system's stability is its state-transition matrix, which reflects the three stability criteria. If *any element* of the state-transition matrix grows to infinity as time becomes large, the system is *unstable*. Then it is possible to find at least *one* initial condition that leads to an unbounded initial response. The state-transition matrix contains information about how a system will respond to an *arbitrary* initial condition. Hence, the stability of a system is deduced from *all possible* initial conditions (i.e. the state-transition matrix), rather than from only some *specific* ones.

4.4 Numerical Solution of Linear Time-Invariant State-Equations

In the previous sections, we saw two methods for calculating the state-transition matrix, which is required for the solution of the linear state-equations. The diagonalization method works only if the system eigenvalues are distinct. Calculating the state-transition matrix by the inverse Laplace transform method of Eq. (4.47) is a tedious process, taking into account the matrix inversion, partial fraction expansion, and inverse Laplace transformation of each element of the resolvent, as Example 4.4 illustrates. While the partial fraction expansions can be carried out using the intrinsic MATLAB function *residue*, the other steps must be performed by hand. Clearly, the utility of Eq. (4.47) is limited to small order systems. Even for the systems which allow easy calculation of the state-transition matrix, the calculation of the steady-state response requires time integration of the input terms (Eq. (4.27)), which is no mean task if the inputs are arbitrary functions of time.

The definition of the matrix exponential, $\exp\{\mathbf{A}(t-t_0)\}$, by the Taylor series expansion of Eq. (4.18) gives us another way of calculating the state-transition matrix. However, since Eq. (4.18) requires evaluation of an *infinite* series, the *exact* calculation of $\exp\{\mathbf{A}(t-t_0)\}$ is impossible by this approach. Instead, we use an *approximation* to $\exp\{\mathbf{A}(t-t_0)\}$ in which only a *finite* number of terms are retained in the series on the right-hand side of Eq. (4.18):

$$\exp\{\mathbf{A}(t-t_0)\} \approx \mathbf{I} + \mathbf{A}(t-t_0) + \mathbf{A}^2(t-t_0)^2/2! + \mathbf{A}^3(t-t_0)^3/3! + \cdots + \mathbf{A}^N(t-t_0)^N/N! \tag{4.54}$$

Note that the approximation given by Eq. (4.54) consists of powers of $\mathbf{A}(t-t_0)$ up to N. In Eq. (4.54), we have neglected the following infinite series, called the *remainder series*, $\mathbf{R}_N$, which is also the *error* in our approximation of $\exp\{\mathbf{A}(t-t_0)\}$:

$$\mathbf{R}_N = \mathbf{A}^{N+1}(t-t_0)^{N+1}/(N+1)! + \mathbf{A}^{N+2}(t-t_0)^{N+2}/(N+2)! + \mathbf{A}^{N+3}(t-t_0)^{N+3}/(N+3)! + \cdots = \sum_{k=N+1}^{\infty} \mathbf{A}^k(t-t_0)^k/k! \tag{4.55}$$

Clearly, the accuracy of the approximation in Eq. (4.54) depends upon how large is the error, $\mathbf{R}_N$, given by Eq. (4.55). Since $\mathbf{R}_N$ is a matrix, when we ask how large is the error, we mean how large is each element of $\mathbf{R}_N$. The magnitude of the matrix, $\mathbf{R}_N$, is

a matrix consisting of *magnitudes* of the elements of $\mathbf{R}_N$. However, it is quite useful to assign a scalar quantity, called the *norm*, to measure the magnitude of a matrix. There are several ways in which the norm of a matrix can be defined, such as the sum of the magnitudes of all the elements, or the square-root of the sum of the squares of all the elements. Let us assign such a scalar norm to measure the magnitude of the error matrix, $\mathbf{R}_N$, and denote it by the symbol $\|\mathbf{R}_N\|$, which is written as follows:

$$\|\mathbf{R}_N\| = \left\| \sum_{k=N+1}^{\infty} \mathbf{A}^k (t - t_0)^k / k! \right\| \leq \sum_{k=N+1}^{\infty} \|\mathbf{A}^k\| (t - t_0)^k / k \tag{4.56}$$

The inequality on the right-hand side of Eq. (4.56) is due to the well known *triangle inequality*, which implies that if a and b are real numbers, then $|a + b| \leq |a| + |b|$. Now, for our approximation of Eq. (4.54) to be accurate, the first thing we require is that the magnitude of error, $\|\mathbf{R}_N\|$, be a *finite quantity*. Secondly, the error magnitude should be *small*. The first requirement is met by noting that the Taylor series of Eq. (4.18) is *convergent*, i.e. the successive terms of the series become smaller and smaller. The inifinite series on the extreme right-hand side of Eq. (4.56) – which is a part of the Taylor series – is also finite. Hence, irrespective of the value of N, $\|\mathbf{R}_N\|$ is always finite. From Eq. (4.56), we see that the approximation error can be made small in two ways: (a) by *increasing* N, and (b) by *decreasing* $(t - t_0)$. The implementation of Eq. (4.54) in a computer program can be done using an *algorithm* which selects the highest power, N, based on the desired accuracy, i.e. the error given by Eq. (4.56). MATLAB uses a similar algorithm in its function named *expm2* which computes the matrix exponential using the finite series approximation. Other algorithms based on the finite series approximation to the matrix exponential are given in Golub and van Loan [1] and Moler and van Loan [2]. The accuracy of the algorithms varies according to their implementation. The MATLAB functions *expm* and *expm1* use two different algorithms for the computation of the matrix exponential based on Laplace transform of the finite-series of Eq. (4.54) – which results in each element of the matrix exponential being approximated by a rational polynomial in s, called the *Padé approximation*. Compared to *expm2* – which directly implements the finite Taylor series approximation – *expm* and *expm1* are more accurate.

There is a limit to which the number of terms in the approximation can be increased. Therefore, for a given N, the accuracy of approximation in Eq. (4.54) can be increased by making $(t - t_0)$ small. How small is small enough? Obviously, the answer depends upon the system's dynamics matrix, $\mathbf{A}$, as well as on N. If $(t - t_0)$ is chosen to be small, how will we evaluate the state-transition matrix for large time? For this purpose, we will use the *time-marching* approach defined by the following property of the state-transition matrix (Table 4.1):

$$\exp\{\mathbf{A}(t - t_0)\} = \exp\{\mathbf{A}(t - t_1)\} \exp\{\mathbf{A}(t_1 - t_0)\} \tag{4.57}$$

where $t_0 < t_1 < t$. The *time-marching* approach for the computation of the state-transition matrix consists of evaluating $e^{\mathbf{A}\Delta t}$ as follows using Eq. (4.54):

$$e^{\mathbf{A}\Delta t} \approx \mathbf{I} + \mathbf{A}\Delta t + \mathbf{A}^2(\Delta t)^2/2! + \mathbf{A}^3(\Delta t)^3/3! + \cdots + \mathbf{A}^N(\Delta t)^N/N! \tag{4.58}$$

where Δt is a *small time-step*, and then marching ahead in time – like an army marches on with fixed footsteps – using Eq. (4.57) with $t = t_0 + n\Delta t$ and $t_1 = t_0 + (n-1)\Delta t$ as follows:

$$\exp\{\mathbf{A}(t_0 + n\Delta t)\} = e^{\mathbf{A}\Delta t} \exp\{\mathbf{A}[t_0 + (n-1)\Delta t]\} \tag{4.59}$$

Equation (4.59) allows successive evaluation of $\exp\{\mathbf{A}(t_0 + n\Delta t)\}$ for $n = 1, 2, 3, \ldots$ until the final time, t, is reached. The time-marching approach given by Eqs. (4.58) and (4.59) can be easily programmed on a digital computer. However, instead of finding the state-transition matrix at time, $t > t_0$, we are more interested in obtaining the solution of the state-equations, Eq. (4.8), $\mathbf{x}(t)$, when initial condition is specified at time t_0. To do so, let us apply the time-marching approach to Eq. (4.27) by substituting $t = t_0 + n\Delta t$, and writing the solution after n time-steps as follows:

$$\mathbf{x}(t_0 + n\Delta t) = e^{\mathbf{A}n\Delta t}\mathbf{x}(t_0) + \int_{t_0}^{t_0+n\Delta t} \exp\{\mathbf{A}(t_0 + n\Delta t - \tau)\}\mathbf{B}\mathbf{u}(\tau)d\tau; \quad (n = 1, 2, 3, \ldots) \tag{4.60}$$

For the first time step, i.e. $n = 1$, Eq. (4.60) is written as follows:

$$\mathbf{x}(t_0 + \Delta t) = e^{\mathbf{A}\Delta t}\mathbf{x}(t_0) + \int_{t_0}^{t_0+\Delta t} \exp\{\mathbf{A}(t_0 + \Delta t - \tau)\}\mathbf{B}\mathbf{u}(\tau)d\tau \tag{4.61}$$

The integral term in Eq. (4.61) can be expressed as

$$\int_{t_0}^{t_0+\Delta t} \exp\{\mathbf{A}(t_0 + \Delta t - \tau)\}\mathbf{B}\mathbf{u}(\tau)d\tau = e^{\mathbf{A}\Delta t}\int_0^{\Delta t} e^{-\mathbf{A}T}\mathbf{B}\mathbf{u}(t_0 + T)dT \tag{4.62}$$

where $T = \tau - t_0$. Since the time step, Δt, is small, we can *assume* that the integrand vector $e^{-\mathbf{A}T}\mathbf{B}\mathbf{u}(t_0 + T)$ is *essentially constant* in the interval $0 < T < \Delta t$, and is equal to $e^{-\mathbf{A}\Delta t}\mathbf{B}\mathbf{u}(t_0 + \Delta t)$. Thus, we can *approximate* the integral term in Eq. (4.62) as follows:

$$e^{\mathbf{A}\Delta t}\int_0^{\Delta t} e^{-\mathbf{A}T}\mathbf{B}\mathbf{u}(t_0 + T)dT \approx e^{\mathbf{A}\Delta t}e^{-\mathbf{A}\Delta t}\mathbf{B}\mathbf{u}(t_0 + \Delta t)\Delta t = \mathbf{B}\mathbf{u}(t_0 + \Delta t)\Delta t = \mathbf{B}\mathbf{u}(t_0)\Delta t \tag{4.63}$$

Note that in Eq. (4.63), we have used $\mathbf{u}(t_0 + \Delta t) = \mathbf{u}(t_0)$, because the input vector is assumed to be *constant* in the interval $t_0 < t < t_0 + \Delta t$. Substituting Eq. (4.63) into Eq. (4.61), the approximate solution after the first time step is written as

$$\mathbf{x}(t_0 + \Delta t) \approx e^{\mathbf{A}\Delta t}\mathbf{x}(t_0) + \mathbf{B}\mathbf{u}(t_0)\Delta t \tag{4.64}$$

For the next time step, i.e. $n = 2$ and $t = t_0 + 2\Delta t$, we can use the solution after the first time step, $\mathbf{x}(t_0 + \Delta t)$, which is already known from Eq. (4.64), as the initial condition and, assuming that the input vector is constant in the interval $t_0 + \Delta t < t < t_0 + 2\Delta t$, the solution can be written as follows:

$$\mathbf{x}(t_0 + 2\Delta t) \approx e^{\mathbf{A}\Delta t}\mathbf{x}(t_0 + \Delta t) + \mathbf{B}\mathbf{u}(t_0 + \Delta t)\Delta t \tag{4.65}$$

The process of time-marching, i.e. using the solution after the previous time step as the initial condition for calculating the solution after the next time step, is continued and the

solution after n time steps can be approximated as follows:

$$\mathbf{x}(t_0 + n\Delta t) \approx e^{\mathbf{A}\Delta t}\mathbf{x}(t_0 + (n-1)\Delta t) + \mathbf{B}\mathbf{u}(t_0 + (n-1)\Delta t)\Delta t; \quad (n = 1, 2, 3, \ldots) \tag{4.66}$$

A special case of the system response is to the *unit impulse inputs*, i.e. $\mathbf{u}(t) = \delta(t - t_0)[\,1; \quad 1; \quad \ldots; \quad 1\,]^T$. In such a case, the integral in Eq. (4.62) is exactly evaluated as follows, using the *sampling property* of the unit impulse function, $\delta(t - t_0)$, given by Eq. (2.24):

$$\int_{t_0}^{t_0+\Delta t} \exp\{\mathbf{A}(t_0 + \Delta t - \tau)\}\mathbf{B}\mathbf{u}(\tau)d\tau = e^{\mathbf{A}\Delta t}\mathbf{B} \tag{4.67}$$

which results in the following solution:

$$\mathbf{x}(t_0 + n\Delta t) = e^{\mathbf{A}\Delta t}[\mathbf{x}(t_0 + (n-1)\Delta t) + \mathbf{B}\Delta t]; \quad (n = 1, 2, 3, \ldots) \tag{4.68}$$

Note that the solution given by Eq. (4.68) is an exact result, and is valid only if all the inputs are unit impulse functions applied at time $t = t_0$.

By the time-marching method of solving the state-equations we have essentially converted the *continuous-time* system, given by Eq. (4.8), to a *discrete-time* (or *digital*) system given by Eq. (4.66) (or, in the special case of unit impulse inputs, by Eq. (4.67)). The difference between the two is enormous, as we will see in Chapter 8. While in a continuous-time system the time is smoothly changing and can assume *any real value*, in a digital system the time can *only be an integral multiple of the time step*, Δt (i.e. Δt multiplied by an integer). The continuous-time system is clearly the limiting case of the digital system in the limit $\Delta t \to 0$. Hence, the accuracy of approximating a continuous-time system by a digital system is crucially dependent on the size of the time step, Δt; the accuracy improves as Δt becomes smaller. The *state-equation* of a linear, time-invariant, digital system with $t_0 = 0$ can be written as

$$\mathbf{x}(n\Delta t) = \mathbf{A}_d\mathbf{x}((n-1)\Delta t) + \mathbf{B}_d\mathbf{u}((n-1)\Delta t); \quad (n = 1, 2, 3, \ldots) \tag{4.69}$$

where $\mathbf{A}_d$ and $\mathbf{B}_d$ are the digital state coefficient matrices. Comparing Eqs. (4.66) and (4.69) we find that the solution of a continuous-time state-equation is approximated by the solution of a digital state-equation with $\mathbf{A}_d = e^{\mathbf{A}\Delta t}$ and $\mathbf{B}_d = \mathbf{B}\Delta t$ when the initial condition is specified at time $t = 0$. The digital solution, $\mathbf{x}(n\Delta t)$, is simply obtained from Eq. (4.69) using the time-marching method starting from the initial condition, $\mathbf{x}(0)$, and assuming that the input vector is constant during each time step. The digital solution of Eq. (4.69) is easily implemented on a *digital computer*, which itself works with a *non-zero time step* and input signals that are specified over each time step (called *digital signals*).

The assumption of a constant input vector during each time step, used in Eq. (4.63), results in a *staircase* like approximation of $\mathbf{u}(t)$ (Figure 4.3), and is called a *zero-order hold*, i.e. a zero-order linear interpolation of $\mathbf{u}(t)$ during each time step, $(n-1)\Delta t < t < n\Delta t$. The zero-order hold approximates $\mathbf{u}(t)$ by a step function in each time step. It is a good approximation even with a large time step, Δt, if $\mathbf{u}(t)$ itself is a *step* like function in continuous-time, such as a *square wave*. However, if $\mathbf{u}(t)$ is a smooth function in continuous-time, then it is more accurate to use a higher order interpolation to approximate $\mathbf{u}(t)$ in each time step, rather than using the zero-order hold. One such approximation is

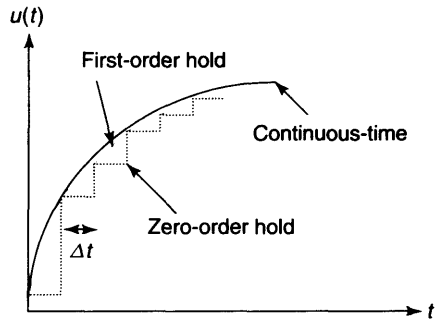

Figure 4.3 The zero-order and first-order hold digital approximations of a continuous-time input, $u(t)$

the *first-order hold* which approximates $\mathbf{u}(t)$ as a *ramp function* (i.e. a first-order linear interpolation) in each time step $(n-1)\Delta t < t < n\Delta t$ (Figure 4.3).

The conversion of a continuous-time system to the corresponding digital approximation using the zero-order hold for the input vector (Eq. (4.66)) is performed by the MATLAB Control System Toolbox (CST) function *c2d*, which calculates $e^{\mathbf{A}\Delta t}$ using the intrinsic MATLAB function *expm*. The command *c2d* is employed as follows:

```
>>sysd = c2d(sysc,Ts,'method') <enter>
```

where *sysc* is the continuous-time state-space LTI object, *Ts* is the specified time step, (Δt), and *sysd* is the resulting digital state-space approximation of Eq. (4.69). The *'method'* allows a user to select among zero-order hold (*'zoh'*), first-order hold (*'foh'*), or higher-order interpolations for the input vector, called *Tustin* (or *bilinear*) approximation (*'tustin'*), and Tustin interpolation with *frequency prewarping* (*'prewarp'*). The Tustin approximation involves a *trapezoidal* approximation for $\mathbf{u}^{(1)}(t)$ in each time step (we will discuss the *Tustin* approximation a little more in Chapter 8). Tustin interpolation with frequency prewarping (*'prewarp'*) is a more accurate interpolation than plain *'tustin'*. An alternative to *c2d* is the CST function *c2dm*, which lets the user work directly with the state coefficient matrices rather than the LTI objects of the continuous time and digital systems as follows:

```
>>[Ad,Bd,Cd,Dd] = c2dm(A,B,C,D,Ts,'method') <enter>
```

where **A, B, C, D** are the continuous-time state-space coefficient matrices, **Ad, Bd, Cd, Dd** are the returned *digital* state-space coefficient matrices, and Ts and *'method'* are the same as in *c2d*. For more information on these MATLAB (CST) commands, you may refer to the *Users' Guide* for MATLAB *Control System Toolbox* [3].

How large should be the step size, Δt, selected in obtaining the digital approximation given by Eq. (4.69)? This question is best answered by considering how fast the system is likely to respond to a given initial condition, or to an applied input. Obviously, a fast changing response will not be captured very accurately by using a large Δt. Since the state-transition matrix, $e^{\mathbf{A}\Delta t}$, has elements which are combinations of $\exp(\lambda_k \Delta t)$, where λ_k, $k = 1, 2$, etc., are the eigenvalues of **A**, it stand to reason that the time step.

Δt, should be small enough to accurately evaluate the *fastest changing element* of $e^{\mathbf{A}\Delta t}$, which is represented by the eigenvalue, λ_k, corresponding to the largest *natural frequency*. Recall from Chapter 2 that the natural frequency is associated with the *imaginary part*, b, of the eigenvalue, $\lambda_k = a + bi$, which leads to *oscillatory* terms such as $\sin(b\Delta t)$ and $\cos(b\Delta t)$ in the elements of $e^{\mathbf{A}\Delta t}$ (Example 4.2). Hence, we should select Δt such that $\Delta t < 1/|b|_{\max}$ where $|b|_{\max}$ denotes the largest imaginary part magnitude of all the eigenvalues of **A**. To be on the safe-side of accuracy, it is advisable to make the time step smaller than the *ten times* the reciprocal of the largest imaginary part magnitude, i.e. $\Delta t < 0.1/|b|_{\max}$. If all the eigenvalues of a system are real, then the oscillatory terms are absent in the state-transition matrix, and one can choose the time step to be smaller than the reciprocal of the largest *real part* magnitude of all the eigenvalues of the system, i.e. $\Delta t < 1/|a|_{\max}$.

Once a digital approximation, Eq. (4.69), to the linear, time-invariant, continuous-time system is available, the MATLAB (CST) command *ltitr* can be used to solve for $\mathbf{x}(n\Delta t)$ using time-marching with $n = 1, 2, 3, \ldots$, given the initial condition, $\mathbf{x}(0)$, and the input vector, $\mathbf{u}(t)$, at the time points, $t = (n-1)\Delta t$, $n = 1, 2, 3, \ldots$ as follows:

```
>>x = ltitr(Ad,Bd,u,x0) <enter>
```

where **Ad, Bd**, are the digital state-space coefficient matrices, **x0** is the initial condition vector, **u** is a matrix having as many columns as there are inputs, and the ith row of **u** corresponds to the ith time point. **x** is the returned matrix with as many columns as there

Table 4.2 Listing of the M-file *march.m*

march.m

```
function [y,X] = march(A,B,C,D,X0,t,u,method)
% Time-marching solution of linear, time-invariant
% state-space equations using the digital approximation.
% A= state dynamics matrix; B= state input coefficient matrix;
% C= state output coefficient matrix;
% D= direct transmission matrix;
% X0= initial state vector; t= time vector.
% u=matrix with the ith input stored in the ith column, and jth row
% corresponding to the jth time point.
% y= returned output matrix with ith output stored in the ith column,
% and jth row corresponding to the jth time point.
% X= returned state matrix with ith state variable stored in the ith
% column, and jth row corresponding to the jth time point.
% method= method of digital interpolation for the inputs(see 'c2dm')
% copyright(c)2000 by Ashish Tewari
n=size(t,2);
dt=t(2)-t(1);
% digital approximation of the continuous-time system:-
[ad,bd,cd,dd]=c2dm(A,B,C,D,dt,method);
% solution of the digital state-equation by time-marching:-
X=ltitr(ad,bd,u,X0);
% calculation of the outputs:-
y=X*C'+u*D';
```

are state variables, and with the *same number* of rows as **u**, with the ith row corresponding to the ith time-point (the first row of **x** consists of the elements of **x0**).

The entire solution procedure for the state-space equation using the digital approximation of Eq. (4.66) can be programmed in a new M-file named *march*, which is tabulated in Table 4.2. This M-file can be executed as follows:

```
>>[y,x] = march(A,B,C,D,x0,t,u,'method') <enter>
```

where **A, B, C, D, x0, u, x**, and '*method*' are the same as those explained previously in the usage of the MATLAB (CST) command *c2dm*, while **t** is the time vector containing the equally spaced time-points at which the input, **u**, is specified, and **y** is the returned matrix containing the outputs of the system in its columns, with each row of **y** corresponding to a different time-point.

Example 4.6

Using the time-marching approach, let us calculate the response of the system given in Example 4.1 when $u(t) = u_s(t)$. Since the largest eigenvalue is -3, we can select the time step to be $\Delta t < 1/3$, or $\Delta t < 0.333$. Selecting $\Delta t = 0.1$, we can generate the time vector regularly spaced from $t = 0$ to $t = 2$ s, and specify the unit step input, u, as follows:

```
>>t=0:0.1:2; u=ones(size(t,2),1); <enter>
```

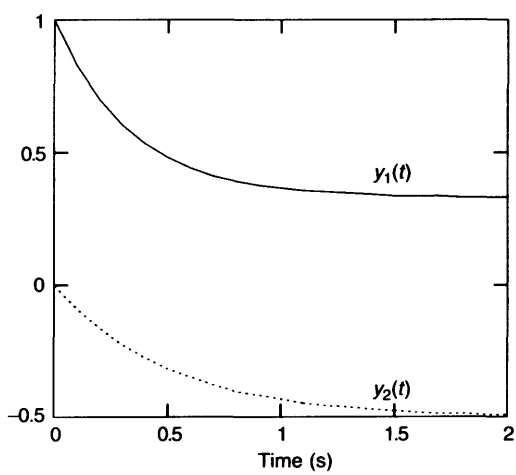

Figure 4.4 The calculated outputs, $y_1(t)$ and $y_2(t)$, for Example 4.6

Then, the M-file *march.m* is executed with zero-order hold, after specifying the state coefficient matrices and the initial condition vector (it is assumed that the outputs are the state variables themselves, i.e. $\mathbf{C} = \mathbf{I}$, $\mathbf{D} = \mathbf{0}$) as follows:

```
>>A= [-3 0; 0 -2]; B = [1; -1]; C = eye(2); D = zeros(2,1); X0 = [1; 0]; <enter>

>>[y,X] = march(A,B,C,D,X0,t,u,'zoh'); <enter>
```

The outputs, which are also the two state variables, $x_1(t)$ and $x_2(t)$, are plotted against the time vector, **t**, in Figure 4.4 as follows:

```
>>plot(t,y) <enter>
```

You may verify that the result plotted in Figure 4.4 is almost indistinguishable from the analytical result obtained in Eqs. (4.13) and (4.14), which for a unit step input yield the following:

$$x_1(t) = (2e^{-3t} + 1)/3; \quad x_2(t) = (e^{-2t} - 1)/2 \tag{4.70}$$

Example 4.7

In Examples 2.10 and 3.10, we saw how the linearized longitudinal motion of an aircraft can be represented by appropriate transfer functions and state-space representations. These examples had involved the assumption that the structure of the aircraft is *rigid*, i.e. the aircraft does not get *deformed* by the *air-loads* acting on it. However, such an assumption is invalid, because most aircraft have rather *flexible* structures. Deformations of a flexible aircraft under changing air-loads caused by the aircraft's motion, result in a complex dynamics, called *aeroelasticity*. Usually, the *short-period mode* has a frequency closer to that of the elastic motion, while the *phugoid mode* has little aeroelastic effect. The longitudinal motion of a *flexible bomber* aircraft is modeled as a second order short-period mode, a second-order *fuselage bending mode*, and two first-order control-surface actuators. The sixth order system is described by the following linear, time-invariant, state-space representation:

$$\mathbf{A} = \begin{bmatrix} 0.4158 & 1.025 & -0.00267 & -0.0001106 & -0.08021 & 0 \\ -5.5 & -0.8302 & -0.06549 & -0.0039 & -5.115 & 0.809 \\ 0 & 0 & 1.0 & 0 & 0 & \\ -1040 & -78.35 & -34.83 & -0.6214 & -865.6 & -631 \\ 0 & 0 & 0 & -75 & 0 & \\ 0 & 0 & 0 & 0 & -100 & \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 75 & 0 \\ 0 & 100 \end{bmatrix}$$

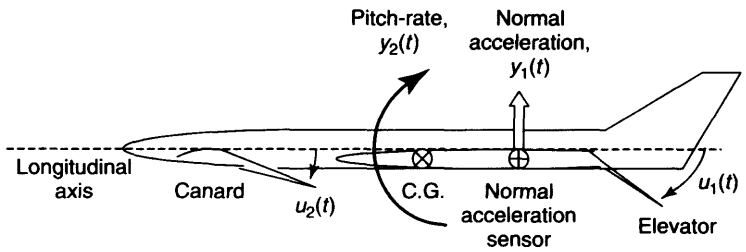

Figure 4.5 Inputs and outputs for the longitudinal dynamics of a flexible bomber aircraft

$$\mathbf{C} = \begin{bmatrix} -1491 & -146.43 & -40.2 & -0.9412 & -1285 & -564.66 \\ 0 & 1.0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{D} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{4.71}$$

The inputs are the *desired elevator deflection* (rad.), $u_1(t)$, and the *desired canard deflection* (rad.), $u_2(t)$, while the outputs are the sensor location's *normal acceleration* (m/s^2), $y_1(t)$, and the *pitch-rate* (rad./s), $y_2(t)$. See Figure 4.5 for a description of the inputs and outputs.

Let us calculate the response of the system if the initial condition and the input vector are the following:

$$\mathbf{x}(0) = [0.1; \quad 0; \quad 0; \quad 0; \quad 0; \quad 0]^T; \quad \mathbf{u}(t) = \begin{bmatrix} -0.1\sin(10t) \\ \sin(12t) \end{bmatrix} \tag{4.72}$$

First, let us select a proper time step for solving the state-equations. The system's eigenvalues are calculated using the command *damp* as follows:

```
>>damp(A) <enter>

Eigenvalue                    Damping      Freq. (rad/sec)
-4.2501e-001+1.8748e+000i     2.2109e-001  1.9224e+000
-4.2501e-001-1.8748e+000i     2.2109e-001  1.9224e+000
-5.0869e-001+6.0289e+000i     8.4077e-002  6.0503e+000
-5.0869e-001-6.0289e+000i     8.4077e-002  6.0503e+000
-7.5000e+001                  1.0000e+000  7.5000e+001
-1.0000e+002                  1.0000e+000  1.0000e+002
```

The largest imaginary part magnitude of the eigenvalues is 6.03, while the largest real part magnitude is 100. Therefore, from our earlier discussion, the time step should be selected such that $\Delta t < 0.1/6$ s and $\Delta t < 1/100$ s. Clearly, selecting the smaller of the two numbers, i.e. $\Delta t < 1/100$ s, will satisfy both the inequalities.

Hence, we select $\Delta t = 1/150$ s $= 6.6667\text{e} - 003$ s. The time vector, **t**, the input matrix, **u**, and initial condition vector, **x0**, are then specified as follows:

```
>>t = 0:6.6667e-3:5; u = [-0.05*sin(10*t); 0.05*sin(12*t)]';
  X0 = [0.1 zeros(1,5)]'; <enter>
```

Then the M-file *march.m* is used with a first-order hold for greater accuracy (since the inputs are smoothly varying) as follows:

```
>>[yf,Xf] = march(A,B,C,D,X0,t,u,'foh'); <enter>
```

To see how much is the difference in the computed outputs if a less accurate zero-order hold is used, we re-compute the solution using *march.m* with '*zoh*' as the '*method*':

```
>>[yz,Xz] = march(A,B,C,D,X0,t,u,'zoh'); <enter>
```

The computed outputs, $y_1(t)$, and $y_2(t)$, are plotted in Figures 4.6 and 4.7, respectively, for the zero-order and first-order holds. It is observed in Figure 4.6 that the output $y_1(t)$ calculated using the first-order hold has slightly lower peaks when compared to that calculated using the zero-order hold. Figure 4.7 shows virtually no difference between the values of $y_2(t)$ calculated by zero-order and first-order holds.

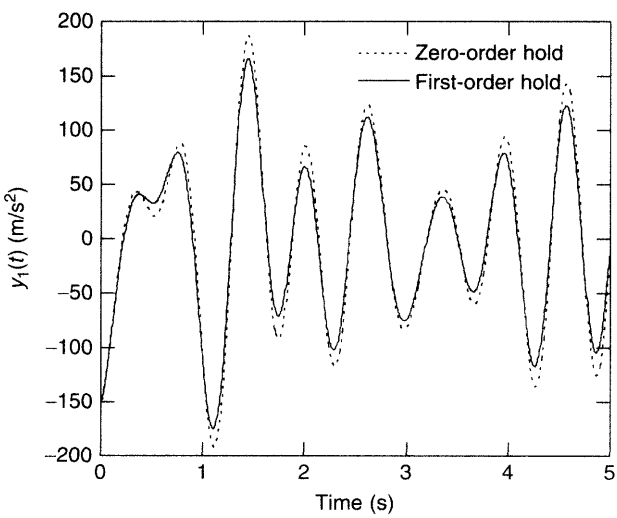

Figure 4.6 The normal acceleration output, $y_1(t)$, for the flexible bomber aircraft of Example 4.7

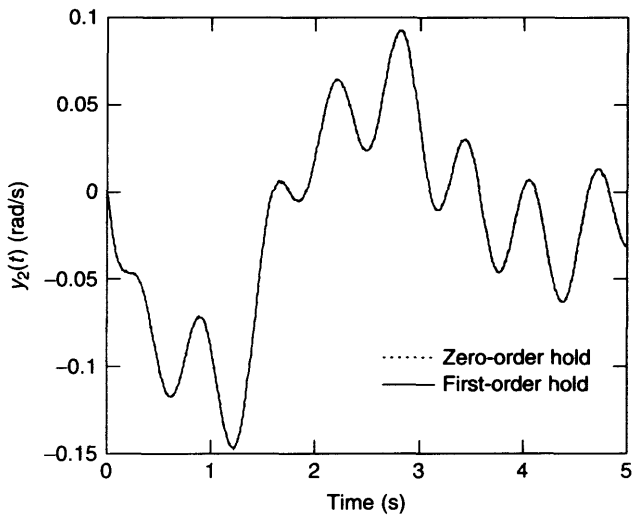

Figure 4.7 The pitch-rate output, $y_2(t)$, of the flexible bomber aircraft of Example 4.7

The MATLAB (CST) function *lsim* is an alternative to *march* for solving the state-equations by digital approximation, and is used as follows:

```
>>[y,t,x] = lsim(sys,u,t,x0,'method'); <enter>
```

where *sys* is an LTI object of the system, while the arguments u, t, 'method' (either '*zoh*', or '*foh*') and the returned output matrix, y, and state solution matrix, x, are defined in the same manner as in the M-file *march*. The user *need not* specify which interpolation method between *'zoh'* and *'foh'* has to be used in *lsim*. If a *'method'* is not specified, the function *lsim* checks the shape of the input, and applies a zero-order hold to the portions which have step-like changes, and the first-order hold to the portions which are smooth functions of time. In this regard, *lsim* is more efficient than *march*, since it optimizes the interpolation of the input, $\mathbf{u}(t)$, portion by portion, instead of applying a user specified interpolation to the entire input done by *march*. However, *lsim* can be confused if there are rapid changes between smooth and step-like portions of the input. Hence, there is a need for selecting a small time step for rapidly changing inputs in *lsim*.

Example 4.8

For the system in Example 4.7, compare the solutions obtained using the MATLAB (CST) command *lsim* and the M-file *march* when the initial condition is $\mathbf{x0} = [0.1;\ 0;\ 0;\ 0;\ 0;\ 0]^T$ when the elevator input, $u_1(t)$ is a rectangular pulse applied at $t = 0$ with amplitude 0.05 rad. and duration 0.05 s, while the canard input, $u_2(t)$, is a sawtooth pulse applied at $t = 0$ with amplitude 0.05 rad. and duration 0.05 s. The rectangular pulse and the sawtooth pulse are defined in Examples 2.5 and 2.7, respectively.

Mathematically, we can express the input vector as follows, using Eqs. (2.32) and (2.33):

$$\mathbf{u}(t) = \begin{bmatrix} 0.05[u_s(t+0.05) - u_s(t)] \\ r(t) - r(t-0.05) - 0.05u_s(t-0.05) \end{bmatrix} \tag{4.73}$$

where $u_s(t)$ and $r(t)$ are the unit step and unit ramp functions, respectively. Using MATLAB, the inputs are generated as follows:

```
>>dt=6.6667e-3; t=0:dt:5; u1=0.05*(t<0.05+dt); i=find(t<0.05+dt);
  u2=u1; u2(i)=t(i); u = [u1' u2']; <enter>
```

Assuming that **A, B, C, D, x0** are already available in the MATLAB workspace from Example 4.7, we can calculate the response of the system using *march* as follows:

```
>>[Y1,X1] = march(A,B,C,D,X0,t,u,'zoh'); <enter>
```

where **Y1** is the returned output matrix for zero-order hold. For comparison, the solution is also obtained using *lsim* and the output is stored in matrix **Y2** as follows:

```
>>sys=ss(A,B,C,D); [Y2,t,X2] = lsim(sys,u,t,X0); <enter>
```

The computed outputs, $y_1(t)$ and $y_2(t)$, by the two different methods are compared in Figures 4.8 and 4.9, respectively. Note that the responses calculated using *march* with zero-order hold and *lsim* are indistinguishable.

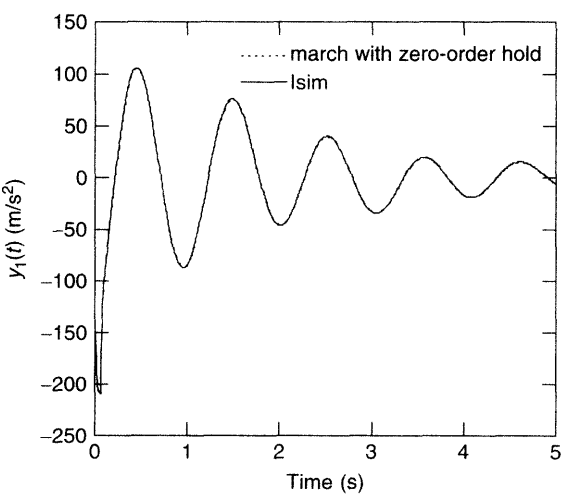

Figure 4.8 Normal acceleration output for the flexible bomber aircraft in Example 4.8 with rectangular pulse elevator input and sawtooth pulse canard input

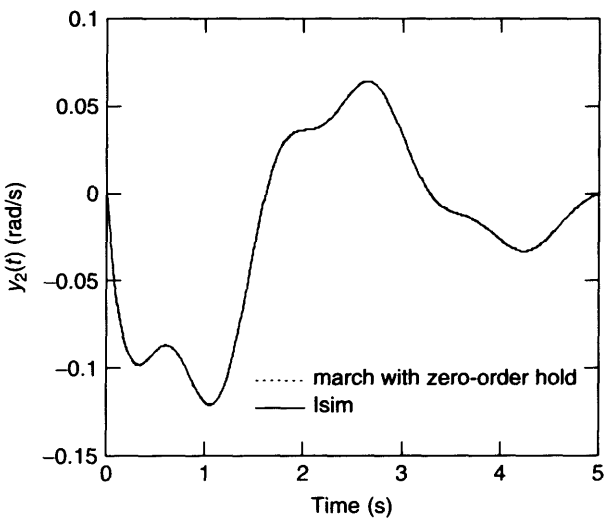

Figure 4.9 Pitch-rate output for the flexible bomber aircraft in Example 4.8 with rectangular pulse elevator input and sawtooth pulse canard input

When the inputs are impulse functions, the M-files *march* and *lsim* cannot be used directly, because it is impossible to describe the impulse function (which, by definition, goes to infinity in almost zero time) by an input vector. Instead, the digital approximation is obtained using an equation similar to Eq. (4.68), which gives the solution if all the inputs are unit impulse functions applied at $t = t_0$. For a more general case, i.e. when the input vector is given by $\mathbf{u}(t) = \delta(t - t_0)[\, c_1; \quad c_2; \quad \ldots; \quad c_m \,]^T$, where $c_1, c_2, \ldots, c_m$ are constants, we can write the solution to the state-equation as follows:

$$\mathbf{x}(t_0 + n\Delta t) = \mathrm{e}^{\mathbf{A}\Delta t}[\mathbf{x}(t_0 + (n-1)\Delta t) + \mathbf{B}\mathbf{c}\Delta t]; \quad (n = 1, 2, 3, \ldots) \tag{4.74}$$

where $\mathbf{c} = [\, c_1; \quad c_2; \quad \ldots; \quad c_m \,]^T$. Comparing Eqs. (4.68) and (4.74), we find that the digital approximation for impulse inputs is given by $\mathbf{A}_\mathrm{d} = \mathrm{e}^{\mathbf{A}\Delta t}$ and $\mathbf{B}_\mathrm{d} = \mathrm{e}^{\mathbf{A}\Delta t}\mathbf{B}\Delta t$, if and only if the input vector is given by $\mathbf{u}(t) = u_s(t)\mathbf{c}$. For calculating the response to general impulse inputs of this type applied at $t = 0$, we can write a MATLAB M-file in a manner similar to *march.m*. Such an M-file, called *impgen.m* is given in Table 4.3. MATLAB (CST) does have a standard M-file for calculating the impulse response called *impulse.m*, which, however, is limited to the special case when *all* of the imputs are simultaneous *unit* impulses, i.e. all elements of the vector **c** are equal to 1. Clearly, *impgen* is more versatile than *impulse*. (MATLAB (CST) also has a dedicated function for calculating the step response, called *step*, which also considers *all inputs to be simultaneous unit step functions*.) The advantage of using the MATLAB command *impulse* (and *step*) lies in quickly checking a new control design, without having to generate the time vector, because the time vector is automatically generated. Also, for systems that are *not* strictly proper (i.e. $\mathbf{D} \neq \mathbf{0}$) the CST function *impulse* disregards the impulse in the response at

Table 4.3 Listing of the M-file *impgen.m*

impgen.m

```
function [y,X] = impgen(A,B,C,D,X0,t,c)
% Time-marching solution of linear, time-invariant
% state-space equations using the digital approximation when the
% inputs are impulse functions scaled by constants. The scaling
% constants for the impulse inputs are contained in vector 'c'.
% A= state dynamics matrix; B= state input coefficient matrix;
% C= state output coefficient matrix;
% D= direct transmission matrix;
% X0= initial state vector; t= time vector.
% y= returned output matrix with ith output stored in the ith
% column, and jth row corresponding to the jth time point.
% X= returned state matrix with ith state variable stored in the
% ith column, and jth row corresponding to the jth time point.
% copyright(c)2000 by Ashish Tewari
n=size(t,2);
m=size(c,2);
dt=t(2)-t(1);
% digital approximation of the continuous-time system:-
[ad,bd,cd,dd]=c2dm(A,B,C,D,dt,'zoh');
Bd=ad*bd;
u=ones(n,1)*c';
% time-marching solution of the digital state equation:-
X=ltitr(ad,Bd,u,X0);
% calculation of the outputs:-
y=X*C'+u*D';
```

$t = 0$ (see Eq. (2.115)). For details on the usage of these specialized CST functions, use the MATLAB *help* command.

Example 4.9

For the flexible bomber aircraft of Example 4.7, let us determine the response if the initial condition is $\mathbf{x0} = [\,0.1;\quad 0;\quad 0;\quad 0;\quad 0;\quad 0\,]^T$ and the input vector is given by:

$$\mathbf{u}(t) = \begin{bmatrix} 0.1\delta(t) \\ -0.1\delta(t) \end{bmatrix} \tag{4.75}$$

First, the time vector, **t**, and the coefficient vector, **c**, are specified as follows:

```
>>dt=6.6667e-3; t=0:dt:10; c=[0.1; -0.1]; <enter>
```

Then, *impgen* is invoked as follows, assuming **A, B, C, D, x0** have been already computed and stored in the MATLAB workspace: