# II. 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:

```
1     const rate := .10 / 12  % interest rate per month, expressed as a fraction
2     const payment := 10     % payment per month
3     var loan : real := 100  % current value of the loan (initially $100)
4     var month :=0           % current month
5
6     put "end of month:" : 15, "loan balance:" : 15, "payment due:" : 10
7     put month : 10, loan : 15 : 2
8
9     loop
10        month += 1
11        loan *= (1 + rate)
12        const thisPayment := min (payment, loan)
13        loan -= thisPayment
14        put month : 10, loan : 15 : 2, thisPayment : 15 : 2
15        exit when loan <= 0.0
16    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 $\pi$ 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.

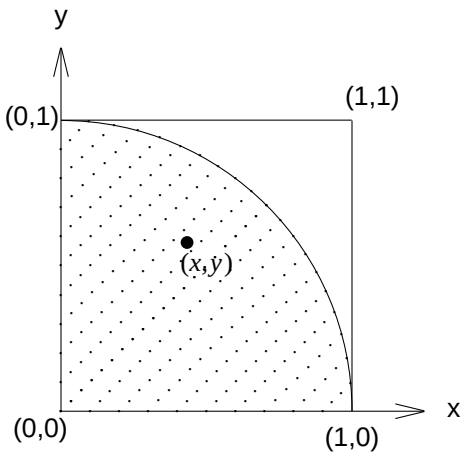


Figure 1. Monte Carlo Integration of a Quarter Circle.

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

The simple 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 previous chapter, we discussed a faster way through numerical integration.

```
1    randseed(1, 1)% could have used interactive input to initialize, but this is simpler
2    randseed(2, 2)
3
4    var x, y: real
5    var repeatCount, inCount := 0
6
7    put "repetition count: " ..
8    get repeatCount
9
10   for i: 1 .. repeatCount
11      randnext(x, 1)
12      randnext(y, 2)
13      if x*x + y*y < 1 then
14         inCount += 1
15      end if
16   end for
17
18   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 integrals 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 C or 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 above 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 program to find out the possible numbers of jumps.

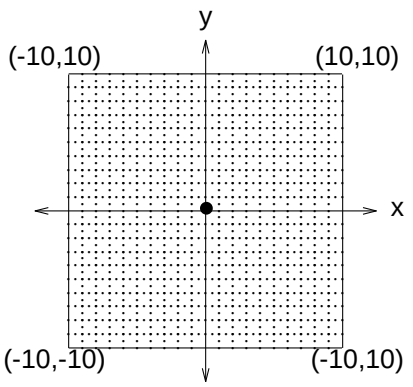


Figure 3. Drunken Flea on a Plate.

```
1     var x, y, step, direction: int := 0
2
3     randomize
4     loop
5        exit when abs(x) > 10 or abs(y) > 10
6        randint(direction, 1, 4)
7        case direction of
8              label 1: x += 1
9              label 2: y += 1
10             label 3: x -= 1
11             label 4: y -= 1
12       end case
13       step += 1
14    end loop
15    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 simply 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** or C **struct** is well-suited for representing an entity and its associated attributes. A group of similar entities may be represented as an **array** of records, or connected together in a linked list or other pointer-based data structure. 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 linked list 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 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 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 a pseudo-random number generator such as C’s *rand* or Turing’s *rand*, *randint* and *randnext*, and then transforming it as necessary to match the specified probability distribution for the stochastic variable. For example,

```
n = 10.0 + 50.0 * ((double) rand())/RAND_MAX;
```

could be used to set *n* to the number of litres needed by the next incoming auto, assuming that any 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 of 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 interval between each arrival time and the corresponding start of service, 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

$$\frac{(10\times 0.5)^5}{5!} \exp(-10\times 0.5) \approx 0.176$$

where the factor 0.5 is the time interval: 30 minutes = 0.5 hour.

### 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 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 C) include built-in **pseudo-random number generators** that produce a sequence of values that are uniformly distributed 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 $\mu$ represents the mean and $\sigma$ 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* function in Section 2.8 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 ''random'' 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

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

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 C, the standard random number generator can be called using

```
int r;
r = rand();
```

to set $r$ to one of the integers $0, 1, \ldots,$ `RAND_MAX` with equal probability. The seed is automatically initialized, but can be set to a particular value $s$ with

```
unsigned int s = ...;
srand (s);
```

Unlike Turing, C does not have standard functions that allow the use of distinct ''streams'' of random numbers.

The effects of changing strategies can be assessed more easily if we can use a separate random number stream to generate each of the random variables in our simulator (customer inter-arrival times, service times, etc.). For example, if we use separate streams of random numbers, it will be 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 *k*th 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.

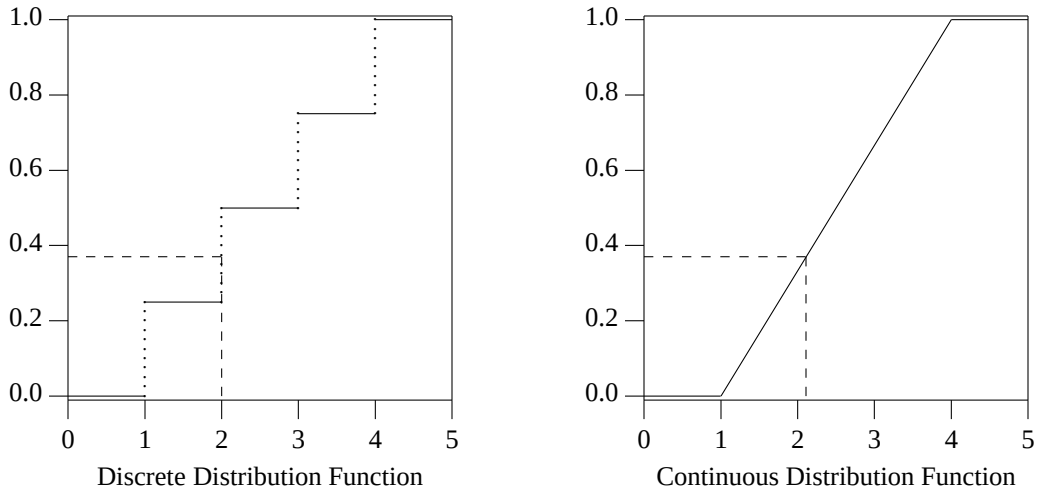


Figure 4. Probability Distribution Functions for Uniform [1, 4] Random Variables

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 is also 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 C.

First, we look at **continuous** distributions. A **continuous uniform** distribution over $(a, b)$ is given by

$$F_X(x) = (x - a)/(b - a)$$

Consequently, if $y$ comes from the uniform distribution over (0, 1), then we can get a pseudo-random variable with a continuous uniform distribution over $(a, b)$ from

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

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 2.7.1.2 to find $F_X(x)$:

$$F_X(x) = \int_0^x r \exp(-r\,t)\,dt = 1 - e^{-rx}$$

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:

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

Turning now to **discrete** distributions, we again have to obtain the equation $x = F^{-1}(y)$ by solving the equation $y = F(x)$. However, $F(x)$ has discontinuous ''steps,'' so instead of getting a formula as a solution, we use a simple algorithm:

1. Obtain a random $y$ from the continuous uniform distribution over (0, 1), just as we did for continuous distributions.
2. Count the steps in $F(x)$ that are needed to reach the value of our random $y$.

If $X$ takes on the value $v_j$ with probability $p_j$, $j = 1, 2, \ldots, N$, then the steps in $F(x)$ are of height $p_j$, so this loop calculates $x$ from $y$:

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

In Figure 4a, with $y = 0.2$ we will get $x = 2$.

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

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

This equation is quite similar to the one given above for the continuous uniform case.

In the case of the **Poisson** distribution, there is again no simple way to express $F_X^{-1}$. The value of $X$ is potentially unbounded, because there can be infinitely many events in the specified interval. Instead of calculating each $p_j$ directly, we exploit the relation between the exponential and Poisson distributions to yield the following algorithm, which calculates the time between successive events until the total time passed would exceed the specified interval.

```
x := 0
```

```
interval := t
loop
    exponential (y, n)          % y set to an exponential value at rate n, as described above
    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
```

### 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. Thus, the remainder of this section is a complete C++ 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.

```
// CSC 270 simulation example
// Adapted March 1996 by J. Clarke from Turing original by M. Molle

// This version edited slightly for the fall 1997 course notes.
// What system are we on?
//
// The system matters only because random-number generating functions
// tend to be system-dependent. To set the system, comment out the
// irrelevant choices and make sure the one you're using is not
// commented out. (A sensible alternative would be to comment out
// all the system #defines and use a compiler flag instead.)
//
// If your system is not mentioned here, look up its random number
// generator and add it to the list. The GNU random number generators
// are certainly preferable to the TURBO (Turbo C++) ones.
#define GNU
//#define TURBO

#include <iostream.h>
#include <iomanip.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define BOOLEAN int
#define TRUE 1
#define FALSE 0
//--------------------------------------------------------------------
// The time, a "keep going" control, and some constants of the model.
//
// There are several other global variables. Should we use them?
// Could we get rid of them? Where would they appear if we broke the
// program up into several separately-compiled modules? We certainly
// ought to do that!
//--------------------------------------------------------------------
```

```
double simulationTime; // A major global variable: the current time.
BOOLEAN simulating; // Are we still going?

const double profit = .025; // dollars per litre
const double cost = 20;
// the incremental cost in dollars to operate one pump for a day

//------------------------------------------------------------------------
// Generating random numbers.
//------------------------------------------------------------------------

// Prototypes for system-provided random number functions.
#if defined(GNU)
double erand48(unsigned short xsubi[3]);
#elif defined(TURBO)
int rand ();
#endif

class randStream {
// Manage one stream of random numbers.
//
// "nextrand" gives the floating-point number obtained from a
// new value in this stream. "xsubi" stores that latest value.
private:
    unsigned short xsubi[3];

public:
    randStream (unsigned short seed = 0);
    double nextrand ();
};

randStream * arrivalStream; // auto arrival times
randStream * litreStream; // number of litres needed
randStream * balkingStream; // balking probability
randStream * serviceStream; // service times

double randStream::nextrand ()
// Returns the next random number in this stream.
{
#if defined(GNU)
    return erand48 (xsubi);
#elif defined(TURBO)
    return ((double) rand())/((double) RAND_MAX);
#endif
}

randStream::randStream (unsigned short seed)
// Initialize the stream of random numbers.
{
#if defined(GNU)
    xsubi[0] = 0;
    xsubi[1] = 0;
    xsubi[2] = seed;
#elif defined(TURBO)
```

```
    // Turbo C doesn't provide a random-number generator with
    // explicit seeds. We will get along with just one stream.
    return;
#endif
}

//--------------------------------------------------------------------
// Events
//
// Remember that events are not entities in the same sense as cars and
// pumps are, and the event queue does not have the same reality as the
// car queue. The event queue is a data structure without a real-world
// equivalent, while the car queue is real and you can see it. Events are
// not quite so imaginary, but they are certainly less "real" than cars.
//--------------------------------------------------------------------

//--------------------------------------------------------------------
// Events--the general class on which the specific event types are based
//--------------------------------------------------------------------

class eventClass {
private:
    double localTime;

public:
    eventClass (double T = 0.0) {localTime = T;};
    inline double whatTime () {return localTime;}; // when it happens
    inline void setTime (const double newTime = 0) {localTime = newTime;};
    virtual void makeItHappen () {}; // the event routine
};

//-------------------------------------------------
// The event list and utilities for manipulating it.
//-------------------------------------------------

class eventListClass {
private:

    struct eventListItem {
        eventClass * data;
        eventListItem * next;
    };
    eventListItem * firstEvent;

public:
    eventListClass () {firstEvent = NULL;};
    void insert (eventClass * event);
    eventClass * getNext ();
};

eventListClass * eventList;

void eventListClass::insert (eventClass * event)
{
```

```
    eventListItem * e = new eventListItem;
    e -> data = event;

    const double T = event -> whatTime();
    if (firstEvent == NULL || T < firstEvent -> data -> whatTime()) {
        e -> next = firstEvent;
        firstEvent = e;
    }
    else {
        eventListItem * behind = firstEvent;
        eventListItem * ahead = firstEvent -> next;
        while (ahead != NULL && ahead -> data -> whatTime() <= T) {
            behind = ahead;
            ahead = ahead -> next;
        }
        behind -> next = e;
        e -> next = ahead;
    }
}

eventClass * eventListClass::getNext ()
{
    // pre firstEvent not= NULL

    if (firstEvent == NULL) {
        cout << "Error! ran out of events0;
        return NULL;
    }

    eventClass * eventToReturn = firstEvent -> data;

    eventListItem * restOfList = firstEvent -> next;
    delete firstEvent;
    firstEvent = restOfList;

    return eventToReturn;
}

//---------------------------------------------------------------------------
// Entities: cars and pumps
//---------------------------------------------------------------------------

class carClass {
private:
    double localArrivalTime;
    double localLitresNeeded;

public:
    carClass();
    void setTime (double T) {localArrivalTime = T;};
    double arrivalTime () {return localArrivalTime;};
    double litresNeeded () {return localLitresNeeded;};
};
```

```
carClass::carClass ()
// The number of litres required is a property of a car, so it
// belongs in this class. It is also something the car "knows"
// when it arrives, so it should be calculated in the constructor.
//
// The distribution of litres required is uniform between 10 and 60.
{
    localLitresNeeded = 10 + litreStream -> nextrand() * 50;
};

class pumpClass {
private:
    carClass * localCarInService;
    double serviceTime ();

public:
    carClass * carInService () {return localCarInService;};
    void startService (carClass *);
};

double pumpClass::serviceTime()
// Determine how long the service will take. This is a property of the
// pump-car combination.
//
// 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.)
{
    if (carInService() == NULL) {
        cout << "Error! no car in service when expected0;
        return -1.0;
    }

    double howLong = -6;

    for (int i = 1; i <= 12; i++)
        howLong += serviceStream -> nextrand();

    return 150 + 0.5 * localCarInService -> litresNeeded() + 30 * howLong;
}

//----------------------------------------
// The actual pumps at the service station
//----------------------------------------

class pumpStandClass {
private:
    typedef pumpClass * pumpListItem;
    pumpListItem * pumps; // A stack of pumps
    int numPumps;
    int topPump;
```

```
public:
    pumpStandClass (int);
    int howManyPumps () {return numPumps;}; // needed for statistics
    BOOLEAN existsAvailablePump () {return topPump >= 0;};
    pumpClass * getAvailablePump ();
    void releasePump (pumpClass *&);
};

pumpStandClass * pumpStand;

pumpStandClass::pumpStandClass (int N)
// Create set of available pumps and put all N pumps into it.
{
    if (N < 1) {
        cout << "Error! pump stand needs more than 0 pumps0;
        return;
    }

    numPumps = N;
    pumps = new pumpListItem [N];
    topPump = N - 1;

    for (int j = 0; j < N; j++)
        pumps[j] = new pumpClass;
}

pumpClass * pumpStandClass::getAvailablePump ()
// Take a pump from the set of free pumps, and return a pointer to it.
{
    if (topPump < 0) {
        cout << "Error! no pump available when needed0;
        return NULL;
    }

    return pumps[topPump--];
}

void pumpStandClass::releasePump (pumpClass *& P)
// Put pump P back on the available list.
// P is set to NULL, just to cause trouble if the caller tries
// to reuse this pump, which is now officially free.
{
    if (topPump >= numPumps) {
        cout << "Error! attempt to release a free pump0;
        return;
    }

    pumps[++topPump] = P;
    P = NULL;
}

//---------------------------------------------------------
// Queue for waiting cars and utilities for manipulating it
//---------------------------------------------------------
```

```
class carQueueClass {
private:
    struct queueItem {
        carClass * data;
        queueItem * next;
    };

    queueItem * firstWaitingCar, * lastWaitingCar;
    int localQueueSize;
    double totalEmptyQueueTime;

public:
    carQueueClass ();
    int queueSize ();
    double emptyTime ();
    void insert (carClass * newestCar);
    carClass * getNext ();
};

carQueueClass * carQueue;

carQueueClass::carQueueClass ()
{
    firstWaitingCar = NULL;
    lastWaitingCar = NULL;
    localQueueSize = 0;
    totalEmptyQueueTime = 0.0;
}

int carQueueClass::queueSize ()
{
    return localQueueSize;
}

double carQueueClass::emptyTime ()
{
    if (localQueueSize > 0)
        return totalEmptyQueueTime;
    else
        return totalEmptyQueueTime + simulationTime;
}

void carQueueClass::insert (carClass * newestCar)
{
    queueItem * newQueueItem;
    newQueueItem = new queueItem;
    newQueueItem -> data = newestCar;
    newQueueItem -> next = NULL;

    if (lastWaitingCar == NULL) {
        // assert queueSize = 0
        firstWaitingCar = newQueueItem;
        totalEmptyQueueTime += simulationTime;
    }
```

```
    else {
        // assert queueSize > 0
        lastWaitingCar -> next = newQueueItem;
    }

    lastWaitingCar = newQueueItem;
    localQueueSize += 1;
}

carClass * carQueueClass::getNext ()
{
    // pre queueSize > 0 and firstWaitingCar not= NULL

    if (firstWaitingCar == NULL) {
        cout << "Error! car queue unexpectedly empty0;
        return NULL;
    }

    carClass * carToReturn = firstWaitingCar -> data;

    localQueueSize -= 1;
    queueItem * restOfList = firstWaitingCar -> next;
    delete firstWaitingCar;
    firstWaitingCar = restOfList;

    if (firstWaitingCar == NULL) {
        // empty queue: update the pointer to the end of queue too!
        lastWaitingCar = NULL;
        totalEmptyQueueTime -= simulationTime;
    }

    return carToReturn;
}

//-------------------------------------
// Collecting and displaying statistics
//-------------------------------------

class statsClass {
private:
    int TotalArrivals, customersServed, balkingCustomers;
    double TotalLitresSold, TotalLitresMissed, TotalWaitingTime,
        TotalServiceTime;

public:
    statsClass ();
    void countArrival () {TotalArrivals += 1;};
    void accumSale (double litres);
    void accumBalk (double litres);
    void accumWaitingTime (double interval) {TotalWaitingTime += interval;};
    void accumServiceTime (double interval) {TotalServiceTime += interval;};
    void snapshot ();
};
```

```
statsClass * stats;

statsClass::statsClass ()
{
    TotalArrivals = 0;
    customersServed = 0;
    balkingCustomers = 0;
    TotalLitresSold = 0.0;
    TotalLitresMissed = 0.0;
    TotalWaitingTime = 0.0;
    TotalServiceTime = 0.0;
}

void statsClass::accumSale (double litres)
{
    customersServed += 1;
    TotalLitresSold += litres;
}

void statsClass::accumBalk (double litres)
{
    balkingCustomers += 1;
    TotalLitresMissed += litres;
}

void statsClass::snapshot ()
{
    printf("%8.0f%7i", simulationTime, TotalArrivals);
    printf("%8.3f", carQueue -> emptyTime()/simulationTime);
    if (TotalArrivals > 0) {
        printf("%9.3f%8.3f", simulationTime/TotalArrivals,
            (TotalLitresSold + TotalLitresMissed) / TotalArrivals);
    }
    else
        printf ("%9s%8s", "Unknown", "Unknown");

    printf ("%7i", balkingCustomers);
    if (customersServed > 0)
        printf ("%9.3f", TotalWaitingTime / customersServed);
    else
        printf ("%9s", "Unknown");
    printf ("%7.3f", TotalServiceTime
        / (pumpStand -> howManyPumps() * simulationTime));
    printf ("%9.2f", TotalLitresSold * profit
        - cost * pumpStand -> howManyPumps());
    printf ("%7.2f0, TotalLitresMissed * profit);
}

//----------------------------------------
// Classes for the specific kinds of events
//----------------------------------------

class arrivalClass : public eventClass {
private:
```

```
    double interarrivalTime();
    BOOLEAN DoesCarBalk (double litres, int queueLength);

public:
    arrivalClass (double T) : eventClass (T) {};
    void makeItHappen ();
};

double arrivalClass::interarrivalTime ()
// The interarrival time of the next car is exponentially distributed.
{
    const int MEAN = 50;
    // seconds, because r is a rate in 1/sec, and this is 1/r
    return - MEAN * log (arrivalStream -> nextrand());
}

BOOLEAN arrivalClass::DoesCarBalk (double litres, int queueLength)
// Deciding whether to balk is an activity that forms part of the
// arrival event, so this procedure belongs among the event routines.
//
// 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 requries 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))
{
    return queueLength > 0 && balkingStream -> nextrand()
        > (40 + litres) / (25 * (3 + queueLength));
}

class departureClass : public eventClass {
private:
    pumpClass * whichPump;

public:
    departureClass (double T) : eventClass (T) {};
    void setPump (pumpClass * p) {whichPump = p;};
    void makeItHappen ();
};

class reportClass : public eventClass {
private:
    double intervalToNext; // time between regular reports

public:
    reportClass (double T) : eventClass (T) {};
    void setInterval (double interval) {intervalToNext = interval;};
    void makeItHappen ();
};

void reportClass::makeItHappen ()
{
    stats -> snapshot ();
```

```
    // schedule the next interim report.
    setTime (simulationTime + intervalToNext);
    eventList -> insert (this);
}

class allDoneClass : public eventClass {
public:
    allDoneClass (double T) : eventClass (T) {};
    void makeItHappen () {stats -> snapshot (); simulating = FALSE;};
};

//---------------
// Event routines
//---------------

void arrivalClass::makeItHappen ()
// The car arrival event routine
{
    // Create and initialize a new auto record
    carClass * arrivingCar = new carClass;
    stats -> countArrival ();
    const double L = arrivingCar -> litresNeeded();
    if (DoesCarBalk (L, carQueue->queueSize())) {
        stats -> accumBalk (L);
        delete arrivingCar;
    }
    else {
        arrivingCar -> setTime (simulationTime);
        if (pumpStand -> existsAvailablePump()) {
            pumpClass * Pump = pumpStand -> getAvailablePump ();
            Pump -> startService (arrivingCar);
        }
        else
            carQueue -> insert (arrivingCar);
    }

    // schedule the next arrival, reusing the current event object
    setTime (simulationTime + interarrivalTime());
    eventList -> insert (this);
}

void departureClass::makeItHappen ()
// The departure event routine
{
    // pre whichPump not= NULL and whichPump -> carInService not= NULL

    // Identify the departing car and collect statistics.
    carClass * departingCar = whichPump -> carInService();
    stats -> accumSale (departingCar -> litresNeeded());

    // The car vanishes and the pump is free; can we serve another car?
    delete departingCar;
    if (carQueue->queueSize() > 0)
        whichPump -> startService (carQueue -> getNext());
```

```
    else
        pumpStand -> releasePump (whichPump);

    // This departure event is all done.
    delete this;
}

void pumpClass::startService (carClass * Car)
// The start-of-service event routine:
// Connect the Car to this pump, and determine when the service will stop.
{
    // pre pumpStand -> existsAvailablePump();

    // Match the auto to an available pump
    localCarInService = Car;
    const double pumpTime = serviceTime ();

    // Collect statistics
    stats -> accumWaitingTime (simulationTime
        - localCarInService -> arrivalTime());
    stats -> accumServiceTime (pumpTime);

    // Schedule departure of car from this pump
    departureClass * Dep = new departureClass (simulationTime + pumpTime);
    Dep -> setPump (this);
    eventList -> insert (Dep);
}

int main()
{
    //---------------
    // Initialization
    //---------------

    simulationTime = 0.0;
    simulating = TRUE;

    double ReportInterval, endingTime;
    cin >> ReportInterval >> endingTime;

    int numPumps;
    cin >> numPumps;
    cout << "This simulation run uses " << numPumps << " pumps ";
    pumpStand = new pumpStandClass (numPumps);

    // Initialize the random-number streams.

    cout << "and the following random number seeds:0;
    unsigned short seed;
    cin >> seed;
    cout << "         " << seed;
    arrivalStream = new randStream (seed);
    cin >> seed;
    cout << "         " << seed;
```

```
litreStream = new randStream (seed);
cin >> seed;
cout << "         " << seed;
balkingStream = new randStream (seed);
cin >> seed;
cout << "         " << seed;
serviceStream = new randStream (seed);
cout << "0;

// Create and initialize the event queue, the car queue,
// and the statistical variables.

eventList = new eventListClass;
carQueue = new carQueueClass;
stats = new statsClass;

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

allDoneClass * lastEvent = new allDoneClass (endingTime);
eventList -> insert (lastEvent);
if (ReportInterval <= endingTime) {
    reportClass * nextReport = new reportClass (ReportInterval);
    nextReport -> setInterval (ReportInterval);
    eventList -> insert (nextReport);
}

// Schedule the first car to arrive at the start of the simulation

arrivalClass * nextArrival = new arrivalClass (0);
    // Is 0 really the time for the first arrival?
eventList -> insert (nextArrival);

// Print column headings for periodic progress reports and final report

printf ("%9s%7s%8s%9s%8s%7s%9s%7s%8s%7s0, " Current", "Total ",
    "NoQueue", "Car->Car", "Average", "Number", "Average", "Pump ",
    "Total", " Lost ");
printf ("%9s%7s%8s%9s%8s%7s%9s%7s%8s%7s0, "   Time ", "Cars ",
    "Fraction", "  Time  ", " Litres ", "Balked", "  Wait ",
    "Usage ", "Profit", "Profit");
for (int i = 0; i < 79; i++)
    cout << "-";
cout << "0;

//------------------------
// The "clock driver" loop
//------------------------

while (simulating) {
    eventClass * currentEvent = eventList -> getNext ();
    simulationTime = currentEvent -> whatTime();
    currentEvent -> makeItHappen();
}
```

```
    return 0;
}
```

### 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 3 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    369   0.510   54.201  34.829     61   51.108  0.864   213.05  48.25
   40000    760   0.478   52.632  34.875    132   58.704  0.873   502.27 100.36
   60000   1154   0.447   51.993  35.293    209   64.662  0.883   799.43 158.77
   80000   1557   0.444   51.381  35.392    296   67.161  0.883  1095.45 222.17
  100000   2029   0.406   49.285  35.619    422   74.775  0.899  1422.18 324.57
  120000   2463   0.398   48.721  35.845    534   77.421  0.903  1732.32 414.84
  140000   2852   0.403   49.088  35.736    614   75.759  0.900  2013.64 474.30
  160000   3269   0.405   48.945  35.635    708   75.187  0.900  2309.15 543.08
  180000   3654   0.413   49.261  35.658    782   73.726  0.898  2596.93 600.39
  200000   4065   0.416   49.200  35.764    874   72.890  0.897  2902.09 672.45
```

#### 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, such as servers that appear to be busy more than 100 percent of the time at the first few progress reports.

```
This simulation run uses 2 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     32   0.365   62.500 33.606      5   79.902 0.987   -16.66   3.55
     4000     73   0.259   54.795 33.626     19  104.945 0.993     6.63  14.74
     6000    122   0.173   49.180 34.570     45  214.501 0.991    25.06  40.38
     8000    149   0.222   53.691 34.690     56  201.399 0.952    41.81  47.41
    10000    183   0.239   54.645 34.510     67  184.811 0.948    61.87  56.01
    12000    211   0.234   56.872 35.224     74  176.170 0.958    83.90  61.91
    14000    242   0.248   57.851 34.762     79  181.328 0.955   103.86  66.45
    16000    284   0.217   56.338 34.956     98  204.478 0.958   125.44  82.74
    18000    323   0.197   55.728 34.616    112  204.344 0.962   146.19  93.33
    20000    369   0.177   54.201 34.614    135  216.659 0.967   166.80 112.52
```

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 $\bar{X} \pm C$, that contain the true mean, say $M$, a specified fraction of the time (the greater the fraction, the wider the resulting confidence interval about $\bar{X}$). This construction is based on the following two facts about the Normal distribution with mean $M$ and variance $\sigma^2$. 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 $\sigma^2$ also has a Normal distribution, but with mean $M$ and variance $\sigma^2/N$. 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 $\sigma^2$

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 $\sigma^2$ (which depends on $M$). To get out of this circular chain of reasoning, we need a method of estimating $\sigma^2$ 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 $\sigma^2$, 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 $\sigma^2$ 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 might look like the lines of this table:

| Current Time | Total Cars | NoQueue Fraction | Car->Car Time | Average Litres | Number Balked | Average Wait | Pump Usage | Total Profit | Lost 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 $\sigma^2$. 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.