

# Chapter 10

---

## □ DIRECT SIMULATION

Advice to persons about to marry — don't. *Punch* magazine 1845.

Advice to persons about to simulate — ?

### 10.1 INTRODUCTION

A system is simulated directly when each part of it is represented by a procedure and driven under the control of a system procedure that calls the various part-procedures to represent the working of the complete system.

This is in contrast to *indirect simulation* in which a mathematical model is built up of various functions and equations which are solved numerically.

Thus, in direct simulation the mathematical formulation is bypassed and the system is represented by a set of procedures which are run under the control of a master or system procedure (a sequencer). The usual reason for bypassing the mathematical formulation is the intractability of the system to mathematical description or the intractability of solving the system equations. Unfortunately the other reasons for bypassing the mathematical formulation are ignorance and laziness. Too often people dive straight into a direct simulation of a problem without any preliminary analysis with the result that we see huge simulation programs solving problems that could be solved with a few pages of diagrams and calculations.

The word *system* is used here for an actual (physical) system or a model or approximation to an actual system.

**A Theory of Simulation?** . We have seen (or you can see Chen 1970) that *Mathematical System Theory* has been highly developed and used especially in Telecommunications and Control Engineering and, to a limited extent, in Mathematical Economics. The simulation of such systems, once they have been formulated as mathematical systems (models), is relatively easy in that all the well-developed tools of numerical mathematics and scientific programming languages are available to solve or 'simulate' the system. For example, the discrete-time linear system

$$x(t+1) = Ax(t) + u(t), \quad x, u \in R^n, \quad A \in R^{n \times n},$$

is easily 'simulated' by the following program fragment

```

procedure DiscreteTime(A:matrix; x,u:vector)
for t := tbeg to tend do
  for i := 1 to n do
    sum := 0
    for j := 1 to n to
      sum := sum + a[i,j] * x[j]
    endfor
    x[i] := sum + u(t)
  endfor
endfor
endproc DiscreteTime

```

Direct simulation on the other hand can be difficult and confusing because the discipline of formulation and analysis is absent. There is no ‘theory of simulation’ to guide the builder of simulators. As a result simulation has tended to be done in an *ad hoc* manner despite the claims of simulation language designers (sellers). Some efforts have been made to tidy up the area by the use of simulation languages that impose a certain ‘world view’ or structure on simulators. Unfortunately there does not seem to be any one language that suits all problems and problem-solvers. Most of these languages impose restrictions that make the translation of a system into a set of procedures difficult. In addition, simulation languages that are not embedded in general-purpose programming languages have very limited programming capabilities (see Bratley, Fox, and Schrage, 1983).

## 10.2 MATHEMATICAL SYSTEM THEORY

Given that mathematical system theory and its concepts have been well-defined and developed it is surprising that there is so much confusion and controversy about basic definitions and concepts in the simulation literature ( see Nance, 1981).

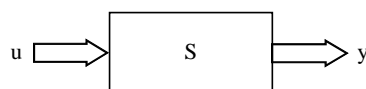
The following is an initial attempt to use, in a very limited sense, mathematical system theory terminology and concepts to define a standard model for systems to be simulated and a general algorithm to simulate the operation of the model.

We now define some of the very basic concepts of mathematical system theory. We will then use these concepts in developing a prototype simulator that will be used as a framework for building actual simulation models.

**Definition.** A **system** is any set of objects that are *united* by some form of interaction or interdependence.

This definition is too imprecise to be useful.

**Input-Output Description of Systems.** We may be more precise by viewing a system as a set of *input-output pairs*  $\{(u, y)\}$  which are *causes* and *effects*. This gives us the input-output or ‘black-box’ description of a system : we view the system as a box  $S$  that accepts inputs  $u$  and produces outputs  $y$ , as shown in Figure 1.



**Figure 10.1.** Input-Output Model.

Normally  $u$  and  $y$  are vector-valued functions of time and so we call such systems *dynamic*. Thus a system is an abstract object that receives inputs (causes, stimuli) which vary over time; the system reacts to (processes, changes) these inputs to produce outputs (effects, results) which also vary over time. Hence we may define a system as

$$S = \{(u(t_0, t_1), (y(t_0, t_1)))\},$$

where  $u(t_0, t_1)$  is a vector-valued input function of  $t \in [t_0, t_1]$  and  $y(t_0, t_1)$  is a vector-valued output function of  $t \in [t_0, t_1]$ . In other words, for every input  $u(t)$  there is an output  $y(t)$ .

If the output  $y(t_1)$  at time  $t_1$  only depends on the input  $u(t_1)$  applied at time  $t_1$  then the system is called *instantaneous* or *zero-memory*.

If the output  $y(t_1)$  at time  $t_1$  only depends on the input  $u(t_2)$  applied at time  $t_2 > t_1$  then the system is called *anticipatory* or *non-causal*.

If the output  $y(t_1)$  at time  $t_1$  depends on all inputs  $u(t)$  applied at time  $t \in [-\infty, t_1]$  then the system is called *causal* and has *memory*.

All physical systems are causal and most have memory, i.e., they do not react instantaneously. For example, a compressed spring has memory in the sense that it stores (remembers) potential energy which it acquired in the past.

The input-output description may be written as a function or mapping

$$y(t) = S(t)u(t), \quad u(t) \in R^m, \quad y(t) \in R^n, \quad S(t) \in R^{n \times m}.$$

The system  $S$  is *linear* if

$$y(t) = S(t)(\alpha_1 u_1(t) + \alpha_2 u_2(t)) = \alpha_1 S(t)u_1(t) + \alpha_2 S(t)u_2(t) = y_1(t) + y_2(t).$$

The system is *time-invariant* if  $S(t) = S, \quad \forall t$ . This does not mean that the output of the system is constant : a variable input will give a variable output.

**State Description of Systems.** The input-output description does not tell us enough about what is going on in the system—the system is just a black box whose innards are not accessible.

The *state variable description* expands the input-output description by defining the concept of *state* and describing the system as mappings from an input space  $U$  to a state space  $X$  to an output space  $Y$ , i.e.,

$$U \mapsto X \mapsto Y.$$

**Definition.** The **state** of a system at  $t_0$  is the *information at  $t_0$*  that, together with the input  $u(t)$ ,  $t \geq t_0$ , determines uniquely the output of the system  $y(t)$  for all  $t \geq t_0$ .

EXAMPLE : (*Newtonian Mechanics*). If an external force (input) is applied to particle of mass  $m$  (system) at time  $t_0$ , the motion of the particle (output) for  $t \geq t_0$  is uniquely determined by Newton's Laws, if the *position*  $p$  and *velocity*  $v$  of the particle at  $t_0$  are known. The vector  $(p, v)$  is the *state vector*. Note that the mass of the particle is not a component of the state vector. It is part (a parameter) of the system description  $S(t)$ . □

We note that the state at time  $t_0$  implicitly records the past behavior of the system because the present state was *caused* by the action of past inputs on past states.

EXAMPLE : (*Forestry*). The state of a forest can be defined as the number of trees and their age or size, and the amount of water and nutrients in the soil. This state is acted on by inputs such as rain, sun, fertilizer, cutting and thinning. The outputs are firewood, boxwood, paper, and recreational facilities. □

**State Transition Function.** This function describes how a system moves or changes from one state to another under the influence of the input  $u(t)$ . It is the mapping

$$T : X \times U \mapsto X.$$

The transition function of a ‘smooth’, causal, non-instantaneous system is a solution of the following equations :

$$\begin{aligned} \dot{x}(t) &= f(x(t), u(t), t) \\ y(t) &= g(x(t), u(t), t) \end{aligned} \quad \text{continuous time.}$$

$$\begin{aligned} x(t+1) &= f(x(t), u(t), t) \\ y(t) &= g(x(t), u(t), t) \end{aligned} \quad \text{discrete time.}$$

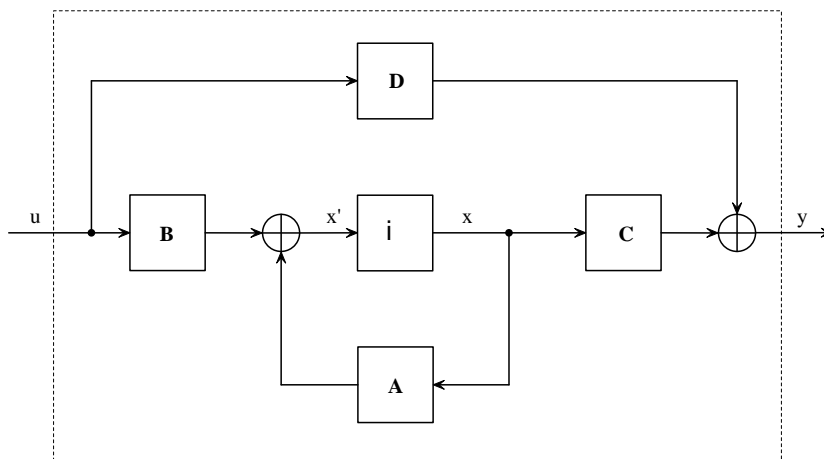
The first set of equations are differential equations and the second are difference equations. Note that if the system is instantaneous (memoryless) then there are no differential or difference equations, simply algebraic equations.

If the system is linear and time-invariant then the equations above become

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \end{aligned} \quad \text{continuous time,}$$

$$\begin{aligned} x(t+1) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \end{aligned} \quad \text{discrete time,}$$

where  $A, B, C, D$  are, respectively,  $n \times n, n \times p, q \times n$  and  $q \times p$ , real constant matrices and  $x, y, u$  are  $n \times 1, q \times 1, p \times 1$ , respectively. Notice that the dotted box in Figure 2 is the input-output black box.



**Figure 10.2.** State Space Model.

EXAMPLE : Consider Samuelson’s ‘multiplier-accelerator’ macro-economic model :

$$\begin{aligned}
 C(t) &= aY(t - 1), \\
 I(t) &= b[C(t) - C(t - 1)], \\
 Y(t) &= C(t) + I(t) + G(t),
 \end{aligned}$$

where  $Y(t), C(t), I(t), G(t)$  are Gross National Product, Consumption, Investment, and Government Spending, respectively. We can transform this model into state-space form as follows : let  $x_1(t) = C(t), x_2(t) = I(t)$  and  $u(t) = G(t)$ . This gives

$$\begin{aligned}
 \begin{bmatrix} C(t+1) \\ I(t+1) \end{bmatrix} &= \begin{bmatrix} a & a \\ ab & ab \end{bmatrix} \begin{bmatrix} C(t) \\ I(t) \end{bmatrix} + \begin{bmatrix} 1 \\ ab \end{bmatrix} [G(t)] \\
 Y(t) &= [1 \quad 1] \begin{bmatrix} C(t) \\ I(t) \end{bmatrix} + G(t)
 \end{aligned}$$

The constants  $a$  and  $b$  determine the eigenvalues of the matrix  $A$  which in turn determine the dynamic behavior of the model. □

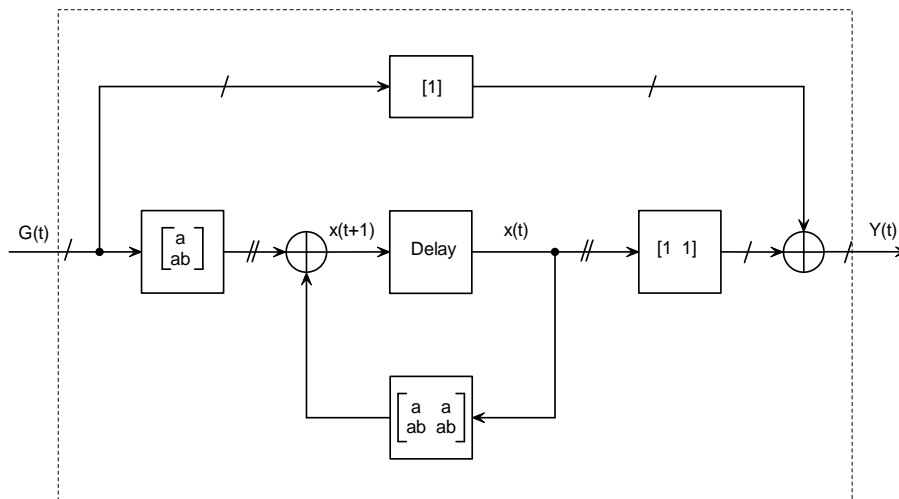


Figure 10.3. State Space Model of the Economy.

### 10.3 THE STRUCTURE OF DIGITAL SIMULATORS

We now develop, using the ideas and terminology above, an outline for a general-purpose discrete-time digital simulator. Broadly speaking, our simulation model will be represented as follows :

1. The state(s) is represented by some data structure(s).
2. The input, state transition, and output functions are represented by procedures which operate on the data structures.
3. A special procedure called a *Sequencer* or *Monitor* applies (calls) the procedures to the data structures in a sequence ordered by a discrete time sequence  $t = \{t_0, t_1, \dots, t_k\}$  or *clock*.

The sequencer is really the heart of the simulator and its unusual function is necessary because we will be running the simulator on a (single-processor) sequential machine-compiler system. Therefore we cannot simulate the concurrent operation of two or more parts of the model. The sequencer's job is to schedule the operations of the model in a time-ordered sequence. Thus it acts very much like a computer operating system that monitors the tasks that are waiting to be done and performs them one-at-a-time, usually ordered by time. (This is not really true because all computers have separate processors for performing screen graphics, disk I/O, and various other tasks). Multi-tasking systems really just time-share the processor with many tasks.

The sequencer also has various book-keeping tasks which vary with the type of problem being simulated. In general it will keep track of the state variable and accumulate statistics on various events and components of the state variable.

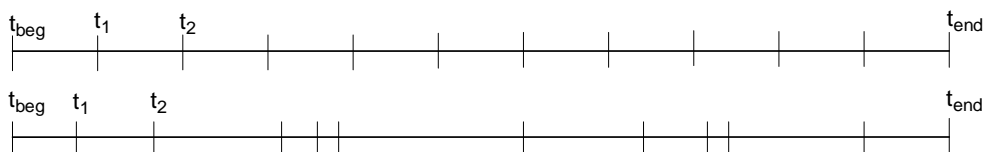
**Sequencers.** A sequencer is essentially a loop in which a *simulated clock* is updated and various procedures are called.

```

procedure DiscreteTimeSequencer( $t_{beg}, t_{end}$ )
     $t := t_{beg}$ 
    while  $t \leq t_{end}$  do
        Apply Procs to State
         $t := t + \delta(t)$ 
    endwhile
end{ PROC DiscreteTimeSequencer}
    
```

There are two types of sequencers :

- *synchronous* sequencers keep track of time by updating a simulated clock by a fixed time increment on each pass through a sequencing loop. This means that in the procedure above  $\delta(t) = \delta$ . Thus the this clock is in synchrony with a real clock.
- *asynchronous* sequencers update the simulated clock by a variable time increment  $\delta(t)$ .



**Figure 10.4.** Synchronous and Asynchronous Timing.

In its simplest form the sequencer is a simple loop. ‘Simulating’ the discrete linear time-invariant system above is simple, as we can see from the following program.

```

procedure LTIDSequencer( $A, B, C, D, u, x, t_{beg}, t_{end}$ )


---


  for  $t := t_{beg}$  to  $t_{end}$  do
     $x := Ax + Bu(t)$ 
     $y := Cx + Du(t)$ 
  endfor
end{ PROC LTIDSequencer}

```

This sequencer is synchronous because the time is incremented by the **for** -loop in fixed steps of 1. Note also that  $x$  and  $y$  have no  $t$  argument :  $x$  on the left of the assignment operator is in fact  $x(t)$  while the one on the right was calculated in the previous iteration, i.e.,  $x(t - 1)$ . The input is an externally supplied function  $u(t)$  that returns a vector that depends on  $t$ .

## 10.4 DISCRETE EVENT SIMULATORS

These are asynchronous sequencers in which time is updated (incremented) every time an *event* occurs.

**DEFINITION (Event).** An **event** is an occurrence that causes a change of state in a system. Associated with every event is an *event time*, which is the time at which the state changes or an event occurs. □

In the simple sequencer above an *update* event occurs on each pass through the loop. The event times are  $\{t_{beg}, t_{beg} + 1, \dots, t_{fin}\}$ .

Before we discuss discrete event simulators it is important to have a clear idea of how events and states are related. We now give some examples states and events for various systems.

**EXAMPLE :** (*A Single Server Queueing System*)

The state vector is: (*No. in system, Server busy or idle*). Note that we can deduce the number in the queue from this state information.

The events are : *arrival, begin service, end service*. Each of these changes the state. How? □

**EXAMPLE :** (*An Airport System*) A small airport with one runway serves planes arriving (departing) from (to)  $n$  cities. The planes arrive and depart according to a fixed time-table. The airport provides service (loading, unloading of passengers and baggage, re-fuelling, etc.) at  $m$  gates. No major maintenance is possible. All planes have the same capacity and type.

The state vector is : (*No. of planes stacked, No. landed and waiting for gate, gates doing service, No. waiting to take off*).

The events are : (*arrival, landing, enter a gate, leave a gate, take off*). □

**EXAMPLE :** (*An Inventory System*) A warehouse stocks one type of product. Demands for the product occur at random times for random amounts. If there are no units of the product to meet a demand then this demand is backlogged and will be filled as soon as a new shipment of the product arrives. The time between placing an order for a shipment and its arrival is random. This is called the *lead time*.

The state vector is : (*No. product on hand, No. on order, No backlogged*).

The events are : (*Demand for product, Order of shipment, Receipt of shipment*). □

**Exercise 10.4.1** : In each of the examples above determine why and how each event changes the state vector  $x$  □

### 10.4.1 The Operation of a Discrete Event Simulator

The operation of a discrete event simulator is fairly simple. The sequencer maintains a *list of future events* and their times of occurrence. Then the following steps are repeatedly executed until the simulation is finished :

1. Select the next (earliest) event from the list. This becomes the *current event*.
2. Update the *clock* to the time of the current event.
3. Call the *event procedure* corresponding to the current event.
4. The event procedure changes the state variable, generates some future event and inserts it into the list of future events.

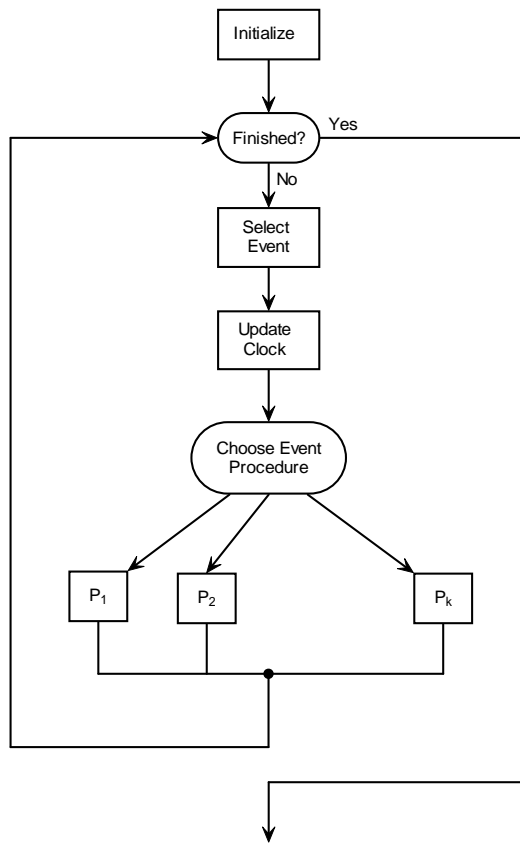
```
procedure DiscreteEventSim (state, tbeg, tend)
```

```

Create(EvList)
Insert((BeginSim, tbeg), EvList)
Insert((EndSim, tend), EvList)
while NOT Empty(EvList) do
    (Event, Time) := Select(EvList)
    Clock := Time
    case Event of
        BeginSim : BeginSimP(state)
        Event1    : Event1P(state)
                .
                .
                .
        Eventk    : EventkP(state)
        EndSim    : EndSimP(state)
    end{ case }
endwhile

end{ PROC DiscreteEventSim}

```



**Figure 10.5.** Discrete Event Simulator.

The first three steps are easily implemented in any language : the event list is operated as a *priority queue*. Updating the clock means setting the clock variable to the time of the current event. Calling the appropriate event procedure is accomplished by a *case* statement. The event procedures need detailed study.

**Event Procedures.** These are the pieces of code that actually perform the simulation, i.e., mimic the system being simulated. They have three main functions :

1. Change the values of the components of the state vector.
2. Collect (update) statistics relating to various elements of the system.
3. Generate future events.

Each of these items requires careful analysis and design and each varies with the system being simulated.

1. The state vector must be carefully defined and the events that change the individual components of the state vector must be identified. The state vector may contain redundant information but if it contains too little we cannot (by definition) determine the future output of the system.

2. The collected statistics are, in effect, the output of the simulation and, in general, they fall into three categories :

1. *Point Statistics.* These are for variables that do not depend on time. For example, the height of a customer, the size of a product. We can calculate averages and record max/min values.

2. *Time Statistics*. These are for variables that change over time. We need to record the value of such a variable and the length of time the variable had that value.
3. *Frequency Counts*. These record the number of times a variable had a particular value or range of values.

We will discuss each of these types of statistics in detail later.

3. Events cause or generate other events. For example, in a queueing system a start-of-service event implies an end-of-service event. What is not so obvious is that an arrival causes a future arrival. Thus an arrival procedure generates a sequence of future arrivals, one-at-a-time. This is more efficient than generating the complete sequence of arrivals at the start of the simulation. It is very important to be clear about which events cause (generate) other events because this interaction is at the heart of any simulator.

We could, in principle, generate and store all sequences of events at the start of the simulation but this would be complex and wasteful of memory.

### 10.4.2 An Example of a Discrete Event Simulator

We will build now a simulator for a simple inventory system. We will describe the various parts of it at a high level and leave the low-level implementations details until we have discussed the implementation of a general-purpose sequencer.

A warehouse contains one type of product. Demands for the product occur at random times with a Poisson distribution with mean  $\lambda$  per day. The time between demands is, therefore, Exponential  $\lambda e^{-\lambda t}$ . The size of individual demand is Normal  $N[\mu_d, \sigma_d^2]$ . If there is no product to meet demand then this demand is *backlogged* and filled as soon as a new shipment of the product arrives. The *lead time* between an order placement and arrival of a new shipment is random  $N[\mu_{lt}, \sigma_{lt}^2]$ .

Without yet stating the objective of the simulation let us build a simulator for this system.

#### System Parameters.

1. Arrival Rate :  $Poisson(\lambda)$ .
2. Demand :  $Normal(\mu_d, \sigma_d^2)$ .
3. LeadTime :  $Normal[\mu_{lt}, \sigma_{lt}^2]$ .
4. Start and Finish :  $t_{beg}, t_{fin}$ .
5. Initial level of inventory.
6. Initial event.

#### State Variable Components.

1. Inventory on hand : *OnHand*.
2. Amount on order : *OnOrder*.
3. Amount Backlogged : *Backlog*.

#### Events.

1. Begin Simulation (pseudo-event).
2. Demand.
3. Sending of Order.
4. Receipt of Order.

## 5. Finish Simulation (pseudo-event).

We assume that we have a proper sequencer and so the main work is to write the event procedures.

**Event Procedures.** It is here that the greatest care is needed because minor errors or oversights can ruin the simulation.

1. **Demand** : This is the pair  $AmountD, time$ . If there is enough inventory then the demand is filled and the amount on hand is decreased. Otherwise the demand is backlogged and a shipment is ordered.

```

procedure Demand(AmountD,time)


---


  if AmountD <= OnHand THEN
    OnHand := OnHand - AmountD      {*}
  else
    BackLog := BackLog + AmountD    {*}
    Insert((Order,AmountO,time), EventList)
  endif
  Dtime := time + Generate(Exp( $\lambda$ ))
  AmountD := Generate(Normal( $\mu_d, \sigma_d^2$ ))
  Insert((Demand,AmountD,Dtime), EventList)
end{ PROC Demand}

```

2. **Order** : This is the pair  $AmountO, time$ . The amount on order is increased and a *Receipt of Order* is generated.

```

procedure Order(AmountO,time)


---


  OnOrder := OnOrder + AmountO      {*}
  RecTime := time + Generate(Normal( $\mu_{it}, \sigma_{it}^2$ ))
  Insert((Receipt,AmountO,RecTime), EventList)
end{ PROC Order}

```

3. **Receipt of Order** : This is the pair  $AmountR, time$ . The amount on hand and backlogged orders are filled immediately.

```

procedure Receipt(AmountR,time)


---


  OnHand := OnHand + AmountR      {*}
  Fill(BacklogList)
end{ PROC Receipt}

```

## 10.5 REFERENCES

1. Banks, J., and Carson, J. : *Discrete-Event System Simulation*, Prentice-Hall, 1984.
2. Bratley, P., Fox, B., and Schrage, L. : *A Guide to Simulation*, Springer, 1983.
3. Chen, C.T. : *Introduction to Linear System Theory*, Holt, Rinehart, and Winston, 1970.
4. Law, A., and Kelton, W. : *Simulation Modeling and Analysis*, McGraw-Hill, 1982.
5. Nance, R.E. : 'Time and State Relationships in Simulation Modeling', *Communications of the ACM*, Vol 24, No 4, Apr 1981.
6. Groff, G.K, and Muth, J.F. : Chapter 12, 'Simulation Models', in *Operations Management*, Irwin, 1972.



**Figure 10.6.** Random Walk with  $n = 10000$ .