CHAPTER

MODELING FORMALISMS AND THEIR SIMULATORS

3

CONTENTS

Introduction 44
3.1 Discrete Time Models and Their Simulators 44
3.1.1 Discrete Time Simulation 46
3.1.2 Cellular Automata 46
3.1.3 Cellular Automaton Simulation Algorithms 48
3.1.4 Discrete Event Approach to Cellular Automaton Simulation 50
3.1.5 Switching Automata/Sequential Machines 51
3.1.6 Linear Discrete Time Networks and Their State Behavior 53
3.2 Differential Equation Models and Their Simulators 55
3.2.1 Linear ODE Models 57
Non-Linear Models 63
3.2.2 Continuous System Simulation 63
3.2.3 Euler's Methods 64
3.2.4 Accuracy of the Approximations 65
3.2.5 Convergence of the Numerical Scheme 68
3.2.6 Numerical Stability 70
3.2.7 One-Step Methods 72
Explicit Runge–Kutta Methods 72
Implicit One-Step Methods 73
3.2.8 Multi-Step Methods 74
Multi-Step Explicit Methods 74
Implicit Multi-Step Methods 75
3.2.9 Step Size Control 75
One-Step Methods 75
3.2.9.1 Multi-Step Methods 78
3.2.10 Stiff, Marginally Stable and Discontinuous Systems 78
Stiff Systems 78
Marginally Stable Systems 80
Discontinuous Systems 82
3.3 Discrete Event Models and Their Simulators 84
3.3.1 Introduction 84
3.3.2 Discrete Event Cellular Automata 84
3.3.3 Discrete Event World Views 87
Event Scheduling World View 87
3.4 Summary 89
3.5 Sources 90
References 91

Theory of Modeling and Simulation. https://doi.org/10.1016/B978-0-12-813370-5.00011-0

INTRODUCTION

This chapter presents, in an informal manner, the basic modeling formalisms for discrete time, continuous and discrete event systems. It is intended to provide a taste of each of the major types of models, how we express behavior in them, and what kinds of behavior we can expect to see. Each modeling approach is also accompanied by its prototypical simulation algorithms. The presentation employs commonly accepted ways of presenting the modeling formalisms. It does not presume any knowledge of formal systems theory and therefore serves as an independent basis for understanding the basic modeling formalisms that will be cast as basic system specifications (DESS, DTSS, and DEVS) after the system theory foundation has been laid. However, since the presentation is rather fast-paced you may want to consult some of the books listed at the end of the chapter for any missing background.

3.1 DISCRETE TIME MODELS AND THEIR SIMULATORS

Discrete time models are usually the most intuitive to grasp of all forms of dynamic models. As illustrated in Fig. 3.1, this formalism assumes a stepwise mode of execution. At a particular time instant the model is in a particular state and it defines how this state changes – what the state at the next time instant will be. The next state usually depends on the current state and also what the environment's influences currently are.

Discrete time systems have numerous applications. The most popular are in digital systems where the clock defines the discrete time steps. But discrete time systems are also frequently used as approximations of continuous systems. Here a time unit is chosen, e.g., one second, one minute or one year, to define an artificial clock and the system is represented as the state changes from one "observation" instant to the next. Therefore, to build a discrete time model, we have to define how the current state and the input from the environment determine the next state of the model.

The simplest way to define the way states change in a model is to provide a table such as Table 3.1. Here we assume there are a finite number of states and inputs. We write down all combinations of states and inputs and next states and outputs for each. For example, let the first column stand for the current state of the model and the second column for the input it is currently receiving. The table gives the next state of the model in column 3 and the output it produces in column 4. In Table 3.1, we have two states (0 and 1) and two inputs (also 0 and 1). There are 4 combinations and each one has an associated state and output. The first row says that if the current state is 0 and the input is 0, then the next state will be 0 and the output will be 0. The other three rows give similar information for the remaining state/input combinations.

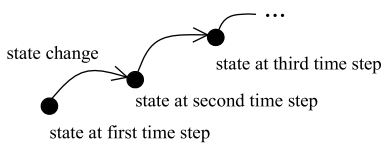

FIGURE 3.1

Stepwise execution of discrete time systems.

Table 3.1 Transition/Output table for a delay system

Current State	Current Input	Next State	Current Output
0	0	0	0
0	1	1	1
1	0	0	0
1	1	1	1

Table 3.2 State and output trajectories

time	0	1	2	3	4	5	6	7	8	9	
input trajectory	1	0	1	0	1	0	1	0	1	0	
state trajectory	0	1	0	1	0	1	0	1	0	1	0
output trajectory	1	0	1	0	1	0	1	0	1	0	

In discrete time models, time advances in discrete steps, which we assume are integers multiples of some basic period such as 1 second, 1 day or 1 year. The transition/output table just discussed would then be interpreted as specifying state changes over time, as in the following:

> if the state at time t is q and the input at time t is x then the state at time $t+1$ will be $\delta(q,x)$ and the output y at time t will be $\lambda(q,x)$.

Here δ is called the state transition function and is the more abstract concept for the first three columns of the table. λ is called the output function and corresponds to the first two and last columns. The more abstract forms, δ and λ constitute a more general way of giving the transition and output information. For example, Table 3.1 can be summarized more compactly as follows:

$$\delta(q,x) = x$$
$$\lambda(q,x) = x$$

which say the next state and current output are both given by the current input. The functions δ and λ are also much more general than the table. They can be thought about and described even when it is tedious to write a table for all combinations of states or inputs or indeed when they are not finite and writing a table is not possible at all. A sequence of states, $q(0), q(1), q(2), \ldots$ is called a state trajectory. Having an arbitrary initial state $q(0)$, subsequent states in the sequence are determined by:

$$q(t+1) = \delta(q(t), x(t)).$$

Similarly, a corresponding output trajectory is given by

$$y(t) = \lambda(q(t), x(t)).$$

Table 3.2 illustrates state and output trajectories (third and fourth rows, respectively) that are determined by the input trajectory in the second row.

Table 3.3 Computing state and output trajectories

time	0	1	2	3	4	5	6	7	8	9	
input trajectory	1	0	1	0	1	0	1	0	1	0	√
state trajectory	0										
output trajectory											√

Table 3.4 State and output trajectories

Current State	Current Input	Next State	Current Output
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

3.1.1 DISCRETE TIME SIMULATION

We can write a little algorithm to compute the state and output trajectories of a discrete time model given its input trajectory and its initial state. Such an algorithm is an example of a simulator (as defined in Chapter 2). Note that the input data for the algorithm corresponds to the entries in Table 3.3.

$T_i = 0, T_f = 9$ – – the starting and ending times, here 0 and 9
$x(0) = 1, ..., x(9) = 0$ – – the input trajectory
$q(0) = 0$ – – the initial state
$t = T_i$
while $(t <= T_f)\{$
    $y(t) = \lambda(q(t), x(t))$
    $q(t+1) = \delta q(t), x(t))$
$\}$

Executing the algorithm fills in the blanks in Table 3.3 (except for those checked).

Exercise 3.1. Execute the algorithm by hand to fill in Table 3.3. Why are the marked squares not filled in.

Exercise 3.2. Table 3.4 is for a model called a binary counter. Hand simulate the model in Table 3.4 for various input sequences and initial states. Explain why it is called a binary counter.

3.1.2 CELLULAR AUTOMATA

Although the algorithm just discussed seems very simple indeed, the abstract nature of the transition and output functions hide a wealth of potentially interesting complexities. For example, what if we connected together systems (as in Chapter 1) in a row with each system connected to its left and right neighbors, as shown in Fig. 3.2.

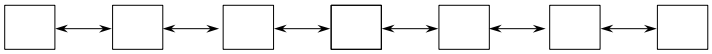

FIGURE 3.2

One dimensional cell space.

Table 3.5 A two input transition function

Current State	Current Left Input	Current Right Input	Next State
0	0	0	?
0	0	1	?
0	1	0	?
0	1	1	?
1	0	0	?
1	0	1	?
1	1	0	?
1	1	1	?

Imagine that each system has two states and gets the states of its neighbors as inputs. Then there are 8 combinations of states and inputs as listed in Table 3.5. We have left the next state column blank. Each complete assignment of a 0 or 1 to the eight rows results in a new transition function. There are $2^8 = 256$ such functions. Suppose we chose any one such function, started each of the components in Fig. 3.2 in one its states and applying an appropriate version of the above simulation algorithm. What would we observe?

The answer to such questions is the concern of the field of cellular automata. A *cellular automaton* is an idealization of a physical phenomenon in which space and time are discretized and the state sets are discrete and finite. Cellular automata have components, called *cells*, which are all identical with identical computational apparatus. They are geometrically located on a one-, two- or multi-dimensional grid and connected in a uniform way. The cells influencing a particular cell, called the *neighborhood* of the cell, are often chosen to be the cells located nearest in the geometrical sense. Cellular automata were originally introduced by von Neumann and Ulam (Burks, 1970) as idealization of biological self-production. Cellular automata show the interesting property that they yield quite diverse and interesting behavior. Actually, Wolfram (Wolfram et al., 1986) systematically investigated all possible transition functions of one-dimensional cellular automata. He found out that there exist four types of cellular automata which differ significantly in their behavior, (1) there are automata where soon any dynamic dies out, (2) automata which soon come to periodic behavior, (3) automata which show chaotic behavior, and (4), the most interesting ones, automata whose behaviors are unpredictable and non-periodic but which show interesting, regular patterns.

Conway's Game of Life in its original representation in Scientific American (Gardner, 1970) can serve as a fascinating introduction to the ideas involved. The game is framed within a two-dimensional cell space structure, possibly of infinite size. Each cell is coupled to its nearest physical neighbors both laterally and diagonally. This means for a cell located at point (0, 0) its lateral neighbors are at $(0, 1)$, $(1, 0)$, $(0, -1)$, and $(-1, 0)$ and its diagonal neighbors are at $(1, 1)$, $(1, -1)$, $(-1, 1)$ and $(-1, -1)$ as shown in Fig. 3.3. The neighbors of an arbitrary cell at (i, j) are the cells at

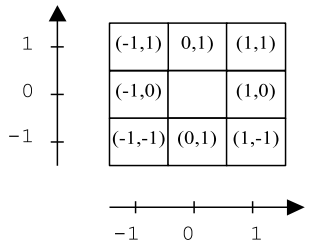

FIGURE 3.3

Cellular coupling structure for Game of Life.

$(i, j+1)$, $(i+1, j)$, $(i, j-1)$, $(i-1, j)$, $(i+1, j+1)$, $(i+1, j-1)$, $(i-1, j+1)$, and $(i-1, j-1)$ which can be computed from the neighbors at $(0, 0)$ by a simple translation. The *state* set of a cell consists of one variable which can take on only two values, 0 (*dead*) and 1 (*alive*). Individual cells survive (are alive and stay alive), are born (go from 0 to 1) or die (go from 1 to 0) as the game progresses. The rules as defined by Conway are:

1. A live cell remains alive if it has between 2 and 3 live cells in its neighborhood.
2. A live cell will die due to overcrowding if the has more than 3 live cells in its neighborhood.
3. A live cell will die due to isolation if it has less than 2 live neighbors.
4. A dead cell will become alive if it has exactly 3 alive neighbors.

When started from certain configurations of alive cells, the Game of Life show interesting behavior over time. As a state trajectory evolves, live cells form varied and dynamically changing clusters. The idea of the game is to find new patterns and study their behavior. Fig. 3.4 shows some interesting patterns.

The Game of Life exemplifies some of the concepts introduced earlier. It evolves on a *discrete time base* (time advances in steps 0, 1, 2, ...) and is a *multi-component system* (it is composed of *components* (cells) that are coupled together). In contrast to the *local* state (the state of a cell), the *global state* refers to the collection of states of **all** cells at any time. In Game of Life, this is a finite pattern, or configuration, of alive cells with all the rest being dead. Every such state starts a *state trajectory* (sequence of global states indexed by time) which either ends up in a cycle or continues to evolve forever.

Exercise 3.3. If an initial state is a finite configuration of alive cells, why are all subsequent states also finite configurations.

3.1.3 CELLULAR AUTOMATON SIMULATION ALGORITHMS

The basic procedure for simulating a cellular automaton follows the discrete time simulation algorithm introduced earlier. This is, at every time step, we scan all cells applying the state transition function to each, and saving the next state in a second copy of the global state data structure. When all next-states have been computed, they constitute the next global state and the clock advances to the next step. For example, let's start with the three live cells in a triangle shown in the upper left of Fig. 3.5. Analyzing the neighborhood of the lower corner cell, we see it is alive and has 2 alive neighbors, thus it survives

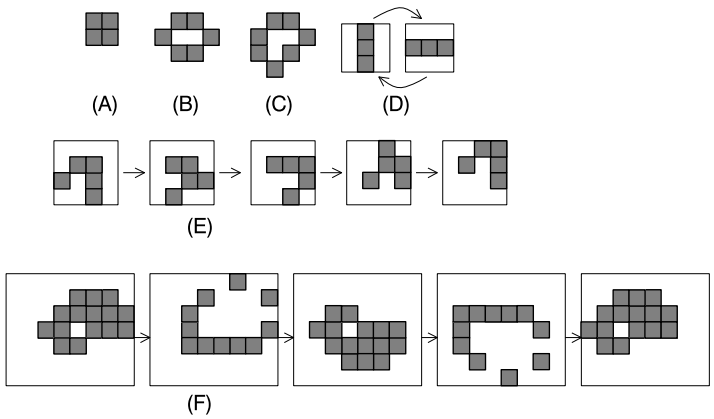

FIGURE 3.4

Patterns from Conway's Game of Life: Patterns (A) to (C) are stable, they don't change, (D) is an oscillating patterns, (E) and (F) are cycles of patterns which move.

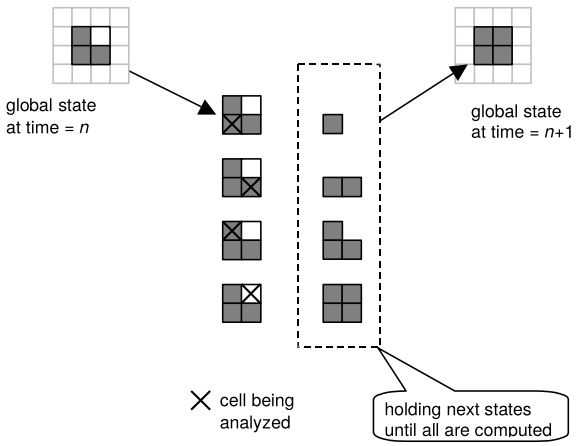

FIGURE 3.5

Cellular Automaton Simulation Cycle.

to the next generation. As shown, the other alive cells also survive. However, only one new cell is born. This is the upper right corner cell which has exactly three neighbors.

Of course as stated, the cell space is an infinite and we can't possibly scan all the cells in a finite amount of time. To overcome this problem, the examined part of the space is limited to a finite region. In many case, this region is fixed throughout the simulation. For example, a two-dimensional space might be represented by a square array of size $N(= 1, 2, ...)$. In this case, the basic algorithm scans all N^2 cells at every time step. Something must be done to take care of the fact that cells at the boundary

lack their full compliment of neighbors. One solution is to assume fixed values for all boundary cells (for example, all dead). Another is to wrap the space in toroidal fashion (done by letting the index N also be interpreted as 0). But there is a smarter approach that can handle a potentially infinite space by limiting the scanning to only those cells that can potentially change states at any time. This is the discrete event approach discussed next.

3.1.4 DISCRETE EVENT APPROACH TO CELLULAR AUTOMATON SIMULATION

In discrete time systems, at every time step each component undergoes a "state transition"; this occurs whether or not its state actually changes. Often, only a small number of components really change. For example, in the Game of Life, the dead state is called a *quiescent state* – if the cell and all its neighbors are in the quiescent state, its next state is also quiescent. Since most cells in an infinite space are in the quiescent state, relatively few actually change state. Put another way, if *events* are defined as changes in state (e.g., births and deaths in the Game of Life), then often there are relatively few events in the system. Scanning all cells in an infinite space is impossible, but even in a finite space, scanning all cells for events at every time step is clearly inefficient. A discrete event simulation algorithm **concentrates on processing events** rather than cells, and is inherently more efficient.

The basic idea is to try to predict whether a cell will possibly change state or will definitely be left unchanged in a next global state transition. Suppose we have a set of potentially changing cells. Then we only examine those cells in the set. Some or all of these cells are observed to change their states. Then, in the new circumstances, we collect the cells which can possibly change state in the next step. The criterion under which cells can change is simple to define. *A cell will not change state at the next state transition time, if none of its neighboring cells changed state at the current state transition time*. The event-based simulation procedure follows from this logic:

In a state transition mark those cells which actually changed state. From those, collect the cells which are their neighbors. The set collected contains all cells which can possibly change at the next step. All other cells will definitely be left unchanged.

For example, Fig. 3.6 continues the simulation just discussed. Since only the black cell changed state, only its eight neighbors can possibly change state in the next simulation cycle. In other words, if we start with cells at which events have occurred, then we can readily predict where the next events can occur.

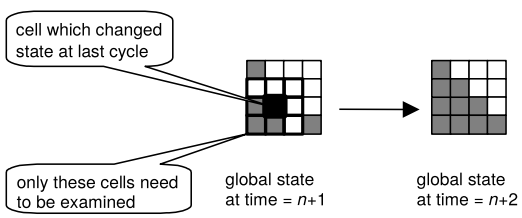

FIGURE 3.6

Collecting cells with possible events.

Table 3.6 State and output trajectories of a delay flip-flop

time	0	1	2	3	4	5	6	7	8	9	
input trajectory	1	0	1	0	1	0	1	0	1	0	
state trajectory	0	1	0	1	0	1	0	1	0	1	0
output trajectory	0	1	0	1	0	1	0	1	0	1	

Note that although we can limit the scanning to those cells that are candidates for events, we cannot *predict which cells will actually have events* without actually applying the transition function to their neighborhood states.

Exercise 3.4. Identify a cell which is in the can-possibly-change set but which does not actually change state.

Exercise 3.5. Write a simulator for one-dimensional cellular automata such as in Table 3.5 and Fig. 3.2. Compare the direct approach which scans all cells all the time with the discrete event approach in terms of execution time. Under what circumstances would it not pay to use the discrete event approach? (See later discussion in Chapter 17.)

Exercise 3.6. Until now we have assumed a neighborhood of 8 cells adjacent to the center cell. However, in general, a neighborhood for the cell at the origin is defined as any finite subset of cells. This neighborhood is translated to every cell. Such a neighborhood is called reflection symmetric if whenever cell (i, j) belongs to it then so does (j, i). For neighborhoods that are not reflection symmetric, define the appropriate set of influencees required for the discrete event simulation approach.

3.1.5 SWITCHING AUTOMATA/SEQUENTIAL MACHINES

Cellular automata are uniform both in their composition and interconnection patterns. If we drop these requirements but still consider connecting finite state components together we get another useful class of discrete time models.

Switching Automata (also called digital circuits) are constructed from flip-flop components and logical gates. The flip-flops are systems with binary states, the most straightforward example of which we have already met. This has the transition function shown in Table 3.1. However, instead of allowing the input to propagate straight through to the output, we will let the output be the current state (as in cellular automata). Table 3.6 shows how the output trajectory now lags the input trajectory by one time step. The difference, we'll see later, is between so-called Mealy and Moore models of sequential machines. In Mealy networks the effects of an input can propagate throughout space in zero time – even cycling back on themselves, causing "vicious circles" or ill-behaved systems.

To construct networks we can not only couple flip-flop outputs to inputs, but also connect them through so-called *gates*, which realize elementary Boolean functions. As we shall see later, these are instantaneous or memoryless systems. As shown in Fig. 3.7, their outputs are directly determined by their inputs with no help from an internal state. The clock usually used in synchronous machines determines a constant time advance. This enables transistor circuits to be adequately represented by discrete time models, at least for their functional behavior.

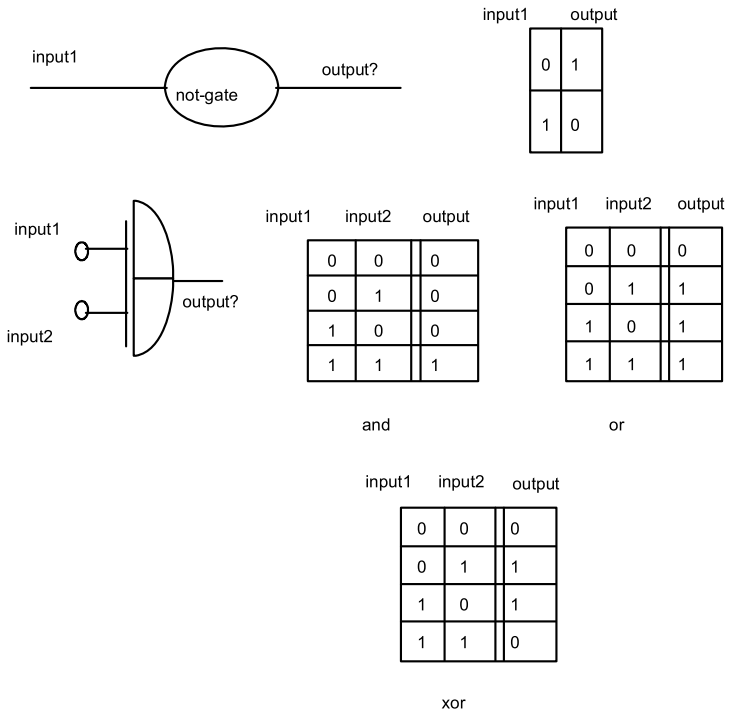

FIGURE 3.7

Gates Realizing Elementary Boolean Functions.

The switching automaton in Fig. 3.8 is made up of several flip-flop components coupled in series and a feedback function defined by a coupling of XOR-gates. Each flip-flop is an elementary memory component as above. Flip-flops q_1 to q_7 are coupled in series, that is, the state of flip-flop i defines the input and hence the next state of flip-flop i-1. This linear sequence of flip-flops is called a *shift-register* since it shifts the input at the first component to the right each time step. The latter input may be

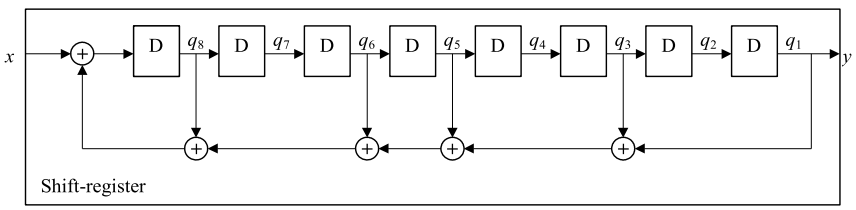

FIGURE 3.8

Shift register coupling flip-flop elements and logical gates.

computed using gates. For example, the input to flip-flop 8 is defined by an XOR-connection of memory values q_i and input value $x : x \oplus q_8 \oplus q_6 \oplus q_5 \oplus q_3 \oplus q_1$. (An XOR, $q_1 \oplus q_2$, outputs the exclusive-or operation on its inputs, i.e., the output is 0 if, and only if, both inputs are the same, otherwise 1.)

In Chapter 8 we will discuss simulators for such combinations of memoryless and memory-based systems.

3.1.6 LINEAR DISCRETE TIME NETWORKS AND THEIR STATE BEHAVIOR

Fig. 3.9 illustrates a Moore network with two delay elements and 2 memoryless elements. However, in distinction to the switching automaton above, this network now is defined over the reals. The structure of the network can be represented by the matrix:

$$\begin{bmatrix} 0 & g \\ -g & 0 \end{bmatrix}$$

where g and $-g$ are the gain factors of the memoryless elements. Started in a state represented by the vector [1, 1], (each delay is in state 1), the next state $[q_1, q_2]$ of the network is $[-g, g]$ which can be computed by the matrix multiplication:

$$\begin{bmatrix} q_1 \\ q_2 \end{bmatrix} = \begin{bmatrix} 0 & g \\ -g & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} g \\ -g \end{bmatrix}$$

The reason that a matrix representation and multiplication can be used is that the network has a *linear* structure. This means that all components produce outputs or states that are linear combinations of their inputs and states. A delay is a very simple linear component – its next state is identical to its input. A memoryless element with a gain factor is called a *coefficient* element and it multiplies its current input by the gain to produce its output. Both of these are simple examples of linearity. A summer, which a memoryless function that adds its inputs to obtain its output, is also a linear element.

A Moore discrete time system is in *linear matrix* form if its transition and output functions can be represented by matrices $\{A, B, C\}$. This means that its transition function can be expressed as:

$$\lambda(q, x) = Aq + Bx$$

Here q is a real n-dimensional state vector, and A is an n by n matrix. Similarly, x is a real m-dimensional input vector, and B has dimension m by n. Also the output function is

$$\lambda(q) = Cq$$

where, if the output y is a p-dimensional vector, then C is an n by p-dimensional matrix. A similar definition holds for Mealy discrete time system.

Exercise 3.7. Any discrete time network with linear structure can be represented by linear system in matrix form. Conversely, any linear discrete time system in matrix form can be realized by a discrete time network with linear structure. Develop procedures to convert one into the other. The state trajectory behavior of the linear system in Fig. 3.9 is obtained by interatively multiplying the matrix A into

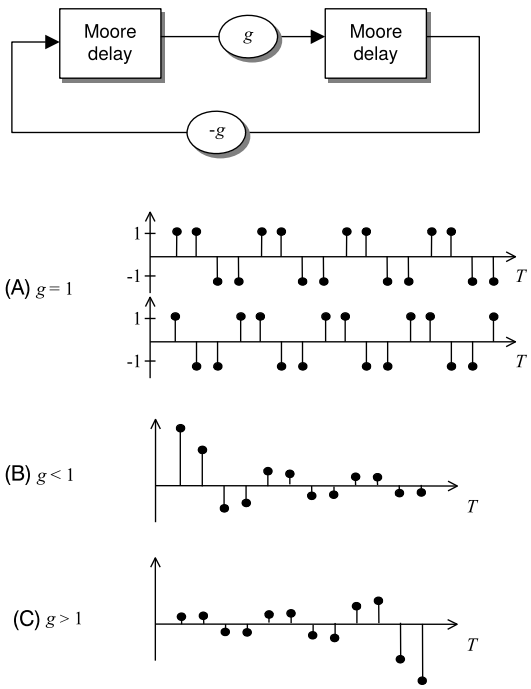

FIGURE 3.9

Simple Linear Moore Network and its Behavior.

successive states. Thus if $[g, -g]$ is the state following $[1, 1]$, then the next state is:

$$\begin{bmatrix} -g^2 \\ g^2 \end{bmatrix} = \begin{bmatrix} 0 & g \\ -g & 0 \end{bmatrix} \begin{bmatrix} g \\ -g \end{bmatrix}$$

Thus, the state trajectory starts out as:

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} \cdots \begin{bmatrix} -g^2 \\ g^2 \end{bmatrix} \cdots \begin{bmatrix} g \\ -g \end{bmatrix}$$

Exercise 3.8. Characterize the remainder of the state trajectory.

There are three distinguishable types of trajectories as illustrated in Fig. 3.9. When $g = 1$, we have a steady alternation between $+1$ and -1. When $g < 1$, the envelope of values decays exponentially; and finally, when $g > 1$, the envelope increases exponentially.

Actually the model above is a discrete form of an oscillating system. In the following sections we will learn to understand its better-known continuous system counterpart. In the discrete form above,

oscillations (switching from positive to negative values) come into being due to the delays in the Moore components and the negative influence $-g$. In one step the value of the second delay is inverted giving a value with opposite sign to the first delay. In the next step, however, this inverted value in the first delay is fed into the second delay. The sign of the delay values is inverted every second step.

We will see that in the continuous domain to be discussed, oscillation also comes into being through feedback. However, the continuous models work with derivatives instead of input values. The delays in the discrete domain correspond to integrators in the continuous domain.

3.2 DIFFERENTIAL EQUATION MODELS AND THEIR SIMULATORS

In discrete time modeling we had a state transition function which gave us the information of the state at the next time instant given the current state and input. In the classical modeling approach of differential equations, the state transition relation is quite different. For differential equation models we do not specify a next state directly but use a derivative function to specify the rate of change of the state variables. At any particular time instant on the time axis, given a state and an input value, we only know the rate of change of the state. From this information, the state at any point in the future has to be computed.

To discuss this issue, let us consider the most elementary continuous system – the simple integrator (Fig. 3.10). The integrator has one input variable $u(t)$ and one output variable $y(t)$. One can imagine it as a reservoir with infinite capacity. Whatever is put into the reservoir is accumulated – but a negative input value means a withdrawal. The output of the reservoir is its current contents. When we want to express this in equation form we need a variable to represent the current contents. This is our state variable $z(t)$. The current input $u(t)$ represents the rate of current change of the contents which we express by equation

$$\frac{\mathrm{d}z(t)}{\mathrm{d}t} = u(t)$$

and the output $y(t)$ is equal to the current state $y(t) = z(t)$.

Fig. 3.11 shows input and output trajectories in a simple integrator.

Usually continuous time systems are expressed by using several state variables. Derivatives are then functions of some, or all, the state variables. Let $z_1, z_2, ..., z_n$ be the state variables and $u_1, u_2, ..., u_m$ be the input variables, then a continuous time model is formed by a set of first-order differential equations

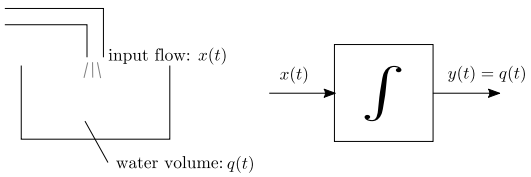

FIGURE 3.10

Integrator: A reservoir model and its representation.

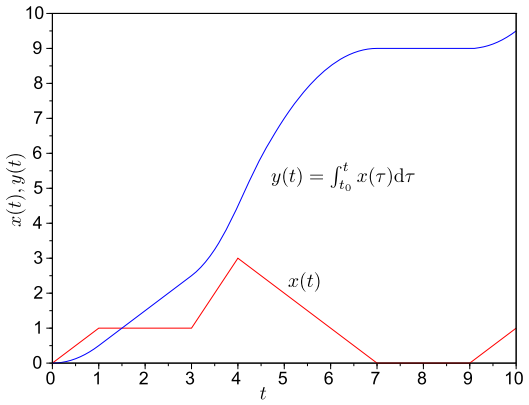

FIGURE 3.11

Integrator: Input and output trajectories.

(ODEs):

$$\begin{aligned}
\dot{z}_1(t) &= f_1(z_1(t), \cdots, z_n(t), u_1(t), \cdots, u_m(t)) \\
\dot{z}_2(t) &= f_2(z_1(t), \cdots, z_n(t), u_1(t), \cdots, u_m(t)) \\
&\vdots \\
\dot{z}_n(t) &= f_n(z_1(t), \cdots, z_n(t), u_1(t), \cdots, u_m(t))
\end{aligned}$$

where $\dot{z}_i$ stands for $\frac{\mathrm{d}z_i}{\mathrm{d}t}$. In order to have a more compact notation, the state variables are grouped in a state vector $\mathbf{z}(t) \triangleq [z_1(t), \cdots, z_n(t)]^T$. Similarly, the input variables for the input vector $\mathbf{u}(t) \triangleq [u_1(t), \cdots u_m(t)]$, so the equations are written as

$$\begin{aligned}
\dot{z}_1(t) &= f_1(\mathbf{z}(t), \mathbf{u}(t)) \\
\dot{z}_2(t) &= f_2(\mathbf{z}(t), \mathbf{u}(t)) \\
&\vdots \\
\dot{z}_n(t) &= f_n((\mathbf{z}(t), \mathbf{u}(t))
\end{aligned} \tag{3.1}$$

Note that the derivatives of the state variables z_i are computed respectively, by functions f_i which have the state and input vectors as arguments. This can be shown in diagrammatic form as in Fig. 3.12. The state and input vector are input to the rate of change functions f_i. Those provide as output the derivatives $\dot{z}_i$ of the state variables z_i which are forwarded to integrator blocks. The outputs of the integrator blocks are the state variables z_i.

Functions f_i in Eq. (3.1) can be also grouped in a vector function $\mathbf{f} \triangleq [f_1, \cdots, f_n]$, obtaining the classic *state equation* representation of ODEs:

$$\dot{\mathbf{z}}(t) = \mathbf{f}(\mathbf{z}(t), \mathbf{u}(t)) \tag{3.2}$$

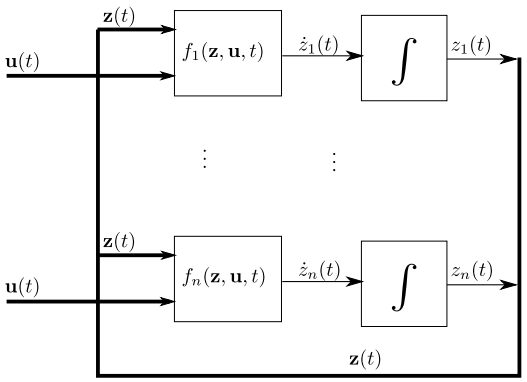

FIGURE 3.12

Block Diagram representation of Eq. (3.1).

Most continuous time models are actually written in (or converted to) the form of Eq. (3.2), a representation that does not provide an explicit value for the state $\mathbf{z}(t)$ after some amount of time. Thus, in order to obtain the state trajectories the ODE must be solved. The problem is that obtaining a solution for Eq. (3.2) can be not only very difficult but also impossible as only very few ODEs have analytical solution in terms of known functions and expressions. This is the reason why ODEs are normally solved using numerical integration algorithms that provide approximate solutions.

Before introducing the main concepts of numerical integration algorithms, we shall study the solution of Linear Time-Invariant (LTI) ODEs, as they can be analytically solved. For background, you may wish to consult some of the many texts that cover such systems and their underlying theory as listed at end of the chapter.

3.2.1 LINEAR ODE MODELS

Consider again the model of the tank represented by an integrator in Fig. 3.10, but suppose that now the tank has a valve in the bottom where the water flows away. Suppose also that the flow is proportional to the volume of water in the tank.

Under these new assumptions, the derivative of the water volume can be written as

$$\dot{z}(t) = u(t) - a \cdot z(t) \tag{3.3}$$

where $z(t)$ is the volume of water, $u(t)$ is the input flow, and $\lambda \cdot z(t)$ is the output flow.

Let us suppose that the input flow is null (i.e., $u(t) = 0$), and that the initial volume of water is $z(t_0) = z_0$. Then, it can be easily checked that the volume of water follows the trajectory:

$$z(t) = e^{-a \cdot t} \cdot z_0 \tag{3.4}$$

Notice that differentiating both terms of Eq. (3.4) with respect to t you obtain back the expression of Eq. (3.3). The trajectory given by Eq. (3.4) is depicted in Fig. 3.13.

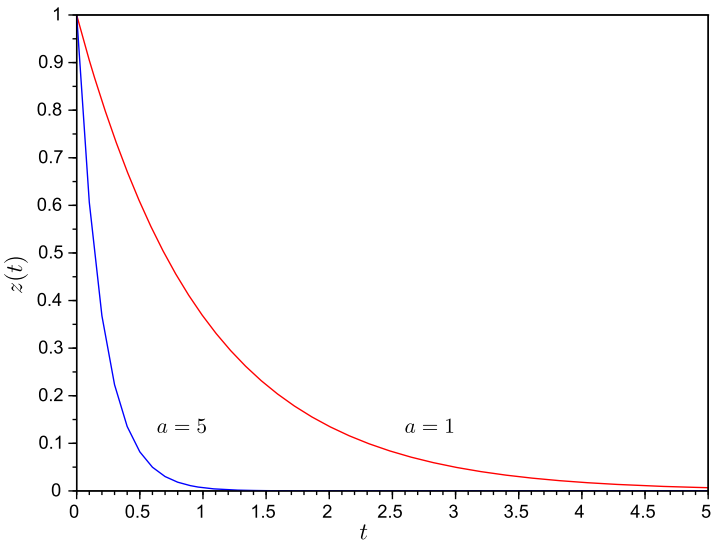

FIGURE 3.13

Solution of the ODE (3.3) for $z_0 = 1$, $a = 1$, and $a = 5$.

The constant $\lambda \triangleq -a$ is the ODE *eigenvalue*, and it provides the *speed* at which the solution converges to the final value. A large value of a implies that the solution converges fast, while a small value of a says that the trajectory slowly goes to its equilibrium value.

Consider now a different model that comes from population dynamics. Let us suppose that $z(t)$ is the number of individuals of certain specie. We may assume that the unit for $z(t)$ is given in billions of specimens so that non-integer numbers have some physical sense. We shall suppose that b is the birth rate per year, i.e., the number of individuals that are born each year is $b \cdot z(t)$. Similarly, d will be the death rate per year.

Thus, the population growth can expressed by the following ODE:

$$\dot{z}(t) = b \cdot z(t) - d \cdot z(t) = (b - d) \cdot z(t) \tag{3.5}$$

which is identical to Eq. (3.3) with $u(t) = 0$ and $a = d - b$. Thus, the solution is

$$z(t) = e^{(b-d)\cdot t} \cdot z_0 \tag{3.6}$$

Let us assume that the model corresponds to the world population of people. According to 2015 data, the birth rate is $b = 0.0191$ (i.e. there are 19.1 births every year per 1000 inhabitants), while the death rate is $r = 0.0076$. Taking into account that the world population of 2015 was about 7355 billion, if we consider 2015 as the initial time of simulation, we have $z_0 = 7355$. The trajectory given by Eq. (3.6) is depicted in Fig. 3.14.

Notice that this time, unlike Fig. 3.13, the solution diverges and after 100 years we may expect a population of 23,000 billion inhabitants (this is assuming that birth and death rates are kept constant).

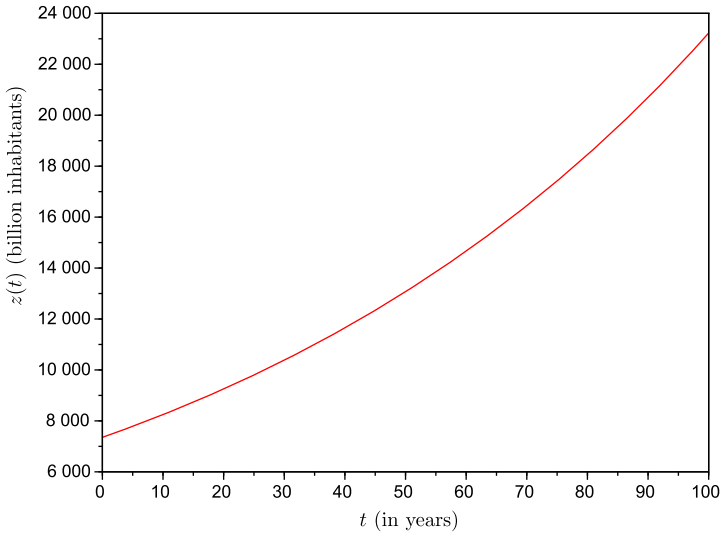

FIGURE 3.14

Solution of the ODE (3.5).

The fact that the solution diverges is mathematically explained by the fact that the eigenvalue $\lambda = b - d$ is now positive, so the expression $e^{\lambda \cdot t}$ goes to infinity as t advances. In conclusion, the sign of the eigenvalue λ dictates the *stability* of a linear time invariant system of first order. The solution is *stable* when $\lambda < 0$ and it is *unstable* when $\lambda > 0$.

Let us now consider a second order model that represents a spring–mass–damper mechanical system:

$$\begin{aligned} \dot{z}_1(t) &= z_2(t) \\ \dot{z}_2(t) &= -\frac{k}{m} z_1(t) - \frac{b}{m} z_2(t) + \frac{F(t)}{m} \end{aligned} \tag{3.7}$$

Here, the variables $z_1(t)$ and $z_2(t)$ are the position and speed, respectively. The parameter m is the mass, b is the friction coefficient, k is the spring coefficient and $F(t)$ is an input trajectory force applied to the mass.

Solving higher order LTI models is a bit more complicated than solving first order models. One way to do it is to introduce a change of variables that converts the n-th ODE in a set of n independent ODEs.

Given a LTI ODE of the form

$$\dot{\mathbf{z}}(t) = A \cdot \mathbf{z}(t) + B \cdot \mathbf{u}(t) \tag{3.8}$$

we introduce a new variable $\xi(t)$ so that $\mathbf{z}(t) = V \cdot \xi(t)$ where V is some $n \times n$ invertible matrix. Then, replacing this expression in Eq. (3.8), we obtain

$$V \cdot \dot{\xi}(t) = A \cdot V \cdot \xi(t) + B \cdot \mathbf{u}(t)$$

and then,

$$\dot{\xi}(t) = V^{-1} A \cdot V \cdot \xi(t) + V^{-1} B \cdot \mathbf{u}(t) \triangleq \Lambda \xi(t) + B_\xi \cdot \mathbf{u}(t) \tag{3.9}$$

If V is computed as an *eigenvector* matrix of A, then matrix $\Lambda \triangleq V^{-1} A \cdot V$ is a diagonal matrix with the eigenvalues of A in its main diagonal. That way, Eq. (3.9) can be expanded as

$$\begin{aligned}
\dot{\xi}_1(t) &= \lambda_1 \cdot \xi_1(t) + B_{\xi,1} \cdot \mathbf{u}(t) \\
\dot{\xi}_2(t) &= \lambda_2 \cdot \xi_2(t) + B_{\xi,2} \cdot \mathbf{u}(t) \\
&\vdots \\
\dot{\xi}_n(t) &= \lambda_n \cdot \xi_n(t) + B_{\xi,n} \cdot \mathbf{u}(t)
\end{aligned} \tag{3.10}$$

where $B_{\xi,n}$ is the i-th row of B_ξ. The solution of this system of equations is given by

$$\begin{aligned}
\xi_1(t) &= e^{\lambda_1 \cdot t} \cdot \xi_1(0) + \cdot \int_0^t e^{\lambda_1 \cdot (t-\tau)} B_{\xi,1} \mathbf{u}(\tau) \mathrm{d}\tau \\
\xi_2(t) &= e^{\lambda_2 \cdot t} \cdot \xi_2(0) + \cdot \int_0^t e^{\lambda_2 \cdot (t-\tau)} B_{\xi,2} \mathbf{u}(\tau) \mathrm{d}\tau \\
&\vdots \\
\xi_n(t) &= e^{\lambda_n \cdot t} \cdot \xi_n(0) + \cdot \int_0^t e^{\lambda_n \cdot (t-\tau)} B_{\xi,n} \mathbf{u}(\tau) \mathrm{d}\tau
\end{aligned} \tag{3.11}$$

where $\xi(0) = V^{-1} \cdot \mathbf{z}(0)$. This solution can be brought back to the original state variables $\mathbf{z}(t)$ using the transformation $\mathbf{z}(t) = V \cdot \xi(t)$. Then, each component $z_i(t)$ follows a trajectory of the form

$$z_i(t) = \sum_{j=1}^{n} c_{i,j} \cdot e^{\lambda_j \cdot t} + \sum_{j=1}^{n} d_{i,j} \cdot \int_0^t e^{\lambda_j \cdot (t-\tau)} B_{\xi,j} \mathbf{u}(\tau) \mathrm{d}\tau \tag{3.12}$$

for certain constants $c_{i,j}$ and $d_{i,j}$.

This way, the generic solution is governed by terms of the form $e^{\lambda_i \cdot t}$ depending exclusively on the eigenvalues of matrix A. The terms $e^{\lambda_i \cdot t}$ are called *system modes*.

Coming back the second order system of Eq. (3.7), we consider first the set of parameters $m = 1$, $b = 3$, $k = 2$, and, for the sake of simplicity, we assume that the input is null ($F(t) = 0$). Matrix A has the form

$$A = \begin{bmatrix} 0 & 1 \\ -k/m & -b/m \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -2 & -3 \end{bmatrix}$$

whose eigenvalues are the solutions of the determinant, $\det(\lambda) \cdot I - A = 0$, resulting $\lambda_1 = -1$, $\lambda_2 = -2$. Then, given an initial state $\mathbf{z}(0) = [z_1(0), z_2(0)]^T$, the solution of Eq. (3.12) results

$$\begin{aligned}
z_1(t) &= 2 \cdot (z_1(0) + z_2(0)) \cdot e^{-t} - (z_1(0) + z_2(0)) \cdot e^{-2 \cdot t} \\
z_2(t) &= (-2 \cdot z_1(0) - z_2(0)) \cdot e^{-t} + 2 \cdot (z_1(0) + z_2(0)) \cdot e^{-2 \cdot t}
\end{aligned} \tag{3.13}$$

Notice that this solution can be thought as the sum of the solutions of first order models like that of the tank in Eq. (3.3).

Let us consider now that the parameters are $b = m = k = 1$. This time, matrix A becomes

$$A = \begin{bmatrix} 0 & 1 \\ -k/m & -b/m \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & -1 \end{bmatrix}$$

whose eigenvalues are now

$$\lambda_{1,2} = -\frac{1}{2} \pm i \cdot \frac{\sqrt{3}}{2} \tag{3.14}$$

where $i = \sqrt{-1}$ is the imaginary unit. In presence of complex eigenvalues like those of Eq. (3.14), the system modes have the form

$$e^{\lambda_i \cdot t} = e^{\mathbb{R}\mathrm{e}(\lambda_i)\cdot t} \cdot e^{i \cdot \mathbb{I}\mathrm{m}(\lambda_i)\cdot t} = e^{\mathbb{R}\mathrm{e}(\lambda_i)\cdot t} \cdot (i \sin(\mathbb{I}\mathrm{m}(\lambda_i) \cdot t) + \cos(\mathbb{I}\mathrm{m}(\lambda_i) \cdot t))$$

so the trajectories have oscillations with a frequency given by $\mathbb{I}\mathrm{m}(\lambda_i)$ and with an amplitude that goes to zero with the time when $\mathbb{R}\mathrm{e}(\lambda_i)$ is negative, or can become larger and diverge when $\mathbb{R}\mathrm{e}(\lambda_i) > 0$. A critical case is that of $\mathbb{R}\mathrm{e}(\lambda_i) = 0$, where sustained oscillations are obtained.

Coming back to the case of the second order system of Eq. (3.7) with $m = b = k = 1$, the solution for input $F(t) = 1$ and initial state $z_1(0) = z_2(0) = 0$ is given by

$$\begin{aligned} z_1(t) &= 1 - \frac{\sqrt{3}}{3} e^{-t/2} \sin(\frac{\sqrt{3}}{2} t) - e^{-t/2} \cos(\frac{\sqrt{3}}{2} t) \\ z_2(t) &= \frac{\sqrt{12}}{3} e^{-t/2} \sin(\frac{\sqrt{3}}{2} t) \end{aligned} \tag{3.15}$$

and depicted in Fig. 3.15.

If we set the friction coefficient $b = 0$, matrix A results

$$A = \begin{bmatrix} 0 & 1 \\ -k/m & -b/m \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

whose eigenvalues are now

$$\lambda_{1,2} = -i \tag{3.16}$$

and the trajectories for $F(t) = 1$ and null initial conditions are now the following undamped oscillations

$$\begin{aligned} z_1(t) &= 1 - \cos(t) \\ z_2(t) &= \sin(t) \end{aligned} \tag{3.17}$$

depicted in Fig. 3.16.

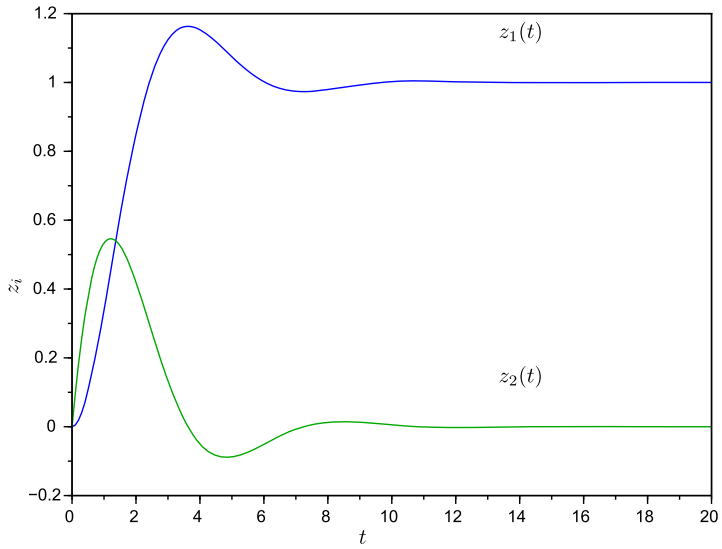

FIGURE 3.15

Solution of Eq. (3.7) for $b = m = k = 1$, and $F(t) = 1$.

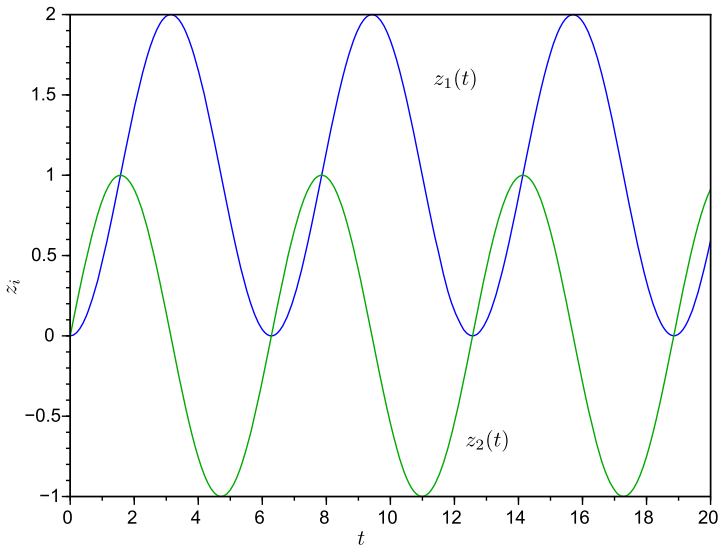

FIGURE 3.16

Solution of Eq. (3.7) for $m = k = 1$, $b = 0$, and $F(t) = 1$.

In conclusion, the solution of a LTI ODE is qualitatively dictated by the position of the eigenvalues of matrix A:

- A negative real valued eigenvalue produces a decaying term with a trajectory like those of Fig. 3.13.
- A positive real valued eigenvalue produces an exponentially growing term like that of Fig. 3.14.
- A pair of complex conjugated eigenvalues with negative real part produce damped oscillations like those of Fig. 3.15.
- A pair of pure imaginary conjugated eigenvalues produce undamped oscillations like those of Fig. 3.16.
- A pair of complex conjugated eigenvalues with positive real part produce oscillations whose amplitude exponentially grows with time.

The solution of a large system with several eigenvalues is ruled by the sum of the different modes. If all eigenvalues are negative (or have real negative part) then all the modes converge to zero and the solution is *asymptotically stable*. In there is one or more eigenvalues with positive real part then there are terms that diverge and the solution is *unstable*. Finally, the trajectories corresponding to eigenvalues with null real part are called *marginally stable*.

Non-Linear Models

While LTI system trajectories are limited to the sum of modes of the form $e^{\lambda_i \cdot t}$, in non-linear systems the solutions are far more complex. Moreover, in most cases the solutions cannot be expressed in terms of known functions.

However, in the vicinity of a given point of the solution trajectory, the behavior can be approximated by that of a LTI model. Thus, the solution of non-linear models can be thought as the concatenation of the solution of several LTI models. For this reason, many non-linear systems exhibit trajectories that resemble those of simple LTI models.

Nevertheless, there are several features of continuous systems that can only appear in non-linear models (chaotic trajectories, limit cycles, finite exit time, etc.).

While there exist some analysis tools to study qualitative properties of non-linear systems (Lyapunov theory is the preferred approach for these purposes), it is in most cases impossible to obtain quantitative information about the solutions. Thus, simulation is the only way to obtain the system trajectories.

3.2.2 CONTINUOUS SYSTEM SIMULATION

After describing the solutions of LTI systems, we are back to the problem of finding the solution of a general ODE:

$$\dot{\mathbf{z}}(t) = \mathbf{f}(\mathbf{z}(t), \mathbf{u}(t)) \tag{3.18}$$

with known initial state $\mathbf{z}(t_0) = \mathbf{z}_0$.

As we already mentioned, in most cases the exact solution cannot be obtained so we are forced to obtain numerical approximations for the values of the state at certain instants of time $t_0, t_1, \cdots, t_N$. These approximations are obtained using *numerical integration algorithms*, and traditionally the term *Continuous System Simulation* is tightly linked to those algorithms.

Nowadays, Continuous System Simulation is a wider topic. In fact, most continuous time models are not originally written like Eq. (3.18). That representation is the one used by numerical integration routines but, for a modeling practitioner, it is not comfortable (and it is sometimes impossible) to express a model in that way.

Modern continuous systems modeling languages like Modelica – the most widely accepted modeling standard language – allow representing the models in an object oriented fashion, reusing and connecting components from libraries. The resulting model is then an object oriented description that, after a flattening stage, results in a large set of differential and algebraic equations. Then, those *Differential Algebraic Equations* (DAEs) must be converted in an ODE like that of Eq. (3.18) in order to use numerical integration routines. This conversion requires different (and sometimes complex) algorithms and it constitutes itself an important branch of the Continuous System Simulation field (an extensive discussion is in Chapter 3 of TMS2000).

Many continuous time models describe the evolution of some magnitudes both in time and space. These models are expressed by *Partial Differential Equations* (PDEs) and their simulation constitute one of the most difficult problems in the discipline.

In this book we shall not cover the problems of DAE or PDE simulation. However, as both DAEs and PDEs can be converted or approximated by ODEs, the numerical algorithms for ordinary differential equations described next can still be applied for those problems.

3.2.3 EULER'S METHODS

The simplest method to solve Eq. (3.18) was proposed by Leonhard Euler en 1768. It is based on approximating the state derivative as follows

$$\dot{\mathbf{z}}(t) \approx \frac{\mathbf{z}(t+h)-\mathbf{z}(t)}{h} \frac{\mathbf{z}(t+h)-\mathbf{z}(t)}{h}$$
$$\Longrightarrow \mathbf{f}(\mathbf{z}(t),\mathbf{u}(t)) \approx \frac{\mathbf{z}(t+h)-\mathbf{z}(t)}{h}$$

so we obtain

$$\mathbf{z}(t+h) \approx \mathbf{z}(t) + h \cdot \mathbf{f}(\mathbf{z}(t),\mathbf{u}(t))$$

where h is the integration *step size*.

Defining $t_k = t_0 + k \cdot h$ for $k = 1, \cdots, N$, Forward Euler method is defined by the following formula:

$$\mathbf{z}(t_{k+1}) = \mathbf{z}(t_k) + h \cdot \mathbf{f}(\mathbf{z}(t_k),\mathbf{u}(t_k)) \tag{3.19}$$

Thus, starting from the known initial state $\mathbf{z}(t_0)$, Forward Euler method allows us to compute a numerical approximation for $\mathbf{z}(t_1)$. Then, using this value, an approximation for $\mathbf{z}(t_2)$ can be obtained as well as the successive values until reaching the final simulation time.

Using this algorithm, we simulated the spring–mass–damper model of Eq. (3.7) for parameters $b = k = m = 1$, input $F(t) = 1$, and null initial state values. We used a step size $h = 0.1$ and simulated until $t = t_f = 15$. The numerical results are depicted in Fig. 3.17 together with the analytical solution.

In spite of using a relatively small step size of $h = 0.1$, Forward Euler method gives a result with a relatively large error. We shall explain this problem later in this chapter.

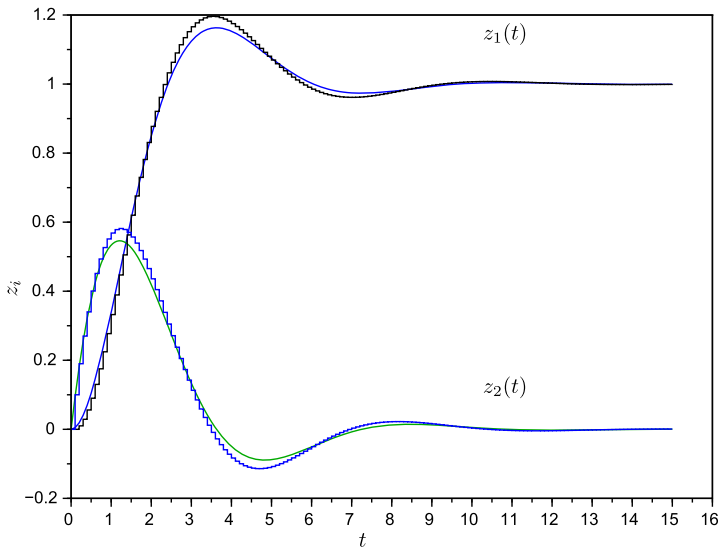

FIGURE 3.17

Forward Euler simulation of the spring–mass–damper model of Eq. (3.7). Numerical solution (stairs plot) vs. Analytical solution.

A variant of Forward Euler method is given by the following formula

$$\mathbf{z}(t_{k+1}) = \mathbf{z}(t_k) + h \cdot \mathbf{f}(\mathbf{z}(t_{k+1}), \mathbf{u}(t_{k+1})) \tag{3.20}$$

known as *Backward Euler* method. This is an *implicit formula*, where the unknown $\mathbf{z}(t_{k+1})$ is at both sides of the equation. Taking into account that function $\mathbf{f}()$ is usually non-linear, obtaining the next state value with this method requires solving a non-linear algebraic equation. This is usually accomplished making use of Newton's iterations.

Using this algorithm, we repeated the simulation of the spring–mass–damper system, obtaining the results depicted in Fig. 3.18.

The results are not significantly different from those obtained using Forward Euler in Fig. 3.17. The implicit nature of the algorithm and the iterations needed to find the next state value at each simulation step make this method far more expensive than its explicit counterpart. However, the comparison of the results suggest that we have gained nothing out of those additional computational costs.

The only difference between Figs. 3.17 and 3.18 is that the numerical solution shows larger oscillations using Forward Euler than using Backward Euler. We will see later that this small difference is connected to the main justifications for the use of implicit algorithms.

3.2.4 ACCURACY OF THE APPROXIMATIONS

Fig. 3.19 shows the results of using Forward Euler with different step sizes. Notice how the error grows with the value of h.

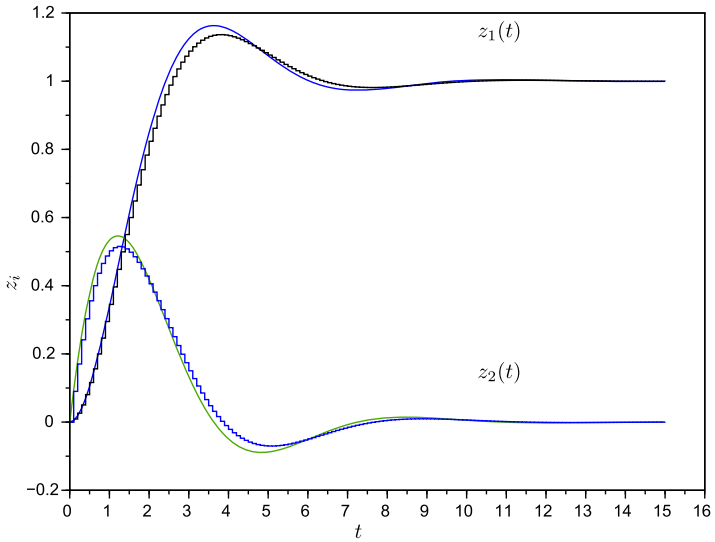

FIGURE 3.18

Backward Euler simulation of the spring–mass–damper model of Eq. (3.7). Numerical solution (stairs plot) vs. Analytical solution.

To confirm our intuition, we will analyze the approximation performed by Forward Euler method in order to find a formal proof that, as expected and observed, the error grows with the step size.

Let us assume that we know the state vector $\mathbf{z}$ at time t_k. Then, we can express the exact solution of the ODE $\dot{\mathbf{z}}(t) = \mathbf{f}(\mathbf{z}(t), \mathbf{u}(t))$ at time t_{k+1} using the following Taylor series

$$\mathbf{z}(t_{k+1}) = \mathbf{z}(t_{k+1}) + h \cdot \frac{\mathrm{d}\mathbf{z}}{\mathrm{d}t}(t_k) + \frac{h^2}{2!} \cdot \frac{\mathrm{d}^2\mathbf{z}}{\mathrm{d}t^2}(t_k) + \frac{h^3}{3!} \cdot \frac{\mathrm{d}^3\mathbf{z}}{\mathrm{d}t^3}(t_k) + \cdots$$

Replacing the state time derivative by function $\mathbf{f}$, we obtain

$$\mathbf{z}(t_{k+1}) = \mathbf{z}(t_{k+1}) + h \cdot \mathbf{f}(\mathbf{z}(t_k), t_k) + \frac{h^2}{2!} \cdot \frac{\mathrm{d}^2\mathbf{z}}{\mathrm{d}t^2}(t_k) + \frac{h^3}{3!} \cdot \frac{\mathrm{d}^3\mathbf{z}}{\mathrm{d}t^3}(t_k) + \cdots \tag{3.21}$$

that shows that Forward Euler method *truncates* the Taylor series of the analytical solution after the first term. For this reason, Forward Euler is a *first order* accurate algorithm.

The truncated terms form a series that start with the term of h^2. Thus, according to Taylor's Theorem, the remainder of the series can be approximated by some function of the state multiplied by h^2. This means that the error grows with h^2 and confirms our initial intuition that the larger the step size, the larger the error.

This error is called *Local Truncation Error*, as it is the difference observed between the analytical and the numerical solution after one step. In general, this error is proportional to h^{p+1} where p is the order of the method.

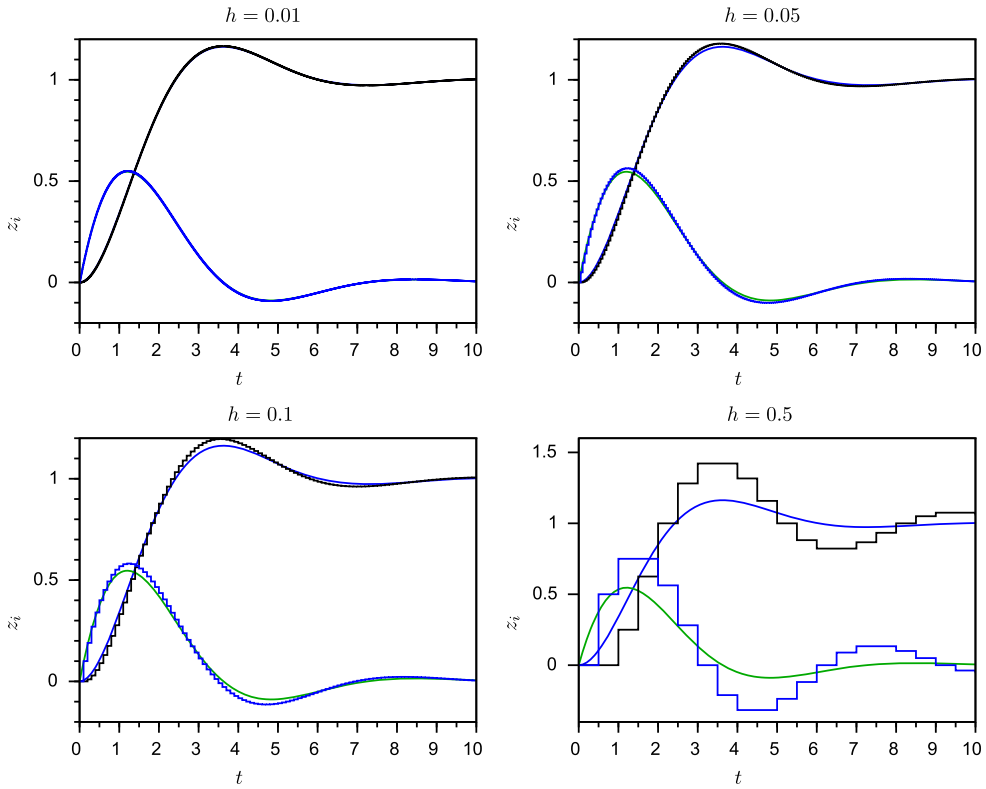

FIGURE 3.19

Accuracy of Forward Euler for different step sizes.

However, note that except for the first step, we do not know the analytical solution for $\mathbf{z}(t_k)$. For this reason, at time t_{k+1} we will have an additional error as we are starting from a non-exact value at time t_k. Thus the difference observed between the analytical and the numerical solution at any given instant of time is not the local truncation error. What is observed is called *Global Error* and it is the result of the accumulating local errors along the simulation run.

It can be proved that, under certain circumstances, the maximum global error has one order less than the local error. This is, a method of order p has a local error proportional to h^{p+1} but a global error proportional to h^p.

Forward and Backward Euler are both first order accurate methods, so their global errors are just proportional with h. Thus, if we reduce the step size h by a factor of 10,000, the error will also be reduced by the same factor. Posing the problem in the opposite direction, if we are not happy with the error obtained in a simulation and we want to obtain a result 10,000 times more accurate, we need to reduce the step size h by that factor, which implies performing 10,000 times more calculations.

If we use a fourth order method instead and we want to obtain a result 10,000 times more accurate, it is sufficient to reduce 10 times the step size h, which implies only 10 times more calculations. This

last observation leads to an important conclusion: if we want to obtain very accurate results, we must use high order methods. Otherwise, we will be forced to use very small step sizes.

We will see later that higher order methods have a larger cost per step than Forward Euler. However, those higher costs are compensated by the capability of performing much larger steps with better accuracy.

3.2.5 CONVERGENCE OF THE NUMERICAL SCHEME

The fact that the global error is proportional to the step size h in Euler method (or to h^p in a p-th order method) also implies that the global error goes to zero as the step size h goes to zero. This property is called *convergence*. A sufficient condition to ensure convergence using a p-th order numerical approximation is that function $\mathbf{f}(\mathbf{z}, \mathbf{u})$ in Eq. (3.18) satisfies a local *Lipschitz* condition on $\mathbf{z}$. This is, given any $\delta > 0$, $\mathbf{z}_0 \in \mathbb{R}^n$, and $\mathbf{u} \in \mathbb{R}^m$, there exists $\lambda > 0$ such that for any pair $\mathbf{z}_a, \mathbf{z}_b \in \mathbb{R}$ verifying

$$\|\mathbf{z}_a - \mathbf{z}_0\| < \delta, \quad \|\mathbf{z}_b - \mathbf{z}_0\| < \delta \tag{3.22}$$

it results

$$\|\mathbf{f}(\mathbf{z}_a, \mathbf{u}) - \mathbf{f}(\mathbf{z}_b, \mathbf{u})\| \leq \lambda \cdot \|\mathbf{z}_a - \mathbf{z}_b\| \tag{3.23}$$

This property says that in a given region around some point $\mathbf{z}_0$, function $\mathbf{f}$ has a limited variation with $\mathbf{z}$. Lipschitz condition is stronger than continuity but is weaker than differentiability.

A simplified version of a convergence Theorem that will be useful for results in Chapter 20 is given below.

Theorem 3.1. *Let $\mathbf{z}_{\boldsymbol{a}}(t)$ be the analytical solution of the ODE*

$$\dot{\mathbf{z}}_{\boldsymbol{a}}(t) = \mathbf{f}(\mathbf{z}_{\boldsymbol{a}}(t), \mathbf{u}(t)) \tag{3.24}$$

for $t \in [t_0, t_f]$, and consider the numerical solution

$$\mathbf{z}(t_{k+1}) = \mathbf{z}(t_k) + h \cdot \mathbf{f}(\mathbf{z}(t_k), \mathbf{u}(t_k)) + h^2 \cdot \gamma(\mathbf{z}(t_k), \mathbf{u}(t_k), h) \tag{3.25}$$

where $\gamma(\mathbf{z}(t_k), \mathbf{u}(t_k), h)$ represents the second order portion of the solution. Assume that $\mathbf{z}_{\boldsymbol{a}}(t) \in D$ in that interval, where $D \subset \mathbb{R}^n$ is a compact set, and that $\mathbf{u}(t) \in U$ where $U \subset \mathbb{R}^n$ is also a compact set. Suppose that $\mathbf{f}(\mathbf{z}_{\boldsymbol{a}}, \mathbf{u})$ is Lipschitz in D for all $\mathbf{u} \in U$.

Suppose also that

$$\|\gamma(\mathbf{z}(t_k), \mathbf{u}(t_k), h)\| \leq \bar{\gamma} \tag{3.26}$$

(This bounds the effect of the state and input on the solution.) Then, the numerical solution $\mathbf{z}(t_k)$ obtained with $\mathbf{z}(t_0) = \mathbf{z}_{\boldsymbol{a}}(t_0)$ converges to the analytical solution $\mathbf{z}_{\boldsymbol{a}}(t_k)$ as h goes to 0 for $t \in [t_0, t_f]$.

Proof. From the Taylor series we know that

$$\mathbf{z}_{\mathbf{a}}(t_{k+1}) = \mathbf{z}_{\mathbf{a}}(t_k) + h \cdot \mathbf{f}(\mathbf{z}_{\mathbf{a}}(t_k), \mathbf{u}(t_k)) + h^2 \cdot \mathbf{d}(t_k) \tag{3.27}$$

where the last term expresses the remainder of the series. The fact that $\mathbf{z_a}(t)$ is bounded in $t \in [t_0, t_f]$ implies that $\mathbf{d}(t_k)$ is also bounded, i.e.,

$$\|\mathbf{d}(t)\| \leq \bar{d} \tag{3.28}$$

for all $t \in [t_0, t_f]$.

The error at the first step is then, from Eqs. (3.25) and (3.27),

$$\|\mathbf{z}(t_1) - \mathbf{z_a}(t_1)\| = \|h^2 \cdot \gamma(\mathbf{z}(t_0), \mathbf{u}(t_0), h) - h^2 \cdot \mathbf{d}(t_0)\| \leq (\bar{\gamma} + \bar{d}) \cdot h^2 = c \cdot h^2 \tag{3.29}$$

where the last inequality comes from Eqs. (3.26) and (3.28) with $c \triangleq \bar{d} + \bar{\gamma}$.

In the $k+1$-th step, we have,

$$\begin{aligned}
&\|\mathbf{z}(t_{k+1}) - \mathbf{z_a}(t_{k+1})\| = \\
&= \|\mathbf{z}(t_k) + h \cdot \mathbf{f}(\mathbf{z}(t_k), \mathbf{u}(t_k)) + h^2 \cdot \gamma(\mathbf{z}(t_k), \mathbf{u}(t_k), h) - (\mathbf{z_a}(t_k) + h \cdot \mathbf{f}(\mathbf{z_a}(t_k), \mathbf{u}(t_k)) + h^2 \cdot \mathbf{d}(t_k))\| \\
&\leq \|\mathbf{z}(t_k) - \mathbf{z_a}(t_k)\| + h \cdot \|\mathbf{f}(\mathbf{z}(t_k), \mathbf{u}(t_k)) - \mathbf{f}(\mathbf{z_a}(t_k), \mathbf{u}(t_k))\| + c \cdot h^2 \\
&\leq \|\mathbf{z}(t_k) - \mathbf{z_a}(t_k)\| + h \cdot L \cdot \|\mathbf{z}(t_k) - \mathbf{z_a}(t_k)\| + c \cdot h^2
\end{aligned}$$

where we used the Lipschitz condition in the last step. Then,

$$\|\mathbf{z}(t_{k+1}) - \mathbf{z_a}(t_{k+1})\| \leq (1 + h \cdot L) \cdot \|\mathbf{z}(t_k) - \mathbf{z_a}(t_k)\| + c \cdot h^2 \tag{3.30}$$

Recalling that at the first step the error, according to Eq. (3.29), is bounded by $c \cdot h^2$, the recursive use of Eq. (3.30) leads to

$$\|\mathbf{z}(t_N) - \mathbf{z_a}(t_N)\| \leq c \cdot h^2 \cdot \sum_{k=1}^{N} (1 + h \cdot L)^k \tag{3.31}$$

Then, at a fixed time $t = t_N$, such that $h = t_N/N$, the last expression becomes

$$\|\mathbf{z}(t_N) - \mathbf{z_a}(t_N)\| \leq c \cdot \frac{t_N^2}{N^2} \cdot \sum_{k=1}^{N} \left(1 + \frac{(t_N \cdot L)}{N}\right)^k \tag{3.32}$$

Using the fact that

$$\lim_{N \to \infty} \left(1 + \frac{(t_N \cdot L)}{N}\right) = e^{t_N \cdot L} \tag{3.33}$$

then the sum in Eq. (3.32) is bounded by $N \cdot e^{t_N \cdot L}$, so the whole expression in Eq. (3.32) goes to 0 as $N \to \infty$, i.e., as $h \to 0$. □

This Theorem is valid for any numerical method that can be written in the form of Eq. (3.25). It then holds for Forward Euler (where $\gamma() = 0$) but also for Backward Euler and for higher order methods with a proper definition of function γ. Later we will see how discontinuities in the ODE violate the conditions required for Theorem 3.1 and require event-based control of solutions.

3.2.6 NUMERICAL STABILITY

A question that comes up after analyzing the difference between the local and global error is: does the latter keeps growing as the simulation advances?

Looking back to Fig. 3.19 we see that as the analytical solution approaches the equilibrium point, the numerical solution goes to the same value. Evidently, in this case, the global error tends to disappear after a certain time. In other words, what we see in Fig. 3.19 is that the numerical solution preserves the *stability* of the analytical solution. Here we say that a numerical solution preserves the stability when it does not diverge away from the analytical solution, provided that the latter is bounded.

Does this occur for any value of the step size h? Fig. 3.20 shows the trajectories (here we only draw the first component $z_1(t)$) obtained as the step size h becomes larger.

Evidently, the stability is not always preserved. For $h = 1$ the asymptotic stability is lost and we have a *marginally stable* numerical solution. For $h = 1.5$ the numerical solution is definitively unstable.

Preserving the numerical stability is a crucial problem in continuous system simulation, and, as we see, in the case of Forward Euler it strongly depends on the correct choice of the step size h.

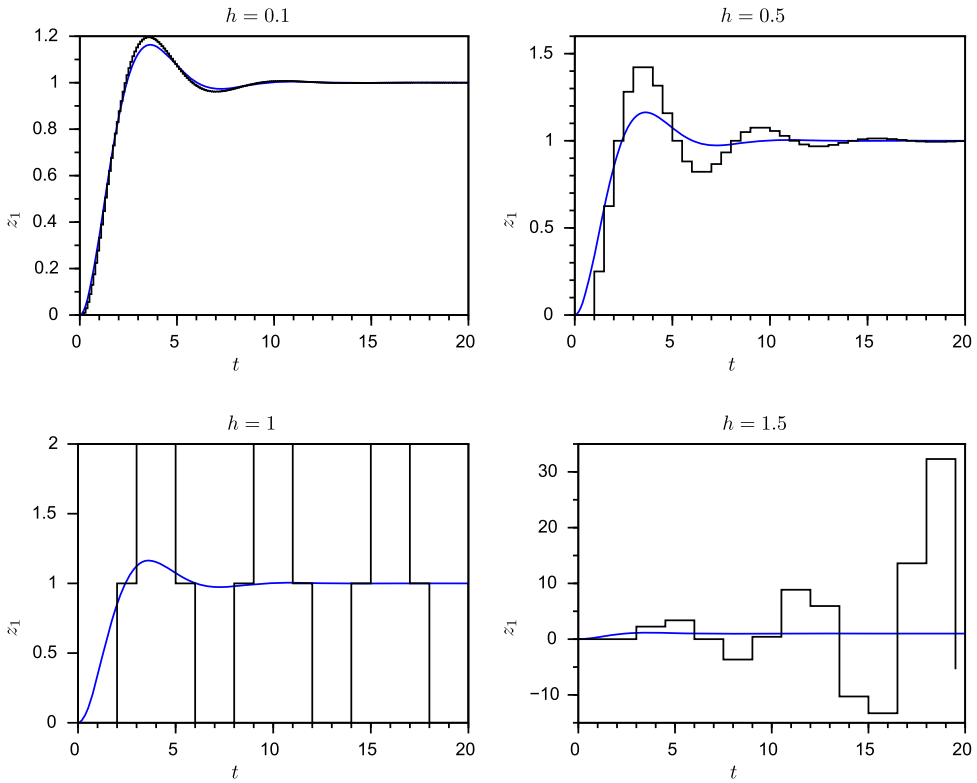

FIGURE 3.20

Stability of Forward Euler for different step sizes.

Let us see now what happens if we use Backward Euler instead of Forward Euler. Fig. 3.21 shows the results of repeating the experiments with the implicit method.

This time, the stability is always preserved. This feature is the main reason for using implicit algorithms. We will see later that for certain classes of systems, only methods that preserve stability like Backward Euler can be used.

In this empirical study on stability we observed two things. First, that Forward Euler becomes unstable when we increase h beyond certain value. Second, that Backward Euler does not seem to become unstable as h grows.

These observations can be easily justified analyzing the Forward and Backward Euler approximation of LTI systems. Given a LTI system

$$\dot{\mathbf{z}}(t) = A \cdot \mathbf{z}(t) + B \cdot \mathbf{u}(t)$$

Forward Euler approximates it by a LTI discrete time of the form

$$\mathbf{z}(t_{k+1}) = A_D \cdot \mathbf{z}(t_k) + B_F \cdot \mathbf{u}(t_k) \tag{3.34}$$

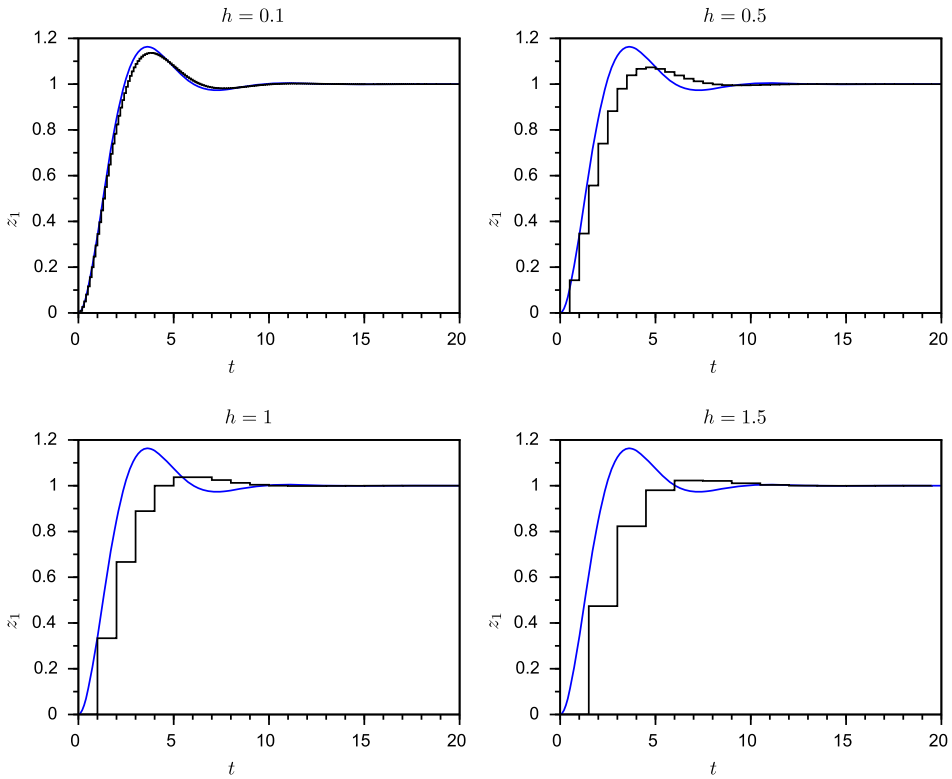

FIGURE 3.21

Stability of Backward Euler for different step sizes.

Let λ_i be the i-th eigenvalue of A. Then, the i-th eigenvalue of matrix A_D is $\lambda_i^D = \lambda_i \cdot h + 1$. The condition for the discrete time LTI system of Eq. (3.34) to be stable is that $|\lambda_i^D| < 1$ for all i. If the continuous time system is stable, then we know that $\mathbb{R}\text{e}(\lambda_i) < 0$. Then, we see that

$$|\lambda_i^D| = |\lambda_i \cdot h + 1| < 1 \tag{3.35}$$

only when h is small. If the product $h \cdot \lambda_i$ becomes too large in absolute value, then the expression $|\lambda_i \cdot h + 1|$ becomes larger than 1 and the discrete time model becomes unstable.

In contrast, Backward Euler approximation produces a similar approximation to that of Eq. (3.34), but the eigenvalues are now

$$\lambda_i^D = \frac{1}{1 - \lambda_i \cdot h} \tag{3.36}$$

When $\mathbb{R}\text{e}(\lambda_i) < 0$ (i.e., when the continuous system is stable) then it can be easily seen that $|\lambda_i^D| < 1$ for all $h > 0$. Thus, the numerical stability is always preserved.

3.2.7 ONE-STEP METHODS

We mentioned above the importance of using higher order methods, as they allow us to simulate with high accuracy using large step sizes. However, so far, we have only introduced two first order accurate algorithms.

In order to obtain a higher order approximation in the Taylor's series of Eq. (3.21), we need to approximate the derivatives of function $\mathbf{f}()$, which requires knowing the value of $\mathbf{f}()$ at more than one point at each step.

When the different values of function $\mathbf{f}()$ used to compute $\mathbf{z}(t_{k+1})$ are obtained only making use of the value of $\mathbf{z}(t_k)$, the resulting algorithms are called *One-Step Methods*. In contrast, when we use information of previous values of the state ($\mathbf{z}(t_{k-1})$, etc.) the algorithms are called *Multi-Step*.

One-Step methods are usually called *Runge–Kutta* (RK) methods because the first of these algorithms was proposed by Runge and Kutta at the end of the 19th century.

Explicit Runge–Kutta Methods

Explicit Runge–Kutta methods perform several evaluation of function $\mathbf{f}()$ around the point $(\mathbf{z}(t_k), t_k)$ and then they compute $\mathbf{z}(t_{k+1})$ using a weighted average of those values.

One of the simplest algorithms is the *Heun's Method*, based in the following formulation:

$$\mathbf{z}(t_{k+1}) = \mathbf{z}(t_k) + \frac{h}{2} \cdot (\mathbf{k}_1 + \mathbf{k}_2)$$

where

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{f}(\mathbf{z}(t_k), t_k) \\ \mathbf{k}_2 &= \mathbf{f}(\mathbf{z}(t_k) + h \cdot \mathbf{k}_1, t_k + h) \end{aligned}$$

Compared with Euler's, this method performs an extra evaluation of $\mathbf{f}()$ in order to compute $\mathbf{k}_2$. Notice that this is an estimate of $\dot{\mathbf{z}}(t_{k+1})$, but it is not used back in the next step. This price is compen-

Table 3.7 Maximum global error committed by Euler, Heun and RK4 methods

Step Size	Euler's Error	Heun's Error	RK4 Error
$h = 0.5$	0.298	0.0406	4.8×10^{-4}
$h = 0.1$	0.042	1.47×10^{-3}	6.72×10^{-7}
$h = 0.05$	0.0203	3.6×10^{-4}	4.14×10^{-8}
$h = 0.01$	3.94×10^{-3}	1.42×10^{-5}	6.54×10^{-11}

sated by the fact that Heun's performs a second order approximation obtaining more accurate results than Euler's.

One of the most used numerical integration algorithms is the fourth order Runge–Kutta (RK4):

$$\mathbf{z}(t_{k+1}) = \mathbf{z}(t_k) + \frac{h}{6} \cdot (\mathbf{k}_1 + 2 \cdot \mathbf{k}_2 + 2 \cdot \mathbf{k}_3 + \mathbf{k}_4)$$

where

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}(\mathbf{z}(t_k), t_k) \\
\mathbf{k}_2 &= \mathbf{f}(\mathbf{z}(t_k) + h \cdot \frac{\mathbf{k}_1}{2}, t_k + \frac{h}{2}) \\
\mathbf{k}_3 &= \mathbf{f}(\mathbf{z}(t_k) + h \cdot \frac{\mathbf{k}_2}{2}, t_k + \frac{h}{2}) \\
\mathbf{k}_4 &= \mathbf{f}(\mathbf{z}(t_k) + h \cdot \mathbf{k}_3, t_k + h)
\end{aligned}$$

This algorithm uses four evaluations of function $\mathbf{f}()$ at each step, obtaining a fourth order approximation.

Table 3.7 synthesizes the results of simulating the spring–mass–damper system of Eq. (3.7) using Euler, Heun and RK4 methods. There, the maximum global error is reported for each step size and for each method.

Notice that Euler's error is linearly reduced with the step size, while Heun's is quadratically reduced and RK4 is quartically reduced.

The literature on numerical integration contains hundreds of explicit Runge Kutta algorithms, including methods of more than 10th order. In practice, the most used RK methods are those of order between 1 and 5.

Regarding stability, explicit Runge–Kutta methods have similar features to those of Forward Euler. This is, they preserve stability provided that the step size h does not become too large. Thus, in practice, the use of high order RK methods allows us to increase the step size while still obtaining good accuracy but the stability of the algorithms establishes limits to the value of h.

Implicit One-Step Methods

We saw that Backward Euler was an implicit algorithm that preserves the numerical stability for any step size h. However, just like Forward Euler, it is only first order accurate.

There are several implicit One-Step methods of higher order that, like Backward Euler, preserve the stability. Among them, a widely used algorithm is the *Trapezoidal Rule* defined as follows:

$$\mathbf{z}(t_{k+1}) = \mathbf{z}(t_k) + \frac{h}{2} \cdot [\mathbf{f}(\mathbf{z}(t_k), t_k) + \mathbf{f}(\mathbf{z}(t_{k+1}), t_{k+1})] \tag{3.37}$$

This implicit method is second order accurate and it not only preserves stability but also marginal stability. This is, when a system with undamped oscillations is simulated using this algorithm, the numerical result also exhibit undamped oscillations.

Except for the trapezoid rule and two or three other algorithms, one-step implicit methods are not widely used in practice since multi-step implicit methods are normally more efficient.

3.2.8 MULTI-STEP METHODS

Multi-Step numerical algorithms use information from past steps to obtain higher order approximations of $\mathbf{z}(t_{k+1})$. Using *past information*, these methods do not need to recompute several times the function $\mathbf{f}()$ at each step as One-Step methods do. In consequence, Multi-Step algorithms usually have a lower cost per step.

Multi-Step Explicit Methods

The most used explicit Multi-Step methods are those of Adams–Bashforth (AB) and Adams–Bashforth–Moulton (ABM).

The second order accurate Adams–Bashforth method (AB2) is defined by the formula:

$$\mathbf{z}(t_{k+1}) = \mathbf{z}(t_k) + \frac{h}{2} \cdot (3\mathbf{f}_k - \mathbf{f}_{k-1}) \tag{3.38}$$

where we defined

$$\mathbf{f}_k \triangleq \mathbf{f}(\mathbf{z}(t_k), t_k)$$

The fourth order AB method (AB4), in turn, is defined as follows:

$$\mathbf{z}(t_{k+1}) = \mathbf{z}(t_k) + \frac{h}{24} (55\mathbf{f}_k - 59\mathbf{f}_{k-1} + 37\mathbf{f}_{k-2} - 9\mathbf{f}_{k-3}) \tag{3.39}$$

Notice that AB methods require a single evaluation of function $\mathbf{f}()$ at each step. The higher order approximation, as we already mentioned, is obtained using past values of function $\mathbf{f}()$.

One of the drawbacks of multi-step methods is related to the startup. The expression of Eq. (3.38) cannot be used to compute $\mathbf{z}(t_1)$ as it would require the value of $\mathbf{f}_{-1}$. Similarly, Eq. (3.39) can be only used to compute $\mathbf{z}(t_k)$ with $k \geq 4$. For this reason, the first steps of Multi-Step methods must be computed using some One-Step algorithm (e.g., Runge–Kutta).

Another problem of AB methods is that of stability. Like RK algorithms, AB only preserve numerical stability provided that h is small. However, the step size must be even smaller in AB than in RK. That feature is partially improved by Adams–Bashforth–Moulton methods, that have better stability features than AB at the price of performing two function evaluations per step.

Implicit Multi-Step Methods

As in One-Step methods, implicit approaches must be used in multi-step methods to obtain numerical solutions that preserve stability irrespective of the step size h.

The most used implicit multi-step methods are the family of *Backward Difference Formulae* (BDF). For instance, the third order BDF method (BDF3) is defined as follows:

$$\mathbf{z}(t_{k+1}) = \frac{18}{11}\mathbf{z}(t_k) - \frac{9}{11}\mathbf{z}(t_{k-1}) + \frac{2}{11}\mathbf{z}(t_{k-2}) + \frac{6}{11} \cdot h \cdot \mathbf{f}(t_{k+1}) \tag{3.40}$$

This algorithm has almost the same computational cost as Backward Euler, since both methods must solve a similar implicit equation. However, using three past state values, BDF3 is third order accurate.

There are BDF methods up to order 8, while the most used are those of order 1 (Backward Euler) to order 5.

3.2.9 STEP SIZE CONTROL

Up to here we always considered that the step size h was a fixed parameter that must be selected before starting the simulation. However, it is very simple to implement algorithms that automatically change the step size as the simulation advances. These *step size control* algorithms have the purpose of keeping a bounded simulation error. For that goal, they increase or decrease h according to the estimated error.

The use of step size control routines leads to *Variable Step Size Methods*, that are used in most modern continuous simulation tools.

One-Step Methods

Variable step size RK methods work as follows:

1. Compute $\mathbf{z}(t_{k+1})$ with certain step size h using the selected RK method.
2. Estimate the error in $\mathbf{z}(t_{k+1})$.
3. If the error is larger than the tolerance, reduce the step size h and recompute $\mathbf{z}(t_{k+1})$ going back to step 1.
4. Otherwise, accept the value $\mathbf{z}(t_{k+1})$, increase the step size h and go back to step 1 to compute $\mathbf{z}(t_{k+2})$.

While the idea is simple, the following questions come up from the procedure above:

- How is the error estimated?
- What should be the rule used to reduce or to increase the step size h?
- What value should be used as the initial value of h at the first step?

We answer those questions below:

Error estimation: The error is estimated using two methods of different order. Given $\mathbf{z}(t_k)$, the value of $\mathbf{z}(t_{k+1})$ can be computed using a fourth order method and then computed again with a fifth order method. Because the fifth order algorithm is far more accurate than the fourth order one, the difference between both results is approximately the error of the fourth order method.

In order to save calculations, RK algorithms that share some stages are commonly employed. One of the most used variable step size RK algorithms is that of Dormand–Prince (DOPRI), that

performs a total of six evaluations of function $\mathbf{f}()$ and uses them to compute a fourth and a fifth order approximation of $\mathbf{z}(t_{k+1})$.

Step Size Adjustment: Let us suppose that we used a p-th and a $p+1$-th order method to compute $\mathbf{z}(t_{k+1})$ with a step size h. Suppose that the difference between both approximations was

$$\|\mathbf{z}(t_{k+1} - \tilde{\mathbf{z}}(t_{k+1})\| = err(h)$$

Assume also that we wanted that this error was equal to the tolerance tol. The problem is then to compute the step size h_0 so that $err(h_0) = tol$.

For this goal, we recall that in a method of order p, the Taylor series expansion coincides with that of the analytical solution up to the term of h^p. For that reason, the difference between both approximations is proportional to h^{p+1}, i.e.,

$$err(h) = c \cdot h^{p+1} \tag{3.41}$$

where according to Taylor's Theorem, c is a constant depending on $\mathbf{z}(t_k)$. If we had used the right step size h_0 (still unknown), we would have obtained

$$err(h_0) = c \cdot h_0^{p+1} = tol \tag{3.42}$$

Then, dividing Eq. (3.41) and Eq. (3.42) we obtain

$$h_0 = h \cdot \sqrt[p+1]{\frac{tol}{err(h)}}$$

This expression allows us to adjust the step size according to the error obtained and the prescribed tolerance. In most variable step methods, the step size adjustment is made in a more conservative way, replacing the expression by

$$h_0 = C \cdot h \cdot \sqrt[p+1]{\frac{tol}{err(h)}} \tag{3.43}$$

for a (different constant) $C < 1$ (a typical value is $C = 0.8$). That way, the error is almost always kept below the tolerance.

Initial Step Size: Initially, most variable step size algorithms use a very small step size h. In this way, they obtain a valid value for $\mathbf{z}(t_1)$ and they can start increasing the step size with Eq. (3.43).

Fig. 3.22 shows the simulation result for the spring–mass–damper system of Eq. (3.7) using the variable step Runge–Kutta–Fehlberg algorithm, a method very similar to DOPRI that uses a fourth and a five order RK approximation with a total of six function evaluations per step.

The simulation was performed using a *relative error tolerance* $tol_{rel} = 10^{-3}$, i.e., the error tolerance was set to change with the signal values ($tol = 0.001 \cdot \|\vec{(t)}\|$). The algorithm completed the simulation after only 18 steps, and the difference between the analytical and the numerical solution at the time steps cannot be appreciated in Fig. 3.22.

The size of the steps is depicted in Fig. 3.23. There, we can see that the step size control algorithm slowly increases the step size as the trajectories approach the equilibrium.

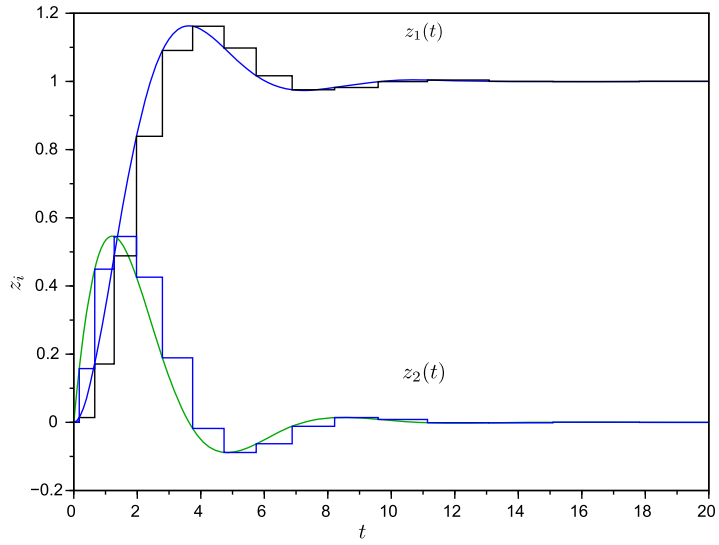

FIGURE 3.22

RKF45 simulation of the spring–mass–damper model of Eq. (3.7). Numerical solution (stairs plot) vs. Analytical solution.

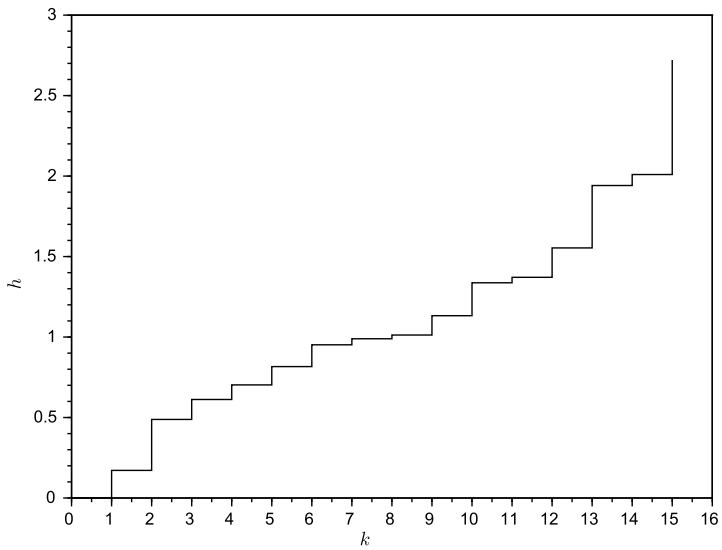

FIGURE 3.23

RKF45 step size evolution in the simulation of the spring–mass–damper model of Eq. (3.7).

3.2.9.1 Multi-Step Methods

In Multi-Step methods the step size can be also controlled in a similar manner to that of RK algorithms. However, the Multi-Step formulas like those of Eq. (3.38), (3.39), and (3.40) are only valid for a constant step size h. Because they use information from past steps, they assume that the information is equally spaced in time. Thus, in order to change the step size, we must first interpolate the required past information according to the new step size h. In consequence, changing the step size has an additional cost here.

For this reason, step size control algorithms for multi-step methods do not increase the step size at every step. They do it only when justified, i.e., when the difference $|h - h_0|$ is sufficiently large relative to the cost of additional calculations.

One of the most used variable step multi-step algorithms is that of DASSL, an implicit solver based on BDF formulas.

3.2.10 STIFF, MARGINALLY STABLE AND DISCONTINUOUS SYSTEMS

There are some continuous time system types that pose some difficulties to the numerical integration algorithms. Among them, we account for *stiff systems*, *marginally stable systems*, and systems with *discontinuities*.

Next, we start analyzing those problematic cases.

Stiff Systems

Let us consider once more the spring–mass–damper model of Eq. (3.7). Consider now the parameters $m = k = 1$, $b = 100.01$ and $F(t) = 1$. Using these parameter values, matrix A becomes

$$A = \begin{bmatrix} 0 & 1 \\ -k/m & -b/m \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & -100.01 \end{bmatrix}$$

whose eigenvalues are $\lambda_1 = -0.01$ and $\lambda_2 = -100$. Then, the analytical solutions will contain two modes: one of the form $e^{-0.01 \cdot t}$ exhibiting a slow evolution towards 0, and another term of the form $e^{-100 \cdot t}$ having a fast evolution towards 0.

Systems with simultaneous slow and fast dynamics like this one are called *stiff*. Fig. 3.24 shows the trajectory of the component $z_2(t)$. The complete trajectory in the left shows the slow dynamics, while the startup detail in the right exhibits the fast dynamics. The difference in speed is so big that the rise of the trajectory looked like an instantaneous change in the value of $z_2(t)$ in the figure of the left.

If we want to simulate this system in an efficient way, it is clear that we need a variable step size algorithm that starts with a small step size to capture the fast dynamics and then increases h at some point once the fast dynamics have been handled.

Applying the previous Runge–Kutta–Fehlberg algorithm, the simulation took a total of 13,603 steps. Although the results were accurate, the number of steps seems too large.

Fig. 3.25 shows the evolution of the step size during the first 200 steps of this simulation. As expected, at the beginning of the simulation the algorithm used a small step size that captured the fast dynamics. However, then it failed to increase the step size beyond $h \approx 0.04$.

The problem is related to stability. For example, using Forward Euler, in order to preserve numerical stability, we must use a value of h so that Eq. (3.35) is true for all eigenvalues. In particular, for the fast

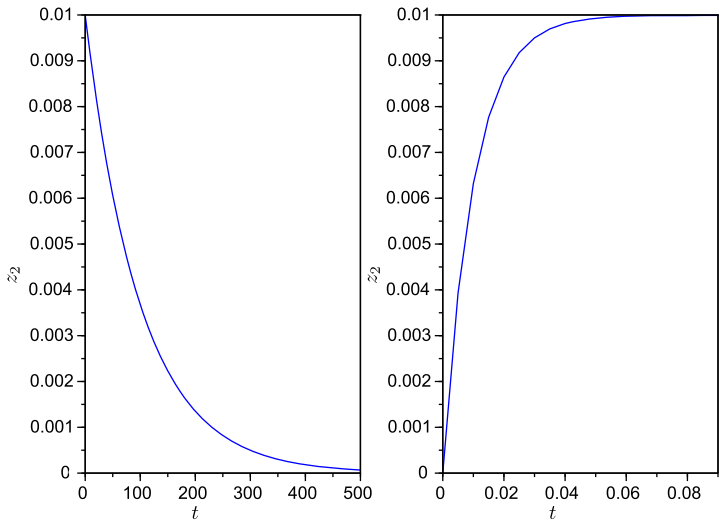

FIGURE 3.24

Trajectory of $z_2(t)$ for $b = 100$ (stiff case) in Eq. (3.7). Complete transient (left) and startup (right).

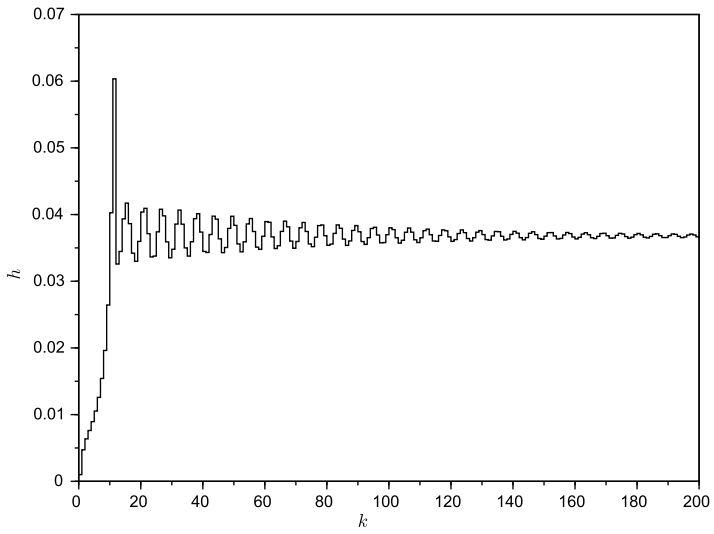

FIGURE 3.25

RKF45 step size evolution in the simulation of the stiff spring–mass–damper model of Eq. (3.7) with $b = 100.01$.

eigenvalue $\lambda_2 = -100$ we need that $|1 + h \cdot (-100)| < 1$, which is accomplished only if $h < 0.02$. For this reason, Forward Euler cannot employ a step size $h > 0.02$ without obtaining unstable results.

RKF45 uses explicit RK algorithms whose stability conditions are similar to that of Forward Euler. In this case, the stability limit is around $h \approx 0.038$. Whenever the step size becomes larger than this value, the solution becomes unstable and the error grows beyond the tolerance. Thus, the step size control algorithm is forced to reduce the step size, that remains around the stability limit.

How can we overcome this problem?

Evidently, we need an algorithm that does not becomes unstable as h is increased. We already know that only implicit algorithms like Backward Euler can have this property.

Thus, we simulated again the system using a variable step size version of the implicit multi-step BDF4 algorithm (we called it BDF45). This time, the simulation was completed after only 88 steps. Fig. 3.26 shows that, as expected, the step size h grows without being limited by the numerical stability.

So we see that stiff systems force the use of implicit methods. Indeed, they are the main reason for the existence of those numerical methods. The problem with implicit solvers is that they have expensive steps specially when the systems are large as they must solve a large system of algebraic equations in each step.

In Chapter 19 we shall see a new family of numerical ODE solvers based on quantization principles that allow the explicit simulation of some stiff systems, while still having good stability and computation properties.

Marginally Stable Systems

We had already derived the analytical solution of the spring–mass–damper system without friction (i.e., with $b = 0$), consisting in the undamped oscillations shown in Fig. 3.16.

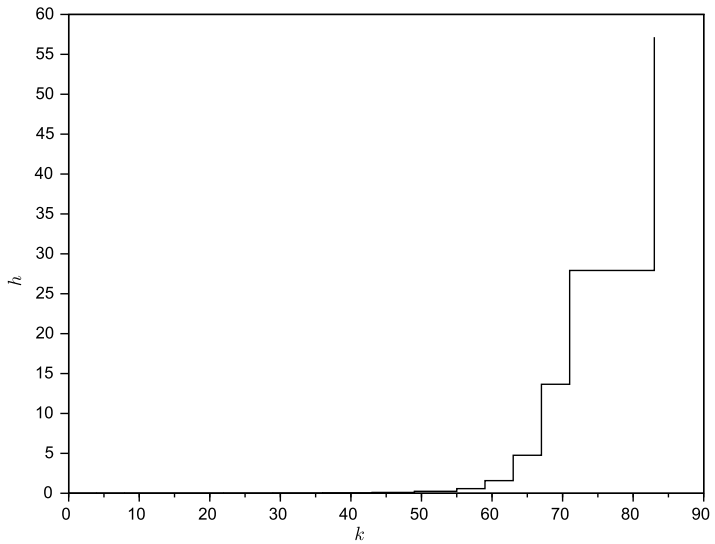

FIGURE 3.26

BDF45 step size evolution in the simulation of the stiff spring–mass–damper model of Eq. (3.7) with $b = 100.01$.

Systems containing undamped oscillations are called *Marginally Stable* and they can only be accurately simulated by some special methods that preserve the marginal stability in the numerical solution.

Fig. 3.27 shows the result of simulating this system using Forward and Backward Euler methods. The explicit Forward Euler method produces unstable oscillations that diverge in amplitude. In contrast, the implicit Backward Euler produces damped oscillations that tend to disappear. None of the algorithms preserve the marginal stability.

This fact can be easily explained from the stability conditions of Eq. (3.35) in Forward Euler and Eq. (3.36) in Backward Euler. In the first case, the moduli of the discrete eigenvalues are

$$|\lambda_i^D| = |\pm i \cdot h + 1|$$

which are always greater than 1 and is therefore unstable.

In the second case, the moduli of the discrete eigenvalues are

$$|\lambda_i^D| = \frac{1}{|1 \pm i \cdot h|}$$

that are always less than 1 so the numerical solution is always asymptotically stable.

We mentioned before that there is a method called the *Trapezoidal Rule*, defined in Eq. (3.37), that preserves the marginal stability. Fig. 3.28 shows the simulation results using this method. We see that, as expected, the marginal stability is preserved and the numerical solution is very close to the analytical solution.

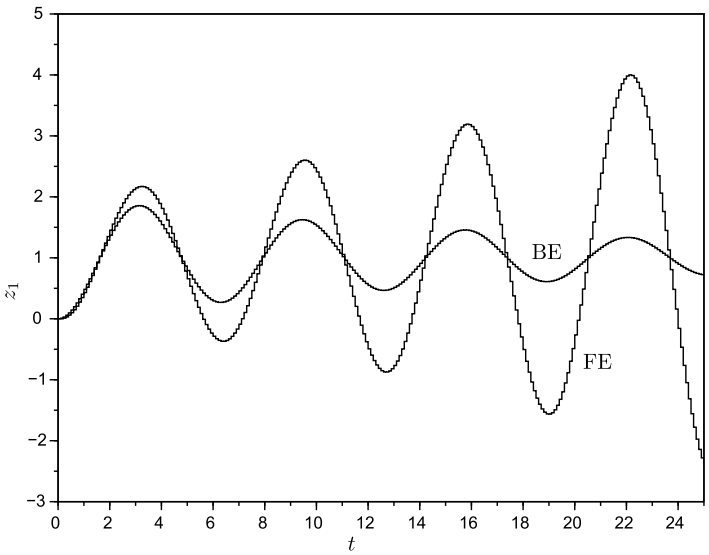

FIGURE 3.27

Forward (FE) and Backward Euler (BE) simulation of the spring–mass–damper model of Eq. (3.7) with $b = 0$ using a step size $h = 0.1$.

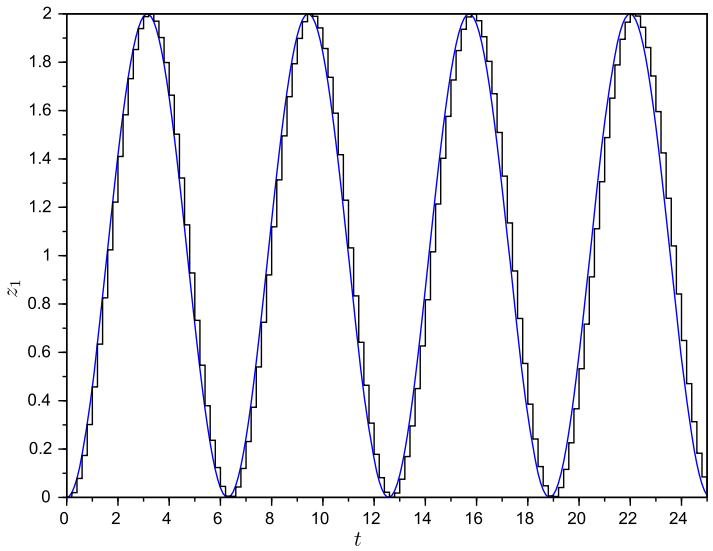

FIGURE 3.28

Trapezoidal rule simulation of the spring–mass–damper model of Eq. (3.7) with $b = 0$ using a step size $h = 0.2$. Numerical solution (stairs plot) vs. Analytical solution.

Methods like the trapezoidal rule are called *F-stable* or faithfully stable and they are very useful for the simulation of systems that do not lose energy.

Discontinuous Systems

The system below represents the dynamics of a ball bouncing against the floor:

$$\begin{aligned} \dot{z}_1(t) &= z_2(t) \\ \dot{z}_2(t) &= -g - d(t) \cdot \left(\frac{k}{m} z_1(t) + \frac{b}{m} z_2(t) \right) \end{aligned} \tag{3.44}$$

where

$$d(t) = \begin{cases} 0 & \text{if } z_1(t) > 0 \\ 1 & \text{otherwise} \end{cases} \tag{3.45}$$

This model says that the derivative of the position z_1 is the speed z_2, and the derivative of the speed (i.e., the acceleration $\dot{z}_2(t)$) depends on the discrete state $d(t)$. This discrete state takes the value 0 when the ball is in the air ($x_1 > 0$), so in that case the equation is that of a *free fall* model. Otherwise, $d(t)$ takes the value 1 when the ball is in contact with the floor ($x_1 \leq 0$), corresponding to a spring–damper model.

We adopted the following values for the parameters: $m = 1$, $b = 30$, $k = 10^6$, and $g = 9.81$. We also considered an initial state $z_1(0) = 2$, $z_2(0) = 0$.

We then simulated this system using the fourth order Runge–Kutta method (RK4) with three different small values for the step size: $h = 0.002$, $h = 0.001$, and $h = 0.0005$. The simulation results are depicted in Fig. 3.29.

In spite of the high order integration method used and the small step size h, the results are very different and there is not an obvious convergence to a reliable solution. Using variable step algorithms does not help much either.

The reason for the poor results is the presence of a discontinuity when $z_1(t) = 0$. Numerical integration methods are based on the continuity of the state variables and its derivatives. *This is in fact what allowed us to express the analytical solution as a Taylor series* in Eq. (3.21). In presence of discontinuities that series expansion is not valid and the numerical algorithms no longer approximate the analytical solution up their order of accuracy.

The solution to this problem requires the use of *event detection* routines, that detect the occurrence of discontinuities. Once a discontinuity is detected, the simulation is advanced until the location of the discontinuity, and it is restarted from that point under the new conditions after the event occurrence (this process is known as *event handling*). That way, the algorithms steps never integrate across a discontinuity. They just simulate a succession of purely continuous systems.

The use of event detection and event handling routines can add significant computational costs to numerical ODE solvers. For that reason, the simulation of systems with frequent discontinuities is still an important problem in the literature on numerical simulation.

In Chapter 19 we will introduce a new family of numerical ODE solvers based on state quantization principles that can considerably reduce the computational costs related to event detection and handling

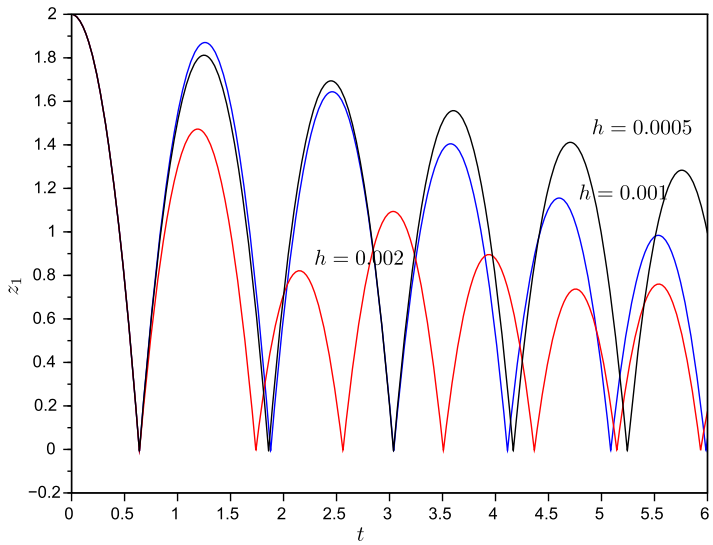

FIGURE 3.29

RK4 simulation of the bouncing ball model of Eq. (3.44) with $b = 0$ using different step sizes.

in discontinuous systems. This new family is an essential component supporting the iterative systems specification and DEVS-based hybrid simulation that are introduced in this third edition.

3.3 DISCRETE EVENT MODELS AND THEIR SIMULATORS

3.3.1 INTRODUCTION

We have already seen how a discrete event approach can be taken to obtain more efficient simulation of cellular automata. In this section, we will consider discrete event modeling as a paradigm in its own right. However, the basic motivation remains that discrete event modeling is an attractive formalism because it is intrinsically tuned to the capabilities and limitations of digital computers. Being the "new guy on the block", it still has not achieved the status that differential equations, with a three hundred year history, have. But it is likely to play more and more of a role in all kinds of modeling in the future. In keeping with this prediction, we start the introduction with a formulation of discrete event cellular automata which naturally follows from the earlier discussion. Only after having made this transition, will we come back to consider the more usual types of workflow models associated with discrete event simulation.

3.3.2 DISCRETE EVENT CELLULAR AUTOMATA

The original version of the Game of Life assumes that all births and deaths take the same time (equal to a time step). A more accurate representation of a cell's life cycle assumes that birth and death are dependent on a quantity that represents the ability of the cell to fit into its environment, called its fitness. A cell attains positive fitness when its neighborhood is supportive, that is, when it has exactly 3 neighbors. And the fitness will diminish rapidly when its environment is hostile. Whenever the fitness then reaches 0, the cell will die. On the other hand, a dead cell will have a negative initial fitness, lets say -2. When the environment is supportive and the fitness crosses the zero level, the cell will be born.

Fig. 3.30 shows an example behavior of such a cell model dependent on its environment (the sum of alive neighbors). The initial sum of alive neighbors being 3 causes the fitness of the dead cell to increase linearly until it crosses zero and a birth occurs. The fitness increases further until it reaches a saturation level (here 6). When the sum of alive neighbors changes from 3 to 4, the environment gets hostile, the fitness will decrease, and the cell will die. The death is accompanied by a discontinuous drop of the fitness level to a minimum fitness of -2.

What is needed to model such a process? One way would be to compute the trajectories of the fitness parameter and see when zero crossings and hence death and birth events occur. This would be the approach of combined discrete event and continuous simulation which we discuss in detail in Chapter 9. However, this approach is computationally inefficient since we have to compute each continuous trajectory at every point along its path. A much more efficient way is to adopt a pure event-based approach where we concentrate on the interesting events only, namely births and deaths as well as the changes in the neighborhood. The event-based approach jumps from one interesting event to the next omitting the uninteresting behavior in-between.

The prerequisite of discrete event modeling therefore is to have a means to determine when interesting things happen. Events can be caused by the environment, like the changes of the sum of alive neighbors. The occurrences of such *external* events are not under the control of the model component

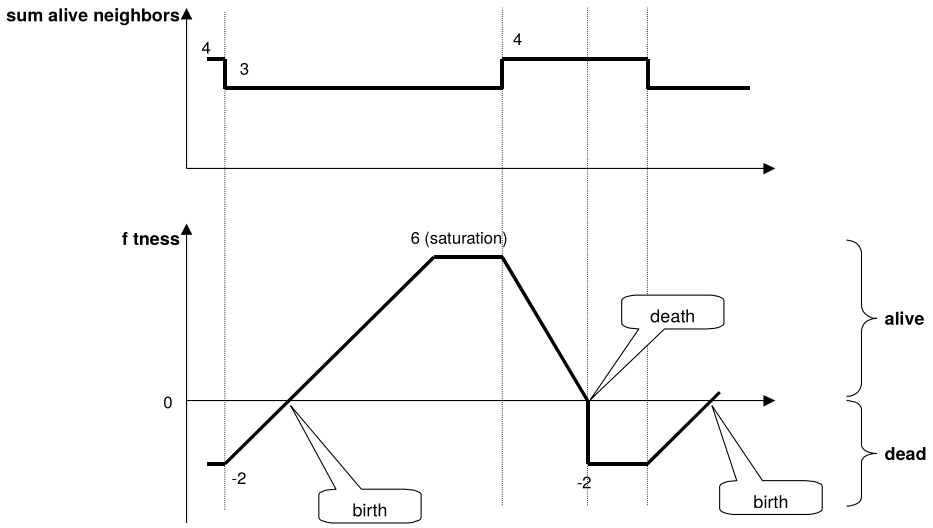

FIGURE 3.30

Behavior of GOL model with fitness.

itself. On the other side, the component may schedule events to occur. Those are called *internal* events and the component itself determines their time of occurrence. Given a particular state, e.g. a particular fitness of the cell, a time advance is specified as the time it takes until the next *internal event* occurs, supposing that no external event happens in the meantime. In our event-based GOL model the time advance is used to schedule the times when a birth or a death is supposed to happen (Fig. 3.31). As fitness changes linearly based on the neighborhood, the times of the events (times of the zero-crossings of fitness) can easily be predicted. Upon expiration of the time advance, the state transition function of a cell is applied, eventually resulting in a new time advance and a new scheduled event.

In discrete event simulation of the GOL model, one has to execute the scheduled internal events of the different cells at their event times. And since now cells are waiting, we must see to it that their time advances. Moreover, at any state change through an internal event we must take care to examine the cell's neighbors for possible state changes. A change in state may affect their waiting times as well as result in scheduling of new events and cancellation of events. Fig. 3.32 illustrates this process. Consider that all shaded cells in Fig. 3.32A are alive. Cell at $(0, 0)$ has enough neighbors to become alive and cell at $(-1, -1)$ only has 1 alive neighbor and therefore will die. Thus, a birth and a death are scheduled in these cells at times, say 1 and 2, respectively (shown by the numbers). At time 1, the origin cell undergoes its state transition and transits to the alive state shown by the thunderbolt in Fig. 3.32B. What effect does this have on the neighboring cells? Fig. 3.32B shows that as a result of the birth, cells at $(0, 1)$, $(0, -1)$ and $(1, 1)$ have three neighbors and must be scheduled for births at time 3. Cell $(-1, 1)$ has 4 alive neighbors now and a death will be scheduled for it at time 3. Cell $(-1, -1)$ has had a death scheduled before. However, due to the birth it has two neighbors now and the conditions for dying do not apply any more. Hence, the death at time 2 must be canceled. We see that the effect of

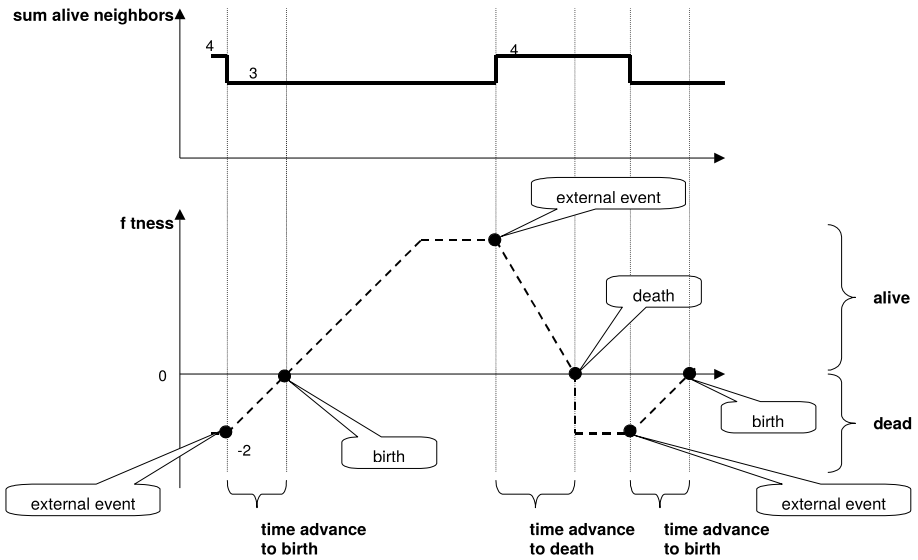

FIGURE 3.31

Behavior of the eventistic GOL model.

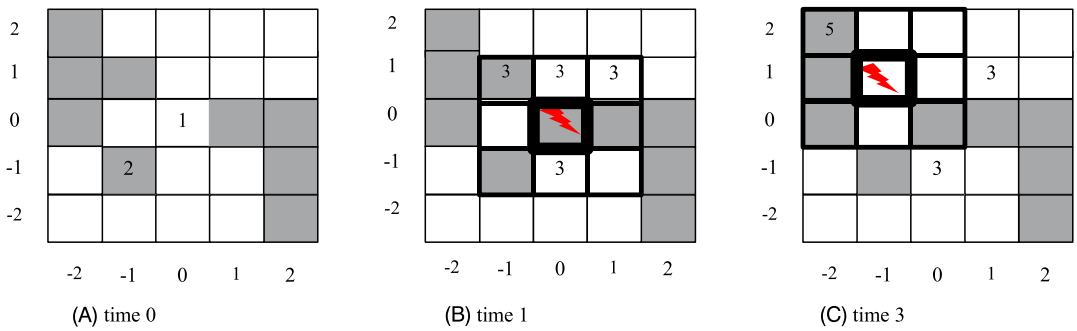

FIGURE 3.32

Event processing in Game of Life event model.

a state transition may not only be to schedule new events but also to cancel events that were scheduled in the past.

Now in Fig. 3.32B, the next earliest scheduled event is at time, 3. Therefore, the system can jump from the current time, 1, to the next event time, 3 without considering the times in-between. This illustrates an important *efficiency advantage in discrete event simulation* – during times when no events are scheduled, no components need be scanned. Contrast this with discrete time simulation in which scanning must be performed at each time step (and involves all cells in the worst case).

The situation at time 3 also illustrates a problem that arises in discrete event simulation – that of *simultaneous events*. From Fig. 3.32B we see that there are several cells scheduled for time 3, namely a birth at (0, 1), (1, 1) and (0, −1) and a death at (−1, 1). The question is who goes first and what is the result. Note, that if (−1, 1) goes first, the condition that (0, 1) has three live neighbors will not apply any more and the birth event just scheduled will be canceled again (Fig. 3.32C). However, if (0, 1) would go first, the birth in (0, 1) is actually carried out. Therefore, the results *will be different for different orderings of activation*. There are several approaches to the problem of simultaneous events. One solution is to let all simultaneous events undergo their state transitions together. This is the approach of parallel DEVS to be discussed in Chapter 6. The approach employed by most simulation packages and classic DEVS is to define a priority among the components. Components with high priority go first. To adopt this approach we employ a *tie-breaking* procedure, which selects one event to process out of a set of contending simultaneous events.

We'll return to show how to formulate the Game of Life as a well defined discrete event model in Chapter 7 after we have introduced the necessary concepts for DEVS representation. In the next section, we show how discrete event models can be simulated using the most common method called event scheduling. This method can also be used to simulate the Game of Life and we'll leave that as an exercise at the end of the section.

3.3.3 DISCRETE EVENT WORLD VIEWS

There are three common types of simulation strategies employed in discrete event simulation languages – *the event scheduling*, *the activity scanning* and *process interaction*, the latter being a combination of the first two. These are also called "world views" and each makes certain forms of model description more naturally expressible than others. In other words, each is best for a particular view of the way the world works. Nowadays, most simulation environments employ a combination of these strategies. In Chapter 7, we will characterize them as multi-component discrete event systems and analyze them in some detail. As an introduction to workflow modeling, here we will now discuss only the simplest strategy, event scheduling.

EVENT SCHEDULING WORLD VIEW

Event oriented models *preschedule* all events – there is no provision for conditional events that can be activated by tests on the global state. Scheduling an event is straightforward when all the conditions necessary for its occurrence can be known in advance. However, this is not always the case as we have seen in the Game of Life above. Nevertheless, understanding pure event scheduling in the following simple example of workflow modeling will help you to understand how to employ more complex world views later.

Fig. 3.33 depicts a discrete event model with three components. The generator *Gen* generates jobs for processing by a server component SS. Jobs are queued in a buffer called *Queue* when the server is busy. Component *Gen* continually reschedules a job output with a time advance equal to *inter-gen-time*. If the server is idle when a job comes in, then it is activated immediately to begin processing the job. Otherwise the number of waiting jobs in the queue is increased by one. When processing starts, the server is scheduled to complete work in a time that represents the job's *service-time*. When this time expires, the server starts to process the next job in the queue, if there is one. If not, it *passivates* in

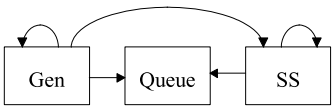

FIGURE 3.33

Generator, queue and server event scheduling model. The arrows indicate the effects of the event routines.

phase *idle*. To passivate means to set your time advance to infinity. The queue is a *passive* component (its time advance is always infinity) with a variable to store the number of jobs waiting currently.

Two event routines and a tie-breaking function express the interaction between these components:

```
Event: Generate_Job
  nr-waiting = nr-waiting + 1
  schedule a Generate_Job in inter-gen-time
  if nr-waiting = 1 then
          schedule a Process_Job in service-time


 Event: Process_Job
  nr-waiting = nr-waiting - 1
  if nr-waiting > 0 then
      schedule a Process_Job in service-time


// Note that the phrase "schedule an event in T"  means
// the same as "schedule an event  at time = clock time + T".

Break-Ties by: Process_Job then Generate_Job
```

As shown in Fig. 3.34, the event scheduling simulation algorithm employs a *list of events* that are ordered by increasing scheduling times. The event with the earliest scheduled time (e3 in Fig. 3.34) is removed from the list and the clock is advanced to the time of this imminent event (3). The routine associated with the imminent event is executed. A tie breaking procedure is employed if there are more than one such imminent events (recall the select function above). Execution of the event routine may cause new events to be added in the proper place on the list. For example $e4$ is scheduled at time 6. Also existing events may be rescheduled or even canceled (as we has seen above). The next cycle now begins with the clock advance to the earliest scheduled time (5), and so on. Each time an event routine is executed, one or more state variables may get new values.

Let's see how this works in our simple workflow example. Initially, the clock is set to 0 and a $Generate\_Job$ event is placed on the list with time 0. The state of the system is represented by $nr\text{-}waiting$ which is initially 0. Since this event is clearly imminent, it is executed. This causes two events to be scheduled: a $Generate\_Job$ at time $inter\text{-}gen\text{-}time$ and (since $nr\text{-}waiting$ becomes 1) a $Process\_Job$ at time $service - time$. What happens next depends on which of the scheduled times is earliest. Let's assume that $inter\text{-}gen\text{-}time$ is smaller than $service\text{-}time$. Then time is advanced to

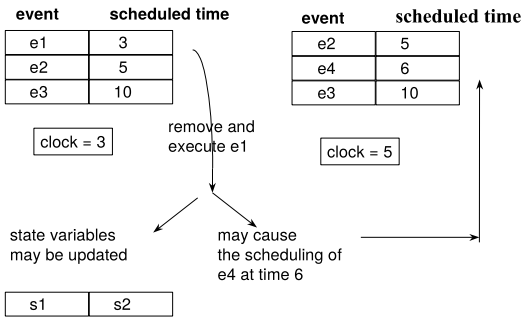

FIGURE 3.34

Event List Scheduling.

inter-gen-time and *Generate_Job* is executed. However, now only a new *Generate_Job* is scheduled – since the *nr-waiting* is 2, the *Process_Job* is not scheduled. Let's suppose that *service-time* $< 2 * inter\text{-}gen\text{-}time$ (the next event time for *Generate_Job*). Then the next event is a *Process_Job* and simulation continues in this manner.

Exercise 3.9. Hand execute the simulation algorithm for several cases. For example, 1) $inter\text{-}gen\text{-}time > service\text{-}time$, 2) $inter\text{-}gen\text{-}time = service\text{-}time$, 3) $inter\text{-}gen\text{-}time < service\text{-}time < 2 * inter\text{-}gen\text{-}time$, etc.

Exercise 3.10. Use the event scheduling approach to simulate the eventistic Game of Life introduced earlier in the section.

3.4 SUMMARY

In this chapter we have discussed the fundamental modeling formalisms which we will introduce in depth later in the book. Here we gained some insight into the intrinsic nature of the different modeling approaches. But the presentation can give us also some insight into the nature of dynamical systems in general. Let us summarize the modeling approaches in this sense.

The discrete time modeling approach which subsumes the popular finite automaton formalism as well as the difference equation formalism stands out through its simple simulation algorithm. It adopts a stepwise execution mode where all the components states are updated based on the state of the previous time step and the inputs.

The simplest form of this model type is the cellular automaton. In a cellular automaton a real system is reduced to basic coupling and communication between components. A cell on its own is quite boring. Only through couplings with other cells do quite astonishing complex and unpredictable behaviors result. So the cellular automaton is actually a model to study the complex behaviors that result from simple interactions among components. They are applied to study the emergence of group phenomena, like population dynamics, or spreading phenomena, like forest fires, avalanches or excitable media.

Continuous modeling and simulation is the classical approach of the natural sciences employing differential equations. Here, we focused only on ordinary differential equations and starting by study-

ing the qualitative forms of the trajectories obtained in linear time invariant systems. We concluded that the simulation of general ODEs required the help of numerical integration methods and we developed a brief introduction to their main concepts. Besides introducing different simulation algorithms (like Euler, Runge Kutta and Multi-Step families), we presented two fundamental properties of the approximations: accuracy and stability. We also introduced the algorithms for automatic step size control and we studied some features that frequently appear in practice and can complicate the correct behavior of numerical ODE solvers: stiffness, marginal stability, and discontinuities.

Finally, the event-based version of the Game of Life model gave us insight into the intrinsic nature of discrete event modeling and simulation. We have seen that considering only the interesting points in time – the events – is the basis of this modeling approach. To be able to find and schedule the events in time is the prerequisite for applying discrete event simulation. A local event is scheduled by specifying a time advance value based on the current state of a component.

In the GOL model defining the event times was quite straightforward. Since the fitness parameter increases and decreases linearly, it is simple to compute the time when a zero crossing will occur – that is when an event occurs. However, in general this will not always be feasible and several different approaches to tackle this problem are known. A very common approach, which is heavily employed in work flow modeling, is to use stochastic methods and generate the time advance values from a random distribution. Another approach is to measure the time spans for different states and keep them in a table for later look-up. These approaches are discussed in Chapter 17. If none of these methods is appropriate, one has the possibility to step back to the more expensive combined simulation approach where the continuous trajectories are actually computed by continuous simulation and, additionally, events have to be detected and executed in-between. This combined continuous/discrete event modeling approach is the topic of Chapter 9.

The GOL event model also showed us two important advantages of discrete event modeling and simulation compared to the traditional approaches. First, only those times, and those components, have to be considered where events actually occur. Second, only those state changes must be reported to coupled components which are actually relevant for their behavior. For example in the GOL model, only changes in the alive state – but not changes in the fitness parameter – are relevant for a cell's neighbors. Actually, in discrete event modeling, the strategy often is to model only those state changes as events which are actually relevant to the environment of a component. As long as nothing interesting to the environment happens, no state changes are made. These two issues give the discrete event modeling and simulation approach a great lead in computational efficiency, making it most attractive for digital computer simulation. We will come back to this issue in Chapter 20 on DEVS representation of dynamical systems.

3.5 SOURCES

Research on cellular automata is summarized in Wolfram et al. (1986). Good expositions of continuous modeling are found in Cellier (1991), Bossel (1994), and Fritzson (2010), where the latter provides a complete reference about Modelica language. There is a vast literature on numerical integration methods. The main source of Section 3.2 is the book (Cellier and Kofman, 2006). A compact introduction to the most popular algorithms is given in Burden and Faires (1989) and algorithms in C can be found in Press et al. (1992). A more rigorous mathematical treatment of numerical algorithms for ODEs in-

cluding the proofs of convergence can be found in the classic books (Hairer et al., 1993; Hairer and Wanner, 1991). Banks et al. (1995) and Law et al. (1991) give popular introductions to discrete event simulation. A control-oriented view of discrete event systems is given in Cassandras (1993).

REFERENCES

Bossel, H., 1994. Modeling and Simulations. Technical report. A K Peters, Ltd. ISBN 1-56881-033-4.

Burks, A.W., 1970. Essays on Cellular Automata. University of Illinois Press.

Cassandras, C., 1993. Discrete Event Systems: Modeling and Performance Analysis. Richard Irwin, New York, NY.

Cellier, F., 1991. Continuous System Modeling. Springer-Verlag, New York.

Cellier, F., Kofman, E., 2006. Continuous System Simulation. Springer, New York.

Fritzson, P., 2010. Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. John Wiley & Sons.

Gardner, M., 1970. Mathematical games: the fantastic combinations of John Conway's new solitaire game "life". Scientific American 223 (4), 120–123.

Hairer, E., Nørsett, S., Wanner, G., 1993. Solving Ordinary Differential Equations. I, Nonstiff Problems. Springer-Verlag, Berlin.

Hairer, E., Wanner, G., 1991. Solving Ordinary Differential Equations. II, Stiff and Differential-Algebraic Problems. Springer-Verlag, Berlin.

Banks, J., Carson, C., Nelson, B.L., 1995. Discrete-Event System Simulation. Prentice Hall Press, NJ.

Law, A.M., Kelton, W.D., Kelton, W.D., 1991. Simulation Modeling and Analysis, vol. 2. McGraw-Hill, New York.

Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P., 1992. Numerical Recipes in C: The Art of Scientific Computing Second.

Burden, Richard L., Faires, J.D., 1989. Numerical Analysis, fourth edition. PWS-KENT Publishing Company.

Wolfram, S., et al., 1986. Theory and Applications of Cellular Automata, vol. 1. World Scientific, Singapore.