Eastern Economy Edition

System Simulation with Digital Computer

Narsingh Deo

PHI

Contents

1. INTRODUCTION 1–14

1-1. Simulation of a pure-pursuit problem—an example 1
1-2. A system and its model 5
1-3. Simulation of an inventory problem 5
1-4. The basic nature of simulation 9
1-5. When to simulate 10
1-6. Remarks and references 11
1-7. Exercises 12

2. SIMULATION OF CONTINUOUS SYSTEMS 15–39

2-1. A chemical reactor 15
2-2. Numerical integration vs. continuous system simulation 19
2-3. Selection of an integration formula 20
2-4. Runge-Kutta integration formulas 22
2-5. Simulation of a servo system 25
2-6. Simulation of a water reservoir system 27
2-7. Analog vs. digital simulation 31
2-8. Remarks and references 33
2-9. Exercises 35

3. DISCRETE SYSTEM SIMULATION 40–61

3-1. Fixed time-step vs. event-to-event model 40
3-2. On simulating randomness 42
3-3. Generation of random numbers 43
3-4. Generation of non-uniformly distributed random numbers 50
3-5. Monte-Carlo computation vs. stochastic simulation 56
3-6. Remarks and references 57
3-7. Exercises 59

4. SIMULATION OF QUEUEING SYSTEMS 62–87

4-1. Rudiments of queueing theory 63
4-2. Simulation of a single-server queue 72
4-3. Simulation of a two-server queue 76
4-4. Simulation of more general queues 82
4-5. Remarks and references 84
4-6. Exercises 85

5. SIMULATION OF A PERT NETWORK 88–111

5-1. Network model of a project 88
5-2. Analysis of an activity network 90
5-3. Critical path computation 92
5-4. Uncertainties in activity durations 98
5-5. Simulation of an activity network 99
5-6. Computer program for simulation 100
5-7. An example 104
5-8. Resource allocation and cost considerations 107
5-9. Remarks and references 108
5-10. Exercises 110

6. INVENTORY CONTROL AND FORECASTING 112–143

6-1. Elements of inventory theory 112
6-2. More complex inventory models 117
6-3. Simulation Example—1 121
6-4. Generation of Poisson and Erlang variates 128
6-5. Simulation Example—2 131
6-6. Forecasting and regression analysis 135
6-7. Remarks and references 140
6-8. Exercises 141

7. DESIGN AND EVALUATION OF SIMULATION EXPERIMENTS 144–168

7-1. Length of simulation runs 144
7-2. Variance reduction techniques 155
7-3. Experimental layout 160
7-4. Validation 162

7-5. Summary and conclusions 164
7-6. Remarks and references 165
7-7. Exercises 166

8. SIMULATION LANGUAGES 169–197

8-1. Continuous and discrete simulation languages 170
8-2. Continuous simulation languages 170
8-3. Block-structured continuous simulation languages 171
8-4. Expression-based languages 175
8-5. Discrete-system simulation languages 181
8-6. SIMSCRIPT 183
8-7. GPSS 186
8-8. SIMULA 187
8-9. Factors in selection of a discrete system simulation language 189
8-10. Remarks and references 193

INDEX 198–200

SYSTEM SIMULATION
WITH
DIGITAL COMPUTER

1

Introduction

Simulation is a powerful technique for solving a wide variety of problems. To simulate is to copy the behaviour of a system or phenomenon under study. Strictly speaking, we will be dealing with only *numerical sequential simulation*; numerical because there are other forms of simulation—for example, electrical analogue or physical simulation; and sequential because the calculations proceed in a time sequence. Some of the basic ideas in simulation can be best understood by performing actual simulations. Let us, therefore, consider the following two very simple examples and see how simulation is actually done.

1-1. Simulation of a pure pursuit problem—an example

A fighter aircraft sights an enemy bomber and flies directly toward it, in order to catch up with the bomber and destroy it. The bomber (the target) continues flying (along a specified curve) so the fighter (the pursuer) has to change its direction to keep pointed toward the target. We are interested in determining the attack course of the fighter and in knowing how long it would take for it to catch up with the bomber.

If the target flies along a straight line, the problem can be solved directly with analytic techniques. (The proof of such a closed-form expression which gives the course of the pursuer, when the target flies in a straight line, is left as an exercise for you. Problem 1-2.)

However, if the path of the target is curved, the problem is much more difficult and normally cannot be solved directly. We will use simulation to solve this problem, under the following simplifying conditions:

1. The target and the pursuer are flying in the same horizontal plane when the fighter first sights the bomber, and both stay in that plane. This makes the pursuit model two-dimensional.
2. The fighter's speed VF is constant (20 kms/minute).
3. The target's path (i.e., its position as a function of time) is specified.
4. After a fixed time span Δt (every minute, in this case) the fighter changes its direction in order to point itself toward the bomber.

Let us introduce a rectangular coordinate system coincident with the horizontal plane in which the two aircraft are flying. We choose the point due south of the fighter and due west of the target (at the beginning of the pursuit) as the origin of this coordinate system. Let the distances be given

in kilometers and the time in minutes. We start measuring the time when the fighter first sights the bomber. (See Fig. 1-1.)

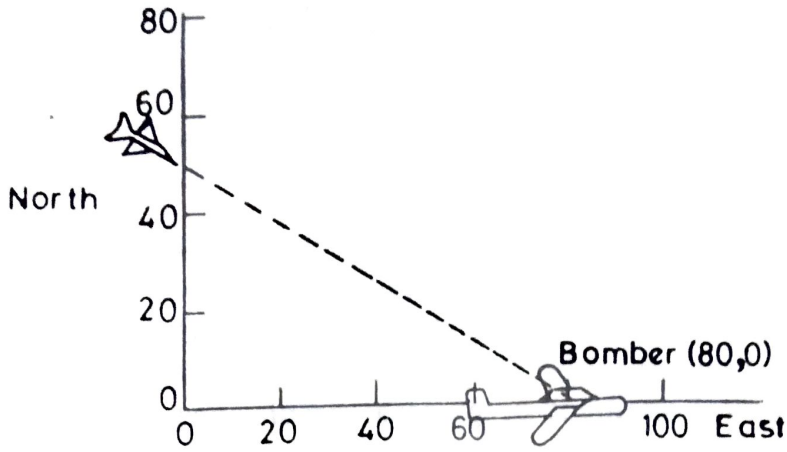


Fig. 1-1: Positions of Pursuer and Target at Time Zero.

We will represent the path of the bomber (which is known to us in advance) by two arrays, the east coordinates and the north coordinates at specified moments (each minute). We call these coordinates $XB(t)$ and $YB(t)$, respectively. They are presented in the form of a table (in kilometers) below.

Time, t	0	1	2	3	4	5	6	7	8	9	10	11	12
$XB(t)$	80	90	99	108	116	125	133	141	151	160	169	179	180
$YB(t)$	0	−2	−5	−9	−15	−18	−23	−29	−28	−25	−21	−20	−17

Table 1-1.

Likewise, we will represent the path of the fighter plane by two arrays $XF(t)$ and $YF(t)$. In this example, initially we are given

$$YF(0) = 50 \text{ kms}, \quad XF(0) = 0 \text{ kms}.$$

Our purpose is to compute the positions of the pursuer, namely, $XF(t)$, $YF(t)$ for $t = 1, 2, \ldots, 12$, or until the fighter catches up with the bomber. We will assume that once the fighter is within 10 kms of the bomber, the fighter shoots down its target by firing a missile, and the pursuit is over. In case the target is not caught up within 12 minutes, the pursuit is abandoned, and the target is considered escaped. From the time $t = 0$ till the target is shot down, the attack course is determined as follows:

The fighter uses the following simple strategy: It looks at the target at instant t, aligns its velocity vector with the line of sight (i.e., points itself toward the target). It continues to fly in that direction for one minute,

till instant $(t + 1)$. At time $(t + 1)$ it looks at the target again and realigns itself.

The distance DIST (t) at a given time t between the target and the pursuer is given by

$$\text{DIST}(t) = \sqrt{(YB(t) - YF(t))^2 + (XB(t) - XF(t))^2} \quad \ldots(1\text{-}1)$$

The angle θ of the line from the fighter to the target at a given time t is given by

$$\sin\theta = \frac{YB(t) - YF(t)}{\text{DIST}(t)} \quad \ldots(1\text{-}2)$$

$$\cos\theta = \frac{XB(t) - XF(t)}{\text{DIST}(t)} \quad \ldots(1\text{-}3)$$

Using this value of the position of the fighter at time $(t + 1)$ is determined by

$$XF(t + 1) = XF(t) + VF\cos\theta \quad \ldots(1\text{-}4)$$

$$YF(t + 1) = YF(t) + VF\sin\theta \quad \ldots(1\text{-}5)$$

With these new coordinates of the pursuer, its distance from the target is again computed using Eq. (1-1). If this distance is 10 kms. or less the pursuit is over, otherwise θ is recomputed, and the process continues.

A flowchart of the logic of this problem is given in Fig. 1-2 (page 4).

The following FORTRAN program (a format-free version) will implement the flowchart.

```
      DIMENSION XB (25), YB (25), XF (25), YF (25)
      INTEGER T, J
      READ, (XB (T), YB (T), T = 1,13)
      READ, XF (1), YF (1), VF
      T = 1
  100 DIST = SQRT ((YB (T) – YF (T))** 2 + (XB (T) – XF (T))** 2)
      IF (DIST. LE. 10.0) GO TO 110
      IF (T.GT.12) GO TO 120
      XF (T + 1) = XF(T) + VF* (XB(T) – XF (T))/DIST
      YF (T + 1) = YF (T) + VF* (YB (T) – YF (T))/DIST
      T = T + 1
      GO TO 100
  110 PRINT 990, T, DIST
  990 FORMAT (10X, 10H CAUGHT AT, 13, 8H MTS AND, F10.3, 4H KMS)
      STOP
  120 PRINT 1000
 1000 FORMAT (10X, 16H TARGET ESCAPED)
      STOP
      END
```

(Note that since Fortran does not permit 0 as an index we had to set $T = 1$ to correspond to $t = 0$ in the flowchart.)

The output of this program for the specified data is as follows:

```
CAUGHT AT 10 MTS AND 2.969 KMS
```

This is an example of simulation. We had to resort to simulation be-

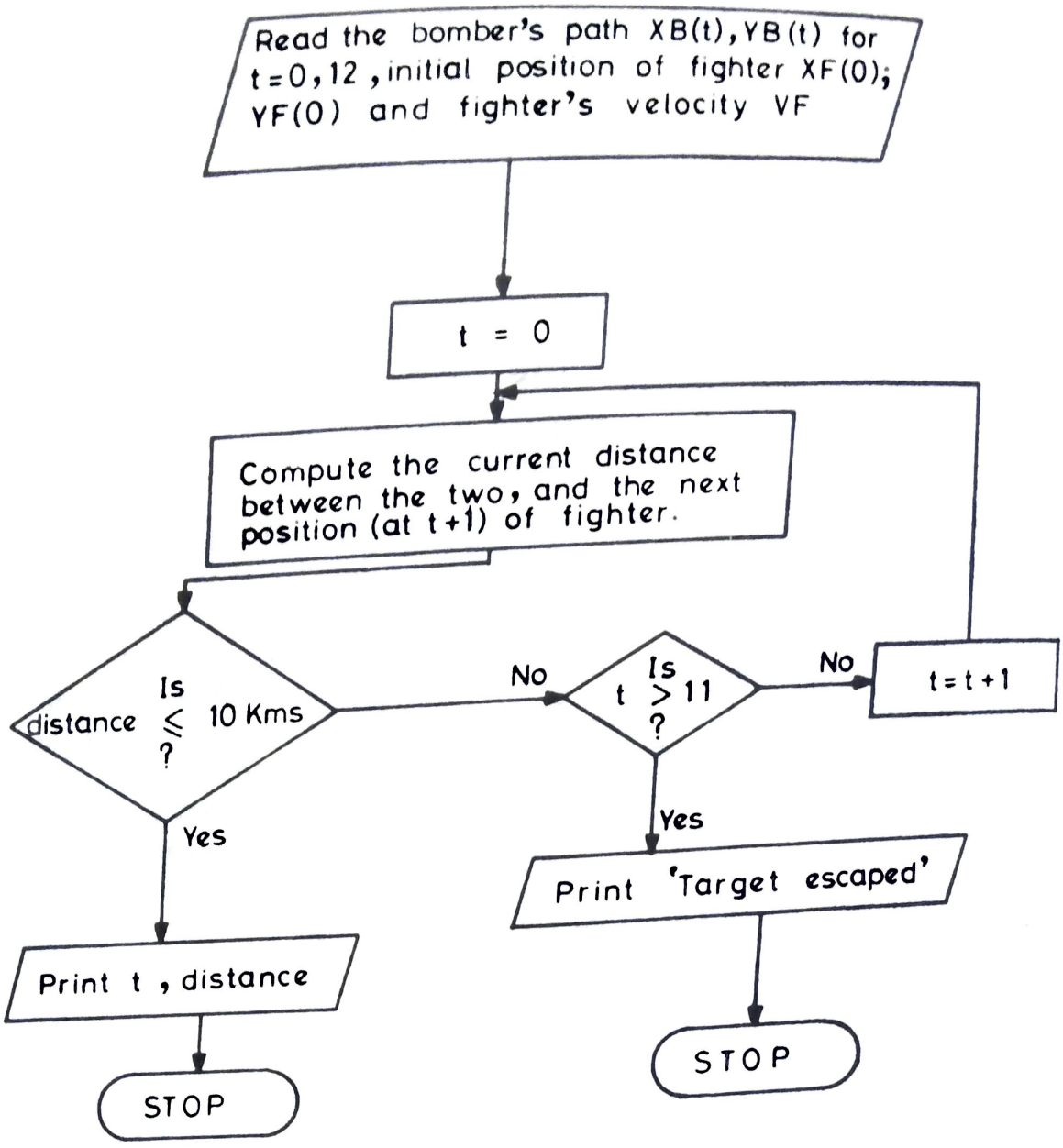


Fig. 1-2: Flowchart of pursuit problem.

cause analytically we could not make a long-term prediction about the path that the fighter plane would take (given the initial position and path of the target). But by simulation we were able to make the computer go through the instant-to-instant predictions for as many instants as we wanted. This was possible only because we knew the basic process involved, namely, how the fighter plane behaves at any particular instant. Such knowledge of the basic process of the system under study is essential for all simulation experiments.

The simple strategy, of pursuer redirecting himself toward the target at fixed intervals of time, while the target goes on its predetermined path without making any effort to evade the pursuer, is called *pure pursuit*. Although in many situations, the strategy used by the pursuer is more sophisticated, this basic approach can be used for any pursuit problem as long as we know

the path the pursued takes and the rule by which the pursuer redirects himself.

1-2. A system and its model

In any simulation experiment we have a system whose behaviour we are studying. Loosely speaking, a *system* is a collection of distinct objects which interact with each other. In the pure-pursuit example, the system consists of the two aircraft. In order to study a system we generally gather some relevant information about it. For most studies it is not necessary to consider all the details of the system. For example, in the example in Sec. 1-1 we did not get the information about the sizes or weights of the aircraft. Such a collection of pertinent information about a system is called a *model* of the system.

The construction of an appropriate model of a system under study is a delicate and an all-important affair in simulation (as it is in analytic study). For example, we have assumed that the target takes no evasive action, nor does the pursuer use any predictive technique. We assumed that the pursuer corrects its course only once a minute. Thus we know the 'law' that governs the system. The same system may be described by different models. Several variations of the pure-pursuit model are considered in problems at the end of this chapter.

Constructing appropriate mathematical models of complex, real-life systems is a vast and intricate subject in itself. It requires a thorough knowledge of the system as well as of modelling techniques. In this book, we will in most cases assume that a model has been arrived at and our concern is how to program the computer so that it behaves like the model as the time progresses.

For a given system there are a number of variables that describe what is known as the *state of the system* at any given time. For example, the state of our system (of pure pursuit) is described by the positions of the two aircraft and their velocities.

For further understanding these and other concepts, let us simulate another very simple system, which is different in a fundamental sense from the pure-pursuit system.

1-3. Simulation of an inventory problem

Suppose you work in a retail store and it is your responsibility to keep replenishing a certain item (say, automobile tyres) in the store by ordering it from the wholesaler. You want to adopt a simple policy for ordering new supplies:

> 'When my stock goes down to P items (called *reorder point*), I will order Q more items (called *reorder quantity*) from the wholesaler.'

If the demand on any day exceeds the amount of inventory on hand, the

excess represents lost sale and loss of goodwill. On the other hand, overstocking implies increased carrying cost (i.e., cost of storage, insurance, interest, deterioration, etc.). Ordering too frequently will result in excessive reorder cost. Assume the following conditions:

1. There is a 3-day lag between the order and arrival. The merchandise is ordered at the end of the day and is received at the beginning of the fourth day. That is, merchandise ordered on the evening of the ith day is received on the morning of the $(i + 3)$rd day.
2. For each unit of inventory the carrying cost for each night is Re. 0.75.
3. Each unit out of stock when ordered results into a loss of goodwill worth Rs. 2.00 per unit plus loss of Rs. 16.00 net income, that would have resulted in its sale, or a total loss of Rs. 18.00 per unit. Lost sales are lost forever; they cannot be backordered.
4. Placement of each order costs Rs. 75.00 regardless of the number of units ordered.
5. The demand in a day can be for any number of units between 0 and 99, each equiprobable.
6. There is never more than one replenishment order outstanding.
7. Initially we have 115 units on hand and no reorder outstanding.

With these conditions in force you have been asked to compare the following five replenishment policies and select the one that has the minimum total cost (i.e., reorder cost + carrying cost + lost sales cost).

	P (reorder point)	Q (reorder quantity)
Policy I	125	150
Policy II	125	250
Policy III	150	250
Policy IV	175	250
Policy V	175	300

(Since we are interested in simulation here and not in inventory theory, we will not investigate the reasonableness of the replenishment policy too critically. Ours is undoubtedly a very simplified model.)

The problem does not easily lend itself to an analytic solution; it is best therefore to solve it by simulation. Let us simulate the running of the store for about six months (180 days) under each of the five policies and then compare their costs.

A simulation model of this inventory system can be easily constructed by stepping time forward in the fixed increment of a day, starting with Day 1, and continuing up to Day 180. On a typical day, Day i, first we check to see if merchandise is due to arrive today. If yes, then the existing stock S is

increased by Q (the quantity that was ordered). If DEM is the demand for today, and $DEM \leqslant S$, our new stock at the end of today will be $(S - DEM)$ units. If $DEM > S$, then our new stock will be zero. In either case, we calculate the total cost resulting from today's transactions, and add it to the total cost C incurred till yesterday. Then we determine if the inventory on hand plus units on order is greater than P, the reorder point. If not, place

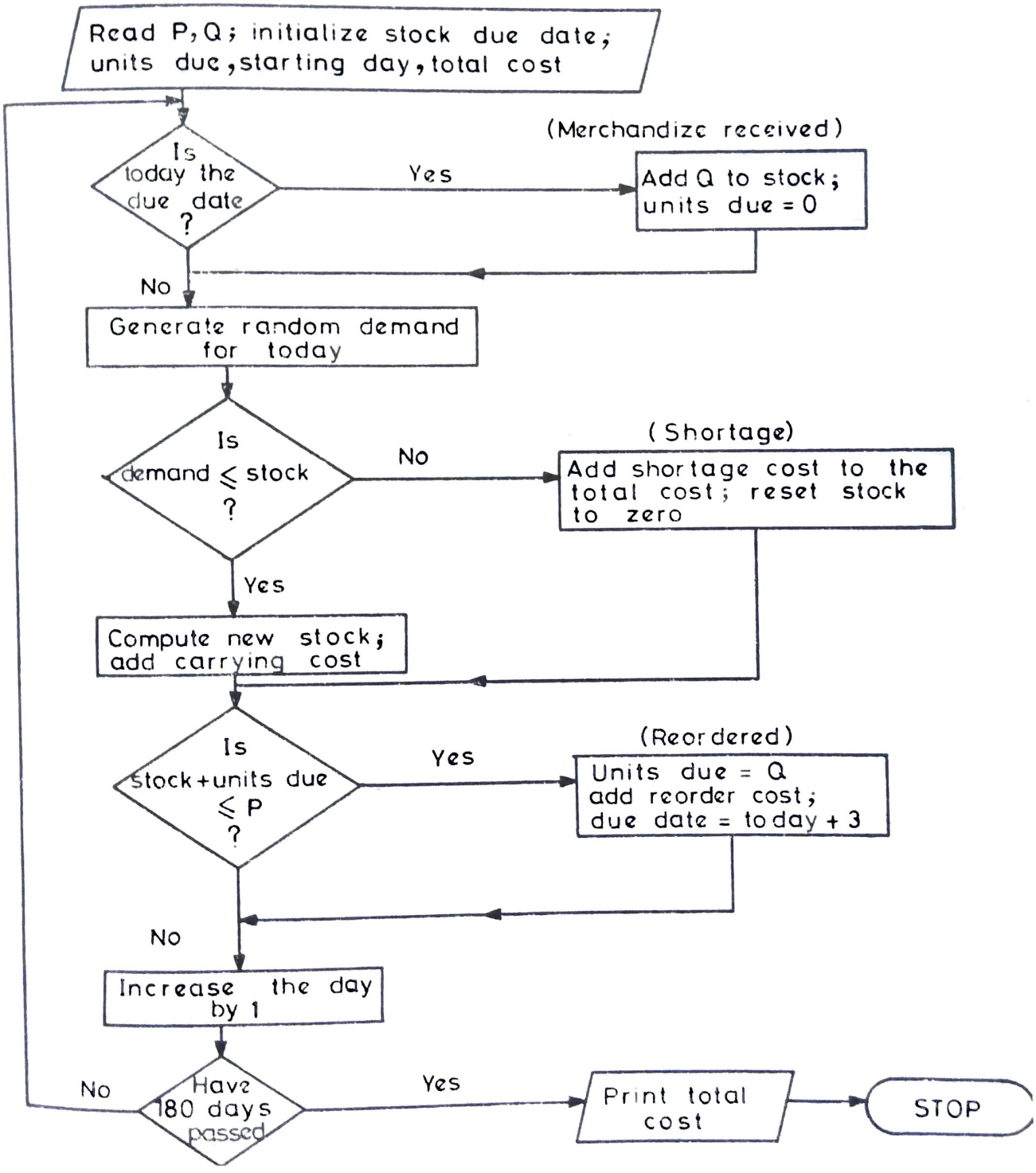


Fig. 1-3: Cost of inventory policy (P, Q).

an order (to be delivered three days hence), by stating the amount ordered and the day it is due to be received. We repeat this procedure for 180 days.

Initially we set day number $i = 1$, stock $S = 115$, number of units due $UD = 0$ (because there is no outstanding order), and the day they are due $DD = 0$.

The demand, *DEM*, for each day is not a fixed quantity but a random variable. It could assume any integral value from 00 to 99 and each with equal probability. We will use a special subroutine, which generates a 2-digit random integer, and will use the numbers thus produced as the daily demands. (Random number generators will be discussed in Chapter 3.) A flowchart for simulating this problem is shown in Fig. 1-3 (page 7).

The following Fortran program (format free) implements the flowchart for evaluating the total cost for a given replenishment policy (*P*, *Q*) for 180 days. Statement No. 110 in the program makes use of the subprogram *RNDY*1 (*DUM*) which is a subprogram to generate a real pseudorandom number between zero and one. The argument of this function can be any dummy number or variable.

Notice how condition 6, that there is no more than one reorder outstanding, has been taken care of. In Statement 130, we add the number of units due (if any) *UD* to the current stock *S* to get the equivalent stock, *ES*. It is this number which is compared to P before an order is placed. Since $UD > P$ if we already have a replenishment order outstanding another order will not be placed.

This is an extremely simple model of an inventory-control system. More realistic models will be dealt with in Chapter 6

```
      INTEGER P, Q, S, ES, UD, DD, DEM
      READ, P, Q
      C = 0.0
      S = 115
      I = 1
      UD = 0
      DD = 0
100   IF (DD .NE. I) GO TO 110
      S = S + Q
      UD = 0
110   DEM = RNDY1 (DUM)* 100.0
      IF (DEM .LE. S) GO TO 120
      C = C + (FLOAT (DEM) - FLOAT (S))*18.0
      S = 0
      GO TO 130
120   S = S - DEM
      C = C + FLOAT (S)*.75
130   ES = S + UD
      IF (ES .GT. P) GO TO 140
      UD = Q
      DD = I + 3
      C = C + 75.0
```

```
140  I= I + 1
     IF (I .LE. 180) GO TO 100
     PRINT, P, Q, C
     STOP
     END
```

The program yielded the following cost figures for the five inventory policies:

P	Q	Cost in Rs.
125	150	38679.75
125	250	31268.25
150	250	29699.25
175	250	26094.00
175	300	27773.25

Thus, Policy IV ($P = 175$, $Q = 250$) is the best amongst the five considered.

1-4. The basic nature of simulation

There does not seem to be much that is common between the two problems and the methods of solving them. The first problem (the pure pursuit) is basically continuous, in the sense that its state changes continuously with time; whereas the second problem is discrete, because the arrival and sale of merchandise occurs only in discrete steps. The first problem is deterministic, while the second one is stochastic in nature. Yet there are some common features and these are the features essential to simulation.

In both cases we started from a mathematical model of the system under study. Some initial conditions (state at time zero) were assumed in both cases. The change of state occurred in accordance with some equations (rules or laws). Using these rules numerically we continued changing the state of the system as time moved forward. This was done for as long a period as needed. At the end of this period we collected the desired information about the system (which is the solution to the problem). These calculations were programmed for a digital computer. Thus the computer was made to simulate or mimic the real-life system as its variables changed. Through this process we managed to get around the necessity of obtaining an analytic solution and therein lies the great advantage of simulation.

It should be noted that the simulation in both examples (pure pursuit as well as inventory control) could have been performed manually with pencil and paper. In theory any system that can be simulated on a digital computer can also be simulated by hand. In practice, however, simulation as an analytic tool is useful only when done on a computer. This is because the practical problems that require simulation are complex and need a very

large number of simple, repetitive calculations. In fact, simulation is one of the most important uses of the digital computer.

To simulate is to experiment: Simulation is basically an experimental technique. It is a fast and relatively inexpensive method of doing an 'experiment' on the computer. Consider, for example, the inventory-control problem. Instead of trying out in the actual store the five replenishment policies, each for a period of six months, and then selecting the best one, we conducted the experiment on the computer and obtained the results within a few minutes, at a very small cost (provided, of course, our model reflects reality). This is why computer simulation is often referred to as performing a *simulation experiment*.

No unified theory: There is no unifying theory of computer simulation. Learning simulation does not consist of learning a few fundamental theorems and then using them and their various corollaries to solve problems. There are no underlying principles guiding the formulation of simulation models. Each application of simulation is *ad hoc* to a great extent. In this sense simulation is an art, and one often hears the expression 'the art of simulation.'

1-5. When to simulate

All of us in our daily lives encounter problems, which although mathematical in nature, are too complex to lend themselves to exact mathematical analysis. The performance of such a system (say, weather or traffic jam) may be difficult to predict, either because the system itself is complex or the theory is not yet sufficiently developed. The difficulties in handling such problems (by means of classical mathematical tools) may also arise due to the effect of uncertainties, or due to dynamic interactions between decisions and subsequent events, or due to complex interdependencies among variables in the system, or due to some combination of these. Formerly (i.e., before the days of computer simulation), in such situations either an intuitive decision was made, or, if the stakes were too high to rely on intuition, elaborate laboratory experiments had to be conducted, which were usually expensive and time consuming. Simulation provides a third alternative which is cheap and fast, and thus fills the gap between exact analysis and physical intuition. Occasionally, simulation is also used even when an exact analytic solution is possible, but it is too expensive in terms of computation time.

Simulation in science and engineering research: Simulation has changed, in a very fundamental sense, the way in which research is conducted today. Earlier most experiments were carried out physically in the laboratories. Thousands and even millions were spent on physical models (e.g., wind tunnels, river basin models, network analyzers, aircraft flight simulators) and expensive experiments. Today a majority of these experiments are simulated on a computer. 'Computer experiments' besides being much faster, cheaper, and easier, frequently provide better insight into the system than laboratory experiments do. Not all laboratory experiments, of course,

can be replaced with computer simulations. Typically, a few key experiments are performed in the laboratory after, say, 80–90 per cent of the experimenting has been done on the computer.

Simulation in soft sciences: Simulation can be expected to play even a more vital role in biology, sociology, economics, medicine, psychology, etc., where experimenting could be very expensive, dangerous, or even impossible. In these areas, the mathematical theories are even less developed than in physical sciences. Moreover, in fields such as biology and economics the problems are truly large, involving tens of thousands of variables. The complications caused by uncertainty are also greater in these areas than in physical sciences. Thus simulation has become an indispensable tool for a modern researcher in most social, biological and life sciences.

Simulation for business executive: There are many problems faced by management that cannot be solved by standard operations research tools like linear and dynamic programming, inventory and queueing theory. Therefore, a business executive had to make decisions based solely on his intuition and experience. Now he can use computer simulation to make better and more meaningful decisions. Utilizing the power of a digital computer, he can build and study a simulation model (of his system) containing arbitrarily high-order complexities and a huge number of interdependencies, as well as uncertainties. Simulation has been used widely for inventory control, facility planning, production scheduling, and the like.

1-6. Remarks and references

Simulation is a very powerful, problem-solving technique. Its applicability is so general that it would be hard to point out disciplines or systems to which simulation has not been applied. The basic idea behind simulation is simple, namely, model the given system by means of some equations and then determine its time-dependent behaviour. The simplicity of approach when combined with the computational power of the high-speed digital computer makes simulation a powerful tool. Normally, simulation is used when either an exact analytic expression for the behaviour of the system under investigation is not available, or the analytic solution is too time-consuming or expensive.

Since the popularity and usefulness of simulation is dependent on the capabilities and availability of the computer, the subject essentially started in the late 1950's. It has grown—in its power, sophistication, and applicability—very rapidly in the past 15–20 years, and is still expanding at a rapid rate. One of the early textbooks and still a good one is:

TOCHER, K.D., *The Art of Simulation*, Van Nostrand Co., Princeton, N.J., 1963.

An extremely readable and elementary 100-page textbook on essentials of computer simulation is

SMITH, J., *Computer Simulation Models*, Hafner Publishing Company, New York, N.Y., 1968.

Some of the other general textbooks on simulation are:

EMSHOFF, J. R. and R. L. SISSON, *Design and Use of Computer Simulation Models*, Macmillan Co., New York, 1970.

GORDON, G., *System Simulation*, Prentice-Hall, Englewood Cliffs, N.J., 1969.

MARTIN, F.F., *Computer Modelling and Simulation*, John Wiley and Sons, New York, 1968.

MIZE, J. H. and J. G. COX, *Essentials of Simulation*, Prentice-Hall, Englewood Cliffs, N.J., 1968.

NAYLOR, T. H., J. L. BALINTFY, D. S. BURDICK, and K. CHU, *Computer Simulation Techniques*, John Wiley and Sons, New York, 1966.

SHANNON, R. E., *Systems Simulation: the Art and Science*, Prentice-Hall, Englewood Cliffs, N.Y., 1975.

FISHMAN, G. S., *Concepts and Methods in Discrete Event Digital Simulation*, John Wiley and Sons, New York, 1973.

ZEIGLER, B.P. *Theory of Modelling and Simulation*, Wiley-Interscience, New York, 1976.

MAISEL, H. and GNUGNOLI, *Simulation of Discrete Stochastic Systems*, Science Research Associates, Chicago, 1975.

The literature on simulation is vast. Numerous bibliographies and survey articles on simulation have been published, containing hundreds of references. Many of these references can be found in the textbooks given above.

Simulation requires several skills. You must be familiar with the system being simulated and know enough of system theory and modelling techniques to be able to make a realistic model of the system. The five different models of the pure-pursuit problem in Exercises 1-2 to 1-6 in the next section are included to emphasize the importance of modelling. Secondly, a certain amount of skill in computer programming is also required. Also, some knowledge of statistics is needed for designing the simulation experiment. Different textbooks lay different amounts of emphasis on these three aspects of simulation.

Besides general textbooks on simulation, there are a number of specialized texts devoted to application of simulation to a particular area such as economics, industrial systems, psychology, social sciences, international relations, ecology, and war games.

There have been a number of cases where more than a million dollars have been spent on a single simulation project. Many of the large successful simulation projects are never publicized because of their military application or their proprietary interest of private industry.

1-7. Exercises

1-1. Identify six different problems from your own experience that you think should be solved using digital simulation rather than analytically.

1-2. In the pure-pursuit problem of Sec. 1-1, suppose the initial (x, y) coordinates of the bomber at time 0 are $(0, 0)$ and of the fighter $(0, d)$. The bomber (the target) moves in a straight line along the x axis (east) at a constant velocity VB and the fighter at a velocity VF pointing itself continously

toward the target. Show that the path followed by the fighter can be described by the following equation:

$$x = \tfrac{1}{2}\left[\frac{y^{1+r}}{(1+r)\,d^r} - \frac{y^{1-r}}{(1-r)\,d^{-r}}\right] + \frac{r \cdot d}{1-r^2},$$

where $$r = \frac{VB}{VF},$$

and (x, y) are the coordinates of the fighter.

1-3. In the special case when the target moves along the x axis, using the FORTRAN program given in Sec. 1-1, find the point where the pursuer catches up with the target. Compare the simulation result with the one obtained using the analytic expression in Exercise 1-2. Explain the difference between the two results, if any.

1-4. The pure-pursuit situation in Sec. 1-1 can also be simulated in terms of polar coordinates (R, θ) instead of rectangular coordinates (x, y). The relevant equations (2-dimensional problem) are

$$\frac{dR}{dt} = VB \cdot \cos\theta - VF$$

$$R\frac{d\theta}{dt} = -VB \cdot \sin\theta$$

where R is the distance between the bomber and the fighter (denoted by DIST in Sec. 1-1) and θ is the angle of the line from the fighter to the bomber. Write a FORTRAN program using polar coordinates.

1-5. Write a program for simulating a 3-dimensional pure-pursuit problem.

1-6. In the pure-pursuit course the fighter flies directly toward the target. In a *lead-pursuit course* the fighter flies in a direction ahead of the line of sight by a lead angle φ. Write a computer program to simulate a 2-dimensional lead-pursuit course with a specified lead angle, the velocities, and the initial positions of the two aircraft.

1-7. In a field there are four animals—a dog, a mongoose, a snake, and a mouse. Dogs kill mongooses, mongooses kill snakes, and snakes kill mice. The speeds of the mouse, snake, mongoose and dog are, respectively, 8, 12, 18 and 30 km/hour. Simulate the chase with different starting positions to see which animal gets killed first.

1-8. Simulate on a computer the following game between two players (called the game of matching pennies). The two players simultaneously flip a coin each. If both sides are the same (both heads or both tails), player A wins a rupee from B, otherwise B wins a rupee from A. Each player starts with 10 rupees. By conducting the experiment 500 times, find out how long the game will last on the average before one of the players goes broke.

1-9. Suppose you are the manufacturer of a car which is in great demand. At present you make 1,000 cars per month and sell it easily at Rs. 20,000 each (ignore excise duty, sales tax, etc., for the time being). By using overtime, etc., you can increase the production, but you must also increase the price in order to make a profit (10 per cent on sale). The number of cars you can produce and the corresponding price (of all units) is given below:

Cars	Production	1,000	1,200	1,300	1,500	2,000
	Price	20,000	21,000	22,000	25,000	30,000

The demand for your cars is predicted to rise at 5 per cent per year provided the price stays at Rs. 20,000 each. As the price rises a certain number of customers will switch to other cars (or scooters). This switchover percentage is estimated to be as below:

Price	20,000	21,000	22,000	25,000	30,000
Percentage switched	0	5	10	25	50

By investing capital you can increase your production without increasing the cost per unit. But you must pay 8 per cent interest on the additional capital. The investments and the corresponding increases in production are as follows:

Investment in Rs.	1,000,000	2,000,000	5,000,000
Increase in production of cars	300	500	1,000

Simulate the company's situation over the next 20 years. By experimenting with the model find out which investment should be made and when (if any).

2

Simulation of Continuous Systems

From the viewpoint of simulation there are two fundamentally different types of systems: (i) systems in which the state changes smoothly or continuously with time are called *continuous systems*, and (ii) systems in which the state changes abruptly at discrete points in time are called *discrete systems*. The system of pure pursuit in Sec. 1-1 is an example of a continuous system, because the positions of the two aircraft can vary continuously with time. The inventory system in Sec. 1-3, on the other hand, is a discrete system because the addition to and the demand from the stock can occur only in discrete numbers. Hence the stock-level can change only discretely with time. A queueing situation is another example of a discrete system because the customers join or leave the queue only in discrete numbers. Usually, the simulation of most systems in engineering and physical sciences turns out to be continuous, whereas most systems encountered in operations research and management science are discrete. Most studies of communication, transportation, and computer systems also involve discrete system simulation. As we will see in this and subsequent chapters, the methodologies of discrete and continuous simulations are inherently different. In this chapter we will deal with continuous systems only.

Continuous dynamic systems, those systems in which the state or the variables vary continously with time, can generally be described by means of differential equations. If the set of (simultaneous) differential equations describing a system are ordinary, linear, and time-invariant (i.e., have constant coefficients), an analytic solution is usually easy to obtain. In general, differential equations of a more difficult nature can only be solved numerically. Simulating the system often gives added insight into the problem besides giving the required numerical solution. As an elementary example of a continuous dynamic system let us consider the following simple chemical plant.

2-1. A chemical reactor

In a certain chemical reaction when two substances A and B are brought together they produce a third chemical substance C. It is known that 1 gram of A combines with 1 gram of B to produce 2 grams of C. Furthermore, the rate of formation of C is proportional to the product of the amounts of A and B present. In addition to this *forward reaction* there is also a *backward reaction* decomposing C back into A and B. The rate of decomposi-

tion of C is proportional to the amount of C present in the mixer. In other words, at any time t if a, b, and c are the quantities of the chemicals A, B and C present, respectively, then their rates of increases are described by the following three differential equations:

$$\frac{da}{dt} = k_2 c - k_1 ab, \qquad \ldots(2\text{-}1)$$

$$\frac{db}{dt} = k_2 c - k_1 ab, \qquad \ldots(2\text{-}2)$$

$$\frac{dc}{dt} = 2k_1 ab - 2k_2 c, \qquad \ldots(2\text{-}3)$$

where k_1 and k_2 are the *rate constants*. (These constants will vary with temperature and pressure, but we do not allow the temperature or pressure of the reaction to vary.) Given the values of the constants k_1 and k_2 and the initial quantities of the chemicals A and B (and $c = 0$), we wish to determine how much of C has been produced as a function of time. Determination of the rate of such chemical reactions is important in many industrial processes.

A straightforward method of simulating this system is to start at time zero and increment time in small steps of Δt. We assume that the quantities of chemicals remain unaltered during each step and only change 'instantaneously' at the end of the step. Thus the quantity of A (or B or C) at the end of one such step is given in terms of the quantity at the beginning of the step as

$$a(t + \Delta t) = a(t) + \frac{da(t)}{dt} \cdot \Delta t \qquad \ldots(2\text{-}4)$$

If Δt is sufficiently small Eq. (2-4) is a reasonable representation. Identical equations can be written for $b(t + \Delta t)$ and $c(t + \Delta t)$.

Suppose we wish to simulate the system for a period T. We will divide this period T into a large number N of small periods Δt. That is

$$T = N . \Delta t$$

At time zero, we know $a(0)$, $b(0)$, $c(0)$. From these initial values and the values k_1 and k_2 we compute the amounts of chemicals at time Δt as

$$a(\Delta t) = a(0) + [k_2 . c(0) - k_1 . a(0) . b(0)] \Delta t$$

$$b(\Delta t) = b(0) + [k_2 . c(0) - k_1 . a(0) . b(0)] \Delta t$$

$$c(\Delta t) = c(0) + [2k_1 . a(0) . b(0) - 2k_2 . c(0)] \Delta t$$

Using these values we calculate the next state of the system, i.e., at time $2t$ as

$$a(2\Delta t) = a(\Delta t) + [k_2 . c(\Delta t) - k_1 a(\Delta t) . b(\Delta t)] . \Delta t$$

$$b(2\Delta t) = b(\Delta t) + [k_2 . c(\Delta t) - k_1 a(\Delta t) . b(\Delta t)] . \Delta t$$

$$c(2\Delta t) = c(\Delta t) + [2k_1 . a(\Delta t) . b(\Delta t) - 2k_2 . c(\Delta t)] . \Delta t$$

Using the state of the system at $2\,\Delta t$, we determine its state at $3\,\Delta t$, and so on. We continue in this vein, moving time forward by Δt and determining the state of the system from the previous state, for N steps, at the end of which we have the desired result. This procedure is shown in the form of a flow-chart in Fig. 2-1.

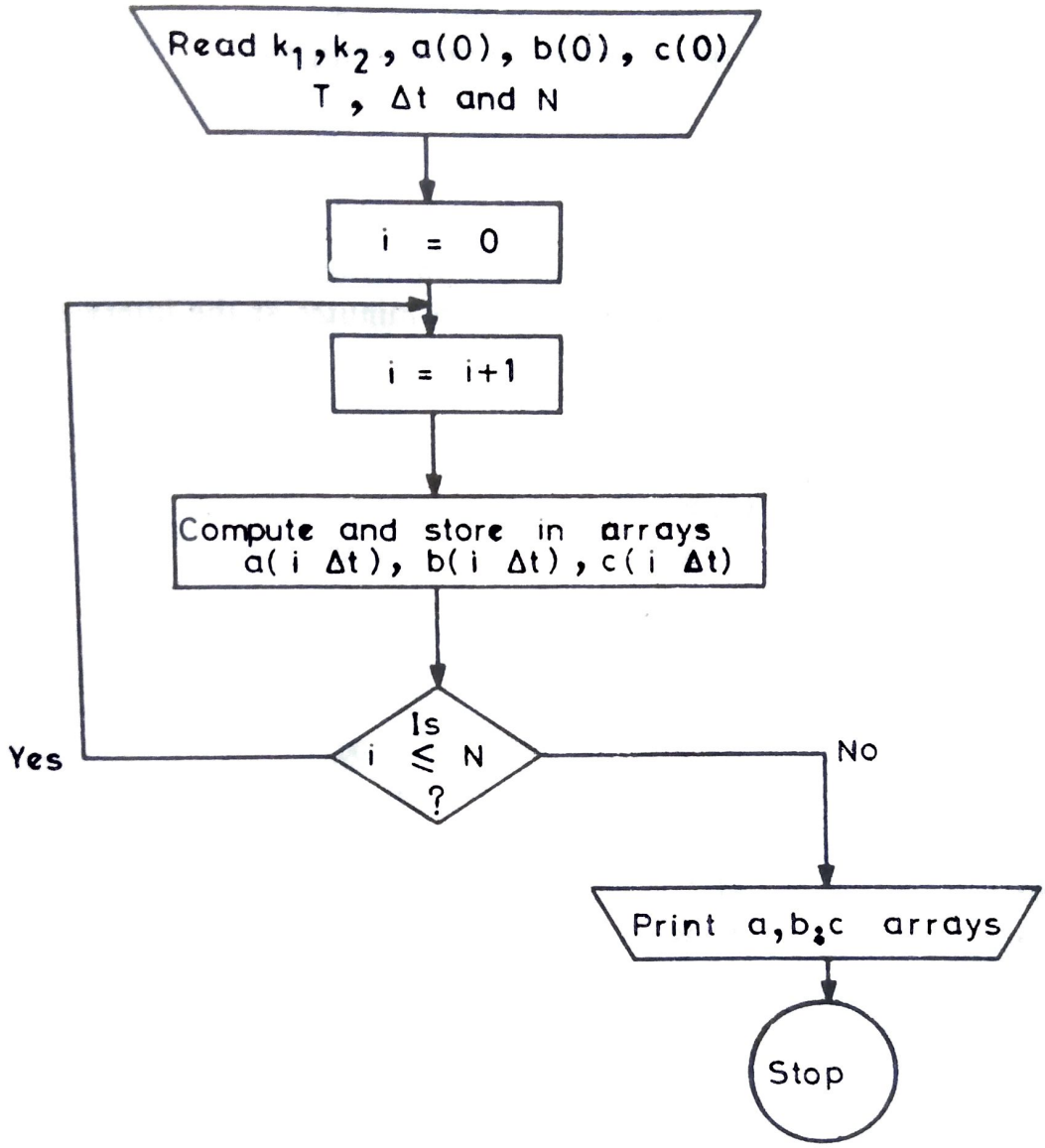


Fig. 2-1: Flowchart of a chemical reaction simulator.

A FORTRAN (format free) program which simulates the system with $k_1 = 0.008$ gram^{-1} min^{-1}, $k_2 = 0.002$ min^{-1}, and $a(0) = 100$ grams, $b(0) = 50$ grams, $c(0) = 0$ grams, for a period of $T = 5$ minutes in steps of $\Delta t = 0.1$ min. is as follows ($N = 50$).

```
      DIMENSION A (52), B (52), C(52),
      REAL K1, K2
      A (1) = 100.0
      B (1) = 50.0
      C (1) = 0.0
```

```
      DELTA = 0.1
      T = 0
      K1 = 0.008
      K2 = 0.002
      DO 3 I = 1,51
      PRINT, T, A(I), B(I), C(I)
      A(I + 1) = A(I) + (K2*C(I) - K1*A(I)*B(I))*DELTA
      B(I + 1) = B(I) + (K2*C(I) - K1*A(I)*B(I))*DELTA
      C(I + 1) = C(I) + 2.*(K1*A(I)*B(I) - K2*C(I))*DELTA
      T = T + DELTA
    3 CONTINUE
      STOP
      END
```

The following is the output of this program, which gives the state of the system (i.e., the values of *a*, *b*, and *c*) for 5 minutes at the intervals of 0.1 minute.

TIME	A(I)	B(I)	C(I)
0.00	100.00	50.00	0.00
0.10	96.00	46.00	8.00
0.20	92.47	42.47	15.06
0.30	89.33	39.33	21.34
0.40	86.52	36.52	26.95
0.50	84.00	34.00	32.00
0.60	81.72	31.72	36.55
0.70	79.66	29.66	40.69
0.80	77.77	27.77	44.45
0.90	76.05	26.05	47.89
1.00	74.48	24.48	51.04
1.10	73.03	23.03	53.94
1.20	71.70	21.70	56.61
1.30	70.46	20.46	59.07
1.40	69.32	19.32	61.36
1.50	68.26	18.26	63.48
1.60	67.28	17.28	65.44
1.70	66.36	16.36	67.28
1.80	65.51	15.51	68.99
1.90	64.71	14.71	70.59
2.00	63.96	13.96	72.08

(Continued...)

(. . . *Continued*)

TIME	A(I)	B(I)	C(I)
2.10	63.26	13.26	73.48
2.20	62.60	12.60	74.79
2.30	61.99	11.99	76.03
2.40	61.41	11.41	77.18
2.50	60.86	10.86	78.27
2.60	60.35	10.35	79.30
2.70	59.87	9.87	80.27
2.80	59.41	9.41	81.18
2.90	58.98	8.98	82.04
3.00	58.57	8.57	82.86
3.10	58.19	8.19	83.63
3.20	57.82	7.82	84.36
3.30	57.48	7.48	85.05
3.40	57.15	7.15	85.70
3.50	56.84	6.84	86.32
3.60	56.55	6.55	86.91
3.70	56.27	6.27	87.46
3.80	56.00	6.00	87.99
3.90	55.75	5.75	88.50
4.00	55.51	5.51	88.97
4.10	55.29	5.29	89.43
4.20	55.07	5.07	89.86
4.30	54.86	4.86	90.27
4.40	54.67	4.67	90.66
4.50	54.48	4.48	91.03
4.60	54.31	4.31	91.39
4.70	54.14	4.14	91.73
4.80	53.98	3.98	92.05
4.90	53.82	3.82	92.35
5.00	53.68	3.68	92.65

2-2. Numerical integration vs. continuous system simulation

Some of you may have recognized that all we have done for studying the dynamics of the reaction rates in Sec. 2-1 is to use Euler formula to perform numerical integration of three simultaneous differential equations, Eqs

(2-1)–(2-3). What then, you might ask, is the difference between an ordinary numerical integration and a continuous system simulation? What is it that makes one study of a dynamic system a computation and another study, a simulation? Although the dividing line between simulation and ordinary computation often becomes quite thin, there are two distinctive features of simulation : (i) In simulation, we always keep track of the state of the system explicitly. The outcome of each step (of numerical calculations) in a simulation experiment can be interpreted directly as the state of the system at some point in time. Simulation, essentially, consists of constructing a state history of the system—a succession of explicit state descriptions at each instant as we move forward in time. Thus there is a one-to-one correspondence between what the computer does and what takes place in the system. In a numerical solution of equations no such correspondence is preserved. Usually in pure computations shortcuts are taken, parameters are lumped and mathematical equations manipulated before the computer program is developed. These destroy the one-to-one correspondence between the computer steps and the original system from which the equations were derived. Consequently the output data have to be interpreted in the light of earlier manipulations before conclusions can be drawn about the system. (ii) Secondly, there is also a difference of attitude. In case of a pure numerical calculation we only see the given set of differential equations as a mathematical object and proceed to integrate them. In simulation we look upon the equations as one of the steps in the entire process. We know the real-life system, and we are aware of the approximations in the model that is being simulated. Finally, by looking at the output data (which directly represent the dynamics of the system) we are also prepared to modify the model, if necessary.

2-3. Selection of an integration formula

As we observed earlier, continuous dynamic systems are generally represented by means of differential equations. Numerical solution of these differential equations involves some integrating procedure. Many different integration formulas are available for this purpose, and the selection of an appropriate formula for integration is the most crucial decision to be made in a continuous system simulation. In Sec. 2-1 we used the simplest possible integration formula, which is known as Euler formula. There are much more efficient ways of performing numerical integration which do not rely simply on the last known values of the variables. Instead, some of them use several previous values to predict the rate at which the variables are changing and also in some formulas the integration step size Δt is adjusted to match the rate at which the variables are changing. As a matter of fact, the simple Euler formula employed in Sec. 2-1 is rarely used in practice because of the rapid accumulation of errors (to be discussed shortly). There are improved versions of the Euler formula available, such as, Euler-

Richardson formula, Euler-Gauss formula. Some of the integration formulas commonly used in simulation are: Simpson's rule ($\frac{1}{3}$ rule, 3/8 rule); trapezoidal rule; the Runge-Kutta family of formulas (second, third or fourth order); predictor-corrector methods; and so on. Many of these are available as standard subroutines. There is no integration method which is the best for simulations. One has to consider several factors when choosing an integration formula. Accuracy and the speed of computation are the most important considerations. Other factors are self-starting ability, solution stability, presence or absence of discontinuity, and ease with which error can be estimated. Any detailed discussion of these is beyond the scope of this book. The following is a very brief excursion.

Errors: There are basically two types of computation errors that occur at each step of integration: (i) *Round off* errors are introduced because of the limited size of the computer word. Every number has to be represented within this size. For example, suppose a number obtained as a product of two 4-digit numbers

$$.2102 \times .8534 = .17938468$$

has to be accommodated within 4-digits. It is therefore rounded off to .1794, introducing thereby an error. Some computer systems (compilers such as ALPS, BASIC and some versions of FORTRAN) have automatic rounding built into them, but many systems do not round off; they simply *chop off*; which is even worse. Thus the product in the foregoing example would end up being 0.1793. (ii) *Truncation errors* are caused when an infinite mathematical series is approximated by a finite number of terms. In continuous system simulation truncation errors arise mainly due to the inadequacy of an integration formula being used. For example, when Euler integration formula is used we are using only the first two terms of the Taylor series (which is infinite).

$$f(t+\Delta t) = f(t) + \frac{df(t)}{dt} \cdot \Delta t + \frac{d^2f(t)}{dt^2} \cdot \frac{(\Delta t)^2}{2!} + \frac{d^3f(t)}{dt^3} \frac{(\Delta t)^3}{3!} + \cdots$$

when Δt is sufficiently small the truncation error in Euler formula can be approximated with

$$\frac{d^2f(t)}{dt^2} \cdot \frac{(\Delta t)^2}{2}$$

The second derivative can be approximated to

$$\frac{d^2f(t)}{dt^2} = \frac{1}{\Delta t}\left[\frac{df(t+\Delta t)}{dt} - \frac{df(t)}{dt}\right]$$

On dividing the error term with value of $f(t)$ to get a relative error and on getting rid of the sign, we get

$$\text{Relative error} = Er = \left[\frac{df(t+\Delta t)}{dt} - \frac{df(t)}{dt}\right]\frac{\Delta t}{2f(t)} \qquad (2\text{-}5)$$

In each individual step *Er* may be small but when accumulated over hundreds of consecutive steps in an integration the truncation error could make the simulation results meaningless.

Integration step: Choice of the step size Δt of integration is another very important decision. The smaller the integration step the smaller is the error. The relative integration error in using Euler formula, for example, is given by Eq. (2-5). Clearly *Er* can be made as small as we please by making Δt sufficiently small. But the number of computation steps and therefore the computation time will increase inversely in proportion to Δt. More steps would also accumulate an increased amount of round off errors. A compromise has to be made between the conflicting requirements of speed and accuracy. For a given accuracy, it is often possible to arrive at an optimal combination of an integration formula and the step size.

Sometimes, when the error varies widely, it is advisable to use a *varying-step integration* formula. That is, the error is evaluated at each integration step and the step size Δt for the next step automatically adjusted accordingly.

It often happens that the time intervals for which we require the output of integration are much larger than the integration step size. In that case the program is so written that values of the variables are saved once, say, every 100 steps of integration. This will be illustrated in Sec. 2-5.

2-4. Runge-Kutta integration formulas

In Euler formula (used in Sec. 2-1) to estimate the value of the variable at time $(t + \Delta t)$ we used its slope at time t,

$$y(t + \Delta t) = y(t) + \frac{dy(t)}{dt} \cdot \Delta t.$$

In other words, the value of y at $(i + 1)$th instant is estimated in terms of its value and slope at the ith instant. That is,

$$y_{i+1} = y_i + \dot{y} \cdot \Delta t$$

where $\dot{y}$ is the usual short-hand notation for the first derivative of y.

As mentioned earlier, Eular formula is the simplest and crudest method of numerical integration. It is not very efficient because it requires a very small step-size Δt for reasonable accuracy.

A much more refined, accurate, and commonly used method is the Runge-Kutta method. It is a method designed to approximate the Taylor series without our having to evaluate the higher order derivatives. The main idea is to compute the first derivate $f(t, y)$ of y at several carefully chosen points in the interval $(t, t + \Delta t)$, and to combine these values in such a way as to get good accuracy in the computed increment $y_{i+1} - y_i$. Based on this strategy, a family of formulas have been derived, known as Runge-Kutta formulas. The best amongst these, and by far the most popularly used

integration formula, is the fourth-order Runge-Kutta formula, defined by the following equations:

$$\left.\begin{aligned} u_1 &= h \cdot f(t_i, y_i) \\ u_2 &= h \cdot f(t_i + h/2,\ y_i + u_1/2) \\ u_3 &= h \cdot f(t_i + h/2,\ y_i + u_2/2) \\ u_4 &= h \cdot f(t_i + h,\ y_i + u_3) \\ y_{i+1} &= y_i + \tfrac{1}{6} \cdot (u_1 + 2u_2 + 2u_3 + u_4) \end{aligned}\right\} \quad \ldots(2\text{-}6)$$

where function f is the first derivative of y at various points, and h denotes the integration step-size Δt.

The equations in (2-6) appear quite complicated at first sight, but they are in fact easy to program. The function f is computed four times in each integration step. The truncation error in using the 4-th order Runge-Kutta method is of the order of h^5, which is far better than order h^3, the error Er in the case of the Euler formula. We can, therefore, use a larger step-size with Eq. (2-6). The price we pay for reduction in error is that four additional function evaluations are required per step. This price may be considerable in computer time if the function $f(t, y)$ is complicated. Like the Euler method, Runge-Kutta methods are *self-starting* (i.e., the computation of the solution can be started with no values other than the initial conditions of the differential equations). The derivation of the Runge-Kutta formulas or any further discussion is outside the scope of this book. There are several good texts on numerical analysis that treat this topic in detail. We will simply illustrate its use in continuous simulation with two examples.

An R-C amplifying circuit: Consider an electronic circuit of Fig. 2-2 where Amp is an ideal voltage amplifier with infinite input impedance and zero output impedance, and amplification factor A. The input capacitor C_1 is a coupling capacitor of a large value, say, $C_1 = 1\mu f$. Capacitor C_2 is a stray capacitance, of a small value, say, $C_2 = .005\ \mu F$. The values of the three resistors are $R_1 = 10^4$ ohms, $R_2 = 10^3$ ohms and $R_3 = 8 \times 10^3$ ohms. Let $A = 100$.

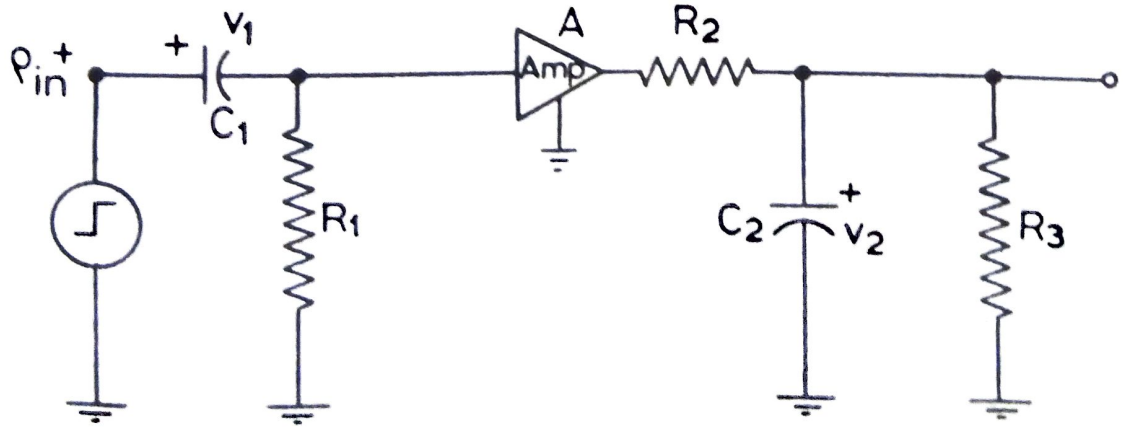


Fig. 2-2: R-C amplifying circuit.

This system is described by the following two equations: The current entering the capacitor C_1 is given by

$$C_1 \frac{dv_1}{dt} = (e_{in} - v_1) \cdot \frac{1}{R_1},$$

and the current entering capacitor C_2 is

$$C_2 \frac{dv_2}{dt} = \frac{A}{R_2}(e_{in} - v_1) - \frac{v_2(R_2 + R_3)}{R_2 R_3}$$

Hence

$$\dot{v}_1 = A_{11} v_1 + B_1 e_{in} \qquad \ldots(2\text{-}7)$$

$$\dot{v}_2 = A_{21} v_1 + A_{22} v_2 + B_2 e_{in}, \qquad \ldots(2\text{-}8)$$

where constants

$$A_{11} = -\frac{1}{R_1 C_1} = -100 \text{ sec}^{-1} = -B_1$$

$$A_{21} = -\frac{A}{R_2 C_2} = -2 \times 10^7 \text{ sec}^{-1} = -B_2$$

$$A_{22} = -\frac{R_2 + R_3}{R_2 R_3 C_2} = -2.25 \times 10^5 \text{ sec}^{-1}$$

The state of the system is given by the voltages v_1 and v_2.

Let us simulate the system and solve the two simultaneous equations, Eq. (2-7), (2-8). Assume that initially $v_1(0) = v_2(0) = 0$ and at time $t = 0$ a unit step voltage is applied at the input terminal. That is, $e_{in} = 1$ for $t \geqslant 0$ and $e_{in} = 0$ for $t < 0$.

As discussed in Sec. 2-3, there are two crucial decisions to be made when using numerical integration: about the integration formula and about the step-size Δt. Let us in this case choose the Runge-Kutta 4th order integration method. Regarding the step-size, since $C_1 \gg C_2$ initially the exponential rise in the output voltage (voltage v_2) will be very fast, at a time constant of

$$C_2 \left(\frac{R_2 R_3}{R_2 + R_3} \right) = 4.4 \ \mu \text{sec},$$

whereas the output voltage will decay at a time constant of

$$C_1 R_1 = 10 \text{ m sec}.$$

Thus the ratio of the time constants is more than 2200. Let us therefore choose the step-size initially to be $h = \Delta t = 0.5 \ \mu$ sec (about 1/9th of the time constant). After, say, 200 iterations (about 23 time constants), we can increase the integration step-size. Can we increase h by, say, a factor of 2000? The investigation of this problem is left as an exercise for you. The following is a FORTRAN program which simulates this system, using the 4th order Runge-Kutta integration method: (Note that initially the step-size is set to a smaller value HS, and after N1 steps it is increased to a larger value HL and then run for N2 steps):

```
      READ, A11, A21, A22, B1, B2, N1, N2, HS, HL
      T = 0.
      V1 = 0.
      V2 = 0.
      H = HS
      N = N1 + N2
      DO 130 I = 1, N
      IF(I.GT.N1) GO TO 120
C     EVALUATE RUNGE-KUTTA TERMS—4 FOR EACH EQUATION
100   U11 = H*((A11* V1) + E1)
      U12 = H*((A21* V1) + (A22*V2) + B2)
      U21 = H*(A11*(V1 + .5*U11) + B1)
      U22 = H*(A21*(V1 + .5*U11) + A22*(V2 + .5*U12) + B2)
      U31 = H*(A11*(V1 + .5*U21) + B1)
      U32 = H*(A21*(V1 + .5*U21) + A22*(V2 + .5*U22) + B2)
      U41 = H*(A11*(V1 + U31) + B1)
      U42 = H*(A21*(V1 + U31) + A22*(V2 + U32) + B2)
C     EVALUATE VOLTAGES
      V1 = V1+(U11+2.*U21+2.*U31+U41)/6.
      V2 = V2 + (U12 + 2.*U22 + 2.*U32 + U42)/6.
      T = T + H
110   PRINT, I, T, V1, V2
      GO TO 130
C     STEP-SIZE INCREASES
120   H = HL
      GO TO 100
130   CONTINUE
      STOP
      END
```

2-5. Simulation of a servo system

A very important application of continuous system simulation is in design and analysis of control systems. Let us study the behaviour of a second-order nonlinear feedback system represented by the following block diagram.

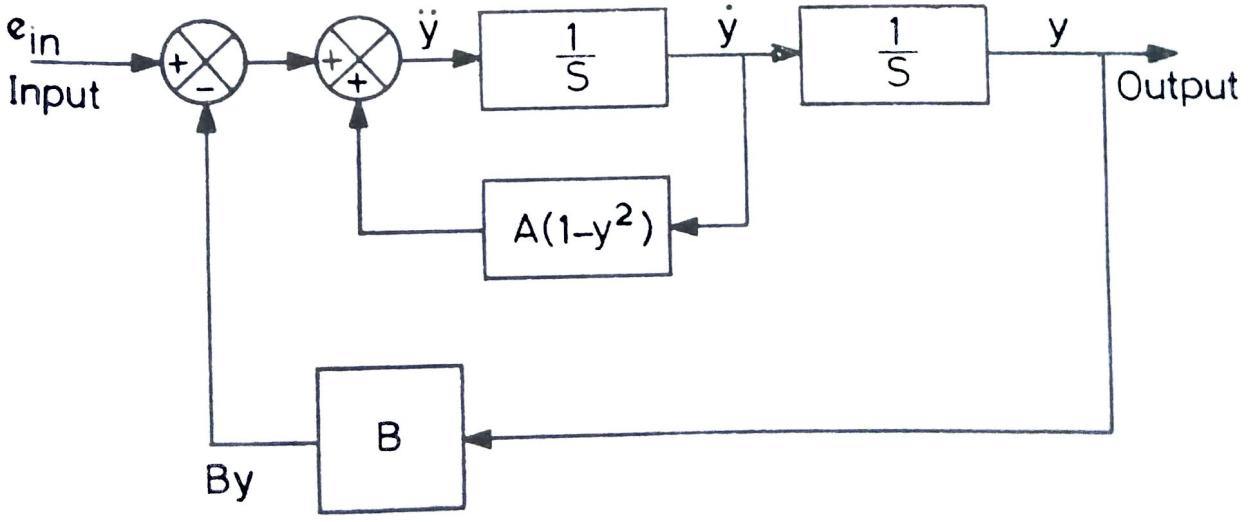


Fig. 2-3: A nonlinear second-order servo system.

(Those of you not familiar with the control theory symbols may ignore the diagram and simply start from the differential equation.)

This block diagram represents many natural as well as man-made servo systems. Some examples are: beating of the heart, periodic opening and closing of flowers in response to the sunlight, rate of variation of prices, squeaking of door with rusty hinges, dripping of a leaky tap, a neon-lamp oscillator, to name a few.

The system of Fig. 2-3 can also be described by the following differential equation

$$\ddot{y} = A(1 - y^2)\dot{y} - By + e_{in}$$

where A and B are positive constants.

In the case of zero input signal, the equation becomes

$$\ddot{y} = A(1 - y^2)\dot{y} - By \qquad \ldots(2\text{-}9)$$

This is the well-known Van der Pol non-linear equation. (It can be seen that when the amplitude y is small, the damping term $A(1 - y^2)$ is negative, but when y becomes large the damping becomes positive. Thus small-amplitude oscillations will tend to build up, while large-amplitude oscillations will be damped out.)

Let us simulate this system. The second-order differential equation can be written as a set of two simultaneous equations of first order. We replace Eq. (2-9) with (using the variable y_1 in place of y)

$$\dot{y}_1 = y_2$$

$$\dot{y}_2 = A(1 - y_1^2)y_2 - By_1$$

To be more specific let constants $A = 0.1$, $B = 1.0$ and let the initial conditions be

$$y_1(0) = 1.0$$

and

$$y_2(0) = 0.$$

Our equations therefore become

$$\dot{y}_1 = y_2$$

and

$$\dot{y}_2 = 0.1(1 - y_1^2)y_2 - y_1 \qquad \ldots(2\text{-}10)$$

An instantaneous description of the state of the system is given by the outputs of the two integrators, i.e., by variables y_1 and y_2. We will use, once again, the fourth-order Runge-Kutta method to obtain the values of y_1 and y_2 as a function of time. We will choose the step-size $\Delta t = H = .001$ second and simulate the system for 5 seconds. Thus the number of steps will be N = 5,000. This is too large a number of outputs to be plotted or examined. We will, therefore, print out the values of y_1 and y_2 only once every 100 integration steps. This can be implemented by keeping a counter K which is decremented by 1 for each integration step. Every time K equals zero,

y_1 and y_2 are printed and K is reset to 100. The following FORTRAN program performs the simulation.

```
C     H IS TIME STEP, N IS NO. OF STEPS, Y1, Y2 INITIAL VALUES
      T = 0.
      Y1 = 1.
      Y2 = 0.
      H = .001
      N = 5000
      K = 1
      DO 120 I = 1, N
      K = K - 1
      IF (K.NE.O) GO TO 110
C     PRINT ONCE IN 100 STEPS
      PRINT, T, Y1, Y2
      K = 100
C     CALCULATE THE RUNGE-KUTTA TERMS
  110 U11 = H*Y2
      U12 = H*(.1*(1. - Y1*Y1)*Y2 - Y1)
      U21 = H*(Y2*+ .5*U12)
      U22 = H*(.1*(1.-(Y1+.5*U11)*(Y1+.5*U11))*(Y2+.5*U12)-(Y1+.5*U11))
      U31 = H*(Y2 + .5*U22)
      U32 = H*(.1*(1. - (Y1+.5*U21)*(Y1+.5*U21))*(Y2+.5*U22)-(Y1+.5*U21))
      U41 = H*(Y2 + U32)
      U42 = H*(.1*(1. - (Y1 + U31)*(Y1 + U31))*(Y2 + U32) - (Y1 + U31))
C     CALCULATE Y1 AND Y2
      Y1 = Y1 + (U11 + 2.*U21 + 2.*U31 + U41)/6.
      Y2 = Y2 + (U12 + 2.*U22 + 2.*U32 + U42)/6.
      T = T + H
  120 CONTINUE
      PRINT, T, Y1, Y2
      STOP
      END
```

By studying the three examples in this chapter so far, you may have acquired the incorrect impression that (i) every dynamic continuous system must first be expressed as a set of simultaneous differential equations before being simulated and, that (ii) such a system is always deterministic. The next example is meant precisely to dispel this notion. Moreover, in this example we are simulating a system which is too large and expensive to experiment with and where an incorrect design could become very costly.

2-6. Simulation of a water reservoir system

Let us consider the following proposal for constructing a dam across a river to create a reservoir. The reservoir is to be constructed at a specified site. The curve of the projected demand for the water from the reservoir has been determined (from the expected growth pattern and the seasonal fluctuations). The input to the reservoir is from the river inflow and from the rainfall directly over the reservoir. The output consists of the seepage and evaporation losses, in addition to the water supplied to meet the pro-

jected demand. This system (called a simple run-of-river storage demand system) is represented symbolically in Fig. 2-4.

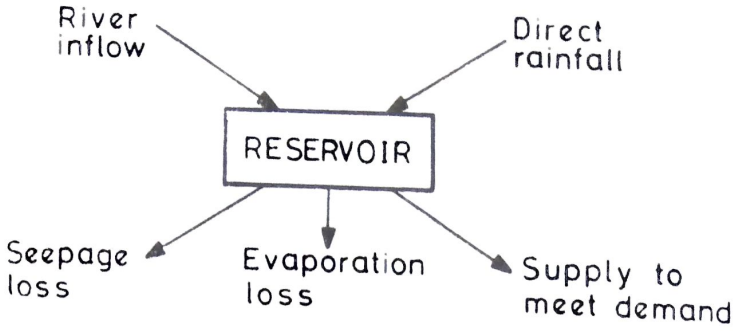


Fig. 2-4: A simple run-of-river storage demand system.

The amount of seepage loss is not a constant but depends on the volume of the water stored. We have been given a curve (converted into a table) showing the seepage loss as a function of volume for the proposed reservoir. Likewise, the evaporation loss depends on the area of the exposed surface and the coefficient of evaporation. We are given another curve showing the surface area as a function of volume as well as the seasonal variation of the coefficient of evaporation. Therefore, for a given volume of water in the reservoir at a particular time of the year we can calculate the two losses.

In reality no reasonable finite-sized reservoir can provide an absolute guarantee of meeting the demand 100% of the time because the river inflow, the rainfall, the losses, the demand are all random variables. To build such a large dam which will never fail (to meet the demand) through its entire life will generally be uneconomical. Therefore, in practice one determines the reservoir size which will meet the demand with a specified risk of failure (of water shortage). For example, a 2% failure means that once in 50 years the reservoir would become empty before meeting the demand for water. The objective of the study is to determine the size of the reservoir with a specified risk of failure.

There is a single state variable in this system, namely, the volume of water in the reservoir. Since the volume varies continuously with time, we are dealing with a continuous system. It is reasonable to take one month as the basic time interval for the simulation study. Thus, for example, if we wish to simulate the system for 100 years, the simulation run length will be 1200. The simulation will be repeated assuming several different capacities of the reservoir. The output will be in series of ranked shortages for each capacity.

The basic procedure, to be repeated for each time step, may be expressed in terms of the following steps:

(1) For the current month M of the current year IY determine the total amount of river inflow and the total rainfall directly over the reservoir. Let the sum of two inputs be denoted by VIN (= RAIN + RFLOW).

(2) Add the input volume VIN to the volume left over in the reservoir at the end of the last month, VOL $(m-1)$. This gives us the gross volume, GROSSV = VIN + VOL $(m-1)$.

(3) On the basis of the last month's volume VOL $(m-1)$ calculate this month's seepage and evaporation losses and add them as total loss TLOSS = SEEP + EVAP.

(4) From the demand curve (stored as a table in the computer memory) determine the demand of water for the current month DEM.

(5) If the TLOSS $\geqslant$ GROSSV, then the reservoir runs dry without supplying any water and therefore shortage, SHORT = DEM. The volume of water left at the end of the current month VOL $(m) = 0$. Spillage SPILL = 0 and go to Step 8; else if TLOSS $<$ GROSSV then the net water volume available to satisfy the demand is VNET = GROSSV − TLOSS.

(6) If DEM $\geqslant$ VNET the reservoir runs dry and the shortage is given by SHORT = DEM − VNET, and SPILL = 0. Go to Step 8; else if DEM $<$ VNET, then the difference DIFF = VNET − DEM is the water left over.

(7) If this leftover water exceeds the capacity CAP of the reservoir there will be a spill over, i.e., if DIFF $>$ CAP then SPILL = DIFF − CAP and VOL (m) = CAP; else if DIFF $\leqslant$ CAP then SPILL = 0 and VOL (m) = DIFF.

(8) Print out SPILL and SHORT for this month, and move to the next month. If the period exceeds the intended simulation length stop, else go to Step 1.

The following format-free FORTRAN program implements these steps:

```
READ, N, VOL, CAP
IY = 1
M = 1
DO 30 IY = 1, N
DO 30 M = 1, 12
SPILL = 0.
SHORT = 0.
CALL RIVFLO (IY,M,RFLOW)
CALL RAINF (IY,M,RAIN)
VIN = RFLOW + RAIN
GROSSV = VOL + VIN
CALL SEPEJ (VOL,SEEP)
CALL EVPRSN (M,VOL, EVAP)
TLOSS = SEEP + EVAP
CALL DEMAND (IY,M,DEM)
VNET = GROSSV − TLOSS
IF (VNET .LE. 0.) VNET = 0.
DIFF = VNET − DEM
IF (DIFF .GE. 0.) GO TO 10
SHORT = −DIFF
```

```
      VOL = 0.
      GO TO 20
   10 VOL = DIFF
      IF (CAP .GT. DIFF) GO TO 20
      SPILL = DIFF - CAP
      VOL = CAP
   20 PRINT, FLOW, RAIN, TLOSS, SHORT, SPILL, VOL
   30 CONTINUE
      STOP
      END
```

Subroutines

The foregoing program contains five subroutines requiring data about the riverflow, rainfall and the demand as a function of time; the seepage loss as a function of water stored in the reservoir; and the evaporation loss as a function of the volume (and hence the exposed surface) and the particular month of the year. The long sequence of data for river inflow and the rainfall could either be obtained from historical records or generated using suitable pseudorandom number generators. Similarly we can design the other subroutines, from an intimate knowledge of the system. As an example, we will write down the subroutine EVPRSN, for computing the evaporation loss.

Let the unit of measuring the volume be a million cubic meters. Suppose the highest possible dam at this site will create a reservoir of capacity 1000 units. Also suppose that we have a curve that gives the exposed surface area as a function of volume from 0 to 1000 units. Let the x-axis be divided into 100 equispaced ranges; and the data be stored in the form of a table (SURTBL) with 100 columns giving the surface area at volume 10, 20, ..., 1000 units. The surface area SAREA for any intermediate value of VOL can be computed using an appropriate interpolation formula. Let us assume that a linear interpolation will suffice. We are also given 12 values for the coefficient of evaporation COEF—one for each month of the year. Then the following subroutine will yield the evaporation loss.

```
      SUBROUTINE EVPRSN (M, VOL, EVAP)
      REAL SURTBL (100), COEF (12)
      DATA SURTBL /..., ..., ... /
      DATA COEF /..., ..., ... /
      IVOL = VOL/10.
      RVOL = IVOL
      FRAC = VOL/10. - RVOL
      SAREA = SURTBL (IVOL) + FRAC* (SURTBL(IVOL+1)-SURTBL(IVOL))
      EVAP = SAREA * COEF (M)
      RETURN
      END
```

Note that the third statement requires 100 values and the fourth statement requires 12 values. Other subroutines can be written down similarly.

Output: The output will be a series of monthly shortages and spills.

The shortages could be combined into total annual shortages. These annual shortages can be ranked according to the amount of shortage involved. From this ranked series of shortages for each capacity of the reservoir we would determine the acceptable reservoir size.

In the foregoing model no distinction was made regarding how the shortage is distributed month-wise within a particular year. For example, the total failure to meet any demand for one month may be more serious than a 10 per cent shortage for ten consecutive months. Such refinement can be easily incorporated by a procedure of assigning weights to different types of failures within a particular year.

There is another refinement which should be made in our computation. The losses for the entire current month were computed on the basis of the volume in storage at the end of last month. If a large change in volume takes place between two consecutive time steps, this would lead to errors. This could be corrected by first finding a temporary value of VOL (m) on the basis of given VOL $(m-1)$ (as usual) and then recalculating the losses on the basis of the average value

$$\frac{\text{VOL}(m) + \text{VOL}(m-1)}{2}$$

of the two volumes.

2-7. Analog vs. digital simulation

Block-diagram programming system: Let us once again consider the continuous, dynamic nonlinear system described by the van der Pol equation in Sec. 2-5. The simulation program for that system can be expressed by means of the following block diagram.

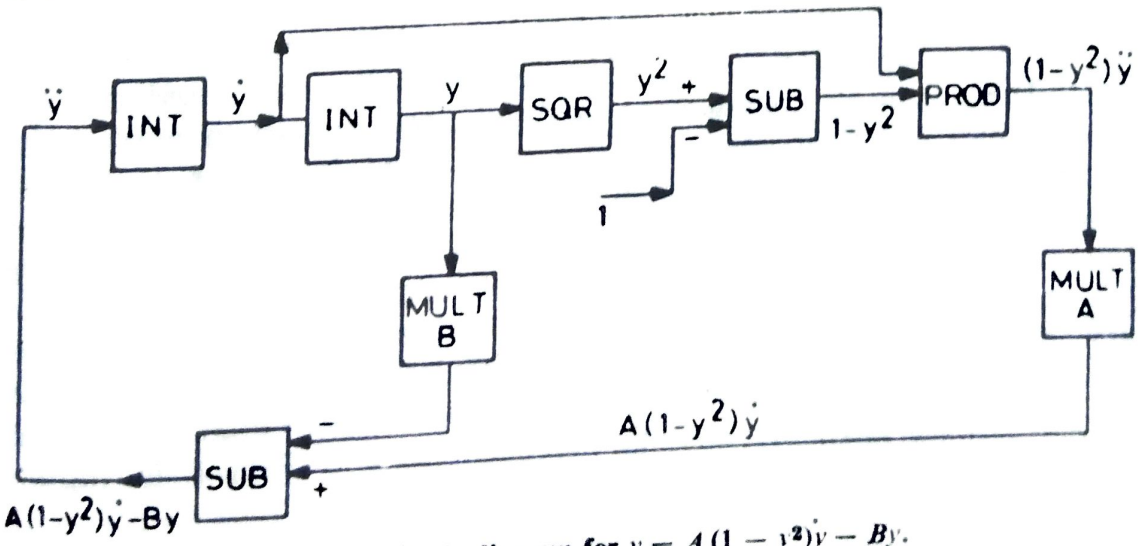


Fig. 2-5: Block diagram for $\ddot{y} = A(1 - y^2)\dot{y} - By$.

Each of the blocks performs a mathematical operation. Block INT performs integration, SQR squares, PROD takes the product, MULT multiplies with a constant, and SUB subtracts. The blocks may be looked upon as

subroutines. The interconnection of the blocks is self-explanatory. Note that the variables $\dot{y}$, $\dot{y}$, y, y^2, etc., vary with time and must be updated every H (i.e., Δt) seconds. The complete sequence of operations (by these blocks) is, therefore, repeated every H seconds if we use Euler integration (or four times every H seconds if we employ the 4th order Runge-Kutta integration).

Block-diagram oriented languages: Any continuous system (requiring numerical solution of differential equations), however complex, can be simulated by suitably 'patching' together a number of blocks such as these. Different systems will vary only in the number of blocks required and in their interconnections. Since the same few types of blocks are used again and again for simulation of all continuous systems, it would be very useful to provide a package of standard subroutines to perform the operations of these blocks. It would be even more convenient to have a special-purpose programming language which allowed direct implementation of block diagrams such as the one in Fig. 2-5. Development in the simulation area has taken place precisely in this direction. A large number of such continuous system simulation languages have been designed and implemented. One or more of these languages are available at almost every computing centre. These languages will be discussed in greater detail in Chapter 9.

Analog implementation: Theoretically speaking, the means by which each of the blocks is actually implemented is immaterial, as long as it does its assigned task so that the entire set-up gives us the state of the system as a function of time. These block functions may all be performed by one digital computer or by, say, different microprocessors—each specially programmed to do a specific job.

A block diagram, such as in Fig. 2-5, can also be simulated on an (electronic) analog computer, which consists of special-purpose elements such as integrators, multipliers, adders, function generation, and nonlinear elements. These elements are interconnected to imitate the system under study. The variables and their derivatives are represented by means of voltages (which vary according to the equations governing the given system). The voltages can be monitored continuously at suitable points in the set-up to get the state of the system as a function of time.

Historically analog computers came into being years before digital computers did, and have played a major role in simulation of continuous dynamic systems. Although analog computers are giving way to digital computers at an increasing pace, they are still in occasional use. The following are some of the disadvantages of analog computers.

1. **Inadequate accuracy**: In general, the result from a digital simulation is more accurate than that from an analog simulation. The accuracy of analog simulation depends on the accuracy of the components being used, which can vary from .01% to 2%. When accuracy required of components is more than 0.1%, the cost of components increases rapidly. An accuracy

of about 1% for a simulation of system with modest complexity is considered good. This limited accuracy cannot meet the need in simulating systems like missiles and space vehicles.

2. **Scaling needed**: The magnitudes of dependant variables are represented in an analog computer by voltages. For a 10-volt analog computer the maximum range of voltages is from -10 to $+10$. All program variables must be scaled carefully so that none exceeds the maximum voltage. If a variable exceeds the maximum voltage, then the corresponding amplifier becomes saturated and results become inaccurate. This magnitude scaling is a tedious task in an analog simulation, because there are usually many variables and their maximum values are not known in advance.

In a digital computer with floating point arithmetic the magnitude scaling problem does not arise, because the quantities that can be represented on a digital computer have a very large range. For the IBM 360, for example, the range is from -10^{75} to $+10^{75}$. It has a precision of 7 decimal digits. Therefore, normally no magnitude scaling is needed.

In addition to magnitude scaling, analog computers also require time scaling. The maximum computing time for an analog computer is limited because of drifts affecting the accuracy of the results. The time scale factor, i.e., the ratio of computing-time to problem-time has to be determined. (If this ratio is 1, simulation is *real time*.) Since there is only one time scale factor, the task of time scaling is much easier. In digital computer simulation time scaling is generally not needed.

3. **Hardware set up necessary**: In analog simulation input constants and initial conditions are incorporated by setting up voltages and potentiometers. Then the various elements (amplifiers, multipliers, voltmeters, etc.) have to be connected on a patch board. Such setting up is not needed in a digital simulation.

A simulation program on a digital computer can be easily stored for reuse. Availability of various mathematical functions on a digital computer is another advantage. Since in a digital simulation no set up is required nor any calibration and accuracy test is needed, rapid switching from one simulation to another can be made. This results in a better machine utilization.

The two main advantages of analog simulation have been higher speed of solution (necessary for certain applications) and direct access to and immediate display of the computed results. Both these advantages are vanishing in the presence of superspeed digital computers available in a multiprogramming environment with on-line CRT displays.

We will not discuss analog simulation any further. The reader may refer to any of several excellent textbooks available on analog computers.

2-8. Remarks and references

Electronic analog computers became available soon after World War II

and they were found to be extremely valuable in analysis and design of numerous engineering systems. Analog computers were used for simulating various continuous dynamic systems which were too difficult to lend themselves to analytic studies. About ten years later, in the mid 1950's, when digital computers became commercially available, their advantages (such as greater accuracy, no need of scaling, greater flexibility, etc., as discussed in Sec. 2-7) over analog computers in simulating complex continuous systems were recognized, and they began to be used for this purpose.

A general purpose digital computer was required to have certain minimum hardware facilities if it was to be used successfully for simulating large continuous systems; such as hardware floating point arithmetic, long word length to reduce round-off errors, and a reasonably large random-access memory for handling a large amount of intermediate data. As the hardware (and software) of the digital computers became increasingly better and cheaper during the last twenty years, the digital computer began to be used more and more for simulating continuous dynamic systems.

The most comprehensive and an early textbook dealing with the simulation of continuous systems on a digital computer is

CHU, Y., *Digital Simulation of Continuous Systems*, McGraw-Hill, New York, 1969.

Some more recent books on this topic are

ORD-SMITH, R. J. and J. STEPHENSON, *Computer Simulation of Continuous Systems*, Cambridge University Press, Cambridge, U.K., 1975.

STEPHENSON, R. E., *Computer Simulation for Engineers*, Harcourt, Brace, Jovanovich, New York, 1971.

SHAH, M. J., *Engineering Simulation Using Small Scientific Computers*, Prentice-Hall, Englewood Cliffs, N.J., 1976.

A journal devoted entirely to computer simulation was started in 1963 and is called SIMULATION. A good collection of articles, dealing exclusively with simulation of continuous systems, from this journal can be found in

McLEOD, J. (ed.)., *SIMULATION : The dynamic Modeling of Ideas and Systems with Computers*, McGraw-Hill, New York, 1968.

As pointed out in the beginning of this chapter, continuous system simulation is encountered mostly in engineering and physical sciences. For example, simulation has played a crucial role in the development of modern aerospace industry and chemical-process industries. A few applications in these areas have already been illustrated, namely, a chemical reactor, an aircraft pursuit system, an electrical network, a feedback control system, and a water reservoir system. Some additional applications are pointed out in the form of exercises. These are trajectory calculations, the multibody problem, vibration and shock absorbers, and econometric models. This list of continuous dynamic systems where simulation has been applied pro-

fitably is indeed very large. The examples we chose in this chapter were simplified to keep the problems manageable. A real problem is more involved computationally (but not conceptually).

The examples of simulation applied to various engineering and other disciplines can best be found in journals, technical reports, and textbooks in respective fields. For instance, the following provides a good reference for simulation of water resource systems (a simple example of which was given in Sec. 2-6).

HUFSCHMIDT, M. and M. B. FIERING, *Simulation Techniques for the Design of Water Resources Systems*, Harvard University Press, Cambridge, Mass., 1966.

Likewise an excellent treatment on digital simulation of chemical engineering plants can be found in Chapter 5 of

LUYBEN, W. L., *Process Modeling, Simulation and Control for Chemical Engineers*, McGraw-Hill, New York, 1973.

An integrator is the heart of an analog simulation. Likewise, the selection of an appropriate integration subroutine (from amongst several available on your computer system or at least in the literature) is of central importance to the program for simulation of a continuous system. There are a number of excellent textbooks on numerical analysis that discuss various integration formulas and their merits from the viewpoint of speed, accuracy, etc. A few of these are:

ACTON, F. S., *Numerical Methods that Work*, Harper and Row, New York, 1970.

CONTE, S. D. and C. De BOOR, *Elementary Numerical Analysis : An Algorithmic Approach*, (2nd Ed.), McGraw-Hill, New York, 1968.

DAHLQUIST G. and A. BJÖRCK, *Numerical Methods*, Prentice-Hall, Englewood Cliffs, N.J., 1974.

RALSTON, A., *A First Course in Numerical Analysis*, McGraw-Hill, New York, 1965.

RAJARAMAN, V., *Computer Oriented Numerical Methods*, Prentice-Hall of India, New Delhi, 1971.

It is the technology of electronic devices which has tilted the balance overwhelmingly in favour of digital simulation for continuous systems, although analog simulation appears more natural. In view of the advancing technology, particularly of microprocessors, the field is far from having been stabilized. An interesting article on the effect of electronic technology on continuous system simulation is

AUS, H. M. and G. A. KORN, *The Future of Continuous-System Simulation*, Proc. AFIPS/FJCSS, 1969.

2-9. Exercises

I. TRAJECTORY SIMULATION

Simulation has been used in computing trajectories of various types of rockets, bullets, space vehicles, etc. Because of the drag (air-resistance) the

equations of motion get sufficiently complicated for a closed form mathematical solution. In the case of a rocket the mass itself does not remain constant but reduces as the fuel is used up.

2-1. Given a cannon that fires spherical balls of mass m, you are asked to produce a range table, i.e., a table that gives ranges for various values of muzzle velocity and gun elevation (firing angle). You are also asked to investigate the sensitivity of range to small changes in muzzle velocity and gun elevation. For simplicity assume that the drag is proportional to the square of the instantaneous velocity of the cannon ball, and it is purely along the direction of flight. Making use of the following four equations, write a computer program to produce a range table:

$$m\frac{dy}{dt} + mg\sin\theta + cv^2 = 0$$

$$mv\frac{d\theta}{dt} + mg\cos\theta = 0$$

$$\frac{dx}{dt} = v\cos\theta$$

$$\frac{dy}{dt} = v\cos\theta$$

Constant c is the drag coefficient for the cannon ball. Variables v and θ are the instantaneous velocity and angle of elevation of the cannon ball, and x, y are the coordinates of its instantaneous position. For various values of the starting conditions, i,e., muzzle velocity v_0, elevation θ_0, and $x_0 = y_0 = 0$, the program should yield the range.

2-2. A missile is fired vertically up. It has a rocket motor that produces a certain constant upward thrust F for as long as the fuel-burning motor is on. The motor burns the fuel at a constant rate b. Assume that the missile does not go so high as to vary the gravity constant g nor does the drag coefficient vary during the flight. Set up the equation of the vertical motion. Write a program which gives the vertical distance y at any time t.

2-3. Modify the simulation program in Exercise 2.2 in order to make it valid for higher altitudes. That is, take into account the reduction in the air resistance and change in the value of the gravitational constant g as the missile rises higher.

2-4. Suppose a 50-kg. heat-seeking missile is fired at a jet bomber flying at a constant speed of 1,200 Km/hour. The missile has a rocket motor that produces a thrust of 5000 Kg. for a period of 4 minutes. Simulate the missile system to test its effectiveness and to estimate how close the target should be before a missile is fired. Assume the drag coefficient $c = .04$ Kg/Km^2 and the weight of liquid fuel is 20 Kg. (which is burned in 4 minutes at a constant rate).

2-5. Assuming that the drag force is proportional to the square of the

speed and proportional to the cross-sectional area of a moving body, how will you simulate the experiment of the Tower of Pisa by dropping two lead balls of diameter 1 cm and 20 cm from a 400 meter tower?

Notice that in all trajectory calculations any refinements, such as, (i) making the air-resistance more complicated than just proportional to v^2, and also including a non-zero component perpendicular to the velocity vector, and (ii) taking into account the effects of the earth's rotation, can be easily accommodated. These refinements require only additional computation and not any increased analytical ability. This is one of the fundamental advantages of simulation.

II. THE MULTIBODY PROBLEM

When there are two bodies in space, it is easy to compute their paths analytically using Newton's laws of motion. But if there are three (or more) bodies influencing each other (such as the sun-earth-moon system), it has not been possible to obtain their paths of motion in a closed analytic form. It is, in fact, one of the oldest unsolved problems in physics. A great deal of effort has been spent in solving this problem. We can, however, simulate any N-body system for any required period given an initial set of coordinates.

2-6. Assuming that there are N heavenly bodies in a plane, write a FORTRAN program to compute their paths for a specified period T. (Divide T into N small periods. Compute the position at each of these instants using the fact that there are only two forces acting on a body—sum of the $N-1$ gravitational forces due to the other $N-1$ bodies and the forces due to acceleration.)

2-7. After gathering the relevant data for the sun-earth-moon system, simulate the motion of this 3-body system for the period of a month.

III. SUSPENSION SYSTEM

In trajectory problems the external forces (drag and gravity) either remain constant or vary slowly. Consequently the trajectory is a smooth curve. In contrast, the suspension system of an automobile is subjected to abruptly changing forces due to potholes in the road. The purpose of a suspension system is to damp out the effect of vibrations, caused by sudden changes in the road surface.

2-8. Each wheel with a mass m in an automobile has an independent suspension system, consisting of a spring and a shock absorber, as shown in Fig. 2-6. The resistance force of the spring is proportional to the displacement (i.e., compression) and the resistive force of the shock absorber is mostly proportional to the rate of change of displacement. The exact equation of motion for the system is

$$m\ddot{y} = k_1(\dot{y} - k_2 y) + k_3 y = k_3 \cdot F(t) \qquad \ldots(2\text{-}11)$$

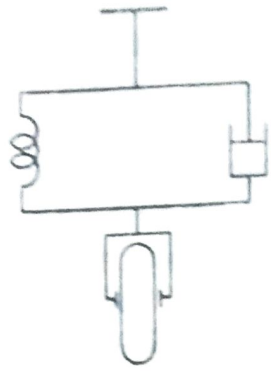


Fig. 2-6.

We wish to find the transient behaviour of this system with the following values of the constants:

$$\text{mass } m = 1.0, \quad k_1 = 5.0, \quad k_2 = 0.05$$

spring stiffness $k_3 = 700$, to two types of forces

(i) step function $F(t) = 0, \quad (t \leqslant 0)$
$a, \quad (t > 0)$

(ii) ramp function $F(t) = 0, \quad (t \leqslant 0)$
$a\,t, \quad (t > 0)$

Write a program to simulate the system. Observe, in particular, the time required for the wheel to return to normal after it runs over a square pothole and the amount of overshoot.

2-9. Simulate the suspension system described in Exercise 2-8, with an added requirement that there is a physical stopper that prevents the maximum displacement from exceeding a value of 2 units.

2-10. Write a computer program to analyze the behaviour of the suspension system in Exercise 2-8 when it runs over a series of equispaced potholes on the road.

2-11. How will you decide on the constants of the spring k_3, and of the shock absorber k_1 (assume k_2 to be one-hundredth of k_1), given m, $F(t)$, the maximum tolerable displacement, and the maximum time allowed to return to normal?

IV. ECONOMETRIC MODELS

Simulation has been used widely to study the dynamic behaviour of large economic systems—of industries, of farms, of nations. The purpose is to determine economic variables of interest as a function of time in response to different actions taken. The following is an extremely elementary model of a nation's economy.

2-12. The gross national product $GNP(t)$ during time t can be expressed as a sum

$$GNP(t) = GS(t) + IN(t) + C(t) \quad \ldots(2\text{-}12)$$

Where $GS(t)$ is the government spending during time t, $IN(t)$ is the investment during t, and $C(t)$ is the consumption. Two of these three quantities can themselves be expressed as

$$IN(t) = m + c[GNP(t-1) - GNP(t-2)] \quad \ldots(2\text{-}13)$$

$$C(t) = a + b.GNP(t-1) + d.GNP(t-2) \quad \ldots(2\text{-}14)$$

where a, b, c, m, d are constants which have been determined for this system. At time t the previous year's GNP is known. Simulate the system for predicting the current year's GNP, given that the government spending has been announced.

3

Discrete System Simulation

In the last chapter our objects of simulation were continuous systems—those systems in which the state changed smoothly with time. In this chapter (and in all subsequent chapters) we will deal with discrete systems—systems in which the changes are discontinuous. Each change in the state of the system is called an *event*. For example, arrival or departure of a customer in a queue is an event. Likewise, sale of an item from the stock or arrival of an order to replenish the stock is an event in an inventory system. Arrival of a car at an intersection is an event if we are simulating street traffic. Therefore, the simulation of a discrete system is often referred to as *discrete-event simulation*.

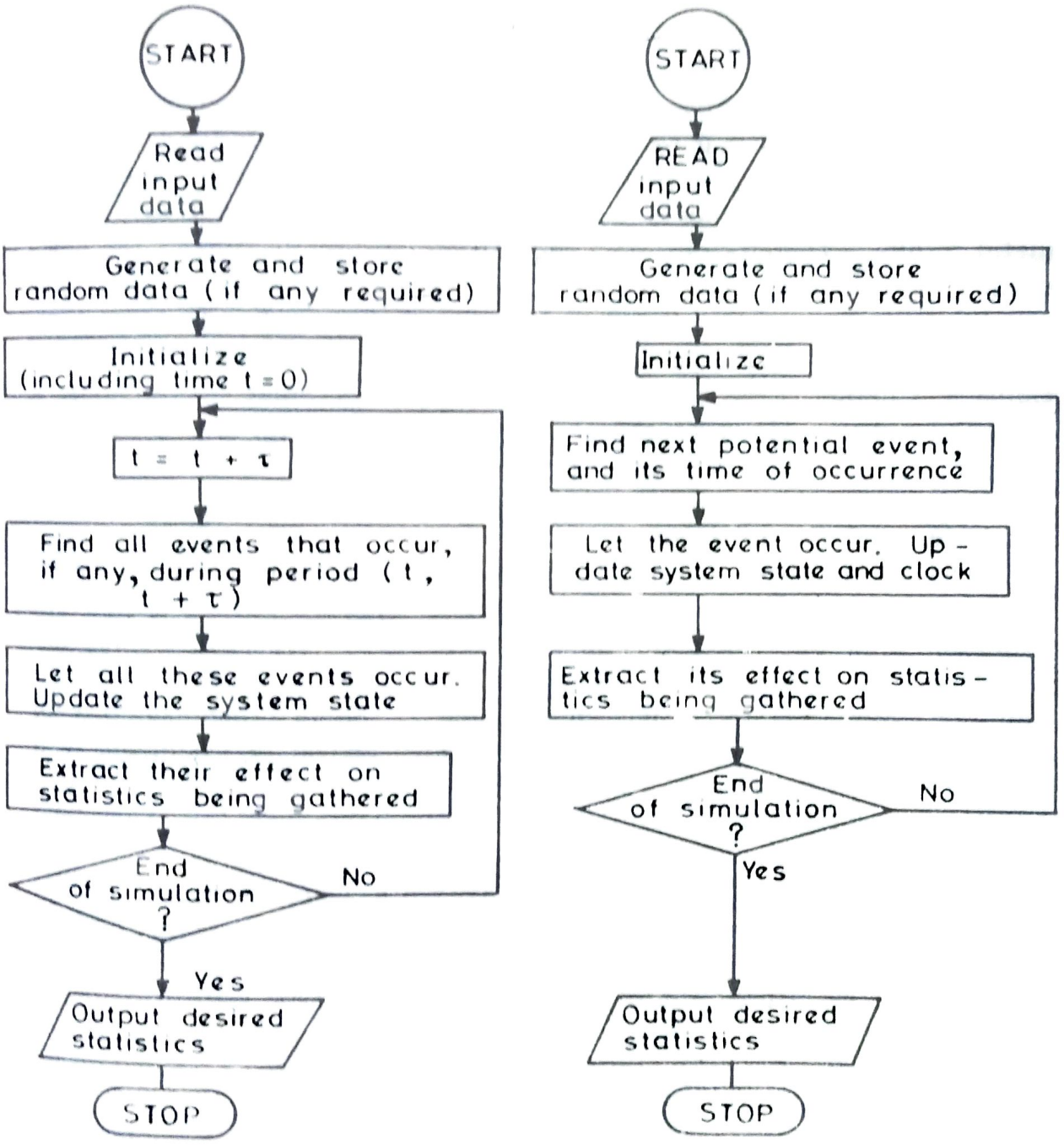
Discrete-event simulation is commonly used by operations research workers to study large, complex systems which do not lend themselves to a conventional analytic approach. The very simple inventory problem simulated in Sec. 1-3 is an example. Some other examples are the study of sea and airports, steel melting shops, telephone exchanges, production line, stock of goods, scheduling of projects, to name a few. Discrete system simulation is more diverse and has less of a theory than continuous system simulation. There are no overall sets of equations to be solved in discrete-event simulation.

3-1. Fixed time-step vs. event-to-event model

In simulating any dynamic system—continuous or discrete—there must be a mechanism for the flow of time. For we must advance time, keep track of the total elapsed time, determine the state of the system at the new point in time, and terminate the simulation when the total elapsed time equals or exceeds the simulation period. For continuous systems, in Chapter 2, we advanced time in small increments of Δt for as long as was needed. In simulation of discrete systems, there are two fundamentally different models for moving a system through time : the *fixed time-step* model and the *event-to-event* (or *next event*) model. In a fixed time-step model a "timer" or "clock" is simulated by the computer. This clock is up-dated by a fixed time interval τ, and the system is examined to see if any event has taken place during this time interval (minutes, hours, days, whatever). All events that take place during this period are treated as if they occurred simultaneously at the tail end of this interval. In a next-event simulation model the

computer advances time to the occurrence of the next event. It shifts from event to event. The system state does not change in between. Only those points in time are kept track of when something of interest happens to the system.

Flowcharts for these two methods of simulating a discrete system in their most general forms are shown in Fig. 3-1.



(a) Fixed time-step simulation

(b) Next-event simulation

Fig. 3-1: Flowcharts for discrete system simulation.

To illustrate the difference between the two models, let us assume that we are simulating the dynamics of the population in a fish bowl, starting with, say, 10 fish. If we used the fixed time-step model with, say, $\tau = 1$ day, then we would scan the fish-bowl (figuratively speaking) once every 24 hours, and any births and deaths that take place are presumed to be during the last moment of this period. On the other hand, if we use a next-event model then we will first find out when the next-event (birth or death) is to take place and then advance the clock exactly to that time.

In general, the next-event model is preferred, (except when we may be forced to use the fixed time-step model) because we do not waste any computer time in scanning those points in time when nothing takes place. This waste is bound to occur if we pick a reasonably small value for τ. On the other hand, if τ is so large that one or more events must take place during each interval then our model becomes unrealistic and may not yield meaningful results. Therefore in most simulations of discrete systems the next-event model is used. The only drawback of the next-event model is that usually its implementation (programming for it) turns out to be more complicated than the fixed time-step model. To construct these two applications, in Chapter 4 we will use these two different models to simulate a queueing system. However, except for that case, in the rest of this book we will employ only next-event models of discrete systems.

3-2. On simulating randomness

There are numerous natural as well as man-made systems where chance plays some part. These are called *stochastic* systems. There is inherent randomness or unpredictability in their behaviour. We have already encountered two such examples, namely, the inventory system in Chapter 1 and the water reservoir system in Chapter 2. Some other examples of randomness that are frequently simulated are: arrival of customers in a store, arrival of vehicles at a traffic light, request for telephone lines at a telephone exchange, births and deaths in a population, collision of particles in a nuclear reactor, arrival of an elevator on a given floor, etc.

Discrete dynamic systems could be classified as deterministic or stochastic. The former are less demanding computationally than the latter and are frequently solved analytically. Hence simulation in the study of discrete dynamic systems is used almost exclusively for stochastic systems—systems in which at least one of the variables is given by a probability function. Complex discrete, dynamic, stochastic systems often defy an analytic solution and are therefore studied through simulation.

To simulate such random variables, we require a source of randomness. In simulation experiments, this is achieved through a source of *uniformly distributed random numbers*. These numbers are samples from a uniformly distributed random variable between some specified interval, and they have equal probability of occurrence in the same manner as all six faces of an

unbiased die have equal chance of occurrence. A random number generator and its appropriate use form the heart of any simulation experiment involving a stochastic system. Before discussing methods of generating random numbers, let us see how they are used.

Use of random numbers: an example: Suppose a situation has three possible outcomes. For example, the price of a certain commodity can go up, down, or stay at the same level at the end of a particular day. Suppose the probabilities of these three outcomes are 0.3, 0.5, and 0.2, respectively. Then we need a random selector which reproduces the outcome, in the long run, with relative frequencies of 0.3, 0.5, and 0.2. They can be obtained with a device generating 10 uniform random numbers 0, 1, 2, ..., 9 with equal probability, and associating the sets of numbers {0, 1, 2} with the first outcome (i.e., price going up); {3, 4, 5, 6, 7} with the second outcome (i.e., price coming down), and {8, 9} with the third outcome (i.e., price remaining unchanged). Thus, we have a device that simulates the stochastic phenomenon of the price going up, down or remaining unchanged with the specified probabilities (of 0.3, 0.5, and 0.2, respectively). Using this device we can now conduct experiments. Thus, in general, the procedure for choosing a single random element from the range of all possible values consists of two steps. First a number is obtained (say 4) from a random number source, then it is transformed to correspond to an appropriate outcome of the experiment (number 4 corresponds to the price going up, in this case).

3-3. Generation of random numbers

Random numbers could be obtained from a sack of numbered beads as in bingo; or from rotations of a roulette wheel; or from any other randomizing device. However, such physical generators of random numbers are not suitable for simulation experiments on computers for two reasons: (i) The generation and feeding into the computer of thousands of such numbers is excessively laborious and time consuming, and more importantly, (ii) a sequence of numbers generated cannot be reproduced at a later time or by another person for repeating a simulation run. Such repetitive runs are required for debugging computer programs as well as for studying the effect of changes in the model.

We would, therefore, like to have some fast and deterministic method of generating a sequence of numbers which have the property of being random. Such deterministically generated numbers which appear to be random are called *pseudorandom* numbers.

Numerous arithmetic methods of generating pseudo-random numbers have been suggested, studied, and used on computers in the past thirty years or so. These methods are usually based on some recurrence relation. Each new number is generated from the previous one by applying some simple 'scrambling' operation. A fast, and the most commonly used method

(or generator) is the so-called *multiplicative congruential generator* (sometimes also called the *power-residue generator*). It consists of computing

$$x_{i+1} = x_i . a \text{ (modulo } m\text{)}, \qquad \ldots(3\text{-}1)$$

where x_i is the ith pseudo-random number, x_{i+1} is the next pseudo-random number, a is a constant multiplier, and, modulo m means that the number $(x_i . a)$ is divided by m repeatedly till the remainder is less than m. The remainder is then set equal to the next number x_{i+1}. The process is started with an initial value x_0, called the *seed*. To illustrate, suppose we start with the seed $x_0 = 3$ and parameters $a = 7$ and $m = 15$. Then the successive random numbers generated are:

$$x_1 = 3 \times 7 = 21 \equiv 6$$

$$x_2 = 6 \times 7 = 42 \equiv 12$$

$$x_3 = 12 \times 7 = 84 \equiv 9$$

$$x_4 = 9 \times 7 = 63 \equiv 3$$

and so on, all between 0 and 14.

Clearly, because of the modulus arithmetic each of the numbers generated by Eq. (3-1) must be one of the integers 0, 1, 2, ..., $m-1$. Thus eventually the series will repeat itself. The period of this generator will never be greater than m (but it can be much shorter than m, if a and x_0 are not chosen properly). With a proper choice of m, a, and x_0 it is possible to obtain a generator with period almost equal to m.

A great deal of analytic and experimental study has been conducted on sequences generated by the congruential generator [Eq. (3-1)] with various combinations of m, a, and the seed x_0. [It can be shown that if x_0 is odd, and $m = 2^r$ $(r > 2)$ and $a = k.8 \pm 3$ (k being any non-negative integer), then the period of the sequence generated is maximum and is equal to 2^{r-2}.] The following choice of the three parameters leads to a good random sequence.

(1) Choose m be one more than the largest integer that can be held in one word of the computer being used. For example, in the IBM 360/370 system a word is 32 bits long, of which the most significant bit is reserved for the sign. Thus the largest integer one computer word can hold is $2^{31} - 1 = 2{,}147{,}483{,}647$, and therefore, m should be 2,147,483,648. In case we are using an IBM 1130/1800 system, (which has 16-bit words), we would choose $m = (2^{15} - 1) + 1$ $= 32{,}768$.

This choice of m makes the division of the product $x_i . a$ in Eq. (3-1) unnecessary, for the modulo operation. A machine word cannot hold an integer larger than $(m - 1)$. Therefore, as soon as the product exceeds $(m - 1)$ an overflow would automatically occur,

and we would be left with only the remainder. (For most FORTRAN environments no action is taken when an integer overflow occurs.)

(2) The seed x_0 must be relatively prime to m. Since m is a power of 2, any odd positive integer for x_0 would do. (It is not difficult to see that if x_0 is even, every x_i would also be even.)

(3) Lastly, the *constant multiplier* a must be properly selected. Clearly, a must also be relatively prime to m, i.e., a must also be an odd number. It has been found that the best choice for a is that it satisfies the relation

$$a = k.8 \pm 3 \qquad \ldots(3\text{-}2)$$

and is close to the integer $2^{b/2}$, where b is the number of bits in the computer word. Thus for an IBM 360/370 system, a 32-bit binary machine, we might pick

$$a = 2^{16} + 3 = 65{,}539.$$

The subprogram to generate random numbers should be tailored to the machine being used. The following FORTRAN Function (actually an assembly-laguage equivalent of it) has been used throughout this book for generating uniformly distributed pseudo-random numbers in the range (0, 1), on the IBM 7044 computer—a 36-bit binary machine.

```
FUNCTION RNDY1 (DUM)
INTEGER A, X
DATA A, X/189277, 11750920161/
X = A*X
AX = X
RNDY1 = AX/34359738368.0
RETURN
END
```

Notice that the seed x_0 is supplied by the program itself. The user must, however, put some dummy argument (DUM) just to comply with the FORTRAN requirement for using and defining a function. The third statement in this program assigns a value 189277 (which is equal to $23{,}660 \times 8 - 3$ and lies between 2^{17} and 2^{18}) to the multiplier A. The same statement sets the value of the seed as 11,750,920,161, a large odd number. The fifth statement converts an INTEGER variable into a REAL one. The sixth statement simply puts a decimal point behind the most significant bit, by dividing the generated number with $2^{35} = 34{,}359{,}738{,}368$. Thus all generated integers (which lie between 0 and $2^{35} - 1$) are put into the range (0, 1). The sequence of the first 250 numbers (each 8-digit long) produced by this program on the IBM 7044 is shown in Table 3-1 (page 46).

It should be noted that the least significant digits of the pseudo-random number generated by the multiplicative congruential method are not random. Therefore, low-order digits should not be used as random numbers.

0.12605673	0.63945656	0.41933851	0.13560773	0.42469418
0.84082449	0.74020635	0.03859064	0.32124565	0.41372764
0.12712769	0.34750068	0.88700985	0.56569849	0.71384876
0.15439663	0.73127418	0.38438318	0.89630192	0.34040167
0.20715574	0.81669794	0.13634698	0.34709622	0.33082132
0.86722385	0.52967285	0.88980526	0.67189881	0.99240244
0.95817188	0.90184278	0.09789246	0.79209450	0.27209922
0.12377563	0.88041454	0.22506470	0.57232984	0.87676947
0.29604600	0.69995449	0.28802593	0.68407930	0.47963840
0.51830582	0.37002663	0.52974694	0.91286987	0.27119800
0.54527316	0.66856805	0.55586741	0.91606059	0.20258154
0.02639124	0.25470810	0.38522433	0.10507539	0.35481164
0.68261174	0.70277291	0.74959149	0.42932156	0.69763737
0.71125819	0.81837045	0.70498065	0.62469626	0.63484096
0.79334503	0.96866811	0.59469365	0.83042011	0.42833864
0.65197936	0.69861210	0.20331260	0.39948643	0.59382989
0.33934862	0.88904662	0.07919170	0.16682845	0.78911820
0.92783065	0.00392972	0.80648247	0.58523636	0.78326067
0.23121197	0.10729789	0.02209292	0.68182824	0.40602278
0.77483593	0.62212986	0.87479187	0.98204138	0.84778971
0.09363551	0.04897409	0.66796139	0.72914251	0.90816759
0.23804685	0.79422058	0.69047086	0.25541778	0.71086777
0.92045329	0.63845453	0.75833005	0.43842954	0.62897794
0.05765601	0.95628558	0.86598516	0.07463225	0.16863216
0.18916636	0.84191151	0.48626862	0.46544028	0.14011042
0.67987362	0.44022211	0.92160124	0.91977078	0.45585227
0.35106367	0.27858218	0.19989099	0.76659736	0.24995090
0.95586113	0.52900933	0.30106957	0.54588907	0.24663635
0.58810474	0.70143276	0.09021481	0.58824246	0.76899630
0.31515973	0.48961480	0.82133635	0.08330946	0.56481367
0.23799234	0.47691497	0.03482086	0.78706795	0.86249388
0.25522759	0.71299479	0.51752671	0.90400832	0.98564532
0.99052353	0.32290863	0.17745147	0.48154996	0.33268388
0.40715661	0.38214801	0.82930494	0.35132289	0.34303564
0.75762388	0.77537080	0.86128480	0.40578150	0.10512107
0.00062777	0.82209201	0.11036270	0.12014770	0.19596767
0.17269229	0.67891822	0.60637572	0.97821812	0.19208509
0.28993362	0.76619879	0.81046108	0.64363914	0.08571769
0.38826279	0.21580850	0.58591397	0.03974269	0.37728143
0.69869744	0.35779788	0.91041011	0.69644971	0.91276737
0.87077474	0.63280492	0.41776729	0.73938178	0.96613230
0.62500039	0.20058468	0.06585702	0.21992429	0.61047640
0.14368453	0.17746056	0.20349902	0.68321339	0.58121061
0.80224132	0.83240682	0.46702461	0.01775690	0.97275453
0.06149900	0.34568924	0.02292932	0.99374360	0.80903049
0.86544868	0.53069273	0.92984154	0.61757918	0.53642869
0.61381547	0.15235440	0.18459393	0.38454978	0.42872409
0.60918928	0.52038404	0.73048838	0.65108073	0.60698931
0.11636278	0.79725136	0.34739572	0.01953820	0.13135707
0.87292619	0.85298774	0.96149679	0.22879631	0.88016954

Table 3-1: First 250 pseudo-random numbers generated by RNDY1.

Random number generators are used so frequently that almost every computer manufacturer provides one or more subprograms tailored to this particular system. For example, RANDU is an IBM supplied subroutine that forms part of their Scientific Subroutine Package. FPMCRV is supplied by ICL as a PLAN scientific subroutine. On the GE 225 computer RAND is the system pseudo-random number generator. And so on. Even some pocket calculators have a built-in function that generates random numbers.

For the purposes of illustration, for sampling, or for hand simulation many textbooks on statistics or simulation provide tables of random numbers.

The tables of random numbers encountered frequently in textbooks are of two-digit integers in the range of 00 to 99. Tables of five-digit random numbers would range from 00000 to 99999. Actually the number of digits in a random number table is not too important. If there are too many digits the numbers can be truncated. If there are too few, the successive numbers can be run together to give the required number of digits.

Tests for randomness. It is obvious that the sequence of numbers generated by the multiplicative congruence method (by any arithmetic method for that matter) is not truly 'random' in the sense that the entire sequence is predetermined and predictable. If the process is started with the same x_0, exactly the same sequence would result. Since m is finite, one of the numbers generated will eventually be exactly the same as a previously generated number. Once this happens the subsequence of numbers will also be repeated. For these reasons the name pseudo-random is used for these generators.

This lack of true unpredictability is not a serious drawback for computer simulation provided the generated sequence possesses some of the important 'random like' characteristics. How random is a given sequence of random numbers is a difficult question to answer and one that has been discussed at great lengths. A bewildering variety of tests have been proposed in the literature for testing the randomness of a sequence. For most simulation purposes a random number sequence is adequate if (i) its uniformity is assured, and (ii) the successive numbers in the sequence are independent. For these characteristics the following two tests would suffice.

(i) **Frequency test:** The *frequency test* or *uniformity test* counts how often numbers in a given range occur in the sequence to ensure that the numbers are uniformly distributed. There should be no favoured numbers, that is, no number should occur more frequently than what is expected from chance variation. For example, if we have a sequence of 5,000 3-digit numbers (from 000 to 999), we should expect that in the range 00 to 99 there are about 500 numbers; similarly in the range 100 to 199 there should be about 500 numbers, and so on. Clearly, we do not expect that there be exactly 500 numbers in each of the 10 ranges (namely, 00 to 99, 100 to 199, ..., 900 to 999). In fact, if we found that there are exactly 500 numbers in each

range, we should suspect some nonrandomness. On the other hand, deviation from 500 should not be too much, otherwise we would suspect some nonuniformity. Then how much deviation should we allow from the expected value of 500 and still accept the sequence as uniformly distributed? A neat quantitative answer to this question is provided by the *chi-square* goodness of fit test (or simply *chi-square test*).

Chi-square test: The chi-square test is a very important and useful statistical test for determining how well certain observed data fit the theoretically expected data. The testing is performed by first dividing the observed data (in our case, the generated random numbers) into k non-overlapping classes; k must be 3 or more (in our example, $k = 10$). Then we count O_i, the number of times the observed data falls in each class i, for i, $i = 1, 2, \ldots, k$. Next, we determine the expected number of occurrences E_i in each class i. Then to measure how far the observed frequency deviates from the expected we compute the chi-square statistic, defined by

$$\text{chi}^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i} \qquad \ldots(3\text{-}3)$$

Intuitively we can see that this sum measures the discrepancy between the observed frequencies and their expected values. The more the actual occurrences depart from their expected values the larger the values of chi^2. Conversely, the closer the O_i's are to E_i's the smaller the values of chi^2, becoming zero when the two sets of frequencies are identically equal.

In our example, $E_i = 500$, for $i = 1, 2, \ldots, 10$. Suppose by actual frequency count of numbers generated we found the occurrences as shown in the third column of the table below. Then the chi^2 statistic can be calculated as follows:

i	Range	O_i No. of observed occurrences	No. of expected occurrences	$(O_i - E_i)^2$	$\frac{(O_i - E_i)^2}{E_i}$
1	000–099	468	500	1024	2.048
2	100–199	519	500	361	.722
3	200–299	480	500	400	.800
4	300–399	495	500	25	.050
5	400–499	508	500	64	.128
6	500–599	426	500	5476	10.952
7	600–699	497	500	9	.018
8	700–799	515	500	225	.450
9	800–899	463	500	1369	2.738
10	900–999	529	500	841	1.682

Total = chi^2 = 19.588.

The question now arises : how large or small a computed value of chi² can we accept before concluding that the discrepancy between the observed frequency and those expected is too great (or too small) to have arisen by mere chance? For this purpose there are statistical tables which give those critical values of chi² which can be exceeded by chance with various probabilities, such as .001, .01, .05, .1, etc. These are called chi²-tables. To read a chi²-table we must have another parameter, called the *degree of freedom*. The degree of freedom is defined by

$$\nu = k - 1,$$

where k is the number of sets into which data is divided. For our example, $\nu = 9$.

Now we read the chi² tables and find the following row of entries for $\nu = 9$.

.995	.99	.95	.90	.75	.50	.25	.10	.05	.01	.005
1.73	2.09	3.33	4.17	5.90	8.34	11.4	14.7	16.9	21.7	23.6

This means that there is a 99.5% probability of chi² exceeding 1.73; a 99% probability of chi² exceeding 2.09, ..., and a .5% probability of chi² exceeding 23.6. Thus the probability of the chi² statistic (for $\nu = 9$) falling below 1.73 and above 23. 6% is only 1%. That is, 990 out of 1,000 sequences from a perfectly uniform random number generator would have given

$$1.73 \leqslant \text{chi}^2 \leqslant 23.6.$$

Likewise, if we take 1 as the cutoff point, we would reject all sequences with chi² below 2.09 and above 21.7. In particular, the random-number sequence whose chi² we have just calculated and found to be 19.588, is barely acceptable.

The frequency test can be performed on the occurrence of each digit also. The test can be repeated on several sets of N (say 10,000) numbers. We can then obtain a value of the chi-square for each set, and see if these values themselves pass a chi-square test.

(ii) **Independence test:** A sequence may be uniformly distributed and yet be far from being random because the adjacent terms may be related. One of the most effective method of determining correlation between adjacent terms is the poker test, which can be explained as follows:

Suppose we are given five independent decimal digits. We could treat these digits as a hand of poker in a card game and classify accordingly, i.e., five of a kind (*a a a a a*); four of a kind (*a a a a b*); full house (*a a a b b*); three of a kind (*a a a b c*); two pairs (*a a b b c*); one pair (*a a b c d*); and bust (*a b c d e*). It is fairly easy to compute the probabilities associated with these seven hands. They are approximately 0.0001, 0.0045, 0.0090, 0.0720,

0.1080, 0.5040, 0.3024, respectively. If we generated $5n$ random digits we can form n random poker hands and then compare the observed frequencies of these seven types of poker hands with the expected distribution. To measure the amount of deviation between the expected and actual distribution we once again use the chi-square test, with degree of freedom $\nu = 6$.

Although there are a very large number of tests for randomness, for most simulation purposes, a sequence of pseudorandom numbers that passes both the frequency test and poker test will be adequate. The multiplicative congruential generator, with the values of a, m, and x_0 suggested has been found to pass both these tests.

3-4. Generation of non-uniformly distributed random numbers

In the last section we learned how to generate numbers which had randomness properties and were uniformly distributed between 0 and 1. These numbers were referred to as pseudorandom numbers. From these numbers we can readily obtain uniformly distributed random numbers y_i's between any arbitrary interval (A, B), using the relation

$$y_i = A + (B - A).u_i \qquad \ldots(3\text{-}4)$$

where u_i's are the previously generated uniform radom numbers in the interval (0, 1).

Many simulation experiments require random samples from nonuniform distributions, such as, normal, exponential, beta, gamma, chi-square, log-normal, Cauchy, and Weibull distributions. It can be shown that samples from any arbitrary distribution can be generated using the uniformly distributed random numbers in the interval (0, 1) $r_1, r_2, \ldots$. In fact, at present there are no fast practical methods of generating samples from arbitrary distributions on the computer, except via the uniform random numbers. There are many special techniques and tricks for converting uniform random numbers into samples from various other distributions. We will briefly discuss two most commonly used general methods:

The inverse transformation method: Suppose we are given a probability distribution function $F(z)$, which is continuous, and we wish to generate n random samples $z_1, z_2, \ldots, z_n$ from this distribution. By definition, every probability distribution function increases monotonically from 0 to 1, and the probability that a random sample z lies in the interval (z_1, z_2) is equal to $F(z_2) - F(z_1)$ for all pairs $z_1 \leqslant z_2$. Moreover, since $F(z)$ is continuous it takes all values between 0 and 1; and therefore for any number u, $0 \leqslant u < 1$, there exists a unique z_u such that $F(z_u) = u$. Symbolically this value of z may be represented by $F^{-1}(u)$, the inverse function. (See Fig. 3-2.)

Let us generate n uniformly distributed random numbers $u_1, u_2, \ldots, u_n$ in the interval (0, 1), using the method given in Sec. 3-3. Next we take their inverse transforms $F^{-1}(u_1), F^{-1}(u_2), \ldots, F^{-1}(u_n)$. That these are the

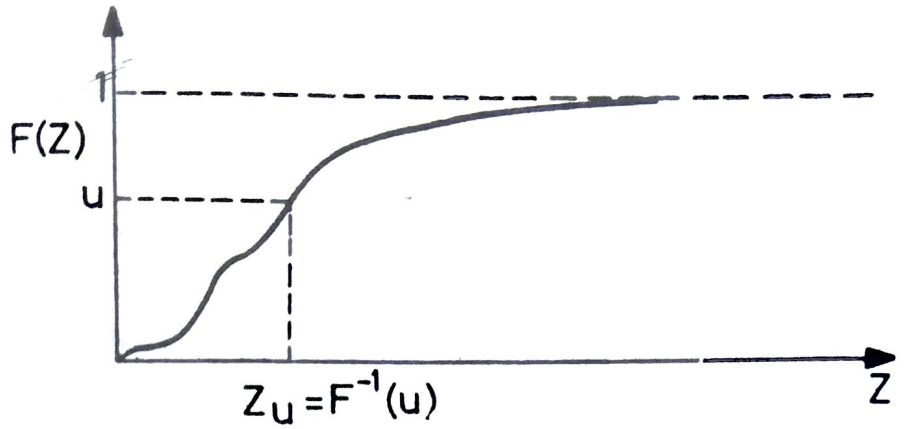


Fig. 3-2: A continuous probability distribution function $F(z)$.

desired numbers $z_i = F^{-1}(u_i)$, for $i = 1, 2, \ldots, n$ can be seen as follows:

Consider any two numbers a and b in the range (0, 1) such that $a < b$. By definition, the probability that a uniform random number in the range (0, 1) lies between a and b is $(b - a)$ for all $0 \leqslant a \leqslant b \leqslant 1$. Let $F^{-1}(a) = z_a$ and $F^{-1}(b) = z_b$. The inverse transform method would hold if the actual probability of the random variable z lying between z_a and z_b equals the generated probability $(b - a)$. That is, the actual probability of z lying between z_a and z_b should be $F(z_b) - F(z_a)$, which indeed is the case because $(b - a) = F(z_b) - F(z_a)$.

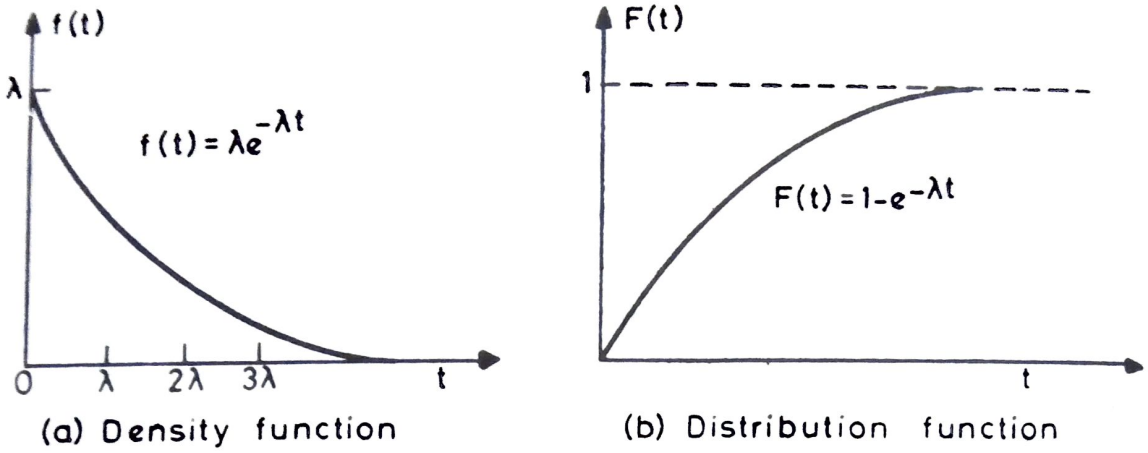
Thus, to generate n samples from any continuous probability distribution function $F(z)$, all we need to do is to generate n uniform random numbers $u_1, u_2, \ldots, u_n$ in the interval (0, 1) and apply inverse transform $F^{-1}(u_i)$ to each. To illustrate how the inverse transform method works, let us generate a random sample from an exponential distribution function.

Exponential distribution function: There are many phenomena that are governed by the exponential probability function. For example, the probability that the waiting time w of a telephone call exceeds time t is given by

$$Pr[w > t] = \lambda e^{-\lambda t} \qquad \ldots(3\text{-}5)$$

where λ is a positive constant depending on characteristics of the telephone system. For a radioactive material expression (3-5) describes the probability that no particle is emitted from the radioactive substance during time t. Constant λ depends on the property of the particular radioactive material. We will see in Chapter 4 that the probability of a nonarrival of a customer in a queueing system is also given by Eq. (3-5). The *probability density function* and the *probability distribution function* (also called the *cumulative probability distribution function*) of this distribution is given in Fig. 3-3. You may recall that the cumulative distribution function $F(t)$ by definition is the integral of the density function, i.e., $F(t) = \int_0^t \lambda e^{-\lambda t}\, dt = 1 - e^{-\lambda t}$. The average or expected value of t is given by

$$Ave = \int_0^\infty t \cdot \lambda\, e^{-\lambda t} \cdot dt = \frac{1}{\lambda}.$$



(a) Density function (b) Distribution function

Fig. 3-3: The exponential distribution.

Now suppose that in order to perform a simulation experiment on some stochastic system governed by an exponential distribution function we require n random samples $t_1, t_2, \ldots, t_n$ from the curve in Fig. 3-3 (*b*). The inverse of F can be obtained by equating u, a generated uniform random number, to

$$u = 1 - e^{-\lambda t}$$

so that

$$-\lambda t = \log_e (1 - u)$$

which gives the corresponding sample t as

$$t = -\frac{\log_e (1 - u)}{\lambda} = F^{-1}(u).$$

Since $(1 - u)$ is as good a uniformly distributed random number between 0 and 1 as u is, we can replace $(1 - u)$ with u itself. That is,

$$t_k = -\frac{1}{\lambda} \log_e u_k. \qquad \ldots(3\text{-}6)$$

We obtain the required n samples $t_1, t_2, \ldots, t_n$ from the exponential distribution $(1 - e^{-\lambda t})$ by transforming n uniform random numbers $u_1, u_2, \ldots, u_n$ in the interval (0, 1), according to Eq. (3-6). To take a specific set of numbers, suppose we need 5 random samples from an exponential distribution with the expected value $\lambda = 3.80$. First we generate 5 random numbers u_k's in the range (0, 1). Suppose these turn out to be .135, .639, .424, .010, .843. Then we take their natural logarithms, $\log_e u_k$, which are -2.00, $-.448$, $-.858$, -4.61, $-.171$. On multiplying these with $-1/3.8$ we get the desired samples, which are:

.526, .118, .226, 1.213, .045.

A pair of FORTRAN statements is sufficient to generate samples (T) from an exponential distribution with any specified LAMDA, λ:

```
RN = RNDY1 (DUM)
 T = -1.0/LAMDA*ALOG (RN)
```

where RANDY is the uniform random number generator in the range (0, 1) as described in Sec. 3-3, and ALOG is the FORTRAN-supplied function for the natural logarithm of a floating-point number.

Look-up tables: Although the two FORTRAN statements for generating samples from an exponential distribution appear to be extremely simple, in reality the computation of the natural logarithm on a digital computer can be time consuming, if thousands of such samples are required. Therefore, sometimes it is more convenient to store the values of $\log_e (1 - u)$ for many values of u; $0 \leqslant u < 1$ along with its slope in the form of a look-up table. In doing so we are trading off memory space for speed of execution. How justified the trade-off is depends on the length of the simulation run, the frequency with which the samples are required, and on the computer being used.

The inverse transformation method (also called the *probability-integral transformation method*), although simple in principle, is often difficult to apply for many probability distributions because we cannot find a mathematical expression for the inverse function F^{-1}. In such cases, the so-called *rejection method* (which does not require an explicit expression for F^{-1}) can be used.

The rejection method: The rejection method for obtaining samples from a given nonuniform distribution basically works by generating uniform random numbers repeatedly and accepting only those that meet a certain condition. This condition of acceptance is so designed that the accepted numbers appear to be drawn from the given distribution. For the rejection method to be applicable, the probability density function $f(t)$ of the distribution must be nonzero only over a finite interval, say (A, B). Let $f(t)$ be bounded by an upper limit C. See Fig. 3-4.

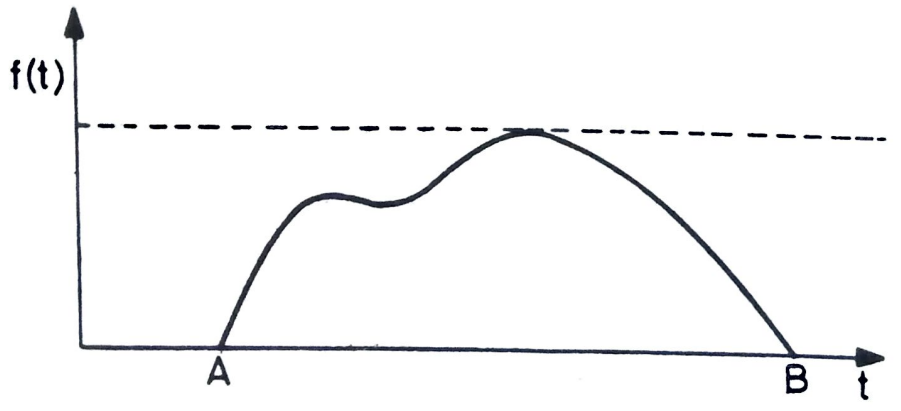


Fig. 3-4: Probability density function.

The rejection procedure consists of the following steps:

(1) Generate a pair of uniform random numbers u_1, u_2 in the interval (0, 1).

(2) Using u_1 locate a point p on the horizontal axis as

$$p = A + (B - A).u_1.$$

(3) Using u_2 locate a point q on the vertical axis as $q = C.u_2$.

(4) If $q > f(p)$ reject the pair and go to the Step (1), othwewise, accept p as the value of a sample from the desired distribution.

The procedure is repeated till the required number of samples have been generated. Fig. 3-5 illustrates how the rejection technique can be used to generate samples from a probability density function.

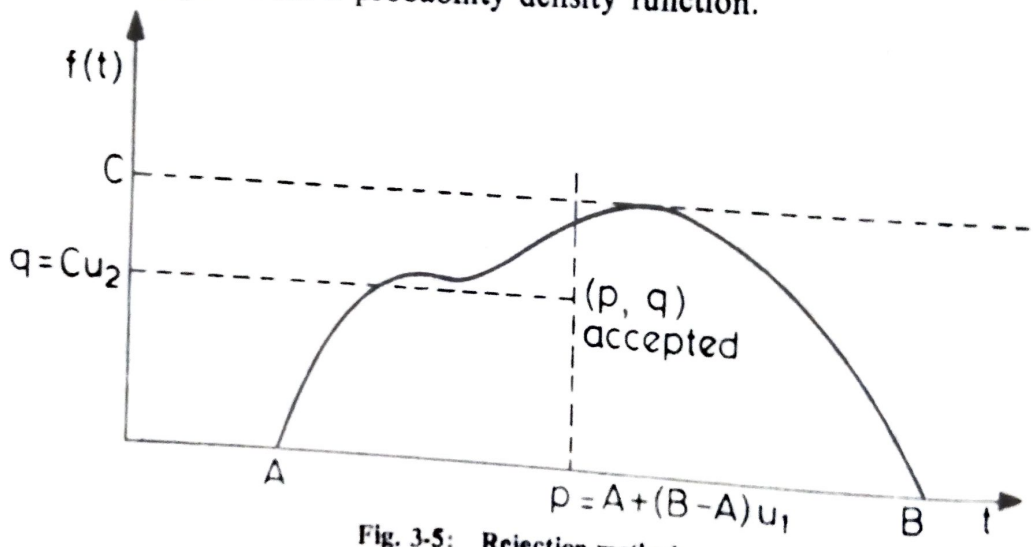


Fig. 3-5: Rejection method.

It is not difficult to see that this procedure does indeed work because all the points above the curve $f(t)$ in the interval (A, B) are rejected. The points that are accepted are within the boundary of the curve and therefore are distributed according to the density function $f(t)$. The restriction that this method works only for a finite interval should be noted.

There are several other methods available for sampling nonuniform distribution functions, and they are chosen according to the requirements of speed, accuracy and the function being sampled. Some of these are applicable for sampling only a specific distribution. Let us consider one such special method, used for the normal distribution only.

Normal distribution: The *normal* or *Gaussian* distribution is the most familiar and frequently encountered probability distribution. Its density function is a bell-shaped curve given by the expression

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \quad \ldots(3\text{-}7)$$

for $-\infty < x < \infty$.

The parameter μ is the expected value and σ is the standard deviation. Note that while a single parameter (λ) was sufficient to specify an exponential distribution, a normal distribution requires two parameters, μ and σ.

If the parameters of the normal distribution have values $\mu = 0$ and $\sigma = 1$, we call it the *standardized normal distribution.* It is expressed by

$$f(s) = \frac{1}{\sqrt{2\pi}} e^{-\frac{s^2}{2}} \qquad \ldots(3\text{-}8)$$

The density function and its integral, i.e., the (cumulative) distribution function are shown in Fig. 3-6.

There does not exist an explicit expression for the cumulative distribution function $F(s)$, although very extensive tables are available in handbooks of statistics.

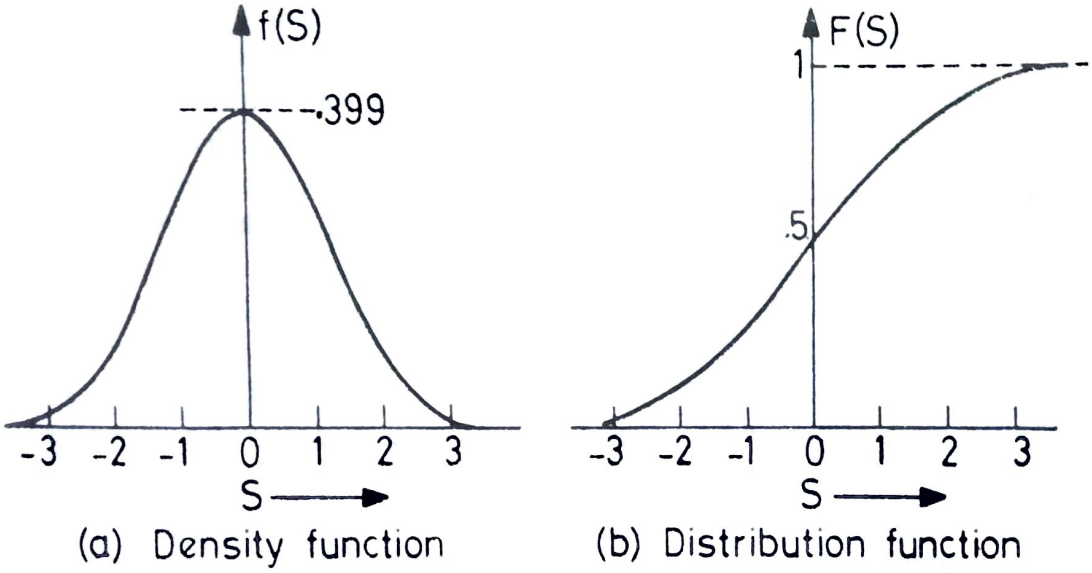


Fig. 3-6: Standardized normal distribution.

One of the commonly used methods for generating random samples from a standardized normal distribution is to use the following relation, called the *Box-Muller transformation.*

$$s = (-2 \log_e r_1)^{1/2} \cos(2\pi r_2) \qquad \ldots(3\text{-}9)$$

where r_1 and r_2 are two uniform random numbers in the range (0, 1), and s is the desired sample from the standardized normal distribution. We will not derive Eq. (3-9) here; its derivation can be found in many textbooks on statistics.

A sample x from any normal distribution with specified μ and σ can be obtained from s, (a sample from the standardized normal distribution), as follows:

$$x = \sigma . s + \mu \qquad \ldots(3\text{-}10)$$

The following FORTRAN subroutine will generate a random sample from a normal distribution using Eqs. (3-9) and (3-10).

```
SUBROUTINE NORMAL (MU, SIGMA, X)
REAL MU, SIGMA, X, R1, R2, V
R1 = RNDY1 (DUM)
R2 = RNDY2 (DUM)
V = (−2.*ALOG (R1))**0.5*COS(6.283*R2)
X = SIGMA*V + MU
RETURN
END
```

Parameters μ and σ are MU and SIGMA in the subroutine. Functions RNDY1 and RNDY2 are two uniform pseudo-random number generators —each with its own seed. Observe that as in the case of exponential distribution, the generation of random samples may become quite time-consuming because of the evaluation of logarithmic and cosine functions. One may therefore resort to look-up tables.

Another commonly used procedure for generating normal samples, which is based on the Central Limit Theorem, will be given in a later chapter.

Multiple Sources of Randomness: In simulation experiments it is often necessary to deal simultaneously with several random variables which are independent. For example, in a queueing situation (with a single server) there may be one random variable for the arrival times of the customer and another one for the service times. Sometimes we need to add two random variables (e.g., two independent sources of delays) with different or same distribution. In all such cases it is best to employ different random number generators, such as RNDY1 and RNDY2 used in SUBROUTINE NORMAL. These different generators may sometimes have the same multiplier, a, and the same modulo, m, but differ only in their seeds, x_0.

3-5. Monte Carlo computation vs. stochastic simulation

The term *Monte-Carlo computation* (or method or technique) was coined during World War II by S. Ulam and J. von Neumann at the Los Alamos Scientific Laboratory. In order to design nuclear shields they needed to know how far neutrons would travel through various materials. The problem was too difficult to solve analytically and too hazardous and time consuming to solve experimentally. Therefore, they simulated the experiment on a high-speed computer using random numbers. This technique was given the code name Monte Carlo, because it was based on a gambling-like principle.

Some people use the term Monte Carlo method to mean any computation that involves random numbers. However, a more generally accepted meaning of Monte Carlo is restricted only to those computations in which random numbers are used to obtain solutions of problems which are inherently deterministic. And for those computations that employ random numbers to solve inherently stochastic problems the term 'stochastic simulation' is used. For example, the inventory problem in Chapter 1 would be a case of stochastic simulation and not Monte Carlo. But the following method of

evaluating the constant π by making use of random numbers would be called a Monte Carlo method.

Deterministic problem through random numbers: Consider a quadrant of a unit circle as shown in Fig. 3-7 below. All points satisfying the equation $x^2 + y^2 < 1$; $x, y \geq 0$ lie in this quadrant. This equation can be rewritten as $y < \sqrt{1 - x^2}$.

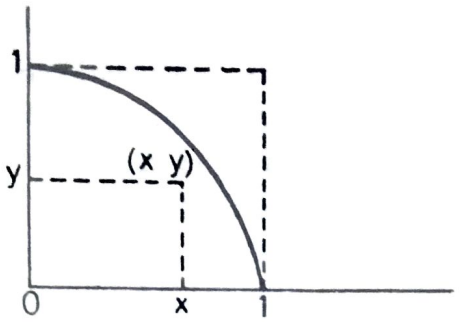


Fig. 3-7: Monte Carlo evaluation of π.

Let us now generate a pair of uniform random numbers u_1 and u_2 in the range (0, 1). We call this pair *acceptable* if

$$u_2 < \sqrt{1 - u_1^2}$$

else we will reject the pair; and generate and test another pair of uniformly distributed random numbers. Clearly all the rejected points like above the curve in Fig. 3-7 and those accepted lie below. If we generate a large number of random pairs and compute the ratio of the number of pairs accepted to those generated, the ratio will approach the area under the curve, which is $\pi/4$.

Thus by using random numbers we have solved a completely deterministic problem. Such an application is called a Monte Carlo technique. (Notice that we have used the rejection technique here; and the area under any curve can be evaluated by this method.)

3-6. Remarks and references

In this chapter we started out with discussing simulation of those dynamic systems in which the changes occur at discrete moments in time. These changes are called events, and our object has been to simulate a sequence of these events and their effect on the state of the system. The passage of time is recorded by a number in the computer called CLOCK. At the beginning of the simulation CLOCK is set to zero and then allowed to run. There are two distinct methods of advancing the clock—(i) steps of fixed lengths or (ii) by updating the value of CLOCK each time an event takes place. In most simulations of discrete systems the latter strategy, called the next-event simulation model, is used.

We also observed that simulation in the study of discrete dynamic systems

is used almost exclusively for stochastic systems—systems in which at least one of the variables is given by a probability function. Complex, discrete dynamic, stochastic systems (which defy an analytic solution) frequently arise in operations research, transportation and traffic engineering, econometrics, etc.

Simulation of a stochastic system requires one or more sequences of random numbers. The idea of generating 'pseudorandom' numbers by means of an (deterministic) arithmetic process was first proposed by J. von Neumann and N. Metropolis in 1946. It was D. H. Lehmer who first proposed the multiplicative congruential method in 1948. Since that time an enormous amount of work has been done in generation, analysis, and testing of pseudorandom numbers. A good account of these techniques can be found in

KNUTH, D. E., *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*, Addison-Wesley, Reading, Mass., 1969 (Chapter 3).

HULL, T. E. and A. R. DOBELL, 'Random Number Generators,' *SIAM Review*, 4, (July 1962), pp. 230–254.

JANSSON, B., *Random Number Generators*, Victor Pettersons, Stockholm, 1966.

REITMAN, J., *Computer Simulation Applications*, John Wiley and Sons, New York, 1971 (Chapter 4).

In many simulation experiments one needs random samples from distributions other than the uniform. The generation of nonuniform stochastic variates from some of the commonly encountered distributions, along with flowcharts and FORTRAN programs, is given in Chapter 4 of

NAYLOR, T. H., J. L. BALINTFY, D. S. BURDICK, and K. CHU, *Computer Simulation Techniques*, John Wiley and Sons, New York, 1966.

Another good source of algorithms for stochastic variate generation is Chapter 8 of

FISHMAN, G. S., *Concepts and Methods in Discrete Event Digital Simulation*, John Wiley, New York, 1973.

The *Communications of the ACM* is a good source of lists for algorithms for generation of different stochastic variates. For example, *CACM* Algorithm Numbers 121 (Sept. 1962), 200 (Aug. 1963), 267 (Oct. 1965), 334 (July 1968), give ALGOL procedures for generation of random normal deviates. Likewise, Algorithm Numbers 342 (Dec. 1968) and 369 (Jan. 1970) give listings for random samples from a Poisson distribution.

A distinction is usually made between Monte Carlo techniques and the simulation methods. The former is applied to the use of random numbers in estimating some quantity which is inherently deterministic, such as evaluation of π. The term simulation (of a discrete dynamic, stochastic system) on the other hand is used when random numbers are used to study a system which is inherently stochastic, such as a queueing system, vehicular traffic pattern, etc. Sometimes the dividing line between the two becomes fuzzy.

The concept of the Monte Carlo technique is quite old. The first Monte Carlo method was proposed in 1777 by Comte de Buffon. The application of the Monte Carlo method (as well as the discrete system simulation) to solve real problems began only since World War II. A thorough survey of mathematical aspects of the Monte Carlo method with a comprehensive bibliography is given in

HALTON, J. H., 'A Retrospective and Prospective Survey of the Monte Carlo Method.' *SIAM Review*, 12, (January 1970), pp. 1–63.

3-7. Exercises

3-1. Write a program to generate and print 5,000 uniformly distributed random numbers between .000 and .999 (3 significant digits). Test the numbers for uniformity by dividing them into 10 ranges and using the chi-square test, as described in Sec. 3-3.

3-2. Write a program to generate uniformly distributed random numbers between 0 and 1 (with as many significant digits as your computer word can hold). Study and comment on the behaviour of the least significant digit of these numbers generated.

3-3. Using a random number generator, write a program to simulate the following game of dice: A pair of six-sided fair dice, each carrying spots 1, 2, 3, 4, 5, and 6 are rolled. If on the first roll the sum of the two dice faces is 7 or 11 the person rolling the dice wins. If this sum is 2, 3, or 12, then the person rolling the dice loses. If the sum is any of the other six numbers (4, 5, 6, 8, 9, 10) this number becomes the 'point' of the person rolling the dice, and the rolling continues till either (i) the 'point' is rolled again, in which case the person rolling the dice wins, or (ii) a 7 is rolled in which case he loses. Run the program to simulate 100 rolls and find who won how man times.

3-4. Write a computer program to simulate 10 dealings of the game of bridge, using random numbers. Neatly print the cards (A, K, Q, J, 10, 9, 8, 7, 6, 5, 4, 3, 2, of clubs, diamonds, hearts and spades) of the four players—North, East, South and West, for each of the 10 dealings.

3-5. The Prisoner's Dilemma (in its various forms) is probably the most frequently used game for live behavioural experiments. Two prisoners *A* and *B* are charged with a joint crime and are held incommunicado. If *A* confesses and *B* does not, *A* is given a reduced sentence of two years for cooperating with the authorities, and *B* gets a 10-year prison term, and vice versa. If both confess each gets six years. If neither confesses, each will receive only a one-year sentence. Neither can control or even know the behaviour of the other. Simulate this game on a computer and study the total penalty under varying probabilities of prisoners taking different actions. [Ref. Emshoff, J., "A Simulation Model of Prisoner's Dilemma," *Behavioral Science*, Vol. 15, No. 4, July 1970.]

3-6. The probability distribution function of the Weibull distribution is defined by

$$F(y) = 1 - e^{-\lambda y^c}, \quad y \geq 0, \quad c > 1, \quad \lambda \geq 0$$

Using the inverse transformation method, write a FUNCTION subprogram WEIB to generate N random samples from the Weibull distribution.

3-7. Write a FUNCTION subprogram ERLANG that uses the rejection technique to obtain random samples from an (*a*) Erlang distribution, described by the density function

$$g(y) = \frac{\lambda(\lambda y)^{n-1} e^{-\lambda y}}{(n-1)!}, \quad \lambda > 0, \; y \geq 0,$$

and n is a positive integer.
(Observe that when $n = 1$, Erlang distribution is the same as the exponential distribution. When n is allowed to have any fractional (positive) value, the more general distribution is called the *gamma* distribution. As n increases, the gamma distribution asymptotically approaches the normal distribution.)

3-8. Write a FUNCTION subprogram BETA that uses the rejection technique to generate random variates from a beta distribution.

3-9. A particle moves along a straight line from some initial position 0 and takes a sequence of N steps. Each step is of unit length and can go left or right, each with probability 1/2. Simulate this experiment for various values of N (say 10, 25, 50, 100, 500), and determine how far the particle is from the origin at the end of each run. This problem, known as the *random-walk problem*, can be solved by statistical techniques, but we prefer to simulate.

3-10. Experiment with the random-walk problem in Exercise 3-9 by making the left and right steps non-equiprobable (.75 and .25; or .6 and 4; etc).

3-11. Instead of keeping the random-walk confined along a straight line (as in Exercises 3-9 and 3-10), let it take place along a 2-dimensional rectangular grid, and simulate.

3-12. Repeat the random-walk simulation on a 3-dimensional (cubic) grid.

3-13. The following is a simplified model of the molecular mixing of two different gases, due to T. Ehrenfest. There are two compartments, separated by a permeable membrane, and each contains M molecules initially. One transfer consists of picking one molecule from each side at random and exchanging them. With $M = 100$, find out what happens after N successive transfers, for various values of N. This problem can also be solved analytically, but we prefer a simulation.

3-14. Write a program to determine the approximate value of $\sqrt{3}$ using 1,000 random numbers. (Hint: Choose 1,000 random numbers x_i's uniformly distributed between 1 and 2. Determine p, the number of these

numbers for which $x_i^2 < 3$. Then estimate the value of 3 as $1 + p/1000$.

3-15. We wish to study the effect of birth rate and death rate on growth of a population, year by year for a given period of N years. We start with a fixed population P_0 (say, 1,000). The birth rate B is constant (say, 20 per thousand per year). The death rate D, on the other hand, is proportional to the current population (say, 15 per thousand, per thousand per year).

Assume that the population is constant throughout the year except on January 1, when all the births and deaths take place. The number of births on January 1 is equal to the integer closest to the birth rate times the number of people alive on December 31. Likewise, the number of deaths is determined by the number of people alive on December 31. Write a computer program to simulate this model of population growth, and plot the population for a 20-year period for several values of parameters, B and D.

3-16. Write a program to simulate the population-growth problem using the following next-event model (instead of the fixed-time-step model of Exercise 3-15): Let $P(t)$ denote the population at time t. Suppose the last birth occurred at time t_B; then since the time between births is inversely proportional to the population and the birth rate B, the next birth would occur at time

$$t_B + \frac{1}{B \times P(T_B)}.$$

Likewise, if the last death occurred at time t_D, the next death would occur at time

$$t_D + \frac{1}{D \times [P(t_D)]^2}.$$

Move the clock from event to event (births and deaths), and trace through a period of 20 years.

SCRIPT II provides eleven functions for generating pseudo-random samples from the following statistical distributions: Uniform, normal, Poisson, exponential, beta, Erlang, lognormal, binomial, gamma, Weibull, and uniformly distributed integers.

The language also provides for collecting statistical information from a simulation run. Two statements, ACCUMULATE and TALLY, are used in the PREAMBLE to instruct the compiler to generate automatic data collection and analysis statements at appropriate places in the program. A FORTRAN subroutine can be invoked from a SIMSCRIPT II program. The language has some capabilities to detect programming errors by the compiler as well as by the runtime system.

8-7. GPSS (*G*eneral *P*urpose *S*imulation *S*ystem)

GPSS, one of the earliest discrete simulation languages, was developed by Geoffrey Gordon and presented in two papers in 1961 and 1962. The first release of this language was implemented on the IBM 704, 709, and 7090 computers. Since then improved and more powerful versions have been developed and implemented, including GPSS II, GPSS III (1965), GPSS/360 (1967), GPSS/360's second version, and GPSS V. The latest version, GPSS V, is a superset of all other implementations of GPSS; a program written in GPSS/360 will run under GPSS V. Through all these versions of this language the fundamental concept of block-diagram structure has been retained.

GPSS was designed specially for analysts who were not necessarily computer programmers. It is particularly suited for modeling traffic and queueing systems.

A GPSS programmer does not write a program in the same sense as a SIMSCRIPT programmer does. Instead, he constructs a *block diagram*—a network of interconnected blocks, each performing a special simulation-oriented function. GPSS V provides a set of 48 different (types of) blocks to choose from—each of which can be used repeatedly. Each block has a name, and a specific task to perform.

Moving through the system of blocks are entities called transactions. Examples of transactions are: customers, messages, machine parts, vehicles, etc. Typical blocks are: (i) GENERATE, creates transactions; (ii) QUEUE, creates a queue of transactions and maintains certain queueing statistics; (iii) TABULATE, tabulates the time it took the transaction to reach that point from the time it entered the simulated system; (iv) TERMINATE, destroys transactions and removes them from the system.

The sequence of events taking place in the simulated systems is realized by the movement of transactions from block to block. There can be many different transactions in the block diagram, but each of them at any given instant of time is located at some block. A block can contain and act upon one or many transactions simultaneously. Usually transactions are

temporary entities that are generated, moved between blocks and removed by the GPSS program. Some transactions are permanent and remain present in the simulation system throughout the entire period of the simulation run. Blocks that can handle only one item at a time are called the *facilities*. An example of a facility is a machine or a person. A related transaction may be a job that has to be performed by the machine or the person. A *storage* represents another block type that can be used by one or more transactions simultaneously. Examples of storages are groups of machines or a warehouse.

GPSS handles the advancement of time by a block called ADVANCE. When a transaction enters this block, an action time is computed and added to the current time to produce a block departure time. When the time reaches the departure time the transaction will be moved, if possible, to the next block in the chart. Transactions might possess certain attributes which can be used to make logical decisions within a block. The GPSS program has also a capability to collect statistical data about the simulated process, e.g., the current number of transactions in a storage or the length of a queue.

Simple mathematical calculations can be carried out with the use of variable statements. Unlike in SIMSCRIPT, there are no elementary mathematical functions in GPSS, such as the trigonometric or logarithmic functions. GPSS can, however, generate a number of basic random variates.

Program's output is produced in a standard form without any specific request from the user. It is also possible to create the output report in some other form specified by the user. Written tabular reports as well as graphs can be generated. Programming errors are detected during the compile as well as the run time. Minor potential errors are communicated to the user by the system-generated warning messages.

Because of its flowchart orientation, GPSS can perhaps be learned more quickly than any other discrete simulation language. A GPSS program is usually much shorter than the same program coded in other discrete simulation languages. It is best suited for systems which can be described in terms of spatial flow (of dynamic elements such as people, cars, telephone calls, orders, etc.) through a fixed-block diagram. But it is difficult to use when the model does not lend itself to being described in terms of transactions flowing through a network of blocks. Thus simplicity and compactness in GPSS (as compared to SIMSCRIPT) has been achieved at the price of flexibility. GPSS is not as general a discrete simulation language as SIMSCRIPT is.

8-8. SIMULA (*SIMU*lation *LA*nguage)

SIMULA 67 is a true extension (i.e., a superset) of ALGOL 60, and is therefore a general-purpose programming language, despite its name. It