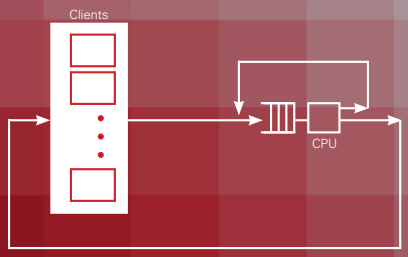
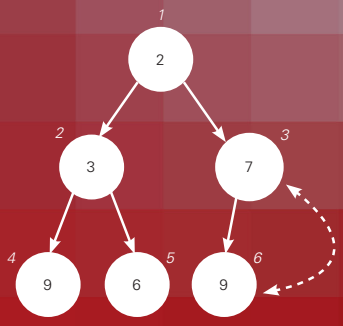
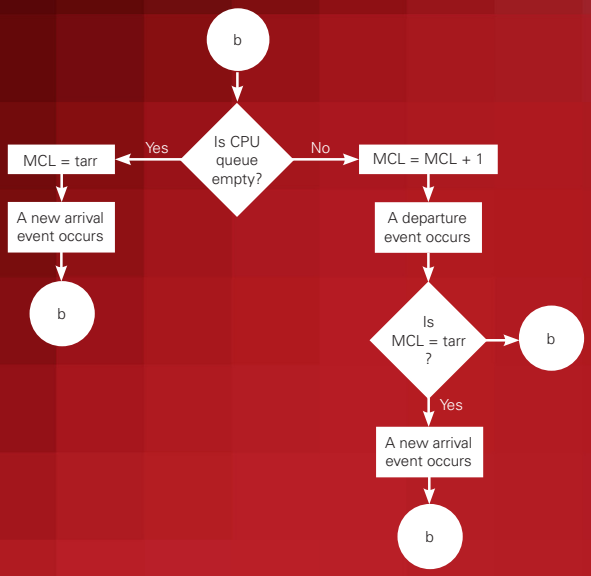


Computer Simulation Techniques: The definitive introduction!





Harry Perros

Computer Simulation Techniques: The definitive introduction!

by

Harry Perros
Computer Science Department
NC State University
Raleigh, NC

TO THE READER…

This book is available for free download from my web site:
http://www.csc.ncsu.edu/faculty/perros//simulation.pdf

***Self-study**: You can use this book to learn the basic simulation techniques by yourself! At the end of Chapter 1, you will find three examples. Select one of them and do the corresponding computer assignment given at the end of the Chapter. Then, after you read each new Chapter, do all the problems at the end of the Chapter and also do the computer assignment that corresponds to your problem, By the time you reach the end of the book, you will have developed a very sophisticated simulation model!*
You can use any high-level programming language you like.

***Errors**: I am not responsible for any errors in the book, and if you do find any, I would appreciate it if you could let me know (hp@csc.ncsu.edu).*

***Acknowledgment**: Please acknowledge this book, if you use it in a course, or in a project, or in a publication.*

I would like to thank Attila Yavuz and Amar Shroff for helping me revise Chapters 2 and 4 and also my thanks to many other students for their suggestions.

Enjoy!

Harry Perros, December 2009

TABLE OF CONTENTS

CHAPTER 1:

INTRODUCTION

1.1 The OR approach

The OR approach to solving problems is characterized by the following steps:

1. Problem formulation
2. Construction of the model
3. Model validation
4. Using the model, evaluate various available alternatives (solution)
5. Implementation and maintenance of the solution

The above steps are not carried out just once. Rather, during the course of an OR exercise, one frequently goes back to earlier steps. For instance, during model validation one frequently has to examine the basic assumptions made in steps 1 and 2.

The basic feature of the OR approach is that of *model building*. Operations Researchers like to call themselves model builders! A model is a representation of the structure of a real-life system. In general, models can be classified as follows: *iconic*, *analogue*, and *symbolic*.

An iconic model is an exact replica of the properties of the real-life system, but in smaller scale. Examples are: model airplanes, maps, etc. An analogue model uses a set of properties to represent the properties of a real-life system. For instance, a hydraulic system can be used as an analogue of electrical, traffic and economic systems. Symbolic models represent the properties of the real-life system through the means of symbols,

such as mathematical equations and computer programs. Obviously, simulation models are symbolic models.

Operations Research models are in general symbolic models and they can be classified into two groups, namely *deterministic models* and *stochastic models*. Deterministic models are models which do not contain the element of probability. Examples are: linear programming, non-linear programming and dynamic programming. Stochastic models are models which contain the element of probability. Examples are: queueing theory, stochastic processes, reliability, and simulation techniques.

Simulation techniques rely heavily on the element of randomness. However, deterministic simulation techniques in which there is a no randomness, are not uncommon. Simulation techniques are easy to learn and are applicable to a wide range of problems. Simulation is probably the handiest technique to use amongst the array of OR models. The argument goes that "when everything else fails, then simulate". That is, when other models are not applicable, then use simulation techniques.

1.2 Building a simulation model

Any real-life system studied by simulation techniques (or for that matter by any other OR model) is viewed as a system. A system, in general, is a collection of entities which are logically related and which are of interest to a particular application. The following features of a system are of interest:

1. *Environment*: Each system can be seen as a subsystem of a broader system.
2. *Interdependency*: No activity takes place in total isolation.
3. *Sub-systems*: Each system can be broken down to sub-systems.
4. *Organization*: Virtually all systems consist of highly organized elements or components, which interact in order to carry out the function of the system.
5. *Change*: The present condition or state of the system usually varies over a long period of time.

When building a simulation model of a real-life system under investigation, one does not simulate the whole system. Rather, one simulates those sub-systems which are related to the problems at hand. This involves modelling parts of the system at various levels of detail. This can be graphically depicted using Beard's managerial pyramid as shown in Figure 1.1. The collection of blackened areas form those parts of the system that are incorporated in the model.

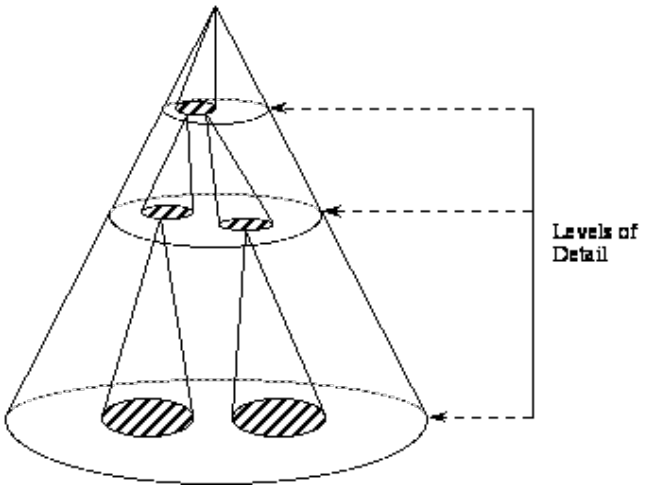


Figure 1.1: Beard's managerial pyramid

A simulation model is, in general, used in order to study real-life systems which do not currently exist. In particular, one is interested in quantifying the performance of a system under study for various values of its input parameters. Such quantified measures of performance can be very useful in the managerial decision process. The basic steps involved in carrying out a simulation exercise are depicted in Figure 1.2.

All the relevant variables of a system under study are organized into two groups. Those which are considered as given and are not to be manipulated (*uncontrollable* variable) and those which are to be manipulated so that to come to a solution (*controllable* variables). The distinction between controllable and uncontrollable variables mainly depends upon the scope of the study.

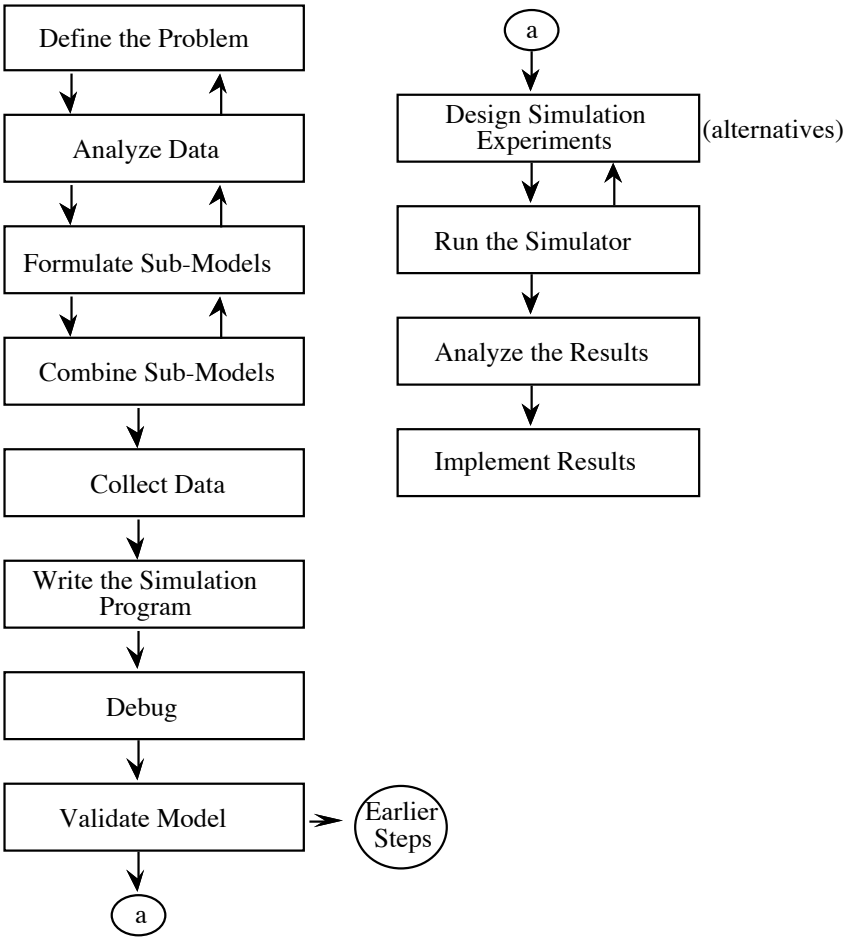


Figure 1.2: Basic steps involved in carrying out a simulation study.

Another characterization of the relevant variables is whether they are affected or not during a simulation run. A variable whose value is not affected is called *exogenous*. A variable having a value determined by other variables during the course of the simulation is called *endogenous*. For instance, when simulating a single server queue, the following variables may be identified and characterized accordingly.

Exogenous variables

1. The time interval between two successive arrivals.
2. The service time of a customer.
3. Number of servers.

4. Priority discipline.

Endogenous variables

1. Mean waiting time in the queue.
2. Mean number of customers in the queue.

The above variables may be controllable or uncontrollable depending upon the experiments we want to carry out. For instance, if we wish to find the impact of the number of servers on the mean waiting time in the queue, then the number of servers becomes an controllable variable. The remaining variables-the time interval between two arrivals and the service time, will remain fixed. (uncontrollable variables)

Some of the variables of the system that are of paramount importance are those used to define the status of the system. Such variables are known as *status* variables. These variables form the backbone of any simulation model. At any instance, during a simulation run, one should be able to determine how things stand in the system using these variables. Obviously, the selection of these variables is affected by what kind of information regarding the system one wants to maintain.

We now proceed to identify the basic simulation methodology through the means of a few simulation examples.

1.3 Basic simulation methodology: Examples

1.3.1 The machine interference problem

Let us consider a single server queue with a finite population known as the *machine interference* problem. This problem arose originally out of a need to model the behavior of machines. Later on, it was used extensively in computer modelling. Let us consider M machines. Each machine is operational for a period of time and then it breaks down. We assume that there is one repairman. A machine remains broken down until it is fixed by the repairman. Broken down machines are served in a FIFO manner, and the service is

non-preemptive. Obviously, the total down time of a machine is made up of the time it has to "queue" for the repairman and the time it takes for the repairman to fix it. A machine becomes immediately operational after it has been fixed. Thus, each machine follows a basic cycle as shown in figure 1.3, which is repeated continuously.

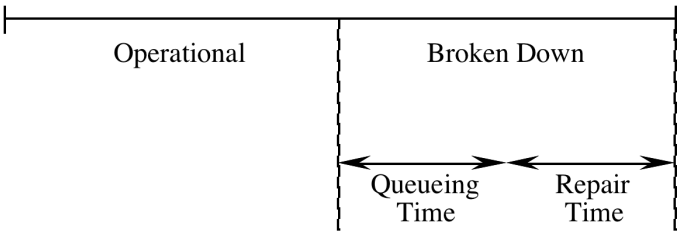


Figure 1.3: The basic cycle of a machine.

In general, one has information regarding the operational time and the repair time of a machine. However, in order to determine the down time of a machine, one should be able to calculate the queueing time for the repairman. If this quantity is known, then one can calculate the utilization of a machine. Other quantities of interest could be the utilization of the repairman.

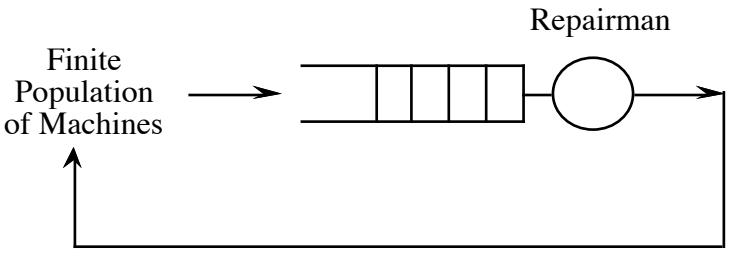


Figure 1.4: The machine interference problem.

Let us now look at the repairman's queue. This can be visualized as a single server queue fed by a finite population of customers, as shown in figure 1.4

For simplicity, we will assume that the operational time of each machine is equal to 10 units of time. Also, the repair time of each machine is equal to 5 units of time. In other words, we assume that all the machines have identical constant operational times.

They also have identical and constant repair times. (This can be easily changed to more complicated cases where each machine has its own random operational and repair times.)

The first and most important step in building a simulation model of the above system, is to identify the basic *events* whose occurrence will alter the *status* of the system. This brings up the problem of having to define the status variables of the above problem. The selection of the status variables depends mainly upon the type of performance measures we want to obtain about the system under study.

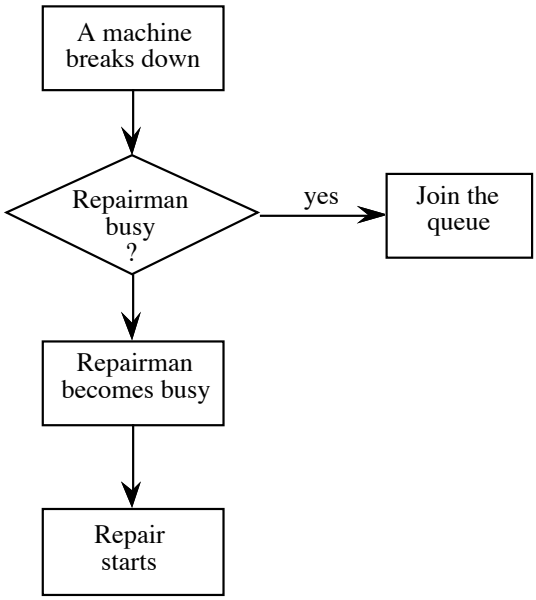


Figure 1.5: An arrival event.

In this problem, the most important status variable is n, the number of broken down machines, i.e., those waiting in the queue plus the one being repaired. If n=0, then we know that the queue is empty and the repairman is idle. If n=1, then the queue is empty and the repairman is busy. If n>1, then the repairman is busy and there are n-1 broken down machines in the queue. Now, there are two events whose occurrence will cause n to change. These are:

1. A machine breaks down, i.e., an arrival occurs at the queue.

2. A machine is fixed, i.e., a departure occurs from the queue.

The flow-charts given in figures 1.5, and 1.6 show what happens when each of these events occur.

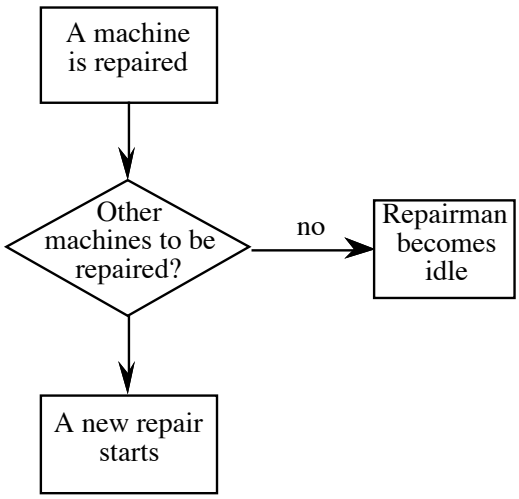


Figure 1.6: A departure event.

In order to incorporate the above two basic events in the simulation model, we need a set of variables, known as *clocks*, which will keep track of the time instants at which an arrival or departure event will occur. In particular, for this specific model, we need to associate a clock for each machine. The clock will simply show the time instant at which the machine will break down, i. e., it will arrive at the repairman's queue. Obviously, at any instance, only the clocks of the operational machines are of interest. In addition to these clocks, we require to have another clock which shows the time instant at which a machine currently being repaired will become operational, i.e., it will cause a departure event to occur. Thus, in total, if we have m machines, we need m+1 clocks. Each of these clocks is associated with the occurrence of an event. In particular, m clocks are associated with m arrival events and one clock is associated with the departure event.

In addition to these clocks, it is useful to maintain a *master clock*, which simply keeps track of the simulated time.

The heart of the simulation model centers around the manipulation of these events. In particular, using the above clocks, the model decides which of all the possible events will occur next. Then the master clock is advanced to this time instant, and the model takes action as indicated in the flow-charts given in figures 1.5 and 1.6. This event manipulation approach is depicted in figure 1.7

.

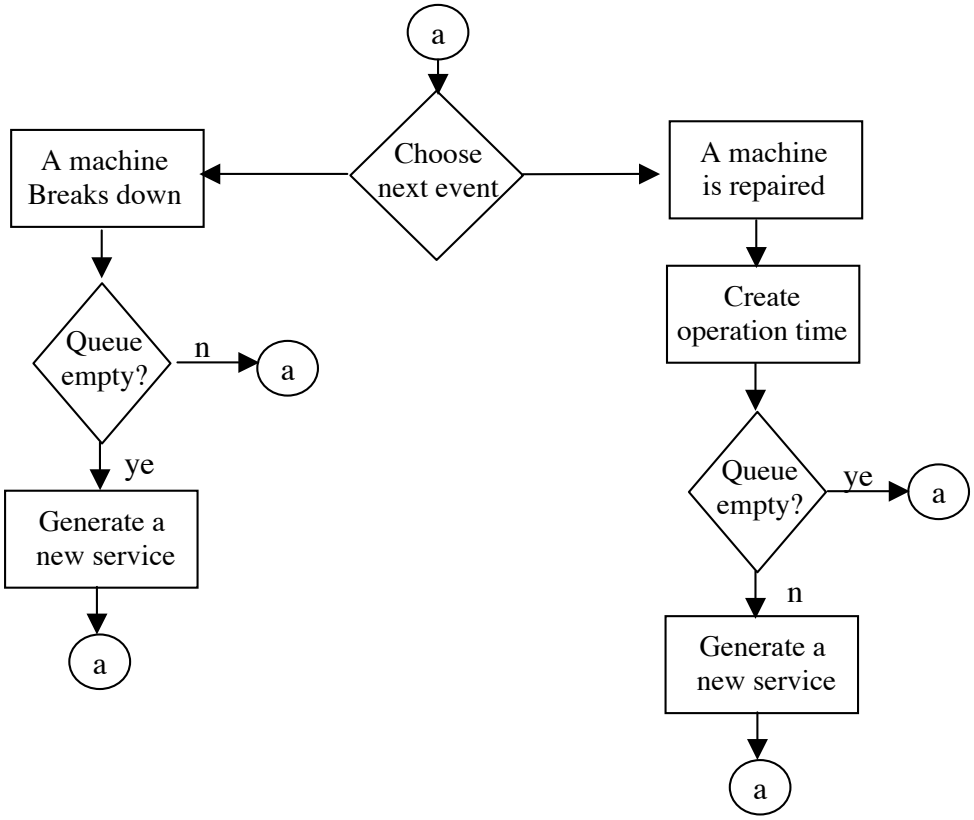


Figure 1.7: Event manipulation.

We are now ready to carry out the hand simulation shown below in table 1. Let us assume that we have 3 machines. Let CL1, CL2, and CL3 be the clocks associated with machine 1, 2, and 3 respectively (arrival event clocks). Let CL4 be the clock associated

with the departure event. Finally, let MC be the master clock and let R indicate whether the repairman is busy or idle. We assume that at time zero all three machines are operational and that CL1=1, CL2=4, CL3=9. (These are known as *initial conditions*.)

MC	CL1	CL2	CL3	CL4	n	
0	1	4	9	-	0	idle
1	-	4	9	6	1	busy
4	-	-	9	6	2	busy
6	16	-	9	11	1	busy
9	16	-	-	11	2	busy
11	16	21	-	16	1	busy
16	-	21	26	21	1	busy

Table 1: Hand simulation for the machine interference problem

We note that in order to schedule a new arrival time we simply have to set the associated clock to MC+10. Similarly, each time a new repair service begins we set CL4=MC+5. A very important aspect of this simulation model is that we only check the system at time instants at which an event takes place. We observe in the above hand simulation that the master clock in the above example gives us the time instants at which something happened (i.e., an event occurred). These times are: 0, 1, 4, 6, 9, 11, 16, ... We note that in-between these instants no event occurs and, therefore, the system's status remains unchanged. In view of this, it suffices to check the system at time instants at which an event occurs. Furthermore, having taken care of an event, we simply advance the Master clock to the next event which has the smallest clock time. For instance, in the above hand simulation, after we have taken care of a departure of time 11, the simulation will advance to time 16. This is because following the time instant 11, there are three events that are scheduled to take place in the future. These are: a) arrival of machine 1 at time 16; b) arrival of machine 2 at time 21; and c) a departure of machine 3 at time 16. Obviously, the next event to occur is the latter event at time 16.

The above simulation can be easily done using a computer program. An outline of the flow-chart of the simulation program is given in figure 1.8. The actual implementation of this simulation model is left as an exercise.

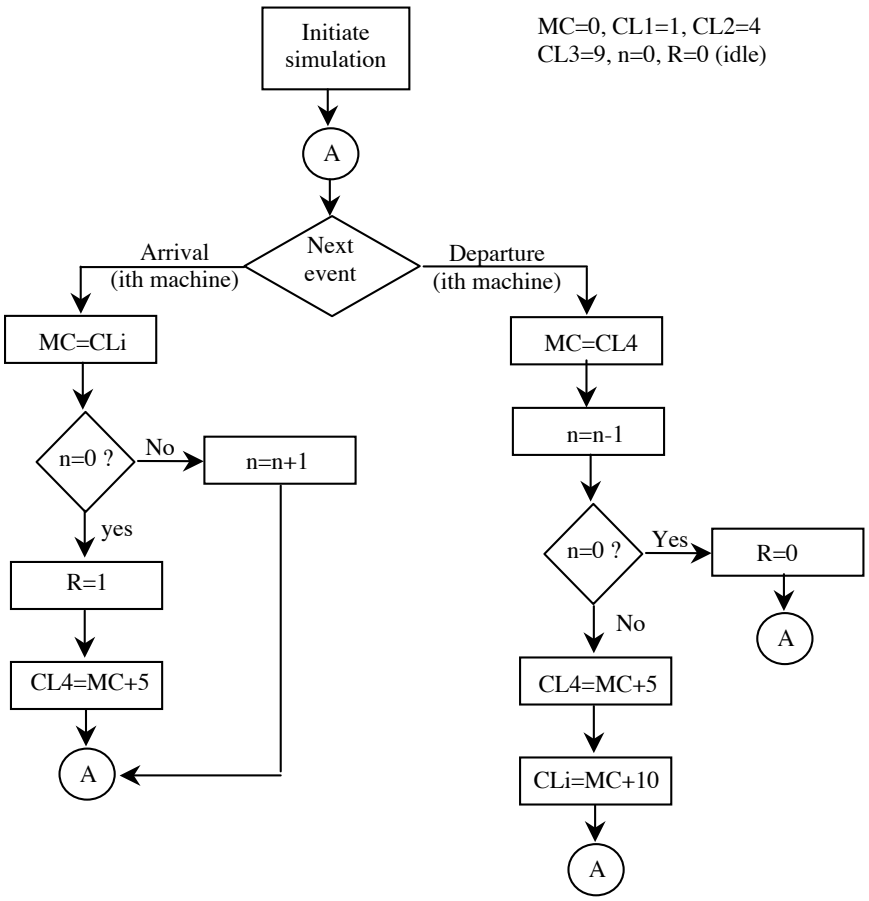


Figure 1.8: A flowchart of the simulation program.

1.3.2 A token-based access scheme

We consider a computer network consisting of a number of nodes interconnected via a shared wired or wireless transport medium, as shown in figure 1.9. Access to the shared medium is controlled by a token. That is, a node cannot transmit on the network unless it has the token. In this example, we simulate a simplified version of such an access scheme, as described below.

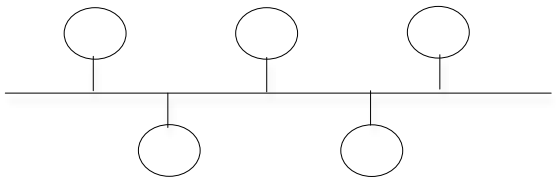


Figure 1.9: Nodes interconnected by a shared medium

There is a single token that visits the nodes in a certain logical sequence. The nodes are logically connected so that they form a logical ring. In the case of a bus-based or ring-based wireline medium, the order in which the nodes are logically linked may not be the same with the order in which they are attached to the network. We will assume that the token never gets lost. A node cannot transmit unless it has the token. When a node receives the token, from its previous logical upstream node, it may keep it for a period of time up to T. During this time, the node transmits packets. A packet is assumed to consist of data and a header. The header consists of the address of the sender, the address of the destination, and various control fields. The node surrenders the token when: a) time T has run out, or b) it has transmitted out all the packets in its queue before T expires, or c) it receives the token at a time when it has no packets in its queue to transmit. If time T runs out and the node is in the process of transmitting a packet, it will complete the transmission and then it will surrender the token. Surrendering the token means, that the node will transmit it to its next downstream logical neighbor.

Conceptually, this network can be seen as comprising of a number of queues, one per node. Only the queue that has the token can transmit packets. The token can be seen as a server, who switches cyclically between the queues, as shown in figure 1.10. Once the token is switched to a queue, packets waiting in this queue can be transmitted on the network. The maximum time that a queue can keep the token is T units of time, and the token is surrendered following the rules described above. The time it takes for the token to switch from one queue to the next is known as switch-over time. Such queueing systems are referred to as polling systems, and they have been studied in the literature under a variety of assumptions.

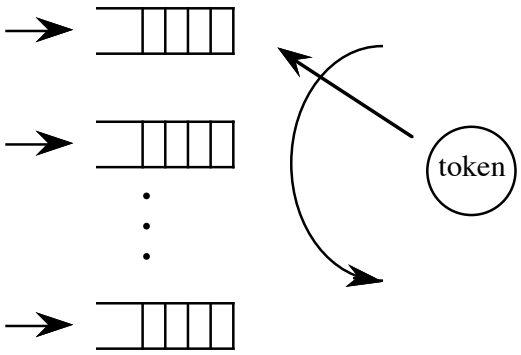


Figure 1.10. The conceptual queueing system.

It is much simpler to use the queueing model given in figure 1.10 when constructing the simulation model of this access scheme. The following events have to be taken into account in this simulation. For each queue, there is an arrival event and service completion event. For the token, there is a time of arrival at the next queue event and the time when the token has to be surrendered to the next node, known as the time-out. For each queue, we keep track of the time of arrival of the next packet, the number of customers in the queue, and the time a packet is scheduled to depart, if it is being transmitted. For the token, we keep track of the time of arrival at the next queue, the number of the queue that may hold the token, and the time-out.

In the hand simulation given below we assume that the token-based network consists of three nodes. That is, the queueing system in figure 1.10 consists of three queues. The inter-arrival times to queues 1, 2, and 3 are constant and they are equal to 10, 15, and 20 unit times respectively. T is assumed to be equal to 15 unit times. The time it takes to transmit a packet is assumed to be constant equal to 6 unit times. The switch over time is equal to 1 unit time. For initial conditions we assume that the system is empty at time zero, and the first arrival to queues 1, 2, and 3 will occur at time 2, 4 and 6 respectively. Also, at time zero, the token is in queue 1. In case when an arrival and a departure occur simultaneously at the same queue, we will assume that the arrival occurs first. In addition, if the token and a packet arrive at a queue at the same time, we will assume that the packet arrives first.

The following variables represent the clocks used in the flow-charts:

a. MC: Master clock
b. AT_i: Arrival time clock at queue *i*, *i*=1,2,3
c. DT_i: Departure time clock from queue *i*, i=1,2,3
d. TOUT: Time out clock for token
e. ANH: Arrival time clock of token to next queue

The logic for each of the events in this simulation is summarized in figures 1.11 to 1.14. Specifically, in figure 1.11 we show what happens when an arrival occurs, figure 1.12 deals with the case where a service is completed at queue *i*, figure 1.13 describes the action taken when a time-out occurs, and figure 1.14 summarizes the action taken when the token arrives at a queue.

The set of all possible events that can occur at a given time in a simulation is known as the Figure 1.13 *event list*. This event-list has to be searched each time in order to locate the next event. This event manipulation forms the basis of the simulation, and it is summarized in figure 1.15.

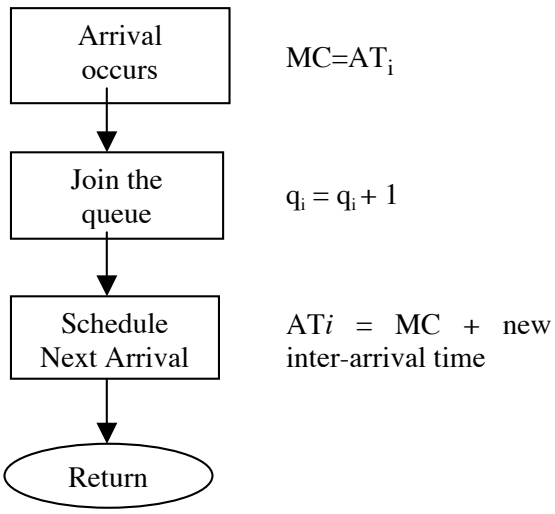


Figure 1.11: Arrival event at queue i.

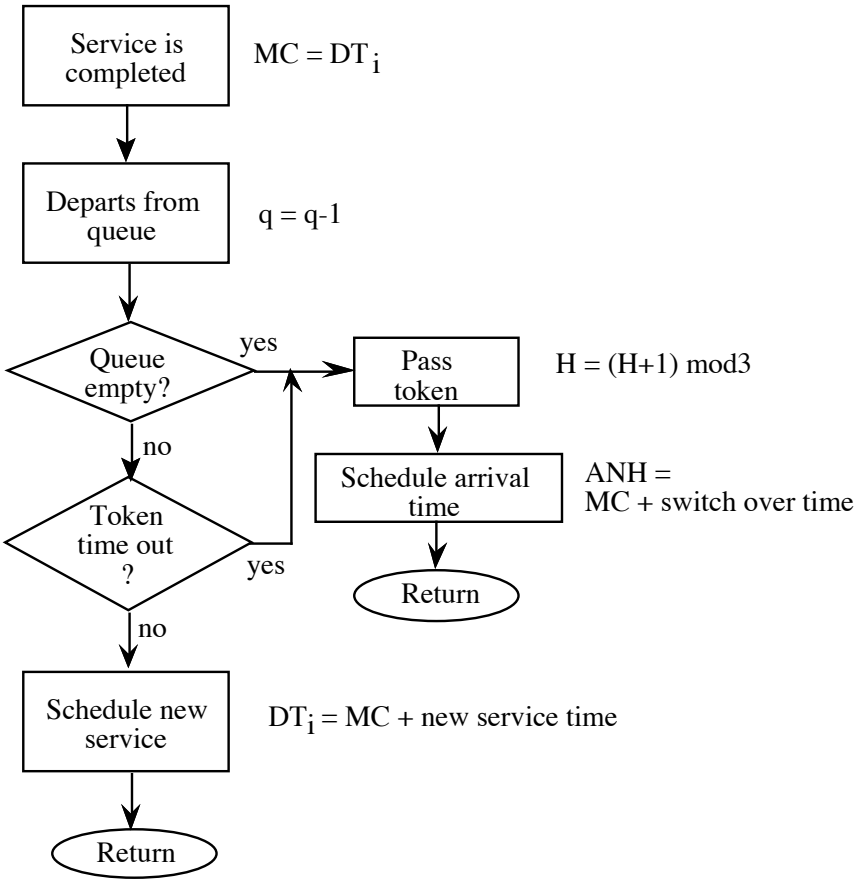


Figure 1.12: Service completion at queue i.

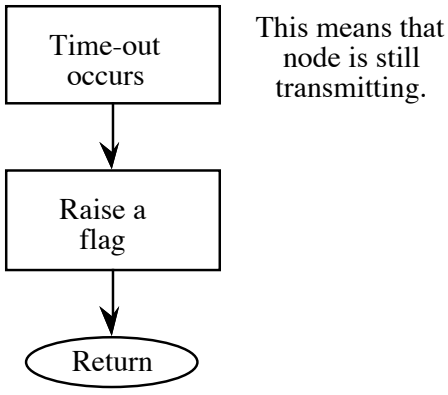


Figure 1.13 : Time-out of token

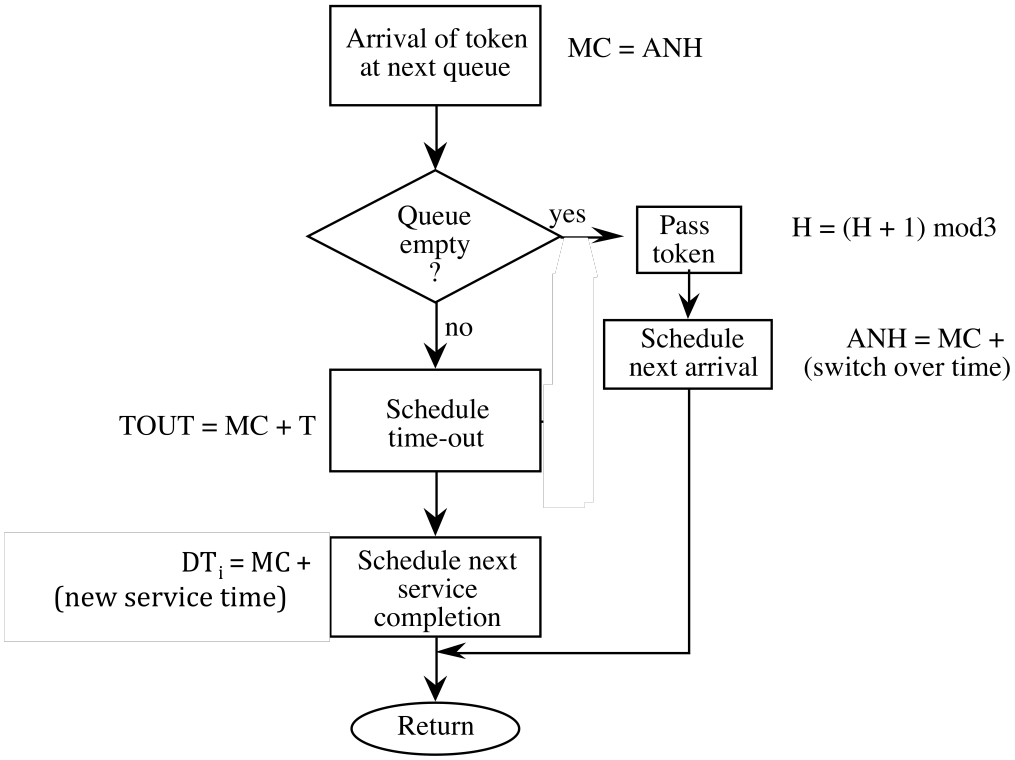


Figure 1.14 : Arrival of token at next queue.

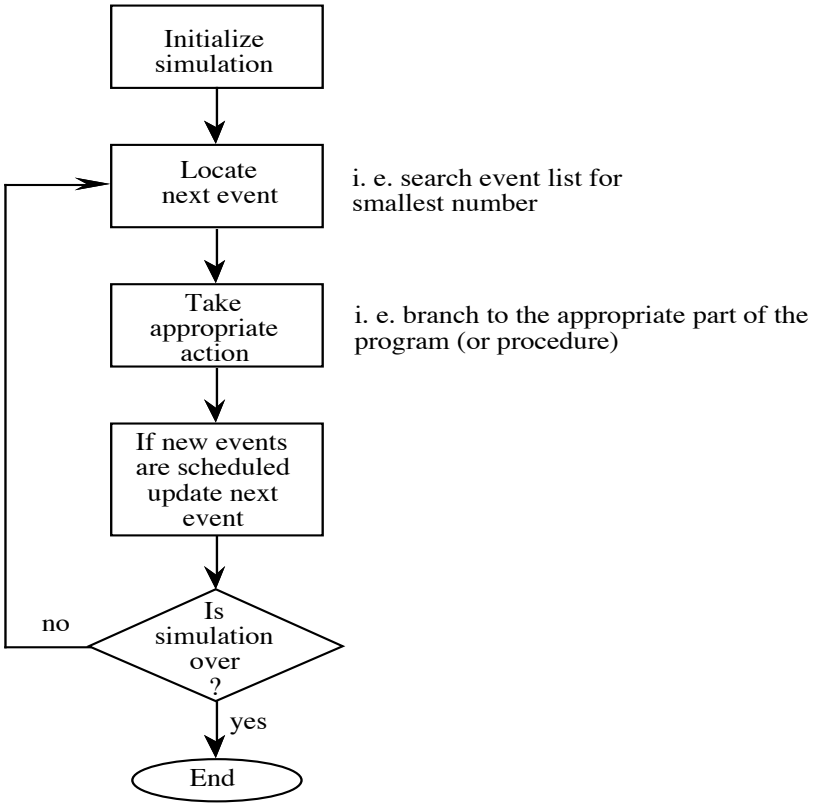


Figure 1.15 : Event manipulation.

The hand simulation is given in table 2.

	Queue 1			**Queue 2**			**Queue 3**			**Token**		
MC	Arr Clock	Depart Clock	Queue size	Arr Clock	Depart Clock	Queue size	Arr Clock	Depart Clock	Queue size	Node No	Time Out Clock	Arrival Next Node
0	2		0	4		0	6		0	1		1
1	2		0	4		0	6		0	2		2
2	12		1	4		0	6		0	3		3
3	12	9	1	4		0	6		0	1	18	
4	12	9	1	19		1	6		0	1	18	
6	12	9	1	19		1	26		1	1	18	
9	12		0	19		1	26		1	1		10
10	12		0	19	16	1	26		1	2	25	
12	22		1	19	16	1	26		1	2	25	
16	22		1	19		0	26		1	2		17
17	22		1	19		0	26	23	1	3	32	
19	22		1	34		1	26	23	1	3	32	
22	32		2	34		1	26	23	1	3	32	
23	32		2	34		1	26		0	3		24
24	32	30	2	34		1	26		0	1	39	
26	32	30	2	34		1	46		1	1	39	
30	32	36	1	34		1	46		1	1	39	
32	42	36	2	34		1	46		1	1	39	
34	42	36	2	49		2	46		1	1	39	
36	42	42	1	49		2	46		1	1	39	
39	42	42	1	49		2	46		1	1	*	
42	52	42	2	49		2	46		1	1	*	
42	52		1	49		2	46		1	1		43
43	52		1	49	49	2	46		1	2	58	

Table 2: Hand simulation for the token-based access scheme

1.3.3 A two-stage manufacturing system

Let us consider a two-stage manufacturing system as depicted by the queueing network shown in figure 1.16. Stage 1 consists of an infinite capacity queue, referred to as queue 1, served by a single server, referred to as server 1. Stage 2, consists of a finite capacity queue, referred to as queue 2, served by a single server, referred to as server 2.

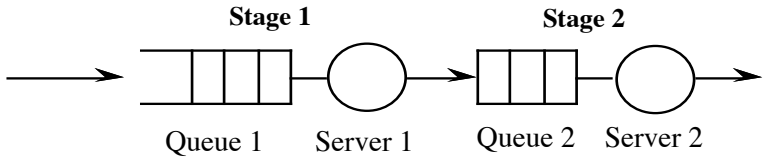


Figure 1.16: A two-stage queueing network.

When queue 2 becomes full, server 1 stops. More specifically, upon service completion at server 1, the server gets blocked if the queue 2 is full. That is, the server cannot serve any other customers that may be waiting in queue 1. Server 1 will remain blocked until a customer departs from queue 2. In this case, a space will become available in queue 2 and the served customer in front of server 1 will be able to move into queue 2, thus freeing the server to serve other customer in queue 1.

Each server may also break down. For simplicity, we will assume that a server may break down whether it is busy or idle. A broken down server cannot provide service until it is repaired. If a customer was in service when the breakdown occurred, the customer will resume its service after the server is repaired without any loss to the service it received up to the time of the breakdown. That is, after the server becomes operational again, the customer will receive the balance of its service.

The following events (and associated clocks) may occur:

1. Arrival of a customer to queue 1 (clock AT)
2. Service completion at server 1 (clock DT1)
3. Service completion at server 2 (clock DT2)
4. Server 1 breaks down (clock BR1)

5. Server 1 becomes operational (clock OP1)
6. Server 2 breaks down (clock BR2)
7. Server 2 becomes operational (clock OP2)

Below, we briefly describe the events that may be triggered when each of the above events occurs.

1. *Arrival to queue 1*.
    a. *Arrival to queue 1 (new value for AT clock):* This event is always scheduled each time an arrival occurs.
    b. *Service completion at server 1 (new value for DT1 clock):* This event will be triggered if the new arrival finds the server idle.
2. *Service completion at server 1*:
    a. *Service completion at server 1(new value for DT1 clock):* This event will occur if there is one or more customers in queue 1.
    b. *Service completion at server 2 (new value for DT2 clock):* This event will occur if the customer who just completed its service at server 1 finds server 2 idle.

    The occurrence of a service completion event at server 1 may cause server 1 to get blocked, if queue 2 is full.
3. *Service completion at server 2*:
    a. *Service completion at server 2 (new value for D21 clock):* This event will occur if there is one or more customers in queue 2.
    b. *Service completion at server 1 (new value for DT1 clock):* This event will occur if server 1 was blocked.
4. *Server 1 breaks down:*
    a. *Server 1 becomes operational (new value for OP1 clock):* This event gives the time in the future when the server will be repaired and will become operational. If the server was busy when it broke down, update the clock of the service completion event at server 1 to reflect the delay due to the repair.

5. *Server 1 becomes operational:*
    a. *Server 1 breaks down (new value for BR1 clock):* This event gives the time in the future when the server will break down. During this time the server is operational.
    b. *Service completion time (new value for DT1):* If the server was idle when it broke down, and queue 1 is not empty at the moment it becomes operational, then a new service will begin.
6. *Server 2 breaks down:*
    a. *Server 2 becomes operational (new value for OP2 clock):* This event gives the time in the future when server 2 will be repaired, and therefore it will become operational. During that time the server is broken down.

    If server 2 was busy when it broke down, update the clock of the service completion event at server 2 to reflect the delay due to the repair.
7. *Server 2 becomes operational:*
    a. *Server 2 breaks down (new value for BR2 clock):* This event gives the time in the future when server 2 will break down. During this time the server is operational.
    b. *Service completion time (new value for DT2):* If the server was idle when it broke down, and queue 2 is not empty at the moment it becomes operational, then a new service will begin.

In the hand-simulation given in table 3, it is assumed that the buffer capacity of the second queue is 4 (this includes the customer in service). All service times, inter-arrival times, operational and repair times are constant with the following values: inter-arrival time = 40, service time at node 1 = 20, service time at node 2 = 30, operational time for server 1 = 200, operational time for server 2 = 300, repair time for server 1 = 50, and repair time for server 2 = 150. Initially the system is assumed to be empty. The first arrival occurs at time 10, server 1 will break down for the first time at time 80, and server 2 at time 90.

Stage 1							Stage 2				
MC	AT	#Cust	DT1	BR1	OP1	Server Status	# Cust	DT2	BR2	OP2	Server Status
10	50	1	30	80		busy			90		idle
30	50	0		80		idle	1	60	90		busy
50	90	1	70	80		busy	1	60	90		busy
60	90	1	70	80		busy	0		90		idle
70	90	0		80		idle	1	100	90		busy
80	90	0			130	down	1	100	90		busy
90	90	0			130	down	1	250		240	down
90	130	1	150		130	down	1	250		240	down
130	170	2	150		130	down	1	250		240	down
130	170	2	150	330		busy	1	250		240	down
150	170	1	170	330		busy	2	250		240	down
170	210	2	170	330		busy	2	250		240	down
170	210	1	190	330		busy	3	250		240	down
190	210	0		330		idle	4	250		240	down
210	250	1	230	330		busy	4	250		240	down
230	250	1		330		blocked	4	250		240	down
240	250	1		330		blocked	4	250	540		busy
250	290	2		330		blocked	4	250	540		busy
250	290	1	270	330		busy	4	280	540		busy
270	290	1		330		blocked	4	280	540		busy
280	290	0		330		idle	4	310	540		busy
290	330	1	310	330		busy	4	310	540		busy
310	330	1	310	330		busy	3	340	540		busy
310	330	0		330		idle	4	340	540		busy
330	370	1	350	330		busy	4	340	540		busy
330	370	1	400		380	down	4	340	540		busy
340	370	1	400		380	down	3	370	540		busy
370	410	2	400		380	down	3	370	540		busy
370	410	2	400		380	down	2	400	540		busy
380	410	2	400	580		busy	2	400	540		busy

Table 3: hand simulation for the two-stage manufacturing system

Since we are dealing with integer numbers, it is possible that more than one clock may have the same value. That is, more than one event may occur at the same time. In this particular simulation, simultaneous events can be taken care in any arbitrary order. In general, however, the order with which such events are dealt with may matter, and it has to be accounted for in the simulation. In a simulation, typically, clocks are represented by real numbers. Therefore, it is not possible to have events occurring at the same time.

Problems

1. Do the hand simulation of the machine interference problem, discussed in section 1.3.1, for the following cases:
   a. Vary the repair and operational times.
   b. Vary the number of repairmen.
   c. Assume that the machines are not repaired in a FIFO manner, but according to which machine has the shortest repair time.

2. Do the hand simulation of the token-based access scheme, described in section 1.3.2, for the following cases:
   a. Vary the inter-arrival times.
   b. Vary the number of queues.
   c. Assume that packets have priority 1 or 2 (1 being the highest). The packets in a queue are served according to their priority. Packets with the same priority are served in a FIFO manner.

3. Do the hand simulation of the two-stage manufacturing system, described in section 1.3.3, for the following cases:
   a. Assume no breakdowns.
   b. Assume a three-stage manufacturing system. (The third stage is similar to the second stage.)

Computer assignments

1. Write a computer program to simulate the machine interference problem as described in section 1.3.1. Each time an event occurs, print out a line of output to show the current values of the clocks and of the other status parameters (as in the hand simulation). Run your simulation until the master clock is equal to 20. Check by hand whether the simulation advances from event to event properly, and whether it updates the clocks and the other status parameters correctly.

2. Write a computer program to simulate the token-based access scheme as described in section 1.3.2. Each time an event occurs, print out a line of output to show the current values of the clocks and of the other status parameters (as in the hand simulation). Run your simulation until the master clock is equal to 100. Check by hand whether the simulation advances from event to event properly, and whether it updates the clocks and the other status parameters correctly.

3. Write a computer program to simulate the two-stage manufacturing system as described in section 1.3.3. Each time an event occurs, print out a line of output to show the current values of the clocks and of the other status parameters (as in the hand simulation). Run your simulation until the master clock is equal to 500. Check by hand whether the simulation advances from event to event properly, and whether it updates the clocks and the other status parameters correctly.

CHAPTER 2:

THE GENERATION OF PSEUDO-RANDOM NUMBERS

2.1 Introduction

We shall consider methods for generating random number uniformly distributed. Based on these methods, we shall then proceed in the next Chapter to consider methods for generating random numbers that have a certain distribution, i.e., exponential, normal, etc.

Numbers chosen at random are useful in a variety of applications. For instance, in numerical analysis, random numbers are used in the solution of complicated integrals. In computer programming, random numbers make a good source of data for testing the effectiveness of computer algorithms. Random numbers also play an important role in cryptography.

In simulation, random numbers are used in order to introduce randomness in the model. For instance, let us consider for a moment the machine interference simulation model discussed in the previous Chapter. In this model it was assumed that the operational time of a machine was constant. Also, it was assumed that the repair time of a machine was constant. It is possible that one may identify real-life situations where these two assumptions are valid. However, in most of the cases one will observe that the time a machine is operational varies. Also, the repair time may vary from machine to machine. If we are able to observe the operational times of a machine over a reasonably long period, we will find that they are typically characterized by a theoretical or an empirical probability distribution. Similarly, the repair times can be also characterized by a theoretical or empirical distribution. Therefore, in order to make the simulation model more realistic, one should be able to randomly numbers that follow a given theoretical or empirical distribution.

In order to generate such random numbers one needs to be able to generate uniformly distributed random numbers, otherwise known as *pseudo-random numbers*. These pseudo-random numbers can be used either by themselves or they can be used to generate random numbers from different theoretical or empirical distributions, known as *random variate*s or *stochastic variates*. In this Chapter, we focus our discussion on the generation and statistical testing of pseudo-random numbers. The generation of stochastic variates is described in Chapter 3.

2.2 Pseudo-random numbers

In essence, there is no such a thing as a *single* random number. Rather, we speak of a *sequence* of random numbers that follow a specified theoretical or empirical distribution. There are two main approaches to generating random numbers. In the first approach, a physical phenomenon is used as a source of randomness from where random numbers can be generated. Random numbers generated in this way are called *true random numbers*.

A true random number generator requires a completely unpredictable and non-reproducible source of randomness. Such sources can be found in nature, or they can be created from hardware and software. For instance, the elapsed time between emissions of particles during radioactive decay is a well-known randomized source. Also, the thermal noise from a semiconductor diode or resistor can be used as a randomized source. Finally, sampling a human computer interaction processes, such as keyboard or mouse activity of a user, can give rise to a randomized source.

True random numbers are ideal for critical applications such as cryptography due to their unpredictable and realistic random nature. However, they are not useful in computer simulation, where as will be seen below, we need to be able to reproduce a given sequence of random numbers. In addition, despite their several attractive properties, the production and storing of true random numbers is very costly.

An alternative approach to generating random numbers, which is the most popular approach, is to use a mathematical algorithm. Efficient algorithms have been developed that can be easily implemented in a computer program to generate a string of random

numbers. These algorithms produce numbers in a deterministic fashion. That is, given a starting value, known as the *seed*, the same sequence of random numbers can be produced each time as long as the seed remains the same. Despite the deterministic way in which random numbers are created, these numbers appear to be random since they pass a number of statistical tests designed to test various properties of random numbers. In view of this, these random numbers are referred to as *pseudo-random numbers*.

An advantage of generating pseudo random numbers in a deterministic fashion is that they are reproducible, since the same sequence of random numbers is produced each time we run a pseudo-random generator given that we use the same seed. This is helpful when debugging a simulation program, as we typically want to reproduce the same sequence of events in order to verify the accuracy of the simulation.

Pseudo-random numbers and in general random numbers are typically generated on demand. That is, each time a random number is required, the appropriate generator is called which it then returns a random number. Consequently, there is no need to generate a large set of random numbers in advance and store them in an array for future use as in the case of true random numbers.

We note that the term pseudo-random number is typically reserved for random numbers that are uniformly distributed in the space [0,1]. All other random numbers, including those that are uniformly distributed within any space other than [0,1], are referred to as *random variates* or *stochastic variates*. For simplicity, we will refer to pseudo-random numbers as *random numbers*.

In general, an acceptable method for generating random numbers must yield sequences of numbers or bits that are:

a. uniformly distributed
b. statistically independent
c. reproducible, and
d. non-repeating for any desired length.

Historically, the first method for creating random numbers by computer was Von Neuman's mid-square method. His idea was to take the square of the previous random

number and to extract the middle digits. For example, let us assume that we are generating 10-digit numbers and that the previous value was 5772156649. The square of this value is 33317792380594909291 and the next number is 7923805949. The question here that arises is how such a method can give a sequence of random numbers. Well, it does not, but it appears to be!

The mid-square method was relatively slow and statistically unsatisfactory. It was later abandoned in favour of other algorithms. In the following sections, we describe the the *congruential* method, the Tausworthe generators, the lagged Fibonacci generators, and the Mersenne twister.

2.3 The congruential method

This is a very popular method and most of the available computer code for the generation of random numbers use some variation of this method. The advantage of this congruential method is that it is very simple, fast, and it produces pseudo-random numbers that are statistically acceptable for computer simulation.

The congruential method uses the following recursive relationship to generate random numbers.

$$x_{i+1} = ax_i + c \pmod{m}$$

where x_i, a, c and m are all non-negative numbers. Given that the previous random number was x_i, the next random number x_{i+1} can be generated as follows. Multiply x_i by a and then add c. Then, compute the modulus m of the result. That is, divide the result by m and set x_{i+1} equal to the remainder of this division. For example, if $x_0 = 0$, $a = c = 7$, and $m = 10$ then we can obtain the following sequence of numbers: 7, 6, 9, 0, 7, 6, 9, 0,...

The method using the above expression is known as the *mixed* congruential method. A simpler variation of this method is the *multiplicative* congruential method. This method utilizes the relation $x_{i+1}=ax_i \pmod{m}$. Historically, multiplicative congruential generators came before the mixed congruential generators. Below we limit our discussion to mixed congruential generators.

The numbers generated by a congruential method are between 0 and m-1. Uniformly distributed random numbers between 0 and 1 can be obtained by simply dividing the resulting x_i by m.

The number of successively generated pseudo-random numbers after which the sequence starts repeating itself is called the *period*. If the period is equal to m, then the generator is said to have a full period. Theorems from number theory show that the period depends on m. The larger the value of m, the larger is the period. In particular, the following conditions on a, c, and m guarantee a full period:

1. m and c have no common divisor.
2. a = 1 (mod r) if r is a prime factor of m. That is, if r is a prime number (divisible only by itself and 1) that divides m, then it divides a-1.
3. a = 1 (mod 4) if m is a multiple of 4.

It is important to note that one should not use any arbitrary values for a, c and m. Systematic testing of various values for these parameters have led to generators which have a full period and which are statistically satisfactory. A set of such values is: a = 314, 159, 269, c = 453, 806, 245, and m = 2^{32} (for a 32 bit machine).

In order to get a generator started, we further need an initial seed value for x. It will become obvious later on that the seed value does not affect the sequence of the generated random numbers after a small set of numbers has been generated.

The implementation of a pseudo-random number generator involves a multiplication, an addition and a division. The division can be avoided by setting m equal to the size of the computer word. For, if the total numerical value of the expression ax_i+c is less than the word size, then it is in itself the result of the operation $ax_i+c \pmod m$, where m is set equal to the word size. Now, let us assume that the expression ax_i+c gives a number greater than the word size. In this case, when the calculation is performed, an overflow will occur. If the overflow does not cause the execution of the program to be aborted, but it simply causes the significant digits to be lost, then the remaining digits left in the register is the remainder of the division $(ax_i+c)/m$. This is because the lost

significant digits will represent multiples of the value of m, which is the quotient of the above division.

In order to demonstrate the above idea, let us consider a fictitious decimal calculator whose register can accommodate a maximum of 2 digits. Obviously, the largest number that can be held in the register is 99. Now, we set m equal to 100. For a=8, x=2, and c=10, we have that $ax_i + c = 26$, and 26 (mod 100) = 26. However, if x=20, then we have that $ax_i + c = 170$. In this case, the product ax_i (which is equal to 8x20) will cause an overflow to occur. The first significant digit will be lost and thus the register will contain the number 60. If we now add c (which is equal to 10) to the above result we will obtain 70, which is, the remaining of the division 170/100.

2.3.1 General congruential methods

The mixed congruential method described above can be thought of as a special case of a following generator:

$$x_{i+1} = f(x_i, x_{i-1}, ...) \text{ (mod m)},$$

where f(.) is a function of previously generated pseudo-random numbers. A special case of the above general congruential method is the *quadratic* congruential generator. This has the form:

$$x_{i+1}=a_1x_i^2 + a_2x_{i-1}+ c \text{ (mod m)}.$$

The special case of $a_1=a_2=1$, c=0 and m being a power of 2 has been found to be related to the midsquare method. Another special case that has been considered is the *additive* congruential method, which is based on the relation

$$f(x_i, x_{i-1}, ..., x_{i-k}) = a_1x_i + a_2x_{i-1} + ... a_kx_{i-k}.$$

The case of $f(x_i, x_{i-1})=x_i+x_{i-1}$ has received attention.

2.3.2 Composite generators

We can develop composite generators by combining two separate generators (usually congruential generators). By combining separate generators, one hopes to achieve better statistical behavior than either individual generator.

One of the good examples for this type of generator uses the second congruential generator to shuffle the output of the first congruential generator. In particular, the first generator is used to fill a vector of size n with its first k generated random numbers. The second generator is then used to generate a random integer r uniformly distributed over the numbers 1, 2, …, k. The random number stored in the rth position of the vector is returned as the first random number of the composite generator. The first generator then replaces the random number in the rth position with a new random number. The next random number that will be returned by the composite generator, is the one selected by the second generator from the updated vector of random numbers. The procedure repeats itself in this fashion. It has been demonstrated that such a combined generator has good statistical properties, even if two separate generators used are bad.

2.4 Tausworthe generators

Tausworthe generators are additive congruential generators obtained when the modulus m is equal to 2. In particular,

$$x_i = (a_1x_{i-1} + a_2x_{i-2} + ...+ a_nx_{i-n}) \pmod 2$$

where x_i can be either 0 or 1. This type of generator produces a stream of bits $\{b_i\}$. In view of this, it suffices to assume that the coefficients a_i are also binary. Thus, x_i is obtained from the above expression by adding some of preceding bits and then carrying out a modulo 2 operation. This is equivalent to the exclusive OR operation, notated as $\oplus$ and defined by the following truth table.

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

A⊕B is true (i.e. equal to 1), when either A is true and B false, or A is false and B true.

The generated bits can be put together sequentially to form an *l*-bit binary integer between 0 and 2*l*-1. Several bit selection techniques have been suggested in the literature.

In the composite generator scheme discussed earlier on, one of the generators (but not both) could be a Tausworthe generator.

Tausworthe generators are independent of the computer used and its word size and have very long cycles. However, they are too slow since they only produce bits. A fast variation of this generator is the trinomial-based Tausworthe generator. Two or more such generators have to be combined in order to obtain a statistically good output.

2.5 The lagged Fibonacci generators

The *lagged Fibonacci generators* (LFG) are an important improvement over the congruential generators, and they are widely used in simulation. They are based on the well-known Fibonacci sequence, an additive recurrence relation, whereby each element is computed using the two previously computed elements, as shown below:

$$x_n = x_{n-1} + x_{n-2}$$

where x_0=0 and x_1=1. The beginning of the Fibonacci sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21. Based on this recurrence relation, the general form of LFG can be expressed as follows:

$$x_n = x_{n-j} \; O \; x_{n-k} \pmod m$$

where 0<j<k, and appropriate initial conditions have been made. In this generator, the next element is determined by combining two previously calculated elements that lag behind the current element utilizing an algebraic operation O. This operation *O* can be an addition, or a subtraction, or a multiplication as well as it can be a binary operation XOR. If *O* is the addition operation, then this LFG is called the *additive LFG* (ALFG). Likewise, if *O* is the multiplication operation, then it is called the *multiplicative LFG* (MLFG).

The additive LFG is the most frequently used generator. In this case, the next element is calculated as follows:

$$x_n = x_{n-j} + x_{n-k} \pmod{M}$$

where 0<j<k. As can be seen, it is very easy to implement and also it is quite fast. A very long period, equal to m^k-1, can be obtained if m is a prime number. However, using a prime number may not be very fast. Thus, typically m is set to 2^{32} or 2^{64}. In this case the maximum period of the additive LFG is $(2^k-1)x2^{m-1}$.

The multiplicative LFG is as follows: $x_n = x_{n-j} * x_{n-k} \pmod{m}$ where m is set to 2^{32} or 2^{64} and 0<j<k. The maximum period is $(2^k-1)*2^{m-3}$.

In general, LFGs generate a sequence of random numbers with very good statistical properties, and they are nearly as efficient as the linear congruential generators. Their execution can also be parallelized. However, LFGs are highly sensitive on the seed. That is, the statistical properties of an output sequence of random numbers varies from seed to seed. Determining a good seed for LFGs is a difficult task.

Further references on this generator can be found in: "A fast, high quality,and reproducible parallel lagged-fibonacci pseudorandom number generator " by M. Mascagni, S. Cuccaro, D. Pryor & M. Robinson, Supercomputing Research Center Technical Report, SRC-TR-94-115, 1994

2.6 The Mersenne twister

The *Mersenne twister* (MT) is an important pseudo-random number generator with

superior performance. Its maximum period is 2^{19937}-1**,** which is much higher than many other pseudo-random number generators, and its output has very good statistical properties. The MT generates a sequence of bits, which is as large as the period of the generator after which it begins to repeat itself. This bit sequence is typically grouped into 32-bit blocks (i.e., blocks equal to the computer word). The blocks are considered to be random.

The following is the main recurrence relation for the generation of random sequence of bits:

$$x_{k+n} = x_{k+m} \oplus \left(x_k^u \mid x_{k+1}^k \right) A, \ \text{k} \geq 0$$

We assume that each block, represented by *x,* has a size of w bits. The meaning of the parameters used in the above equation is as follows:

- x_k^u : The upper w-r bits of x_k,where 0≤r≤w.
- x_{k+1}^l : The lower *r* bit of x_{k+l}.
- $\oplus$: exclusive OR
- |: It indicates the concatenation (i.e., joining) of two bit strings.
- *n:* Degree of recurrence relation
- *m:* Integer in the range of 1≤m≤n.
- *A:* A constant w x w matrix defined as below so that the multiplication operation in the above recurrence can be performed extremely fast.

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 & 0 \\ 0 & 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ a_{w-1} & a_{w-2} & \cdots & \cdots & \cdots & a_0 \end{pmatrix}$$

The recurrence relation is initialized by providing seeds for the first d blocks, i.e., $x_0, x_1, \ldots, x_{n-1}$. The multiplication operation xA can be done very fast as follows:

$$xA = \begin{cases} shiftright(x) \text{ if } x_0 = 0 \\ shiftright(x) \oplus a \text{ if } x_0 = 1 \end{cases}$$

where $a = \{a_{w-1}, a_{w-2}, \cdots a_0\}$ *and* $x = \{x_{w-1}, x_{w-2}, \cdots x_0\}$.

At the last state of the algorithm, in order to increase the statistical properties of generator's output, each generated block is multiplied from the right with a special wxw *invertible tempering matrix T*. This multiplication is performed in a similar manner as the multiplication with matrix A above and it involves only bitwise operations, as follows.

$$y = x \oplus (x >> u)$$
$$y = y \oplus ((y << s) \wedge b)$$
$$y = y \oplus ((y << t) \wedge c)$$
$$z = y \oplus (y >> l)$$

where

- b and c are block size binary bitmasks (vector parameters).
- l, s, t and u are pre-determined integer constants.
- (x >> u) indicates a shiftright operation by u times for variable x.
- (x << u) indicates a shiftleft operation by u times for variable x.

Below we present a description of the MT algorithm following the paper: "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator", M. Matsumoto and T. Nishimura, ACM Trans. Model. Comput. Simul. 8, 1, 3-30 (1998).

1. *Bit Mask Initializations:*

$u \leftarrow \underbrace{1\cdots1}_{w-r}\underbrace{0\cdots0}_{r}$: (The bitmask for upper *w-r* bits)

$ll \leftarrow \underbrace{0\cdots0}_{w-r}\underbrace{1\cdots1}_{r}$: (The bitmask for lower *r* bits)

$a \leftarrow \left[a_{w-1}, a_{w-2}, \cdots, a_1, a_0\right]$: (the last row of the matrix A)

2. *Seed Initialization:*

$i \leftarrow 0$

$(x[0], x[1], \cdots, x[n-1]) \leftarrow$"any non - zero initial values"

3. *Compute* $(x_k^u \mid x_{k+1}^l)$

$y \leftarrow (x[i] \ \wedge \ u) \ \vee \ (x[(i+1) \bmod n] \ \wedge \ ll)$

4 *Multiplication with A:*

$x[i] \leftarrow x[(i+m) \bmod n] \oplus (y >> 1)$

$$\oplus \begin{cases} 0 \text{ if the least significant bit of y} = 0 \\ a \text{ if the least significant bit of y} = 1 \end{cases}$$

5. *Multiplication operation x[i]*T*

$y \leftarrow x[i]$

$y \leftarrow y \oplus (y >> u)$

$y \leftarrow y \oplus ((y << s) \wedge b)$

$y \leftarrow y \oplus ((y << t) \wedge c)$

$y \leftarrow y \oplus (y >> l)$

Output y

6. $i = i+1 \bmod n$ and goto step 3.

2.7 Statistical tests for pseudo-random number generators

It is a good practice to check statistically the output of a pseudo-random number generator prior to using it. Not all random-number generators available through programming languages and other software packages have a good statistical behavior.

In this section, we describe the following five statistical tests:

1. frequency test,
2. serial test,
3. autocorrelation test,
4. runs test, and
5. chi-square test for goodness of fit.

Many useful statistical tests can be found in: “A statistical testing for pseudorandom number generators for cryptographic applications”, NIST special publication 800-22.

The first three statistical tests described below, namely, the frequency test, the serial test, and the autocorrelation test, test the randomness of a sequence of bits. The last two statistical tests, namely, the runs test and the chi-square test for goodness of fit, check the randomness of pseudo-random numbers in [0,1]. The bitwise statistical tests can be used on pseudo-random number generators that generate real numbers, by simply concatenating a very large number of random numbers, each represented by a binary string, into a sequence of bits.

Statistical testing involves testing whether a particular assumption, known in statistics as a *hypothesis*, is correct. Before we proceed to describe the statistical tests, we review the basic principles of hypothesis testing in statistics. (More information on this subject can be found in any introductory book on statistics.)

2.7.1 Hypothesis testing

Hypothesis testing is used in statistics to decide whether a particular assumption is correct or not. This assumption is called a *hypothesis*, and it typically is an assertion about a distribution of one or more random variables, or about some measure of a distribution, such as the mean and the variance. The tested statistical hypothesis is called the *null hypothesis,* and it is denoted as H_0. An *alternative hypothesis,* denoted as H_a, is also formulated. As an example, let us formulate the assertion that 30% of the cars are red. This assertion is our null hypothesis. In this case, the alternative hypothesis is that this is not true, that is, it is not true that 30% of the cars are red. In the case of the testing a pseudo-random number generator, the null hypothesis is that “the produced sequence from the generator is random”. The alternative hypothesis is that “the produced sequence is not random”.

To test a hypothesis, we first collect data and then based on the data we carry out a *statistical hypothesis test*. For instance, in order to test the assertion that 30% of the cars are red, we first have to collect a representative sample of the colour of cars, red or not

red, and then based on the sample we calculate p_{red}, the percent of cars that were red. Due to sampling variation, p_{red} will never be exactly equal to 0.30, even if our assumption is correct. The statistical test will help us determine whether the difference between p_{red} and 0.30 is due to randomness in the obtained sample or because our assumption that "30% of the cars are red" is incorrect.

In the case of testing the randomness of a generator, we will need an output sequence of random numbers. Based on sampled data, a statistical hypothesis test is used to decide whether to accept or reject the null hypothesis. The result of the test (accept or reject) is not deterministic but rather probabilistic. That is, with some probability we will accept or reject the null hypothesis.

Real situation	**Decision**	
	H_o is not rejected	*H_0 is rejected*
H_o is true	Valid	Type I error
H_o is not true	Type II Error	Valid

Table 2.1: Type I and type II errors

There are two errors associated with hypothesis testing, namely, type I error and type II error. A type I error occurs when we reject the null assumption, whereas in fact it is correct (which of course we do not know). A type II error occurs when we accept the null hypothesis when in fact it is not correct. Since we do not know the truth, we do not know whether we have committed a type I or a type II error. The type I error is commonly known as a *false negative*, and the type II error is known as a *false positive*. Table 2.1 summarizes the type I and type II errors.

The probability of rejecting the null hypothesis when it is true, that is, the probability that a type I error occurs, is denoted by α and it is known as *the level of significance*. Typically, we set α to 0.01 or 0.05 in practical applications.

2.7.2 Frequency test (Monobit test)

The frequency test is one of the most fundamental tests for pseudo-random number

generators. Thus, if a pseudo-random number generators fails this test, then it is highly likely that it will also fails more sophisticated tests.

In a true random sequence, the number of 1's and 0's should be about the same. This test checks whether this is correct or not. In order to achieve this, the test makes use of the *complementary error function (erfc)*. The code for this function can be found in a statistical package like Matlab, or in a programming language like Java.

The following steps are involved:

1. Generate m pseudo-random numbers and concatenate them into a string of bits. Let the length of this string be n. The 0's are first converted into –1's and then the obtained set of -1 and 1 values are all added up. Let
$$S_n = X_1 + X_2 + \ldots X_n$$
2. Compute the test statistic $S_{obs} = \frac{|S_n|}{\sqrt{n}}$
3. Compute the $P\text{-}value = erfc\left(\frac{S_{obs}}{\sqrt{2}}\right)$

If $P\text{-}value < 0.01$ then we decide to reject the null hypothesis. Thus the sequence is non-random else the sequence is accepted as a random sequence. It is recommended that a sequence of a few thousand bits is tested.

Note that the *P-value* is small, i.e., less than 0.01, if $|S_n|$ or $|S_{obs}|$ is large. A large positive value of S_n is indicative of too many ones, and a large negative value of S_n is indicative of too many zeros.

As an example, let us consider the bit string: 1011010101. Then, S_n = 1-1 +1+1-1+1-1+1-1+1 = 2 and S_{obs}=0.6324. For a level of significance $\alpha = 0.01$, we obtain that *P-value = erfc*(0.6324*) = 0.5271* > 0.01. Thus, the sequence is accepted as being random.

For further details see: "A statistical testing for pseudorandom number generators for cryptographic applications", NIST special publication 800-22.

2.7.3 Serial test

There are 2^k different ways of combining k bits. Each of these combinations has the same chance of occurring, if the sequence of the k bits is random. The serial test determines whether the number of times each of these combinations occurs is uniformly distributed. If k=1, then this test becomes equivalent to the frequency test described above in section 2.6.2.

Let e denote a sequence of n bits created by a pseudo-random number generator. The minimum recommended value for n is 100. The serial statistical test checks the randomness of overlapping blocks of *k, k-1, k-2* bits found in e, where $k < \lfloor \log_2 n \rfloor - 2$. The following steps are involved:

1. Augment e by appending its first k-1 bits to the end of the bit string e, and let e_1' be the resulting bit string. Likewise, augment e by appending its first k-2 bits to the end of the bit string e, and let e_2' be the resulting bit string. Finally, augment e by appending its first *k-2* bits to the end of the bit string e, and let e_3' be the resulting bit string.
2. Compute the frequency of occurrence of each *k, k-1* and *k-2* bit overlapping combination using the sequence e_1', e_2', and e_3' respectively. Let f_i, f_i', and f_i'' represent the frequency of occurrence of the ith *k, k-1,* and *k-2* overlapping bit combination respectively.
3. Compute the following statistics:

$$S_k^2 = \frac{2^k}{n} \sum_i f_i^2 - n$$

$$S_{k-1}^2 = \frac{2^{k-1}}{n} \sum_i f_i'^2 - n$$

$$S_{k-2}^2 = \frac{2^{k-2}}{n} \sum_i f_i''^2 - n$$

4. Compute the following difference statistics:

$$\Delta S_k^2 = S_k^2 - S_{k-1}^2$$

$$\Delta^2 S_k^2 = S_k^2 - 2S_{k-1}^2 + S_{k-2}^2$$

5. Compute the P-values:

$$P-value_1 = IncompleteGamma(2^{k-2}, \Delta S_k^2/2)$$
$$P-value_2 = IncompleteGamma(2^{k-3}, \Delta^2 S_k^2/2)$$

6. If $P\text{-}value_1$ or $P\text{-}value_2 < 0.01$ then we reject the null hypothesis that the sequence is random, else we accept the null hypothesis that the sequence is random. (0.01 is the level of significance.) The code for the IncompletGamma (.) function can be found in software packages.

As in the frequency test, if the overlapping bit combinations are not uniformly distributed, ΔS_k^2 and $\Delta^2 S_k^2$ become large that causes the P-values to be small.

As an example, let assume the output sequence e = 0011011101, with $n=10$, and $k=3$.

1. Append the k-1=2 bits from the beginning of the sequence e to the end of e, obtaining $e_1' = 001101110100$. We repeat the process by appending the first $k\text{-}2=1$ bits of e to the end of e, obtaining $e_2' = 00110111010$. Do not need to repeat the process for k-3=0 since $e_3' = e = 0011011101$.
2. Calculate the frequency of occurrence of each 3-bit, 2-bit and 1-bit overlapping combination.

3 For the 3-bit blocks we use e_1'. The bit combinations are: c_1=000, c_2=001, c_3=010, c_4=011, c_5=100, c_6=101, c_7=110, and c_8=111. Their corresponding frequencies of occurrence are: f_1=0, f_2=1, f_3=1, f_4=2, f_5= 1, f_6=2, f_7=2 and $f_8 = 0$.

4 For the 2-bit blocks we use e_2'. The 2 bit combinations are: $c_1' = 00$, $c_2' = 01$, $c_3' = 10$, and $c_4' = 11$. Their corresponding frequency of occurrence is: $f_1' = 1$, $f_2' = 3$, $f_3' = 3$ and $f_4' = 3$.

5 For the 1-bit blocks we use e. The 1-bit combinations are $c_1'' = 0$ and $c_2'' = 1$ and the frequency of occurrence is $f_1'' = 4$ and $f_2'' = 6$.

3. Compute the following statistics and differences:

$$S_3^2 = \frac{2^3}{10}(0+1+1+4+1+4+4+1) - 10 = 2.8$$
$$S_2^2 = \frac{2^2}{10}(1+9+9+9) - 10 = 1.2$$
$$S_1^2 = \frac{2}{10}(16+36) - 10 = 0.4$$
$$\Delta S_k^2 = \Delta S_3^2 - \Delta S_2^2 = 2.8 - 1.2 = 1.6$$
$$\Delta^2 S_k^2 = \Delta S_3^2 - 2\Delta S_2^2 + \Delta S_1^2 = 0.8$$

4. Compute P-values:

$$P - value_1 = IncompleteGamma(2,0.8) = 0.9057$$
$$P - value_2 = IncompleteGamma(1,0.4) = 0.8805$$

5. Since both P-value$_1$ and P-value$_2$ are greater than 0.01, this sequence passes the serial test.

For further details see: "A statistical testing for pseudorandom number generators for cryptographic applications", NIST special publication 800-22

2.7.4 Autocorrelation test

Let e denote a sequence of n bits created by a pseudo-random number generator. The intuition behind the autocorrelation test is that if the sequence of bits in e is random, then it will be different from another bit string obtained by shifting the bits of e by d positions. For instance, let e = 1001101. Then 0011011 is a bit string obtained by shifting e by 1 position, 0110110 by 2 positions, 1101100 by 3 positions, etc. We compare shifted versions of e to find how many bits are different, and this information is then used in a statistic for the hypothesis testing.

The following steps are involved:

1. Let e_i be the bit sequence obtained by shifting e by i positions. Given an integer value d, where $1 \le d \le n/2$, we find the number of bits that are different e_i and e_{i+d}, where $0<i<n-d-1$, as follows:

$$A(d) = \sum_{i=0}^{n-d-1} e_i \oplus e_{i+d}$$

2. We then compute the following statistic:

$$S = \frac{2\left(A(d) - \frac{n-d}{2}\right)}{\sqrt{n-d}}$$

3. S follows approximately the standardized normal distribution $N(0,1)$ if $n\text{-}d \geq 10$. At 5% level of significance, we will accept the null hypothesis that the bit string e is random if $S \leq 1.96$. (Values for various level of significance can be readily computed using a function available in software packages. They can also be found in a table for the standardized Normal distribution in any introductory statistics book.)

As an example, let us consider a bit string e which is made up by combining the following bit sequence e' = (11100 01100 010000 10100 11101 11100 10010 01001) four times. Then, total length of e is n=160. Obviously, e is not very random! For d =8, we have that $A(8)$=100, and consequently S=3.8933. For a significance level of 0.05, S>1.96, and therefore we reject the hypothesis that e is random.

Further details can be found in: “Handbook of Applied Cryptography” by A. J. Menezes, P. C. van Oorschot and S. A. Vanstone, CRC Press, 1996.

2.7.5 Runs test

The runs test can be used to test the assumption that the pseudo-random numbers are independent of each other. We start with a sequence of pseudo-random numbers in [0,1]. We then look for unbroken subsequences of numbers, where the numbers within each subsequence are monotonically increasing. Such a subsequence is called a *run up*, and it may be as long as one number.

For example, let us consider the sequence: 0.8, 0.7, 0.75, 0.55, 0.6, 0.7, 0.3, 0.4, 0.5. Starting from the beginning of this sequence, i.e., from the left, we find a run up of

length 1, i.e. 0.8, then a run up of length 2, i.e. 0.7, 0.75, followed by two successive run ups of length 3, i.e. 0.55, 0.6, 0.7, and 0.3, 0.4, 0.5.

In general, let r_i be the number of run ups of length i. (In the above example we have r_1=1, r_2=1, r_3=2.) All run-ups with a length $i \geq 6$ are grouped together into a single run-up. The r_i values calculated for a particular sequence are then used to calculate the following statistic:

$$R = \frac{1}{n} \sum_{1 \leq i, j \leq 6} (r_i - nb_i)(r_j - nb_j) a_{ij}, \quad 1 \leq i \leq 6, \; 1 \leq j \leq 6,$$

where n is the sample size and b_i, i=1,...6, and a_{ij} are known coefficients. The a_{ij} coefficient is obtained as the (i,j)th element of the matrix

$$\begin{bmatrix} 4529.4 & 9044.9 & 13568 & 18091 & 22615 & 27892 \\ 9044.9 & 18097 & 27139 & 36187 & 45234 & 55789 \\ 13568 & 27139 & 40721 & 54281 & 67852 & 83685 \\ 18091 & 36187 & 54281 & 72414 & 90470 & 111580 \\ 22615 & 45234 & 67852 & 90470 & 113262 & 139476 \\ 27892 & 55789 & 83685 & 111580 & 139476 & 172860 \end{bmatrix},$$

and the b_i coefficient is obtained as the i^{th} element of the vector

$$(b_1,...,b_6) = \left(\frac{1}{6}, \frac{5}{24}, \frac{11}{120}, \frac{19}{720}, \frac{29}{5040}, \frac{1}{840} \right).$$

For $n \geq 4000$, R has a chi-square distribution with 6 degrees of freedom (d.f) under the null hypothesis that the random numbers are independent and identically distributed (i.i.d). (See below in section 2.7.6, to see how to do a hypothesis testing with the chi-square distribution).

The runs test can be also applied to a sequence of 0's and 1's. In this case, a run is an uninterrupted sequence of 0's or 1's.

For further information see "The Art of Programming – Volume 2" by Knuth, Addison-Wesley).

2.7.6 Chi-square test for goodness of fit

This test checks whether a sequence of pseudo-random numbers in [0,1] are uniformity distributed. The chi-square test, in general, can be used to check whether an empirical distribution follows a specific theoretical distribution. In our case, we are concerned about testing whether the numbers produced by a generator are uniformly distributed.

Let us consider a sequence of pseudo-random numbers between 0 and 1. We divide the interval [0,1] into k subintervals of equal length, where $k > 100$. Let f_i be the number of pseudo-random numbers that fall within the i^{th} subinterval (make sure that enough random numbers are generated so that $f_i > 5$). The f_i values are called the *observed* values. Now, if these generated random numbers are truly uniformly distributed, then the mean number of random numbers that fall within each subinterval is n/k, where n is the sample size. This value is called the *theoretical* value. The chi-square test measures whether the difference between the observed and the theoretical values is due to random fluctuations or due to the fact that the empirical distribution does not follow the specific theoretical distribution. For the case where the theoretical distribution is the uniform distribution, the chi-square statistic is given by the expression

$$\chi^2 = \frac{k}{n} \sum_{i=1}^{k} (f_i - \frac{n}{k})^2 ,$$

and it has k-1 degrees of freedom. The null hypothesis is that the generated random numbers are uniformly distributed in [0,1]. This hypothesis is rejected if the computed value of χ^2 is greater than the one obtained from the chi-square tables for k-1 degrees of freedom and α level of significance.

The chi-square tables can be found in any introductory statistics book, and of course, the function to generate χ^2 values can be found in software packages.

Problems

1. Consider the multiplicative congruential method for generating random digits.

Assuming that m=10, determine the length of the cycle for each set of values of a and x_0 given below.

a. a = 2, x_0 =1, 3, 5.

b. a = 3, x_0 = 1, 2, 5.

Computer Assignments

1. Use the statistical tests described in section 2.7, to test a random number generator available on your computer.
2. This is a more involved exercise! Implement the random number generators described in this chapter, and then test them using the statistical tests described in section 2.7. Measure the execution time for each generator for the same output size. Present your results in a table that gives the execution speed of each generator and whether it has failed or passed each statistical test. Discuss your results. What is your favorite generator for your simulation assignments?

CHAPTER 3:

GENERATING STOCHASTIC VARIATES

3.1 Introduction

In the previous Chapter we examined techniques for generating random numbers. In this Chapter, we discuss techniques for generating random numbers with a specific distribution. Random numbers following a specific distribution are called *random variate*s or *stochastic variates*. Pseudo-random numbers which are uniformly distributed are normally referred to as *random numbers*. Random numbers play a major role in the generation of stochastic variates.

There are many techniques for generating random variates. The *inverse transformation* method is one of the most commonly used techniques. This is discussed below. Sections 3.3 and 3.4 give methods for generating stochastic variates from known continuous and discrete theoretical distributions. Section 3.5 discusses methods for obtaining stochastic variates from empirical distributions. Section 3.6 describes an alternative method for stochastic variates generation known as the *rejection method*.

3.2 The inverse transformation method

This method is applicable only to cases where the cumulative density function can be inversed analytically. Assume that we wish to generate stochastic variates from a probability density function (pdf) f(x). Let F(x) be its cumulative density function. We note that F(x) is defined in the region [0,1]. We explore this property of the cumulative density function to obtain the following simple stochastic variates generator.

We first generate a random number r which we set equal to F(x). That is, $F(x) = r$. The quantity x is then obtained by inverting F. That is, $x = F^{-1}(r)$, where $F^{-1}(r)$ indicates the inverse transformation of F.

As an example, let us assume that we want to generate random variates with probability density function

$$f(x) = 2x, \quad 0 \leq x \leq 1.$$

A graphical representation of this probability density function is given in figure 3.1a. We first calculate the cumulative density function F(x). We have

$$F(x) = \int_0^x 2t\,dt$$
$$= x^2, \qquad 0 \leq x \leq 1.$$

Let r be a random number. We have

$$r = x^2,$$

or

$$x = \sqrt{r}\,.$$

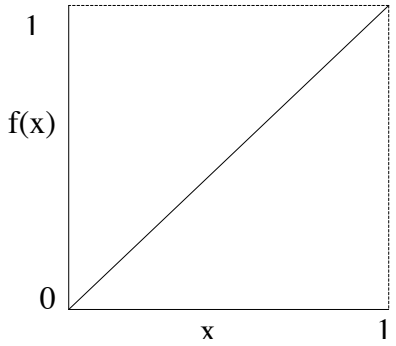


Figure 3.1a: pdf f(x)

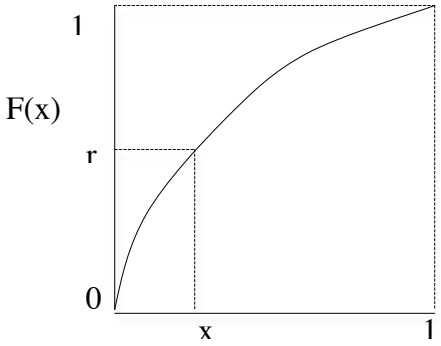


Figure 3.1b: Inversion of F(x)

This inversion is shown graphically in figure 3.1b.

In sections 3 and 4 we employ the inverse transformation method to generate random variates from various well-known continuous and discrete probability distributions.

3.3 Sampling from continuous-time probability distributions

In this section, we use the inverse transformation method to generate variates from a uniform distribution, an exponential distribution, and an Erlang distribution. We also describe two techniques for generating variates from the normal distribution.

3.3.1 Sampling from a uniform distribution

The probability density function of the uniform distribution is defined as follows:

$$f(x) = \begin{cases} \frac{1}{b\text{-}a} & a < x < b \\ 0 & \text{otherwise} \end{cases},$$

and it is shown graphically in figure 3.2.

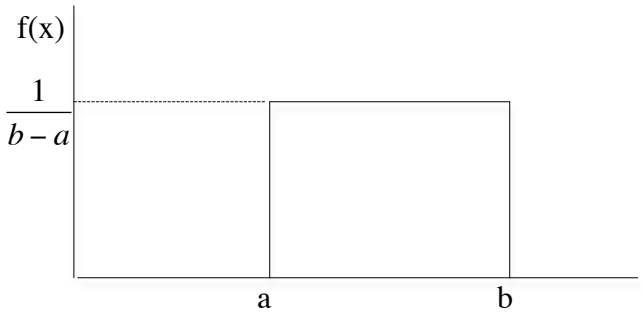


Figure 3.2: The uniform distribution.

The cumulative density function is:

$$F(x) = \int_{a}^{x} \frac{1}{b\text{-}a}\, dt = \frac{x-a}{b-a}.$$

The expectation and variance are given by the following expressions:

$$E(X) = \int_{a}^{b} f(x)x dx = \frac{1}{b-a}\int_{a}^{b} x dx = \frac{b+a}{2}$$

$$Var(X) = \int_{a}^{b} (x - E(X))^2 F(x) dx = \frac{(b-a)^2}{12} .$$

The inverse transformation method for generating random variates is as follows.

$$r = F(x) = \frac{x-a}{b-a}$$

or

$$x = a + (b - a)r.$$

3.3.2 Sampling from an exponential distribution

The probability density function of the exponential distribution is defined as follows:

$$f(x) = ae^{-ax} , a > 0, x \geq 0.$$

The cumulative density function is:

$$F(x) = \int_{0}^{x} f(t)dt = \int_{0}^{x} ae^{-at}dt = 1 - e^{-ax}.$$

The expectation and variance are given by the following expressions:

$$E(X) = \int_0^\infty aet^{-at}dt\,dt = \frac{1}{a}$$

$$Var(X) = \int_0^\infty (t - E(X))^2 e^{-at}tdt\,dt = \frac{1}{a^2} .$$

The inverse transformation method for generating random variates is as follows:

$$r = F(x) = 1 - e^{-ax}$$

or

$$1 - r = e^{-ax}$$

or

$$x = -\frac{1}{a} \log(1-r) = - E(x)\log(l-r).$$

Since 1-F(x) is uniformly distributed in [0,1], we can use the following short-cut procedure

$$r = e^{-ax},$$

and therefore,

$$x = -\frac{1}{a} \log r.$$

3.3.3 Sampling from an Erlang distribution

In many occasions an exponential distribution may not represent a real life situation. For example, the execution time of a computer program, or the time it takes to manufacture an item, may not be exponentially distributed. It can be seen, however, as a number of

exponentially distributed services which take place successively. If the mean of each of these individual services is the same, then the total service time follows an *Erlang* distribution, as shown in figure 3.3.

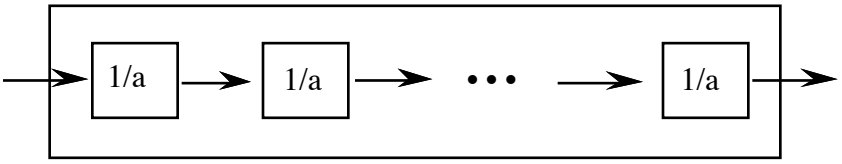


Figure 3.3 : The Erlang distribution.

The Erlang distribution is the convolution of k exponential distributions having the same mean 1/a. An Erlang distribution consisting of k exponential distributions is referred to as E_k. The expected value and the variance of a random variable X that follows the Erlang distribution are:

$$E(X) = \frac{k}{a}$$

and

$$Var(X) = \frac{k}{a^2} \ .$$

Erlang variates may be generated by simply reproducing the random process on which the Erlang distribution is based. This can be accomplished by taking the sum of k exponential variates, $x_1, x_2, ..., x_k$ with identical mean 1/a. We have

$$x = \sum_{i=1}^{k} x_i$$

$$= -\frac{1}{a} \sum_{i=1}^{k} \log r_i = \frac{-1}{a} \left(\log \sum_{i=1}^{k} r_i \right) .$$

3.3.4 Sampling from a normal distribution

A random variable X with probability density function

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \; e^{-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}} \quad , \qquad -\infty < x < +\infty,$$

where σ is positive, is said to have a normal distribution with parameters μ and σ. The expectation and variance of X are μ and σ^2 respectively. If $\mu=0$ and $\sigma=1$, then the normal distribution is known as the *standard* normal distribution and its probability density function is

$$f(x) = \frac{1}{\sqrt{2\pi}} \; e^{-\frac{1}{2}x^2} \quad , \quad -\infty < x < +\infty \; .$$

If a random variable X follows a normal distribution with mean μ and variance σ^2, then the random variable Z defined as follows

$$Z = \frac{X - \mu}{\sigma}$$

follows the standard normal distribution.

In order to generate variates from a normal distribution with parameters μ and σ, we employ the central limit theorem. (This approach is named after this particular theorem.) The central limit theorem briefly states that if $x_1, x_2, \ldots, x_n$ are n independent random variates, each having the same probability distribution with $E(X_i)=\mu$ and $Var(X_i)=\sigma^2$, then the sum $\Sigma X_i = X_1+X_2+\ldots+X_n$ approaches a normal distribution as n becomes large. The mean and variance of this normal distribution are:

$$E(\Sigma X_i) = n\mu$$

$$\text{Var}\,(\Sigma X_i) = n\sigma^2.$$

The procedure for generating normal variates requires k random numbers $r_1, r_2, \ldots, r_k$. Since each r_i is a uniformly distributed random number over the interval [0 ,1], we have that

$$E(r_i) = \frac{a+b}{2} = \frac{1}{2}$$

$$\text{Var}(r_i) = \frac{(b-a)^2}{12} = \frac{1}{12}\ .$$

Using the Central Limit theorem, we have that the sum Σr_i of these k random numbers approaches the normal distribution. That is

$$\Sigma r_i \sim N\left(\frac{k}{2}, \frac{k}{\sqrt{12}}\right)\ ,$$

or

$$\frac{\Sigma r_i - k/2}{k/\sqrt{12}} \sim N(0, 1). \qquad (3.1)$$

Now, let us consider the normal distribution with parameters μ and σ from which we want to generate normal variates. Let x be such a normal variate. Then

$$\frac{x - \mu}{\sigma} \sim N(0,1). \qquad (3.2)$$

Equating (3.1) and (3.2) gives

$$\frac{x - \mu}{\sigma} = \frac{\Sigma r_i - k/2}{k/\sqrt{12}} \quad ,$$

or

$$x = \sigma\sqrt{\frac{12}{k}}\left(\Sigma r_i - \frac{k}{2}\right) + \mu \, .$$

This equation provides us with a simple formula for generating normal variates with a mean μ and standard deviation σ. The value of k has to be very large, since the larger it is the better the accuracy. Usually, one has to balance accuracy against efficiency. The smallest value recommended is k=10. (In fact, one can observe that k=12 has computational advantages).

An alternative approach to generating normal variates (known as the direct approach) is the following. Let r_1 and r_2 be two uniformly distributed independent random numbers. Then

$$x_1 = (-2 \log_e r_1)^{\frac{1}{2}} \cos 2\pi r_2$$

$$x_2 = (-2 \log_e r_1)^{\frac{1}{2}} \sin 2\pi r_2$$

are two random variates from the standard normal distribution. This method produces exact results and the speed of calculations compares well with the Central Limit approach subject to the efficiency of the special function subroutines.

3.4 Sampling from discrete-time probability distributions

In this section, we use the inverse transformation method to generate variates from a geometric distribution. Also, we describe a technique for sampling from a binomial distribution, and a technique for sampling from a Poisson distribution.

3.4.1 Sampling from a geometric distribution

Consider a sequence of independent trials, where the outcome of each trial is either a failure or a success. Let p and q be the probability of a success and failure respectively. We have that p+q=1. The random variable that gives the number of successive failures that occur before a success occurs follows the geometric distribution. The probability density function of the geometric distribution is

$$p(n) = pq^n, \quad n = 0,1,2, \ldots,$$

and its cumulative probability density function is

$$F(n) = \sum_{s=0}^{n} pq^s \quad , n = 0, 1, 2, \ldots$$

The expectation and the variance of a random variable following the geometric distribution are:

$$E(X) = \frac{p}{q}$$

$$Var(X) = \frac{p}{q^2} \ .$$

The generation of geometric variates using the inverse transformation method can be accomplished as follows.

$$F(n) \quad = \sum_{s=0}^{n} pq^s$$

$$= p \sum_{s=0}^{n} q^s$$

$$= p \frac{1-q^{n+1}}{1-q}.$$

Since p = 1 - q, we have that F(n) = $1-q^{n+1}$. From this expression we obtain that 1-F(n) = q^{n+1}. We observe that 1-F(n) varies between 0 and 1. Therefore, let r be a random number, then we have

$$r = q^{n+1}$$

or

$$\log r = (n+1) \log q$$

or

$$n = \frac{\log r}{\log q} - 1.$$

Alternatively, since (1-F(n))/q=q^n, and (1-F(n))/q varies between 0 and 1, we have

$$r = q^n$$

or

$$n = \frac{\log r}{\log q} .$$

3.4.2 Sampling from a binomial distribution

Consider a sequence of independent trials (Bernoulli trials). Let p be the probability of success and q=1-p the probability of a failure. Let X be a random variable indicating the

number of successes in n trials. Then, this random variable follows the Binomial distribution. The probability density function of X is

$$p(k) = \binom{n}{k} p^k q^{n-k}, \; k = 0, 1, 2, \ldots$$

The expectation and variance of the binomial distribution are:

$$E(X) = np$$

$$Var(X) = npq.$$

We can generate variates from a binomial distribution with a given p and n as follows. We generate n random numbers, after setting a variable k_0 equal to zero. For each random number r_i, i=1, 2, ..., n, a check is made, and the variable k_i is incremented as follows:

$$k_i = \begin{cases} k_{i-1} + 1 & \text{if } r_i < p \\ k_{i-1} & \text{if } r_i > p \end{cases}$$

The final quantity k_n is the binomial variate. This method for generating variates is known as the *rejection* method. This method is discussed in detail below in section 6.

3.4.3 Sampling from a Poisson distribution

The Poisson distribution models the occurrence of a particular event over a time period. Let λ be the average number of occurrences during a unit time period. Then, the number of occurrence x during a unit period has the following probability density function

$$p(n) = e^{-\lambda}(\lambda^n/n!), \; n = 0, 1, 2, \ldots$$

It can be demonstrated that the time elapsing between two successive occurrences of the event is exponentially distributed with mean $1/\lambda$, i.e., $f(t) = \lambda e^{-\lambda t}$. One method for generating Poisson variates involves the generation of exponentially distributed time intervals t_1, t_2, t_3,... with an expected value equal to $1/\lambda$. These intervals are accumulated until they exceed 1, the unit time period. That is,

$$\sum_{i=1}^{n} t_i < 1 < \sum_{i=1}^{n+1} t_i .$$

The stochastic variate n is simply the number of events occurred during a unit time period. Now, since $t_i = -\frac{1}{\lambda} \log r_i$, n can be obtained by simply summing up random numbers until the sum for n+1 exceeds the quantity $e^{-\lambda}$. That is, n is given by the following expression:

$$\sum_{i=0}^{n} r_i > e^{-\lambda} > \sum_{i=0}^{n+1} r_i .$$

3.5 Sampling from an empirical probability distribution

Quite often an empirical probability distribution may not be approximated satisfactorily by one of the well-known theoretical distributions. In such a case, one is obliged to generate variates which follow this particular empirical probability distribution. In this section, we show how one can sample from a discrete or a continuous empirical distribution.

3.5.1 Sampling from a discrete-time probability distribution

Let X be a discrete random variable, and let $p(X = i) = p_i$, where p_i is calculated from empirical data. Let $p(X \leq i) = P_i$ be the cumulative probability. Random variates from this

probability distribution can be easily generated as follows. Now let r be a random number. Let us assume that r falls between P_2 and P_3 (see figure 3.4). Then, the random variate x is equal to 3. In general, if $P_{i-1}<r<P_i$ then x=i. This method is based on the fact that $p_i=P_i-P_{i-1}$ and that since r is a random number, it will fall in the interval (P_i, P_{i-1}) p_i% of the time.

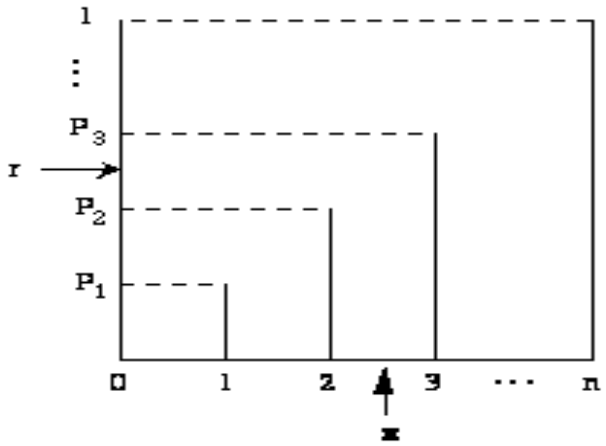


Figure 3.4: Sampling from an empirical discrete probability distribution.

As an example, let us consider the well-known newsboy problem. Let X be the number of newspapers sold by a newsboy per day. From historical data we have the following distribution for X.

X	1	2	3	4	5
f(x)	0.20	0.20	0.30	0.15	0.15

The cumulative probability distribution is:

X	1	2	3	4	5
f(x)	0.20	0.40	0.70	0.85	1

The random variate generator can be summarized as follows:

1. Sample a random number r.
2. Locate the interval within which r falls in order to determine the random variate x.
   a. If $0.85 < r \leq 1.00$ then $x = 5$
   b. If $0.70 < r \leq 0.85$ then $x = 4$
   c. If $0.40 < r \leq 0.70$ then $x = 3$
   d. If $0.20 < r \leq 0.40$ then $x = 2$
   e. Otherwise then $x = 1$

3.5.2 Sampling from a continuous-time probability distribution

Let us assume that the empirical observations of a random variable X can be summarized into the histogram shown in figure 3.5. From this histogram, a set of values $(x_i, f(x_i))$ can be obtained,

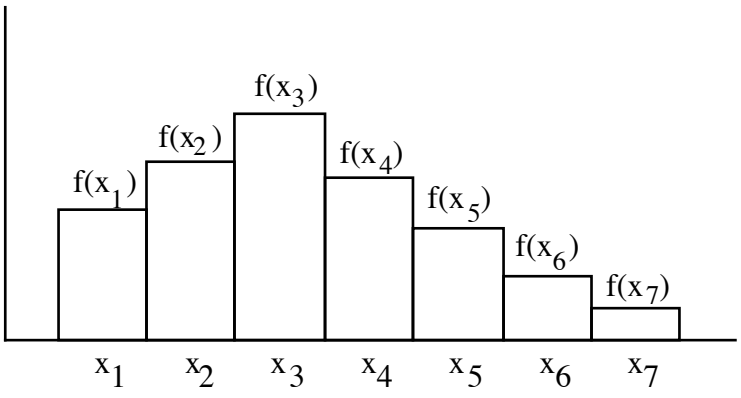


Figure 3.5: Histogram of a random variable X.

where x_i is the midpoint of the ith interval, and $f(x_i)$ is the length of the ith rectangle. Using this set of values we can approximately construct the cumulative probability distribution shown in figure 3..6, where $F(x_i) = \Sigma_{1 \leq k \leq 1} f(x_k)$. The cumulative distribution is assumed to be monotonically increasing within each interval $[F(x_{i-1}), F(x_i)]$.

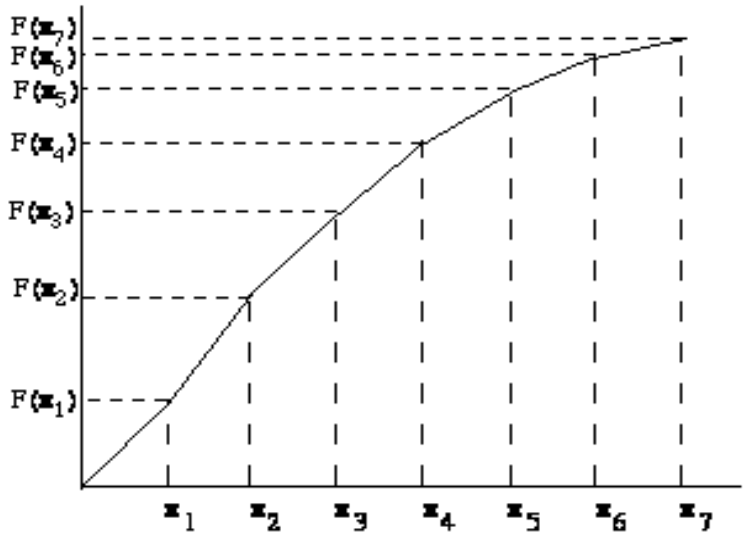


Figure 3.6: The cumulative distribution.

Now, let r be a random number and let us assume that $F(x_{i-1})<r<F(x_i)$. Then, using linear interpolation, the random variate x can be obtained as follows:

$$x = x_{i-1} + (x_i - x_{i-1}) \frac{r - F(x_{i-1})}{F(x_i) - F(x_{i-1})} ,$$

where x_i is the extreme right point of the ith interval.

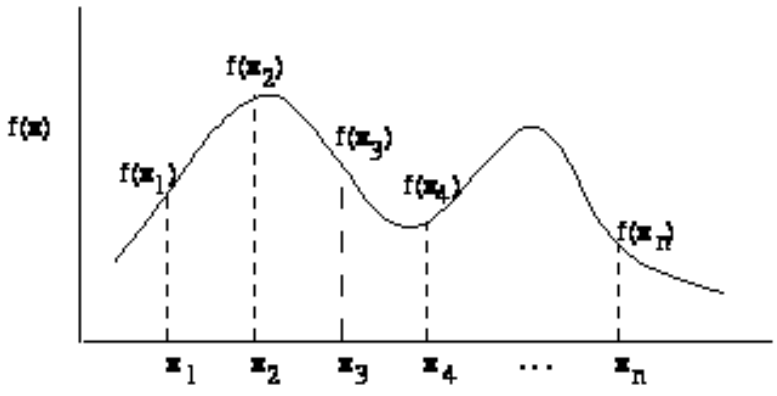


Figure 3.7: "Discretizing" the probability density function.

This approach can be also used to generate stochastic variates from a known continuous probability distribution f(x). We first obtain a set of values (x_i, $f(x_i)$) as shown in figure 3.7. This set of values is then used in place of the exact probability density function. (This is known as "discretizing" the probability density function.) Using this set of values we can proceed to construct the cumulative probability distribution and then obtain random variates as described above. The accuracy of this approach depends on how close the successive x_i points are.

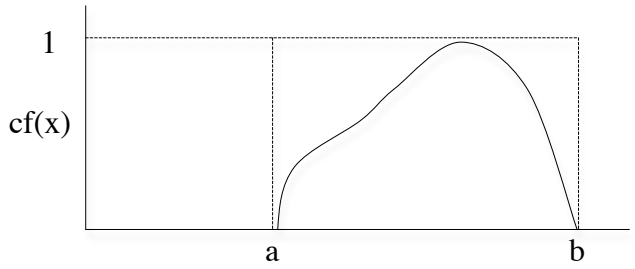


Figure 3.8: Normalized f(x)

3.6 The rejection method

The rejection technique can be used to generate random variates, if f(x) is bounded and x has a finite range, say $a \leq x \leq b$. The following steps are involved:

1. Normalize the range of f(x) by a scale factor c so that $cf(x) \leq 1$, $a \leq x \leq b$. (See figure 3.8)
2. Define x as a linear function of r, i.e. $x = a + (b-a)\, r$, where r is a random number.
3. Generate pairs of random numbers (r_1, r_2).
4. Accept the pair and use $x = a + (b-a)r_1$ as a random variate whenever the pair satisfies the relationship $r_2 \leq cf(a + (b-a)r_1)$, i.e. the pair ($x$,$r_2$) falls under the curve in figure 3.8.

The idea behind this approach is that the probability of r_2 being less than or equal to cf(x) is $p[r_2 \le cf(x)] = cf(x)$. Consequently, if x is chosen at random from the range (a,b) and then rejected if $r_2 > cf(x)$, the probability density function of the accepted x's will be exact.

We demonstrate the rejection method by giving two different examples. The first example deals with random variate generation, and the second example deals with a numerical integration problem.

Example 1: Use the rejection method to generate random variates with probability density function $f(x)=2x$, $0 \le x \le 1$.

This can be accomplished using the following procedure:

1. Select c such that $df(x) \le 1$, i.e. $c = 1/2$.
2. Generate r_1, and set $x = r_1$.
3. Generate r_2. If $r_2 < cf(r_1) = (1/2)2r_1 = r_1$ then accept r_2, otherwise, go back to step 2.

Example 2: Use the rejection method to compute the area of the first quadrant of a unit circle.

We first note that any pair of uniform numbers (r_1, r_2) defined over the unit interval corresponds to a point within the unit square. A pair (r_1, r_2) lies on the circumference if

$$r_1^2 + r_2^2 = 1.$$

The numerical integration can be accomplished by carrying out the following two steps for a large number of times:

1. Generate a pair of random numbers (r_1, r_2).
2. If $r_2 < f(r_1)$, , where $f(r_1) = \sqrt{1 - r_1^2}$, then r_2 is under (or on) the curve and hence the pair (r_1, r_2) is accepted. Otherwise, it is rejected.

The area under the curve can be obtained as the ratio

$$\text{area} = \frac{\text{total number of acceptable pairs}}{\text{total number of generated pairs}} \quad .$$

The rejection method is not very efficient when c(b-a) becomes very large. The method of mixtures can be used, whereby the distribution is broken into pieces and the pieces are then sampled in proportion to the amount of distribution area each contains. This process is identical to the rejection method for each piece of the distribution, plus a straightforward sampling of data.

3.7 Monte Carlo methods

Monte Carlo methods comprise that branch of experimental mathematics which is concerned with experiments on random numbers. Monte Carlo methods are usually associated with problems of theoretical interest, as opposed to the simulation methods described in this book, otherwise known as direct simulation. Unlike direct simulation techniques, Monte Carlo methods are not concerned with the passage of time. Every Monte Carlo computation that leads to quantitative results may be regarded as estimating the value of a multiple integral.

The previously demonstrated rejection method for calculating the integral of a function is an example of Monte Carlo, known as *hit-or-miss* Monte Carlo. An alternative method for calculating an integral of a function is the *crude* Monte Carlo method. Let us assume that we wish to calculate the one-dimensional integral

$$\theta = \int_0^1 f(x)dx \quad .$$

Let $\zeta_1,\zeta_2,...\zeta_n$, be random numbers between 0 and 1. Then, the quantities $f_i = f(\zeta_i)$ are independent random variates with expectation θ. Therefore,

$$\bar{f} = \frac{1}{n} \sum_{i=1}^{n} f_i$$

is an unbiased estimator of θ. Its variance can be estimated using the expression

$$s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (f_i - \bar{f})^2 .$$

Thus, the standard error of $\bar{f}$ is $s/\sqrt{n}$.

The above technique of evaluating θ is based on the following idea. In general, if X is a random variable that takes values in [0,1] and f(x) is a function of this random variable, then

$$E(f(x) = \int_0^1 f(x)g(x)dx$$

where g(x) is the density function of X. Assuming that X is uniformly distributed in (0,1), i.e. g(X) = 1, we obtain

$$E(f(x)) = \int_0^1 f(x)dx \quad .$$

Thus, $\bar{f}$ is an unbiased estimate of E(f(X)).

This technique is more efficient than the technique mentioned in the previous section.

Problems

1. Use the inverse transformation method to generate random variates with probability density function

$$f(x) = \begin{cases} 3x^2 & , \quad 0 \le x \le 1 \\ 0 & , \quad \text{otherwise} \end{cases}$$

2. Apply the inverse transformation method and devise specific formulae that yield the value of variate x given a random number r. (Note that f(x) below needs to be normalized.)

$$f(x) = \begin{cases} 5x & 0 \le x \le 4 \\ x - 2 & 4 < x \le 10 \end{cases}$$

3. Set up a procedure to generate stochastic variates from

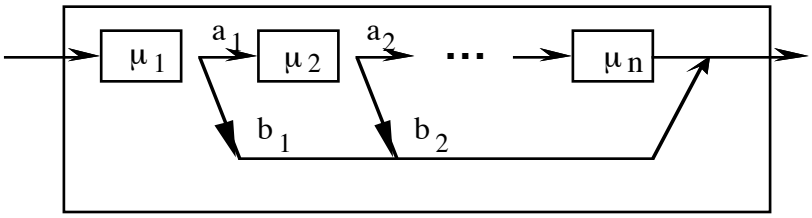
$$f(x) = \begin{cases} x & 0 \le x \le 1/2 \\ 1 - x & 1/2 < x \le 1 \end{cases}$$

4. A customer in a bank may receive service which has an Erlang distribution E_3 (each phase with mean 10) or an Erlang distribution E_4 (each phase with mean 5) with probability 0.4 and 0.6 respectively. Set-up a procedure to generate random variates of a customer's service.

5. Use the rejection method to generate stochastic variates from

$$f(x) = (x-3)^4, \quad 0 \le x \le 10$$

6. Modify the procedure for generating stochastic variates from an Erlang distribution, in order to generate variates from a Coxian distribution. A Coxian distribution consists of a series of exponential distributions, each with a different mean, and it has the following structure:



After receiving an exponential service time with parameter μ_1, there is a probability $b_1(=1-a_1)$ of departing, or a probability a_1 of receiving another exponential service time with parameter μ_2, and so on until the kth exponential service is received.

Computer Assignments

1. Test statistically one of the stochastic variates generator discussed in this Chapter.

2. Consider the machine interference problem. Change your simulation program so that the operational time and the repair time of a machine are exponentially distributed with the same means as before. Make sure that your clocks are defined as real variables. Run your simulation model as before. Each time an event occurs, print out a line of output to show the new value of the clocks and the other relevant parameters.

3. Consider the token-based access scheme. Change your simulation program so that the inter-arrival times are exponentially distributed with the same means as before. The switch over time and the time period T remain constant as before. The packet transmission time is calculated as follows. We assume that 80% of the transmitted packets are short and the remaining 20% are long. The time to transmit a short packet is exponentially distributed with a mean 2.5, and the time to transmit a long packet is exponentially distributed with mean 20.

   Make sure that your clocks are defined as real variables. Run your simulation model as before. Each time an event occurs, print out a line of output to show the new value of the clocks and the other relevant parameters

4. Consider the two-stage manufacturing system. Change your simulation program so that the inter-arrival, service, operational, and repair times are all exponentially distributed. Make sure that your clocks are defined as real variables. Run your simulation model as before. Each time an event occurs, print out a line of output to show the new value of the clocks and the other relevant parameters.

Solutions

1. $$\int_0^1 f(x)\ dx = \int_0^1 3x\ dx = 3\,\frac{x^3}{3}\Big|_0^1 = 3\frac{1}{3} = 1$$

$$F(x) = \int_0^x 3t^2\ dt = 3\,\frac{t^3}{3}\Big|_0^x = \frac{3x^3}{3} = x^3$$

Hence, $r = x^3$ or $x = \sqrt[3]{r}$.

2. $$F(x) = \int_0^x stdt \qquad ,x < 4$$

$$= 5\frac{t^2}{2}\Big|_0^x = \frac{5}{2}\,x^2$$

$$F(x) = \int_0^4 5tdt = \int_4^x (t-2)dt \qquad ,x > 4$$

$$= 5\frac{t^2}{2}\Big|_0^4 + \frac{t^2}{2} - 2t\Big|_4^x$$

$$= 40 + \frac{x^2}{2} - 2x - \left(\frac{16}{2} - 8\right)$$

$$F(x) = \begin{cases} \frac{5}{2}x^2 & 0 \le x \le 4 \\ 40 + \frac{x^2}{2} - 2x & 4 < x \le 10 \end{cases}$$

In order to normalize f(x), we have

$$\int_0^4 5xdx + = \int_4^{10} (x-2)dx$$

or

$$= \frac{5}{2}\, x^2 \Big|_0^4 + \frac{x^2}{2} - 2x \Big|_4^{10}$$

$$= \frac{5}{2}\ 16 + \frac{100}{2} = 2 \times 10 - \frac{42}{2} + 2 - 4$$

Thus,

$$F(x) = \begin{cases} \frac{1}{70}\frac{5}{2}x^2 & 0 \le x \le 4 \\ \frac{1}{70}\left(40 + \frac{x^2}{2} - 2x\right) & 4 < x \le 10 \end{cases}$$

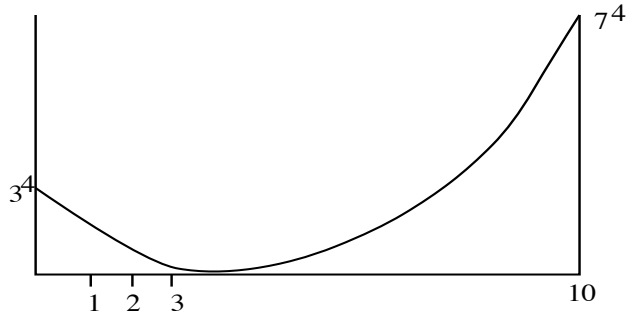
Procedure:

Draw a random number r.

If $r < \frac{40}{70}$ then $r = \frac{5}{140}\ x^2$ or $x = \sqrt{\frac{140}{5} xr}$.

Otherwise, $r = \frac{1}{70}\left(40 + \frac{x^2}{2} - 2x\right)$, from which one can solve for x.

5. Step 1. Use calculus to establish the bounds of f(x)



Thus $c = 1/7^4$

Step 2:

2.1 generate r_1. Then $10r_1$.

2.2 generate r_2.

if $r_2 < \frac{1}{7^4}\ f(10r_1)$ then accept $10r_1$ as a stochastic variate.

Otherwise go back to 2.1.

CHAPTER 4:

SIMULATION DESIGNS

4.1 Introduction

In this Chapter, we examine three different designs for building simulation models. The first two designs are: a) *event-advance* and b) *unit-time advance*. Both these designs are event-based but utilize different ways of advancing the time. The third design is *activity-based*. The event-advance design is probably the most popular simulation design.

4.2 Event-advance design

This is the design employed in the three examples described in Chapter 1. The basic idea behind this design is that the status of the system changes each time an event occurs. During the time that elapses between two successive events, the system's status remains unchanged. In view of this, it suffices to monitor the changes in the system's status. In order to implement this idea, each event is associated with a clock. The value of this clock gives the time instance in the future that this event will occur. The simulation model, upon completion of processing an event, say at time t_1, regroups all the possible events that will occur in the future and finds the one with the smallest clock value. It then advances the time, i.e., the master clock, to this particular time when the next event will occur, say at time t_2. It takes appropriate action as dictated by the occurrence of this event, and then repeats the process of finding the next event (say at time t_3). The simulation model, therefore, moves through time by simply visiting the time instances at which events occur. In view of this it is known as *event-advance design*.

In the machine interference problem, described in section 3.1 of Chapter 1, there are two types of events. That is, the event of an arrival at the repairman's queue, and the event of a departure from the repairman's queue. These events are known as *primary events*. Quite often the occurrence of a primary event may trigger off the creation of a new event. For instance, the occurrence of an arrival at the repairman's queue may trigger off the creation of a departure event if this arrival occurs at a time when the repairman is idle. Such triggered events are known as *conditional events*. The basic approach of the event-based design is shown in the flow chart in figure 4.1.

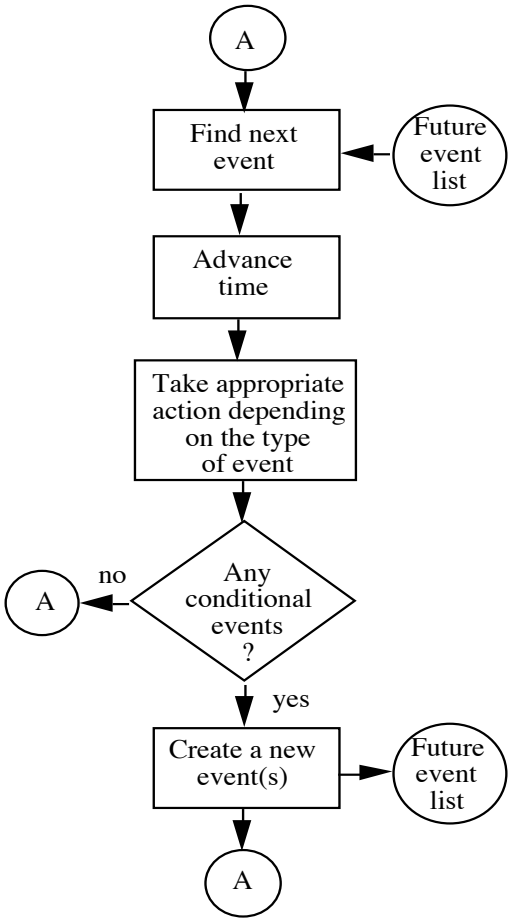


Figure 4.1: The event-advance simulation design.

4.3 Future event list

Let us assume that a simulation model is currently at time t. The collection of all events scheduled to occur in the future, i.e., events with clock greater than t, is known as the *future event list*. For each event scheduled to occur in the future, the list contains the following information:

- Time of occurrence (i.e., value of the event's clock)
- Type of event

The event type is used in order to determine what action should be taken when the event occurs. For instance, using the event type the program can determine which procedure to call.

In the simulation examples described in section 3 of Chapter 1, there were only a few events. For instance, in the case of the machine interference problem there were only two: an arrival to the repairman's queue and a service-ending (departure) event. However, when simulating complex systems, the number of events may be very large. In such cases, finding the next event might require more than a few comparisons. Naturally, it is important to have an efficient algorithm for finding the next event since this operation may well account for a large percentage of the total computations involved in a simulation program. The efficiency of this algorithm depends on how the event list is stored in the computer. An event list should be stored in such a way so as to lend itself to an efficient execution of the following operations.

1. Locating the next future event time and the associated event type.
2. Deleting an event from the list after it has occurred.
3. Inserting newly scheduled events in the event list.

Below we examine two different schemes for storing an event list. In the first scheme, the event list is stored in a sequential array. In the second scheme, it is stored as a linked list.

4.4 Sequential arrays

In this scheme, all future event times are stored sequentially in an array. The simplest way to implement this, is to associate each event type with an integer number i. The clock associated with this event is always stored in the ith location of the array. For instance, in figure 4.2, the clock CL_1 for event type 1 is kept in the first location of the array, the clock CL_2 for the event type 2 is kept in the second position of the array, and so on.

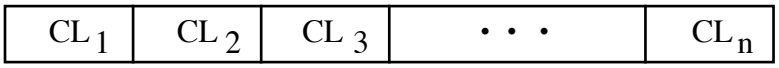


Figure 4.2: Future event list stored sequentially in an array.

Finding the next event is reduced to the problem of locating the smallest value in an array. The following simple algorithm can be used to find the smallest value in an array A.

```
1.   minIndex ← 1
2.   minValue ← A(1)
3.   for i ← 1,n
4.        if minValue ≤ A(i)
5.             continue
6.        else
4.             minValue ← A(i)
5.             minIndex ← i
```

Variable minIndex will eventually contain the location of the array with the smallest value. If minIndex = i, then the next event is of type i and it will occur at time A(i).

An event is not deleted from the array after it has occurred. If an event is not valid at a particular time, then its clock can be set to a very large value so that the above algorithm will never select it. A newly-scheduled event j is inserted in the list by simply updating its clock given by A(j).

The advantage of storing an event list in a sequential array is that insertions of new events and deletions of caused events can be done very easily (i.e., in constant time). The time it takes to find the smallest number in the array depends, in general, on the length of the array n (i.e., its complexity is linear in time). Locating the smallest number in an array does not take much time if the array is small. However, if the array is large, it becomes time consuming. To overcome these problems, one should store the future event list in a linked list.

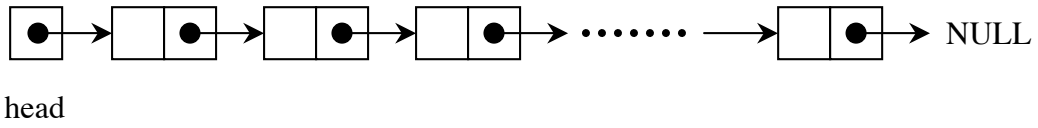


Figure 4.3: A linked list.

4.5 Linked lists

A linked list representation of data allows us to store each data element of the list in a different part of the memory. Thus, we no longer need contiguous memory locations and data can be dynamically added at runtime. In order to access the data elements in the list in their correct order, we store along with a data element the address of the next data element. This is a pointer that points to the location of the next data element. This pointer is often referred to as a *link*. The data element and the link (or links) is generally referred to as a *node*. In general, a node may consist of a number of data elements and links. Linked lists are drawn graphically as shown in figure 4.3. Each node is represented by a box consisting of as many compartments as the number of data elements and links stored in the node. In the example given in figure 4.3, each node consists of two compartments, one for storing a data element and the other for storing the pointer to the next node. The pointer called *head* points to the first node in the list. If the linked list is empty, i.e., it contains no nodes, then head is set to a special value called NULL indicating that it does not point to any node and that the list ends there. The pointer of the last node is always set to NULL indicating that this is the last node in the linked list structure. Due to the fact

that two successive nodes are connected by a single pointer, this data structure is known as a *singly linked list*.

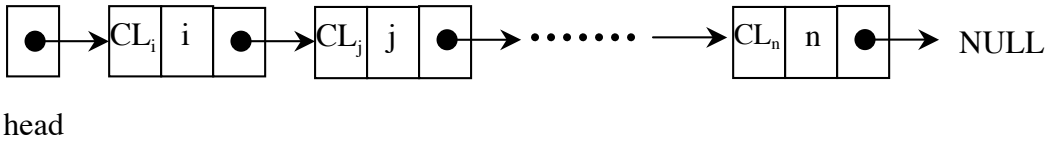


Figure 4.4: Future event list stored sequentially in a linked list.

A single linked list can be used to store a future event list as shown in figure 4.4. Each node will consist of two data elements, namely a clock CL_i showing the future time of an event, and a value i indicating the type of event. The nodes are arranged in an ascending order so that $CL_i \leq CL_j \leq \ldots \leq CL_n$.

4.5.1 Implementation of a future event list as a linked list

The following basic operations are required in order to manipulate the future event list

a. Organize information such as data elements and pointers into a node.
b. Access a node through the means of a pointer.
c. Create a new node(s) or delete an existing unused node(s).
d. Add a node to the linked list.
e. Remove a node from the linked list.

a. Creating and deleting nodes

Let us first see how we can organize the data elements and pointers in a node. We shall use the C programming language syntax to explain the implementation. A simple node which saves event type, event clock and pointer to the next node can be represented as follows:

```
1.  struct node {
2.      int type;
3.      float clock;
4.      struct node* next;
5.  }
```

The above structure is an example of a self referential structure since it contains a pointer to a structure of the same type as itself which would be the next node. In addition to the data elements (i.e. type and clock) shown above the structure can contain other satellite data which may be specific to each event.

A new node must be dynamically allocated each time a new event is created. Dynamic allocation of memory refers to reserving a number of bytes in the memory and setting up a pointer to the first memory location of the reserved space. In C, the system call malloc() is used to return such a pointer. Thus, a node can be dynamically allocated as follows:

```
struct node* nodePtr = (struct node*) malloc (sizeof (struct node));
```

and the above call can be visualized as shown in figure 4.5:

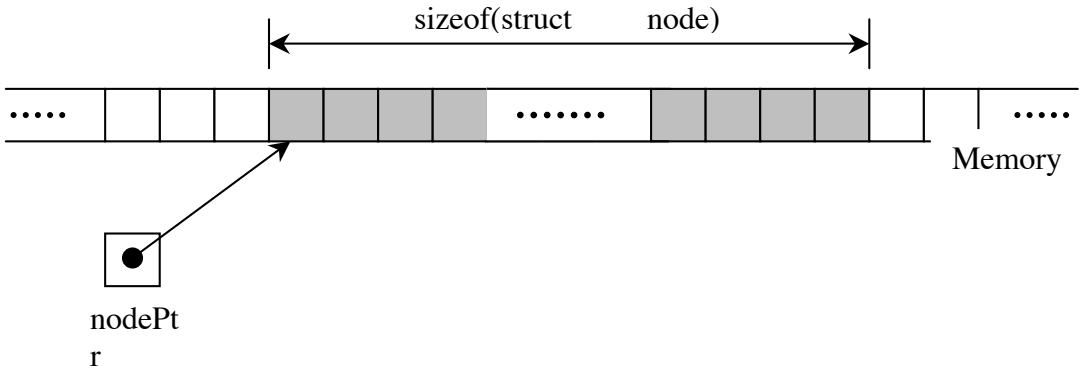


Figure 4.5: Allocating memory for a node using malloc().

We use the cast (struct node*) to explicitly tell the compiler that the memory location returned by this call will be pointed to by a variable which is a pointer to the

node structure that we have defined. Some C compilers may give warnings if such a cast is not applied. We can modify the fields of the structure any time in the program as follows:

```
nodePtr->type = ARRIVAL;
nodePtr->next = NULL;
```

A simple function can be written which creates a new node and set its fields using the parameters passed to it.

b. Function to create a new node

Consider the following function that creates a new node and returns a pointer to it:

```
1.  struct node* createNode(int type, float clock) {
2.      struct node* nodePtr = (struct node*) malloc (sizeof (struct node));
3.      nodePtr->type = type;
4.      nodePtr->clock = clock;
5.      nodePtr->next = NULL;
6.
7.      return nodePtr;
8.  }
```

In the above function we allocate memory for the node and assign values to the node passed into the function as parameters. The last line returns a pointer to the created node. The function can be called from the program as follows:

```
newNodePtr = createNode(ARRIVAL, MCL+5.5);
```

where ARRIVAL is one of the predefined event types and MCL is be the master clock. Thus, we are creating a new ARRIVAL event 5.5 units of time in the future. This function can be visualized as shown in figure 4.6.

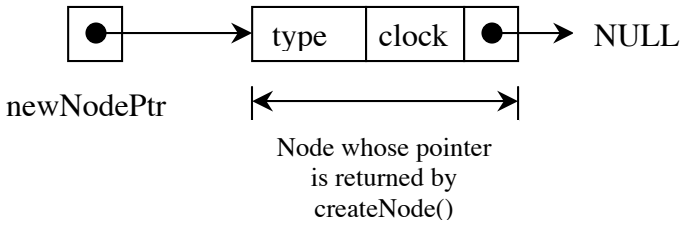


Figure 4.6: Result of calling createNode().

c. Deletion of a node

When a node is deleted the memory allocated from it must be returned to the system. This can be easily done with the system call free() as follows:

```
free(nodePtr);
```

After the above call, nodePtr points to NULL.

d. Creating linked lists, inserting and removing nodes

Now comes the interesting part of creating a linked list, add nodes and removing nodes. To access a linked list we need the pointer to the first node of the list, called a *head* of the linked list. In C, the head can be defined as follows

```
struct node* head = NULL;
```

and the assignment can be visualized as shown in figure 4.7.

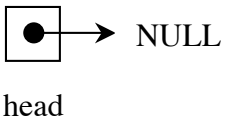


Figure 4.7: Result of calling createNode().

The linked list is initially empty and the head does not point to any node. Thus the head must be explicitly initialized to point to NULL. After inserting nodes the list can be traversed using the head and the pointer to next node in each member node of the list.

e. Inserting a node in a linked list

In order to insert a node into the linked list the pointer to the head of the list and the pointer to the new node must be provided. The node can then be inserted into the list in an appropriate position such that the nodes are ordered according to a specified field. For the purpose of simulation this field will be the event clock.

```
1.  struct node* insertNode(struct node* head, struct node* newNodePtr) {
2.      if (head == NULL) {                                  // Case 1
3.          return newNodePtr;
4.      } else if (head->clock > newNodePtr->clock) {        // Case 2
5.          newNodePtr->next = head;
6.          return newNodePtr;
7.      } else {                                             // Case 3
8.          struct node* prev = NULL;
9.          struct node* curr = head;
10.
11.         while ( (curr!=NULL) && (curr->clock <= newNodePtr->clock) ) {
12.             prev = curr;
13.             curr = curr->next;
14.         }
15.
```

```
16.                 prev->next = newNodePtr;
17.                 newNodePtr->next = curr;
18.
19.                 return head;
20.             }
21.     }
```

The above function can be called from a program as follows:

```
newNodePtr = createNode(ARRIVAL, MCL+5.5);
head = insertNode(head, newNodePtr);
```

As commented in the code above there are three separate cases that we must handle, examined below.

Case 1: head is NULL

In this case, the head is currently pointing to NULL, which means that the list is empty and the node being inserted will be the first element in the list. Thus, after the call to insertNode(), head must point to the node that is being inserted. This can be visualized as shown in figures 4.8 and 4.9.

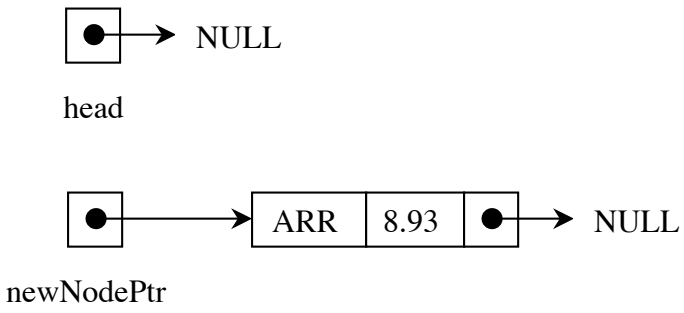


Figure 4.8: Case 1- head and newNodePtr before the call to insertNode().

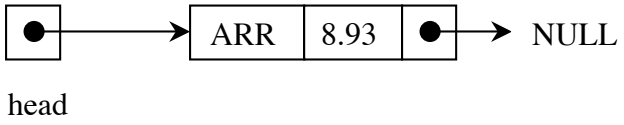


Figure 4.9: Case 1 - head after the call to insertNode().

Case 2: head->clock > newNodePtr->clock

In the second case, the head in not pointing to NULL and the clock value of the head is greater than the clock value of the new node. Thus, the new node must be inserted before the head and the pointer to the new head node must be returned. This can be visualized as shown in figures 4.10 and 4.1.

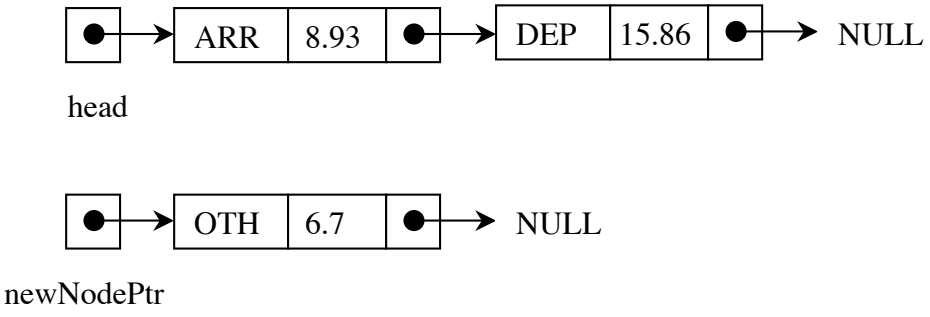


Figure 4.10: Case 2 - head and newNodePtr before the call to insertNode()

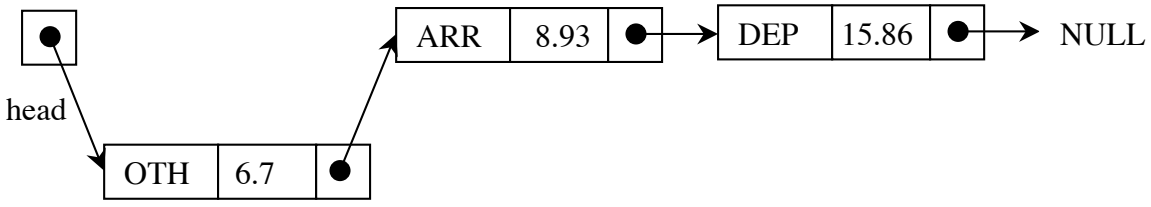


Figure 4.11: Case 2 - head after the call to insertNode()

Notice that in cases 1 and 2 the pointer to the head changes. Thus, the pointer returned by insertNode() must be assigned to the head.

Case 3: Insertion between head and last node of the list

In the third case, the head in not pointing to NULL and the clock value of head is less than the clock value of the new node. Thus the new node must be inserted after the head. This can be visualized as shown in figures 4.12 and 4.3.

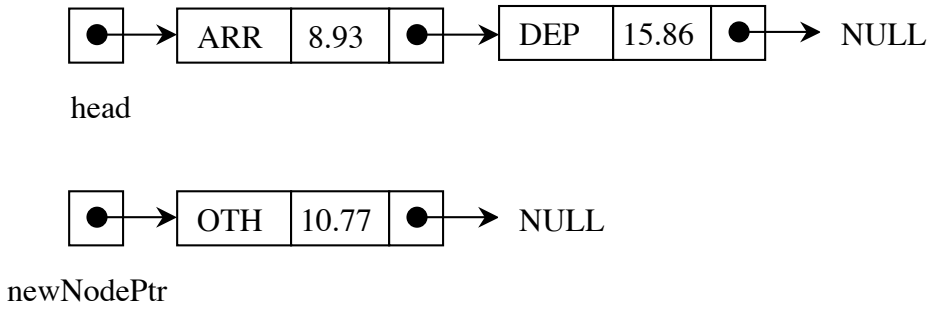


Figure 4.12: Case 3 - head and newNodePtr before the call to insertNode() .

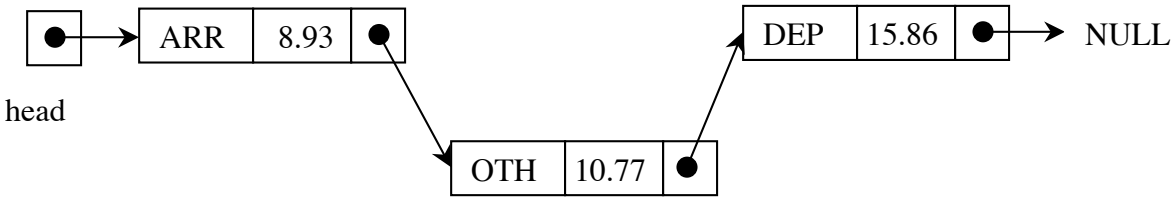


Figure 4.13: Case 3 - head after the call to insertNode().

f. Removing a node from the linked list

The linked list is always maintained in an ascending order of the clock value. Therefore, the next event to occur is the first node of the linked list. The node can be obtained from

the linked list as follows:

```
        struct node* removeNode(struct node* head, struct node** nextRef) {
2.              *nextRef = head;
3.
4.              if (head != NULL) {
5.                      head = head->next;
6.              }
7.              (*nextRef)->next = NULL;
8.
9.              return head;
10.     }
```

After the node is removed, all that needs to be done is to return the node that was pointed to by the head and make the head to point to the next node. This can be done as follows:

```
struct node* nextNodePtr = NULL;
head = removeNode(head, &nextNodePtr);
```

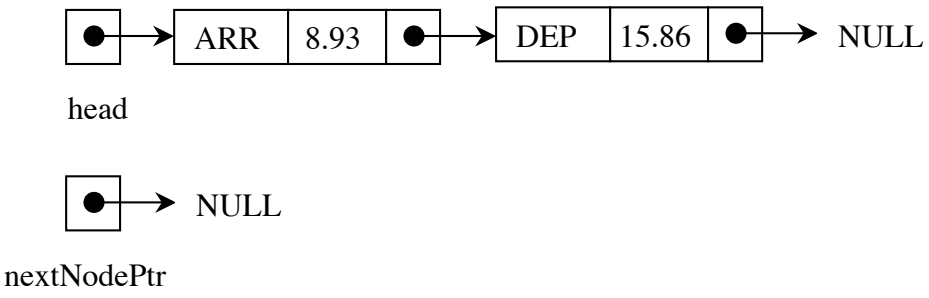


Figure 4.14: head and nextNodePtr before the call to removeNode().

The first parameter passed to removeNode is the head of the linked list. The second parameter (struct node** nextRef) may look different from what we have seen so far. This parameter is of the type pointer to a pointer to a struct node. Such variables are

sometimes called reference pointers. We need a reference pointer in this function to return to the calling function the memory location of the next node. Also, note that in the function call, the address of the nextNodePtr (&nextNodePtr) is passed. This can be visualized as shown in figure 4.14.

It is interesting to see what goes on inside the function removeNode(). When the function gets called, the address of the variable nextNodePtr is passed to it, as shown in figure 4.15.

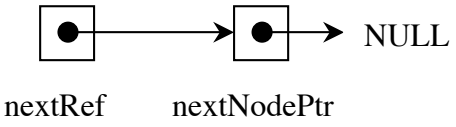


Figure 4.15: nextRef points to nextNodePtr since address of nextNodePtr (&nextNodePtr) is passed to removeNode().

After line 2 of the function above, this changes as shown in figure 4.16:

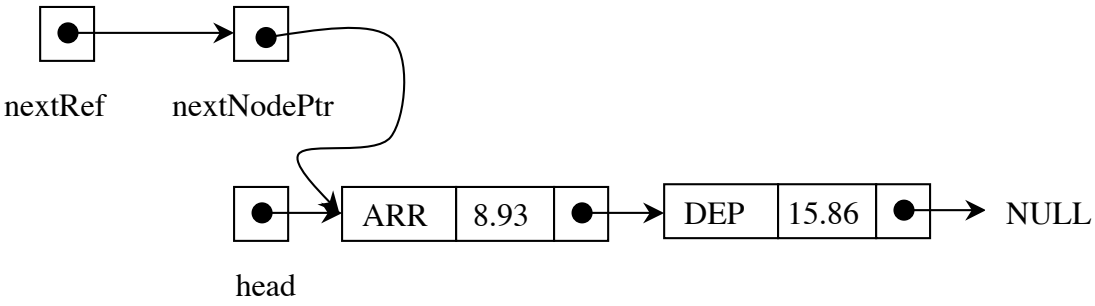


Figure 4.16: nextNodePtr now points to head node of the list.

After line 6, as shown in figure 4.17, the head points to the next node after the head. This new head is returned by the function removedNode() as its return value. And after line 7, the node that will be removed will point to NULL.

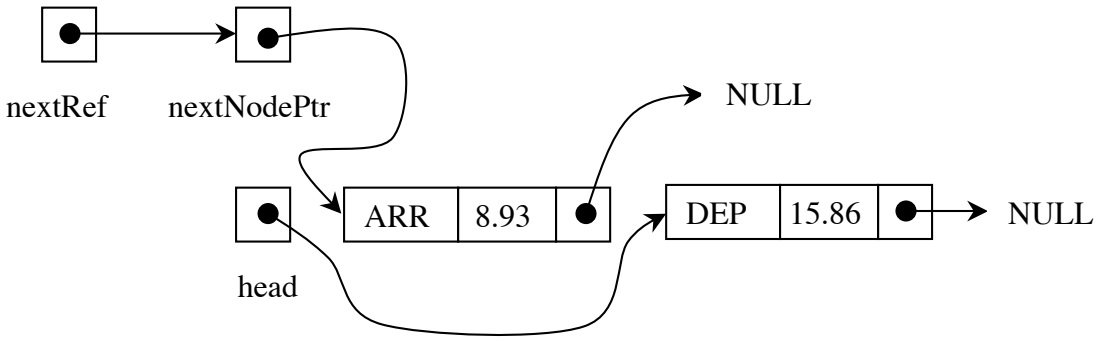


Figure 4.17: The previous head node of the list is now removed from the list and head points to the second node.

Finally, let us take a look at the calling function. As shown in figure 4.18, the head of the list returned by the function removeNode() is assigned to the head variable in the calling function. The nextNodePtr is returned at the address (&nextNodePtr) passed while calling the function.

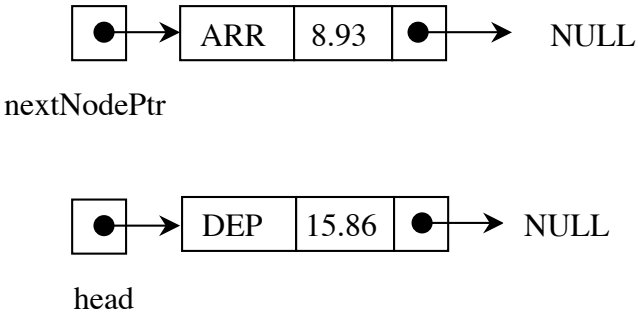


Figure 4.18: Result of calling function removeNode().

4.5.2 Time complexity of linked lists

Let us try to analyze the time complexity of the insert and remove operations on the linked list. In order to insert a node in the linked list we have to traverse the linked list and compare each node until we find the correct insertion position. The maximum number of nodes compared in the worst case will be the total number of nodes in the list. Thus the complexity of the insert operation is linear on n.

Searching a linked list might be time consuming if n is very large. In this case, one can employ better searching procedures. For example, a simple solution is to maintain a pointer B to a node which is in the middle of the linked list. This node logically separates the list into two sublists. By comparing clock value of the node to be inserted with the clock stored in this node, we can easily establish in which sublist the insertion is to take place. The actual insertion can then be located by sequentially searching the nodes of the sublist.

Other data structures like heaps have a much better time complexity for these operations.

4.5.3 Doubly linked lists

So far we have examined singly linked lists. The main disadvantage of these lists is that they can be only traversed in one direction, namely from the first node to the last one. Doubly linked lists allow traversing a linked list in both directions. This is possible because any two successive nodes are linked with two pointers, as shown in figure 4.19. Depending upon the application, a doubly-linked list may be more advantageous than a singly-linked list. A doubly-linked list can be processed using procedures similar to those described above in section 4.5.1.

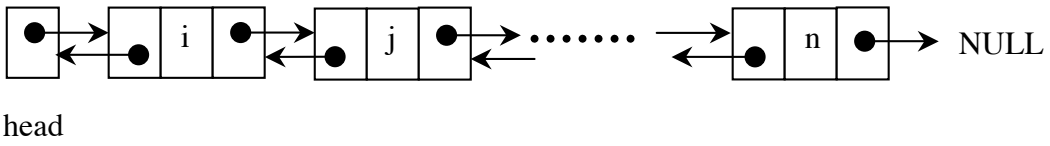


Figure 4.9: A doubly linked list.

4.6 Unit-time advance design

In the event-advance simulation, the master clock is advanced from event to event. Alternatively, the master clock can be advanced in fixed increments of time, each increment being equal to one unit of time. In view of this particular mode of advancing

the master clock, this simulation design is known as the *unit-time advance design*. Each time the master clock is advanced by a unit time, all future event clocks are compared with the current value of the master clock. If any of these clocks is equal to the current

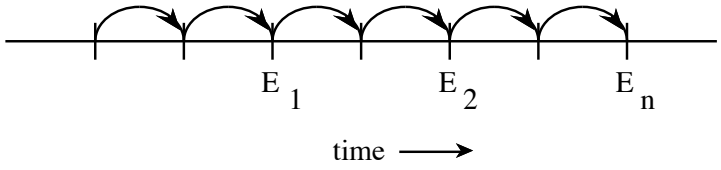


Figure 4.20: The unit-time advance design.

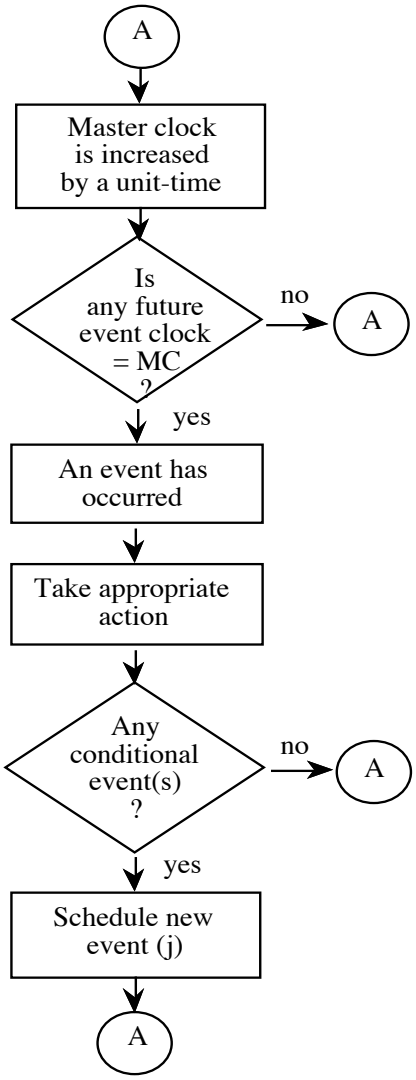


Figure 4.21: The unit-time advance design.

value of the master clock, then the associated event has just occurred and appropriate action has to take place. If no clock is equal to the current value of the master clock, then no event has occurred and no action has to take place. In either case, the master clock is again increased by unit time and the cycle is repeated. This mode of advancing the simulation through time is depicted in figure 4.20. The basic approach of the unit-time design is summarized in the flow-chart given in figure 4.21.

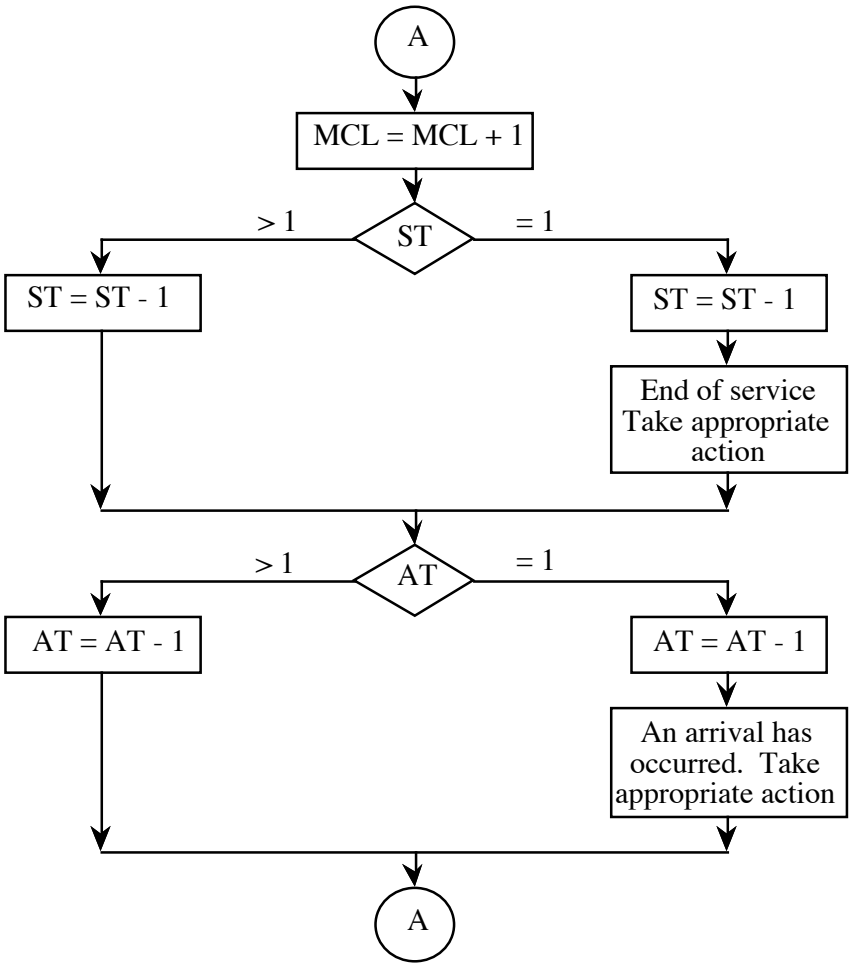


Figure 4.22: A unit-time advance design of a single server queue.

In the flow-chart of the unit-time simulation design, given in figure 4.21, it was implicitly assumed that a future event clock is a variable which, as in the case of the

event-advance design, contains a future time with respect to the origin. That is, it contains the time at which the associated event will occur. Alternatively, a future clock can simply reflect the duration of a particular activity. For instances, in the machine interference problem, the departure clock will simply contain the duration of a service, rather than the future time at which the service will be completed. In this case, the unit-time design can be modified as follows. Each time the master clock is advanced by a unit of time, the value of each future clock is decreased by a unit time. If any of these clocks becomes equal to zero, then the associated event has occurred and appropriate action has to take place. Obviously, the way one defines the future event clock does not affect the unit-time simulation design.

In order to demonstrate the unit-time advance design, we simulate a single queue served by one server. The population of customers is assumed to be infinite. Figure 4.22 gives the flow-chart of the unit-time design. Variable AT contains the inter-arrival time between two successive arrivals. Variable ST contains the service time of the customer in service. Finally, variable MCL contains the master clock of the simulation model.

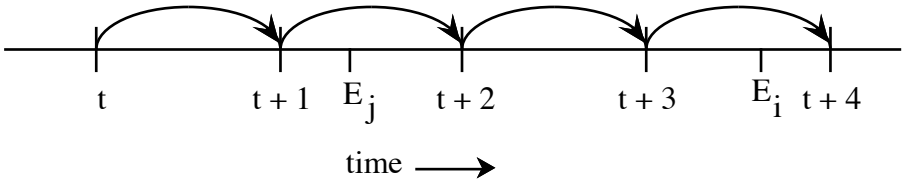


Figure 4.23: Events occurring in between two successive values of the master clock.

4.6.1 Selecting a unit time

The unit time is readily obtained in the case where all future event clocks are represented by integer variables. For, each event clock is simply a multiple of the unit time. However, quite frequently future event clocks are represented by real variables. In this case, it is quite likely that an event may occur in between two successive time instants of the master clock as shown in figure 4.23. Obviously, the exact time of occurrence of an event E is not known to the master clock. In fact, as far as the simulation is concerned, event E

occurs at time t+1 (or t+2 depending upon how the program is set-up to monitor the occurrence of events). This introduces inaccuracies when estimating time parameters related to the occurrence of events. Another complication that might arise is due to the possibility of having multiple events occurring during the same unit of time.

In general, a unit time should be small enough so that at most one event occurs during the period of a unit of time. However, if it is too small, the simulation program will spend most of its time in non-productive mode, i.e. advancing the master clock and checking whether an event has occurred or not. Several heuristic and analytic methods have been proposed for choosing a unit time. One can use a simple heuristic rule such as setting the unit time equal to one-half of the smallest variate generated. Alternatively, one can carry out several simulation runs, each with a different unit time, in order to observe its impact on the computed results. For instance, one can start with a small unit time. Then, it can be slightly increased. If it is found to have no effect on the computed results, then it can be further increased, and so on.

4.6.2 Implementation

The main operation related to the processing of the future event list is to compare all the future event clocks against the master clock each time the master clock is increased by a unit time. An implementation using a sequential array as described in section 4.3.1 would suffice in this case.

4.6.3 Event-advance vs. unit-time advance

The unit-time advance method is advantageous in cases where there are many events which occur at times close to each other. In this case, the next event can be selected fairly rapidly provided that an appropriate value for the unit time has been selected. The best case, in fact, would occur when the events are about a unit time from each other.

The worst case for the unit-time advance method is when there are few events and they are far apart from each other. In this case, the unit-time advance design will spend a

lot of non-productive time simply advancing the time and checking if an event has occurred. In such cases, the event-advance design is obviously preferable.

4.7 Activity-based simulation design

This particular type of simulation design is *activity* based rather than event based. In an event oriented simulation, the system is viewed as a world in which events occur that trigger changes in the system. For instance, in the simulation model of the single server queue described in section 4.6, an arrival or a departure will change the status of the system. In an activity oriented simulation, the system modelled is viewed as a collection of activities or processes. For instance, a single server queueing system can be seen as a collection of the following activities: a) inter arriving, b) being served, and c) waiting for service. In an activity based design, one mainly concentrates on the set of conditions that determine when activities start or stop. This design is useful when simulating systems with complex interactive processing patterns, sometimes referred to as machine-oriented models.

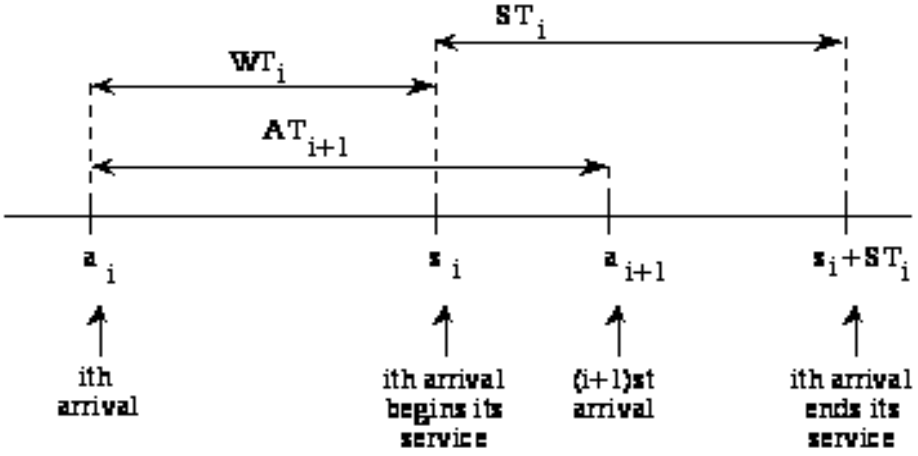


Figure 4.24: Time components related to the ith arrival.

We demonstrate this design by setting up an activity-based simulation model of the single server queue problem studied in section 4.6. Let ST_i and WT_i be the service

time respectively the waiting time of the ith arrival. Also, let AT_{i+1} be the interarrival time between the ith and (i+1)st arrival. Finally, let us assume that the ith arrival occurs at time a_i, starts its service at time s_i and ends its service at time $s_i + ST_i$, as shown in figure 4.24. Let us assume now that we know the waiting time WT_i and the service time ST_i of the ith customer. Let AT_{i+1} be the inter-arrival time of the (i+1)st customer. Then, one the following three situations may occur.

1. The (i+1)st arrival occurs during the time that the ith arrival is waiting.
2. The (i+1)st arrival occurs when the ith arrival is in service
3. The (i+1)st arrival occurs after the ith arrival has departed from the queue

For each of these three cases, the waiting time WT_{i+1} of the (i+1)st customer can be easily determined as follows:

a. $WT_{i+1} = (WT_i - AT_{i+1}) + ST_i$

$= (WT_i + ST_i) - AT_{i+1}$

$= TW_i - AT_{i+1}$,

where TW_i is the total waiting time in the system of customer i.

b. $WT_{i+1} = (WT_i + ST_i) - AT_{i+1}$

$= TW_i - AT_{i+1}$

c. $WT_{i+1} = 0$

Having calculated WT_{i+1}, we generate a service time ST_{i+1} for the (i+1)st arrival, and an inter-arrival time AT_{i+2} for the (i+2)nd arrival. The waiting time of the (i+2)nd arival can be obtained using the above expressions. The basic mechanism of this activity-based simulation model is depicted in figure 4.25.

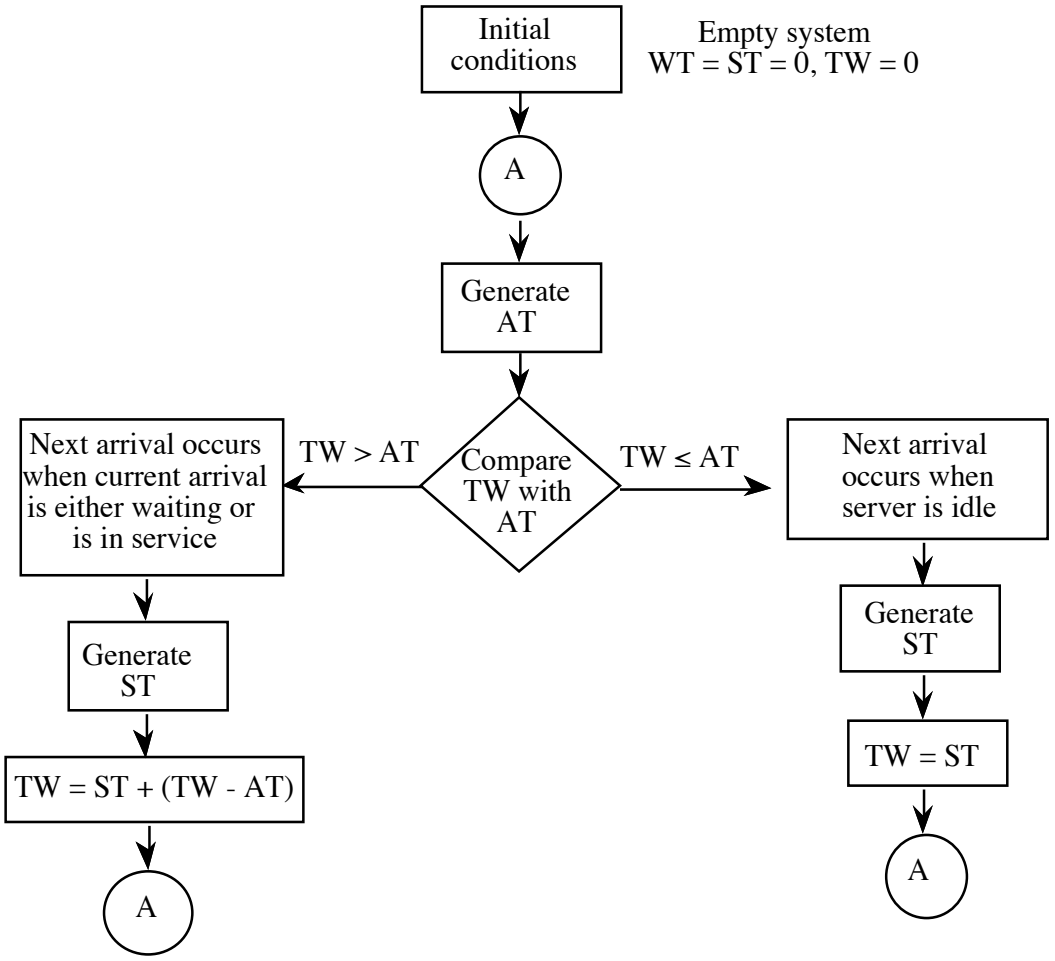


Figure 4.25: An activity-based simulation design for a single server queue.

4.8 Examples

In this section, we further highlight the simulation designs discussed in this Chapter by presenting two different simulation models.

4.8.1 An inventory system

In an inventory system, one is mainly concerned with making decisions in order to minimize the total cost of the operation. These decisions are mainly related to the quantity of inventory to be acquired (or produced) and the frequency of acquisitions. The total cost of running an inventory system is made up of different types of costs. Here, we

will consider the following three costs: a) holding cost, b) setup cost, and c) shortage cost. The holding cost is related to the cost of keeping one item of inventory over a period of time. One of the most important components of this cost is that of the invested capital. The setup cost is related to the cost in placing a new order or changes to production. Finally, the shortage cost is associated with the cost of not having available a unit of inventory when demanded. This cost may be in the form of transportation charges (i.e., expediting deliveries), increased overtime, and loss of future business.

Let It be the inventory at time t. Let S be the quantity added in the system between time t and t´. Also, let D be the demand between these time instances. Then, the inventory at time t´ is

$$I_{t'} = I_t + S - D.$$

If $I_{t'}$ is below a certain value, then an order is placed. The time it takes for the ordered stock to arrive is known as the *lead time*. We assume that the daily demand and the lead time follow known arbitrary distributions. The inventory level is checked at the end of each day. If it is less than or equal to the re-ordering level, an order is placed. The lead time for the order begins to count from the following day. Orders arrive in the morning and they can be disposed of during the same day. During stockout days, orders are backlogged. They are satisfied on the day the order arrives. The fluctuation in the inventory level is shown in figure 4.26.

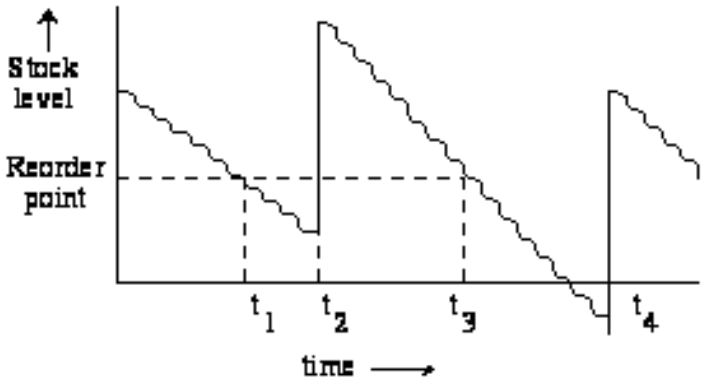


Figure 4.26: An inventory system.

The simulation model is described in the flow chart given in figures 4.27 and 4.28. The model estimates the total cost of the inventory system for specific values of the reordering point and the quantity ordered. The model keeps track of I_t on a daily basis. In view of this, the model was developed using the unit-time advance design. We note that this design arises naturally in this case. A unit of time is simply equal to one day. The lead time is expressed in the same unit time. The basic input parameters to the simulation model are the following. a) ROP, reordering point, b) Q, quantity ordered, c) BI, the

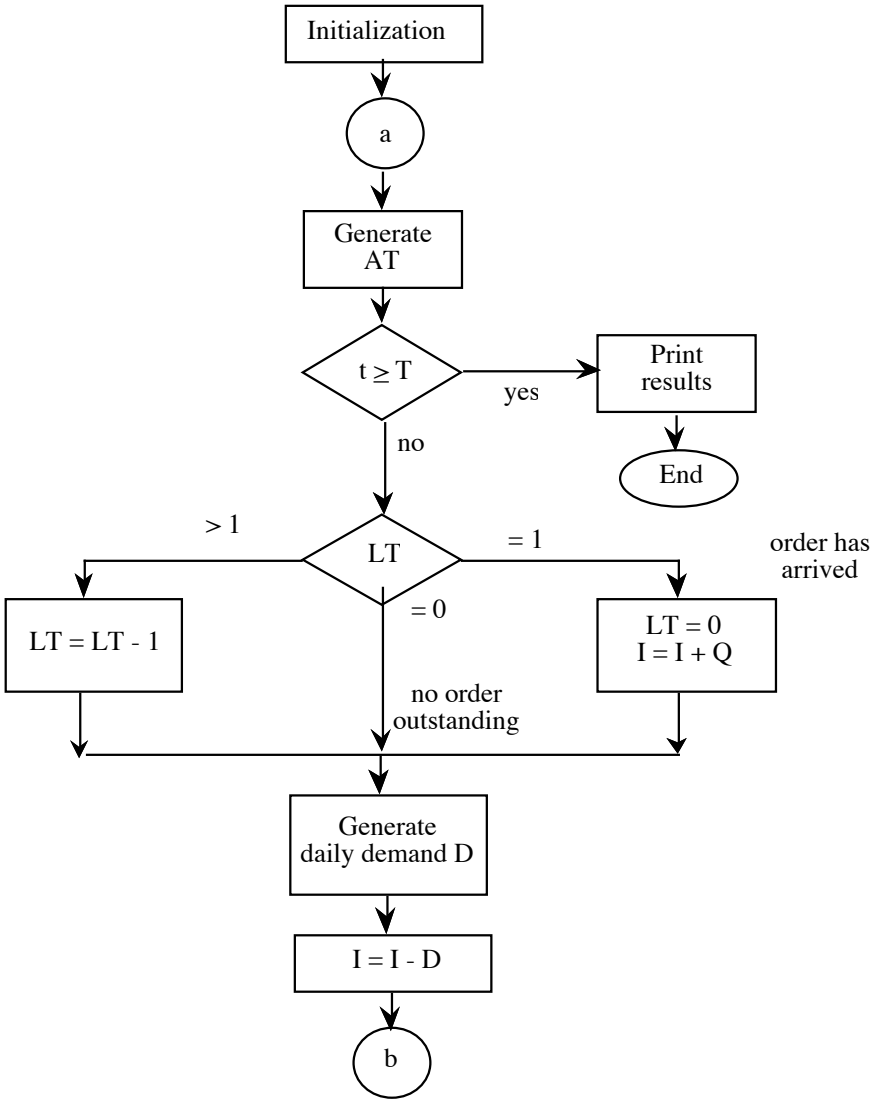


Figure 4.27: A unit-time simulation design of an inventory system.

beginning inventory, d) probability distributions for variables D and LT representing the daily demand and lead time, respectively, e) T, the total simulation time, and f) C1, C2, C3, representing the holding cost per unit per unit time, the setup cost per order, and the shortage cost per unit per unit time, respectively. The output parameters are TC1, TC2, TC3 representing the total holding, setup and shortage costs respectively.

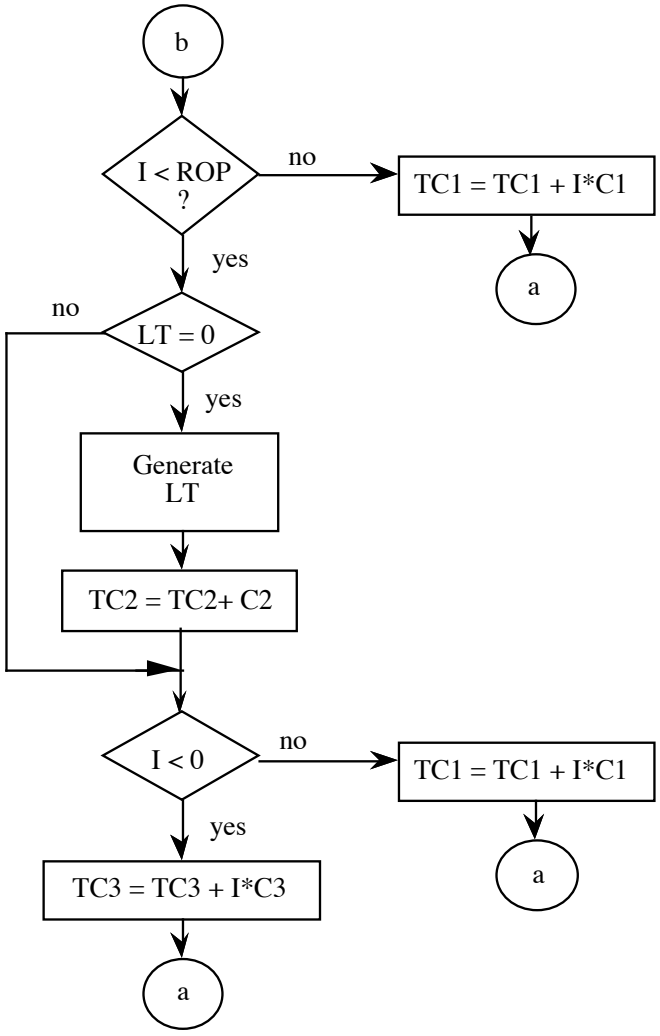


Figure 4.28: A unit time simulation design of an inventory system.

4.6.2 A round-robin queue

We consider a client-server system where a number of clients send requests to a

particular server. For simplicity, we assume that a client is a computer operated by a user, and it is connected to a server over the Internet. Each time a user sends a request, the computer gets blocked, that is, the user cannot do any more processing nor can it send another request, until it gets an answer back from the server. (These requests are often referred to in the client-server parlance as *synchronous* requests). The requests are executed by the server in a round-robin fashion, as will be explained below.

We assume that each user has always work to do. That is, it never becomes idle. A user spends sometime thinking (i.e., typing a line or thinking what to do next), upon completion of which it creates a request that is executed at the CPU. Since the user never becomes idle, it continuously cycles through a think time and a CPU time, as shown in figure 4.29.

Figure 4.29: Cycling through a think time and a CPU time

The requests are executed by the CPU in a *round robin* manner. That is, each request is allowed to use the CPU for a small *quantum* of time. If the request is done at the end of this quantum (or during the quantum), then it departs from the CPU. Otherwise, it is simply placed at the end of the CPU queue. In this manner, each request in the CPU queue gets a chance to use the CPU. Furthermore, short requests get done faster than long ones. A request that leaves the CPU simply goes back to the originating computer. At that instance the user at the computer goes into a think state. The think time is typically significantly longer than the duration of a quantum. For instance, the mean think time could be 30 seconds, whereas a quantum could be less than 1 msec. A request, in general, would require many CPU quanta. If a request requires 5 seconds of CPU time, and a quantum is 1 msec, then it would cycle through the CPU queue 5000 times.

We model this client-server scheme using the queueing system shown in figure 4.30. (We note that we only model the user's think time and the CPU queue, which operates under the round-robin scheme.) This system is similar to the machine

interference problem. The clients are the machines and the CPU is the repairman. A client in the think state is like a machine being operational. The main difference from the machine interference problem is that the CPU queue is served in a round robin fashion rather in a FIF0 manner.

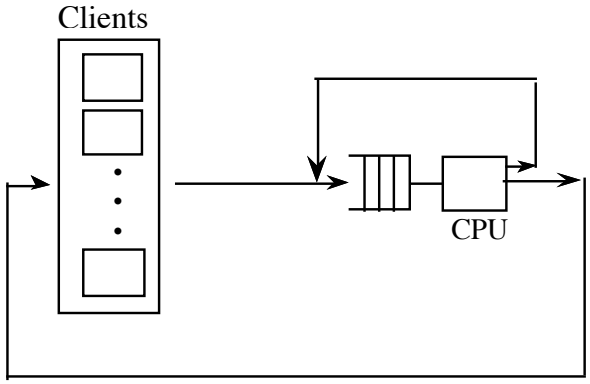


Figure 4.30: A round-robin queue

The basic events associated with this system are: a) arrival of a request at the CPU queue, and b) service completion at the CPU. A request arriving at the CPU queue may be either a *new arrival* (i.e., a user sends a request, thus ending the think state), or it may be a request that has just received a quantum of CPU time and it requires further processing. A departing request from the CPU may either go back to the originating client, or it may simply join the end of the CPU queue for further execution. We observe that during the time that the CPU is busy, departure events occur every quantum of time. However, new arrivals of requests at the CPU queue occur at time instances which may be a few hundreds of quanta apart.

Following the same approach as in the machine interference problem, described in section 1.3.1 of Chapter 1, we can easily develop an event-advance simulation model. We observe that in this case, the event-advance design is quite efficient. The future event list will contain one event associated with a departure from the CPU and the remaining events will be associated with future new arrivals of requests. When a departure occurs, most likely a new departure event will be scheduled to occur in the next quantum of time. This new event will more likely be the next event to occur. In view of this, most of the

time, a newly created event will be simply inserted at the top of the future event list. Such insertions can be done in 0(1) time.

We now give an alternative simulation design of the round-robin queue which utilizes both the event-advance and unit-time advance designs! Specifically, all the events related to new arrivals at the CPU queue are kept in a linked list as shown in figure 4.31. Each node contains a future event time and a client identification number. The event time simply shows the time at which the user will stop thinking and will access the CPU. Nodes are ordered in an ascending order of the data element that contains the future event time. Thus, the next new arrival event is given by the first node of the linked list.

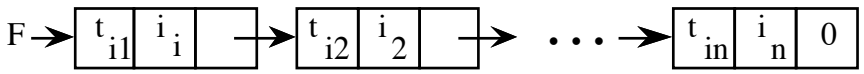


Figure 4.31: Future event list of all new arrivals to the CPU queue.

All the information regarding the requests in the CPU queue, including the one in service, is maintained in the separate linked shown in figure 4.32. This linked list is known as a *circular* singly linked list, since it is singly linked and the last node is linked to the first node. Each node contains the number of quanta required by a request and its terminal identification number. Pointers B and E point to the beginning and end of the list respectively. The nodes are ordered in the same way that the requests are kept in the CPU queue. Thus, the first node corresponds to the request currently in service. When a new request arrives at the CPU queue, a new node is created which is attached after node E. If a request requires further service upon completion of its quantum, then its node is simply placed at the end of the list. This is achieved by simply setting E←B and B←LINK(B).

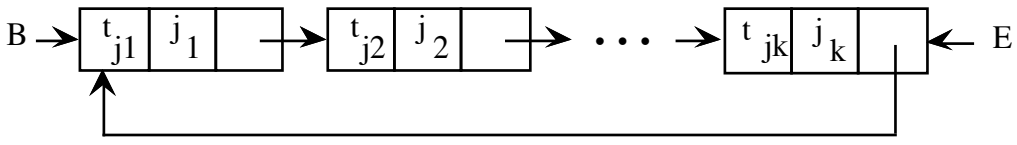


Figure 4.32: Future event list of all new arrivals to the CPU queue.

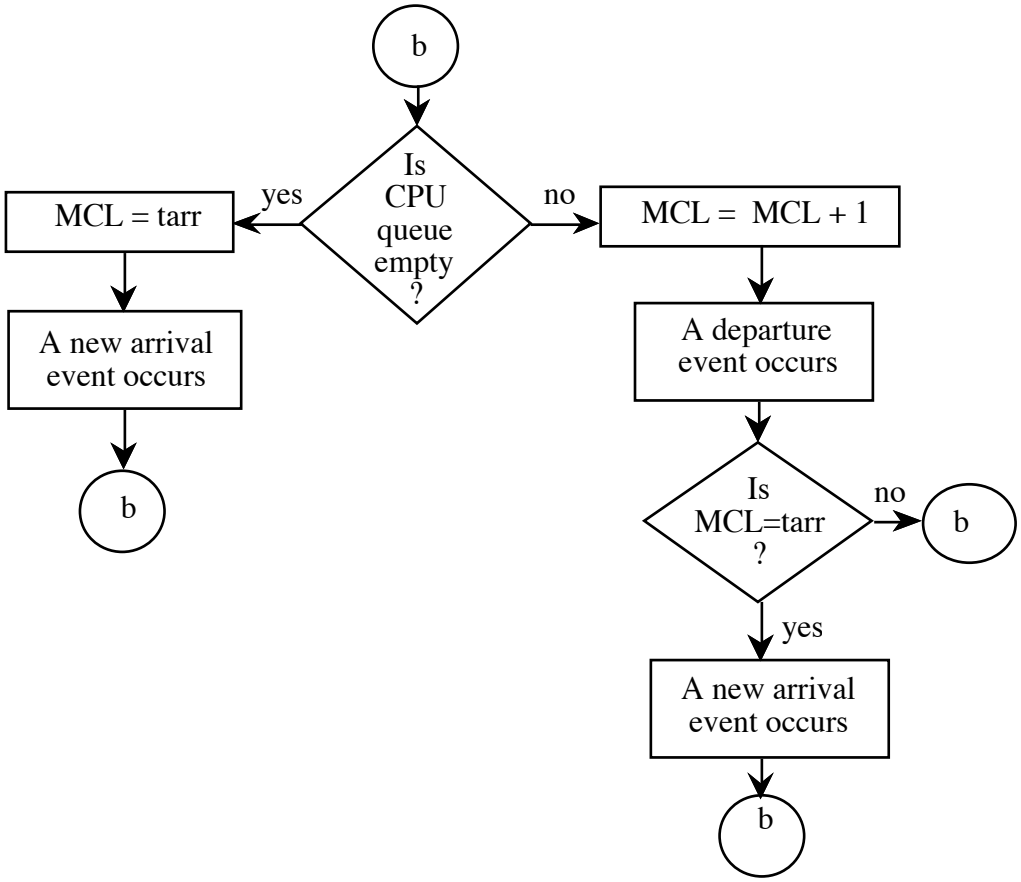


Figure 4.33: Hybrid simulation design of the round-robin queue.

The simulation model operates under the unit-time advance design during the period of time that the CPU is busy. During the time that CPU is idle, the simulation model switches to an event-advance design. This hybrid design is summarized in figure 4.33. Note that tarr gives the time of the next new arrival at the CPU queue. The remaining details of this program. are left up to the reader as an exercise!

Problems

Consider the following systems:

a. Checkout stands at a supermarket

b. Teller's window at a bank
c. Elevators serving an office building
d. Traffic lights in a configuration of 8 city blocks
e. Outpatient clinic
f. Pumps at a gasoline station
g. Parking lot
h. Runways at an airport
i. A telecommunication system with 5 nodes (virtual HDLC three linked lists)
j. Solar heating of a house

Choose any of the above systems. First, describe how the system operates. (Make your own assumptions whenever necessary. Make sure that these assumptions do not render the system trivial!) Then, set up a simulation model to represent the operations of the system. If you choose an event-based simulation design, then state clearly which are the state variables and what are the events. For each caused event, state clearly what action the simulation model will take. If you choose an activity-based design, state clearly which are the activities and under what conditions they are initiated and terminated.

Computer assignments

1 Implement the hybrid simulation model of the round-robin queue discussed in section 4.6.2.

2 Consider the machine interference problem. Modify your simulation model so that the event list is maintained in the form of a linked list. Assume that the queue of broken-down machines can be repaired by more than one server (i.e., there are more than one repairman repairing machines off the same queue). Parametrize your program so that it can run for any number of machines and repairmen (maximum 20 and 10, respectively). Run your simulation model until 20 repairs have been completed. As before, each time an event occurs, print out the usual line of output and also

additional information pertaining to the event list. Check by hand that your linked list implementation is correct.

3 Consider the token-based access scheme problem. Assume that transmissions are not error-free. That is, when a packet arrives at the destination node, it may contain errors. In this case, the packet will have to be re-transmitted. The procedure is as follows:

Upon completion of the transmission of a packet, the host will wait to hear from the receiving host whether the packet has been received correctly or not. The time for the receiver to notify the sender may be assumed to be constant. If the packet has been correctly received, the host will proceed with the next transmission. If the packet has been erroneously transmitted, the sender will re-transmit the packet. There is 0.01 probability that the packet has been transmitted erroneously. The sender will re-transmit the packet immediately. This procedure will be repeated until the packet is correctly transmitted. No more than 5 re-transmissions will be attempted. After the 5th re-transmission, the packet will be discarded, and the sender will proceed to transmit another packet. All these re-transmissions take place while the host has the token. When the token's time-out occurs, the host will carry on re-transmitting until either the packet is transmitted correctly, or the packet is discarded.

Describe how you will modify your simulation model in order to accommodate the above acknowledgement scheme. Can this retsransmission scheme be accommodated without introducing more events? If yes, how? If no, what additional events need to be introduced? Describe what action will be taken each time one of these additional events takes place. Also, describe how these new events will interact with the existing events (i.e., triggering-off each other).

4 Consider the token-based access scheme problem. Modify the simulation design in order to take advantage of the structure of the system. Specifically, do not generate arrivals to each of the nodes. Store the residual inter-arrival time when the node surrenders the token. Then, when the token comes back to the station, generate arrivals (starting from where you stopped last time) until the token times-out or it is

surrendered by the node. This change leads to a considerably simpler simulation model.

Appendix A: Implementing a priority queue using a heap

A priority queue is an interesting data structure used for storing events and it can be used to implement a future event list. It supports two important operations: insertion of a new event and removal of an event. The highest priority event can be obtained from a priority queue without performing a time intensive search as is done in arrays and linked lists. This is because the events in the priority queue are organized in such a way so that the highest priority event will always be stored at a specific location in its internal data structure. Whenever an event is added to the priority queue certain operations are performed in order to maintain this property. These operations are done much faster than when we use arrays or linked lists. Thus, simulations with large number of events will run much faster if the future event list is implemented as a priority queue.

A priority queue is a binary tree called *heap* with functions to insert a new event and remove the highest priority event. In the next section we explain the heap data structure. (We note that the heap algorithms were taken from Chapter 6 of " Introduction to Algorithms" by T. H. Cormen et al.)

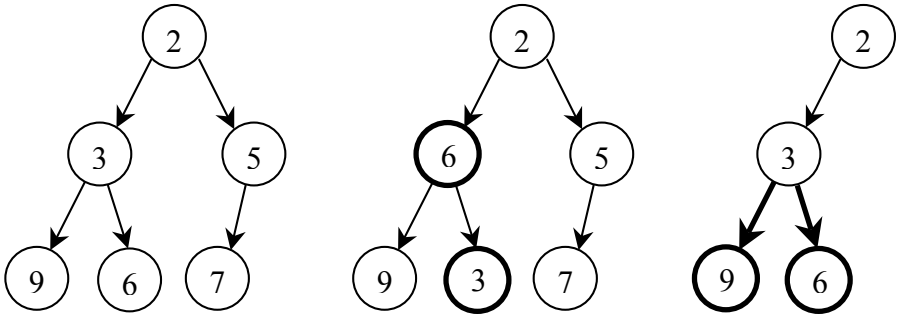
A.1 A heap

A heap is a binary tree data structure with the following important properties:

1 The binary tree is filled completely at each level, except possibly the last level. At the last level the elements are filled from left to right. Such trees are called *complete binary trees*.

2 The value of a child element is always greater than or equal to the value of its parent. This ensures that the value of the root element is the smallest. This property is also called the *heap property*. Also, such heaps are called *min-heaps*. Alternatively, a max-heap is a structure where the value of child node is always smaller than or equal to its parent. Thus for a max-heap the root element will always be the one with the largest value.

In this section we will only deal with min-heaps. Three different examples of such heaps are shown in figure A.1, of which only (a) satisfies both the properties of a heap. The data structure (b) does not satisfy the heap property since the element containing 6 has a value greater than its right child, which has value 3. The data structure (c) does not satisfy the property of complete binary trees either since the second level is not completely filled. In order for (c) to be a heap either the element with value 9 or 6 must be the child of the root element which has value 2.



(a) Valid **(b) Invalid** **(c) Invalid**

Figure A1: Examples of valid and invalid heaps

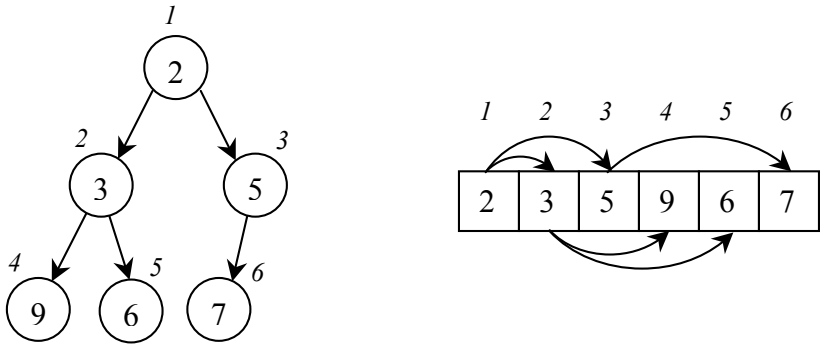
A.1.1 Implementing a heap using an array

The binary tree of a heap can be efficiently stored using an array of n sequential locations. If we number the nodes from *1* at the root and place:

- left child of node *i* at *2i*
- right child of node *i* at *2i +1* in the array

then the elements of the heap can be stored in consecutive locations of an array. An example of this is shown in figure A2. The indices *1* through *6* represent the six consecutive array locations and the values 2, 5, 3, 9, 6 and 7 are the data elements stored in the nodes of the tree. Observe that the root element is stored at location *1* of the array. The left child of the root element is at

location 2, i.e., 2 x *1*, and the right child at location 3, i.e., 2 x *1* + 1. Each node shares this relationship with its left and right child.



(a) Heap **(b) Array representation**

Figure A2: Storing a heap in an array

The indices of the parent, left child and right child of node with index *i* can be computed as follows:

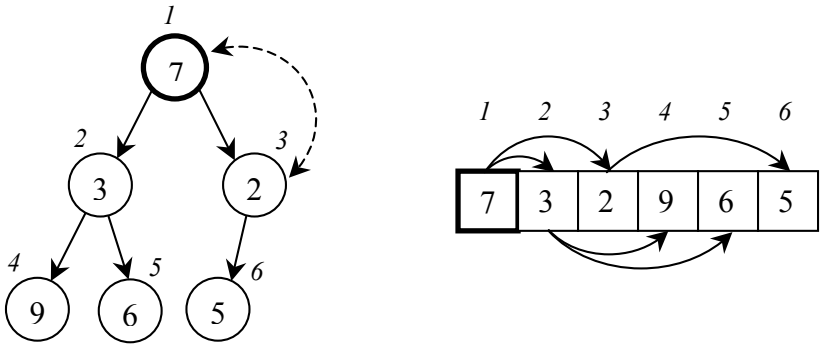
```
PARENT(i)
    return (floor (i/2))
LEFT(i)
    return (2i)
RIGHT(i)
    return (2i + 1)
```

a. Maintaining the heap property

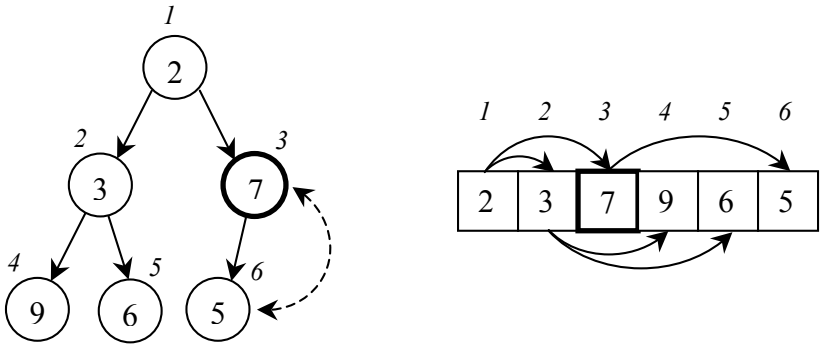
In order to maintain the heap property the elements of a heap must be rearranged by calling the HEAPIFY routine. This is a recursive routine called on a node that violates the heap property, i.e. its value is bigger than one of its child's value. The violating element moves down in the tree in each of the recursive calls until it finds a place so that the heap property is satisfied.

When HEAPIFY is called on a node, it is assumed that the trees sub-rooted at its left and right children already satisfy the heap property. All through our discussion our first call to HEAPIFY will be to the root node of the heap. However the recursive calls will be made to nodes at lower levels.

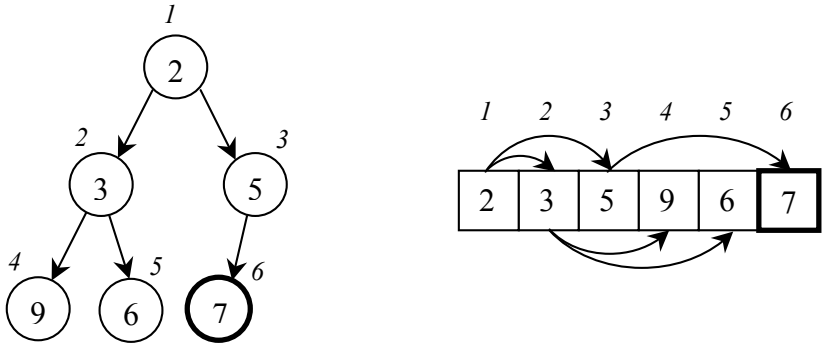
In order to understand how HEAPIFY works, we consider the example given in figures A.3, A.4 and A.5. In figure A.3, the violating node is the root node (array index *1*). Observe that the trees sub-rooted at its left and right children are satisfying the heap property. Let A be the name of the array representing the heap, then the first call to HEAPIFY will be HEAPIFY(A,1). In this call A[1] is swapped with min(A[2],A[3]) which in this example is A[3]. The violating element with value 7 moves to A[3], as shown in figure A.4, but it still violates the heap property in its new location. Thus, HEAPIFY(A,3) is called recursively and A[3] is swapped with A[6], and, as shown in figure A.5, the element with value 7 no longer violates the heap property.



A.3: The violating node is the root



A.4: The node with value 7 still violates the heap property.



A5: The node with value 7 no longer violates the heap property.

The HEAPIFY algorithm

HEAPIFY (A, i)

```
1.    left ← LEFT(i)
2.    right ← RIGHT(i)

3.    if (left ≤ heap-size[A]) and (A[left] < A[i])
4.        smallest ← left
5.    else
6.        smallest ← i
7.    if (right ≤ heap-size[A]) and (A[right] < A[smallest])
8.        smallest ← right

9.    if (smallest ≠ i)
10.       SWAP(A[i], A[smallest])
11.       HEAPIFY (A, smallest)
```

In lines 1 and 2 LEFT(*i*) and RIGHT(*i*) are obtained using the method described above. In lines 3 to 8 the index of smallest of A[*left*], A[*right*] and A[*i*] is stored in the variable *smallest*. In line 9 to 11 if *i* was not the *smallest* then the values A[*i*] and A[*smallest*] are swapped and HEAPIFY is called recursively on the node to which the violating element was moved.

A.2 A priority queue using a heap

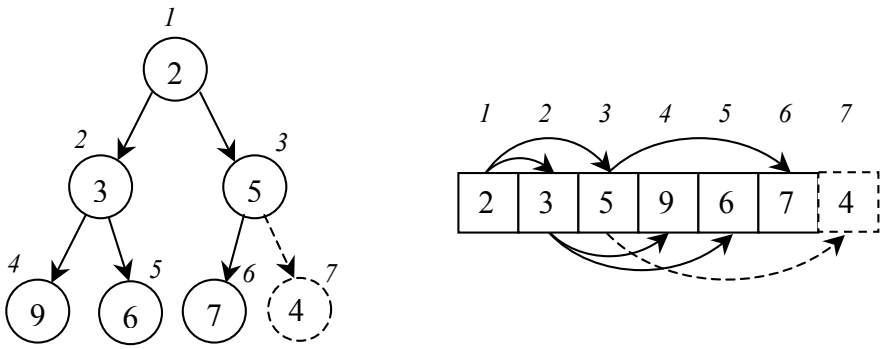
As mentioned earlier, a priority queue is a heap with sub-routines to insert a new event and remove the highest priority event. HEAP-INSERT is used to insert a new event and EXTRACT-MIN is used to remove the event with the highest priority from the heap. Below, we describe each sub-routine and highlight its operation through an example.

HEAP-INSERT (A, key)

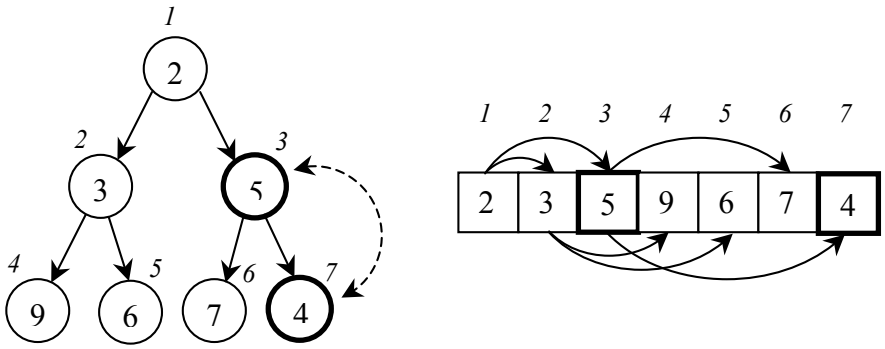
1. heap-size[A] ← heap-size[A] + 1
2. A[heap-size[A]] ← key
3. i ← heap-size[A]

4. while (i > 1) and (A[PARENT(i)] > A[i])
5. SWAP(A[i], A[PARENT(i)])
6. i ← PARENT(i)

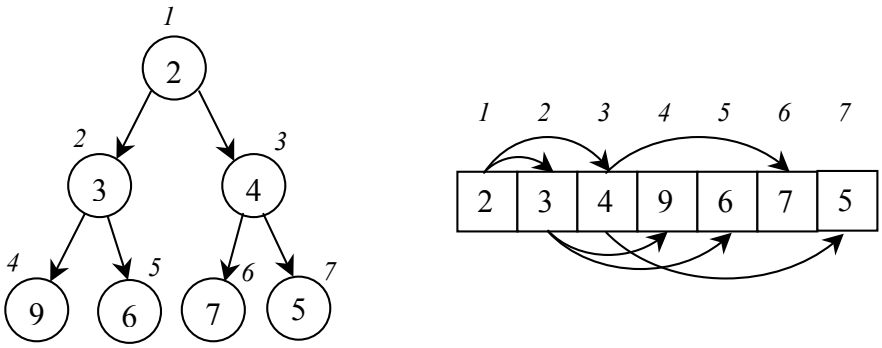
As an example, let us consider a heap with six nodes numbered *1* through *6*. Again let A be the array representing the heap. The heap and its array representation are shown in figure A6. Now, if want to insert a new element with value 4 into this heap we would call HEAP-INSERT with key equal to 4. Figure A7 shows the result of this operation. The newly inserted node is compared with its parent, and since the parent has a greater value than the new node, the parent will move down while the new node moves into the place of its parent, as shown in figure A8.



A6: The heap and its array representation



A7: The new node is inserted in the heap and it is compared to its parent



A8: The parent node and the new node switch places

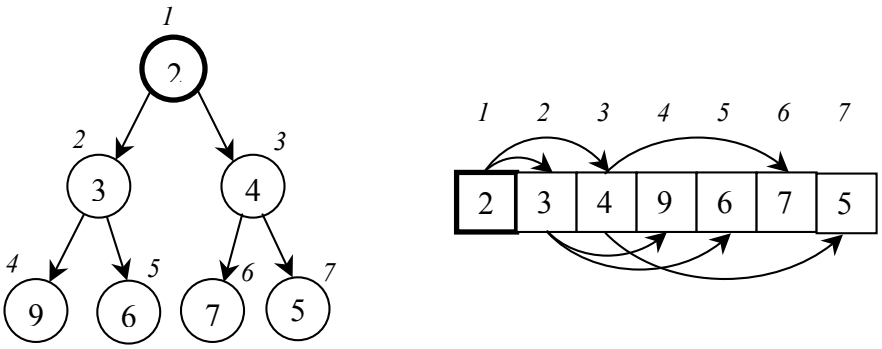
EXTRACT-MIN (A)

1. if (heap-size[A] < 1)
2. error "No elements in the heap"

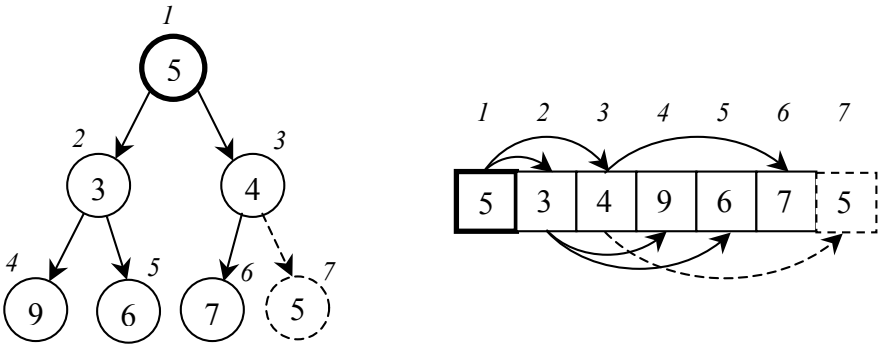
3. min ← A[1]
4. A[1] ← A[heap-size[A]]
5. heap-size[A] ← heap-size[A] – 1
6. HEAPIFY (A, 1)
7. return (min)

Let us perform EXTRACT_MIN on the heap obtained after the insert operation in previous example. The call will be made as EXTRACT-MIN(A). The minimum is always the root element of the heap. As shown in figure A9, the root element has a value of 2. The last

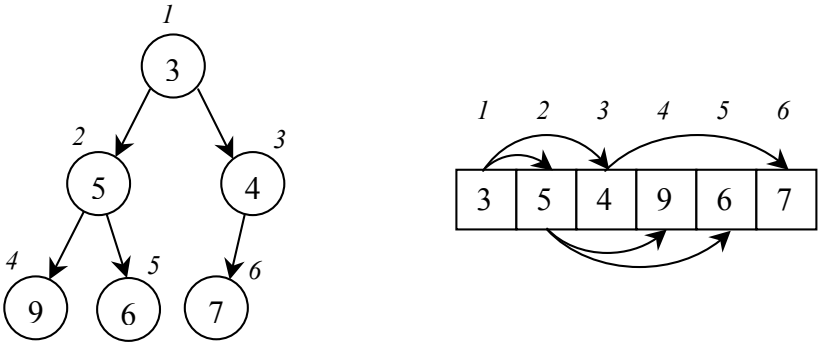
element A[heap-size[A]] is moved to the root node A[1] and the heap-size is reduced by one, see figure A10. HEAPIFY(A,1) is now called, and the final result is shown in figure A11.



A9: The minimum is always the root element



A10: The last element A[heap-size[A]] is moved to the root node A[1]



A11: The heap now satisfies the heap property.

A.3 Time complexity of priority queues

So far we have discussed how to implement a priority queue as a heap and how to insert a new event and remove the highest priority event. In this section, we show that a priority queue runs much faster than arrays and linked lists. The running time (called the algorithmic time complexity) of an algorithm is proportional to the number of comparisons performed by the algorithm in order to achieve its objective. The efficiency of various algorithms is compared based on their time complexity.

The two basic operations performed on arrays, linked lists and priority queues are: insert a new event and remove the next event. In table A.1, we approximate the maximum number of comparisons done when these two operations are performed using each of the three data structures. We assume that we already have n events in the data structure before either operation is performed.

	Insert new event	**Get/Remove next event**
Array	Store the new event at the first available location. *Maximum Comparisons = 0*	Search the entire array to get the event whose execution time is least. *Maximum comparisons = n*
Linked list	Search for the correct position where the new event must be inserted. *Maximum comparisons = n*	Remove the first event. *Maximum comparisons = 0*
Priority queue	Use HEAP-INSERT which searches for the correct position to insert the event from the lowest level up to the root level in the worst case. *Maximum comparisons=* *height of the tree* = $log_2(n)$* *If a heap has n elements, then its height is equal to $log_2(n)$.	Use EXTRACT-MIN which removes the root element, moves the last element to the root and runs HEAPIFY on the root node. HEAPIFY performs two comparisons (one with the left and one with the right child) and calls itself recursively until the correct position for the violating element is found. The maximum number of times this procedure will be called recursively equals the height of the binary tree. *Maximum comparisons = 2 x height of the tree = 2 $log_2(n)$*

Table A.1: Time complexity comparisons

A.4 Implementing a future event list as a priority queue

An event should be defined as a structure or an object with one of its attributes being the clock associated with the occurrence of this event. An array of such objects can then be declared to hold all the events. The size of the array must be big enough to hold the maximum number of events at any point during the simulation. In the examples above, the values used in the nodes of the binary trees or in the arrays can be seen as the clock attribute of the event objects. All comparisons between the events of the heap should be made by comparing this attribute, but whenever an element is be moved to another location in the array, the entire event object must be moved.

The size of the array must be declared at the beginning of the simulation, which means that the maximum required size of the array must be determined in advance. Alternatively, one can allocate memory for events dynamically. For instance, in C one can use the malloc and realloc functions whereas in Java one can use an ArrayList or a Vector. Also, almost all high level programming languages have a library class for the priority queue. Thus, you can directly use the library without implementing them on your own.

CHAPTER 5:

ESTIMATION TECHNIQUES FOR ANALYZING ENDOGENOUSLY CREATED DATA

5.1 Introduction

So far we have examined techniques for building a simulation model. These techniques were centered around the topics of random number generation and simulation design. The reason why one develops a simulation model is because one needs to estimate various performance measures. These measures are obtained by collecting and analyzing endogenously created data. In this Chapter, we will examine various estimation techniques that are commonly used in simulation. Before we proceed to discuss these techniques, we will first discuss briefly how one can collect data generated by a simulation program.

5.2 Collecting endogenously created data

A simulation model can be seen as a reconstruction of the system under investigation. Within this reconstructed environment, one can collect data pertaining to parameters of interest. This is similar to collecting actual data pertaining to parameters of interest in a real-life system. Using the techniques outlined in the previous Chapters, one can construct, say, an event simulation model of a system. This model simply keeps track of the state of the system as it changes through time. Now, one can incorporate additional logic to the simulation program in order to collect various statistics of interest such as the frequency of occurrence of a particular activity, and the duration of an activity. These

statistics are obtained using data generated from within the simulation programs, known as *endogenous data*.

Endogenous data can sometimes be collected by simply introducing in the simulation program single variables, acting as counters. However, quite frequently one is interested in the duration of an activity. For instance, in the machine interference problem one may be interested in the down time of a machine. This is equal to the time the machine spends queueing up for the repairman plus the duration of its repair. In such cases, one needs to introduce a storage scheme where the relevant endogenously created data can be stored. The form of the storage scheme depends, of course, upon the nature of the problem.

In the machine interference problem, the down time of a machine can be obtained by keeping the following information: a) time of arrival at the repairman's queue, and b) time at which the repair was completed. This information can be kept in an array. At the end of the simulation run, the array will simply contain arrival and departure times for all the simulated breakdowns. The down time for each breakdown can be easily calculated. Using this information one can then obtain various statistics such as the mean, the standard deviation, and percentiles of the down time.

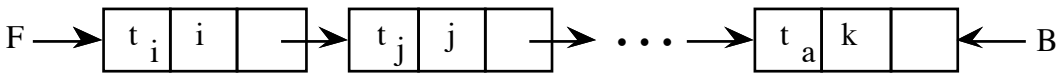


Figure 5.1: The linked list for the repairman's queue.

A more efficient method would be to maintain a linked list representing those machines which are in the repairman's queue. This linked list is shown in figure 5.1. Each node contains the following two data elements: a) time of arrival at the repairman's queue, and b) index number of the machine. The nodes are linked so that to represent the FIFO manner in which the machines are served. Thus, the first node, pointed by F, represents the machine currently in service. If a machine arrives at the repairman's queue, a new node will be appended after the last node, pointed to by B. The total down time of a machine is calculated at the instance when the machine departs from the repairman.

This is equal to the master clock's value at that instance minus its arrival time. In this fashion we can obtain a sample of observations. Each observation is the duration of a down time. Such a sample of observations can be analyzed statistically.

Another statistic of interest in is the probability distribution of the number of broken down machines. In this case, the maximum number of broken down machines will not exceed m, the total number of machines. In view of this, it suffices to maintain an array with m + l locations. Location i will contain the total time during which there were i broken down machines. Each time an arrival or a departure occurs, the appropriate location of the array is updated. At the end of the simulation run, the probability p(n) that there are n machines down is obtained by dividing the contents of the nth location by T, the total simulation time.

As another example on how to collect endogenously created data, let us consider the simulation model of the token-based access scheme. The simulation program can be enhanced so that each node is associated with a two-dimensional array, as shown in figure 5.2. In each array, the first column contains the arrival times of packets, and the second column contains their departure time. When a packet arrives at the node, its arrival time is stored in the next available location in the first column. When the packet departs from the node, its departure time is stored in the corresponding location of the second column. Thus, each array contains the arrival and departure times of all packets that have been through the node. Also, it contains the arrival times of all the packets currently waiting in the queue. Instead of keeping two columns per node, one can keep one column. When a packet arrives, its arrival time is stored in the next available location. Upon departure of the packet, its arrival time is substituted by its total time in the system. These arrays can be processed later in order to obtain various statistics per node.

An alternative approach is to maintain a linked list per node, as described above for the machine interference problem. Each time a packet is transmitted, its total time is calculated by subtracting the current master clock value from it arrival time stored in the linked list. This value is then stored into a single array containing all durations of the packets in the order in which they departed.

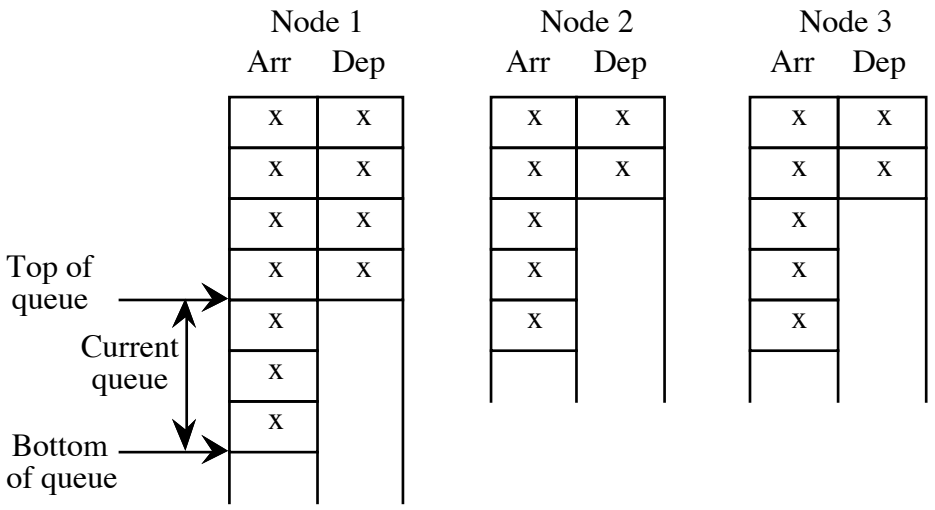


Figure 5.2: Data structure for the token-based access scheme simulation model.

5.3 Transient state vs. steady-state simulation

In general, a simulation model can be used to estimate a parameter of interest during the *transient state* or the *steady state*.

Let us consider the machine interference problem. Let us assume that one is interested in obtaining statistics pertaining to the number of broken down machines. The simulation starts by assuming that the system at time zero is at a given state. This is known as the *initial condition*. Evidently, the behaviour of the system will be affected by the particular initial condition. However, if we let the simulation run for a long period, its statistical behaviour will eventually become independent of the particular initial condition. In general, the initial condition will affect the behavior of the system for an initial period of time, say T. Thereafter, the simulation will behave statistically in the same way whatever the initial condition. During this initial period T, the simulated system is said to be in a transient state. After period T is over, the simulated system is said to be in a steady state.

5.3.1 Transient-state simulation

One may be interested in studying the behavior of a system during its transient state. In

this case, one is mostly interested in analyzing problems associated with a specific initial starting condition. This arises, for example, if we want to study the initial operation of a new plant. Also, one may be forced to study the transient state of a system, if this system does not have a steady state. Such a case may arise when the system under study is constantly changing.

5.3.2 Steady-state simulation

Typically, a simulation model is used to study the steady-state behaviour of a system. In this case, the simulation model has to run long enough so that to get away from the transient state. There are two basic strategies for choosing the initial conditions. The first strategy is to begin with an *empty system*. That is, we assume that there are no activities going on in the system at the beginning of the simulation. The second strategy is to make the initial condition to be as representative as possible of the typical states the system might find itself in. This reduces the duration of the transient period. However, in order to set the initial conditions properly, an a priori knowledge of the system is required.

One should be careful about the effects of the transient period when collecting endogenously created data. For, the data created during the transient period are dependent on the initial condition. Two methods are commonly used to remove the effects of the transient period. The first one requires a very long simulation run, so that the amount of data collected during the transient period is insignificant relative to the amount of data collected during the steady state period. The second method simply requires that no data collection is carried out during the transient period. This can be easily implemented as follows. Run the simulation model until it reaches its steady state, and then clear all statistical accumulations (while leaving the state of the simulated system intact!). Continue to simulate until a sufficient number of observations have been obtained. These observations have all been collected during the steady-state. The second method is easy to implement and it is quite popular.

The problem of determining when the simulation system has reached its steady state is a difficult one. A simple method involves trying out different transient periods $T_1,T_2,T_3,...,T_k$, where $T_1<T_2<T_3<...<T_k$. Compile steady-state statistics for each

simulation run. Choose T_i so that for all the other intervals greater than T_i, the steady-state statistics do not change significantly. Another similar method requires to compute a moving average of the output and to assume steady-state when the average no longer changes significantly over time.

5.4 Estimation techniques for steady-state simulation

Most of the performance measures that one would like to estimate through simulation are related to the probability distribution of an endogenously created random variable. The most commonly sought measures are the mean and the standard deviation of a random variable. Also, of importance is the estimation of percentiles of the probability distribution of an endogenously created random variable.

For instance, in the machine interference problem one may be interested in the distribution of the down time. In particular, one may settle for the mean and standard deviation of the down time. However, percentiles can be very useful too. From the management point of view, one may be interested in the 95% percentile of the down time. This is the down time such that only 5% of down times are greater than it. Percentiles often are more meaningful to the management than the mean down time.

5.4.1 Estimation of the confidence interval of the mean of a random variable

Let x_1, x_2,..., x_n be n consecutive endogenously obtained observations of a random variable. Then

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{5.1}$$

is an unbiased estimate of the true population mean, i.e., the expectation of the random variable. In order to obtain the confidence interval for the sample mean we have to first

estimate the standard deviation. If the observations $x_1, x_2, \ldots, x_n$ are independent of each other, then

$$s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 \tag{5.2}$$

or, using the short-cut formula

$$s^2 = \frac{1}{n-1} \left(\sum_{i=1}^{n} x_i^2 - \frac{(\Sigma x_i)^2}{n} \right)$$

and, therefore, we obtain the confidence interval

$$\left(\bar{x} - 1.96 \frac{s}{\sqrt{n}} , \bar{x} + 1.96 \frac{s}{\sqrt{n}} \right)$$

at 95% confidence. The confidence interval provides an indication of the error associated with the sample mean . It is a very useful statistical tool and it should be always computed. Unfortunately, quite frequently it is ignored. The confidence interval tells us that the true population mean lies within the interval 95% of the time. That is, if we repeat the above experiment 100 times, 95% of these times, on the average, the true population mean will be within the interval.

The theory behind the confidence interval is very simple indeed. Observations x_1, x_2, $\ldots x_n$ are assumed to come from a population known as the *parent population* whose mean μ we are trying to estimate. Let σ^2 be the variance of the parent population. The distribution that $\bar{x}$ follows is known as the *sampling distribution*. Using the Central Limit Theorem we have that $\bar{x}$ follows the normal distribution $N(\mu, \sigma/\sqrt{n})$, as shown in figure 5.3. Now, let us fix points a and b in this distribution so that 95% of the observations (an observation here is a sample mean $\bar{x}$) fall in-between the two points. Points a and b are

symmetrical around μ. The areas (-∞, a) and (b, +∞) account for 5% of the total distribution. Using the table of the standard normal distribution, we have that a is 1.96 standard deviation below μ, i.e., $a=\mu-1.96\sigma/\sqrt{n}$. Likewise, $b=\mu+1.96\sigma/\sqrt{n}$. Now, if we consider an arbitrary observation $\bar{x}$, this observation will lie in the interval [a,b] 95% of the time. That is, its distance from μ will be less than $1.96\sigma/\sqrt{n}$ 95% of the time. Therefore, μ will lie in the interval

$$(\bar{x} - 1.96\,\sigma\sqrt{n},\ \bar{x} + 1.96\,\sigma\sqrt{n})$$

95% of the time. If σ is not known, one can use in its place the sample standard deviation s.

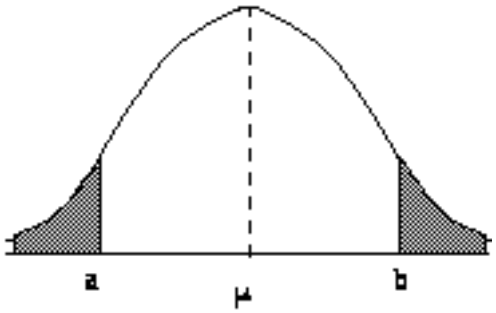


Figure 5.3: The normal distribution.

The value 95% is known as the *confidence*. In general, a confidence interval can be calculated for any value of confidence. Most typical confidence values are 99%, 95% and 90%. For each value, points a and b are calculated from the table of the standard normal distribution. If the sample size is small (less than 30), then we can construct similar confidence intervals, but points a and b will be obtained using the t distribution, i.e.,

$$(\bar{x} - t_{.95}\frac{s}{\sqrt{n}}\ ,\ \bar{x} + t_{.95}\frac{s}{\sqrt{n}}\ )$$

with (n-1) degrees of freedom.

In general, the observations x_1, x_2,..., x_n that one obtains endogenously from a simulation model are correlated. For instance, the down time of a machine depends on the down time of another machine that was ahead of it in the repairman's queue. In the presence of correlated observations, the above expression (5.2) for the variance does not hold. Expression (5.1) for the mean holds for correlated or uncorrelated observations. The correct procedure, therefore, for obtaining the confidence interval of the sample mean is to first check if the observations x_1, x_2,..., x_n are correlated. If they are not, one can proceed as described above. If the observations are correlated, then one has to use a special procedure to get around this problem. Below, we discuss the following four procedures for estimating the variance of correlated observations:

a. Estimation of the autocorrelation function.
b. Batch means.
c. Replications.
d. Regenerative method.

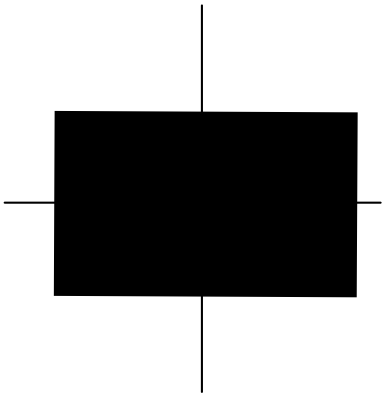
a. Estimation of the autocorrelation coefficients

Let X and Y be two random variables. Let μ_X and μ_Y be the expectation of X and Y respectively. Also, let σ_X^2 and σ_Y^2 be the variance of X and Y respectively. We define the *covariance* between X and Y to be

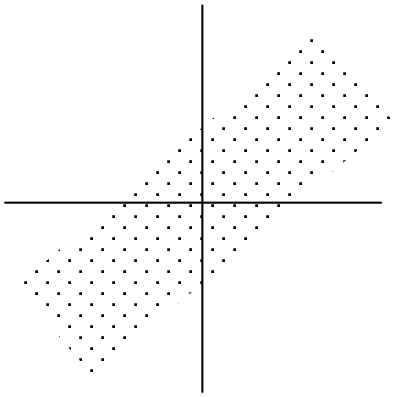
$$\text{Cov}(X,Y) = E[(X-\mu_X)(Y-\mu_Y)]$$

$$= E(XY) - \mu_X \mu_Y$$

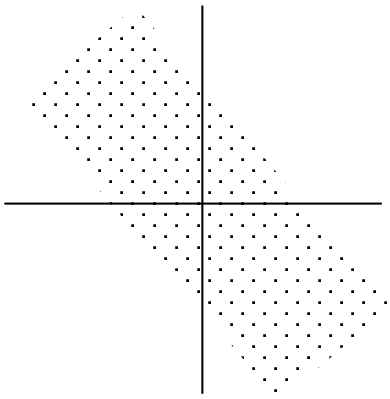
This statistic reflects the dependency between X and Y. If X and Y are uncorrelated, then Cov(X,Y) = 0. If Cov (X,Y) > 0, then X and Y are positively correlated, and if Cov (X,Y) < 0, then they are negatively correlated. If Y is identical to X, then Cov(X,X) = σ_X^2.



(i) uncorrelated, i.e. Cov(X,Y) = 0.



(ii) positive correlation, i.e. Cov (X,Y) > 0.



(iii) negative correlation, i.e. Cov (X,Y) <0.

Figure 5.4: The three cases of correlation.

Let us assume that we have obtained actual observations of the random variables X and Y in the form of (x,y). Then, the scatter diagram can be plotted out. The scatter diagrams given in figure 5.4 shows the three cases of correlation between X and Y mentioned above.

The Cov (X,Y) may take values in the region (-∞, +∞). Also, it is not dimensionless, which makes its interpretation troublesome. In view of this, the *correlation* ρ_{XY} defined by

$$\rho_{XY} = \frac{Cov(X,Y)}{\sigma_X \sigma_Y}$$

is typically used as the measure of dependency between X and Y. It can be shown that $-1 \le \rho_{XY} \le 1$. If ρ_{XY} is close to 1, then X and Y are highly positively correlated. If ρ_{XY} is close to -1, then they are highly negatively correlated. Finally, if $\rho_{XY}=0$, then they are independent from each other.

Now, let us assume we have n observations $x_1, x_2, ..., x_n$. We form the following n-1 pairs of observations: $(x_1,x_2), (x_2,x_3), (x_3,x_4), ..., (x_i,x_{i+1}), ..., (x_{n-1},x_n)$. Now, let us regard the first observation in each pair as coming from a variable X and the second observation as coming from a variable Y. Then, in this case ρ_{XY} is called the *autocorrelation* or the *serial correlation coefficient*. It can be estimated as follows:

$$r_1 = \frac{\sum_{i=1}^{n-1} (x_i - \bar{X})(x_{i+1} - \bar{Y})}{\sqrt{\sum_{i=1}^{n-1} (x_i - \bar{X})^2 \sum_{i=1}^{n-1} (x_{i+1} - \bar{Y})^2}},$$

where $\bar{X} = \frac{1}{n-1} \sum_{i=1}^{n-1} x_i$ and $\bar{Y} = \frac{1}{n-1} \sum_{i=2}^{n} x_i$. For n reasonably large, ρ_{XY} can be approximated by

$$r_1 = \frac{\sum_{i=1}^{n-1} (x_i - \bar{X})(x_{i+1} - \bar{X})}{\sum_{i=1}^{n-1} (x_i - \bar{X})^2}$$

where $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} x_i$ is the overall mean.

We refer to the above estimate of ρ_{XY} as r_1 in order to remind ourselves that this is the correlation between observations which are a distance of 1 apart. This auto-correlation is often referred to as *lag 1 autocorrelation*. In a similar fashion, we can obtain the *lag k autocorrelation*, that is the correlation between observations which are a distance k apart. This can be calculated using the expression:

$$r_k = \frac{\sum_{i=1}^{n-k} (x_i - \bar{X})(x_{i+k} - \bar{X})}{\sum_{i=1}^{n-1} (x_i - \bar{X})^2})$$

In practice, the autocorrelation coefficients are usually calculated by computing the series of autocovariances $R_0, R_1, ...$, where R_k is given by the formula

$$R_k = \frac{1}{n}\sum_{i=1}^{n-k} (x_i - \bar{X})(x_{i+k} - \bar{X}) \qquad (5.3)$$

We then compute r_k as the ratio

$$r_k = \frac{R_k}{R_0}, \qquad (5.4)$$

where $R_0 = \sigma^2$. Typically, r_k is not calculated for values of k greater than about n/4.

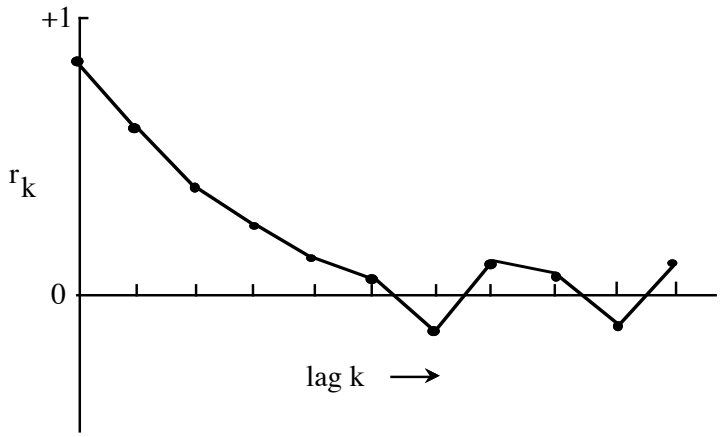


Figure 5.5: A correlogram with short-term correlation.

A useful aid in interpreting the autocorrelation coefficients is the *correlogram*. This is a graph in which r_k is plotted against lag k. If the time series exhibits a short-term correlation, then its correlogram will be similar to the one shown in figure 5.5. This is characterized by a fairly large value of r_1 followed by 2 or 3 more coefficients which, while significantly greater than zero, tend to get successively smaller. Values of r_k for longer lags tend to be approximately zero. If the time series has a tendency to alternate, with successive observations on different sides of the overall mean, then the correlogram also tends to alternate as shown in figure 5.6.

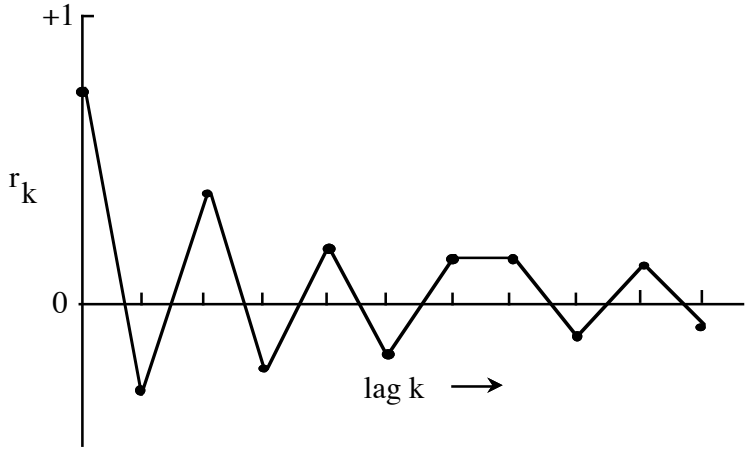


Figure 5.6: An alternating correlogram.

Let us now return to our estimation problem. Having obtained a sample of n observations $x_1, x_2, ..., x_n$, we calculate the autocorrelation coefficients using the expressions (5.3) and (5.4). Then, the variance can be estimated using the expression

$$s^2 = s_X^2[1 + 2\sum_{k=1}^{n/4}(1 - \frac{k}{n} r_k)]$$

where s_X^2 is the standard deviation given by

$$s_X^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{X})^2$$

and $\overline{X} = \frac{1}{n}\sum_{i=1}^{n} x_i$.

b. Batch means

This is a fairly popular technique. It involves dividing successive observations into batches as shown in figure 5.7. Each batch contains the same number of observations. Let the batch size be equal to b. Then batch 1 contains observations $x_1, x_2, ..., x_b$, batch 2

| $x_1, x_2, ..., x_b$ | $x_{b+1}, x_{b+2}, ..., x_{2b}$ | . . . | $x_{kb+1}, x_{kb+2}, ..., x_{(k+1)b}$ |

batch 1 *batch 2* *batch k*

Figure 5.7: The batch means method.

contains $x_{b+1}, x_{b+2}, ..., x_{2b}$, and so on. The observations close to batch 2 are likely to be correlated with the observations in batch 2 which are close to batch 1. Also, the observations in batch 2 which are close to batch 3 are likely to be correlated with those in batch 3 which are close to batch 2, and so on. Let $\overline{X}_i$ be the sample mean of the

observations in batch i. If we choose b to be large enough, then the sequence $\overline{X}_1$, $\overline{X}_2$, ..., $\overline{X}_k$ can be shown that it is approximately uncorrelated. Therefore, we can treat these means as a sample of independent observations and calculate their mean and standard deviation. We have

$$\bar{\bar{X}} = \frac{1}{k} \sum_{i=1}^{k} \bar{X}_i$$

$$s^2 = \frac{1}{k-1} \sum_{i=1}^{k} (\overline{X}_i - \bar{\bar{X}})^2 .$$

Therefore, for large k (i.e. k ≥30), we obtain the confidence interval

$$(\bar{\bar{X}} - 1.96\frac{s}{\sqrt{k}} , \bar{\bar{X}} + 1.96\frac{s}{\sqrt{k}}).$$

For small k, we can construct our confidence interval using the t distribution.

In general, the batch size b has to be large enough so that the successive batch means are not correlated. If b is not large, then the successive batch means will be correlated, and the above estimation procedure will yield severely biased estimates. An estimate of b can be obtained by plotting out the correlogram of the observations $x_1, x_2, ..., x_n$, which can be obtained from a preliminary simulation run. We can fix b so that it is 5 times the smallest value b' for which $r_{b'}$ is approximately zero.

c. Replications

Another approach to constructing a confidence interval for a mean is to replicate the simulation run several times. Suppose we make n replications, each resulting to n observations as follows:

replication 1: $x_{11}, x_{12}, ..., x_{1m}$

replication 2: $x_{21}, x_{22}, ..., x_{2m}$

.

.

.

replication n: $x_{nm}, x_{n2}, ..., x_{nm}$

For each sample, we can construct the sample mean

$$\bar{X}_i = \frac{1}{m} \sum_{j=1}^{n} x_{ij} .$$

We can then treat the sample means $\bar{X}_1, \bar{X}_2, ..., \bar{X}_n$ as a sample of independent observations, thus obtaining

$$\bar{\bar{X}} = \frac{1}{n} \sum_{i=1}^{n} \bar{X}_i$$

$$s^2 = \frac{1}{n} \sum_{i=1}^{n} (X_i - \bar{X})^2.$$

Using the above statistics we can construct our confidence interval.

The problems that arise with this approach are: a) decide on the length of each simulation run, i.e., the value m, and b) decide on the length of the transient period. One of the following two approaches can be employed:

Start each simulation run with different values for the seeds of the random number generators. Allow the simulation to reach its steady state and then collect the sample observations. Repeat this procedure n times. In order to obtain n samples, therefore, we have to run n independent simulations, each time having to allow the simulation to reach its steady state.

Alternatively, we can run one very long simulation. Allow first the simulation to reach its steady state, and then collect the first sample of observations. Subsequently, instead of terminating the simulation and starting all over again, we extend the simulation run in order to collect the second sample of observations, then the third sample and so on. The advantage of this method is that it does not require the simulation to go through a transient period for each sampling period. However, some of the observations that will be collected at the beginning of a sampling period will be correlated with observations that will be collected towards the end of the previous sampling period.

The replication method appears to be similar to the batch means approach. However, in the batch means method, the batch size is relatively small and, in general, one collects a large number of batches. In the above case, each sampling period is very large and one collects only a few samples.

d. Regenerative method

The last two methods described above can be used to obtain independent or approximately independent sequences of observations. The method of independent replications generates independent sequences through independent runs. The batch means method generates approximately independent sequences by breaking up the output generated in one run into successive subsequences which are approximately independent. The regenerative method produces independent subsequences from a single run. Its applicability, however, is limited to cases which exhibit a particular probabilistic behaviour.

Let us consider a single server queue. Let t_0, t_1, t_2,... be points at which the simulation model enters the state where the system is empty. Such time instances occur when a customer departs and leaves an empty system behind. Let t_0 be the instance when the simulation run starts assuming an empty system. The first customer that will arrive will see an empty system. During its service, other customers may arive thus forming a queue. Let t_1 be the point at which the last customer departs and leaves an empty system. That is, t_1 is the time instance where the server becomes idle. This will repeat itself as shown in figure 5.8. It is important to note that the activity of the queue in the interval (t_i,

t_{i+1}) is independent of its activity during the previous interval (t_{i-1},t_i). That is to say, the probability of finding customer n in the interval (t_i, t_{i+1}) does not depend on the number of customers in the system during the previous interval. These time instances are known as *regeneration points*. The time between two such points is known as the *regeneration cycle* or *tour*. The queue-length distribution observed during a cycle is independent of the distribution observed during the previous cycles. The same applies for the density probability distribution of the total time a customer spends in the system.

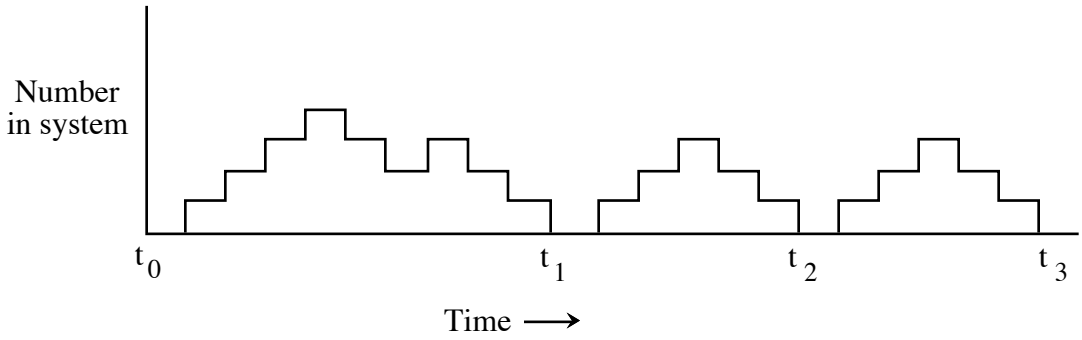


Figure 5.8: Regeneration points.

The regeneration points identified above in relation with the queue-length (or waiting time) probability distribution are the time points when the system enters the empty state. Regeneration points identified with time instances when the system enters another state, can be also obtained if the service time is exponentially distributed. For instance, the time instances when the system contains 5 customers are regeneration points. Due to the memoryless property of the service time, the process repeats itself probabilistically.

In the case of the machine interference problem, regeneration points related to the repairman's queue-length probability, or waiting time in the queue, can be identified with the time instances when the repairman is idle. Such time instances occur when the last machine is repaired and the system enters the state where all machines are operational. Due to the assumption that operational times are exponentially distributed, the process

repeats itself probabilistically following the occurrence of this state. Other states can be used to identify regeneration points, assuming exponentially distributed repair times.

Now, let us assume that we are interested in estimating the mean value of a random variable X. Let $(x_{i1}, x_{i2}, ..., x_{in_i})$ be a sequence of realizations of X during the ith regeneration cycle occurring between t_i and t_{i+1} regeneration points. Let n_i be the total number of observations. This sequence of observations will be independent from the one obtained during the (i+1)st cycle. Thus, the output is partitioned into independent sequences. In the batch means case, the output was partitioned into approximately independent sequences. However, the number of observations in each batch was constant equal to batch size. In this case, the number of observations in each cycle is a random variable.

Due to the nature of the regeneration method, there is no need to discard observations at the beginning of the simulation run. The simulation starts by initially setting the model to the regeneration state. Then the simulation is run for several cycles, say M cycles. Let

$$Z_i = \sum_{j=1}^{n_i} x_{ij}$$

be the sum of all realizations of X in the ith cycle. Then a point estimate of E(X) can be obtained using the expression:

$$\bar{X} = \frac{\frac{1}{M}\sum_{i=1}^{M} Z_i}{\frac{1}{M}\sum_{i=1}^{M} n_i} .$$

That is, E(X) is estimated using the ratio E(Z)/E(N), where Z is a random variable indicating the sum of all observations of $\overline{X}$ in a cycle, and N is a random variable indicating the number of observations in a cycle.

$\overline{X}$ is not an unbiased estimator of E(X). That is E($\overline{X}$) $\neq$ E(X). However, it is consistent. That is, $\overline{X} \rightarrow$ E(X) as M $\rightarrow \infty$. Now, in order to construct a confidence interval of $\overline{X}$, we note that $\overline{X}$ is obtained as the ratio of two random variables. The following two methods can be used to construct confidence intervals for a ratio of two random variables: the central limit theorem, and the jackknife method.

The central limit theorem

Let σ_{12} = Cov (Z, N), σ_{11}^2= Var(Z), and σ_{22}^2= Var(N). Also, let V=Z-NE(X). That is, for cycle i we have $V_i=Z_i-n_iE(X)$. Then, the V_i's are independent and identically distributed with a mean E(V) = 0. This because

$$\begin{aligned} E(V) &= E(Z - NE(X)) \\ &= E(Z) - E(N)E(X) \\ &= 0. \end{aligned}$$

The variance σ_v^2 of the V_i's is as follows:

$$\begin{aligned} E(V - E(V))^2 &= E(V^2) \\ &= E(Z - NE(X))^2 \\ &= E(Z^2 - 2ZNE(X) + N^2E(X)^2) \\ &= E(Z^2) - 2E(X)E(ZN) + E(N^2)E(X)^2. \end{aligned}$$

Since $Var(Y) = E(Y^2) - E(Y)^2$ we have that

$$\sigma_v^2 = \sigma_{11}^2 + E(Z)^2 - 2E(X)E(ZN) + E(X)^2\,\sigma_{22}^2 + E(N)^2E(X)^2$$

$$= \sigma_{11}^2 + E(X)^2\,\sigma_{22}^2 - 2(E(X)E(ZN) + E(Z)^2 + E(N)^2E(X)^2.$$

Given that E(Z) = E(N)E(X) we have

$$\begin{aligned}\sigma_v^2 &= \sigma_{11}^2 + E(X)^2\,\sigma_{22}^2 - 2E(X)E(ZN) + 2E(N)^2E(X)^2 \\ &= \sigma_{11}^2 + E(X)^2\,\sigma_{22}^2 - 2E(X)[E(ZN) - E(X)E(N)^2 \\ &= \sigma_{11}^2 + E(X)^2\,\sigma_{22}^2 - 2E(X)[E(ZN) - E(Z)E(N)]\end{aligned}$$

or

$$\sigma_v^2 = \sigma_{11}^2 + E(X)^2\,\sigma_{22}^2 - 2E(X)\sigma_{12}$$

since

$$\begin{aligned}\mathrm{Cov}(Y_1, Y_2) &= E(Y_1 - E(Y_1))(Y_2 - E(Y_2)) \\ &= E(Y_1Y_2 - E(Y_1)Y_2 - E(Y_2)Y_1 + E(Y_1)E(Y_2)) \\ &= E(Y_1Y_2) - E(Y_1)E(Y_2) - E(Y_2)E(Y_1) + E(Y_1)E(Y_2) \\ &= E(Y_1Y_2) - E(Y_1)E(Y_2).\end{aligned}$$

By the central limit theorem, we have that as M increases $\bar{V}$ becomes normally distributed with a mean equal to 0 and a standard deviation equal to $\sqrt{\sigma_v^2 / M}$, where

$$\begin{aligned}\bar{V} &= \frac{1}{M}\sum_{i=1}^{M} V_i \\ &= \frac{1}{M}\sum_{i=1}^{M} (Z_i - n_i E(X)) \\ &= \frac{1}{M}\sum_{i=1}^{M} Z_i - E(X)\,\frac{1}{M}\sum_{i=1}^{M} n_i \\ &= \bar{Z} - E(X)\bar{N}\ .\end{aligned}$$

Hence,

$$\bar{Z} - E(X)\bar{N} \sim N\left(0, \sqrt{\frac{\sigma_v^2}{M}}\right)$$

or

$$\frac{\bar{Z} - E(X)\bar{N}}{\sqrt{\frac{\sigma_v^2}{M}}} \sim N(0, 1)$$

Dividing by N we obtain that

$$\frac{\bar{Z}/\bar{N} - E(X)}{(1/\bar{N})\sqrt{\frac{\sigma_v^2}{M}}} \sim N(0, 1).$$

Therefore, we obtain the confidence interval

$$\frac{\bar{Z}}{\bar{N}} \pm 1.96 \frac{\sqrt{\sigma_v^2 / M}}{\bar{N}}$$

Now, it can be shown that $s_v^2 = s_Z^2 - 2\bar{X} R_{Z,N} + \bar{X}^2 s_N^2$ is an estimate of σ^2. Therefore, it can be used in place of σ_v^2 in the above confidence interval.

The above method can be summarized as follows:

1. Run the simulation for M cycles and obtain $Z_1, Z_2, Z_3, ..., Z_M$ ($n_1, n_2, n_3, ..., n_M$.)
2. Estimate $\bar{Z}$, $\bar{N}$ and set $\bar{X} = \bar{Z} / \bar{N}$.
3. Estimate

$$s_Z^2 = \frac{1}{M-1} \sum_{i=1}^{M} (z_i - Z^2)$$

$$s_N^2 = \frac{1}{M-1}\sum_{i=1}^{M}(n_i - N^2)$$

$$s_{Z,N}^2 = \frac{1}{M-1}\sum_{i=1}^{M}(z_i - Z)(n_i - N).$$

The jacknife method

This method constitutes an alternative method to obtaining a confidence interval of a ratio of two random variables. It also provides a means of obtaining a less biased point estimator of the mean of the ratio of two random variables, sine the classical point estimator is usually biased.

Let X,Y be two random variables. Suppose we want to estimate ϕ=E(Y)/E(X) from the data $y_1, y_2, ..., y_n$ and $x_1, x_2, ..., x_n$, where the y_i's are i.i.d., the x_i's are i.i.d. and $Cov(y_i,x_j)\neq 0$ for $i\neq j$. The classical point estimator of ϕ is $\phi_c = \overline{Y} / \overline{X}$ and its confidence interval can be obtained as shown above. The jacknife point and interval estimator can be constructed as follows. Let

$$\theta_g = n\,\hat{\phi}_C - (n-1)\frac{\sum_{j\neq g} y_j}{\sum_{j\neq g} x_j}, \quad g = 1,2, ...,n.$$

The jacknife point estimator of ϕ is

$$\hat{\phi}_J = \sum_{g=1}^{n}\theta_g/n .$$

This is, in general, less biased than $\hat{\phi}_C$. Let

$$\sigma_J^2 = \sum_{g=1}^{n}\frac{(\theta_g - \phi_j)^2}{n-1}.$$

Then, it can be shown that

$$\hat{\phi}_J \sim N(\phi, \sqrt{\frac{\sigma_J^2}{n}}), \text{ as } n \to \infty.$$

That is, we can obtain the confidence interval at 95%

$$(\hat{\phi}_J - 1.96 \frac{\sigma_J}{\sqrt{n}}, \hat{\phi}_J + 1.96 \frac{\sigma_J}{\sqrt{n}}).$$

The difficulty in using the regenerative method is that real-world simulations may not have regeneration points. If they do happen to have regeneration points, the expected regeneration cycle may be too large so that only a few cycles may be simulated.

5.4.2 Estimation of other statistics of a random variable

So far we considered estimation techniques for constructing a confidence interval of the mean of an endogenously created random variable. Other interesting statistics related to the probability distribution of a random variable are:

a. Probability that a random variable lies within a fixed interval.
b. Percentiles of the probability distribution.
c. Variance of the probability distribution.

Below, we examine ways of estimating the above statistics.

a. Probability that a random variable lies within a fixed interval

The estimation of this type of probability can be handled exactly the same way as the estimation of the mean of a random variable. Let I be the designated interval. We want to estimate $p = \Pr(X \in I)$ where X is an endogenously created random variable. We generate M replications of the simulation. For each replication i we collect N

observations of X. Let v_i be the number of times X was observed to lie in I. Then, $p_i = v_i/N$ is an estimate of probability p. Thus,

$$\bar{p} = \frac{1}{M}\sum_{i=1}^{M} p_i$$

and

$$s^2 = \frac{1}{M-1}\sum_{i=1}^{M} (p_i - \bar{p})^2.$$

As before, $(\bar{p} - p)/(s/\sqrt{M}) \sim N(0,1)$ if M is large.

We observe, that the estimation of p requires M independent replications, each giving rise to one realization of p. Other methods examined in section 5.4 of this Chapter can be used in order to remove unwanted autocorrelations. For instance, instead of replications, the batch means method or the regenerative method can be used.

Alternatively, the estimation of p can be seen as estimating a population mean. Let $Y_i=1$ if ith realization of X belongs to I. Otherwise, $Y_i=0$. Then, in one replication we have that $(1/N)\Sigma Y_i$ is equal to V_i/N or p_i. Thus, the techniques developed in section 5.4 of this Chapter can be employed to estimate p.

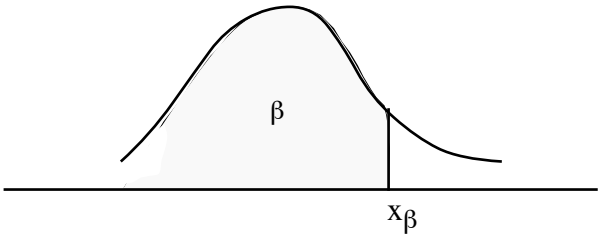


Figure 5.9: Percentile x_β.

b. Percentile of a probability distribution

This is a very important statistic that is often ignored in favour of the mean of a random variable X. Management, sometimes, is not interested in the mean of a particular random variable. For instance, the person in charge of a web service may not be interested in its

mean response time. Rather, he or she may be interested in "serving as many as possible as fast as possible". More specifically, he or she may be interested in knowing the 95th percentile of the response time. That is, a value such that the response time of the real time system is below it 95% of the time.

In general, let us consider a probability density function f(x). The 100βth percentile is the smallest value x_β such that $f(x_\beta) < \beta$. That is, the area from $-\infty$ to x_β under f(x) is less or equal to β as shown in figure 5.9. Typically, there is interest in the 50th percentile (median) $x_{0.50}$ or in extreme percentiles such as $x_{0.90}$, $x_{0.95}$, $x_{0.99}$.

We are interested in placing a confidence interval on the point estimator of x_β of a distribution of a random variable X. Let us assume independent replications of the simulation. Each replication yields N observations, having allowed for the transient period. For each replication i, let x_{i1}, x_{12}, ..., x_{1N} be the observed realizations of X. Now, let us consider a reordering of these observations y_{i1}, y_{12}, ..., y_{1N} so that $y_{ij}<y_{i,j+1}$. Then, the 100βth percentile $x_\beta^{(i)}$ for the ith replication is observation y_{ik} where k=Nβ, if Nβ is an integer, or k= $\lfloor N\beta \rfloor + 1$ if Nβ is not an integer ($\lfloor x \rfloor$ means the largest integer which is less or equal to x). For instance, if we have a sample of 50 observations ordered in an ascending order, then the 90th percentile is the observation number 0.90·50 = 45. The 95th percentile is $\lfloor 50x0.95 \rfloor + 1 = \lfloor 47.5 \rfloor + 1 = 47 + 1 = 48$.

Hence

$$\bar{x}_\beta = \frac{1}{M}\sum_{i=1}^{M} x_\beta^{(i)}$$

and

$$s^2 = \frac{1}{M-1}\sum_{i=1}^{M} (x_\beta^{(i)} - \bar{x}_\beta)^2$$

Confidence intervals can now be constructed in the usual manner.

The estimation of extreme percentiles requires long simulation runs. If the runs are not long, then the estimates will be biased. The calculation of a percentile requires that a) we store the entire sample of observations until the end of the simulation, and b) that we order the sample of observations in an ascending order. These two operations can

be avoided by constructing a frequency histogram of the random variable on the fly. When a realization of the random variable becomes available, it is immediately classified into the appropriate interval of the histogram. Thus, it suffices to keep track of how many observations fall within each interval. At the end of the replication, the 100βth percentile can be easily picked out from the histogram. Obviously, the accuracy of this implementation depends on the chosen width of the intervals of the histogram.

Finally, we note that instead of independent replications of the simulation, other methods can be used such as the regeneration method.

c. Variance of the probability distribution

Let us consider M independent replications of the simulation. From each replication i we obtain N realizations of a random variable $x_{i1},x_{i2},...,x_{iN}$, after we allow for the transient period. We have

$$\bar{\mu}_i = \frac{1}{N}\sum_{j=1}^{N} x_{ij}$$

and

$$\bar{\bar{\mu}} = \frac{1}{M}\sum_{i=1}^{M} \mu_i$$

is the grand sample mean. Hence,

$$s_i^2 = \frac{1}{N}\sum_{j=1}^{N} (x_{ij} - \bar{\mu})^2$$

$$= \frac{1}{N}\left[\sum_{j=1}^{N} x_{ij}^2 - 2\bar{\mu}_i\bar{\bar{\mu}} + \bar{\bar{\mu}}^2\right].$$

As the point estimate of the variance σ^2 we take

$$s_i^2 = \frac{1}{M}\sum_{i=1}^{M} s_i^2$$

$$= \frac{1}{M}\sum_{i=1}^{M}\left(\frac{1}{N}\sum_{j=1}^{N} x_{ij}^2\right) - \frac{1}{M}\frac{2}{N}\sum_{i=1}^{M}\bar{\mu}_i\bar{\bar{\mu}} + \frac{2}{M}\bar{\bar{\mu}}^2$$

$$= \frac{1}{M}\sum_{i=1}^{M}\left(\frac{1}{N}\sum_{j=1}^{N} x_{ij}^2\right) - \bar{\bar{\mu}}^2$$

The estimates of s_i^2 are all functions of $\bar{\bar{\mu}}$. Thus, they are not independent. A confidence interval can be constructed by jacknifing the estimator s^2. Let σ_i^2 be an estimate of σ^2 with the ith replication left out. That is,

$$\bar{\sigma}_i^2 = \frac{1}{M-1}\sum_{j\neq i}\left(\frac{1}{N}\sum_{i=1}^{N} x2\right) - \frac{1}{M-1}\sum_{j\neq i}\bar{\mu}_j^2 .$$

Define $z_i = Ms^2 - (M-1)\,\bar{\sigma}_i^2$. Then,

$$Z = \frac{1}{M}\sum_{i=1}^{M} Z_i$$

$$s_Z^2 = \frac{1}{M-1}\sum_{i=1}^{M}(Z_i - \bar{Z})^2$$

and a confidence interval can be constructed in the usual way.

Alternatively, we can obtain a confidence interval of the variance by running the simualtion only once, rather than using repications, and then calculating the standard deviation assuming that the successive observations are independent! This approach is correct when the sample of observations is extremely large.

5.5 Estimation techniques for transient-state simulation

The statistical behaviour of a simulation during its transient state depends on the initial condition. In order to estimate statistics of a random variable X during the transient state one needs to be able to obtain independent realizations of X. The only way to get such independent observations is to repeat the simulation. That is, employ the replications

technique discussed above in section 5.4. Each independent simulation run has to start with the same initial condition.

As an example, let us assume that we want to estimate the 95th percentile of the probability distribution of a random variable X. This can be achieved by simply replicating the simulation times. For each replication we can obtain an estimate $x_{.95}$. A confidence interval can be obtained as shown in 5.4. Each replication will be obtained by starting the simulation with the same initial condition, and running the simulation during its transient period. This, of course, requires advanced knowledge of the length of the transient period. Furthermore, the pseudo-random numbers used in a replication have to be independent of those used in previous replications. This can be achieved by using a different seed for each replication. The difference between two seeds should be about 10,000.

5.6 Pilot experiments and sequential procedures for achieving a required accuracy

So far, we discussed techniques for generating confidence intervals for various statistics of an endogenously generated random variable. The expected width of the confidence interval is, in general, proportional to $1/\sqrt{N}$, where N is the number of i.i.d. observations used. Obviously, the larger the value of N, the smaller the width confidence interval. (This is defined as half the confidence interval.) For instance, in order to halve the width of the confidence interval of the mean of a random variable, N has to be increased by four times so that $1/\sqrt{4N} = (1/2)(1/\sqrt{N})$. In general, the accuracy of an estimate of a statistic depends on the width of the confidence interval. The smaller the width, the higher is the accuracy.

Quite frequently one does not have prior information regarding the value of N that will give a required accuracy. For instance, one does not know the value of N that will yield a width equal to 10% of the estimate. Typically, this problem is tackled by conducting a pilot experiment. This experiment provides a rough estimate of the value of N that will yield the desired confidence interval width. An alternative approach is the sequential method. That is, the main simulation experiment is carried out continuously.

Periodically, a test is carried out to see if the desired accuracy has been achieved. The simulation stops the first time it senses that the desired accuracy has been achieved. Below, we examine the methods of independent replications and batch means.

5.6.1 Independent replications

This discussion is applicable to transient and steady state estimation. Let us assume that we want to estimate a statistic θ of a random variable using independent replications. A pilot experiment is first carried out involving N_1 replications. Let $\hat{\theta}_1$ be a point estimate of θ and let Δ_1 be the width of its confidence interval. We assume that the width of the confidence interval is required to be approximately less or equal to $0.1\hat{\theta}$. If $\Delta_1 \leq 0.1\hat{\theta}_1$, then we stop. If $\Delta_1 > 0.1\hat{\theta}_1$, then we need to conduct a main experiment involving N_2 replications where, $N_2 = (\Delta_1/0.1\hat{\theta}_1)^2 N_1$ appropriately rounded off to the nearest integer. Let $\hat{\theta}_2$ and Δ_2 be the new estimate and width based on N_2 replications. Due to randomness, the N_2 observations will not exactly yield a confidence interval width equal to $0.1\hat{\theta}_2$. We note that it is not necessary to run a new experiment involving N_2 replications. Rather, one can use the N_1 replications from the pilot experiment, and then carry out only $N_2 - N_1$ additional replications.

The above approach can be implemented as a sequential procedure as follows. The simulation program is modified to conduct N^* independent replications. The point estimate $\hat{\theta}_1$ and the width Δ_1 of its confidence interval are calculated. If $\Delta_1 \leq 0.1\hat{\theta}_1$, then the simulation stops. Otherwise, the simulation proceeds to carry out N^* additional replications. The new point estimate $\hat{\theta}_2$ and width Δ_2 are constructed based on the total $2N^*$ replications. The simulation will stop if $\Delta_2 \leq 0.1\hat{\theta}_2$. Otherwise, it will proceed to generate another N^* replication and so on. N^* can be set equal to 10.

5.6.2 Batch means

This discussion is obviously applicable to steady state estimations. Using the pilot

experiment approach, one can run the simulation for k_1 batches. From this, one can obtain $\hat{\theta}_1$ and Δ_1. If $\Delta_1 \leq 0.1\hat{\theta}_1$, then the desired accuracy has been achieved. Otherwise, k_2-k_1 additional batches have to be simulated, where $k_2=(\Delta_1/0.1\hat{\theta}_1)^2 k_1$. These additional batches can be obtained by simply re-running the simulator for k_2-k_1 batches.

Alternatively, the simulation program can be enhanced to include the following sequential procedure. The simulation first runs for k^* batches. Let $\hat{\theta}_1$ and Δ_1 be the point estimate and the width of its confidence interval respectively. The simulation stops if $\Delta_1 \leq 0.1\hat{\theta}_1$. Otherwise, it runs for k^* additional batches. Now, based on the $2k^*$ batches generated to this point, $\hat{\theta}_2$ and Δ_2 are constructed. If $\Delta_2 \leq 0.1\hat{\theta}_2$, then the simulation stops. Otherwise, it runs for another k^* batches, and so on.

Computer Assignments

1. Consider the machine interference problem. Modify the simulation model to carry out the following tasks:
   a. First remove the print statement that prints out a line of output each time an event occurs. Set-up a data structure to collect information regarding the amount of time each machine spends being broken down, i.e., waiting in the queue and also being repaired (see section 2 of Chapter 3).
   b. Augment your simulation program to calculate the mean and standard deviation. Then, run your simulation for 1050 observations (i.e., repairs). Discard the first 50 observations to account for the transient state. Based on the remaining 1000 observations, calculate the mean and the standard deviation of the time a machine is broken down.
   c. The calculation of the standard deviation may not be correct, due to the presence of autocorrelation. Write a program (or use an existing statistical package) to obtain a correlogram based on the above 1000 observations. (You may graph the correlogram by hand!) Based on the correlogram, calculate the batch size. Implement the batch means approach in your program and run your program for

31 batches. Disregard the first batch, and use the other 30 batch means to construct a confidence interval of the mean time a customer spends in the system.

d. Using the batch means approach, implement a sequential procedure for estimating the mean time a machine is broken down (see section 5.6.2).

e. Implement a scheme to estimate the 95th percentile of the time a machine is broken down.

2. Consider the token-based access scheme. Change the inter-arrival times to 20, 25, and 30 instead of 10, 15, and 20 that you used before (which lead to an unstable system). Then, modify the simulation model to carry out the following tasks:

   a. First remove the print statement that prints out a line of output each time an event occurs. Augment your simulation to record how much time each packet spends in a queue, from the time when it joins the queue to the time it is transmitted out. We will refer to this time as the *response* time. Record this information in the order in which the packets arrive.

      Augment your simulation program to calculate the mean and standard deviation of the response times collected during the simulation. Run your simulation for a total of 3100 packet departures (to all three queues). Discard the first 100 observations to account for the transient state. Based on the remaining observations calculate the mean and the standard deviation of the response time.

   b. The calculation of the standard deviation may not be correct, due to the presence of autocorrelation. Write a program (or use an existing statistical package) to calculate the correlogram based on the response times collected from the above task a. (You may graph the correlogram by hand.) Based on the correlogram, calculate the batch size. Implement the batch means approach in your program and run your program for 31 batches. Disregard the first batch, and use the other 30 batch means to construct a confidence interval of the mean time a customer spends in a queue.

   c. Using the batch means approach, implement a sequential procedure for estimating the mean time a packet spends in a queue (see section 5.6.2).

d. Implement a scheme to estimate the 95th percentile of the time a packet spends in a queue.

3. Consider the two-stage manufacturing system. Modify the simulation model to carry out the following tasks. (Make sure that the input values you use lead to a stable system, i.e. the utilization of the first server is less than 100%.)
   a. First remove the print statement that prints out a line of output each time an event occurs. Set-up a data structure to collect information regarding the amount of time each customer spends in the system, i.e. from the time it arrives to stage 1 to the time it departs from stage 2.
   b. Augment your simulation program to calculate the mean and standard deviation. Then, run your simulation for 1050 observations (i.e., customer arrivals). Discard the first 50 observations to account for the transient state. Based on the remaining 1000 observations, calculate the mean and the standard deviation of the time a customer spends in the system.
   c. The calculation of the standard deviation may not be correct, due to the presence of autocorrelation. Write a program (or use an existing statistical package) to obtain a correlogram based on the above 1000 observations. (You may graph the correlogram by hand!) Based on the correlogram, calculate the batch size. Implement the batch means approach in your program and run your program for 31 batches. Disregard the first batch, and use the other 30 batch means to construct a confidence interval of the mean time a customer spends in the system.
   d. Using the batch means approach, implement a sequential procedure for estimating the mean time a customer spends in the system (see section 5.6.2).
   e. Implement a scheme to estimate the 95th percentile of the time a customer spends in the system.

.

CHAPTER 6:

VALIDATION OF A SIMULATION MODEL

Validation of a simulation model is a very important issue that is often neglected. How accurately does a simulation model (or, for that matter, any kind of model) reflect the operations of a real-life system? How confident can we be that the obtained simulation results are accurate and meaningful?

In general, one models a system with a view to studying its performance. This system under study may or may not exist in real-life. Let us consider, for instance, a communications equipment manufacturer who is currently designing a new communications device, such as a switch. Obviously, the manufacturer would like to know in advance if the new switch has an acceptable performance. Also, the manufacturer would like to be able to study various alternative configurations of this new switch so that to come up with a good product. The performance of such a switch can be only estimated through modelling since the actual system does not exist. The question that arises here is how does one make sure that the model that will be constructed is a valid representation of the system under study?

Let us consider another example involving a communication system already in operation. Let us assume that it operates nearly at full capacity. The management is considering various alternatives for expanding the system's capacity. Which of these alternatives will improve the system's performance at minimum cost? Now, these alternative configurations do not exist. Therefore, their performance can be only evaluated by constructing a model, say using simulation techniques. The standard method is to construct a model of the existing system. Then, change the model appropriately in order to analyze each of the above alternatives. The model of the existing system can be

validated by comparing its results against actual data obtained from the system under investigation. However, there is no guarantee that when altering the simulation model so that to study one of the above alternatives, this new simulation model will be a valid representation of the configuration under study!

The following checks can be carried out in order to validate a simulation model.

1. *Check the pseudo-random number generators*. Are the pseudo-random numbers uniformly distributed in (0,1) and do they satisfy statistical criteria of independence? Usually, one takes for granted that a random number generator is valid.
2. *Check the stochastic variate generators*. Similar statistical tests can be carried out for each stochastic variate generator built into a simulation model.
3. *Check the logic of the simulation program*. This is a rather difficult task. One way of going about it is to print out the status variables, the future event list, and other relevant data structures each time an event takes place in the simulation. Then, one has to check by hand whether the data structures are updated appropriately. This is a rather tedious task. However, using this method one can discover possible logical errors and also get a good feel about the simulation model.
4. *Relationship validity*. Quite frequently the structure of a system under study is not fully reflected down to its very detail in a simulation model. Therefore, it is important that the management has the opportunity to check whether the model's assumptions are credible.
5. *Output validity*. This is one of the most powerful validity checks. If actual data are available regarding the system under study, then these data can be compared with the output obtained from the simulation model. Obviously, if they do not compare well, the simulation model is not valid.

Computer assignments

1. Consider the machine interference problem. This problem can be also analyzed using queueing theory. Set up the following validation scheme. Obtain exact values of the

mean time a machine is broken down using queueing theory results for various values of the mean operational time. Let it vary, for instance, from 1 to 50 so that to get a good spread. For each of these values, obtain an estimate of the mean down time of a machine using your simulation model. Graph both sets of results. Be sure to indicate the confidence interval at each simulated point. Compare the two sets of results.

Note: Remove the iterative option from the simulation. By now, you should have a good idea as to how many batches and the batch size you will need in order to get the required accuracy. So, use these values in all the experiments.

2. Consider the token-based access scheme. Carry out the following validations of your simulation.

   a. Modify your code so that the token is always in queue 1 and it never visits the other queues. (You can do this by setting T to a very large number so that the time-out never occurs during the life of the simulation, and do not let the token change queues if queue 1 is empty.) Then, queue 1 will be the only queue that will be served, while the other queues will grow in length for ever. Assume an inter-arrival mean time of 20 and a mean service time of 5 for all packets. Use the batch means method to estimate the mean waiting time in queue 1. Compare your results to the mean waiting time of an M/M/1 queue with the same arrival and service rates. The analytic result should be within the confidence interval obtained from the simulation. (The only departures are from queue 1, and consequently your estimation procedure will automatically give the mean waiting time of queue 1. Also, make sure that the data structures for the other queues do not become very large so that to cause memory problems.)
   b. Remove the changes you made in the code in task a above so that to restore the code in its original state. Assume that T is very small, i.e., T=0.001, and that the switch over time is very small as well, i.e., it is equal to 0.001. Then, in this case the queues will be served in a round-robin fashion where a maximum of one customer is served from each queue, and the overhead time due to switch-over time is almost zero. Assume that the inter-arrival time to each queue is equal to 20

and the mean service time is the same for all packets and it is equal to 2. Use the batch means method to estimate the mean waiting time in the system. Compare your results to the mean waiting time of an M/M/1 queue with an arrival rate equal to the sum of the arrival rates to the three queues, i.e., λ=0.15, and a mean service time of 2. The M/M/1 mean waiting time should be within the confidence intervals of your simulated mean waiting time.

Note: By now, you should have a good idea as to how many batches and the batch size you will need in order to get the required accuracy. So, use these values in all the experiments.

3. Consider the two-stage manufacturing system. As the buffer capacity of the second queue increases, the probability that server 1 will get blocked upon service completion due to queue 2 being full will decrease, and eventually will become zero. At that moment, the two-stage manufacturing system is equivalent to two infinite capacity queues in tandem. Now, if we assume that the percent of time each machine is down is very small, i.e. the mean downtime divided by the sum of the mean operational time and the mean downtime is less than 2%, then the service time at each queue can be safely approximated by an exponential distribution. Each queue then becomes an M/M/1 queue, and the mean waiting time in the two-stage manufacturing system is equal to the mean waiting time W_1 in the first queue plus the mean waiting time W_2 in the second queue. The mean waiting time in the first queue is given by the expression: $W_1 = 1/[(\beta_1\mu_1/(\alpha_1 + \beta_1)) - \lambda]$, where μ_1, $1/\alpha_1$ and $1/\beta_1$ is the service rate, the mean operational time and the mean downtime respectively at the first queue, and λ is the arrival rate to the two-stage manufacturing system. Likewise, the mean waiting time in the second queue is given by the expression: $W_2 = 1/[(\beta_2\mu_2/(\alpha_2 + \beta_2)) - \lambda]$ where μ_2, $1/\alpha_2$ and $1/\beta_2$ is the service rate, the mean operational time and the mean downtime respectively at the second queue. Based on the above, carry out the following experiment.

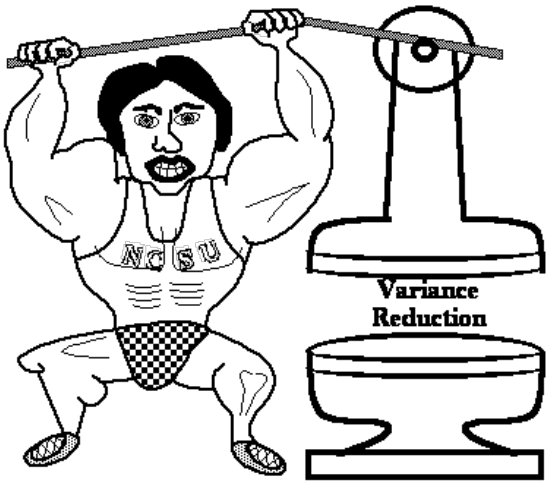
   Run the simulation to calculate the mean waiting time in the two-stage manufacturing system for different buffer capacities of queue 2. Keep λ, μ_1 $1/\alpha_1$, $1/\beta_1$,

μ_2, $1/\alpha_2$ and $1/\beta_2$the same for all the simulations. (Remember to fix the mean downtime and mean operational times so that the mean downtime divided by the sum of the mean operational time and the mean downtime is less than 2%). Plot the mean waiting time against the buffer capacity of queue 2. Show that this curve approximates W_1+W_2 as calculated using the above expressions for the M/M1 queue.

Note: Remove the iterative option for the simulation. By now, you should have a good idea as to how many batches and the batch size you will need in order to get the required accuracy. So, use these values in all the experiments.

CHAPTER 7:

VARIANCE REDUCTION TECHNIQUES



7.1 Introduction

In Chapter 6, it was mentioned that the accuracy of an estimate is proportional to $1/\sqrt{n}$, where n is the sample size. One way to increase the accuracy of an estimate (i.e., reduce the width of its confidence interval) is to increase n. For instance, the confidence interval width can be halved if the sample size is increased to 4n. However, large sample sizes required long simulation runs which, in general, are expensive. An alternative way to increasing the estimate's accuracy is to reduce its variance. If one can reduce the variance of an endogenously created random variable without disturbing its expected value, then the confidence interval width will be smaller, for the same amount of simulation.

Techniques aiming at reducing the variance of a random variable are known as *Variance Reduction Techniques*. Most of these techniques were originally developed in connection with Monte Carlo Techniques.

Variance reduction techniques require additional computation in order to be implemented. Furthermore, it is not possible to know in advance whether a variance reduction technique will effectively reduce the variance in comparison with straightforward simulation. It is standard practice, therefore, to carry out pilot simulation runs in order to get a feel of the effectiveness of a variance reduction technique and of the additional computational cost required for its implementation.

In this Chapter, we will examine two variance reduction techniques, namely, a) the antithetic variates technique and (b) the control variates technique.

7.2 The antithetic variates technique

This is a very simply technique to use and it only requires a few additional instructions in order to be implemented. No general guarantee of its effectiveness can be given. Also, it is not possible to know in advance how much of variance reduction can be achieved. Therefore, a small pilot study may be useful in order to decide whether or not to implement this technique.

Let X be an endogenously created random variable. Let $x_1^{(1)}, x_2^{(1)}, ..., x_n^{(1)}$ be n i.i.d. realizations of X obtained in a simulation run. Also, let $x_1^{(2)}, x_2^{(2)}, ..., x_n^{(2)}$ be n i.i.d. observations of X obtained in a second simulation run. Now, let us define a new random variable

$$z_i = (x_i^{(1)} + x_i^{(2)})/2, \quad i = 1, 2, ..., n. \tag{7.1}$$

More specifically, let $Z=(X^{(1)}+X^{(2)})/2$, where $X^{(i)}$, i=1,2, indicates the random variable X as observed in the ith simulation run. We have

$$E(Z)= E\left(\frac{X(1)+X(2)}{2}\right)$$

$$= \frac{1}{2} [E(X^{(1)}) + E(X^{(2)})]$$

$$= E(X)$$

seeing that the expected value of $X^{(1)}$ or $X^{(2)}$ is that of X. Thus, the expected value of this new random variable Z is identical to that of X. Now, let us examine its variance. We have

$$\text{Var}(Z) = \text{Var}\left(\frac{X^{(1)}+X^{(2)}}{2}\right)$$

$$= \frac{1}{4} [\text{Var}(X^{(1)}) + \text{Var}(X^{(2)}) + 2\text{Cov}(X^{(1)}, X^{(2)})].$$

Remembering that $\text{Var}(X^{(1)})=\text{Var}(X^{(2)})=\text{Var}(X)$, we have that

$$\text{Var}(Z) = \frac{1}{2} (\text{Var}(X) + \text{Cov}\ (X^{(1)}, X^{(2)})) .$$

Since $\text{Cov}(X,Y)=\rho\sqrt{\text{Var}(X)\text{Var}(Y)}$, we have that

$$\text{Var}(Z) = \frac{1}{2} \text{Var}(X) (1 + \rho), \tag{7.2}$$

where ρ is the correlation between $X^{(1)}$ and $X^{(2)}$.

In order to construct an interval estimate of E(X), we use random variable Z. Observations z_i are obtained from (7.1) and the confidence interval is obtained using (7.2). As will be seen below, by appropriately constructing the two samples

$$x_1^{(1)}, x_2^{(1)}, ..., x_n^{(1)} \text{and } x_1^2, x_2^2, ..., x_n^2$$

we can cause Var(Z) to become significantly less than Var(X). This is achieved by causing ρ to become negative. In the special case where the two sets of observations

$x_1^1, x_2^1, ..., x_n^1$ and $x_1^2, x_2^2, ..., x_n^2$ are independent of each other, we have that ρ=0. Hence, Var(Z)=Var(X)/2.

The antithetic variates technique attempts to introduce a negative correlation between the two sets of observations. As an example, let us consider a simulation model of a single server queue, and let X and Y indicate the waiting time in the queue and the interarrival time respectively. If Y is very small, then customers arrive faster and, therefore, the queue size gets larger. The larger the queue size, the more a customer has to wait in the queue, i.e. X is larger. On the other hand, if Y is large, then customers arrive slower and, hence, the queue size gets smaller. Obviously, the smaller the queue size, the less a customer has to wait in the queue, i.e., X is small. Therefore, we see that X and Y can be negatively correlated.

This negative correlation between these two variables can be created in a systematic way as follows. Let F(t) and G(S) be the cumulative distribution of the inter-arrival and service time respectively. Let r_i and v_i be pseudo-random numbers. Then, $t_i=F^{-1}(r_i)$ and$s_i=G^{-1}(v_i)$ are an interarrival and a service variate. These two variates can be associated with the ith simulated customer. An indication of whether the queue is tending to increase or decrease can be obtained by considering the difference $d_i=t_i-s_i$. This difference may be positive or negative indicating that the queue is going through a busy or slack period respectively. Now, let us consider that in the second run, we associate pseudo-random number r_i' and v_i' with the ith simulated customer, so that

$$d_i' = t_i' - s_i' \text{ (where } t_i'=F^{-1}(r_i') \text{ and } s_i' = G^{-1}(v_i'))$$

has the opposite sign of d_i. That is, if the queue was going through a slack (busy) period in the first run at the time of the ith simulated customer, now it goes through a busy (slack) period. It can be shown that this can be achieved by simply setting $r_i' = 1-r_i$ and $v_i' = 1-v_i$.

In this example, we make use of two controllable variables, Y_1 and Y_2, indicating the interarrival time and the service time respectively. These two random variables are strongly correlated with X, the waiting time in the queue. $Y_j(1)$ and $Y_j(2)$, j=1,2 can be

negatively correlated by simply using the compliment of the pseudo-random numbers used in the first run.

This technique can be implemented as follows. Simulate the single server queue, and let $x_1^{(1)}, x_2^{(1)}, ..., x_n^{(1)}$ be n i.i.d observations of X. Re-run the simulation, thus replicating the results, using pseudo-random numbers $(r_i, v_i)=(1-r, 1-v_i)$. Let $x_1^2, x_2^2, ..., x_n^2$ be realizations of X. Construct the interval estimate of E(X) using the random variable Z as described above. Obviously, the correlation between the two samples of observations is as good as the correlation between Y_j^1 and Y_j^2, j=1,2.

The antithetic variates technique, as described above, was implemented in simulation of an M/M/1 queue. The random variable X is the time a customer spends in the system. The i.i.d. observations of X were obtained by sampling every 10th customer. The results given in table 7.1 were obtained using straight simulation. Using the antithetic variates technique, we obtained a confidence interval of 13.52±1.76. We note that the antithetic variates techniques were employed using two sets of observations each of size equal to 300, i.e., a total of 600 observations. From table 7.1, we see that a similar width was obtained using a sample size of n=1800. Figure 7.2 shows the actual values for the original sample, the sample obtained using the antithetic variates and Z. We see, that the two samples of observations are fairly negatively correlated. Also, we observe that the Z values are all close to the mean, indicating that their variance is small.

Sample size n	Confidence interval
600	13.86 ± 3.46
900	13.03 ± 2.70
1200	13.11 ± 2.30
1500	12.82 ± 1.99
1800	12.86 ± 1.84

Table 7.1: Straight simulation of an M/M/1 queue.

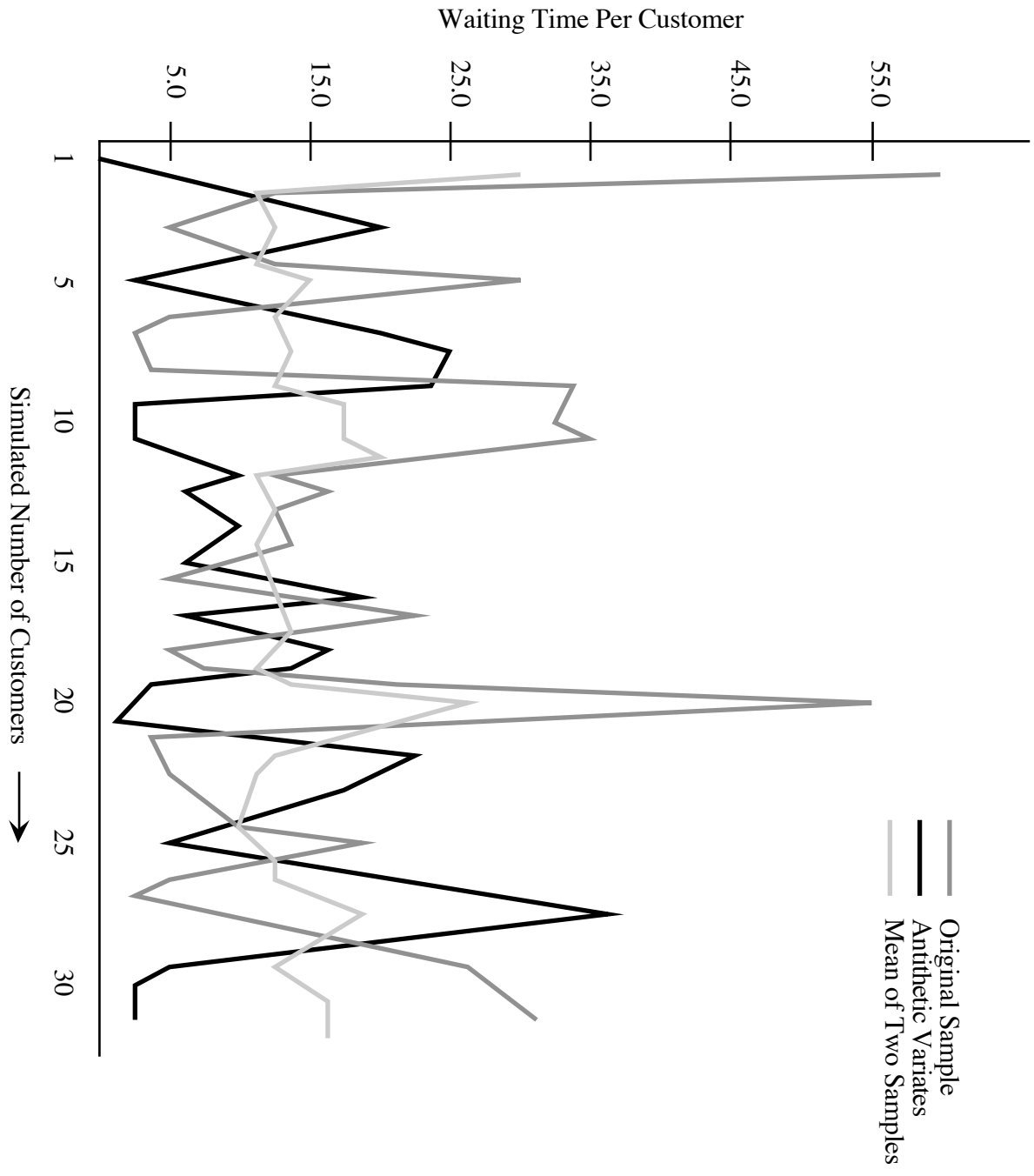


Figure 7.1: Antithetic variates technique applied to an M/M/1 queue.

In the above example, the antithetic variates technique worked quite well. However, this should not be construed that this method always works well. In particular, in the following example, an M/M/2 queuing system was simulated. Table 7.2 and figure 7.2 show that in this case there is little benefit to be gained from using the antithetic variates technique.

Type of Simulation	Sample Size	Mean	Standard Dev.	Conf. Interval	Standard Error	Standard Error as a % of Mean	Cost
Straight	400	18.85	9.70	+/-3.31	1.69	8.96	$.68
Straight	800	16.00	8.60	2.07	1.06	6.61	.72
Standard Antithetic	800	17.23	6.73	2.30	1.17	6.79	.97
Straight	1600	16.04	9.67	1.64	0.84	5.22	.79
Standard Antithetic	1600	15.69	5.98	1.44	0.74	4.69	1.03
Straight	2400	16.48	9.82	1.36	0.69	4.21	.88
Standard Antithetic	2400	15.84	6.36	1.25	0.64	4.01	1.09
Straight	3000	15.92	9.59	1.19	0.61	3.81	.93
Standard Antithetic	3000	16.11	6.96	1.17	0.60	3.72	1.12

Table 7.3: Straight simulation and antithetic variates techniques for an M/M/2 queue.

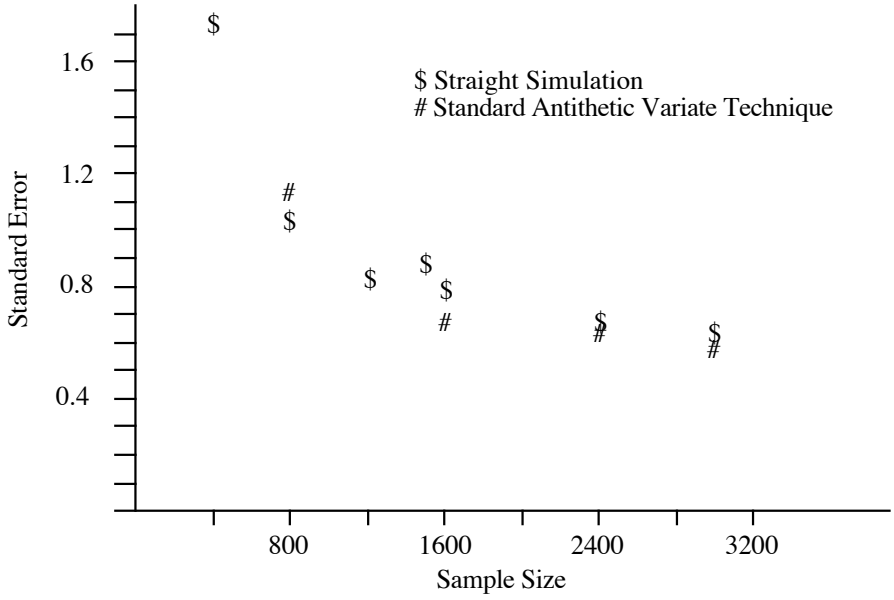


Figure 7.2: Standard error plotted against sample size for straight simulation and antithetic variates techniques for an M/M/2 queue.

7.3 The control variates technique

This method is otherwise known as the method of *Concomitant Information*. Let X be an endogenously created random variable whose mean we wish to estimate. Let Y be another endogenously created random variable whose mean is known in advance. This is known as the *control* variable. Random variable Y is strongly correlated with X. We have the following two cases.

a. *X and Y are negatively correlated:*

Define a new random variable Z so that Z=X+Y-E(Y). We have

$$E(Z)=E(X+Y-E(Y))=E(X),$$
$$Var(Z) = Var(X) + Var(Y) + 2\ Cov\ (X, Y).$$

Since X and Y are negatively correlated, we have that Cov(X,Y)<0. Therefore, if $Var(Y) - 2\ |Cov(X,Y)| < 0$ then, a reduction in the variance of Z has been achieved.

b. *X and Y are positively correlated:*

Define Z=X-Y+E(Y). Then

$$E(Z) = E(X - Y + E(Y)) = E(X).$$
$$Var(Z) = Var(X) + Var(Y) - 2|Cov(X,Y)|$$

Therefore, if $Var(Y) - 2|Cov(X,Y)| < 0$ then a reduction in the variance of Z has been achieved.

Let us consider the example of a single server queue mentioned in section 7.2. Let X be the time a customer spends in the system. Then X is negatively correlated with the

random variable Y representing the inter-arrival time. (It is also positively correlated with the random variable representing the service time.) Let x_1, x_2, ..., x_n be n i.i.d. observations of X. Likewise, let y_1, y_2, ..., y_n be n observations of Y. (These observations are i.i.d. by nature of the simulation model.) y_i is the inter-arrival time associated with the x_i observation. Let

$$z_i = x_i + y_i - E(Y), \quad i=1,2,..., n.$$

Then we can construct a confidence interval for the estimate of E(X) using $\bar{Z} + 1.96 \frac{s_Z}{\sqrt{n}}$ where

$$\bar{Z} = \frac{1}{n} \sum_{i=1}^{n} z_i$$

and

$$s_Z^2 = \frac{1}{n-1} \sum_{i=1}^{n} (z_i - Z)^2 .$$

More generally, random variable Z can be obtained using

$$Z = X - a\,(Y - E(Y)),$$

where a is a constant to be estimated and Y is positively or negatively correlated to X. Again, we have E(Z)=E(X). Also,

$$Var(Z) = Var(X) + a^2\, Var(Y) - 2a\, Cov(X,Y)$$

so that Z has a smaller variance than X if

$$a^2\, Var(Y) - 2a\, Cov(X,Y) < 0.$$

We select a so that to minimize the right-hand side given in the above expression. We have

$$2a\ Var(Y) - 2Cov(X,Y) = 0$$

or

$$a^* = \frac{Cov\ (X,Y)}{Var\ (Y)}$$

Now, substituting into the expression for Var(Z) we have

$$Var(Z) = Var(X) - \frac{[Cov(X,Y)]^2}{Var(Y)}$$

$$= (1-\rho_{XY}^2)\ Var(X).$$

Thus, we always get a reduction in the variance of Z for the optimal value of a, provided that X and Y are correlated. The determination of a^* requires a priori knowledge of the Var(Y) and Cov (X,Y). Sample estimates can be used in order to approximately obtain a^*.

The definition of Z can be further generalized using multiple control variables, as follows

$$Z = X - \sum_{i=1}^{m} a_i(Yi - E(Y_i)) ,$$

where a_i, i=1,2,...,m, are any real numbers. In this case,

$$Var(Z) = Var(X) + \sum_{i=1}^{m} a_i^2\ Var(Y_i) - 2 \sum_{i=1}^{m} a_i\ Cov(X,Y_i) + 2 \sum_{i=2}^{m} \sum_{j=1}^{i-1} a_i a_j\ Cov(Y_i,Y_j).$$

Computer assignments

1. Consider the machine interference problem. Carry out the following tasks:
   a. Implement the antithetic variance reduction technique.
   b. Implement the control variates technique.
   c. Compare the above two variance reduction techniques against straight simulation. In particular, compare the three techniques for various sample sizes. Observe how the variance and the standard error expressed as a percentage of the mean, educe as the sample size goes up. Contrast this against the execution cost (if available).

CHAPTER 8:

SIMULATION PROJECTS

8.1 An advance network reservation system – blocking scheme

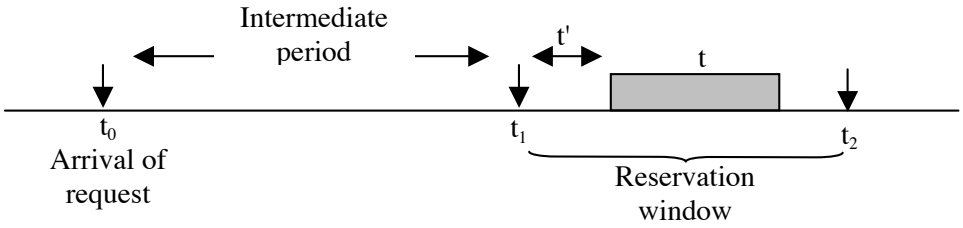
The objective of this project is to simulate an advance reservation system that is used to reserve bandwidth for connections in a mesh network of LSRs in advance of the time that the connections are required. In this project, we will assume *reservations with blocking*. That is, a reservation request is blocked if network resources cannot be reserved within a certain window of time defined by the reservation request.

Project description

You have to simulate a fully connected network of N LSRs and an advance reservation system in which reservation requests arrive dynamically. Requests are independent of each other and each one specifies a required bandwidth and a time interval in the future for which it needs a path to be reserved between two LSRs. The requests arrive long before the actual start of the requested reservation interval. This period between the arrival of the request and the start of the interval is called the *intermediate period*.

Each LSR is connected to every other LSR by two simplex links in opposite directions. The bandwidth of all the links is the same and finite and will be one of the inputs to the simulation. A path can be established between two LSRs directly over the link connecting them or via other LSRs over multiple links. For this simulation we will limit the path to be over a maximum of two links, i.e., either one link or two links. Every time a link is reserved for an interval, its available bandwidth is reduced appropriately during that interval.

A reservation request, arriving at time t_0 specifies the source LSR x, destination LSR y, duration of the *reservation interval* t, and a *window* $[t_1, t_2]$ within which the reservation has to be scheduled, as shown in the figure below. For simplicity, we will assume that the window is twice the length of the reservation interval, i.e., $t_2 - t_1 = 2t$. Also, we will assume that time is slotted and the length of a slot is equal to some given duration. All values for t, t_1, t_2 are therefore integers. In this simulation, we assume that a slot is equal to 5 minutes.



For each request, you have to check if each of the link(s) along the one-hop or along any of the two-hop paths from x to y has sufficient available bandwidth during the requested window. Specifically, you have to check if a path can be reserved for any interval t within a window t_1 to t_2. Thus, you have to check if a path can be reserved for the request during any interval $[t_1+t' , t_1+t'+t]$, where t' varies from 0 to t. The request is accepted if a path can be reserved from x to y during any such interval for the requested bandwidth. Otherwise, it is blocked.

The simulation model

The following are the inputs to the simulation:

1. N, number of LSRs.
2. Link capacity.
3. Mean inter-arrival time of reservation requests.
4. Maximum intermediate period.
5. Maximum duration of reservation interval.
6. Total simulated number of arrivals.

As mentioned above, we assume that time is slotted, and each slot is equal to five minutes. Thus, in the simulation time scale, the first five minutes are represented by 1, the next five minutes by 2, and so on. Now, if we decied to set the maximum duration of a reservation interval to two hours, then the value of the input 'maximum duration of reservation interval' will be 48 (i.e., 12 five minute slots/hour * 2 hours). If requests arrive on the average every forty minutes, then the value of the 'mean inter-arrival time of reservation requests' will be 8. If reservations can be done up to five hours in advance then the value of 'maximum intermediate period' will be 60.

The link capacity is the maximum available bandwidth of a link when it is idle. As mentioned above, all links are assumed to have the same capacity. For simplicity, we will assume that the link capacity of each link is 1 Gbps, and that the bandwidth requested each time a reservation request arrives is also 1 Gbps.

Initialization

You must first define the maximum simulation time (in slots), as this will be used to dimension an integer array for each link in the network. A safe estimate is: 1.5 * Total simulated number of arrivals * mean inter-arrival period. This array will be used to indicate the link's available bandwidth at each time slot, and it should be initialized to the link capacity. Thus each element of the array gives the available bandwidth on the link, which will be either 1 or 0, at the time instant given by the index of the array. For example link03[5] is the available bandwidth on the link from LSR 0 to LSR 3 at simulation time = 5. There will be as many such integer arrays as the number of links.

Next, you have to generate a list of all possible paths from each LSR to every other LSR. As an example, in a mesh network of 5 LSRs, the possible paths from LSR 0 to LSR 3 are: {0-3}, {0-1, 1-3}, {0-2, 2-3} and {0-4, 4-3}. Recall that we will only consider paths of up to a maximum of two links. Consequently, you do not need to generate paths longer than 2 links. Save the paths in a data structure of your choice. Alternatively, you may chose to generate these paths on the fly during the execution of each reservation request. This method is, of course, more CPU intensive.

Arrivals of reservation requests

Arrivals of reservation requests occur in a Poisson fashion. That is, the inter-arrival times are exponentially distributed with a mean inter-arrival time specified as input to the simulation. The actual arrival times can be generated as:

arrival_time = previous_arrival_time - (mean inter-arrival time) * $\log_e$(random()).

where random() is a pseudo random number generator that yields a random number between 0 and 1. Such a generator is readily available in most programming languages. Occasionally, these generators do not have good statistical behaviour. Consequently, it may not be a bad idea to download one from a reliable mathematical package, such as Matlab. Also, be aware that a random number generator requires an initial value, known as *seed*. Some of the random number generators are automatically seeded. For debugging purposes, it is advisable that you always use the same seed, so that to recreate the same sequence of events. In view of this, you should use a random number generator that you can seed yourself. (Make sure you read the rules of the random number generator as to what type of number it will accept as a seed).

The expression: - (mean inter-arrival time) * $\log_e$(random()) yields a positive value which is a random sample from an exponential distribution with a mean equal to the "mean inter-arrival time". This is basically, a random value of the next inter-arrival time, which if added to the previous one will give you the time instance in the future when the arrival will occur.

Each reservation request specifies the reservation parameters: source LSR x, destination LSR y, beginning of the reservation interval t_1, and duration of the reservation interval t. (The reservation window is set equal to 2*t, and the required bandwidth is 1 Gbps.) You have to generate these values as described below before processing an arrival request.

1. The source LSR is generated randomly as: x = N * random(). (Number the LSRs from 0 to N).
2. The destination LSR is generated in the same way, i.e., y = N * random(). You should ensure that x and y are not the same.
3. The intermediate period is generated as: t_1 = arrival_time + (maximum intermediate period) * random().

4. Duration of the reservation interval is generated as: t = (maximum duration of reservation interval) * random().

Now check if a path can be reserved from x to y. There are several search schemes you can employ to find such a path, and below, we describe one such scheme. Let us assume a mesh network of 5 LSRs and that x = 0, y = 3. Then, the one-link path is {0-3}, and the two-link paths are{0-1, 1-3}, {0-2, 2-3} and {0-4, 4-3}. Check the one-link path to find a free period of time of length t starting at time t_1+t', where t'=0,1,2,…,t. Failing to find a free period of length t proceed to check each two-link path, selected randomly, to find a free period of time of length t starting at time t_1+t', where t'=0,1,2,…,t.. The request is blocked, if no free period of duration t is found on the one-link and all the two-link paths. If a free period is found, the links on that path are reserved for the time period t starting at t_1+t'.

The main logic of the simulation program is as follows:

1. Generate next arrival
2. Check to see whether it can be scheduled using scheme 1 or 2
3. Update variables accordingly
4. Update statistics
5. Is the total number of simulated arrivals equal to the value of "total simulated number of arrivals" specified at input time? If no, go back to step 1. If yes, stop simulation and calculate final statistics, as described below.

Statistical estimation

You will use the simulation model to calculate a) the probability that a reservation request is blocked, and b) the mean delay to schedule a reservation request, for a given set of input values. In addition, you will have to calculate confidence intervals.

A confidence interval provides an indication of the error associated with the sample mean. To calculate the confidence interval you will have to obtain about 30 different estimates of the blocking probability under the same input values. An easy way to do this is to use the *batch means* method. Run your simulation for a large number of

arrivals, and then group your results as follows. Let us say that you run the simulation for 9,000 arrivals, and that you block your results in groups of 300 arrivals. That is, you run your simulation for the first 300 arrivals and you calculate the blocking probability by dividing the number of times an arrival was blocked over the total number of arrivals, i.e., 300. You store this number in an array, zero the variable where you keep count of the number of blocked requests, and continue to simulate for another 300 arrivals without changing anything in your simulation. You then calculate the new blocking probability over the second set of 300 observations. You repeat this process until you have obtained 30 different blocking probabilities. Now, let x_1, x_2, x_3, ... ,x_n be the calculated blocking probabilities from n=30 batches of observations. Then, the mean blocking probability is:

$$x_{mean} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

and the standard deviation s is:

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_{mean} - x_i)^2}{n-1}}$$

The confidence interval at 95% confidence is given by:

$$(x_{mean} - 1.96\frac{s}{\sqrt{n}}, x_{mean} + 1.96\frac{s}{\sqrt{n}})$$

The confidence interval tells us that the true population's mean lies within the interval 95% of the time. That is, if we repeat the above experiment 100 times, 95% of these times, on the average, the true population mean will be within the interval. Accurate simulation results require that the confidence interval is very small. That is, the error $1.96(s/\sqrt{n}) \sim 0.01x_{mean}$. If the error is not small enough, increase the batch size from 300 observations to 350, and run the simulation again for 30 batches. Keep increasing the batch size until you are satisfied that the error is within the limits.

The delay to schedule a reservation request is the difference from the time of the beginning of the requested window, t_1, to the beginning of the time that the request was actually scheduled. It is given by the value of t'. Use the batch means method as described above to calculate the mean delay and its confidence intervals. Keep track of

all the values t' for all the successfully scheduled requests, and at the end of each batch calculate the average delay. Then follow the above method, assuming that x_i is the mean delay of the i^{th} batch.

When you start the simulation, you will assume that there are no reservations, i.e. the system is empty. The initial behaviour of the simulation will be affected by this "system empty" initial condition. In order to eliminate the effects of the initial condition, you should run the simulation for 100 arrivals. After that, start the batch means method described above.

Input values and required results

For your simulation experiments, use the following input values:

	Inputs	**Values**	**Distribution**
1	Number of LSRs (N)	5,6,7	-
2	Link capacity	1 Gbps	-
4	Mean inter-arrival period	5,6,7,…,20	Exponential
5	Maximum intermediate period	50	Uniform
6	Maximum duration of reservation	100,110,120,…,200	Uniform
7	Simulated number of arrivals	Initially 9000	

Recall that in this simulation, the requested bandwidth is always equal to 1 Gbps, which is the link's capacity.

Run your simulation to obtain the following:

1. Plot the mean blocking probability and its confidence interval vs. the mean inter-arrival time (range given in the above table) for N (number of LSRs,) = 5,6, and 7.
2. Plot the mean blocking probability and its confidence interval vs. the maximum duration of reservation (range given in the above table) for N = 5, 6, and 7.
3. Plot the mean delay and its confidence interval vs. the mean inter-arrival times (range given in the above table) for N = 5,6, and 7.
4. Plot the mean delay and its confidence interval vs. the maximum duration of reservation (range given in the above table) for N = 5,6, and 7.

5. For (a) and (c), assume that the maximum reservation duration is 150. For (b) and (d) assume that the mean inter-arrival time be equal to 12.

Discuss your results.

8.2 An advance network reservation system - delayed scheme

This project builds on the previous simulation model of an advance reservation system. You will be able to re-use most of the previous code. The objective of this project is to simulate the same advance reservation system as before under the assumption of *delayed reservations*. In project 1, a request was blocked if it was not possible to find a path for the requested duration within the requested reservation window. In this project, we will assume that a reservation request is never blocked. Rather, a reservation is made on one of the paths at the first available time from the requested start time of the reservation window.

The simulation model

All the assumptions regarding the simulation and data structures are the same as before, with the exception of the search algorithm to find a free path. The search algorithm, unlike the one used in Project 1, searches for a free period of time over the one-link and over all the two-link paths at each time slot. As an example, let us assume a mesh network of 5 LSRs and that x = 0, y = 3. Then, the one-link path is {0-3}, and the two-link paths are{0-1, 1-3}, {0-2, 2-3} and {0-4, 4-3}. The algorithm first checks the one-link path to see if it is available for t units of time starting at time t_1. Failing that, repeat the same process by checking each of the two-link paths, selected randomly. Failing to find a free period, repeat the above but start at time t_1+t', where t'=1. Continue by increasing t' by one, until you find a free period on one of the paths. The value of t' is the delay from the beginning of the requested reservation window to the beginning of the actual reservation.

Values and required results

For your simulation experiments, use the following input values:

	Inputs	**Values**	**Distribution**
1	Number of LSRs (N)	5,6,7	-
2	Link capacity	10 Gbps	-
4	Mean inter-arrival time	5,6,7,…,20	Exponential
5	Maximum intermediate period	50	Uniform
6	Maximum duration of reservation	200,210,220,…,300	Uniform

The requested bandwidth is an integer number between 1 and 10, uniformly distributed. The simulated number of arrivals should be the same as in Project 1. That is, use 30 batches with the same batch size you used in Project 1.

Run your simulation to obtain the following:

1. Plot the mean delay and its confidence interval vs. the mean inter-arrival time (range given in the above table) for N (number of LSRs,) = 5,6, and 7.
2. Plot the mean delay and its confidence interval vs. the maximum duration of reservation (range given in the above table) for N = 5,6, and 7.
3. Plot the percent of time that a reservation was made within its requested reservation window and its confidence interval vs. the mean inter-arrival times (range given in the above table) for N = 5,6, and 7. This percent of time can be calculated in the same way you calculate the blocking probability in project 1. That is, for each batch i, calculate the ratio of the number of reservations made within their requested reservation window divided by the total number of successful reservations. Follow the batch means method, by associating this value with x_i.
4. Plot the percent of time that a reservation was made within its requested reservation window and its confidence interval vs. the maximum duration of reservation (range given in the above table) for N = 5,6, and 7.
5. Plot the percent of time a reservation was made on a one-link path and its confidence interval vs vs. the mean inter-arrival times (range given in the above

table) for N= 5,6, and 7. This percentage can be calculated in the same way as in item (c) above. That is, for each batch *i*, calculate the ratio of the number of reservations made on one-link paths divided by the total number of successful reservations. Follow the batch means method, by associating this value with $x_{i.}$

6. Plot the percent of time a reservation was made on a one-link path and its confidence interval vs. the maximum duration of reservation (range given in the above table) for N = 5,6, and 7.

For (a), (c) and (e), assume that the maximum reservation duration is 250. For (b), (d), and (f) assume that the mean inter-arrival time is 12.

Discuss your results.