconds
          % (It can be shown that the sum of 12 random numbers from the
          % uniform distribution on (0,1) has a near Normal distribution
          % with mean 6 and standard deviation 1.)

          var r : real
          howLong := – 6
          for i : 1 .. 12
             randnext (r, serviceStream)
             howLong += r
          end for
          howLong := 150 + 0.5 * litres + 30 * howLong
   end serviceTime
end random

%----------------------------------------------------------------------------
%       Event Routines
%----------------------------------------------------------------------------

procedure startService (arrivingCar : pointer to car)
   pre availablePump > 0

   TotalWaitingTime += simulationTime – car (arrivingCar).arrivalTime
   var pumpTime : real
   random.serviceTime (car (arrivingCar).litresNeeded, pumpTime)
   TotalServiceTime += pumpTime

   % Match the auto to first available pump
   pump (availablePump).carInService := arrivingCar
   pump (availablePump).completionTime := simulationTime + pumpTime
   % Schedule departure of car from this pump
   event.insert (availablePump, pump (availablePump).completionTime)
   availablePump := pump (availablePump).next
end startService

procedure carArrives
   TotalArrivals += 1
   var L : real
   random.numLitres (L)
   var carLeaves : boolean
   random.DoesCarBalk (L, carQueue.queueSize, carLeaves)
   if carLeaves then
          balkingCustomers += 1
          TotalLitresMissed += L
   else
          % Create and initialize a new auto record

          var arrivingCar : pointer to car
          new car, arrivingCar
          car (arrivingCar).arrivalTime := simulationTime
          car (arrivingCar).litresNeeded := L
```

```
            car (arrivingCar).next := nil (car)

            if availablePump > 0 then
                startService (arrivingCar)
            else
                carQueue.insert (arrivingCar)
            end if
    end if
    var timeToNextCar : real
    random.interarrivalTime (timeToNextCar)
    event.insert (arrival, timeToNextCar + simulationTime)
end carArrives

procedure departure (whichPumpFinished : int)
    pre pump (whichPumpFinished).carInService not= nil (car)
    var departingCar := pump (whichPumpFinished).carInService
    pump (whichPumpFinished).next := availablePump
    availablePump := whichPumpFinished
    customersServed += 1
    TotalLitresSold += car (departingCar).litresNeeded
    free car, departingCar
    if carQueue.queueSize > 0 then
            carQueue.getNext (departingCar)
            startService (departingCar)
    end if
end departure

procedure snapshot
    put simulationTime : 8, TotalArrivals : 7, carQueue.emptyTime / simulationTime : 8 : 3 ..
    if TotalArrivals > 0 then
            put simulationTime / TotalArrivals : 9 : 3,
                (TotalLitresSold + TotalLitresMissed) / TotalArrivals : 8 : 3 ..
    else
            put "Unknown" : 9, "Unknown" : 8 ..
    end if
    put balkingCustomers : 8,
            TotalWaitingTime / max (1, customersServed) : 9 : 3,
            TotalServiceTime / (numPumps * simulationTime) : 7 : 3,
            TotalLitresSold * profit – cost * numPumps : 8 : 2,
            TotalLitresMissed * profit : 8 : 2
    event.insert (progressReport, simulationTime + ReportInterval)
end snapshot

%----------------------------------------------------------------------------
%       Initialization
%----------------------------------------------------------------------------

% Schedule the end-of-simulation and the first progress report.

event.insert (allDone, endingTime)
if ReportInterval <= endingTime then
    event.insert (progressReport, ReportInterval)
end if
```

```
% Schedule the first car to arrive at the start of the simulation

event.insert (arrival, 0)

% Print column headings for periodic progress reports and final report

put " Current" : 10, "Total" : 7, "NoQueue" : 8, "Car–>Car" : 9, "Average" : 8, "Number" : 8,
   "Average" : 9, "Pump " : 7, "Total " : 8, " Lost " : 6
put "   Time " : 10, " Cars" : 7, "Fraction" : 8, "  Time  " : 9, " Litres " : 8, "Balked" : 8,
   "  Wait " : 9, "Usage" : 7, "Profit" : 8, "Profit" : 6
put repeat ("–", 80)

%---------------------------------------------------------------------------
%       Main program
%---------------------------------------------------------------------------

var currentEvent : int
loop
   event.getNext (currentEvent, simulationTime)
   exit when currentEvent = allDone
   if currentEvent = progressReport then
         snapshot
   elsif currentEvent = arrival then
         carArrives
   elsif currentEvent > 0 and currentEvent <= numPumps then
         departure (currentEvent)
   else
         put "Error: unknown event code encountered"
   end if
end loop
snapshot
```

2.9. Output Analysis in Stochastic Simulation

Once the simulation *program* has been completely written and carefully tested, the actual simulation *experiment* must be carried out. The idea of the experiment is to conduct one or more simulation ‘‘runs’’ from which measurements are collected. The output from these runs is used to assess the performance of the system (or systems, if alternative strategies are being evaluated). For example, one run from our service station simulator might have produced the following output:

```
This simulation run uses three pumps and the following random number seeds:
        1         2         3         4
Current  Total   NoQueue   Car—>Car   Average   Number   Average   Pump    Total    Lost
  Time    Cars   Fraction    Time      Litres   Balked    Wait    Usage   Profit   Profit
-----------------------------------------------------------------------------------------
  20000    382    0.529     52.356    34.342      67     54.877   0.875   221.34   46.62
  40000    771    0.470     51.881    34.454     136     59.949   0.887   505.85   98.25
  60000   1157    0.475     51.858    34.767     206     61.015   0.888   794.76  150.87
  80000   1554    0.476     51.480    35.052     290     60.764   0.884  1088.26  213.51
 100000   1967    0.475     50.839    34.974     379     60.524   0.884  1376.51  283.35
 120000   2351    0.477     51.042    34.957     451     60.753   0.882  1652.25  342.38
 140000   2760    0.473     50.725    34.824     539     61.889   0.883  1934.74  408.09
 160000   3175    0.468     50.394    34.682     625     62.332   0.886  2225.05  467.86
 180000   3574    0.464     50.364    34.753     696     62.592   0.888  2524.16  521.05
 200000   3957    0.468     50.543    34.746     769     61.654   0.887  2807.68  569.61
```

2.9.1. Initial and Final Conditions

When calculating performance figures from a simulation run, care must be taken to ensure that the results obtained are not systematically distorted because of either the initial state of the system at the beginning of each simulation run or the final state of the system when end-of-simulation is encountered.

For example, suppose we were concerned with the time for an auto to fill up its tank during a busy time of day, such as afternoon rush hour before a holiday. If we ran the simulation for a long time, we might find that under those operating conditions an average of 15 autos are in the station having their tanks filled or waiting for a free pump. But if the initial state for the simulation did not have *any* autos in the service station, then it would be obvious that the first few autos spent less time waiting than a ‘‘typical’’ auto would. Thus, our estimate for the mean system time would turn out too low unless we

i) ran our simulation for such a long time that the measurements from these first few customers had very little influence on the mean,

ii) began our measurements part way through the simulation run, by which point the system should have reached a state that is more ‘‘normal’’, or

iii) used the average values from one simulation run as the initial conditions for the next simulation run so that, hopefully, the system is in a normal state throughout the entire run.

In the output from our service station simulator, the effect of the initial conditions (i.e., a completely idle station) are quite evident. Notice that even though the first progress report was not printed until almost 400 cars had passed through our service station, we *still* see a downward trend in the fraction of time that the queue of waiting cars is empty, and an upward trend in both the average waiting time and the utilization of the pumps.

The end conditions of the simulation must also be checked before we can trust our calculated performance statistics. For example, the waiting times and service times for customers in the system when end-of-simulation is reached may or may not be counted, depending on the method used for recording these measurements. If such customers are not counted, then we risk underestimating the utilization of the server, because some customers who might have been part way through their service time are ignored. If such customers are counted, then we risk underestimating the mean waiting time and overestimating the mean service rate because, in effect, we may be claiming that these customers were served during our simulation run when in fact their service would not have been completed until later.

In the case of our service station simulator, notice that we are adding the service times for each car to the total service performed in the *startService* procedure. Thus, we are always overestimating the pump utilization: in effect, we are (optimistically) claiming that the *entire* service times for all cars not still in the waiting queue have been completed by the end of simulation — even those cars still in service. If the run length is short enough, this choice of stopping condition can lead to ‘‘impossibly good’’ results. For example, notice that in the following data (representing a short run with a heavily loaded service station) the pumps appear to be busy *more than 100 percent* of the time at the first few progress reports!

```
This simulation run uses two pumps and the following random number seeds:
        1        2        3        4
Current  Total  NoQueue  Car->Car  Average  Number  Average   Pump   Total    Lost
  Time    Cars  Fraction   Time    Litres   Balked   Wait    Usage  Profit   Profit
------------------------------------------------------------------------------------
   2000    39    0.072    51.282   31.508     11   182.897  1.002   -18.43     9.15
   4000    82    0.039    48.780   31.037     28   184.429  1.004     2.70    20.93
   6000   124    0.059    48.387   33.032     46   190.620  1.014    26.29    36.10
   8000   162    0.068    49.383   34.455     62   190.959  1.005    50.97    48.57
  10000   207    0.069    48.309   33.815     81   205.277  1.002    71.16    63.84
  12000   240    0.067    50.000   34.090     92   204.176  1.005    92.20    72.33
  14000   284    0.058    49.296   34.094    111   215.161  1.005   115.38    86.68
  16000   304    0.088    52.632   34.567    113   210.377  0.993   134.53    88.18
  18000   343    0.094    52.478   34.308    127   210.086  0.996   155.13    99.06
  20000   382    0.084    52.356   34.227    140   211.138  0.997   178.59   108.28
```

Of course, we could easily replace the overestimate with an underestimate by moving the accumulation of service time to the *departure* routine. However, calculating the correct value for pump utilization is much trickier.

Another possibility that must be considered in assessing the end conditions of the simulation is whether or not the system being simulated is overloaded. Since only a finite period of simulated time elapses during each simulation run, the average waiting time will *always* be finite, even if the system is in fact so overloaded that the customer waiting times are growing without bound as simulated time advances. Thus, our performance statistics are unreliable if a significant proportion of the customers are still in the system at the end of simulation: either the simulation run was too short (and the effects of the initial and final conditions are large), or the system was overloaded.

2.9.2. Confidence Intervals on the Mean of a Random Variable

Once we have carried out one or more simulation runs, and convinced ourselves that the boundary conditions due to initial and final conditions have not spoiled our results, we are left with the task of trying to conclude something about the values of various random variables in the model, such as the mean waiting time for an auto in our service station, for example. It is simple to calculate the **sample mean**

$$\overline{X} \equiv \frac{1}{N} \sum_{j=1}^{N} x_j,$$

i.e., the average of the N measurements that we made of the value of the random variable X. Similarly, we can calculate the **sample variance**

$$S_X^2 \equiv \frac{1}{N-1} \sum_{j=1}^{N} \left(x_j - \overline{X} \right)^2$$

by averaging the squared differences between each measurement and the sample mean. But as we scan down the columns of measurements in the output, it is obvious that these sample quantities *vary* as a function of (simulated) time. Furthermore, if we repeat the experiment using different random number seeds, we soon see that the measurements also vary from one run to the next. Consequently, we should be skeptical of concluding that any of these *sample* means or variances are equal to the *true* mean and variance of X that we set out to find in the first place. And indeed to proceed further, we need to make some assumptions about the distribution of X.

As a first step, suppose a ‘‘helpful genie’’ were to tell us that the random variable X has a **Normal distribution** (i.e., a ‘‘bell-shaped’’ probability density function). In this case, knowledge of the Normal distribution would allow us to construct **confidence intervals** about the sample mean, say $\overline{X} \pm C$, that contain the true mean, say M, a specified fraction of the time (the greater the fraction, the wider the resulting confidence interval about $\overline{X}$). This construction is based on the following two facts about the Normal distribution with mean M and variance σ^2. First, the proportion of measurements that fall within a distance of $K\sigma$ from M is known. For example, if we make a large number of measurements $x_1, x_2, \ldots,$ then approximately 95 percent of them will fall within the interval $M \pm 2\sigma$. The key observation for us is that, since the distance from M to a measurement x_j is the same as the distance from x_j to M, we can turn this result around to say that 95 percent of the time M falls within the interval $x_j \pm 2\sigma$. A table of results in this form is shown below in Figure 5. Second, the *average* of N measurements taken from a Normal distribution with mean M and variance σ^2 also has a Normal distribution, but with mean M and variance σ^2/N. Thus, if the confidence interval ends up too wide, it can be tightened by averaging in additional measurements. For example, using four times as many measurements cuts the width of the confidence interval in half.

Interval	Fraction of time that the interval includes the true mean
$(x-\sigma,\ x+\sigma)$	0.67
$(x-2\sigma,\ x+2\sigma)$	0.95
$(x-2.6\sigma,\ x+2.6\sigma)$	0.99

Figure 5. Intervals about a measurement x that will contain the true mean a given fraction of the time, when the random variable has a Normal distribution with mean M and variance σ^2

The problem with the result above is that even if we knew that the random variable had a Normal distribution, we still couldn't write down a confidence interval for M without knowing σ^2 (which depends on M). To get out of this circular chain of reasoning, we need a method of estimating σ^2 that is based on $\bar{X}$ instead of M. Clearly the sample variance, defined above, is not a very good estimate, especially when the number of measurements is small. Indeed, if only *one* measurement is available, then the sample variance is always zero, indicating that we have not *observed* any variability in our measurements. But $\sigma^2 = 0$ means something quite different, namely that X is not a *random* variable at all! The solution is to use an ''unbiased'' estimate of the sample variance to represent σ^2, i.e.,

$$\sigma^2 \approx \frac{1}{N-1} \sum_{j=1}^{N} \left\{ x_j - \bar{X} \right\}^2,$$

and then simply go ahead and use the results described above in Table 1. (By dividing the sum by $N-1$ instead of N, it is clear that we cannot estimate σ^2 from a single measurement.)

A more serious problem with the method outlined above is that the kinds of random variables that usually come up in simulation models, such as the mean time in system for autos at our service station, almost *never* have a Normal distribution! Fortunately, the **Central Limit Theorem** in statistics tells us that under fairly general circumstances, the *average* of a large number of measurements *does* have a Normal distribution even if the individual measurements themselves do not have a Normal distribution. It follows that we can still find a confidence interval if we carry out K independent repetitions of the same simulation experiment and treat the K sample means $\bar{X}_j$, $j=1, \ldots, K$, as the measurements.

For example, suppose we ran four more runs of the service station simulation with 3 pumps, giving us, in total, five independent measurements of the mean waiting time. The last line of output from each of these runs is reproduced in the following table:

```
Current   Total   NoQueue   Car->Car   Average   Number   Average   Pump    Total    Lost
  Time     Cars   Fraction    Time     Litres    Balked    Wait     Usage   Profit   Profit
------------------------------------------------------------------------------------------
 200000    3957    0.468     50.543    34.746     769     61.654    0.887  2807.68  569.61
 200000    4096    0.397     48.828    35.081     881     76.382    0.903  2870.20  662.09
 200000    4062    0.432     49.237    34.850     848     66.428    0.899  2826.69  652.29
 200000    3961    0.440     50.492    35.108     787     69.013    0.894  2821.66  594.93
 200000    3972    0.429     50.352    34.886     790     66.098    0.890  2799.53  604.62
```

Because each of these five numbers is the average of a large number of individual measurements, we can use the Central Limit Theorem to conclude that they can be treated as measurements from a Normal distribution with mean

$$\bar{X} = \frac{61.654+76.382+66.428+69.013+66.098}{5} = 67.915$$

with variance

$$\sigma^2 \approx \frac{1}{4}\left\{(61.654-\bar{X})^2+(76.382-\bar{X})^2+(66.428-\bar{X})^2+(69.013-\bar{X})^2+(66.098-\bar{X})^2\right\}$$

Next, we note that 67.915 is (almost) the average of five measurements from the Normal distribution with variance

σ^2. Thus, we can use the second fact about the Normal distribution mentioned above to conclude that the average of five measurements, such as 67.915, is itself Normally distributed with the same mean, *M*, but with the variance reduced to $\sigma^2/5$. Finally, we can use Table 1 to construct a confidence interval about 67.915 that includes the true mean waiting time 95 percent of the time. Namely, $\bar{X} \pm 2\sigma$, i.e., the interval (63.07, 72.77), contains the true mean, *M*, about 95% of the time.