BACKINTERPOLATION METHODS FOR THE NUMERICAL SOLUTION OF ORDINARY DIFFERENTIAL EQUATIONS AND APPLICATIONS

by

Wei Xie

Wei Xie

A Thesis Submitted to the Faculty of the

DEPARTMENT OF SYSTEMS AND INDUSTRIAL ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE
WITH MAJOR IN INDUSTRIAL ENGINEERING

In the Graduate College

THE UNIVERSITY OF ARIZONA

1995

BACKINTERPOLATION METHODS FOR THE NUMERICAL SOLUTION OF ORDINARY DIFFERENTIAL EQUATIONS AND APPLICATIONS

by

Wei Xie

An Abstract of a Thesis Submitted to the Graduate

College of University of Arizona

in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Major Subject: Electrical and Computer Engineering

The original of the complete thesis is on
file in the University of Arizona
Science Library

Approved by the
Examining Committee:

Francois E. Cellier

Hal S. Tharp

Xue Xin

University of Arizona
Tucson, Arizona

May 1995

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED:_____

ACKNOWLEDGMENTS

I wish to express my gratitude and appreciation to my thesis advisor, Professor Francois E. Cellier, for his guidance and encouragement through the course of my three-year MS study at the University of Arizona. I also want to thank Professors Hal S. Tharp and Xue Xin serving on my thesis committee.

I wish to express my love and affection to my daughter Lin and my husband Fei-Yue. Words are not enough to express my deep gratitude for all the love, support, and understanding you have given me.

ABSTRACT

This thesis presents backinterpolation (BI) methods for the numerical solution of ordinary differential equations (ODE) and their applications. Compared with conventional numerical integration algorithms, BI methods are more effective in handling marginally stable and stiff problems. A detailed analysis of various properties of BI methods has been conducted. Stability properties of BI methods have been investigated and a method for calculating the stability domain of a BI algorithm has been developed. Issues related to accuracy considerations of BI algorithms have been addressed. Procedures for constructing accuracy domains for BI methods have been given. A study of damping and frequency properties of BI algorithms has been conducted. Programs for comparing analytical and discrete damping and frequency along an arbitrary axis (not just the real axis) have been specified. A scheme of stepsize control for BI methods has been proposed. Two algorithms of adjusting stepsizes have been evaluated through simulations.

To demonstrate the effectiveness and efficiency of BI methods in solving marginally stable and stiff problems, ODE models for dynamic responses of one-link flexible manipulators have been developed and solved using BI methods. For both open-loop (marginally stable) and closed-loop (stiff) configurations, numerical results have demonstrated the effectiveness of BI methods for solving marginally stable and stiff problems.

Both Matlab and C codes for analysis of BI methods and their application for finding numerical solutions for flexible manipulators have been developed.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

1.1 Motivation and Objective

1.1.1 Why We Need an ODE Solver ?

As a mathematical form, the ordinary differential equation (ODE) is a very important tool. It is used in the modeling of a wide variety of physical phenomena– chemical reactions, satellite orbits, vibrating or oscillating systems, electrical networks, and so on. In many cases, the independent variable represents time so that the differential equation describes changes, with respect to time, in the system being modeled. The solution of the equation will be a representation of the state of the system at any point in time and one can use it to study the behavior of the system. Consequently, the problem of finding the solution of a differential equation plays a significant role in scientific research, particularly, in the simulation of physical phenomena [15].

Simulation is a powerful alternative to practical experiments. It provides a fast, convenient, and economic tool for system design, assessment, analysis, and evaluation. However, it is usually impossible to obtain a closed form solution of differential equations for systems to be modeled, especially complex ones encountered in real-world problems. A simulation program therefore has to implement some numerical integration method to solve equations that describe system behavior. Since most of these equations are, or can be approximated by, ODEs, a fast, accurate, and efficient ODE solver is much needed.

1.1.2 Why We Need a BI-based ODE Solver ?

Numerous ODE solvers have been developed and implemented since the digital computer was introduced four decades ago. Among them, Runge-Kutta (RK) Methods are probably the most popular and widely used ones. Many commercial simulation packages use RK-based ODE solvers, e.g., Matlab. Therefore, one must ask why do we need another ODE solver, especially, the backinterpolation (BI) techniques studied in this thesis ?

To answer this question, we need to know the criteria used to assess an ODE solver. To this end, we use three criteria, namely, that an ODE solver should

1. Give an accurate approximation;

2. Use less computation time;

3. Be easy to implement.

In addition, it must obtain a unique (numerical) solution.

Most conventional ODE solvers, including RK methods, are easy to implement and also meet the first two criteria in normal circumstances. However, they become much less efficient and accurate when dealing with either marginally stable problems or stiff problems, especially when the number of state variables involved is large. In terms of eigenvalues, a system is marginally stable if many of its eigenvalues are on the imaginary axis of the complex plane. A system is stiff if its eigenvalues are widely spread over the complex plane, i.e., the ratio of its largest and smallest eigenvalues is very large. But a wide range of physical phenomena, such as structural vibrations, chemical processes, feedback control, and so on, are described by marginally stable or stiff systems.

Therefore, we need new ODE solvers that can handle marginally stable and stiff problems effectively. Our previous study has indicated that BI-based methods are a good selection for this purpose [5]. *The objective of this thesis is to conduct a detailed analysis of various properties of BI methods, and to demonstrate their effectiveness and efficiency in solving marginally stable and stiff problems through applications.*

1.2 Contributions

The following major results have been accomplished in this thesis for the analysis and application of backinterpolation methods for numerical solutions of ordinary differential equations:

1. Stability properties of BI methods have been investigated and a method for calculating the stability domain of a BI algorithm has been developed. Detailed numerical results of stability domains for BI45 and BI55 have been obtained.

2. Issues related to accuracy considerations of BI algorithms have been addressed. Procedures of constructing accuracy domains for BI methods have been given. Numerical examples of accuracy domains for BI45 and BI55 have been presented.

3. A study of damping and frequency properties of BI algorithms has been conducted. Programs for comparing analytical and discrete damping and frequency along an arbitrary axis (not just the real axis) have been specified. Detailed numerical results for BI45 and BI55 have been given.

4. A scheme of stepsize control for BI methods has been proposed. Two algorithms of adjusting stepsizes have been evaluated through simulations. A

numerical comparison of the BI45 algorithm and the RK-based *ode45* implemented in Matlab has been made with a simple marginally stable problem and a stiff problem. The results have shown that BI methods are indeed more effective in solving these two classes of problems.

5. ODE models for solving dynamic responses of one-link flexible manipulators have been developed using both Euler-Bernoulli and Timoshenko beam theories. Original partial differential equations of flexible manipulators have been derived and approximated by ODE using the method of lines. Numerical results have indicated clearly that modeling of flexible manipulators has provided an ideal application for testing BI methods with real-world complicated problems. This is due to the fact that the ODE system of flexible manipulators is marginally stable in the open loop simulation and becomes strongly stiff in the closed loop simulation. Both BI45 and *ode45* have been used and compared in obtaining numerical dynamic responses of flexible manipulators. Again, numerical results have demonstrated the effectiveness of BI methods for solving marginally stable and stiff problems.

In addition, Matlab and C codes for analysis of BI methods and their application for finding numerical solutions for flexible manipulators have been developed in this thesis.

1.3 Organization of This Thesis

In Chapter 2, we discuss briefly the principles and properties of numerical ODE solvers and introduce the concepts and procedures of BI methods.

A detailed analysis of various aspects of BI methods is given in Chapter 3. Procedures of constructing stability domains and accuracy domains are developed. Discrete damping and frequency are calculated and compared with analytical ones. Algorithms for controlling the stepsize in BI methods are proposed.

Chapter 4 presents a detailed procedure of developing ODE models for one-link flexible manipulators based on the Euler-Bernoulli beam theory for link deflection. Numerical simulations are conducted for solving dynamic responses in both the open and closed-loop cases. To take the effect of link shear deformation into account, the results presented in the chapter are generalized to the case of a Timoshenko flexible manipulator model in Chapter 5.

Chapter 6 concludes the thesis with a summary and suggestions for future work.

Finally, Matlab and C codes developed in this thesis are given in Appendix A. Appendix B provides a C-code implementation of the BI methods.

CHAPTER 2

Backinterpolation Methods for Numerical Solutions of ODE Systems

This chapter introduces backinterpolation techniques for finding numerical solutions of continuous-time differential equations [5]. To explain basic concepts and methods in numerical techniques, we start our discussion in Section 2.1 with the simplest numerical integration schemes: the first-order Forward and Backward Euler algorithms. In Section 2.2 we introduce the concept of stability of numerical integration algorithms and give several stability definitions. Section 2.3 presents the motivation for backinterpolation methods and the basic ideas for their construction schemes. Explicit forms of various F-stable and L-stable backinterpolation algorithms are described in Sections 2.4 and 2.5. These algorithms provide the background material for analysis and evaluation of backinterpolation in the next chapter. Finally, Section 2.6 summarizes the discussion of this chapter.

2.1 ODE Solvers: Forward and Backward Euler Methods

Consider a linear ordinary differential equation (ODE) of n state variables,

$$\dot{x} = A \cdot x, \qquad x(t_0) = x_0, \qquad t_0 \leq t \leq t_f \tag{2.1}$$

where A, called the *system matrix*, is of size $n \times n$, x_0 is the initial state, and t_0, t_f are initial and final times, respectively.

The simplest way of finding the numerical solution of Eq. (2.1) is by approximating the time derivative with numerical differences. To this end, we discretize the time interval $[t_0, t_f]$ uniformly by stepsize h and denote the state vector at time instance $t_k = t_0 + k \cdot h$ by x_k, i.e., $x_k = x(t_k)$.

Now, approximating $\dot{x}$ by the left difference, we get,

$$\dot{x}(t_k) = \frac{x_{k+1} - x_k}{h}$$

and, correspondingly, we convert the ODE (2.1) into the following difference equation,

$$x_{k+1} = [I + A \cdot h]x_k \tag{2.2}$$

where I is the $n \times n$ identity matrix. Eq. (2.2) is called the *Forward Euler Method* [6]. Integration schemes similar to this method are called *explicit type.*

Alternatively, approximating $\dot{x}$ by the right difference, we get,

$$\dot{x}(t_{k+1}) = \frac{x_{k+1} - x_k}{h}$$

and, correspondingly, we convert the ODE (2.1) into the following difference equation,

$$x_k = [I - A \cdot h]x_{k+1} \text{ or } x_{k+1} = [I - A \cdot h]^{-1}x_k \tag{2.3}$$

which is called the *Backward Euler Method* [6]. Integration schemes similar to this method are called *implicit type.*

Eqs. (2.2) and (2.3) are the two simplest numerical integration schemes ever developed. These two methods are not widely used as numerical ODE solvers because of their low accuracy. Generally speaking, a numerical ODE solver converts the original linear continuous-time system (2.1) into an equivalent discrete-time system,

$$x_{k+1} = Fx_k, \qquad k = 0, 1, 2, \cdots \tag{2.4}$$

where F is a system matrix generated by the integration scheme used by the ODE solver. For example, explicit Runge-Kutta algorithms of orders 2 to 4 are characterized by the following matrices,

$$F_2 = I + A \cdot h + \frac{(A \cdot h)^2}{2!}$$

$$F_3 = I + A \cdot h + \frac{(A \cdot h)^2}{2!} + \frac{(A \cdot h)^3}{3!}$$

$$F_4 = I + A \cdot h + \frac{(A \cdot h)^2}{2!} + \frac{(A \cdot h)^3}{3!} + \frac{(A \cdot h)^4}{4!}$$

and the corresponding implicit Runge-Kutta algorithms are characterized by,

$$F_2 = \left[I - A \cdot h + \frac{(A \cdot h)^2}{2!}\right]^{-1}$$

$$F_3 = \left[I - A \cdot h + \frac{(A \cdot h)^2}{2!} - \frac{(A \cdot h)^3}{3!} \right]^{-1}$$

$$F_4 = \left[I - A \cdot h + \frac{(A \cdot h)^2}{2!} - \frac{(A \cdot h)^3}{3!} + \frac{(A \cdot h)^4}{4!} \right]^{-1}$$

Clearly, once F is given, difference equation (2.4) can be solved easily by means of finite iteration starting with the specified initial condition x_0.

2.2 Numerical Stability of Integration Schemes

The most important property of an ODE solver is its numerical stability. Let λ represent an eigenvalue of the original linear continuous-time system, i.e., an eigenvalue of matrix A, then each ODE solver is characterized by a numerical stability domain in the $\lambda \cdot h$ plane. All eigenvalues λ_i of an analytically stable linear continuous-time system (i.e., the real parts of all λ_i are negative) multiplied by the integration time step h must lie within the numerical stability domain. Otherwise, the integration scheme used is numerically unstable, and thus may produce incorrect numerical results.

To see this, consider a simple scalar equation $\dot{x} = ax$, where a is a scalar. Figs. 2.1 and 2.2 present the numerical stability domains of the Forward and Backward Euler schemes. A detailed procedure of finding the stability domain for a numerical integration scheme is described in Section 3.1 of Chapter 3. Using the Forward Euler scheme, in order to make $a \cdot h$ lie within the numerical stability domain, we

must choose the integration step size h such that

$$|1 + ah| < 1 \tag{2.5}$$

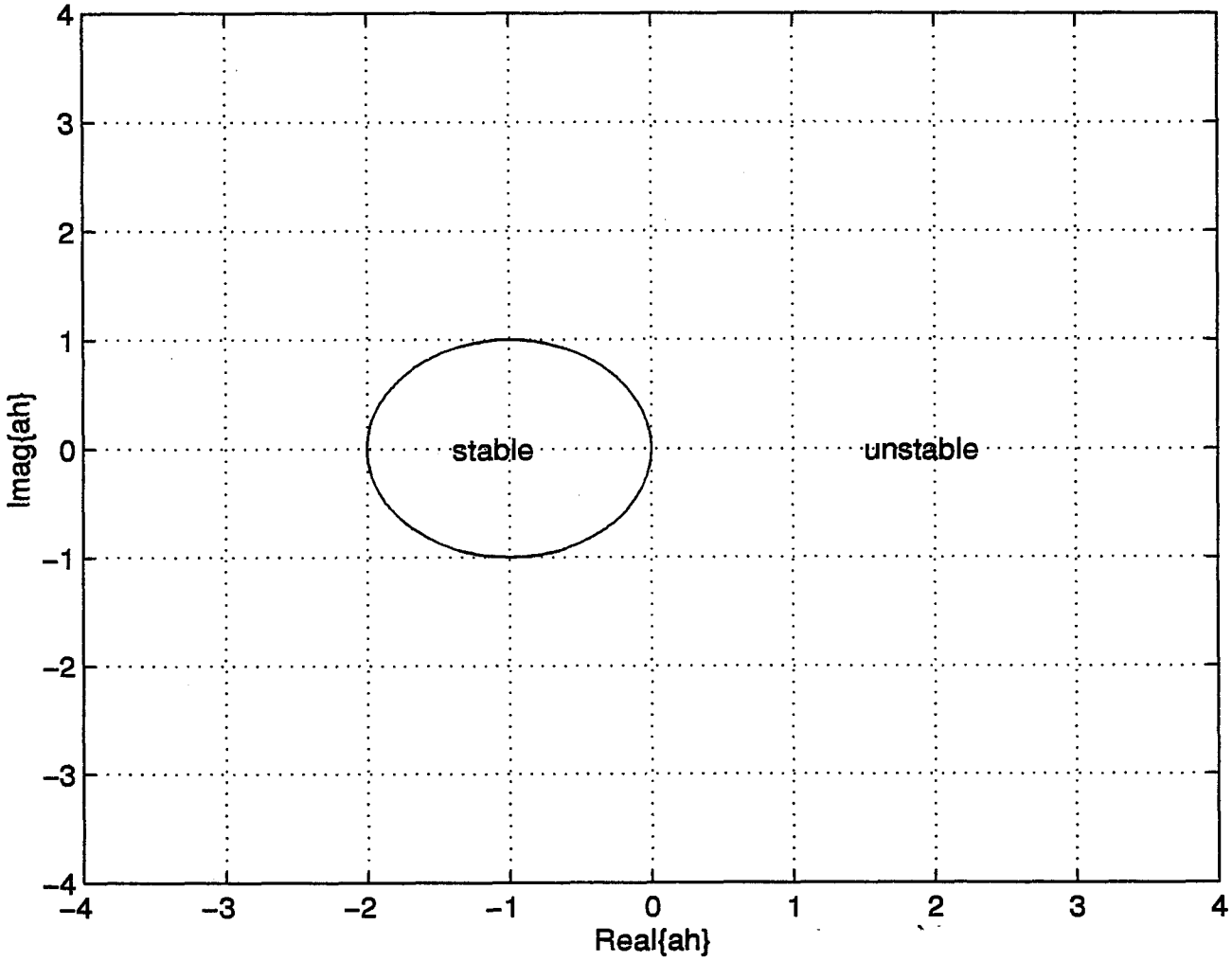

Figure 2.1: Stability Domain of Forward Euler Method

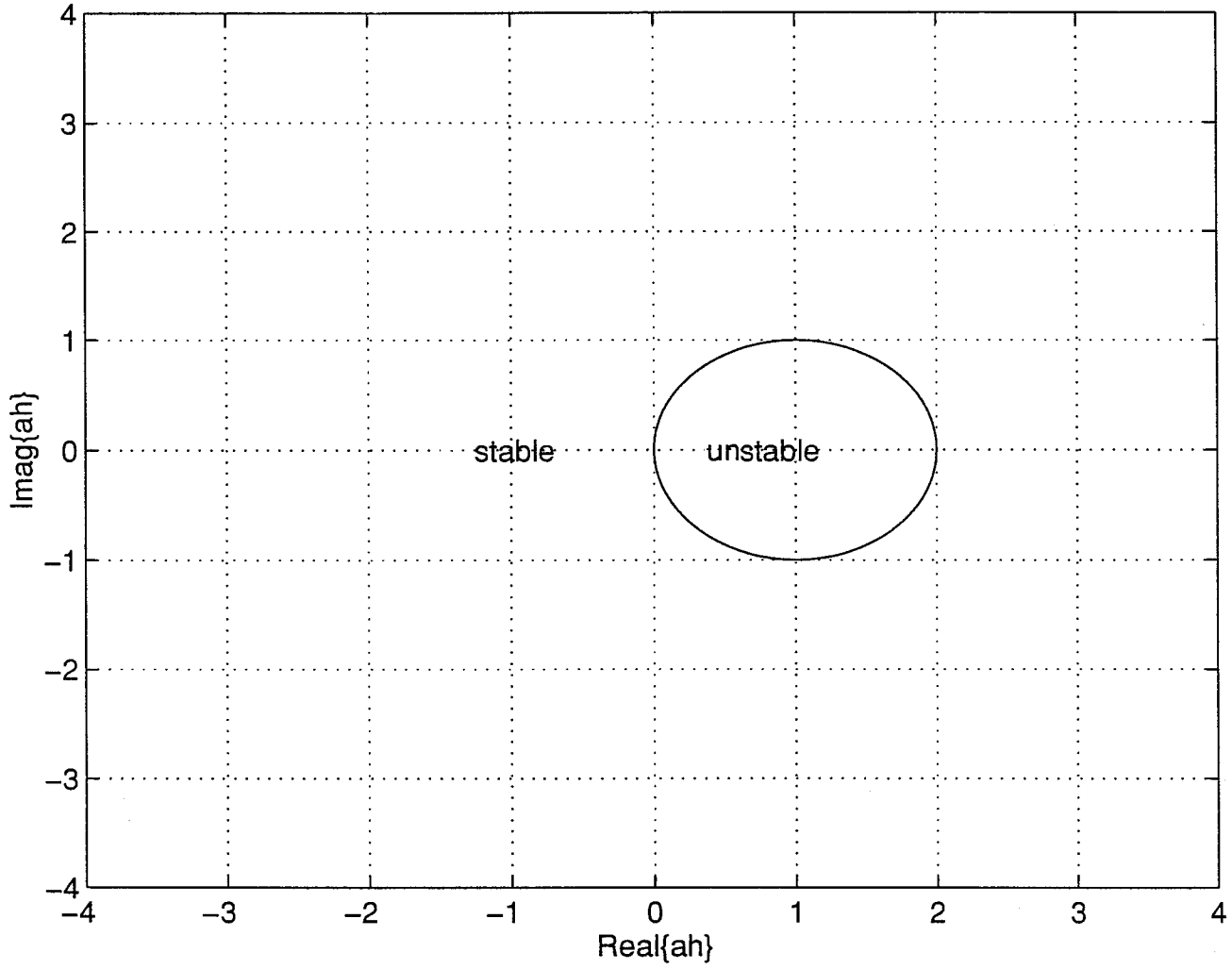

Figure 2.2: Stability Domain of Backward Euler Method

Fig. 2.3 shows the comparison of the analytical solution and the two numerical solutions obtained by the Forward Euler method using $h = 0.05$ and $h = 0.5$ for $a = -5$. Initial condition $x_0 = 1$ is assumed. The original continuous-time system is stable in this case. Clearly, when $h = 0.05$ is used, $a \cdot h$ lies within

To clarify these situations, we introduce three additional definitions of numerical stability: A-stability, F-stability and L-stability.

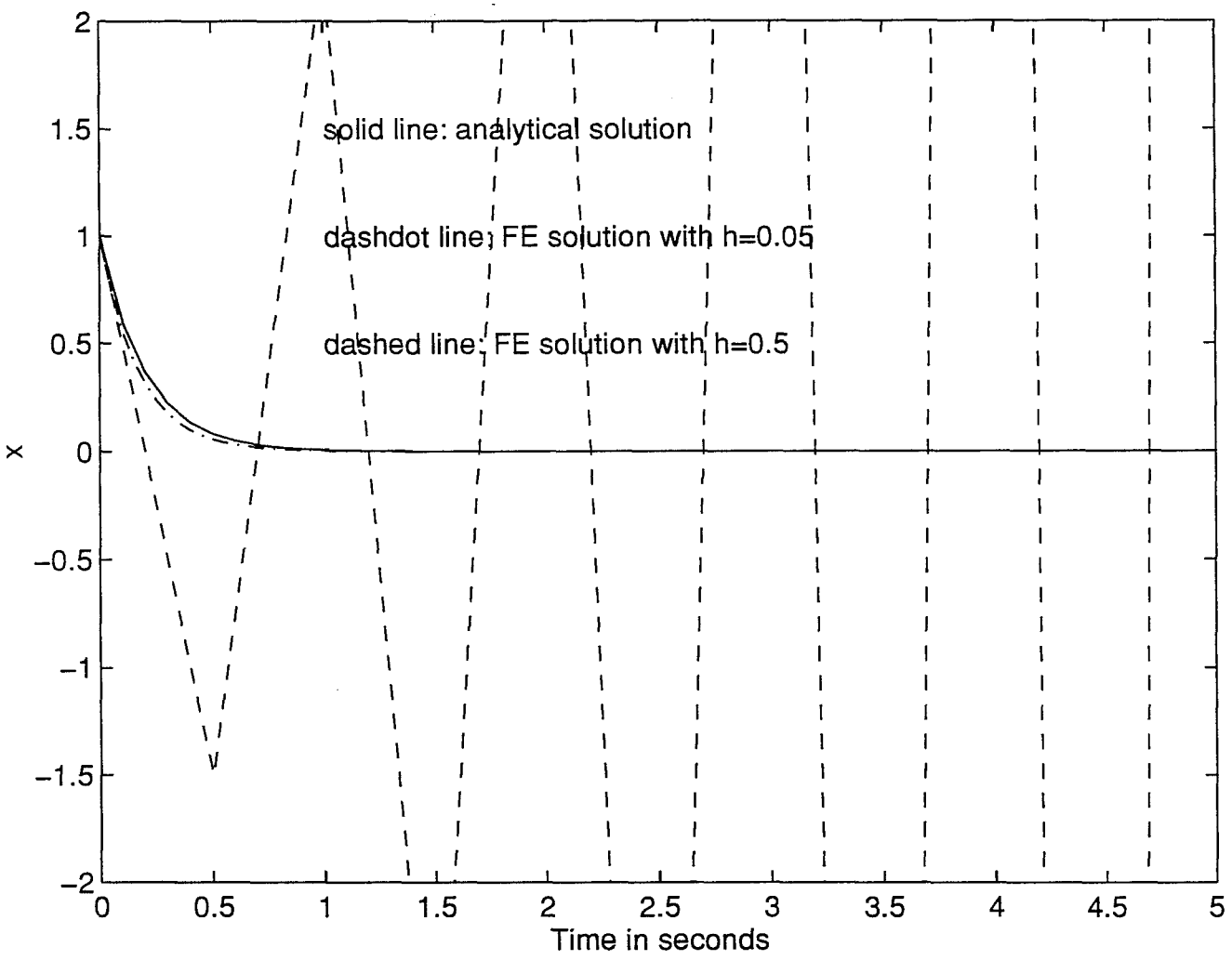

Figure 2.3: Numerical Solutions by Forward Euler Method

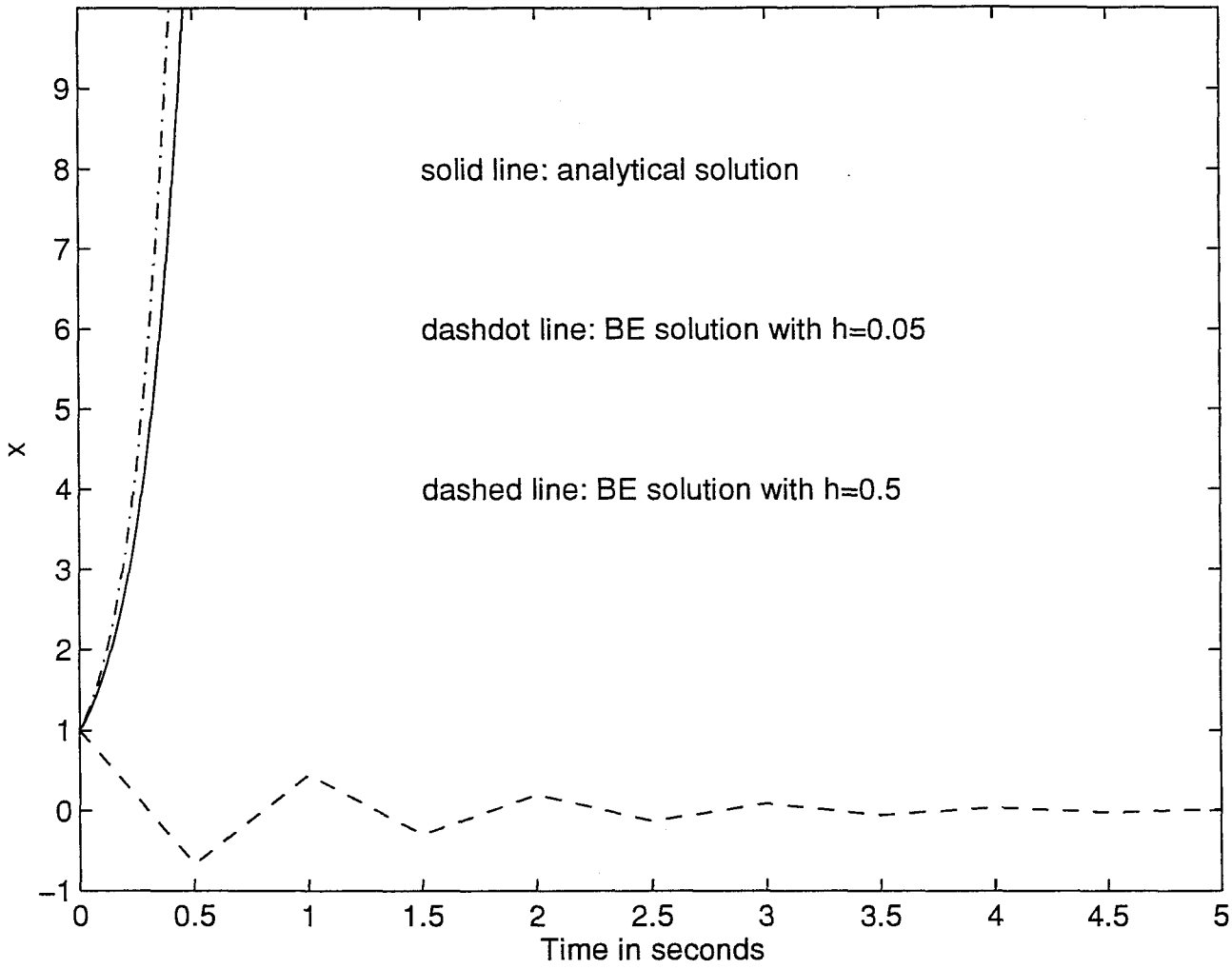

Figure 2.4: Numerical Solutions by Backward Euler Method

Definition 2.1: A numerical integration scheme that contains the entire left half $(\lambda \cdot h)$-plane as part of its numerical stability domain is called *absolute stable*, or, more simply, *A-stable.*

Clearly, the Backward Euler method is A-stable, while Forward Euler is not.

> **Definition 2.2:** A numerical integration scheme that contains the entire left half $(\lambda \cdot h)$-plane and nothing else as its numerical stability domain is called *faithfully stable*, or, more simply, *F-stable.*

Neither the Forward nor Backward Euler method is F-stable.

> **Definition 2.3:** A numerical integration scheme that is A-stable, and, in addition, whose damping properties increase to infinity as $\Re\{\lambda\} \rightarrow -\infty$, is called *L-stable.*

The Backward Euler method is L-stable, whereas the Forward Euler method is not. More discussion about L-stability is given in the next chapter.

When dealing with stiff systems, it is not sufficient to demand A-stability from the integration algorithm. We truly need an L-stable algorithm. Evidently, F-stable algorithms are also A-stable, but never L-stable.

2.3 Backinterpolation Methods: Motivation and Basic Ideas

In many engineering problems, we have to solve partial differential equations (PDEs) by converting them to ODEs. However, many of these PDEs, such as wave equations and flexible manipulator equations, are of the hyperbolic type, which will lead to ODEs that are marginally stable and whose solutions exhibit undamped oscillations since all their eigenvalues are entirely located on the imaginary axis. This

has presented a serious problem to many conventional ODE solvers: they transfer this type of ODEs into either unstable numerical systems or stable numerical systems with damped oscillation responses.

To see this, consider the simplest case of a harmonic oscillator equation, $\ddot{y}+\omega^2 y = 0$, or in state variable form, $x_1 = y$ and $x_2 = \dot{y}$,

$$\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} = \begin{bmatrix} 0 & 1 \\ -\omega^2 & 0 \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

Using the Forward Euler algorithm to solve the discretized oscillator equation, both eigenvalues will always be outside the numerically stable domain however small the time step h is chosen, and thus, the numerical solution is unstable. Using the Backward Euler algorithm, both eigenvalues will always be inside the numerically stable region however small the time step h is used, and thus, the numerical solution shows damped oscillations.

Fig. 2.5 presents specific numerical solutions obtained by the two types of algorithms (F_4 is used in both cases), in which $\omega = 10, x_1(0) = 1, x_2 = 0$, and $h = 0.01$ is used for both the forward and the backward algorithms. Clearly, in this case, both algorithms give unacceptable numerical solutions for the simple oscillator equation.

Generally, all explicit integration algorithms give numerical solutions that resemble those of the Forward Euler algorithm, and some implicit integration algorithms obtain numerical solutions that are similar to those of the Backward Euler algorithm, whereas others behave like Forward Euler. Thus, new integration schemes

must be developed to find accurate numerical solutions for marginally stable ODE systems. This was the main motivation for the development of the backinterpolation algorithms.

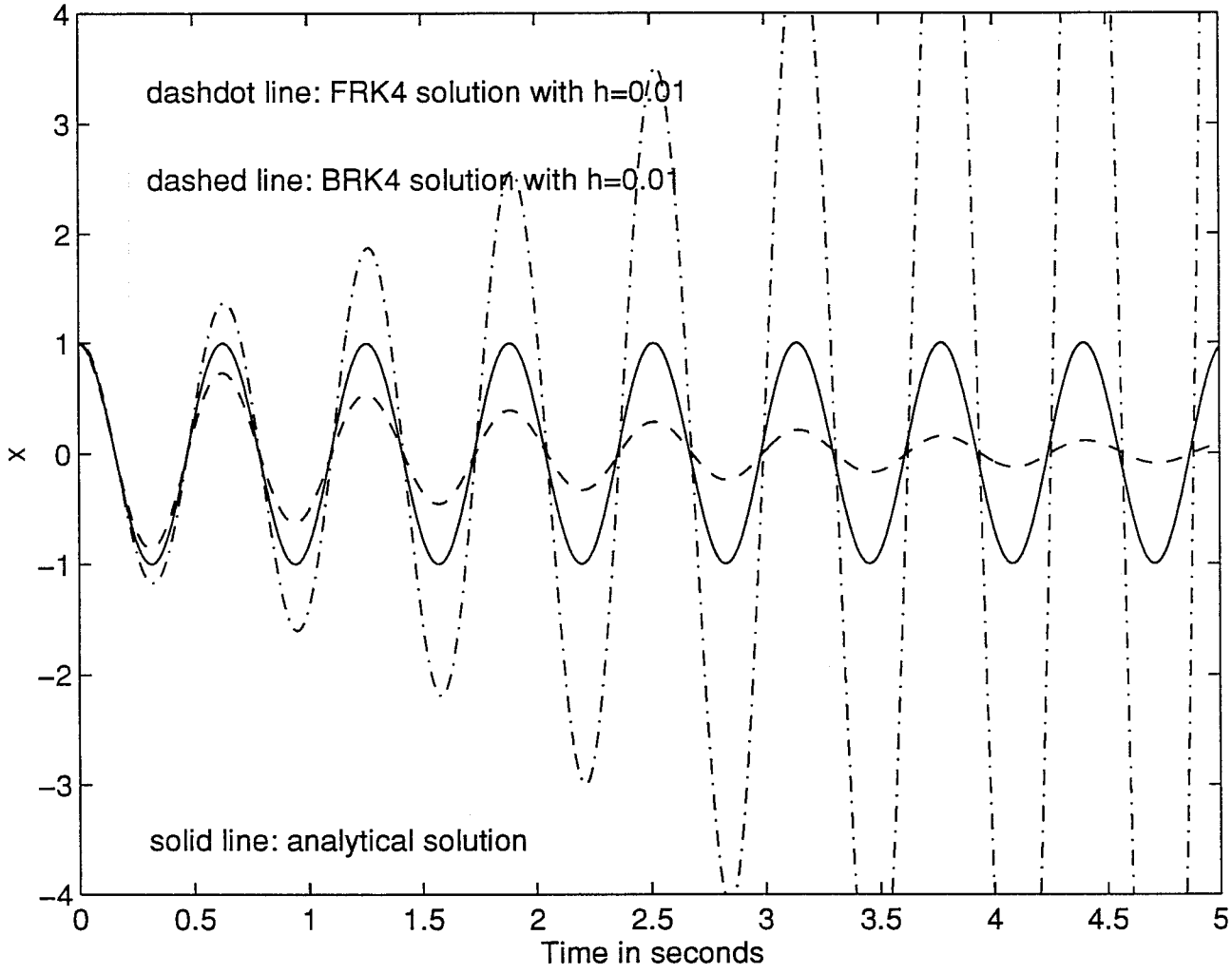

Figure 2.5: Marginally Stable Problem: Runge-Kutta Method

Actually, the basic idea behind the backinterpolation methods is quite simple. From Eq. (2.3), it is very clear that the F matrix of the Backward Euler method can be realized by integrating backward through time from t_{k+1} to time t_k with a negative step size of $-h$. This observation gives rise to a straightforward implementation of backinterpolation algorithms: while the overall integration proceeds *forward* through time from initial time t_0 to final time t_f, each individual step is integrated *backward* through time from time t_{k+1} to time t_k.

Specifically, a backinterpolation algorithm starts with the known initial state, x_0. This state is projected forward to t_1, i.e.,

$$x_1^P = x_0 \tag{2.7}$$

Using this predictor for x_1, the algorithm proceeds by integrating one step of size $-h$ backward through time. This generates a new value for the initial state, x_0^P. At this point, a Newton iteration can be applied to iterate on the (unknown) "initial" state x_1, until the (known) "final" state x_0 is hit with sufficient accuracy. For a linear system, no iteration is needed since x_1 can be found immediately by matrix inversion, which leads to the Backward Euler algorithm.

The last value of the "initial" state is then taken as the final value of the first step of the overall integration. It is again projected forward through time and used as the first estimate of the next step, x_2^P.

Clearly, each backward integration step can be performed with any explicit algorithm of arbitrary order. The overall algorithm belongs to a special kind of implicit algorithms.

Now the question is, how can we use the backinterpolation techniques to construct numerical integration schemes that avoid the drawbacks of the conventional explicit and implicit integrations method when dealing with marginally or stiff ODE systems ? We shall discuss these issues in the next two sections.

2.4 F-Stable Backinterpolation Algorithms

An F-stable BI algorithm can be generated by conducting the one-step integration from time t_k to time t_{k+1} in two half-steps. The first half-step is a forward step, whereas the second half-step is a backward step.

Specifically, let's use the FE as the integration scheme. Starting with the initial state x_0, two steps of FE of length $h/2$ are performed. The first step produces the *true* state $x_{1/2}$, the second step gives the estimate x_1^P. The algorithm proceeds by integrating backward from the (unknown) "initial" state x_1^P to the (known) "final" state $x_{1/2}$ using a step size of -$h/2$. Newton iteration can be applied to the unknown "initial" state, until the known "final" state is hit with sufficient accuracy. The last value of the "initial" state is then taken as the final value of the first step of the overall integration. The process continues to the next integration step.

For a linear system, the backward half-step can be considered as a BE step. Due to the symmetry between the stability domains of the FE and BE, the resulting integration scheme is of higher order and F-stable [7].

Higher order Runge-Kutta algorithms can also be used in this process. For a linear systems, the F matrices of the resulting F-stable algorithms are listed as follows,

$$F_1 = \left[I - A\frac{h}{2}\right]^{-1}\left[I + A\frac{h}{2}\right] \tag{2.8}$$

$$F_2 = \left[I - A\frac{h}{2} + \frac{(Ah)^2}{8}\right]^{-1}\left[I + A\frac{h}{2} + \frac{(Ah)^2}{8}\right] \tag{2.9}$$

$$F_3 = \left[I - A\frac{h}{2} + \frac{(Ah)^2}{8} - \frac{(Ah)^3}{48}\right]^{-1}\left[I + A\frac{h}{2} + \frac{(Ah)^2}{8} + \frac{(Ah)^3}{48}\right] \tag{2.10}$$

$$F_4 = \left[I - A\frac{h}{2} + \frac{(Ah)^2}{8} - \frac{(Ah)^3}{48} + \frac{(Ah)^4}{384}\right]^{-1} \left[I + A\frac{h}{2} + \frac{(Ah)^2}{8} + \frac{(Ah)^3}{48} + \frac{(Ah)^4}{384}\right] \tag{2.11}$$

Obviously, F_1 is also the F matrix of the trapezoidal rule [6], which has second order accuracy. Since F_2 is also of order 2 accurate, but requires more computation, it is not very useful. One can show easily that all these matrices are F-stable.

Fig. 2.6 presents specific numerical solutions for the harmonic oscillator using F_1 and F_4, where $h = 0.05$ is used for both algorithms. Clearly, both algorithms give acceptable numerical solutions for the simple oscillator equation. Note that when $h = 0.01$ is used, the error from either algorithm is not visible.

F-stable BI matrices can also be generated using other methods. Note that every numerical ODE solver tries to somehow approximate the true F matrix which is,

$$F = \exp(A \cdot h) \tag{2.12}$$

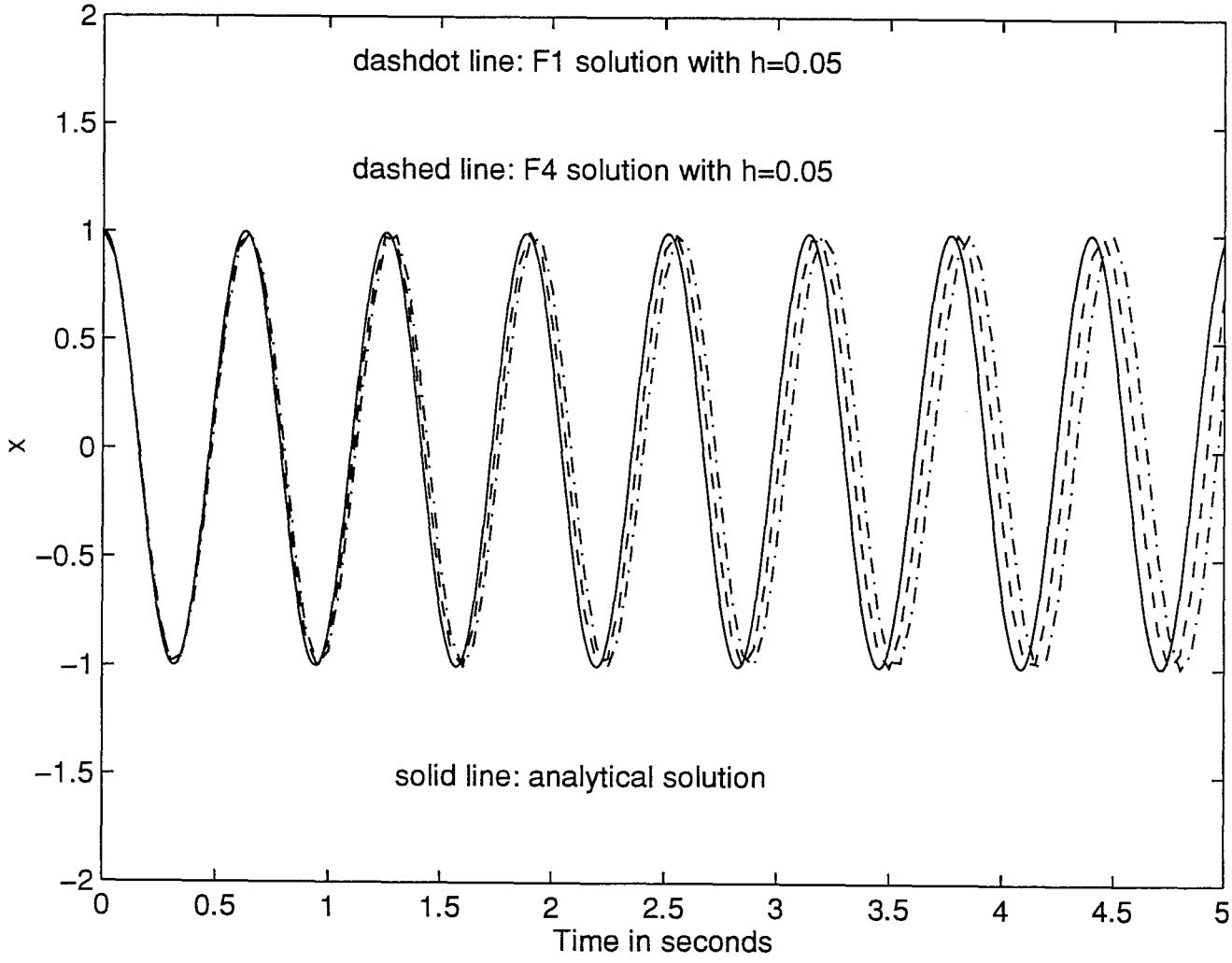

Figure 2.6: Marginally Stable Problem: BI Method

The formulae presented in Eqs. (2.8-2.11) look very similar to Pade approximants of a matrix exponential. To see this, we rewrite Eq. (2.12) as,

$$F = \exp(A\frac{h}{2}) \cdot \exp(A\frac{h}{2}) = \left[\exp(A(-\frac{h}{2}))\right]^{-1} \cdot \exp(A\frac{h}{2}). \tag{2.13}$$

According to [17], this can be approximated by,

$$F \approx D(p,q)^{-1} \cdot N(p,q) \tag{2.14}$$

where,

$$D(p,q) = \sum_{j=0}^{q} \frac{(p+q-j)!q!}{(p+q)!j!(q-j)!}(-Ah)^j, N(p,q) = \sum_{j=0}^{p} \frac{(p+q-j)!p!}{(p+q)!j!(p-j)!}(Ah)^j \tag{2.15}$$

which, for $p = q$, leads to the following set of F matrices,

$$F_2 = \left[I - A\frac{h}{2}\right]^{-1}\left[I + A\frac{h}{2}\right] \tag{2.16}$$

$$F_4 = \left[I - A\frac{h}{2} + \frac{(Ah)^2}{12}\right]^{-1}\left[I + A\cdot\frac{h}{2} + \frac{(Ah)^2}{12}\right] \tag{2.17}$$

$$F_6 = \left[I - A\frac{h}{2} + \frac{(Ah)^2}{10} - \frac{(Ah)^3}{120}\right]^{-1}\left[I + A\frac{h}{2} + \frac{(Ah)^2}{10} + \frac{(Ah)^3}{120}\right] \tag{2.18}$$

$$F_8 = \left[I - A\frac{h}{2} + \frac{3(Ah)^2}{28} - \frac{(Ah)^3}{84} + \frac{(Ah)^4}{1680}\right]^{-1} \left[I + A\frac{h}{2} + \frac{3(Ah)^2}{28} + \frac{(Ah)^3}{84} + \frac{(Ah)^4}{1680}\right] \tag{2.19}$$

As the subscripts indicate, these formulae are all accurate to the double order, i.e., while the individual half-steps are no longer proper Runge-Kutta steps, the overall method attains a considerably higher order of accuracy. Due to the symmetry between $D(p,q)$ and $N(p,q)$ when $p = q$, all these methods are still F-stable.

These techniques have been intensively studied in [7, 13]. The problem with them is that an accuracy analysis is only available for the linear case. When exposed to nonlinear sysems, the methods may drop several orders of accuracy. Thereby, F_8 may degenerate to an algorithm of merely second order.

2.5 L-Stable Backinterpolation Algorithms

The previous set of F-stable backinterpolation techniques has exploited the symmetry of the stability domains of its two half-steps. However, there is no compelling reason why the two semi-steps have to meet exactly in the middle. The explicit semi-step could span a distance of $\alpha \cdot h$, and the implicit semi-step could span the remaining distance $(1-\alpha) \cdot h$. The resulting algorithm would still be at least of the same accuracy order as its two semi-steps. Thereby, the stability domains can be shaped by varying α.

The case with $\alpha > 0.5$ is of not much interest since it will not lead to an L-stable algorithm. But the case with $\alpha < 0.5$ is very useful. This case can generate a series of L-stable backinterpolation algorithms that may be well suited for the numerical solution of stiff systems. From Eqs. (2.8-2.11), F matrices in this case are,

$$F_1 = [I - A \cdot (1-\alpha)h]^{-1}[I + A \cdot \alpha h] \tag{2.20}$$

$$F_2 = \left[I - A \cdot (1-\alpha)h + \frac{(A \cdot (1-\alpha)h)^2}{2}\right]^{-1}\left[I + A \cdot \alpha h + \frac{(A\alpha h)^2}{2}\right] \tag{2.21}$$

$$F_3 = \left[I - A \cdot (1-\alpha)h + \frac{(A \cdot (1-\alpha)h)^2}{2} - \frac{(A \cdot (1-\alpha)h)^3}{6}\right]^{-1}$$

$$\left[I + A\cdot\alpha h + \frac{(A\alpha h)^2}{2} + \frac{(A\alpha h)^3}{6} + \frac{(A\alpha h)^3}{48}\right] \tag{2.22}$$

$$F_4 = \left[I - A\cdot(1-\alpha)h + \frac{(A\cdot(1-\alpha)h)^2}{2} - \frac{(A\cdot(1-\alpha)h)^3}{6} + \frac{(A\cdot(1-\alpha)h)^4}{24}\right]^{-1}$$
$$\left[I + A\cdot\alpha h + \frac{(A\alpha h)^2}{2} + \frac{(A\alpha h)^3}{6} + \frac{(A\alpha h)^4}{24}\right] \tag{2.23}$$

The stability domains of these matrices will be discussed in the next chapter (see Figs. 3.1-6). Note that except for F_4, all the matrices could be L-stable for some value of $\alpha < 0.5$. F_4 will become L-stable when a smaller α is used. More detailed discussions on stability domains are given in the next chapter. Clearly, these methods result in very nice stability domains with large unstable domains in the right half plane. The selection of a good value for α is a compromise: it should be chosen large enough to generate meaningfully large unstable domains in the right half plane, yet small enough to damp out the high frequency components appropriately in the left half plane.

Note that the Backward Runge-Kutta (BRK) algorithms listed in Section 2.1 are special cases of this new class of algorithms with $\alpha = 0$, the explicit Forward Runge-Kutta (FRK) algorithms are special cases of this class of algorithms with $\alpha = 1$, and the F-stable BI algorithms are special cases with $\alpha = 0.5$.

However, there is no value of α that will raise the order of accuracy of the overall algorithm for any order of the semi-step algorithm higher than 1. In the special case of order 1, $\alpha = 0.5$ raises the overall order of accuracy of F_2, given by Eq. (2.8) (the trapezoidal rule), to 2.

2.6 Summary

This chapter summarizes some basic concepts and algorithms for numerical integration techniques. Section 2.1 discussed the Forward and the Backward Euler algorithms. Section 2.2 introduced the concept of numerical stability and illustrated its importance. Section 2.3 explained the motivation for backinterpolation techniques and the basic ideas of their construction methods. Sections 2.4 and 2.5 presented several F-stable and L-stable backinterpolation algorithms.

CHAPTER 3

Analysis and Evaluation of BI Methods

In this chapter, we shall look at various aspects of the proposed BI methods. We start in Section 3.1 with stability analysis and investigate stability domains of BI algorithms. Next in Section 3.2 we address issues related to accuracy considerations of BI algorithms. We then study damping and frequency properties of BI algorithms in Section 3.3. Finally, we present simulation results of two step-size control methods for BI methods in Section 3.4.

3.1 Stability Analysis of BI Algorithms

The discrete-time system of Eq. (2.12) is analytically stable if and only if all its eigenvalues are located inside a circle of radius 1.0 about the origin, the so-called *unit circle.* For a specific numerical integration scheme, the unit circle will be mapped into a stability domain in the $\lambda \cdot h$ plane. Take FE as an example, from Eq. (2.5), we conclude that all eigenvalues of A multiplied by the step size, h, must lie inside a circle of radius 1.0 about the point (-1.0, 0) on the $\lambda \cdot h$ plane.

We define that the linear time-invariant continuous-time system integrated using a given fixed-step integration algorithm is numerically stable if and only if the "equivalent" linear time-invariant discrete-time system is analytically stable [7]. Notice that the numerical stability domain is, in a rigorous sense, only defined for linear time-invariant continuous-time systems, and applies only to fixed-step algorithms. Nevertheless, it is appealing that the numerical stability domain of an integration algorithm can be computed and drawn once and for all, and does not depend on any system properties other than the location of its eigenvalues.

There exist analytical techniques to determine the domain of numerical stability, however, they are somewhat cumbersome and error prone. Therefore, we use a general purpose computer program that can determine the domain of numerical stability of any integration algorithm [7].

To find the stability domain for a BI algorithm, we start out with a second-order system with a pair of complex conjugate eigenvalues along the unit circle. For example,

$$\dot{x} = Ax, \qquad x(t_0) = x_0, \qquad A = \begin{bmatrix} 0 & 1 \\ -1 & -2\cos\theta \end{bmatrix} \tag{3.1}$$

represents such a system, where θ denotes the angle of one of the two eigenvalues counted counterclockwise away from the negative real axis.

Now the stability domain for a BI algorithm can be found as following,

1. For a given α and for θ from 0 to 180 degree:

a) Calculate A matrix for given θ using Eq. (3.1);

b) Calculate F matrix for given A using Eqs. (2.20-2.23);

c) Determine the largest possible value of h, denoted as h_{max}, for which all eigenvalues of F are inside the unit circle.

2. Stability domain is obtained by plotting h_{max} as a function of θ in polar coordinates.

Matlab subroutines, i.e., *Fmat, CalMaxH, Stab, PlotStab*, for these calculations are given in Appendix A.

Using this procedure, next we discuss in detail the effect of α on stability for two specific BI algorithms, BI45 and BI55, whose F matrices are given as,

$$F_{45} = \left[I-(1-\alpha)hA+\frac{((1-\alpha)hA)^2}{2}-\frac{((1-\alpha)hA)^3}{6}+\frac{((1-\alpha)hA)^4}{24}-\frac{((1-\alpha)hA)^5}{125}\right]^{-1}\left[I+\alpha hA+\frac{(\alpha hA)^2}{2}+\frac{(\alpha hA)^3}{6}+\frac{(\alpha hA)^4}{24}\right] \quad (3.2)$$

$$F_{55} = \left[I-(1-\alpha)hA+\frac{((1-\alpha)hA)^2}{2}-\frac{((1-\alpha)hA)^3}{6}+\frac{((1-\alpha)hA)^4}{24}-\frac{((1-\alpha)hA)^5}{125}\right]^{-1}\left[I+\alpha hA+\frac{(\alpha hA)^2}{2}+\frac{(\alpha hA)^3}{6}+\frac{(\alpha hA)^4}{24}+\frac{(\alpha hA)^5}{125}\right]. \quad (3.3)$$

Figs. 3.1 to 3.3 give the stability domains of BI45 for various α ranging from 0.1 to 0.5. As indicated by these figures, the stability domains become smaller as α increases. When $\alpha = 0$, it becomes a pure implicit Runge-Kutta algorithm. However, when $0 < \alpha < 0.3$ or $0.47 < \alpha \leq 0.5$, BI45 is no longer A-stable since a small portion of the left half plane becomes unstable in these cases. Therefore, we

have to choose α between 0.3 to 0.47 in order to maintain the L-stability of BI45. As we can see in the next section, the accuracy domain increases with the value of α. Therefore, it is recommended that $\alpha = 0.47$ should be used for BI45 in numerical integration.

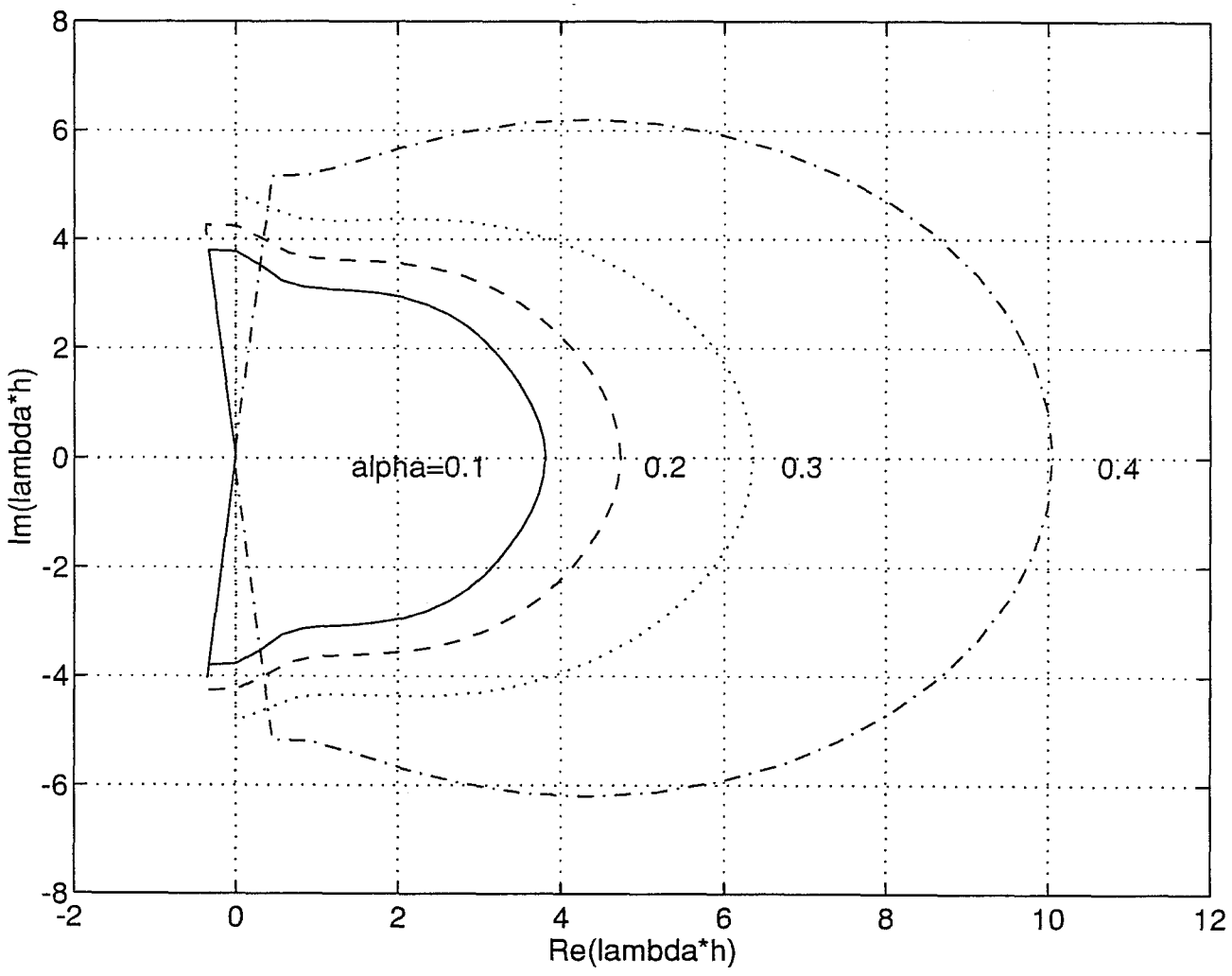

Figure 3.1: Stability Domain of BI45: $\alpha = 0.1 - 0.4$

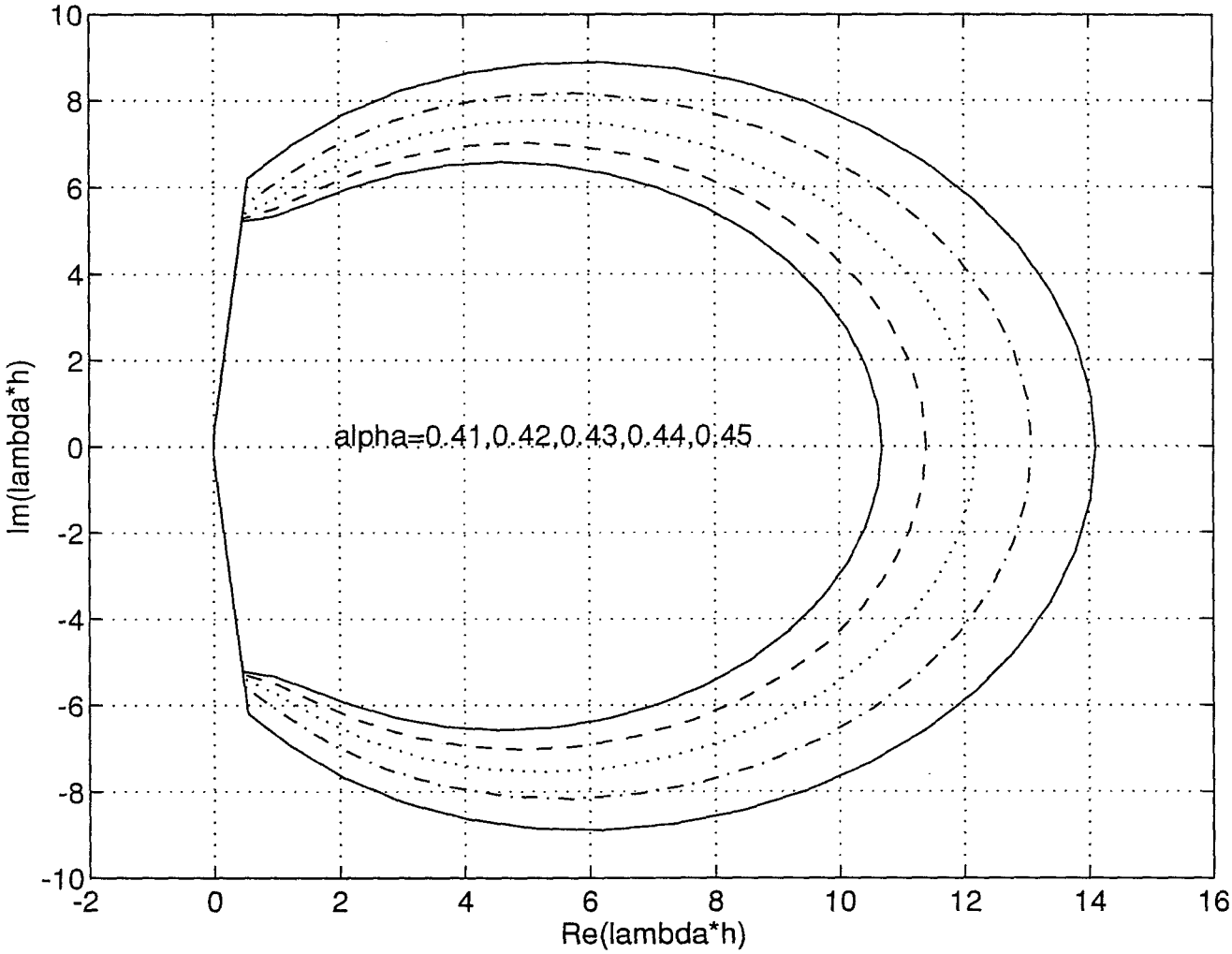

Figure 3.2: Stability Domain of BI45: $\alpha = 0.41 - 0.45$

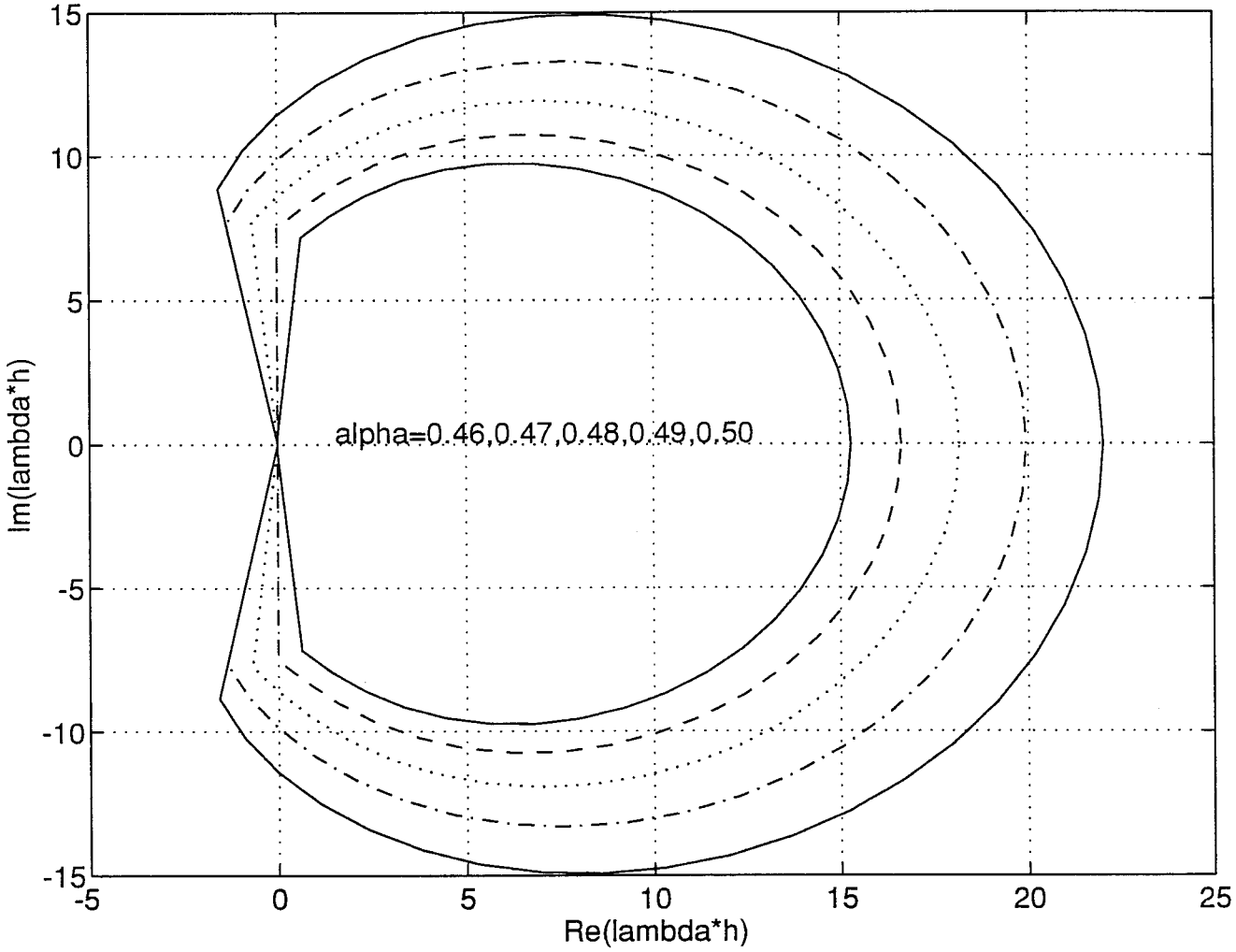

Figure 3.3: Stability Domain of BI45: $\alpha = 0.46 - 0.50$

Similarly, Figs. 3.4 to 3.6 give the stability domains of BI55 for various α. As for BI45, the stability domains of BI55 become smaller as α increases. When $0 < \alpha < 0.3$, BI55 is no longer A-stable. To maintain the L-stability of BI55, we must have $0.3 \leq \alpha < 0.5$. Note that when $\alpha = 0.5$, BI55 become F-stable.

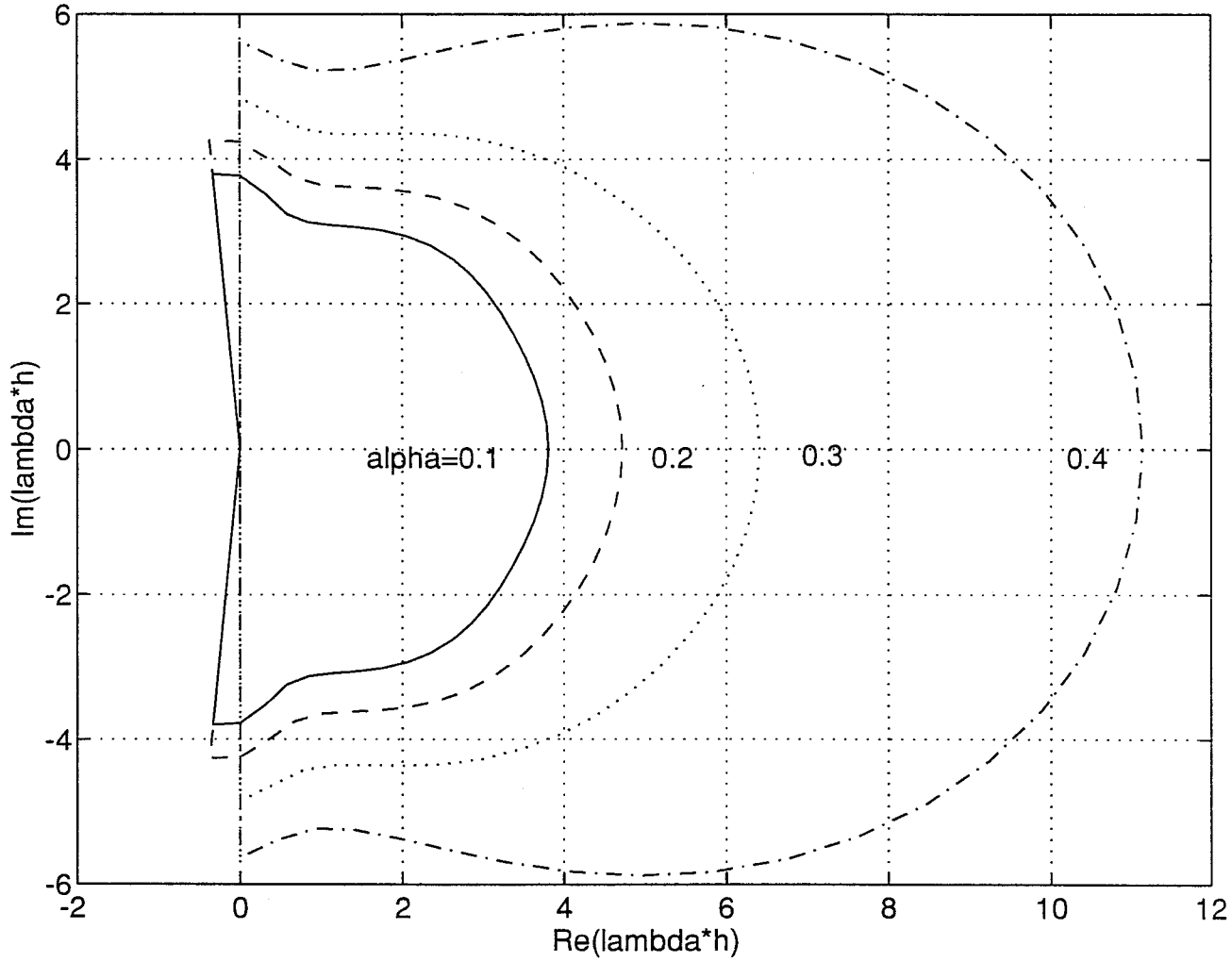

Figure 3.4: Stability Domain of BI55: $\alpha = 0.1 - 0.4$

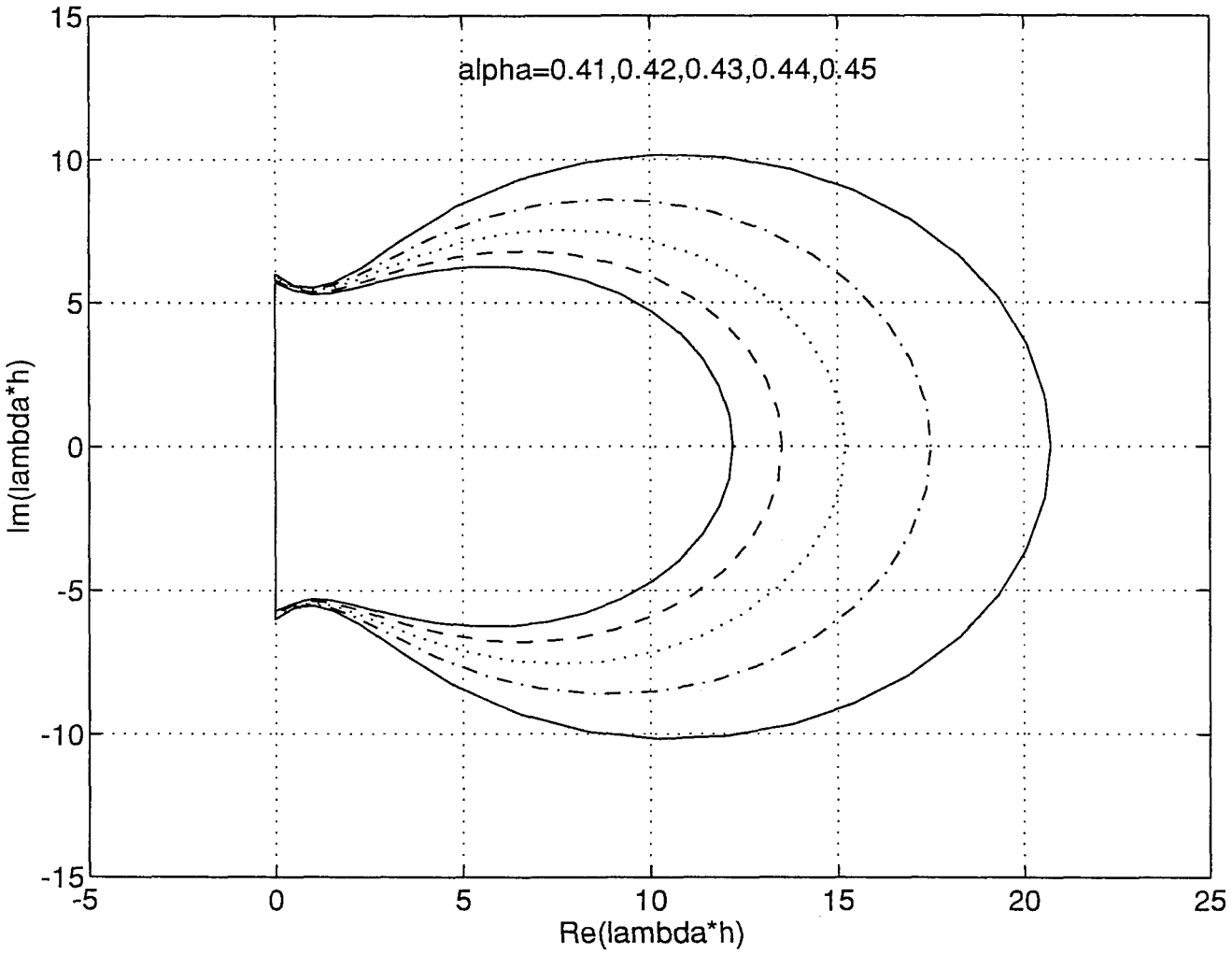

Figure 3.5: Stability Domain of BI55: $\alpha = 0.41 - 0.45$

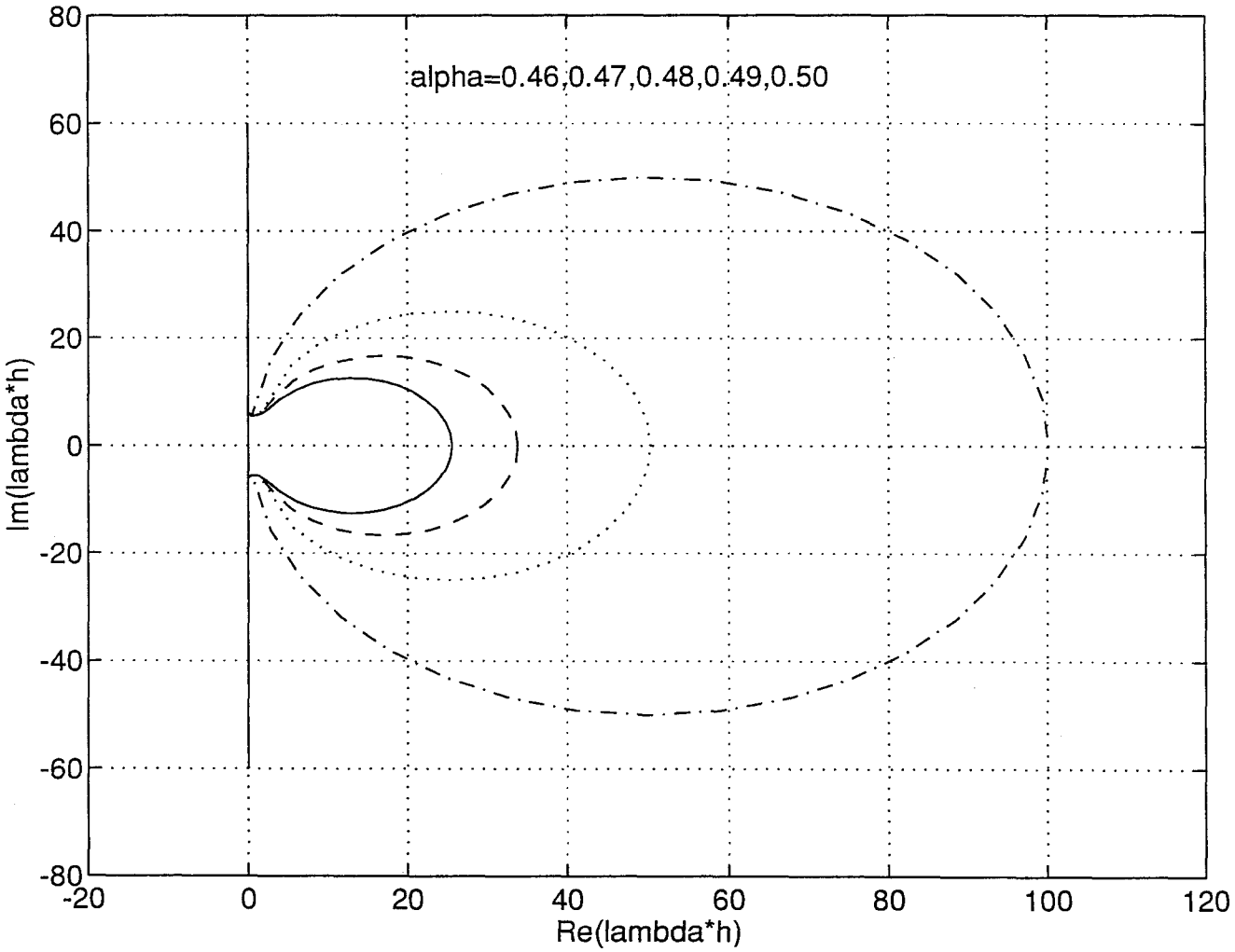

Figure 3.6: Stability Domain of BI55: $\alpha = 0.46 - 0.50$

3.2 Accuracy Analysis of BI Algorithms

To gain more insight and detailed information on the aspect of approximation accuracy for BI methods, in this section we determine the accuracy domain for a BI algorithm using our standard linear test problem (3.1).

For a given initial state x_0, the analytical solution of (3.1) is

$$x_{anal}(t) = \exp(A \cdot t)x_0. \tag{3.4}$$

Let x_{simul} be the solution obtained by a BI algorithm, we define the global error of the BI algorithm as,

$$\varepsilon_{global} = \| x_{anal} - x_{simul} \|_{\infty}. \tag{3.5}$$

where $\| (\cdot) \|_{\infty}$ represents the maximum value of $(\cdot)$ over all discrete time instances used by the BI algorithm during its numerical solution process.

Now the accuracy domain of a BI algorithm is defined as the upper bound of $\| \lambda \cdot h \|$ in order to achieve a specified global accuracy requirement. For a given initial state, the accuracy domain can be obtained by the following steps:

1. Prescribe a tolerance for the global error;

2. For θ from 0 to 180 degree:

   a) Calculate the analytical solution;

   b) Calculate the numerical solution using the BI method;

c) Determine the largest possible value of h, denoted as h_{max}, for which the global error matches the prescribed tolerance;

3. The accuracy domain is obtained by plotting h_{max} as a function of θ in polar coordinates.

Similar to our stability discussion, we study the effect of α on accuracy for two specific BI algorithms, i.e., BI45 and BI55.

For $\alpha = 0.1, 0.2, 0.3, 0.4$, Figs. 3.7-9 show the accuracy domain of BI45 for initial conditions $x_0 = (1,0), (0,1)$, and (1,1), respectively, while Figs. 3.10-12, presents the corresponding accuracy domains of BI55. Clearly, the accuracy domains become larger as α increases. The specified accuracy is 10e-4 and an evaluation time of 10 seconds is used for error calculation in Eq. (3.5).

As we have seen from these figures, the accuracy domain of an algorithm depends heavily on the specified initial condition. Therefore, accuracy domains of BI algorithms are not very useful. Approximately, the largest step size that can be chosen for a prescribed tolerance is inversely proportional to the largest gradient in the simulation, and thus for a linear system $\dot{x} = Ax$, since the gradient is proportional to the norm of the state vector if the system is stable, the largest step size is also inversely proportional to the norm of the initial condition for stable systems. This can be used as a rule of thumb when one has to use the accuracy domain to determine an upper bound of step sizes for an algorithm.

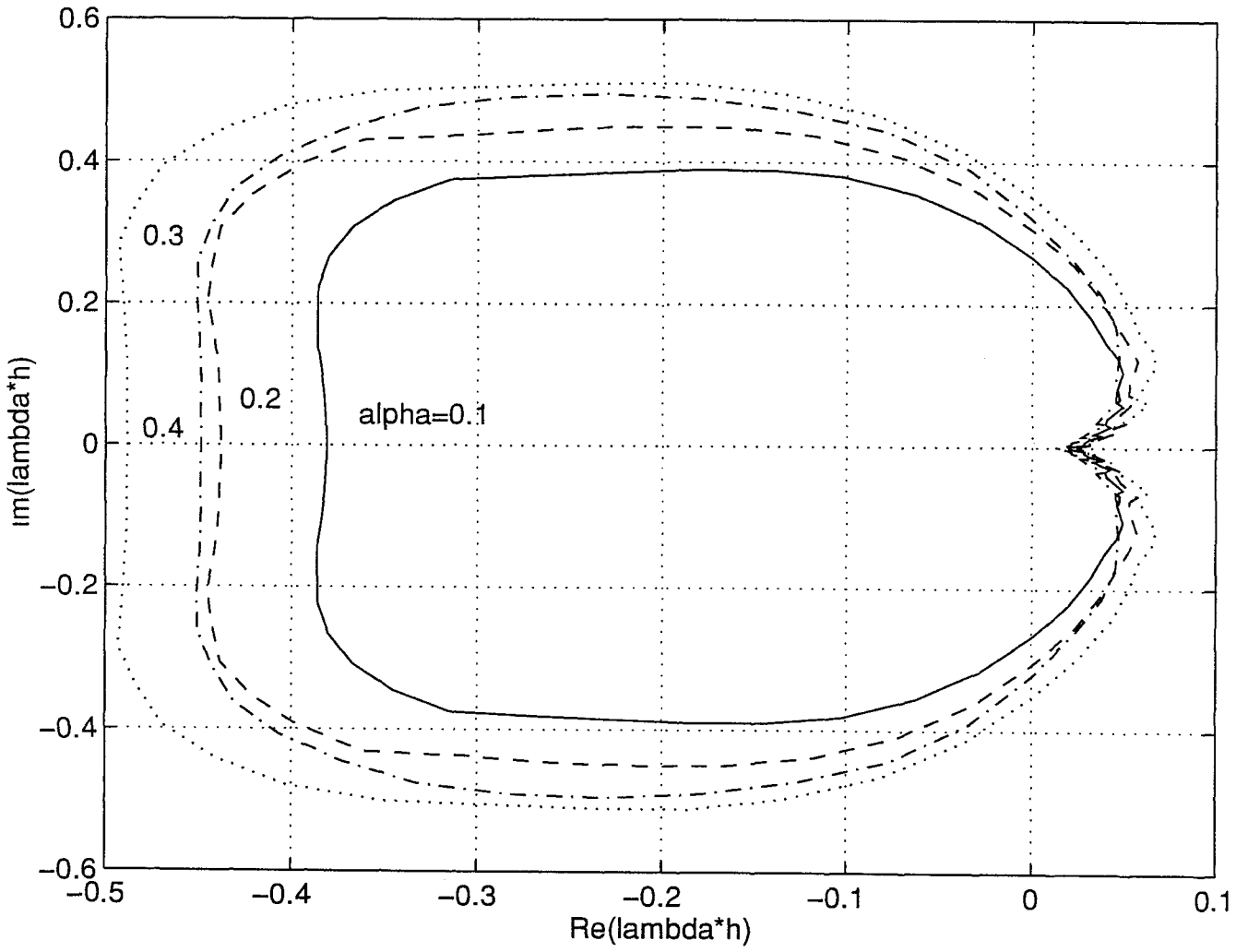

Figure 3.7: Accuray Domain of BI45: $x_0 = (1, 0), \varepsilon_{global} = 10^{-4}$

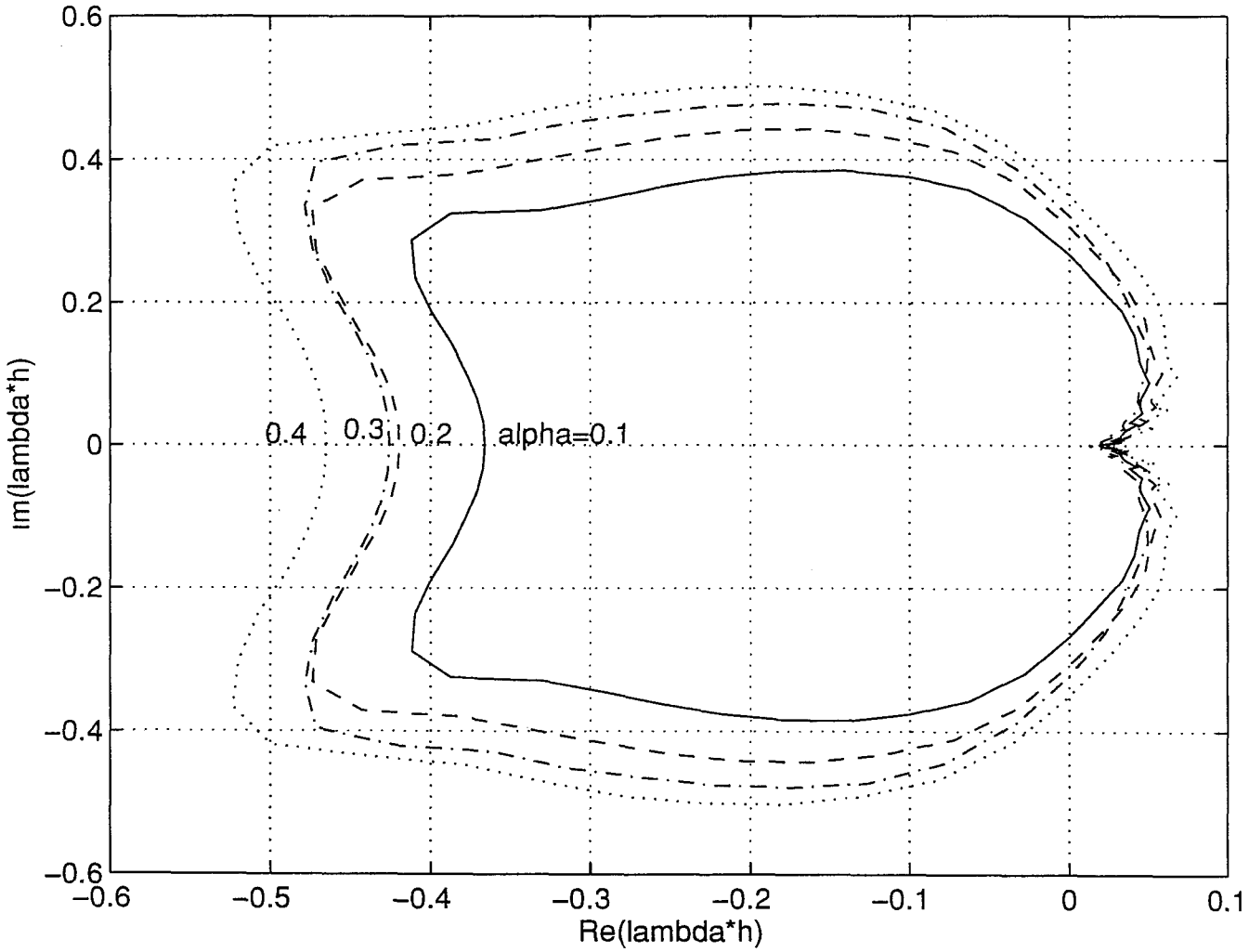

Figure 3.8: Accuray Domain of BI45: $x_0 = (0, 1), \varepsilon_{global} = 10^{-4}$

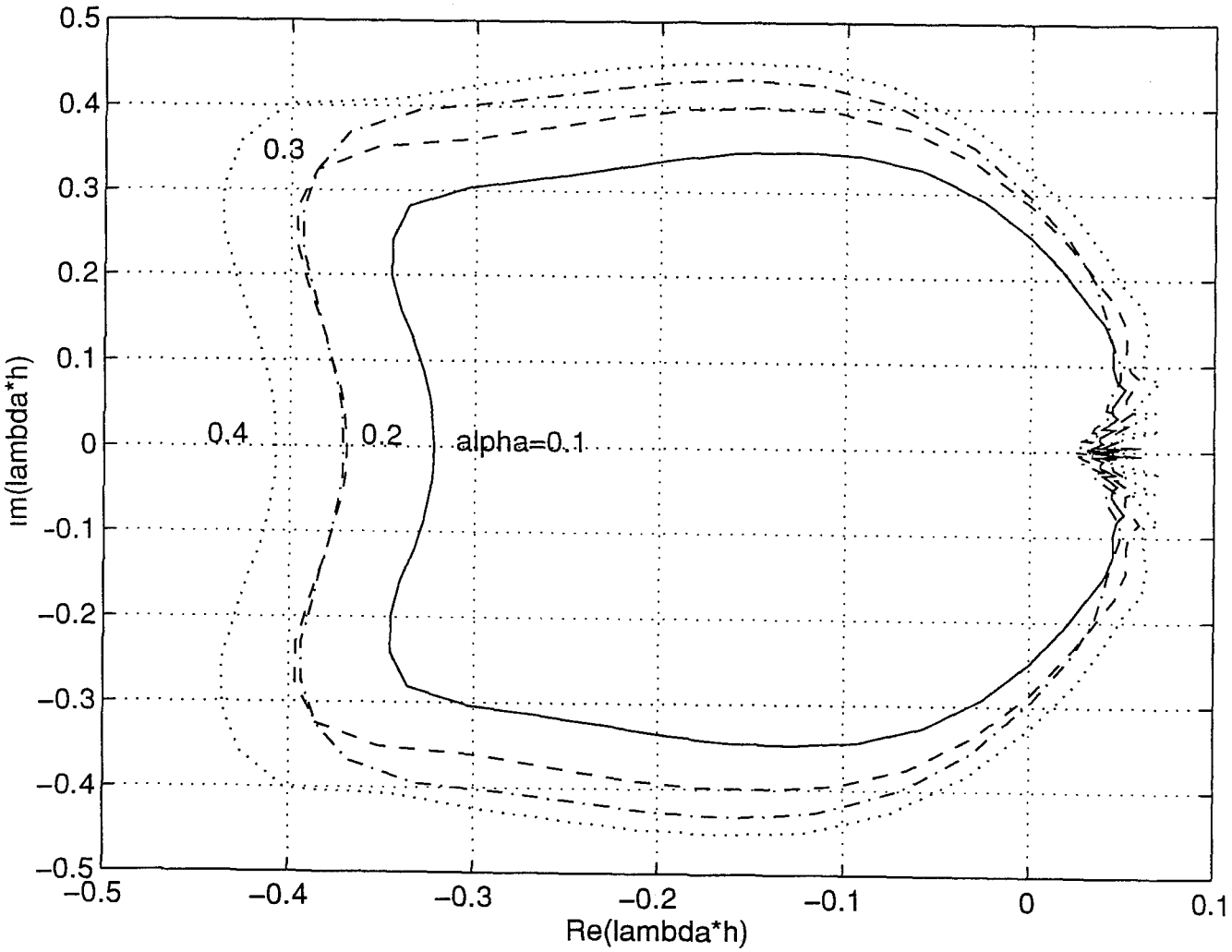

Figure 3.9: Accuray Domain of BI45: $x_0 = (1,1), \varepsilon_{global} = 10^{-4}$

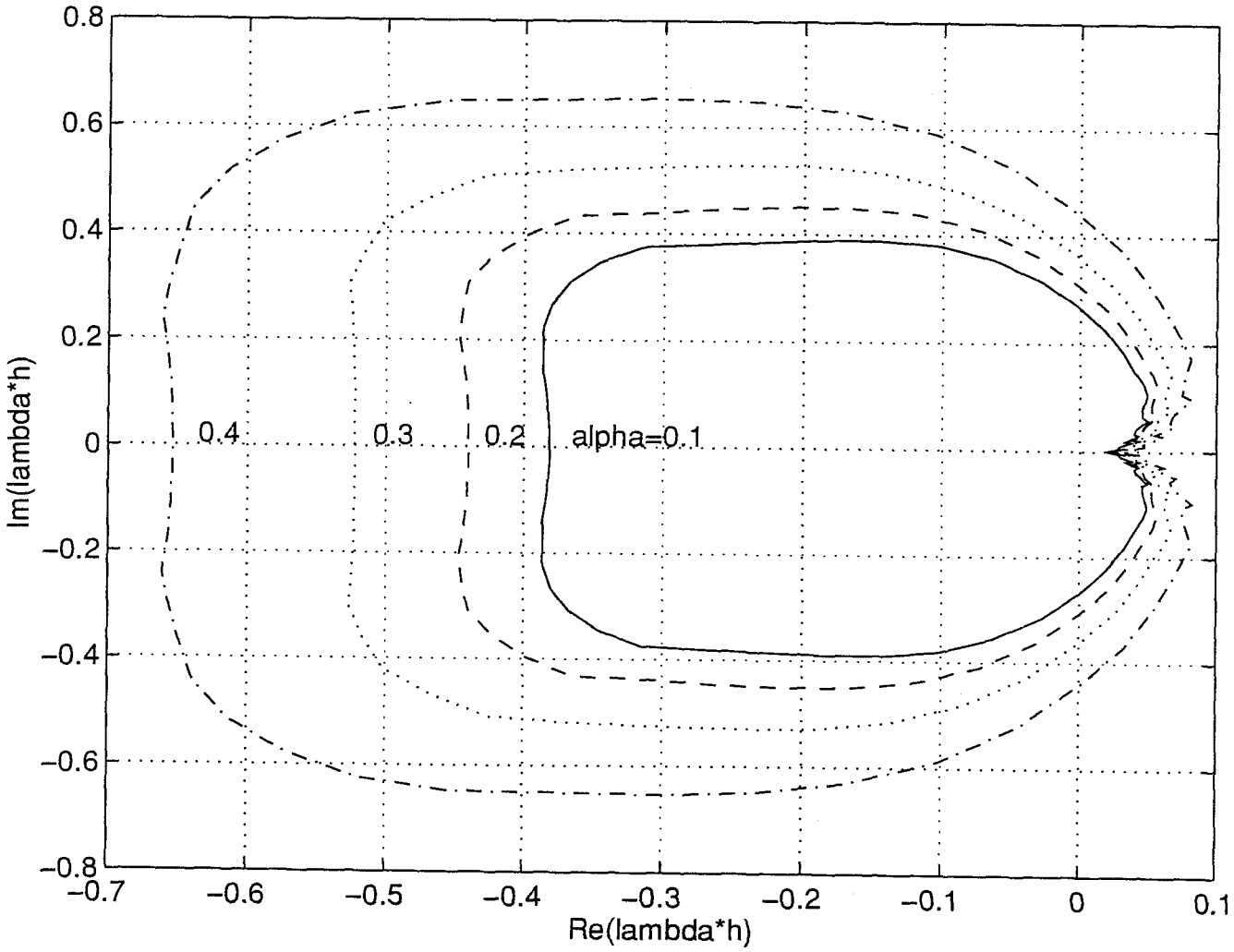

Figure 3.10: Accuray Domain of BI55: $x_0 = (1, 0), \varepsilon_{global} = 10^{-4}$

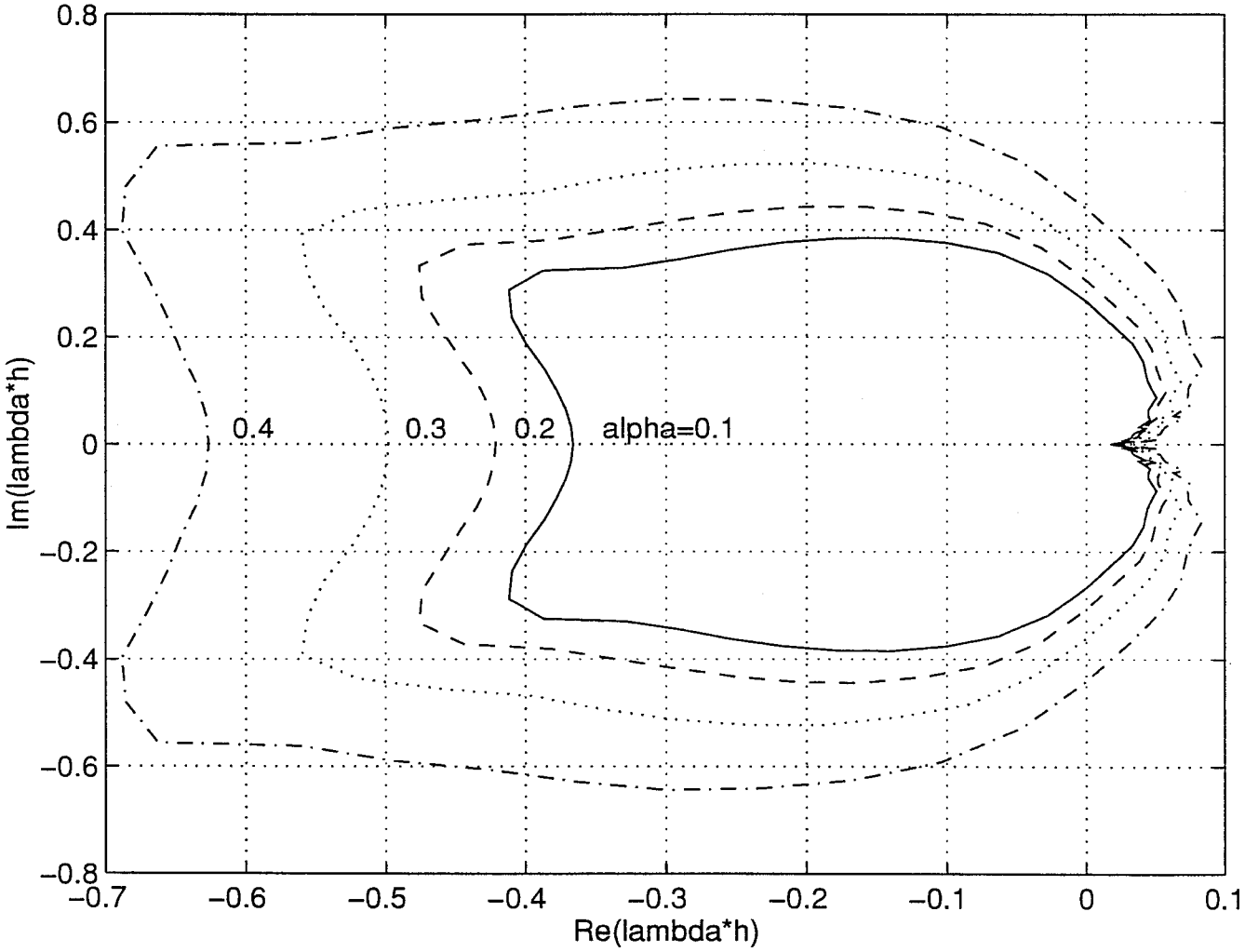

Figure 3.11: Accuray Domain of BI55: $x_0 = (0, 1), \varepsilon_{global} = 10^{-4}$

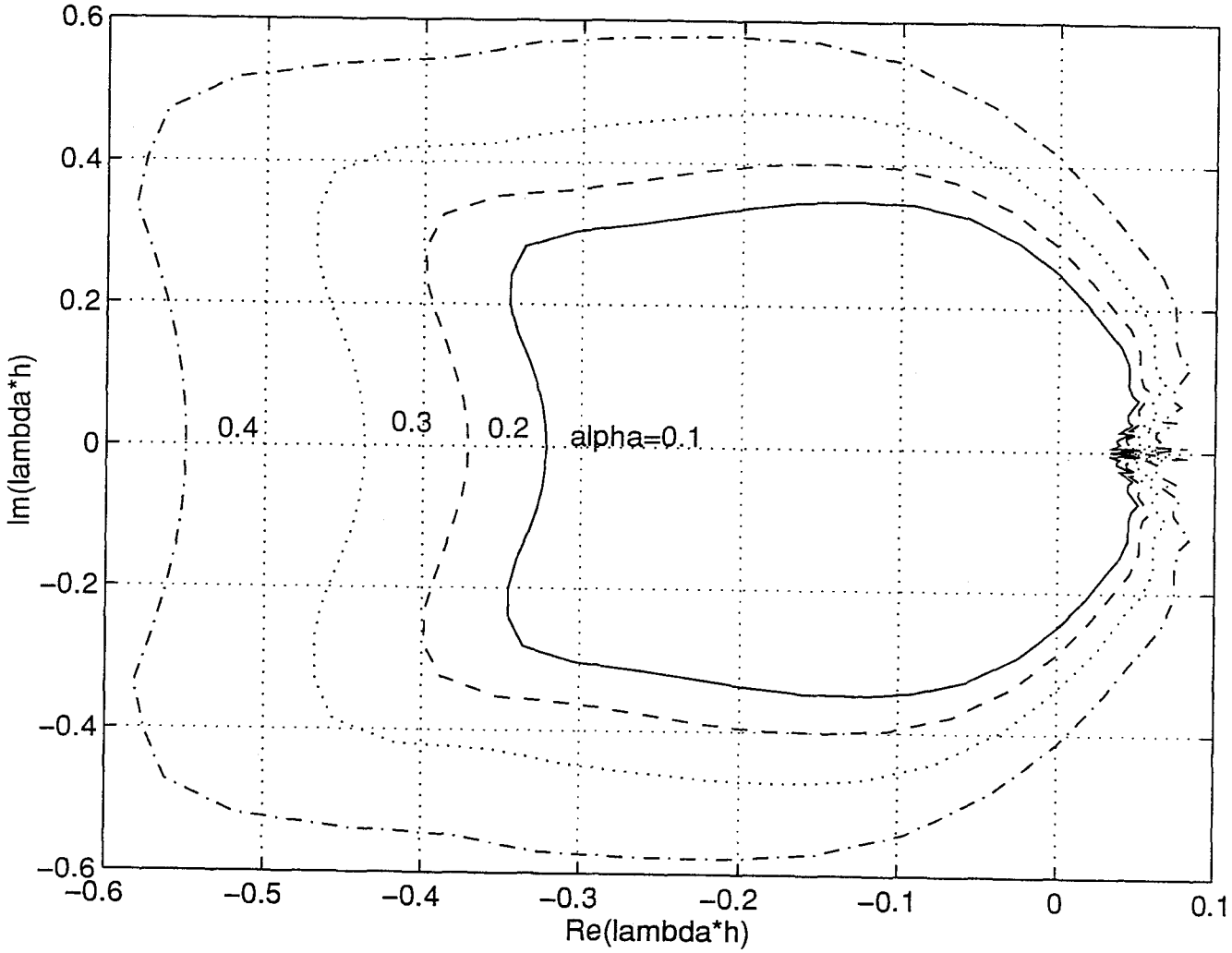

Figure 3.12: Accuray Domain of BI55: $x_0 = (1, 1), \varepsilon_{global} = 10^{-4}$

3.3 Frequency-Domain Analysis of BI Algorithms

As we have discussed, accuracy domains are not as convenient as one would hope for due to their strong dependence on the chosen initial conditions. To avoid this problem, we investigate the damping and frequency properties of BI algorithms. The basic idea is to see how close the numerical frequency domain properties obtained by BI algorithms are to the analytical ones of the original systems.

Consider our standard linear test problem of Eq. (3.1) and its analytical solution (3.4). Obviously, the analytical solution is correct for any value of x_0 and any value of t, and in particular, it is true for $x_0 = x_k$ and $t = t_{k+1}$. This substitution leads to:

$$x_{k+1} = \exp(A \cdot h) \cdot x_k \tag{3.6}$$

Therefore, the analytical F-matrix of this system is:

$$F_{anal} = \exp(Ah) \tag{3.7}$$

Let $\lambda = -\sigma \pm j\omega$ be the eigenvalues of A. One form to represent the general solution of a second-order system with complex eigenvalues is:

$$x(t) = c_1 \exp(-\sigma t)\cos(\omega t) + c_2 \exp(-\sigma t)\sin(\omega t) \tag{3.8}$$

where σ is the distance of the eigenvalues from the imaginary axis and is called the *damping* of the eigenvalues, and ω is called the *frequency* of the eigenvalues.

From the above solution, it is clear that damping and frequency determine the characteristics of a solution.

Now let us introduce the *discrete eigenvalues* by the following equation:

$$\lambda_d = \lambda \cdot h = -\sigma_d \pm j\omega_d \tag{3.9}$$

where $\sigma_d = \sigma \cdot h$ and $\omega_d = \omega \cdot h$ are *discrete damping* and *discrete frequency*, respectively.

Mapping the λ-plane to the z-plane, the corresponding eigenvalues of F_{anal} are found by:

$$z = \exp(\lambda_d) \tag{3.10}$$

In other words,

$$|z| = \exp(-\sigma_d), \qquad \angle z = \omega_d \tag{3.11}$$

Now, let us replace the analytical F_{anal}-matrix by the one that is obtained by a BI algorithm, F_{simul}. The numerical F_{simul}-matrix is a rational approximation of the analytical F_{anal}-matrix. Letting $\hat{z}$ be the eigenvalue of F_{simul}, we define *approximate discrete eigenvalues* of F_{simul} by:

$$\hat{z} = \exp(\hat{\lambda}_d) \tag{3.12}$$

Assume $\hat{\lambda}_d = -\hat{\sigma}_d \pm j\hat{\omega}_d$, then,

$$|\hat{z}| = \exp(-\hat{\sigma}_d), \qquad \angle \hat{z} = \hat{\omega}_d \tag{3.13}$$

Clearly, as $\hat{z}$ approximates z, so must $\hat{\sigma}_d$ be an approximation of σ_d, and $\hat{\omega}_d$ must be an approximation of ω_d.

To see the performance of BI45 and BI55 in terms of approximate discrete eigenvalues, we consider the case where

$$\lambda = r(-\cos\theta \pm j \sin\theta)$$

For α =0.1 and 0.4, Figs. 3.13-17 show σ_d (solid line) vs $\hat{\sigma}_d$ and ω_d vs $\hat{\omega}_d$ (dotted line for $\alpha = 0.1$ and dotdashed line for $\alpha = 0.4$) when r changes from 0 to 20 for $\theta = 0, 15, 45, 75$, and 90 degrees, respectively, while Figs. 3.18-22 give the corresponding results for BI55. From Eq. (3.8), since a solution is characterized by its damping and frequency, it is clear that a good approximation is possible only when $r < 4$. Also, the approximation result is better when θ is not close to 90 degree.

3.4 Stepsize Control for BI Algorithms

An integration method is supposed to produce a numerical solution within a specified error tolerance *tol*. Ideally, the global error should be used to select a step size so the given tolerance is satisfied. But the global error is difficult to estimate and most integration methods confine themselves to keeping an estimate of the local truncation error close to ϵ, where ϵ is chosen in relation to *tol*. This is achieved by controlling the stepsize during the integration.

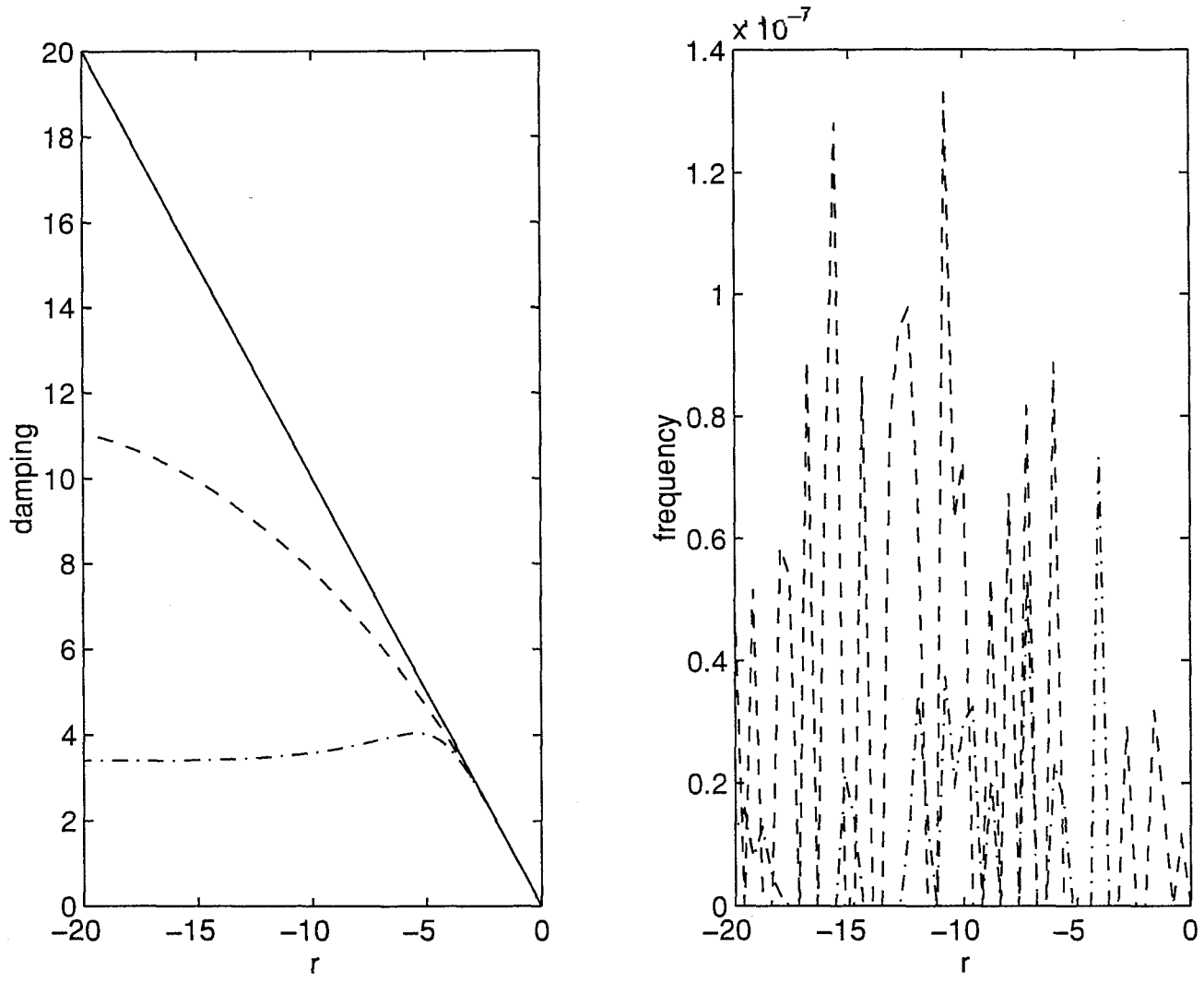

Figure 3.13: Damping and Frequency Plots for BI45: $\theta = 0°$

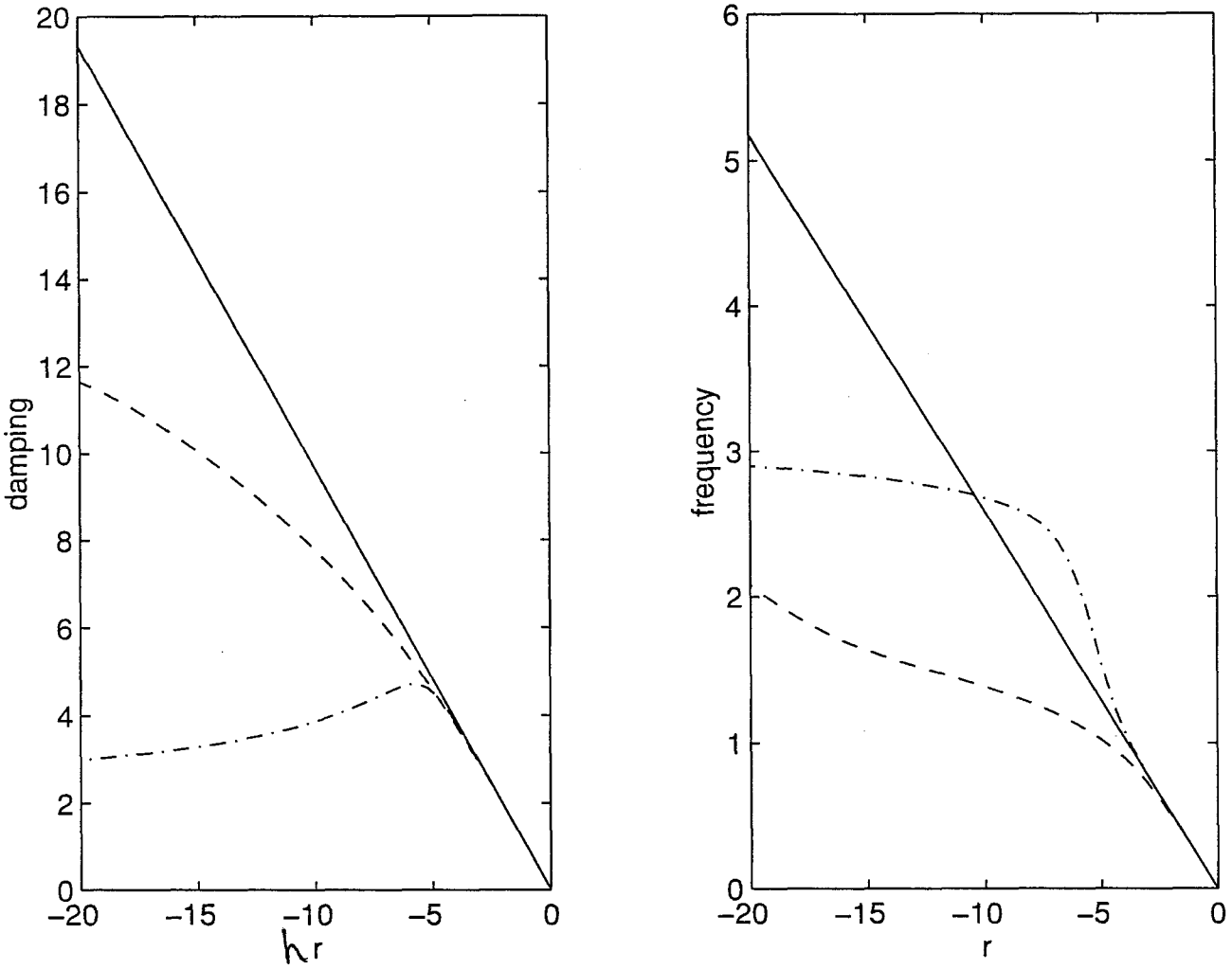

Figure 3.14: Damping and Frequency Plots for BI45: $\theta = 15°$

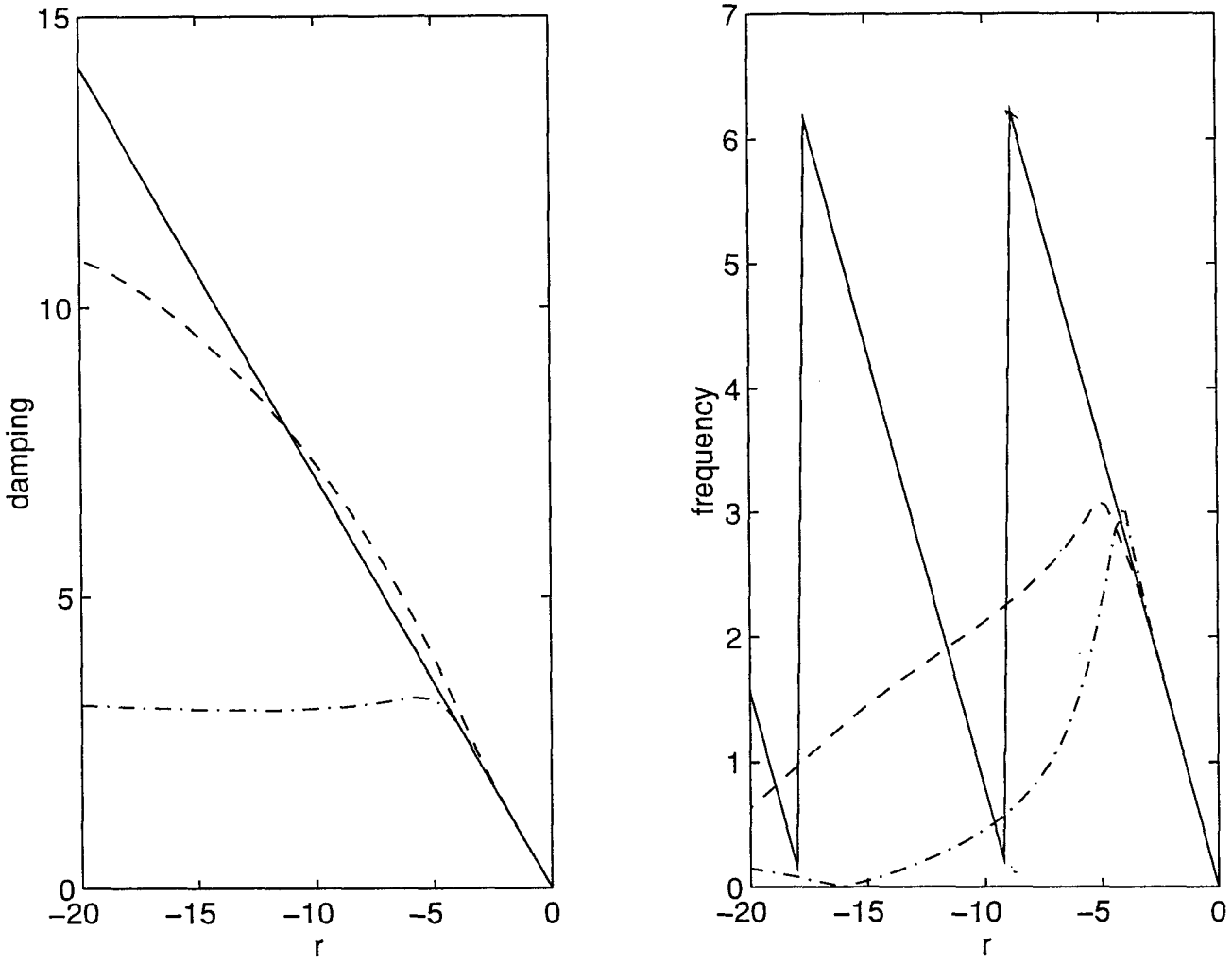

Figure 3.15: Damping and Frequency Plots for BI45: $\theta = 45°$

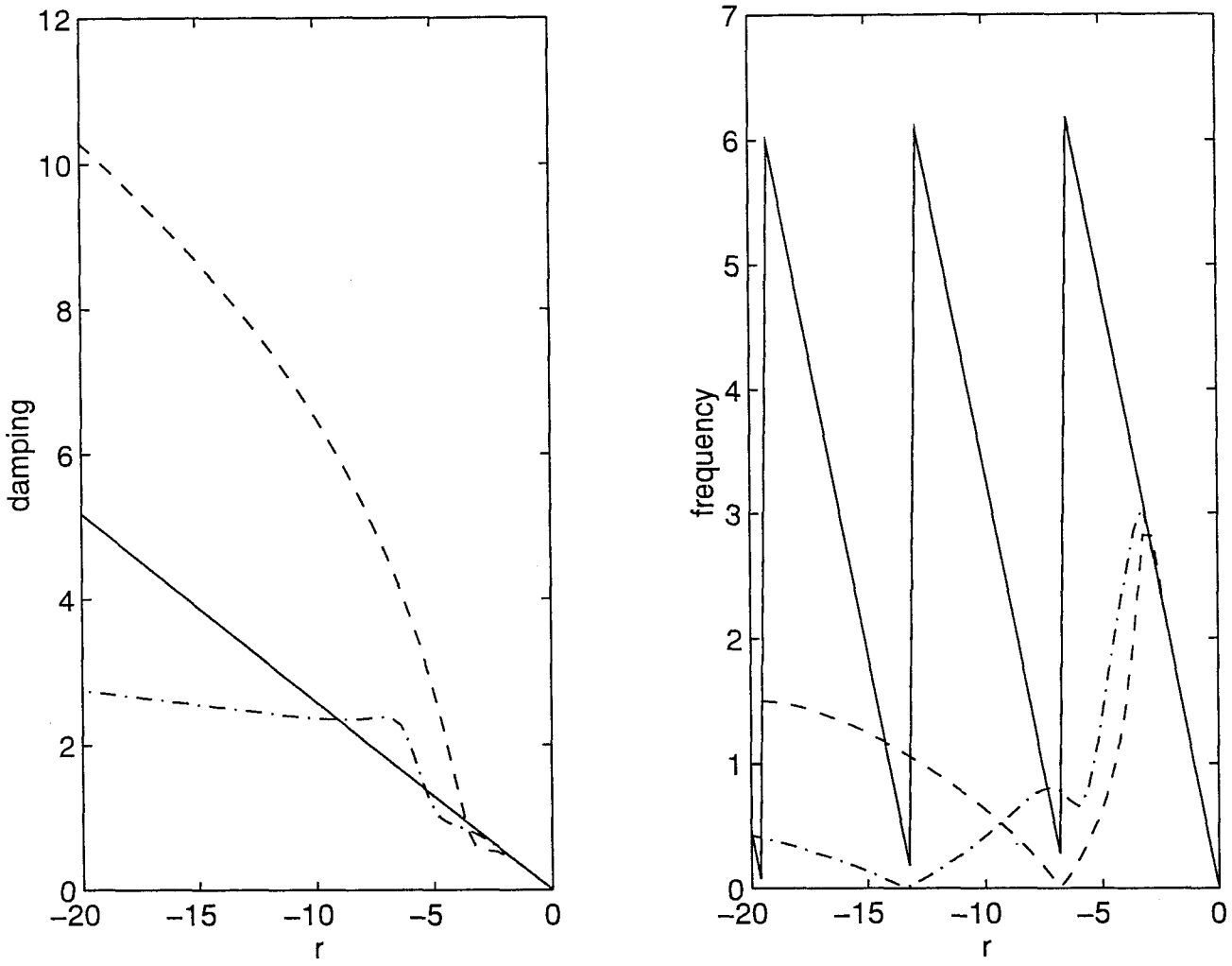

Figure 3.16: Damping and Frequency Plots for BI45: $\theta = 75°$

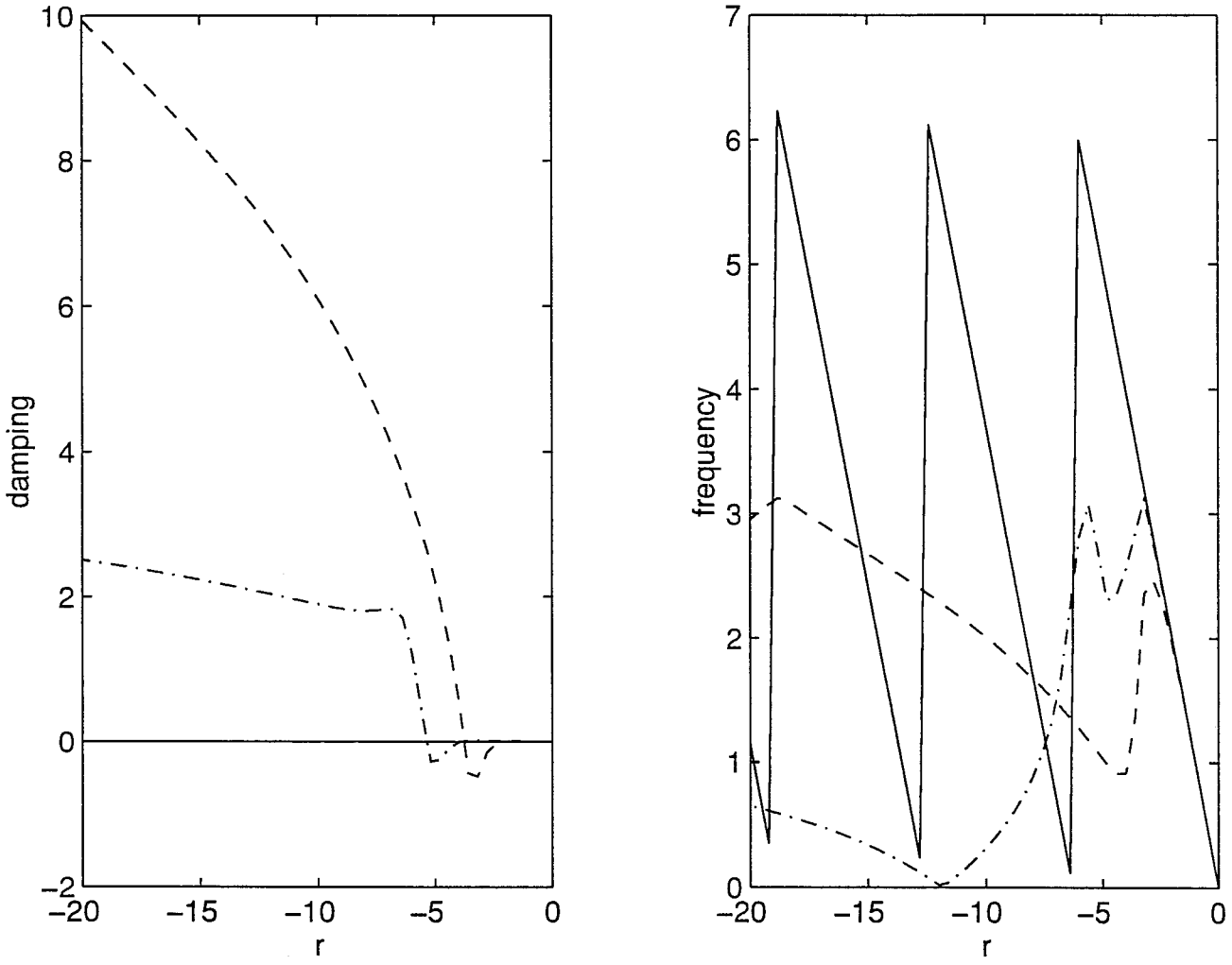

Figure 3.17: Damping and Frequency Plots for BI45: $\theta = 90°$

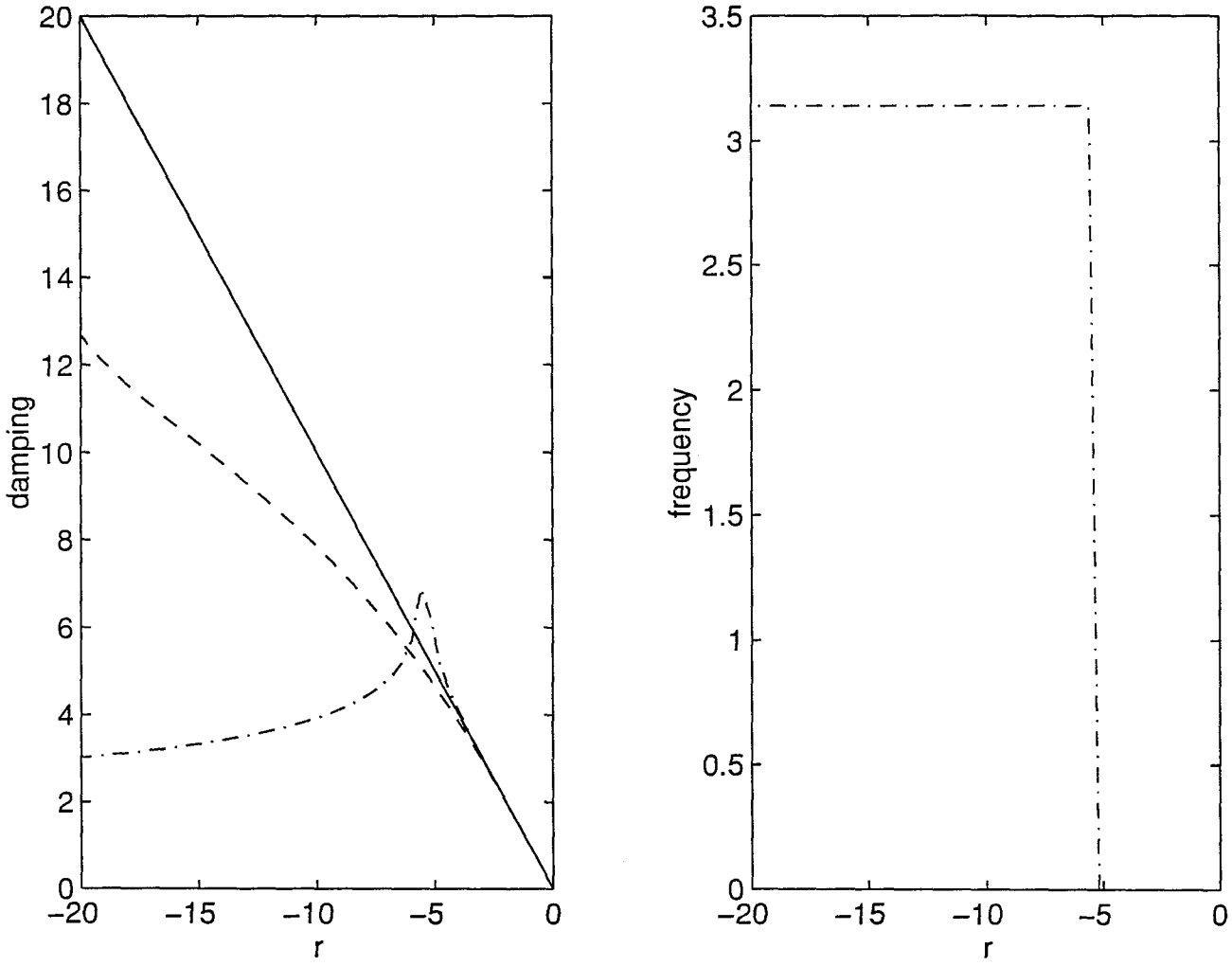

Figure 3.18: Damping and Frequency Plots for BI55: $\theta = 0°$

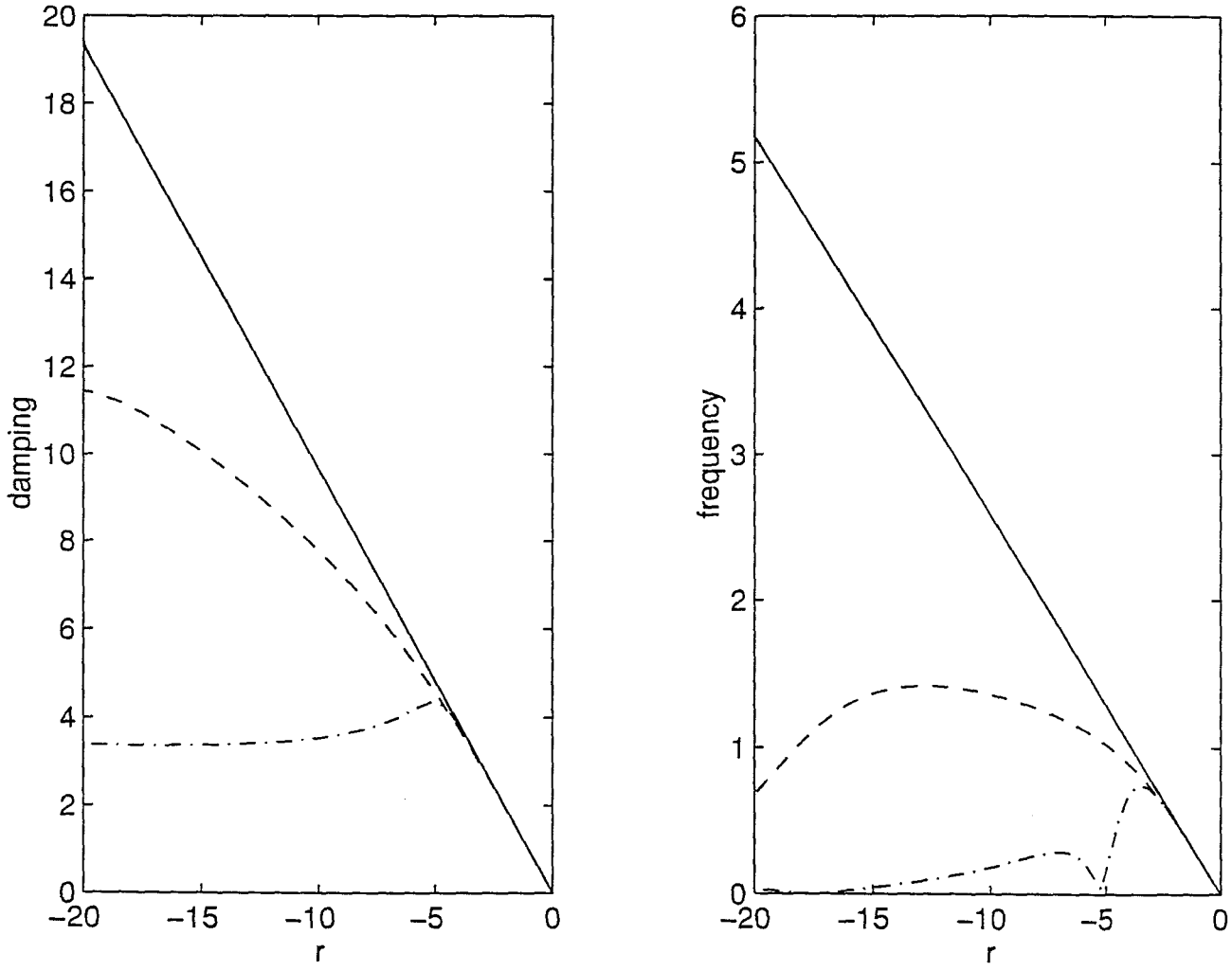

Figure 3.19: Damping and Frequency Plots for BI55: $\theta = 15°$

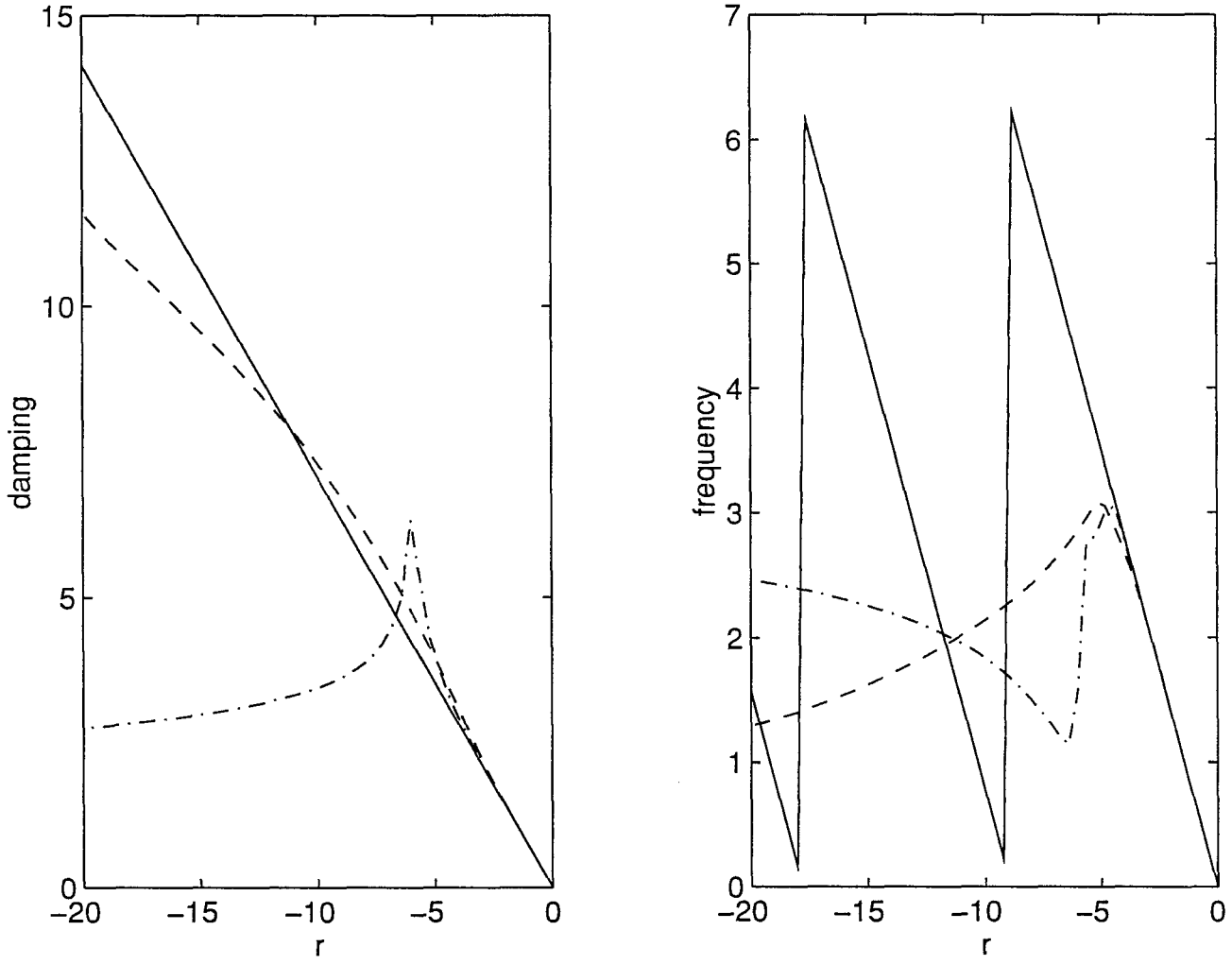

Figure 3.20: Damping and Frequency Plots for BI55: $\theta = 45°$

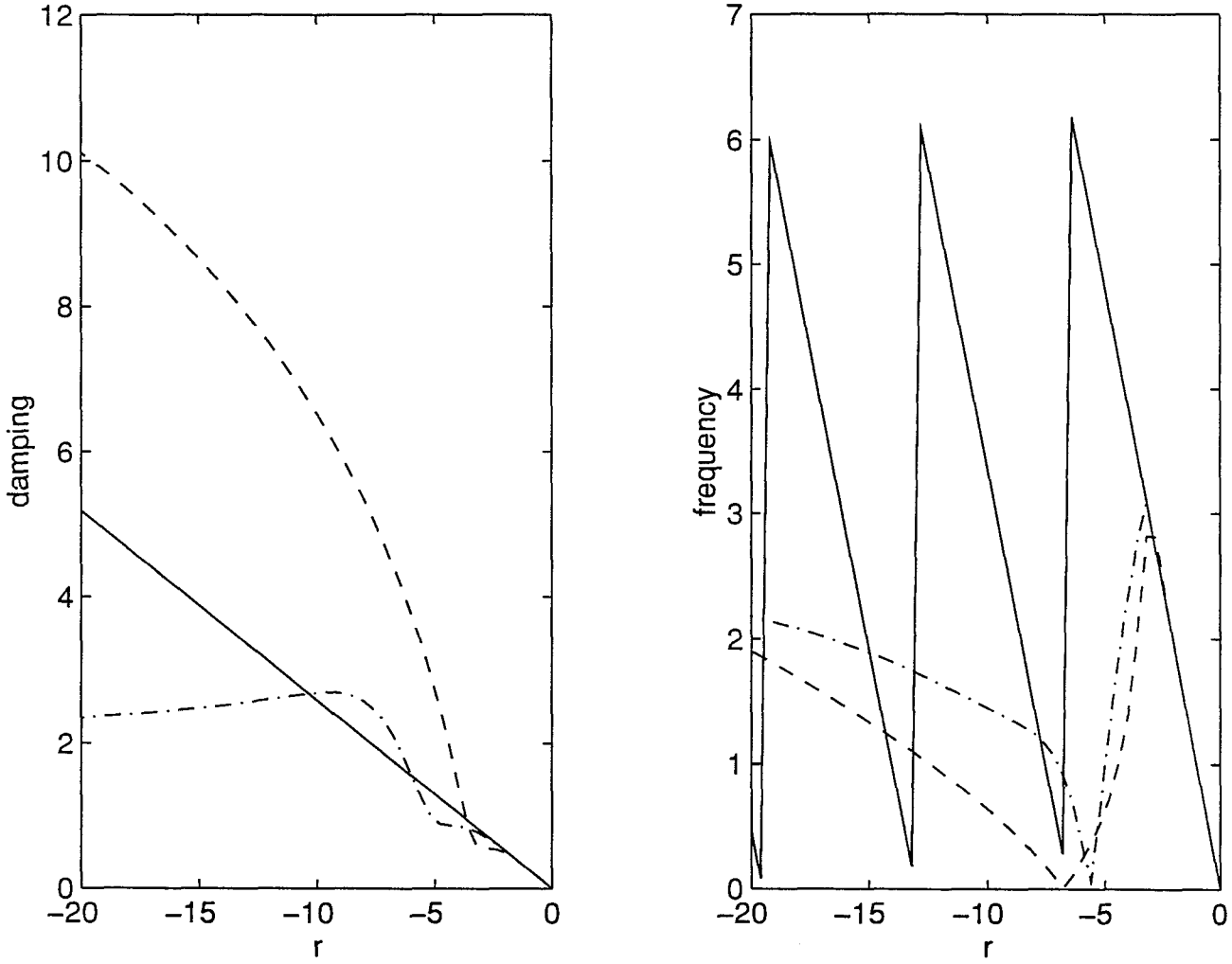

Figure 3.21: Damping and Frequency Plots for BI55: $\theta = 75°$

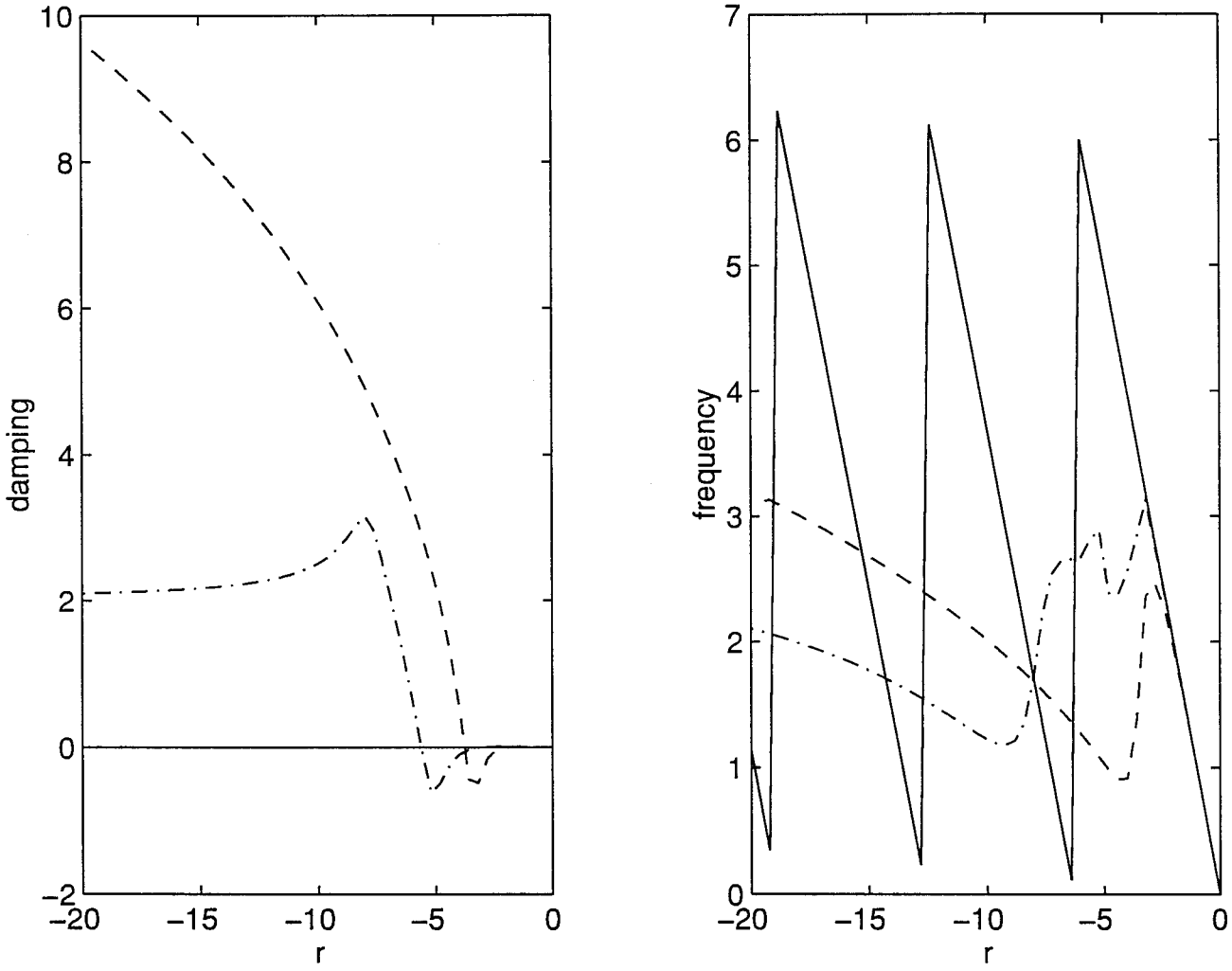

Figure 3.22: Damping and Frequency Plots for BI55: $\theta = 90°$

For a single-step integration method, the standard stepsize control is given as,

$$h_i = \left(\frac{\epsilon}{r_i}\right)^{1/p} h_{i-1} \tag{3.14}$$

where p is the order of the truncation error and r_i is the error estimate. If the error in a step is too large, e.g., $r_i > \rho\ tol$, the step is rejected, and a new attempt is made with a smaller stepsize. By having $\epsilon < tol$ and $\rho > 1$, typically, $0.2tol < \epsilon < 0.8tol$ and $1 < \rho < 1.3$, a safety factor is introduced in the choice of the stepsize, and the risk for a rejected step is reduced [12].

No work is available for stepsize control of BI methods yet. The following procedure is suggested by Professor F. E. Cellier.

1. During the forward half-step, estimate the error using (see SBI45 in Appendix A)

   err = ZROW(n,1);

   FOR $i = 1 : k$, ...

   err = err + $\gamma(i)$*$f(:, i)$; ...

   END

   where γ is a weighting coefficient vector for error estimation. Then the error estimate for the forward half-step is computed as,

   err($h/2$)_left = err($h/2$)

2. During the backward half-step, the error is not estimated until the iteration is completed and $x(k+1/2)$_right is calculated. After the iteration is completed, the error from the backward half-step is then estimated using the same algorithm:

   err = ZROW(n,1);

   FOR $i = 1 : k$, ...

   err = err + gamma(i)*$f(:, i)$; ...

   END

   and

   err($h/2$)_right = err($h/2$)

3. In the best case, these two errors will compensate. Therefore, then the total error from k to $k+1$ would be:

   err(h) = | err($h/2$)_left - err($h/2$)_right |

4. Once the total error of the entire step has been estimated, the new stepsize can be computed as either [12],

$$h_{new} = \min\left\{h_{max}, 0.8h_{old}\left(\frac{\tau}{err}\right)^{pow}\right\} \tag{3.15}$$

   or

$$h_{new} = \min\left\{h_{max}, 0.8h_{old}\left(\frac{\tau}{err}\right)^{pow1}\left(\frac{err\_last}{err}\right)^{pow2}\right\}; \tag{3.16}$$

where h_{max} is the maximum allowable stepsize, $\tau = tol \max\{\| x(k+1) \|, 1.0\}$, err_last is the previous error estimate, pow=1/5, pow1=3/50 and pow2=4/50 are used in our BI implementation.

Figs. 3.23-3.27 present maximum errors achieved for each state and the number of iterations required for BI with stepsize control (3.14) (dash lines) , BI45 with stepsize control (3.15) (solid lines), and the standard Matlab ODE solver *ode45* (a fifth order RK algorithm using stepsize control (3.14), dashdot lines), respectively. Clearly, BI45 algorithms can achieve higher accuracy with a much smaller number of iterations, especially when the specified tolerance is small. Stepsize control (3.15) is better than (3.14) in this case.

Figs. 3.28-3.32 give the corresponding results for a stiff problem (a third order system, with poles (-100, -1 + 0.5j, -1 - 0.5j), so its characteristic equation is: $(s+100)[(s+1)^2+1/4] = 0)$. In this case, ode45 can achieve a better accuracy for state $x_1 = y$ with a much larger number of iterations than the BI45 algorithms.

Actually, all three algorithms have achieved a very good accuracy for state x_1 and their difference is negligible. For states $x_2 = \dot{y}$ and $x_3 = \ddot{y}$, the maximum errors are much larger and the two BI45 algorithms perform much better than ode45. In this case, BI45 with stepsize control (3.14) is a little better than with (3.15), but (3.15) has achieved a much better accuracy than (3.14) for state x_3.

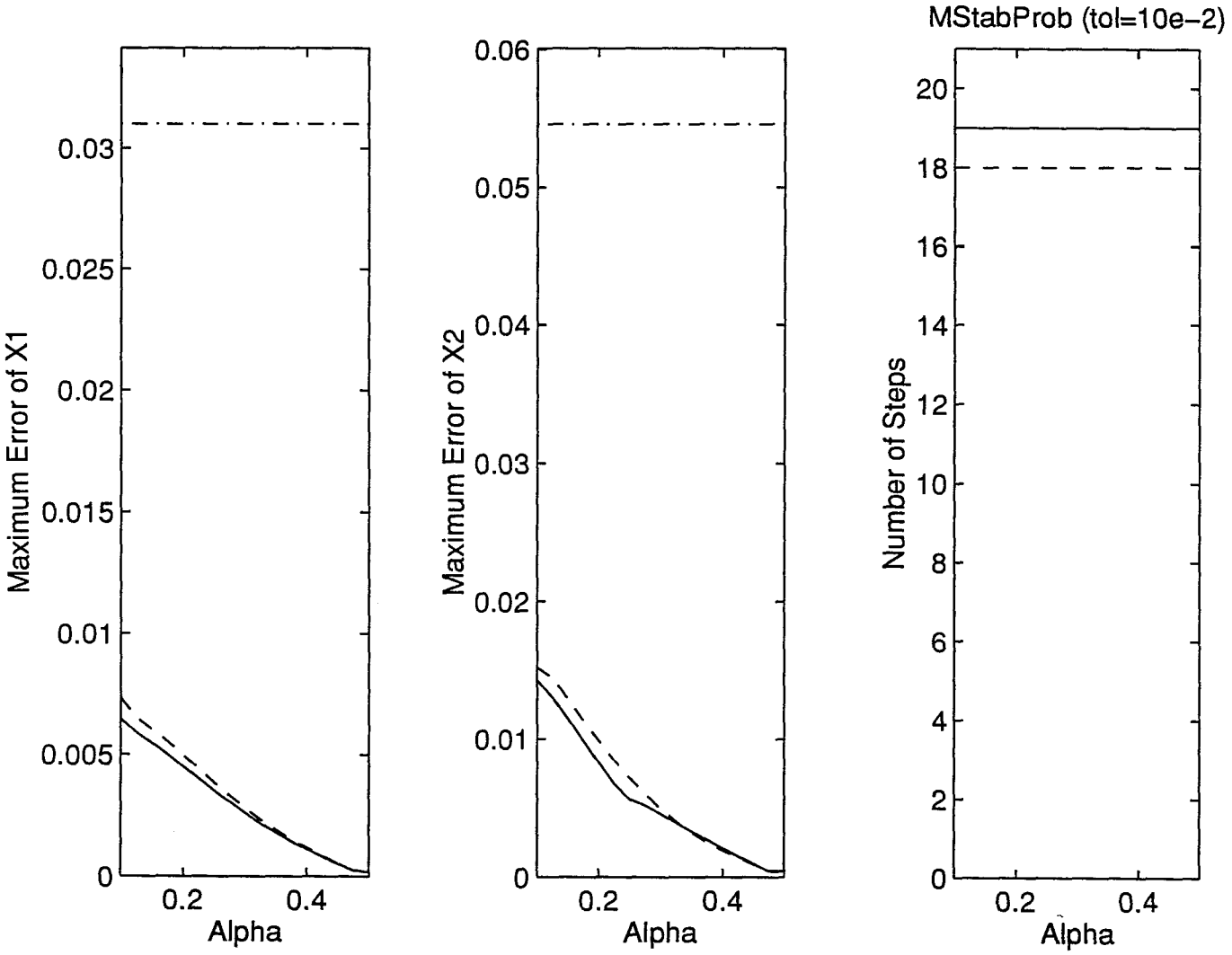

Figure 3.23: Maximum Errors and Steps: Marginally Stable Problem (tol=10^{-2})

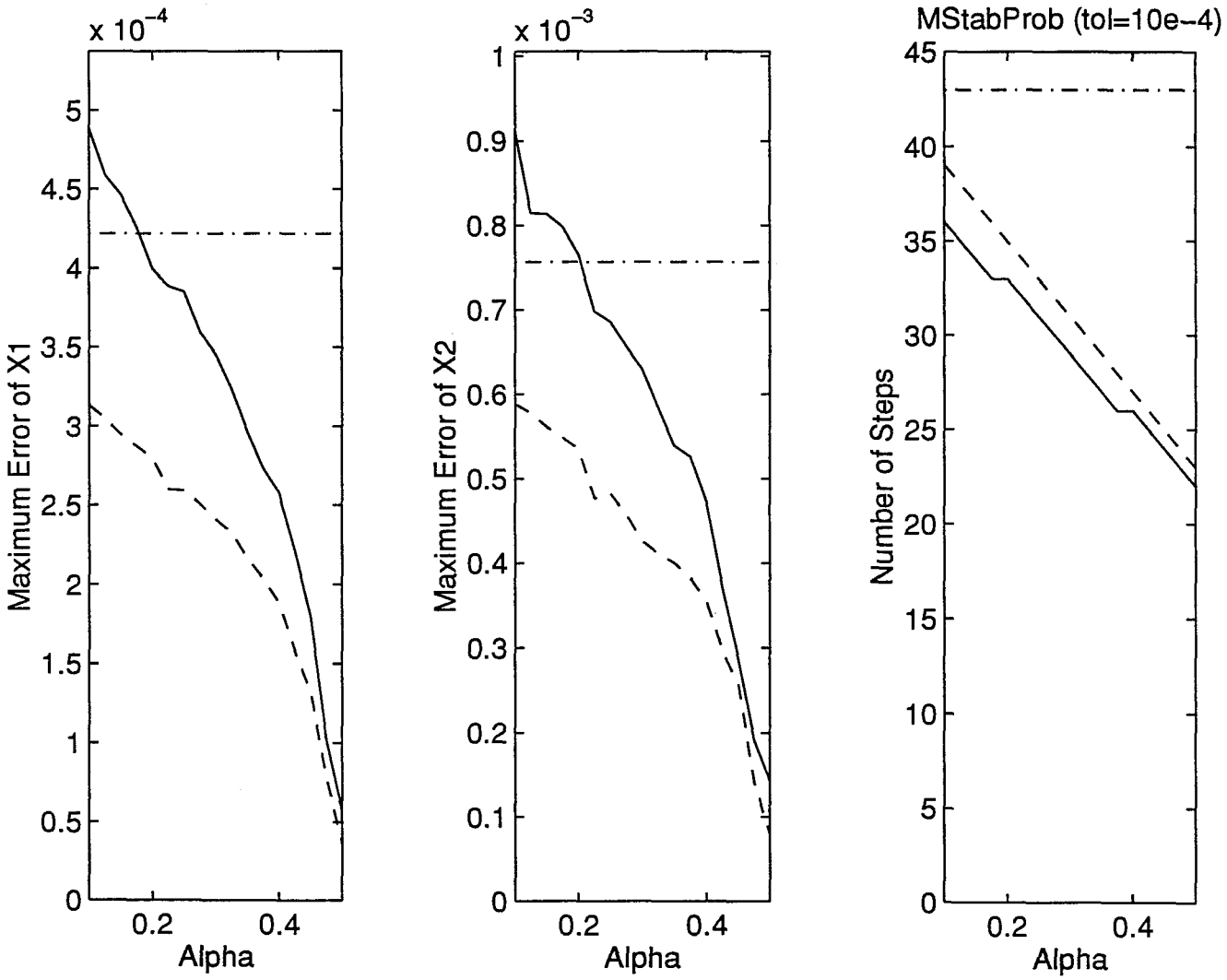

Figure 3.24: Maximum Errors and Steps: Marginally Stable Problem (tol=10^{-4})

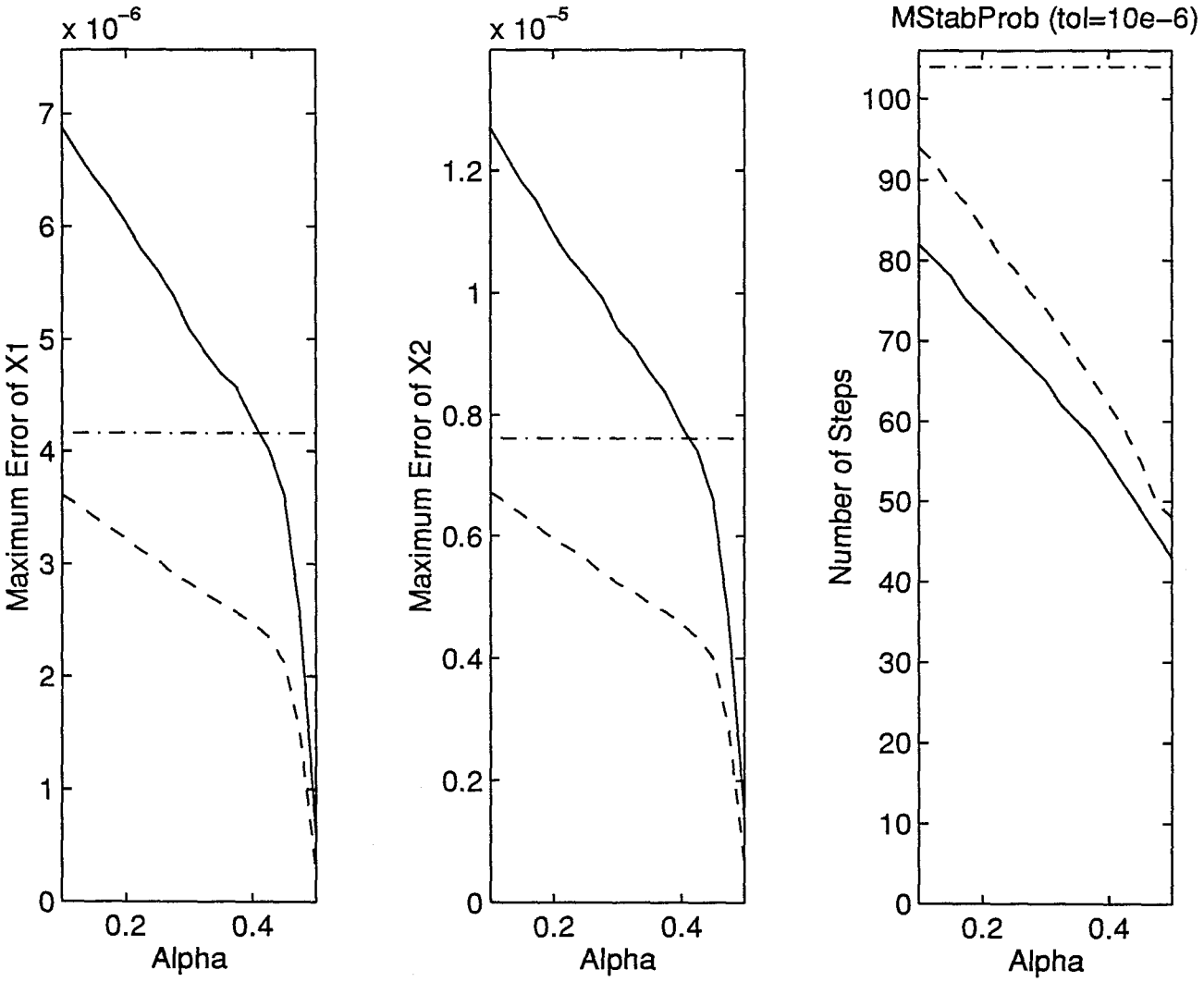

Figure 3.25: Maximum Errors and Steps: Marginally Stable Problem (tol=10^{-6})

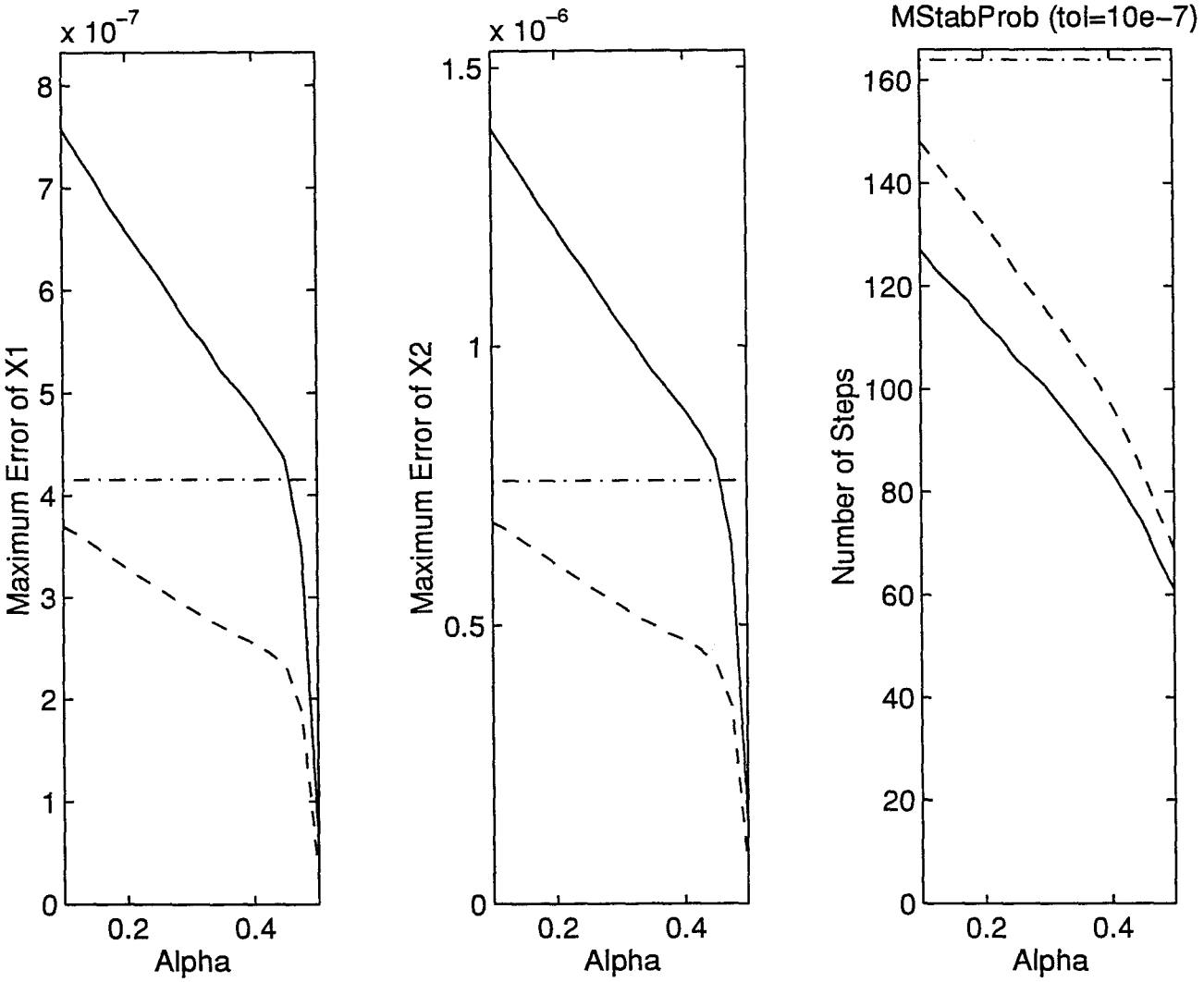

Figure 3.26: Maximum Errors and Steps: Marginally Stable Problem (tol=10^{-7})

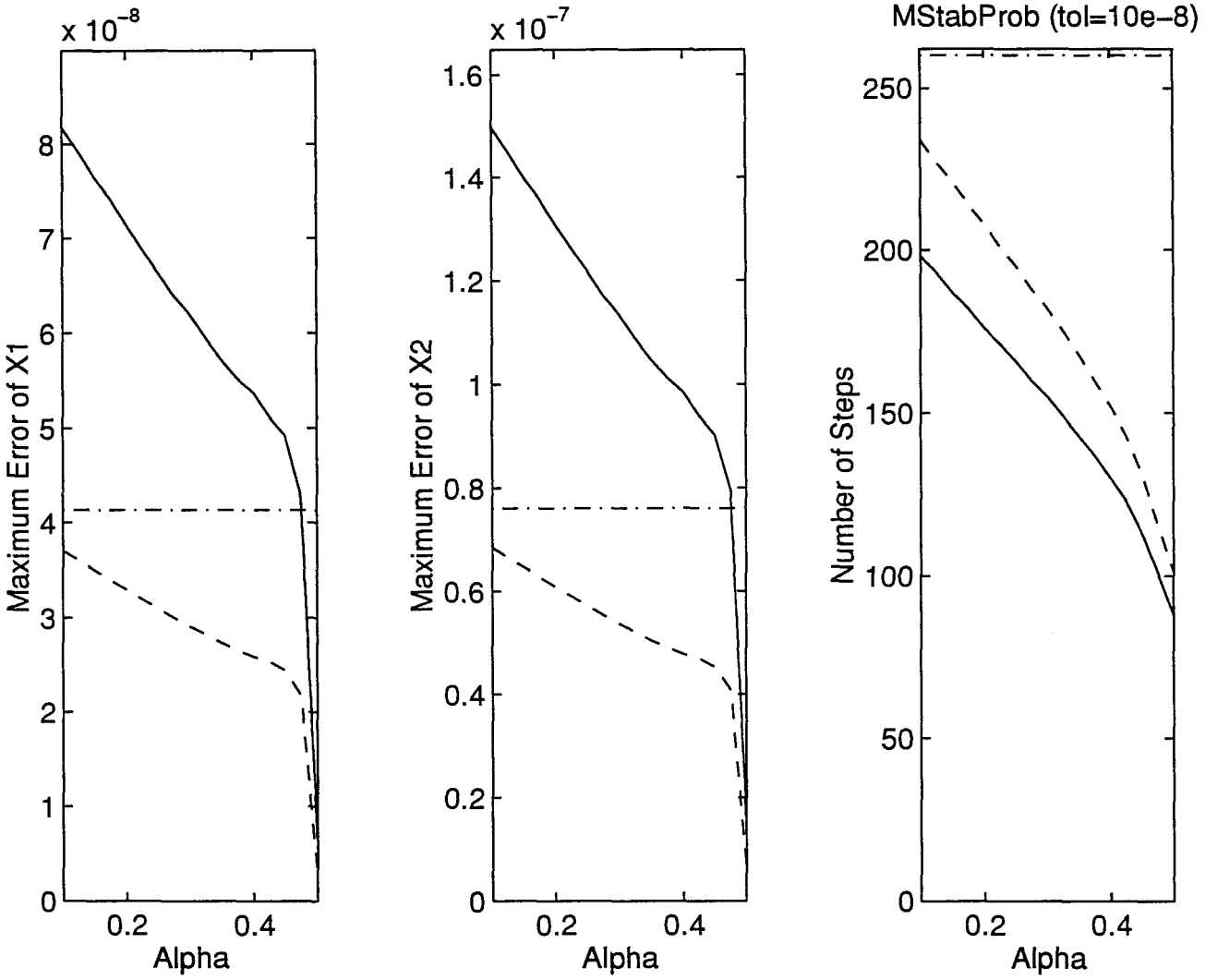

Figure 3.27: Maximum Errors and Steps: Marginally Stable Problem (tol=10^{-8})

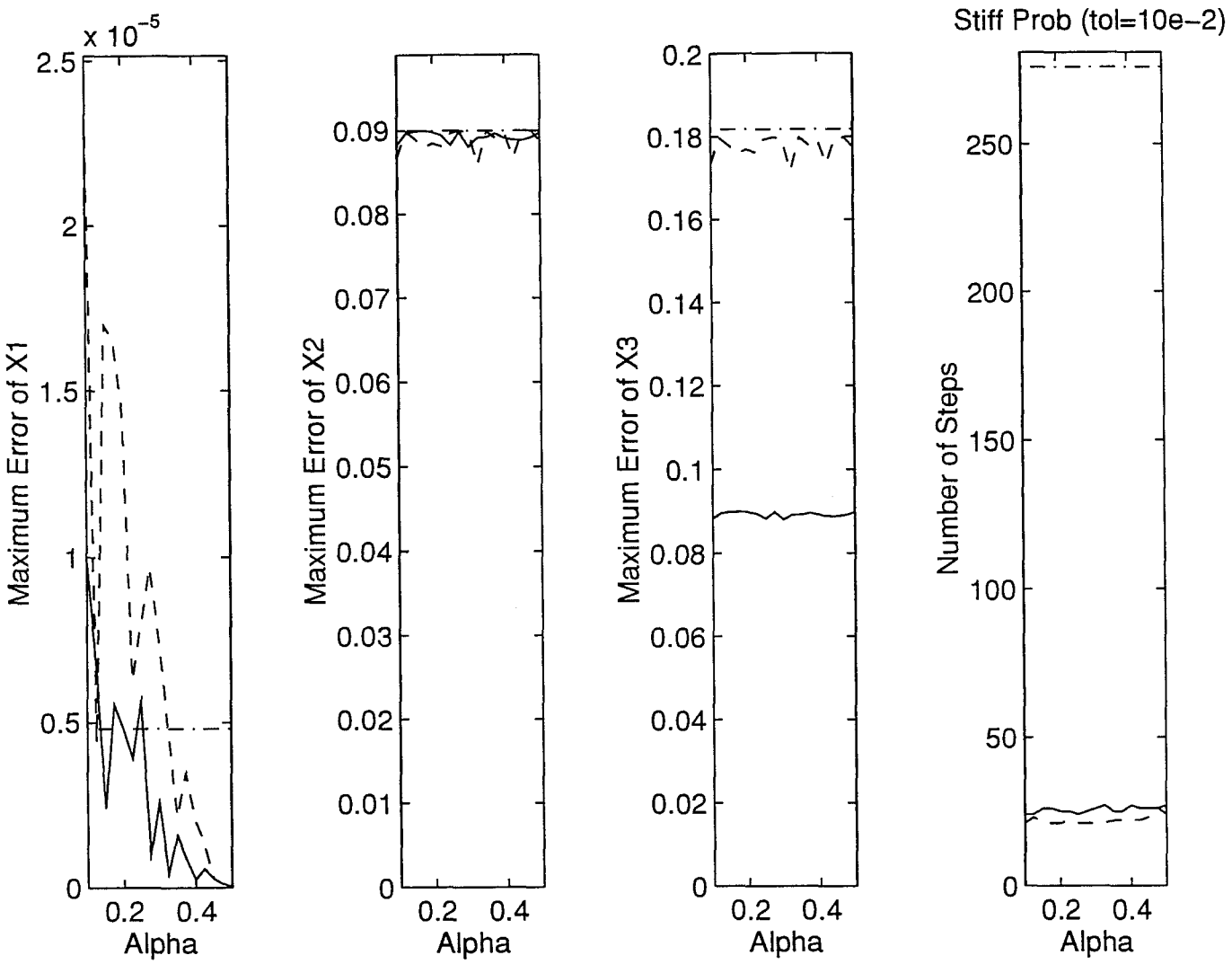

Figure 3.28: Maximum Errors and Steps: Stiff Problem (tol=10^{-2})

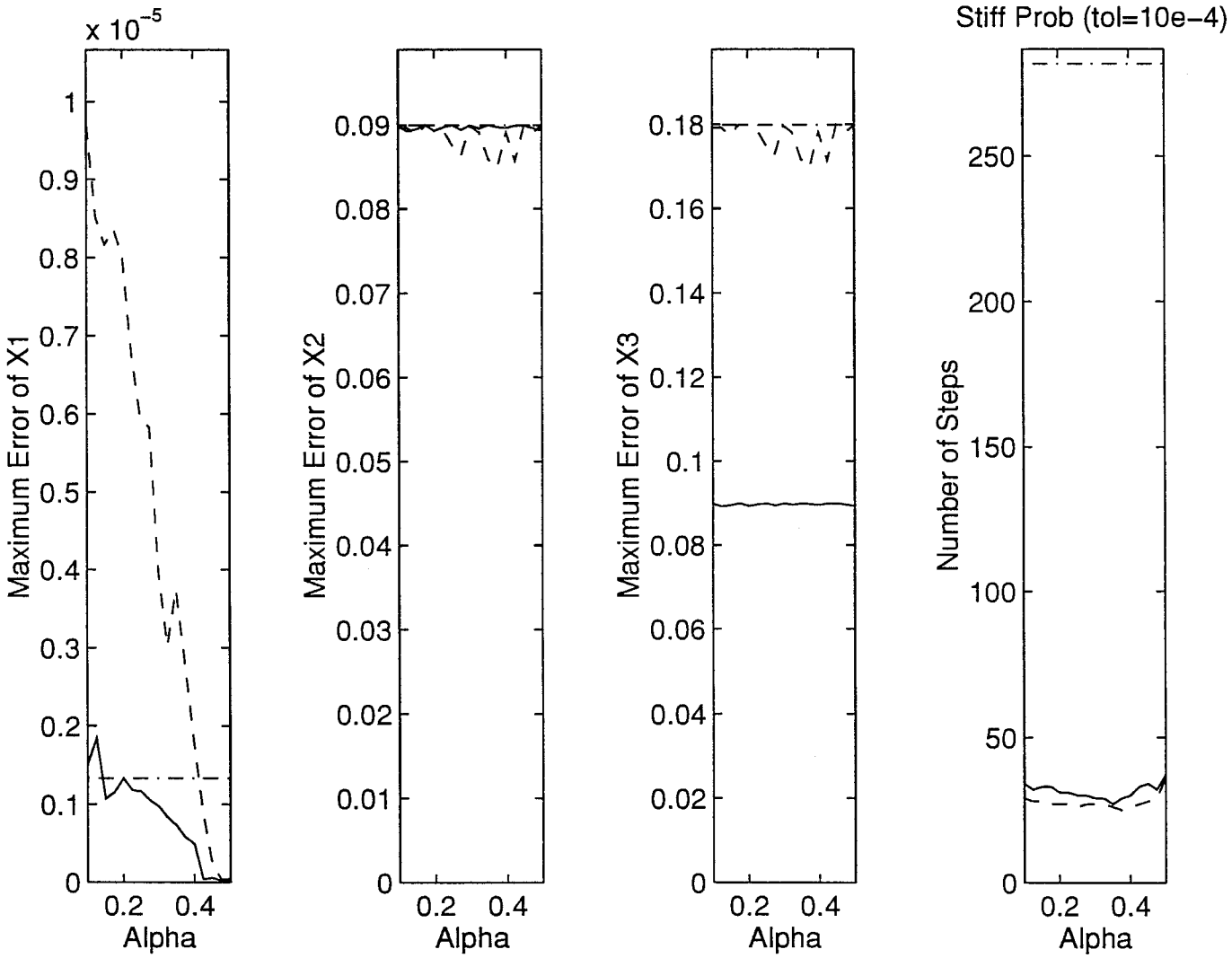

Figure 3.29: Maximum Errors and Steps: Stiff Problem (tol=10^{-4})

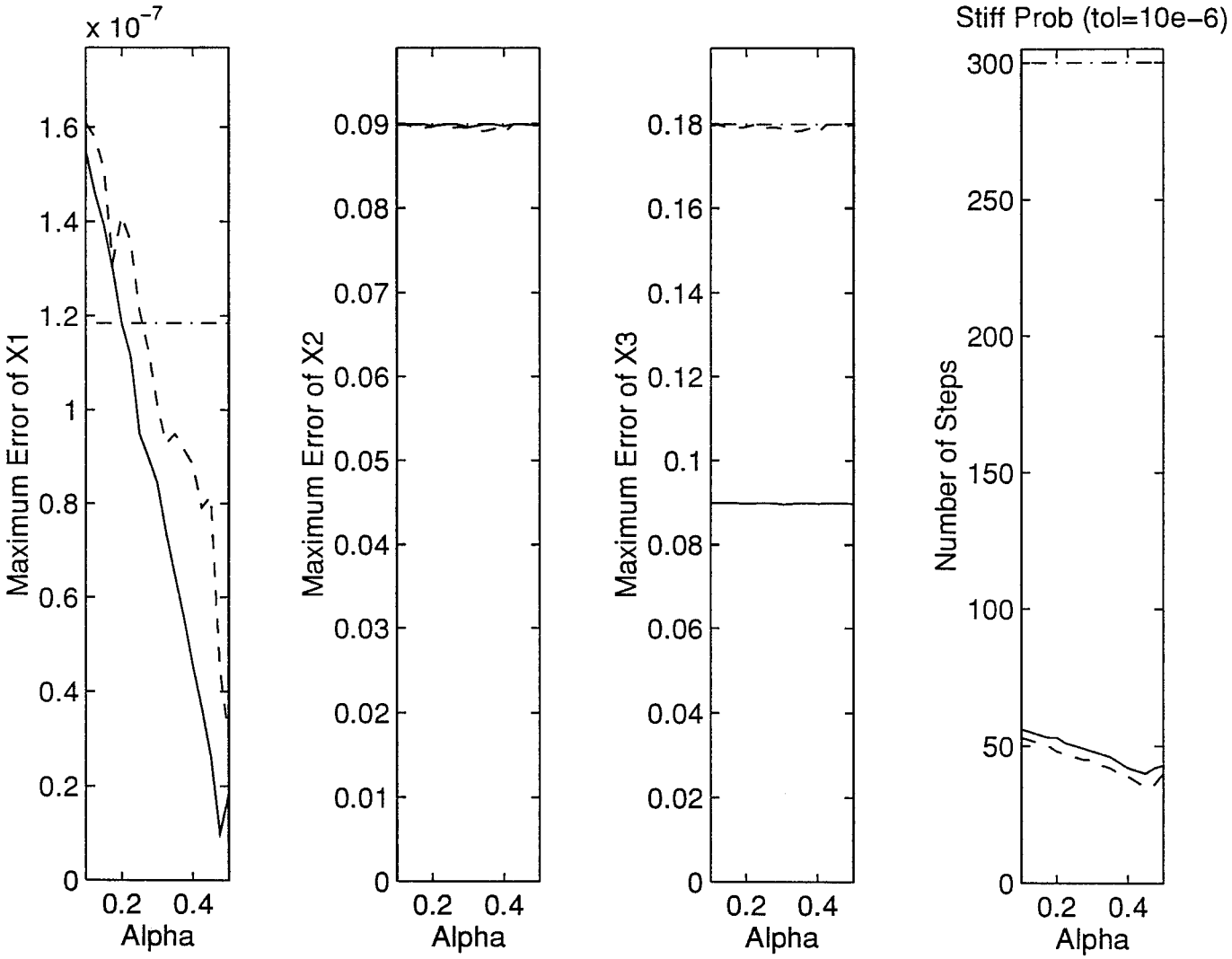

Figure 3.30: Maximum Errors and Steps: Stiff Problem (tol=10^{-6})

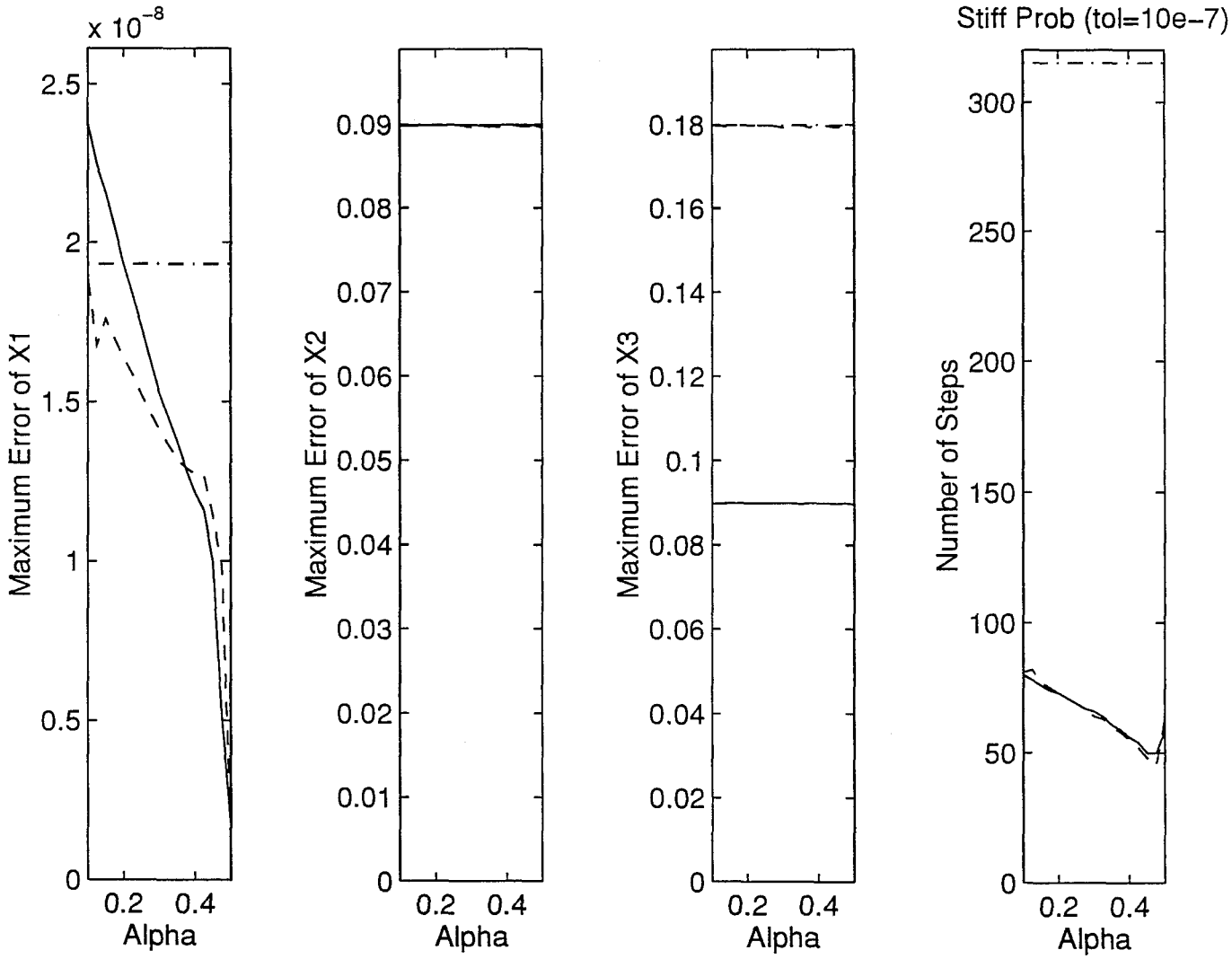

Figure 3.31: Maximum Errors and Steps: Stiff Problem (tol=10^{-7})

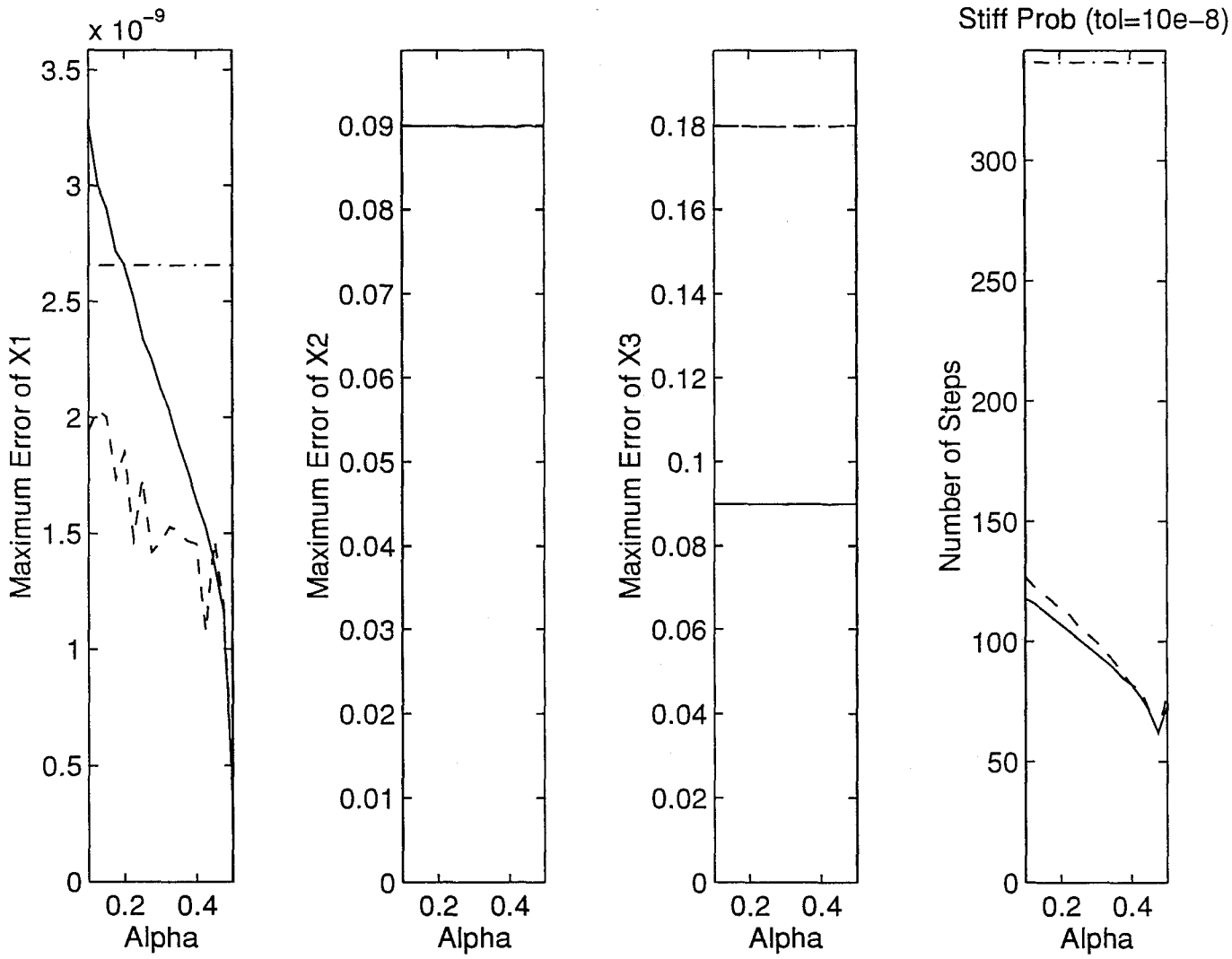

Figure 3.32: Maximum Errors and Steps: Stiff Problem (tol=10^{-8})

3.5 Broyden-Newton Iteration for Nonlinear Systems

For nonlinear problems, during the backinterpolation a numerical method must be used to iterate on the (unknown) "initial" state until the (known) "final" state is hit with sufficient accuracy. To this end, Newton iteration can be applied. Since the commonly used Newton-Raphson iteration requires information of the Hessian matrix (the evaluation of which is usually costly), in our implementation of BI methods we use the Broyden-Newton iteration method that replaces the true Hessian by a much cheaper approximation thereof. To solve equation $f(x) = 0$, starting from an initial guess $x0$, a given maximum number of iterations (NumIter) and a specified accuracy (tol), the iteration process can be described as follows:

Let i=0, iteration = true, B=I, f0=f(x0);

WHILE iteration & (i < NumIter)

ds= inv(B)*f0; x1=x0 - ds; f1=f(x1);

% X-test and F-test

xnorm=max(abs(x0)), dnorm=max(abs(ds)), fnorm=max(abs(f1));

if (dnorm > tol*(1 + xnorm)) or (fnorm > tol)

% Update

ds=ds/norm(ds); df=(f1 - f0)/norm(ds);

B = B + (df - B*ds)*ds^T;

f0=f1; x0=x1; i=i+1;

```
else
iteration = false;
END
if (i > NumIter)
display('Broyden-Newton Method Fails');
```

The detailed implementation of this algorithm has been given in Appendix A.

To test the nonlinear implementation, let us consider the Lotka-Volterra Equation [6]:

$$\dot{x}_1 = -ax_1 + kbx_1x_2, \quad \dot{x}_2 = cx_2 - bx_1x_2 \tag{3.17}$$

where x_1, x_2 are insect populations (of predator and prey), and a, b, c are model parameters.

Figs 3.33-34 represent the simulation results with $a = b = c = 1, k = 0.1$ and initial populations $x_1 = x_2 = 10$, using both matlab ode45 and BI45 with $\alpha = 0.4$ and stepsize control (3.15). Note that in this case, there is no visible difference between the ode45 and BI45 solution. Also note that the Lotka-Volterra model does not approach a constant steady-state value ($x_1 = c/b, x_2 = a/(bk)$). Instead, it approaches a periodic steady-state value, i.e., the solution oscillates. This is a well-known characteristic of Lotka-Volterra equations [6]. Both facts indicate the correctness of our nonlinear implementation of the BI methods.

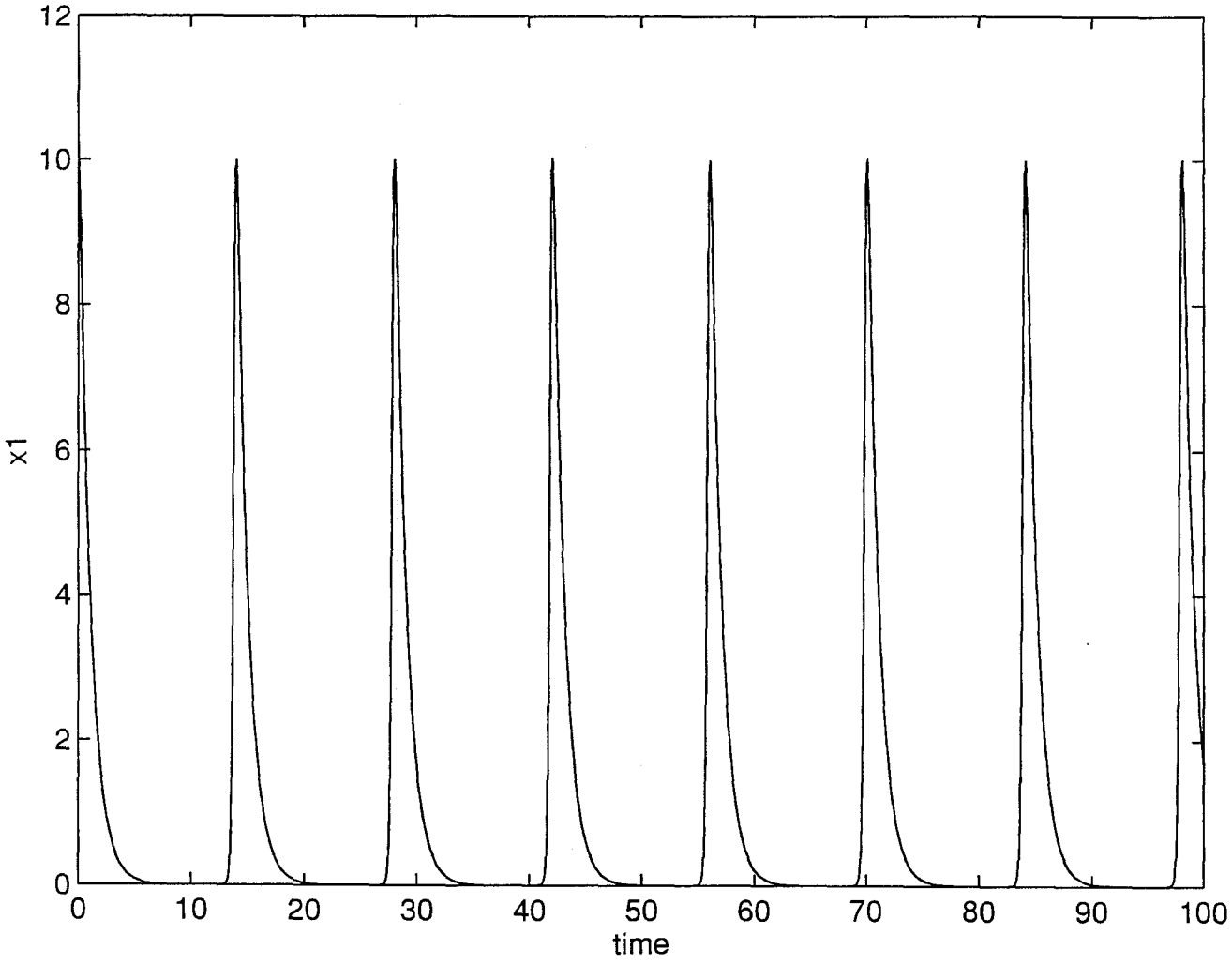

Figure 3.33: Lotka-Volterra Nonlinear Equation: Population x_1

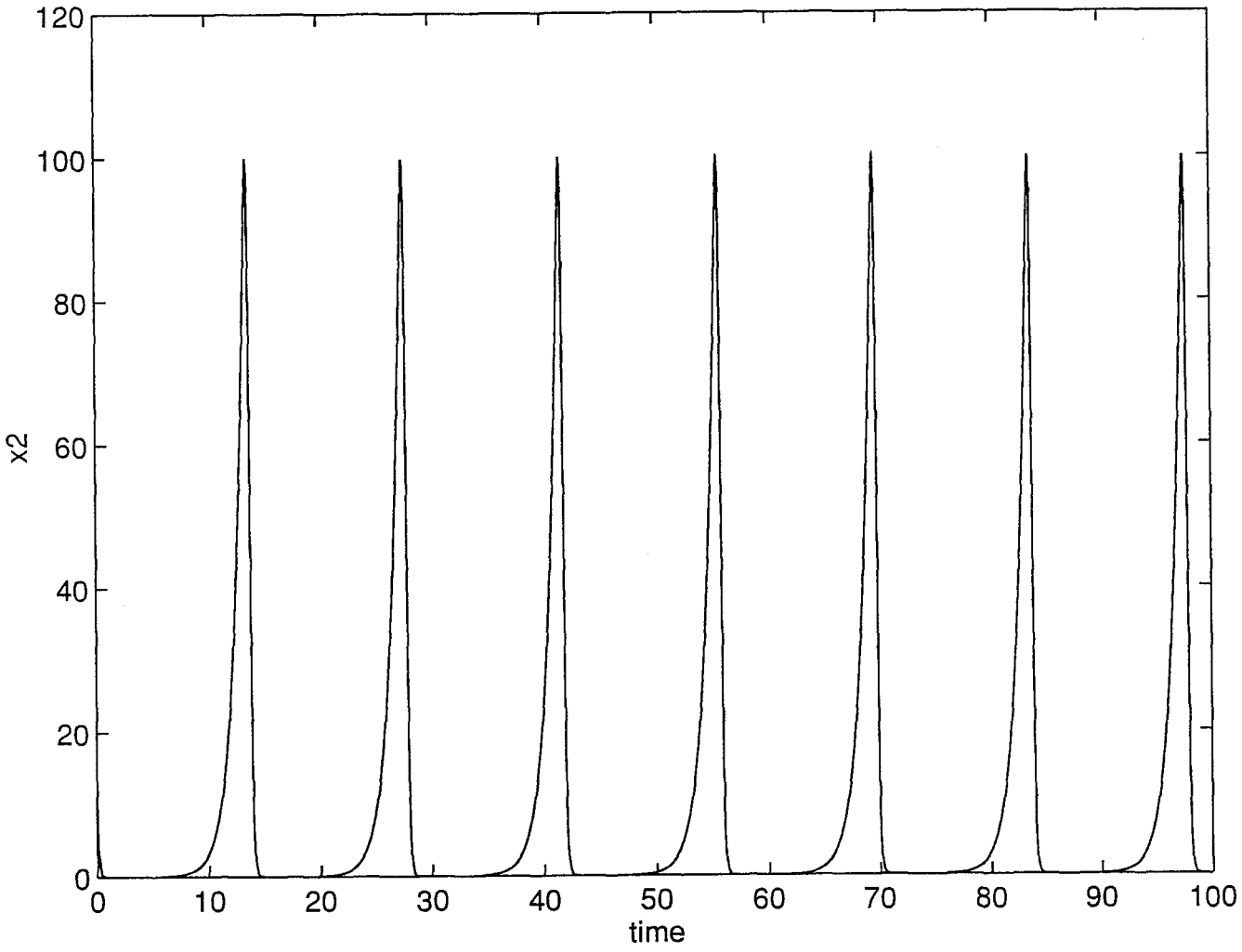

Figure 3.34: Lotka-Volterra Nonlinear Equation: Population x_2

CHAPTER 4

Dynamic Responses of Flexible Manipulators: Euler-Bernoulli Model

The link flexibility of a robotic manipulator must be considered in modeling and control when the manipulator is of large dimension or light weight. Large manipulators play important roles in many applications, such as construction automation, environmental robotics, and space engineering. Lightweight arms are one of the major goals in design of high-performance industrial robotic manipulators, which will lead to higher speed and better energy consumption. However, due to the complexity involved with link deformation and the characteristics of distributed parameter systems, modeling and control of flexible manipulators still remain as a major challenge in robotic research [2].

Over the last decade, a significant effort has been made in modeling and control of one-link flexible manipulators, which are essential for link design and understanding of multiple-link flexible manipulators. The most commonly used deformation model in the current robotic literature is the Euler-Bernoulli beam theory. Based on this theory, various dynamic equations have been formulated for one-link flexible

manipulators [1, 4, 8, 16, 18, 20, 23, 24, 26]. Complete studies of different dynamic equations under various boundary conditions for flexible manipulators were carried out in [1, 25].

However, numerical modeling, especially numerical dynamic responses of a flexible manipulator under various load conditions or control algorithms have received very little attention so far. A major reason is that most researchers have used modal shape approximation methods in the past. Since 1) the derivation of the exact modal shape functions is very complicated, 2) numerical values of higher order modal shape functions cannot be computed easily, and 3) modal shape functions and the corresponding vibration frequencies depend nonlinearly on payloads. In addition, no efficient numerical solutions for flexible manipulators have been presented until now.

The purpose of this chapter is to establish an ODE model for one-link Euler-Bernoulli flexible manipulators using the method of lines and apply BI algorithms to solve them. As we shall show, flexible manipulators provide an ideal case for BI application since their ODEs are marginally stable in the open loop case and stiff in the feedback control case. We have also compared BI methods with the popular *ode45*, an ODE solver based on RK45 in Matlab.

4.1 Basic Equations

Consider a flexible manipulator carrying a tip load. It consists of a flexible beam fixed on a rigid hub in the horizontal plane, as shown in Fig. 4.1, where (x_0, y_0), (x_1, y_1), and (x_2, y_2) are the coordinate systems attached to the base, the hub and the tip load, respectively. It is assumed that initially the neutral longitudinal axis (x_1-axis) of the beam and the x_2-axis of the tip load coincide with the x_0-axis. We also use the following differentiation notation:

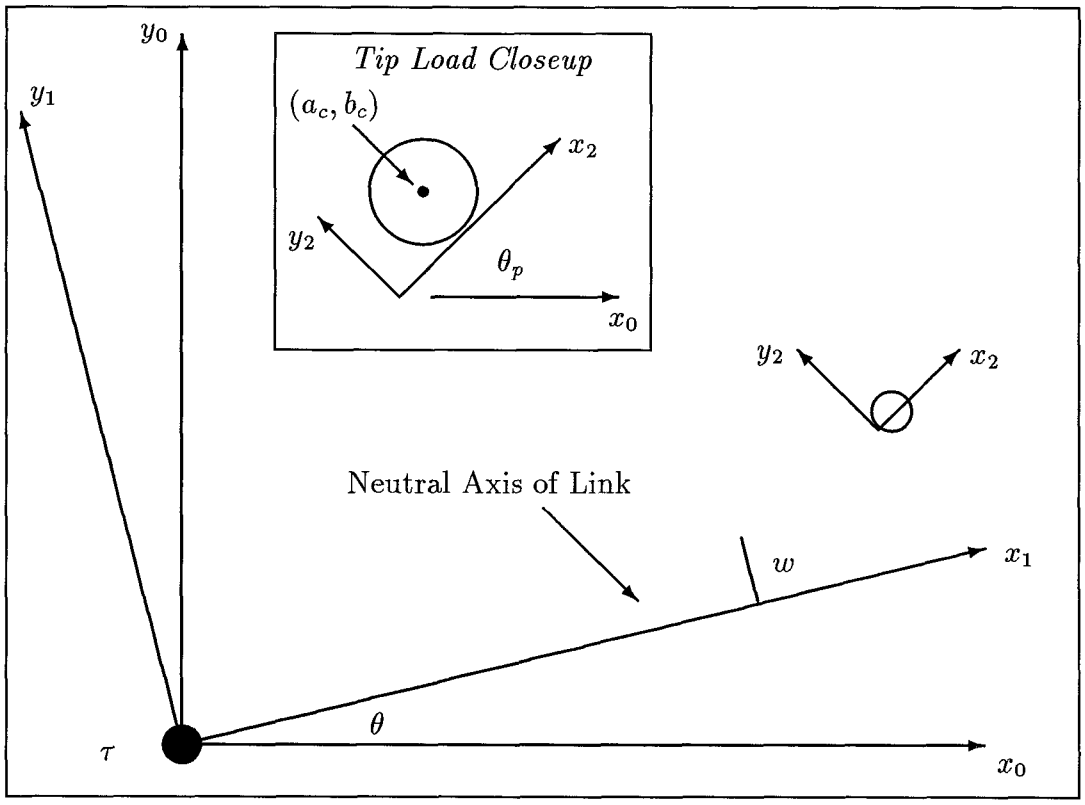

Figure 4.1: Coordinate Systems of a Flexible Manipulator.

$$()' = \frac{\partial()}{\partial x}, \qquad \dot{()} = \frac{\partial()}{\partial t}.$$

The motion of the manipulator system is described by rigid rotation θ of the hub, flexible displacement w and rotation ψ of the beam. In the Euler-Bernoulli theory, the so-called *normal plane assumption* is used, i.e., it is assumed that the entire transverse section of the beam, originally plane, remains plane and normal to the longitudinal axis of the beam after deformation [22]. In this case $\psi = \partial w/\partial x = w'$ and the shear deformation of the beam is completely neglected.

From Fig. 4.1, the base coordinate of a point (x, y) on the beam can be found as,

$$\begin{aligned} x_b &= x\cos\theta - w\sin\theta - y\sin(\psi+\theta), \\ y_b &= x\sin\theta + w\cos\theta + y\cos(\psi+\theta). \end{aligned}$$

Similarly, the base coordinate of a point (a, b) on the tip load is,

$$\begin{aligned} x_p &= L\cos\theta - w(L)\sin\theta + a\cos(\psi+\theta) - b\sin(\psi+\theta), \\ y_p &= L\sin\theta + w(L)\cos\theta + a\sin(\psi+\theta) + b\cos(\psi+\theta); \end{aligned}$$

where L is the length of the beam.

The derivation of dynamic models for flexible manipulators is carried out by using Hamilton's Principle [19], i.e.,

$$\delta \int_{t_0}^{t_f} (T + W - P)dt = 0$$

where T, P, and W are the total kinetic energy, potential energy, and work performed by external forces of the manipulator system, respectively.

The total kinetic energy is calculated as follows,

$$T = \frac{1}{2} I_H \dot{\theta}^2 + T_b + T_p \tag{4.1}$$

where I_H is the rotational inertia of the hub, and T_b and T_p are the kinetic energies of the beam and tip load, respectively. For T_b, we have,

$$T_b = \frac{1}{2} \int_0^L \iint_{S_b} \rho(\dot{x}_b^2 + \dot{y}_b^2) dx_b dy_b ds = \frac{1}{2} \int_0^L \rho[\Delta_x^2 + \Delta_y^2 + S(\dot{\psi} + \dot{\theta})^2] ds \tag{4.2}$$

in which ρ is the mass density per unit length of the beam, S_b the domain occupied by the beam cross section, $S = I/A$, I and A are the moment and area of the beam cross section, and

$$\Delta_x(x) = x\dot{\theta} + \dot{w}(x,t), \qquad \Delta_y(x) = \dot{\theta} w(x,t). \tag{4.3}$$

Note that parameter S reflects the effect of rotatory inertia, i.e., the rotational effect of the cross section of the beam. Ignoring the rotatory inertia (set $S = 0$) implies that one has assumed that the velocity of every point on the same transverse section is identical and equal to the velocity of the point at the neutral axis on that transverse section. For T_p, we have,

$$\begin{aligned} T_p = \quad & \frac{1}{2} \iint_{S_p} \rho_p(\dot{x}_p^2 + \dot{y}_p^2) dx_p dy_p = \frac{M_p}{2}[\Delta_x^2(L) + \Delta_y^2(L)] + \frac{J_p}{2}[\dot{\psi}(L) + \dot{\theta}]^2 \\ + \quad & M_p[\dot{\psi}(L) + \dot{\theta}](G_x \cos\psi + G_y \sin\psi) \end{aligned} \tag{4.4}$$

in which ρ_p is the mass density per unit area, S_p is the domain occupied by the tip load, and

$$M_p = \iint_{S_p} \rho_p dS, \qquad J_p = \iint_{S_p} \rho_p (a^2 + b^2) dS,$$

are the mass and inertia of the tip load, respectively, and

$$G_x = a_c \Delta_x(L) + b_c \Delta_y(L), \qquad G_y = a_c \Delta_y(L) - b_c \Delta_x(L).$$

where (a_c, b_c) is the mass center of the tip load with respect to the local tip load coordinate system. G_x and G_y represent the size effect of the tip load. If one assumes that the tip load is a point of mass M_p and inertia J_p, then $G_x = G_y = 0$.

From beam theory [22], the total potential energy can be found as,

$$P = \frac{1}{2} \int_0^L D\psi'^2 ds \tag{4.5}$$

in which D is the bending rigidity of the beam.

Finally, the work performed by external forces to the manipulator system is,

$$W = \tau\theta \tag{4.6}$$

where τ is the torque applied on the hub.

4.2 Dynamical Equations of Flexible Arms

Substituting Eqs. (4.1-4.6) into Hamilton's Principle, we can get the dynamic motion equations of flexible arms. Note that we drop all deformation terms that are

higher than first order. This is justified since the *small deformation* assumption, in which all the second or higher order displacement and strain terms are ignored [22], has been used in the Euler-Bernoulli beam theory. These simplifications have been adapted in almost all the published literature on modeling of flexible manipulators. As we can see later from the simulation results, experiments indicate that nonlinear terms have noticeable effects on dynamic responses of flexible arms only when motion speeds are extremely high [2].

To make the form of the dynamic equations simpler, we further define the total deflection v of the manipulator as,

$$v(x,t) = w(x,t) + x\theta(t). \tag{4.7}$$

After a tedious process of simplification, we arrive at the following dynamic equations in terms of the total deflection,

$$(Dv'')'' - (\rho S\ddot{v}')' + \rho\ddot{v} = 0, \tag{4.8}$$

$$I_H\ddot{\theta} - Dv''(0) = \tau, \tag{4.9}$$

with boundary conditions,

$$x = 0, \qquad v = 0, \qquad v' = \theta; \tag{4.10}$$

$$x = L, Dv'' + J_p\ddot{v}' + a_cM_p\ddot{v} = 0, (Dv'')' - \rho S\ddot{v}' = M_p(\ddot{v} + a_c\ddot{v}'). \tag{4.11}$$

Note that in the dynamic models presented in [1], neither the rotatory inertia nor the size of the tip load had been considered (i.e., $S = 0$ and $a_c = 0$ were assumed).

For the sake of numerical computation, we introduce the following dimensionless functions, variables, and parameters,

$$\xi = \frac{x}{L}, \quad t_{new} = \frac{t_{old}}{c}, \quad c^2 = \frac{M_0 L^3}{D_0}, \tag{4.12}$$

$$z(\xi) = \frac{v}{L}, \ \alpha(\xi) = \frac{\rho(L\xi)L}{M_0}, \ \beta(\xi) = \frac{D(L\xi)}{D_0}, \ \delta(\xi) = \frac{S(L\xi)}{L^2}, \tag{4.13}$$

$$\mu = \frac{M_p}{M_0}, \quad \eta = \frac{I_H}{M_0 L^2}, \quad \kappa = \frac{J_p}{M_0 L^2}, \quad \zeta = \frac{a_c}{L}. \tag{4.14}$$

where M_0 and D_0 are nominal values of beam mass and bending rigidity.

In terms of these new functions, variables, and parameters, the dynamic equations can be rewritten as follows,

$$(\beta z'')'' - (\alpha\delta\ddot{z}')' + \alpha\ddot{z} = 0, \tag{4.15}$$

$$\eta\ddot{\theta} - \beta(0)z(0)'' = \frac{\tau L}{D_0}; \tag{4.16}$$

with boundary conditions,

$$z(0) = 0, \qquad z' = \theta; \tag{4.17}$$

$$\beta(1)z''(1) + \kappa\ddot{z}'(1) + \zeta\mu\ddot{z}(1) = 0, (\beta z'')'(1) - \alpha\delta\ddot{z}'(1) = \mu[\ddot{z}(1) + \zeta\ddot{z}'(1)]. \tag{4.18}$$

A prime now indicates the differentiation with respect to coordinate ξ, and a dot now indicates the differentiation with respect to time t_{new}.

4.3 Discretization by the Method of Lines

The method of lines is used to approximate partial differential equations described in the previous section with a set of ordinary differential equations. In this

method, we substitute space derivatives by finite differences. To this end, the interval $0 < \xi < 1$ is divided into n uniform segments with $\Delta\xi = h = 1/n$, and space derivatives are approximated by

$$z' = \frac{1}{h}[z(\xi+h,t) - z(\xi,t)], \tag{4.19}$$

$$z'' = \frac{1}{h^2}[z(\xi+h,t) - 2z(\xi,t) + z(\xi-h,t)], \tag{4.20}$$

$$z''' = \frac{1}{h^3}[z(\xi+2h,t) - 3z(\xi+h,t) + 3z(\xi,t) - z(\xi-h,t)], \tag{4.21}$$

$$z'''' = \frac{1}{h^4}[z(\xi+2h,t) - 4z(\xi+h,t) + 6z(\xi,t) - 4z(\xi-h,t) + z(\xi-2h,t)]. \tag{4.22}$$

Define $\xi_i = ih$, and $z_i = z(ih,t)$, $\alpha_i = \alpha(ih)$, $\beta_i = \beta(ih)$, $\delta_i = \delta(ih)$, $i = 0, 1, ..., n$. Clearly, from the first two boundary conditions (4.17),

$$z_0 = 0, \qquad z_{-1} = -\theta/n. \tag{4.23}$$

Therefore, Eq. (4.16) can be approximated by

$$\eta\ddot{\theta} + n\beta_0\theta - n^2\beta_0 z_1 = \frac{\tau L}{D_0} \tag{4.24}$$

For $i = 1$, we have,

$$n^4[\beta_2 z_3 - 2(\beta_2+\beta_1)z_2 + (\beta_2 + 4\beta_1 + \beta_0)z_1 + \beta_0\frac{\theta}{n}] + [\alpha_1 + n^2(\alpha_1\delta_1 + \alpha_0\delta_0)]\ddot{z}_1 - n^2\alpha_1\delta_1\ddot{z}_2 = 0, \tag{4.25}$$

for $i = 2$, we have,

$$n^4[\beta_3 z_4 - 2(\beta_3+\beta_2)z_3 + (\beta_3 + 4\beta_2 + \beta_1)z_2 - 2(\beta_2+\beta_1)z_1] + [\alpha_2 + n^2(\alpha_3\delta_3 + \alpha_2\delta_2)]\ddot{z}_2 - n^2[\alpha_3\delta_3\ddot{z}_3 + \alpha_2\delta_2\ddot{z}_1] = 0, \tag{4.26}$$

and for $i = 3$ to $n-2$,

$$n^4[\beta_{i+1}z_{i+2} - 2(\beta_{i+1}+\beta_i)z_{i+1} + (\beta_{i+1}+4\beta_i+\beta_{i-1})z_i - 2(\beta_i+\beta_{i-1})z_{i-1} + \beta_{i-1}z_{i-2}]$$
$$+[\alpha_i + n^2(\alpha_{i+1}\delta_{i+1}+\alpha_i\delta_i)]\ddot{z}_i - n^2[\alpha_{i+1}\delta_{i+1}\ddot{z}_{i+1} + \alpha_i\delta_i\ddot{z}_{i-1}] = 0. \quad (4.27)$$

Since we have $n+1$ unknowns, $Z = [\theta, z_1, \ldots, z_n]^T$. We still need two more equations. The difference equation for (4.15) at $i = n-1$ and the last boundary condition (4.18) can be written as,

$$n^2[\beta_n z_n'' - 2\beta_{n-1}z_{n-1}'' + \beta_{n-2}z_{n-2}''] + [\alpha_{n-1}+$$
$$n^2(\alpha_{n-1}\delta_{n-1}+\alpha_{n-2}\delta_{n-2})]\ddot{z}_{n-1} - n^2[\alpha_{n-1}\delta_{n-1}\ddot{z}_n + \alpha_{n-2}\delta_{n-2}\ddot{z}_{n-2}] = 0, (4.28)$$

$$\beta_n z_n'' + \kappa\ddot{z}_n' + \zeta\mu\ddot{z}_n = 0, (4.29)$$

$$n\beta_n z_n'' - n\beta_{n-1}z_{n-1}'' - n\alpha_n\delta_n(\ddot{z}_n - \ddot{z}_{n-1}) - \mu[\ddot{z}_n + n\zeta(\ddot{z}_n - \ddot{z}_{n-1})] = 0. (4.30)$$

From Eq. (4.29) we have,

$$\beta_n z_n'' = -\kappa\ddot{z}_n' - \zeta\mu\ddot{z}_n.$$

Substituting this equation into (4.28) and (4.30), we get the last two equations we need,

$$n^4[-\beta_{n-1}z_n + (4\beta_{n-1}+\beta_{n-2})z_{n-1} - 2(\beta_{n-1}+4\beta_{n-2})z_{n-2} + \beta_{n-2}z_{n-3}]$$
$$-n^2[n\kappa + \zeta\mu + \alpha_{n-1}\delta_{n-1}]\ddot{z}_n + [\alpha_{n-1} + n^2(\alpha_{n-1}\delta_{n-1}+\alpha_{n-2}\delta_{n-2})n\kappa]\ddot{z}_{n-1}$$
$$-n^2\alpha_{n-2}\delta_{n-2}\ddot{z}_{n-2} = 0 \quad ,(4.31)$$

$$n^3\beta_{n-1}(z_n - 2z_{n-1} + z_{n-2}) + (n^2\kappa + 2n\zeta\mu + n\alpha_n\delta_n + \mu)\ddot{z}_n$$
$$-(n^2\kappa + n\alpha_n\delta_n + n\mu\zeta)\ddot{z}_{n-1} = 0.(4.32)$$

Eqs. (4.24-4.27, 4.31-4.32) can be written in a matrix form as,

$$M \cdot \ddot{Z} + K \cdot Z = B\frac{\tau L}{D_0}, \tag{4.33}$$

where

$$Z = [\theta, z_1, z_2, \ldots, z_{n-1}, z_n]^T_{(n+1)}, \quad B = [1, 0, 0, \ldots, 0, 0]^T_{n+1}$$

$$M = \begin{bmatrix} \eta & 0 & 0 & 0 \\ 0 & \alpha_1 + n^2(\alpha_1\delta_1 + \alpha_0\delta_0) & -n^2\alpha_1\delta_1 & 0 \\ 0 & -n^2\alpha_1\delta_1 & \alpha_2 + n^2(\alpha_2\delta_2 + \alpha_1\delta_1) & -n^2\alpha_2\delta_2 \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$
\begin{array}{cc}
. & 0 \\
. & 0 \\
. & 0 \\
. & . \\
. & . \\
. & . \\
. & -n^2\alpha_{n-2}\delta_{n-2} \\
. & 0 \\
. & 0
\end{array}
\qquad
\begin{array}{c}
0 \\
0 \\
0 \\
. \\
. \\
. \\
\alpha_{n-2}+n^2(\alpha_{n-2}\delta_{n-2}+\alpha_{n-3}\delta_{n-3}) \\
--n^2\alpha_{n-2}\delta_{n-2} \\
0
\end{array}
$$

$$
\left.
\begin{array}{cc}
0 & 0 \\
0 & 0 \\
0 & 0 \\
. & . \\
. & . \\
. & . \\
-n^2\alpha_{n-1}\delta_{n-1} & 0 \\
\alpha_{n-1}+n^2(n\kappa+\alpha_{n-2}\delta_{n-2}+\alpha_{n-1}\delta_{n-1}) & -n^2(n\kappa+\zeta\mu+\alpha_{n-1}\delta_{n-1}) \\
-n(n\kappa+n\alpha_n\delta_n+\zeta\mu) & n(n\kappa+2\zeta\mu+\alpha_n\delta_n)+\mu
\end{array}
\right]_{(n+1,n+1)}
$$

$$K = \frac{1}{n^4}\begin{bmatrix}
\frac{\beta_0}{n^3} & -\frac{\beta_0}{n^2} & 0 & 0 & 0 & 0 \\
-\frac{\beta_0}{n} & \beta_0+4\beta_1+\beta_2 & -2(\beta_1+\beta_2) & \beta_2 & 0 & 0 \\
0 & -2(\beta_1+\beta_2) & \beta_1+4\beta_2+\beta_3 & -2(\beta_2+\beta_3) & \beta_3 & 0 \\
0 & \beta_2 & -2(\beta_2+\beta_3) & \beta_2+4\beta_3+\beta_4 & -2(\beta_3+\beta_4) & \beta_4 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

$$\begin{bmatrix}
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
\cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot \\
\beta_{n-3} & -2(\beta_{n-2}+\beta_{n-3}) & \beta_{n-1}+4\beta_{n-2}+\beta_{n-3} & -2(\beta_{n-1}+\beta_{n-2}) & \beta_{n-1} \\
0 & \beta_{n-2} & -2(\beta_{n-1}+\beta_{n-2}) & 4\beta_{n-1}+\beta_{n-2}) & -2\beta_{n-1} \\
0 & 0 & \frac{\beta_{n-1}}{n} & -2\frac{\beta_{n-1}}{n} & \frac{\beta_{n-1}}{n}
\end{bmatrix}_{(n+1,n+1)}$$

The state vector is defined as,

$$q = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} = \begin{bmatrix} Z \\ \dot{Z} \end{bmatrix} \tag{4.34}$$

and the corresponding state variable equations are,

$$\dot{q} = Aq + bu \tag{4.35}$$

where

$$A = \begin{bmatrix} 0 & I \\ -M^{-1}K & 0 \end{bmatrix} \qquad b = \begin{bmatrix} 0 \\ M^{-1}B \end{bmatrix} \qquad u = \frac{L\tau}{D_0} \tag{4.36}$$

4.4 Open-Loop Responses: Marginally Stable Problem

For the sake of simplicity, we use a uniform link for the flexible arm. In other words, we have assumed α, β, and δ as constants in our simulation.

For open-loop responses, i.e., control $u(t)$ is a predetermined function and no feedback is used, Eq. (4.35) is a marginally stable system since all its poles are on the imaginary axis. For example, when

$$\eta = 0.01; \qquad \mu = 0.01; \qquad \kappa = 0.01; \qquad \zeta = 0.001;$$

$$\alpha = 1.0, \qquad \beta = 1.0, \qquad \delta = 0.01, \qquad n = 20. \tag{4.37}$$

The poles of the flexible arm are computed as:

$$(1.0e+02)\times\begin{bmatrix} 0.0000-0.0830i \\ 0.0000+0.0830i \\ 0.0000-0.2071i \\ 0.0000+0.2071i \\ 0.0000-0.3391i \\ 0.0000+0.3391i \\ 0.0000-0.5643i \\ 0.0000+0.5643i \\ 0.0000-0.8726i \\ 0.0000+0.8726i \\ 0.0000-1.1987i \\ 0.0000+1.1987i \\ 0.0000-1.5184i \\ 0.0000+1.5184i \\ 0.0000-1.8234i \\ 0.0000+1.8234i \\ 0.0000-2.1113i \\ 0.0000+2.1113i \\ 0.0000-2.3805i \\ 0.0000+2.3805i \end{bmatrix}$$

$$(1.0e+02)\times\begin{bmatrix}0.0000-2.6302i\\0.0000+2.6302i\\0.0000-2.8595i\\0.0000+2.8595i\\0.0000-3.0677i\\0.0000+3.0677i\\0.0000-3.2539i\\0.0000+3.2539i\\0.0000-3.4172i\\0.0000+3.4172i\\0.0000-3.5570i\\0.0000+3.5570i\\0.0000-3.6725i\\0.0000+3.6725i\\0.0000-3.7631i\\0.0000+3.7631i\\0.0000-3.8282i\\0.0000+3.8282i\\0.0000-3.8675i\\0.0000+3.8675i\\0.0000-4.9455i\\0.0000+4.9455i\end{bmatrix}$$

Since BI55 with $\alpha = 0.5$ is good for marginally stable problems, we have used BI55 and stepsize control (3.15) to solve Eq. (4.35).

Figs. 4.2-4.3 represent the hub rotations and tip deflections using BI45 and Matlab ode45 when the input is 1) a pulse: $u(t) = 0.1$ for $0 < t < 0.5$ and $u(t) = 0$ otherwise; 2) a step: $u(t) = 0.1$; 3) a ramp: $u(t) = 0.1t$, respectively. The tolerance is set as *tol*=10e-7, and n=10. Although the difference between BI45 and ode45 are invisible from these figures, they do give different results. Note that ode45 will not converge if the simulation time is long (say > 100 seconds), while BI55 still does. Since the exact solution is not available for the flexible arm, no accuracy comparison is conducted. However, we can consider the number of steps used to find a solution. From Table 1, we can see that BI55 uses fewer steps to find its solution than ode45 does. However, since BI55 requires more computation at each step, CPU times used by the two methods are about the same.

	pulse	step	ramp
BI55	151	112	39
ode45	358	338	125

Table 1: Number of Integration Steps: Open Loop, Euler-Bernoulli Model

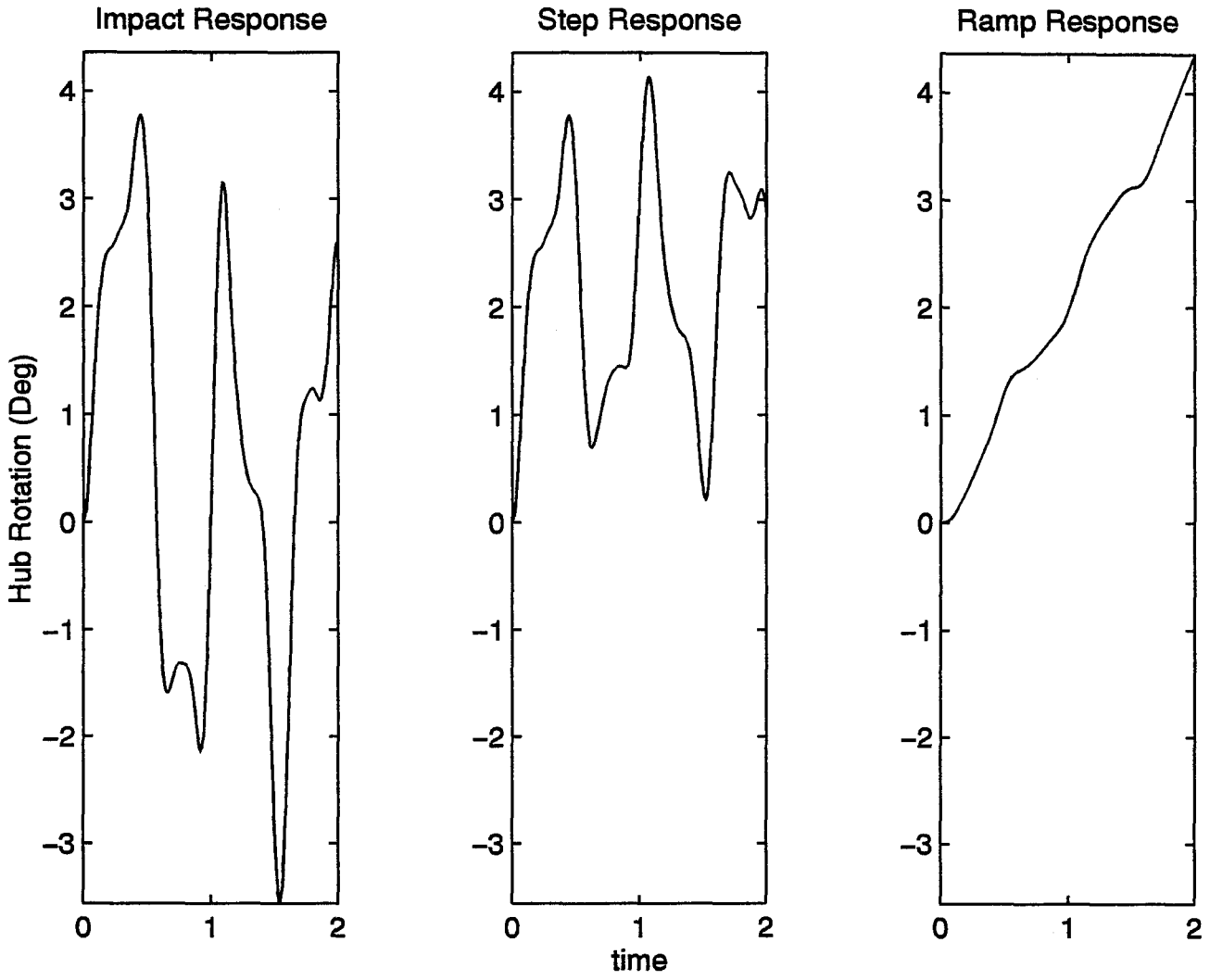

Figure 4.2: Open-Loop Hub Rotations under Various Inputs: Euler-Bernoulli Model

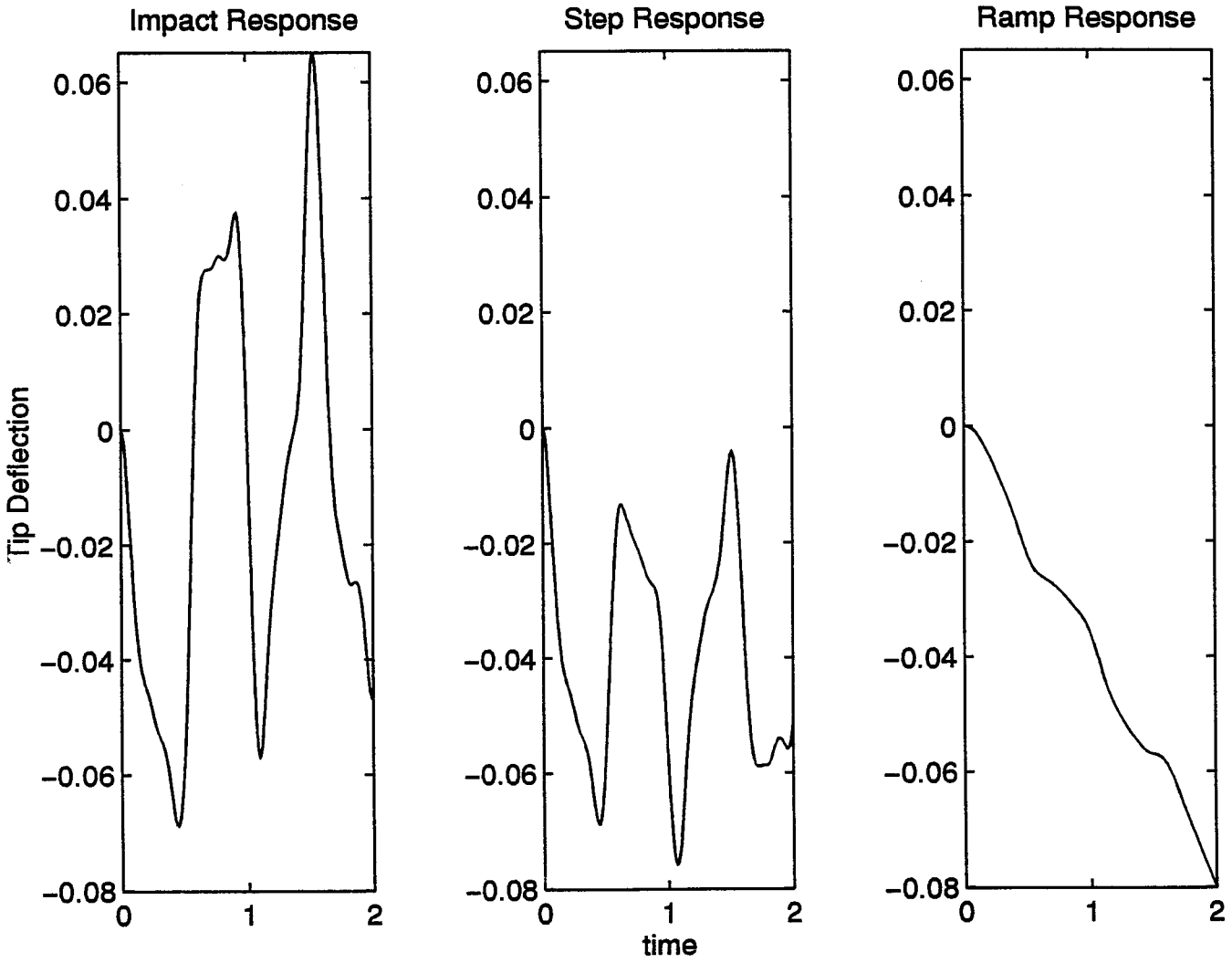

Figure 4.3: Open-Loop Tip Deflections under Various Inputs: Euler-Bernoulli Model

4.5 Closed-Loop Responses: Stiff Problem

For the closed-loop responses, i.e., control $u(t)$ is determined from feedback, Eq. (4.35) is a stiff system since some of its poles are much larger than others. From physics, this can be understood due to the fact that the rigid hub rotation is much faster than the flexible link deformation. The coexistence of fast and slow motion within the same system makes it stiff.

For example, consider the following simple PD control based on hub rotation and tip deflection feedback (since both can be measured):

$$u(t) = k_\theta(\theta_d - \theta) + k_{\dot\theta}(\dot\theta_d - \dot\theta) - k_w w - k_{\dot w}\dot w \tag{4.38}$$

where θ_d is the desired hub position, and $w = z_n - \theta$ is the tip deflection (therefore, the desired value is zero).

Assume the same link and payload parameters as in the previous section, and the following feedback gains are used to make the arm stable:

$$k_\theta = 18.6550; \qquad k_{\dot\theta} = 5.5285; \qquad k_w = 1.5000; \qquad k_{\dot w} = 0.0050.$$

The closed-loop poles in this case become:

$$(1.0e+02) \times \begin{bmatrix} -5.4579 \\ -0.0369 \\ -0.0021 - 0.1244i \\ -0.0021 + 0.1244i \end{bmatrix}$$

$$(1.0e+02)\times\begin{bmatrix}-0.0017-0.2955i\\-0.0017+0.2955i\\-0.0019-0.5415i\\-0.0019+0.5415i\\-0.0017-0.8610i\\-0.0017+0.8610i\\-0.0014-1.1919i\\-0.0014+1.1919i\\-0.0012-1.5140i\\-0.0012+1.5140i\\-0.0010-1.8204i\\-0.0010+1.8204i\\-0.0008-2.1091i\\-0.0008+2.1091i\\-0.0007-2.3789i\\-0.0007+2.3789i\\-0.0005-2.6290i\\-0.0005+2.6290i\\-0.0004-2.8587i\\-0.0004+2.8587i\end{bmatrix}$$

$$(1.0e+02)\times\begin{bmatrix}-0.0003-3.0671i\\-0.0003+3.0671i\\-0.0003-3.2534i\\-0.0003+3.2534i\\-0.0002-3.4169i\\-0.0002+3.4169i\\-0.0001-3.5568i\\-0.0001+3.5568i\\-0.0001-3.6724i\\-0.0001+3.6724i\\-0.0000-3.7630i\\-0.0000+3.7630i\\-0.0005-2.6290i\\-0.0000-3.8282i\\-0.0000+3.8282i\\-0.0000-3.8674i\\-0.0000+3.8674i\\-0.0000-4.9455i\\-0.0000+4.9455i\end{bmatrix}$$

Note that the real parts of the last 4 conjugate pole pairs are -2.1074e-03, -4.7697e-03, -5.2912e-04 and -2.2514e-07, respectively. In terms of real parts, the ratio of the

largest and the smallest is 2.4242e+09 ! In terms of the magnitude of the poles, the ratio of the largest and the smallest is still 147.9106. Therefore, we have a real stiff problem in this case.

Figs. 4.4-4.5 represent the hub rotations and tip deflections using BI45 with $\alpha = 0.47$ and stepsize control (3.15) (solid lines) and Matlab ode45 (dotted lines) when the initial hub position is 90° while the desired position is 0. The tolerance is set as *tol*=10e-7, and n=10. Note that ode45 will not converge with *tol*<10e-7 or simulation time longer than 10 seconds, while BI45 still does. As one can see from these figures, the difference between BI45 and ode45 is quite visible in this case. Note that the BI45 solution is smoother that the ode45 solution. From the mechanics point of view, the BI45 solution is much more reasonable. Again, since the exact solution is not available for the flexible arm in this case, no accuracy comparison is conducted. The numbers of integration steps used to find a solution by BI45 and ode45 are 853 and 1367 respectively.

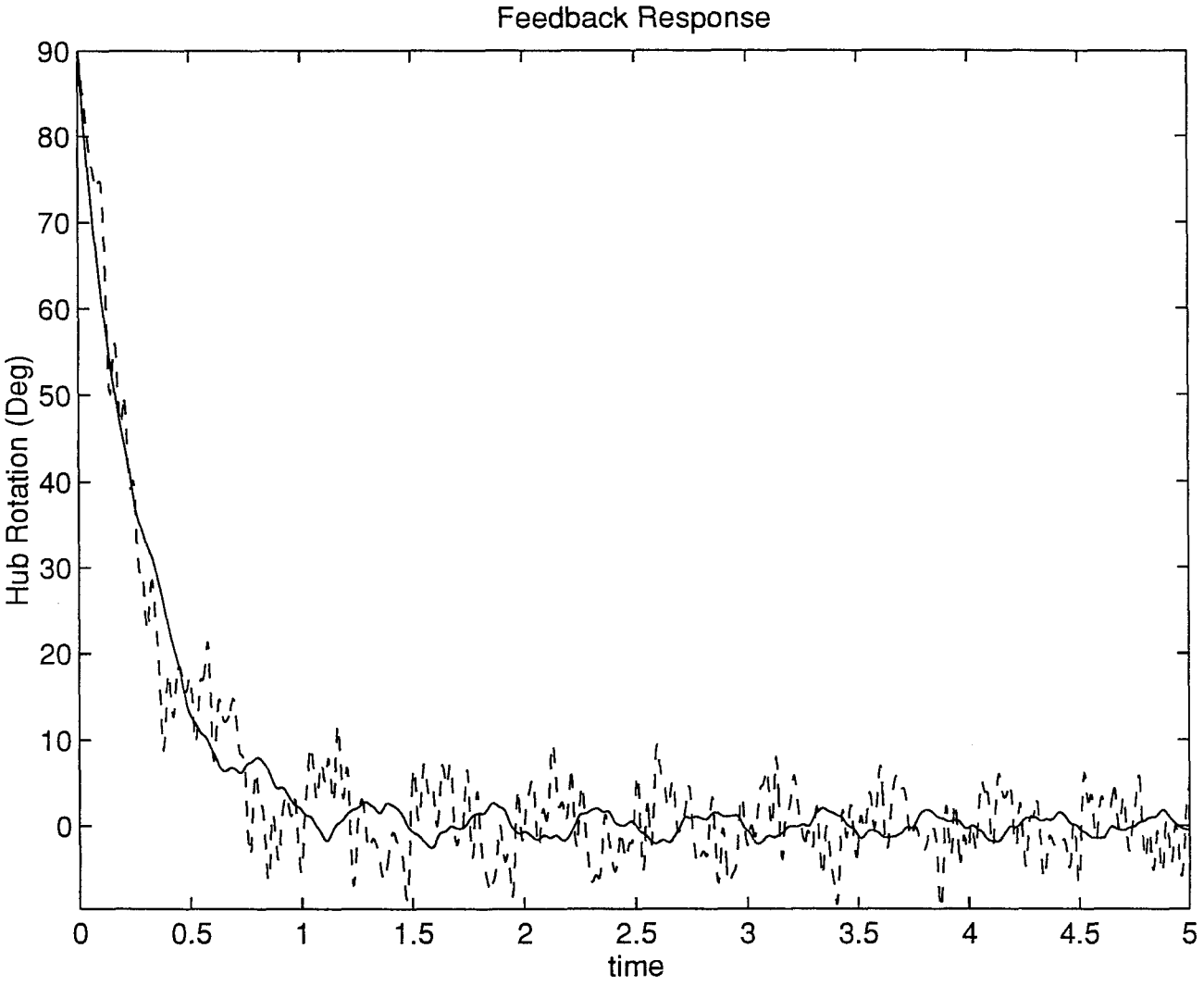

Figure 4.4: Hub Rotation under PD Feedback: Euler-Bernoulli Model

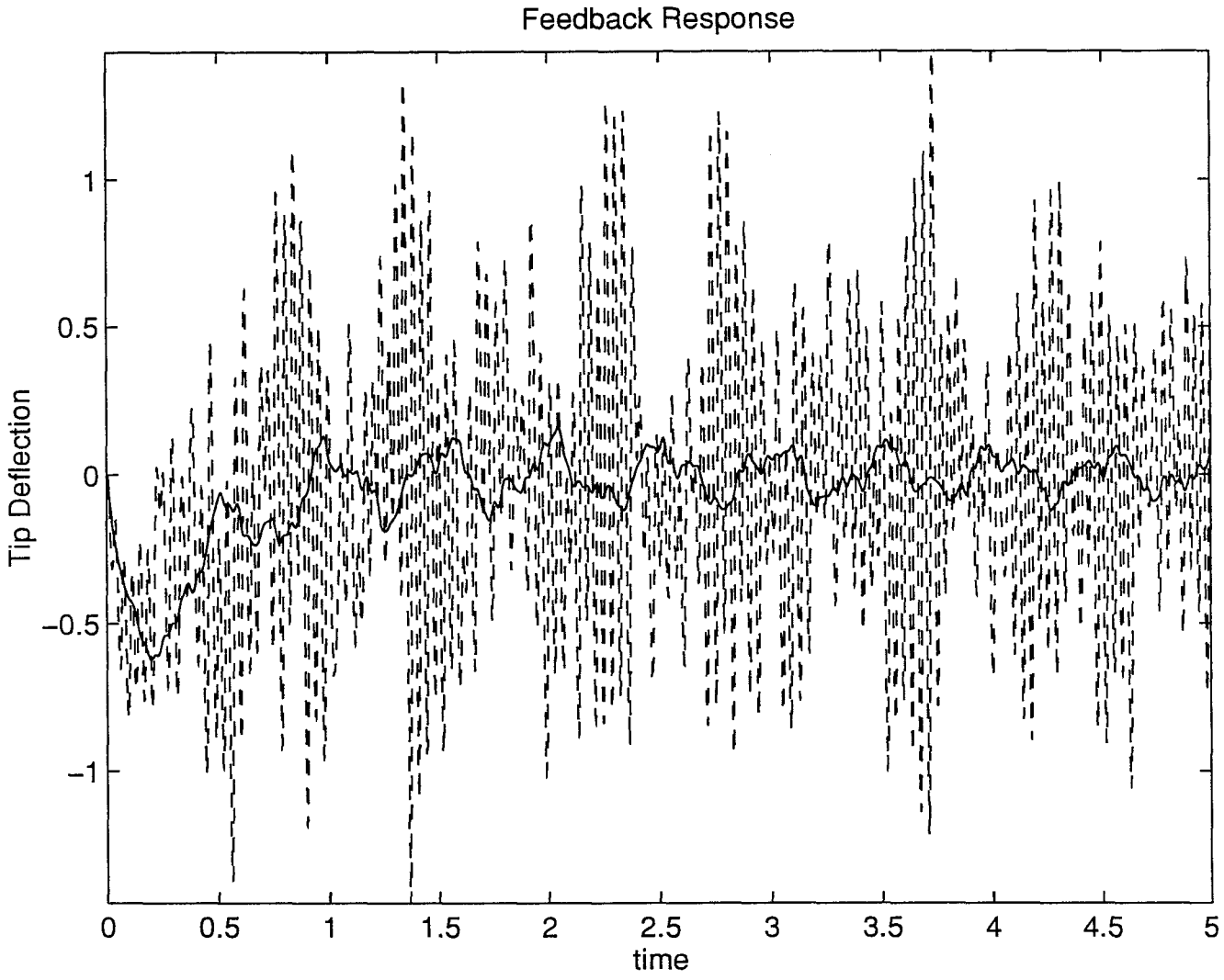

Figure 4.5: Tip Deflection under PD Feedback: Euler-Bernoulli Model

CHAPTER 5

Dynamic Responses of Flexible Manipulators: Timoshenko Model

In this chapter, we continue to study the numerical solution of dynamic responses of flexible manipulators when one has to take their shear deformation into account. In this case, we must use the Timoshenko beam model to describe the flexible arms [3, 9, 10, 21, 25].

5.1 Basic Equations

Again, consider the flexible manipulator given in Fig. 4.1. The motion of the manipulator system is described by rigid rotation θ of the hub, flexible displacement w and rotation ψ of the beam. In Euler-Bernoulli theory, the normal plane assumption is used and we have $\psi = \partial w/\partial x = w'$. The Timoshenko theory takes the effect of shear deformation into account by replacing the normal plane assumption by a more accurate one, the *plane assumption.* The plane assumption states that the entire transverse section of the beam, originally plane, remains plane but may not be normal to the longitudinal axis of the beam after deformation [22]. As a

result, $\psi \neq \partial w/\partial x = w'$, thus beam rotation ψ remains as an *independent* function of deformation.

All basic equations in section 4.2 are still valid here, except that the total potential energy now become,

$$P = \frac{1}{2}\int_0^L [D\psi'^2 + C(\psi - w')^2]ds \tag{5.1}$$

in which D and C are the bending and shear rigidities of the beam. For a beam of uniform cross section, $D = EI$ and $C = kGA$, where E is Young's modulus, G the shear modulus, and k the shape factor [11, 22].

5.2 Dynamical Equations of Flexible Arms

Using Hamilton's Principle, we can get the dynamic motion equations of flexible arms based on the Timoshenko model. To make the form of the dynamic equations simpler, we introduce the total deflection v and rotation α of the manipulator as,

$$v(x,t) = w(x,t) + x\theta(t), \qquad \phi(x,t) = \psi(x,t) + \theta(t). \tag{5.2}$$

In terms of the total deflection and rotation, the Timoshenko dynamic equations of flexible manipulators can be written as follows:

$$(D\phi')' - C(\phi - v') - \rho S\ddot{\phi} = 0, \qquad [C(\phi - v')]' + \rho\ddot{v} = 0, \tag{5.3}$$

$$I_H\ddot{\theta} - D\phi'(0) = \tau, \tag{5.4}$$

with boundary conditions,

$$x = 0, \qquad v = 0, \qquad \phi = \theta; \tag{5.5}$$

$$x = L, \qquad D\phi' + J_p\ddot{\phi} + a_c M_p \ddot{v} = 0, \qquad C(\phi - v') = M_p(\ddot{v} + a_c\ddot{\phi}). \tag{5.6}$$

Besides the dimensionless functions, variables, and parameters introduced in section 4.3, we introduce

$$\gamma(\xi) = \frac{C(x)L^2}{D_0}$$

as the dimensionless shear rigidity function. The dynamic equations now can be rewritten as,

$$(\beta\phi')' - \gamma(\phi - z') - \alpha\delta\ddot{\phi} = 0; \qquad [\gamma(\phi - z')]' + \alpha\ddot{z} = 0, \tag{5.7}$$

$$\eta\ddot{\theta} - \beta(0)\phi'(0) = \frac{\tau L}{D_0}; \tag{5.8}$$

boundary conditions,

$$z(0) = 0, \qquad \phi(0) = \theta \tag{5.9}$$

$$\beta(1)\phi'(1) + [\kappa\ddot{\phi}(1) + \zeta\mu\ddot{z}(1)] = 0, \gamma(1)[\phi(1) - z'(1)] = \mu[\ddot{z}(1) + \zeta\ddot{\phi}(1)]. \tag{5.10}$$

5.3 Discretization by the Method of Lines

Similar to section 4.4, using the method of lines, we can approximate PDEs (5.7-5.10) by a set of ODEs.

Define $\xi_i = ih$, and $z_i = z(ih,t), \phi_i = \phi(ih)$, $\alpha_i = \alpha(ih)$, $\beta_i = \beta(ih)$, $\delta_i = \delta(ih)$, $\gamma_i = \gamma(ih)$, $i = 0, 1, ..., n$. From the first two boundary conditions (5.9),

$$z_0 = 0, \qquad \phi_0 = \theta. \tag{5.11}$$

Therefore, Eq. (5.8) can be approximated by

$$\eta\ddot{\theta} + n\beta_0(\theta - \phi_1) = \frac{\tau L}{D_0} \tag{5.12}$$

For $i = 1$, Eqs. (5.7) is approximated by,

$$n^2[\beta_2\phi_2 - (\beta_2 + \beta_1 + \gamma_1/n^2)\phi_1 + \beta_1\theta] + n\gamma_1 z_1 - \alpha_1\delta_1\ddot{\phi}_1 = 0, \tag{5.13}$$

$$n[\gamma_2\phi_2 - \gamma_1\phi_1] - n^2[\gamma_2 z_2 - (\gamma_2 + \gamma_1)z_1] + \alpha_1\ddot{z}_1 = 0, \tag{5.14}$$

and for $i = 2, ..., n-1$,

$$n^2[\beta_{i+1}\phi_{i+1} - (\beta_{i+1} + \beta_i + \gamma_i/n^2)\phi_i + \beta_i\phi_{i-1}] + n\gamma_i z_i - n\gamma_i z_{i-1} - \alpha_i\delta_i\ddot{\phi}_i = 0, \tag{5.15}$$

$$n[\gamma_{i+1}\phi_{i+1} - \gamma_i\phi_i] - n^2[\gamma_{i+1}z_{i+1} - (\gamma_{i+1} + \gamma_i)z_i + \gamma_i z_{i-1}] + \alpha_i\ddot{z}_i = 0. \tag{5.16}$$

Since we have $2n + 1$ unknowns, $Z = [\theta, z_1, \ldots, z_n, \phi_1, \cdots, \phi_n]^T$. We still need two more equations, which can be obtained from the last two boundary conditions in (5.10). That is,

$$n\beta_n(\phi_n - \phi_{n-1}) + \kappa\ddot{\phi}_n + \xi\mu\ddot{z}_n = 0, \tag{5.17}$$

$$\gamma_n\phi_n - n\gamma_n(z_n - z_{n-1}) - \mu(\ddot{z}_n + \zeta\ddot{\phi}_n) = 0. \tag{5.18}$$

Eqs. (5.12-5.18) can be written in a matrix form as,

$$M \cdot \ddot{Z} + K \cdot Z = B\frac{\tau L}{D_0}, \tag{5.19}$$

As in section 4.3, one can find M, K, and B easily.

Define the state vector as,

$$q = \begin{bmatrix} q_1 \\ \\ q_2 \end{bmatrix} = \begin{bmatrix} Z \\ \\ \dot{Z} \end{bmatrix} \tag{5.20}$$

The corresponding state variable equations are given by,

$$\dot{q} = Aq + bu \tag{5.21}$$

where

$$A = \begin{bmatrix} 0 & I \\ \\ -M^{-1}K & 0 \end{bmatrix} \qquad b = \begin{bmatrix} 0 \\ \\ M^{-1}B \end{bmatrix} \qquad u = \frac{L\tau}{D_0} \tag{5.22}$$

5.4 Open-Loop Responses: Marginally Stable Problem

For the sake of simplicity, we have used a uniform link for the flexible arm. In other words, we have assumed α, β, δ and γ as constants in our simulation.

For open-loop responses, Eq. (5.21) is a marginally stable system since all its poles are on the imaginary axis. For example, when

$$\eta = 0.01; \qquad \mu = 0.01; \qquad \kappa = 0.01; \qquad \zeta = 0.001;$$

$$\alpha = 1.0, \qquad \beta = 1.0, \qquad \delta = 0.01, \gamma = 100, \qquad n = 10. \tag{5.23}$$

The poles of the flexible arms are computed as:

$$(1.0e+02)\times\begin{bmatrix} 0.0000-0.0714i \\ 0.0000+0.0714i \\ 0.0000-0.1652i \\ 0.0000+0.1652i \\ 0.0000-0.2590i \\ 0.0000+0.2590i \\ 0.0000-0.3506i \\ 0.0000+0.3506i \\ 0.0000-0.4497i \\ 0.0000+0.4497i \\ 0.0000-0.5437i \\ 0.0000+0.5437i \\ 0.0000-0.6406i \\ 0.0000+0.6406i \\ 0.0000-0.7307i \\ 0.0000+0.7307i \\ 0.0000-0.8204i \\ 0.0000+0.8204i \\ 0.0000-0.9027i \\ 0.0000+0.9027i \\ 0.0000-0.9820i \end{bmatrix}$$

$$(1.0e+02)\times\begin{bmatrix}0.0000+0.9820i\\0.0000-1.0538i\\0.0000+1.0538i\\0.0000-1.1207i\\0.0000+1.1207i\\0.0000-1.1797i\\0.0000+1.1797i\\0.0000-1.2325i\\0.0000+1.2325i\\0.0000-1.2771i\\0.0000+1.2771i\\0.0000-1.3144i\\0.0000+1.3144i\\0.0000-1.3435i\\0.0000+1.3435i\\0.0000-1.3645i\\0.0000+1.3645i\\0.0000-1.3771i\\0.0000+1.3771i\\0.0000-3.2985i\\0.0000+3.2985i\end{bmatrix}$$

Note that the poles are smaller than the poles calculated from the Euler-Bernoulli Model.

Figs. 5.1-5.2 represent the hub rotations and tip deflections using BI55 ($\alpha = 0.5$) and Matlab ode45 when the input is 1) a pulse: $u(t) = 0.1$ for $0 < t < 0.5$ and $u(t) = 0$ otherwise; 2) a step: $u(t) = 0.1$; and 3) a ramp: $u(t) = 0.1t$, respectively. The tolerance is set as *tol*=10e-7, and n=10. Again, the differences between BI45 and ode45 are invisible but do exist. No accuracy comparison is conducted since the exact solution is not available for the flexible arm. From Table 2, we can see that BI45 uses fewer steps to find its solution than ode45 does.

	pulse	step	ramp
BI55	137	109	63
ode45	342	313	195

Table 2: Number of Integration Steps: Open Loop, Timoshenko Model

5.5 Closed-Loop Responses: Stiff Problem

For closed-loop responses, Eq. (5.21) is a stiff system. For example, consider the following simple PD control based on hub rotation and tip deflection feedback:

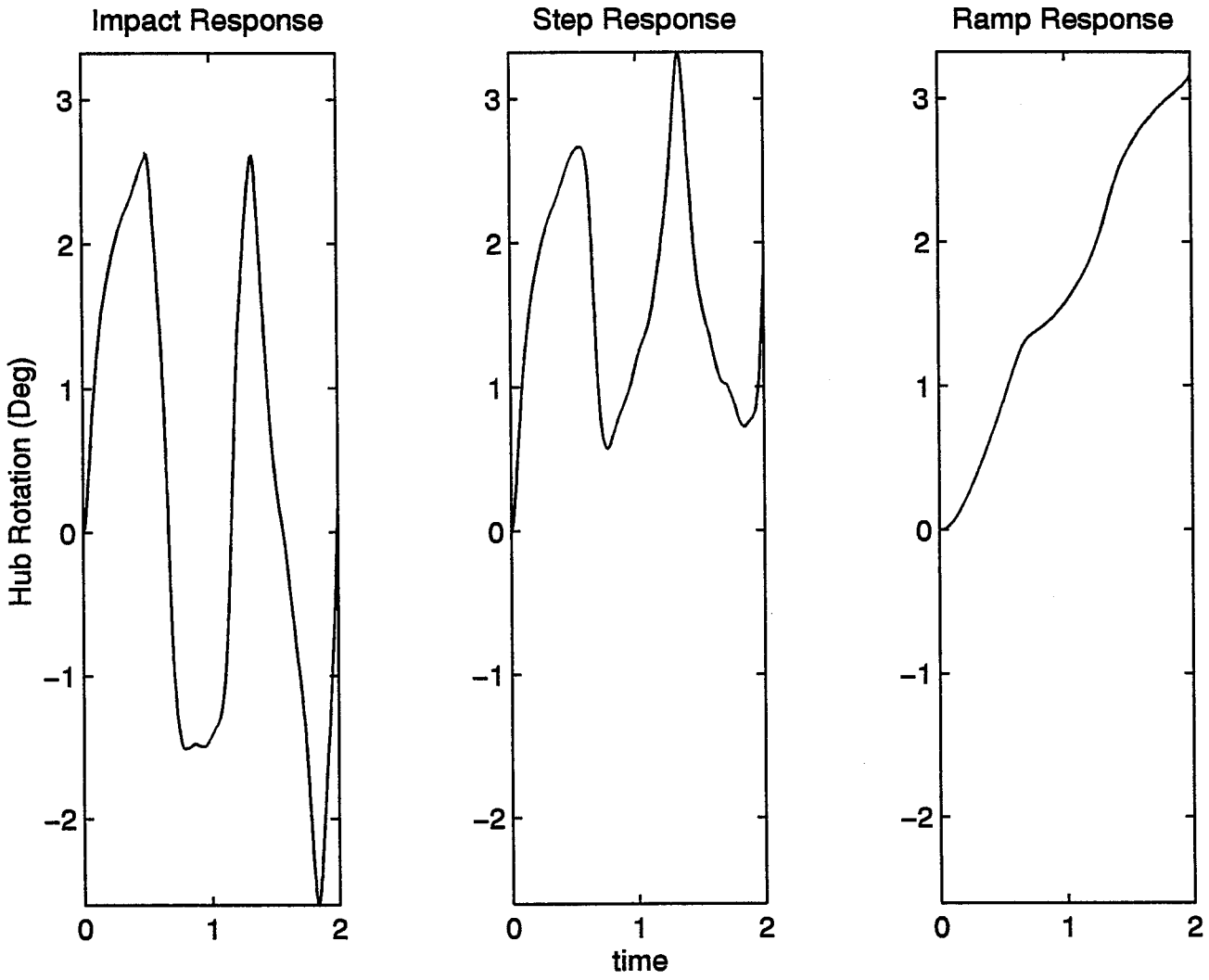

Figure 5.1: Open-Loop Hub Rotations under Various Inputs: Timoshenko Model

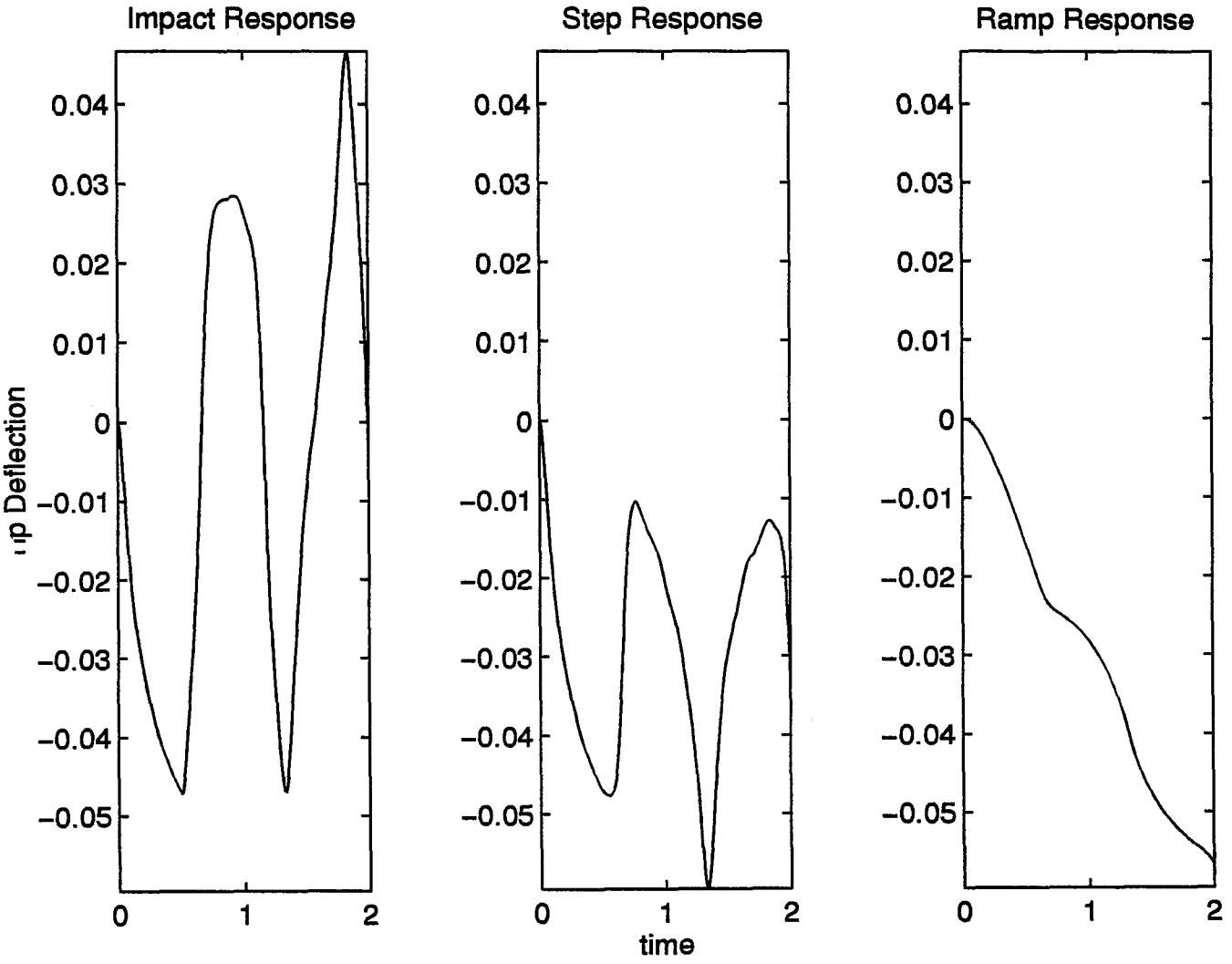

Figure 5.2: Open-Loop Tip Deflections under Various Inputs: Timoshenko Model

$$u(t) = k_\theta(\theta_d - \theta) + k_{\dot\theta}(\dot\theta_d - \dot\theta) - k_w w - k_{\dot w}\dot w \tag{5.24}$$

where θ_d is the desired hub position, and $w = z_n - \theta$ is the tip deflection (therefore, the desired value is zero).

Assume the same link and payload parameters as in the previous section, and the following feedback gains:

$$k_\theta = 16.3225; \qquad k_{\dot\theta} = 4.2234; \qquad k_w = 1.4012; \qquad k_{\dot w} = 0.0061.$$

The closed-loop poles in this case become:

$$(1.0e+02) \times \begin{bmatrix} -4.1243 \\ -0.0454 \\ -0.0028 - 0.1025i \\ -0.0028 + 0.1025i \\ -0.0026 - 0.2136i \\ -0.0026 + 0.2136i \\ -0.0024 - 0.3153i \\ -0.0024 + 0.3153i \\ -0.0023 - 0.4228i \\ -0.0023 + 0.4228i \end{bmatrix}$$

$$(1.0e+02)\times\begin{bmatrix}-0.0021-0.5228i\\-0.0021+0.5228i\\-0.0020-0.6239i\\-0.0020+0.6239i\\-0.0018-0.7177i\\-0.0018+0.7177i\\-0.0016-0.8097i\\-0.0016+0.8097i\\-0.0014-0.8944i\\-0.0014+0.8944i\\-0.0012-0.9753i\\-0.0012+0.9753i\\-0.0010-1.0487i\\-0.0010+1.0487i\\-0.0008-1.1166i\\-0.0008+1.1166i\\-0.0006-1.1768i\\-0.0006+1.1768i\\-0.0005-1.2303i\\-0.0005+1.2303i\end{bmatrix}$$

$$(1.0e+02)\times\begin{bmatrix} -0.0003-1.2757i \\ -0.0003+1.2757i \\ -0.0002-1.3135i \\ -0.0002+1.3135i \\ -0.0001-1.3430i \\ -0.0001+1.3430i \\ -0.0001-1.3643i \\ -0.0001+1.3643i \\ -0.0000-1.3771i \\ -0.0000+1.3771i \\ -0.0000-3.2985i \\ -0.0000+3.2985i \end{bmatrix}$$

Note that the real parts of the last 2 conjugate pole pairs are -1.4045e-03 and -7.4071e-05, respectively. In terms of real parts, the ratio of the largest and the smallest is 5.5680e+06 ! In terms of the magnitude of the poles, the ratio of the largest and the smallest is still 90.8123. Again, we have a real stiff problem in this case.

Figs. 5.3-5.4 represent the hub rotations and tip deflections using BI45 with $\alpha = 0.47$ and Matlab ode45 when the initial hub position is 90° while the desired position is 0. The tolerance is set as *tol*=10e-7, and n=5. As we have seen in the Euler-Bernoulli model, the difference between BI45 (solid lines) and ode45 (dotted

lines) are quite visible in this case. Again, the BI45 solution is much smoother that the ode45 solution. The numbers of integration steps used to find a solution by BI45 and ode45 are 784 and 1273 respectively.

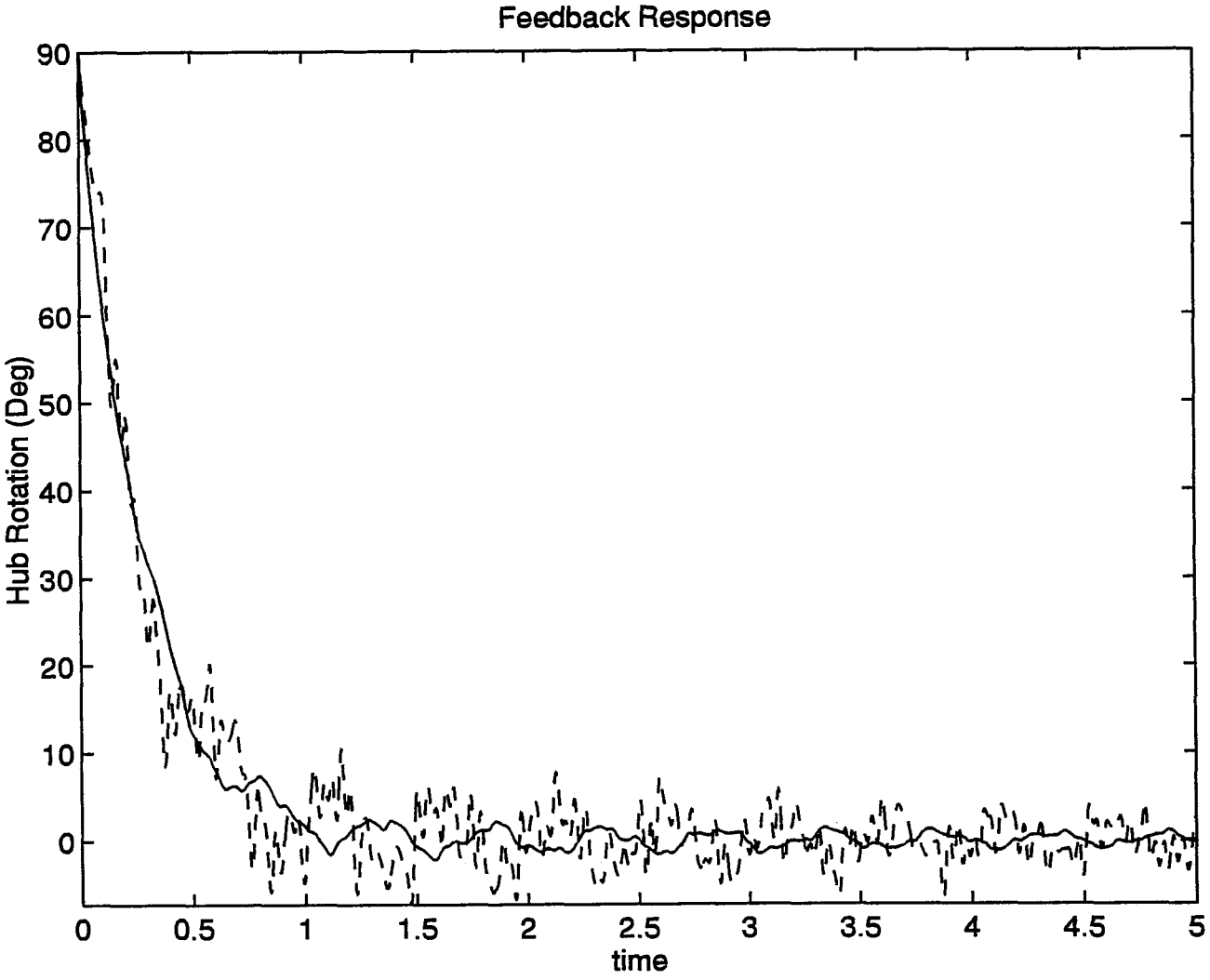

Figure 5.3: Hub Rotation under PD Feedback: Timoshenko Model

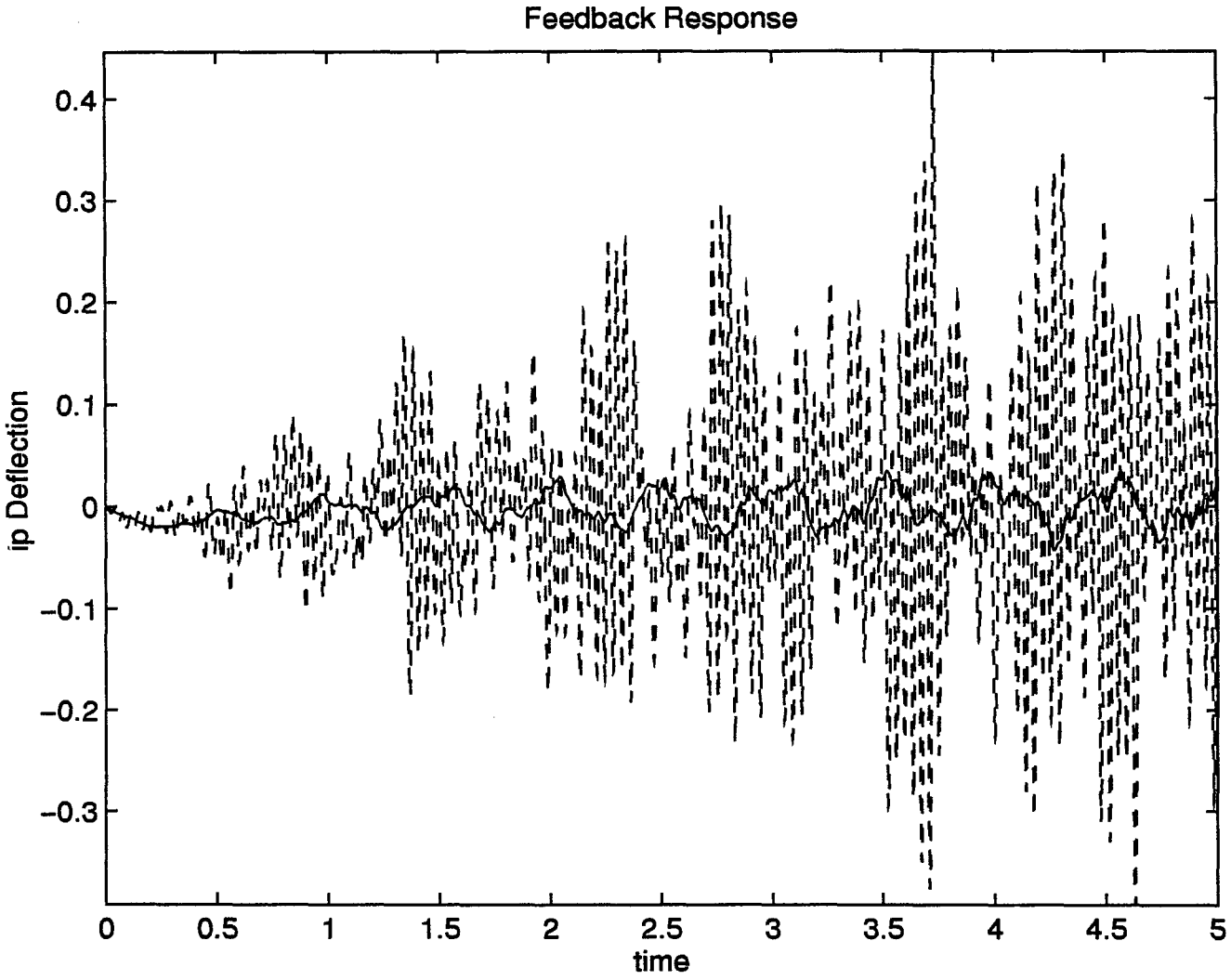

Figure 5.4: Tip Deflection under PD Feedback: Timoshenko Model

CHAPTER 6

Conclusion and Future Work

This thesis has investigated backinterpolation (BI) techniques for finding numerical solutions of continuous-time differential equations. We have looked at various aspects of the proposed BI methods, such as stability domains, accuracy domains, damping and frequency properties, and stepsize control problems. For BI45, it has been found from the stability domain that we have to choose α between 0.3 to 0.47 in order to maintain the L-stability of BI45. We have also found that for BI45 its accuracy domain increases with the value of α. Based on these observations, we conclude that $\alpha = 0.47$ should be used for BI45 in the numerical integration.

Our computer simulation results have indicated that the BI methods are very efficient in solving marginally stable and stiff problems.

For applications, we have first established a numerical model for one-link flexible manipulators based on both Euler-Bernoulli and Timoshenko theories. The partial differential equations for flexible manipulator motion are approximated by the corresponding ordinary differential equations using the method of lines. As indicated by numerical examples, the dynamic equations of flexible manipulators become

marginally stable in the open loop case and stiff in the closed loop case. Simulation results show that both cases can be solved efficiently using the BI methods.

Future research for the BI methods could include:

1. Finding an efficient method for backward iterations that does not need to inverse a matrix or use Gaussian elimination. For example, in [14], an iterative scheme for calculating the inverse of the B matrix in the Broyden-Newton method is suggested as:

$$B_{new}^{-1} = B_{old}^{-1} + (df - B_{old}^{-1} ds) ds^T$$

   However, we can show that this is not correct numerically.

2. Developing interface programs to implement BI methods in simulation languages, such as ACSL, as special ODE solvers for marginally stable and stiff problems.

REFERENCES

[1] F. Bellezze, L. Lanari and G. Ulivi, Exact Modeling of the Flexible Slewing Link, *IEEE 1990 International Conference on Robotics and Automation*, Vol.2, pp.734-739, 1990.

[2] W.J. Book, Structural Flexibility of Motion Systems in the Space Environment, *IEEE Trans. on Robotics and Automation*, RA-9, No.5, pp524-538, 1993.

[3] M. Boutaghou and A. G. Erdman, A Unified Approach for the Dynamics of Beams Undergoing Arbitrary Spatial Motion, *Journal of Vibration and Acoustics-Transactions of the ASEM*, Vol.113, No.4, pp.494-507, 1991.

[4] R. H. Cannon, Jr. and E. Schmitz, Precise Control of Flexible Manipulators, *Robotics Research: The First Int. Symposium*, MIT Press, Cambridge, MA, pp.841-861, 1984.

[5] F. E. Cellier, Backinterpolation Methods for The Numerical Solutions of Ordinary Differential Equations, 1993.

[6] F. E. Cellier, *Continuous System Modeling*, Spring-Verlag, New York, 1991.

[7] F. E. Cellier, *Continuous System Simulation*, Spring-Verlag, New York, 1995.

[8] S. Cetinkunt and W.-L. Yu, Closed-Loop Behavior of a Feedback-Controlled Flexible Arm: A Comparative Study, *Int. J. of Robotics Res.*, Vol.10, No.3, pp.263-275, 1991.

[9] R. W. Clough and J. Penzien, *Dynamics of Structures*, McGraw-Hill, New York, 1975.

[10] G. R. Cowper, On the Accuracy of Timoshenko's Beam Theory, *Jouranl of the Engineering Mechanics Division*, Proc. of the ASCE, Vol.94, EM6, pp.1447-53, 1968.

[11] G. R. Cowper, The Shear Coefficients in Timoshenko's Beam Theory, *Journal of Applied Mechanics*, Vol.33, pp.335-339, 1968.

[12] K. Gustafsson, em Control of Error and Convergence in ODE Solvers, Ph.D Dissertation, Dept. of Automatic Control, Lund Inst. of Technology, Lund, Sweden, 1992.

[13] E. Hairer, S. P. Norsett, and G. Wanner Solving Ordinary Differential Equations I: Nonstiff Problems, *Series in Computational Mathematics*, Vol.8, Springer-Verlag, Berlin, Germany, 1987.

[14] John G. Herriot, *Methods of Mathematical Analysis and Computation*, John Wiley & Sons, New York.London 1963.

[15] R. L. Johnston, *Numerical Methods: A Software Approach*, John Wiley & Sons, New York.Chichester.Brisbane.Toronto.Singapore 1982.

[16] F. Khorrami and Ümit Özgöner, Perturbation Method in Control of Flexible Link Manipulators, *IEEE Int. Conf. on Robotics and Automation*, Vol.1, pp.310-315, 1988.

[17] C. Moler and C. van Loan, Nineteen Dubious Ways to Compute the Exponential of a Matrix," *SIAM Review,* Vol.20 (4), pp.801-836, 1978.

[18] K. A. Morris and M. Vidyasagar, A Comparision of Different Models for Beam Vibrations from the Standpoint of Control Design, *J. of Dynamic Systems, Measurement, and Control*, Vol.112, pp.349-356, 1990.

[19] A. Pars, *A Treatise on Analytical Dynamics*, OX BOW Press, Woodbridge, CT, 1979

[20] T. Sakawa, F. Matsuno, and S. Fukushima, Modeling and Feedback Control of a Flexible Arm, *Journal of Robotic Systems*, Vol.2, pp.453-472, 1985.

[21] S. Timoshenko, D. H. Young, and W. Weaver, Jr., *Vibration Problems in Engineering*, New York: D. Van Nostrand Company, Inc., Third Edition, 1957.

[22] S. Timoshenko and G. H. MacCullough, *Elements of Strength of Materials*, Third Edition. D. Van Nostrand Company, Inc., New York, 1949.

[23] D. Wang and M. Vidyasagar, Control of a Flexible Beam for Optimum Step Response, *IEEE Conf. on Robotics and Automation* (Rayleigh, NC), pp.1567-1572, 1987.

[24] D. Wang and M. Vidyasagar, Passive Control of a Stiff Flexible Link, *International Journal of Robotics Research*, Vol.11, No.3, pp.572-578, 1992.

[25] Fei-Yue Wang and G. G. Guan, Influences of Rotatory Inertia, Shear Deformation and Tip Load on Vibrations of Flexible Manipulators, *Journal of Sound and Vibrations*, Vol.171, No.4, pp.433-452, 1994.

[26] Fei-Yue Wang, C. Wong, and X.Y. Fan, Dynamic Effects of Rotatory Inertia and Shear Deformation of Flexible Robot Arms, in *Advanced Studies in Flexible Robotic Manipulators: Modeling, Design, Control, and Applications*, Editor, F.-Y. Wang, World Scientific Publishers, New Jersey, 1995.

APPENDIX A

Matlab Programs

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   Fat.m                                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function F = Fmat(A,h,algor,alpha)
%
%  This function computes the F-matrix for different integration algorithms
%  55 = BI55
%  45 = BI45
%
I = eye(size(A));
Ah1=A*(1-alpha)*h; Ah2=A*alpha*h;
if algor==55,
  F1=I - Ah1*(I - Ah1*(I/2 - Ah1*(I/6 - Ah1*(I/24 - Ah1/120))));
  F2=I + Ah2*(I + Ah2*(I/2 + Ah2*(I/6 + Ah2*(I/24 + Ah2/120))));
  F=inv(F1)*F2;
elseif algor==45,
  F1=I - Ah1*(I - Ah1*(I/2 - Ah1*(I/6 - Ah1*(I/24 - Ah1/120))));
  F2=I + Ah2*(I + Ah2*(I/2 + Ah2*(I/6 + Ah2/24)));
  F=inv(F1)*F2;
else
  F=I;
  end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   CalMaxH.m                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function hmax = CalMaxH(alpha,algor,hmn0,hmx0,theta)
%
%  This function computes the maximal step size for an
%  arbitrary algorithm, and a given alpha value.
%
radtheta = theta*pi/180;
A = [0 1; -1 +2*cos(radtheta)];

maxerr = 1e-6;
err = 100;
hmn = hmn0;
hmx = hmx0;
while err > maxerr,
  h = (hmn + hmx)/2;
  F = Fmat(A,h,algor,alpha);
  lmax = max(abs(eig(F)));
  err = lmax - 1;
  if err > 0,
    hmn = h;
  else,
    hmx = h;
    end,
  err = abs(err);
  if ((hmx-hmn) < 1.0e-6) & (err > maxerr)
    h = -10.0;
    err = 0.0;
    end,
  end
hmax = h;
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   Stab.m                                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [x,y]=stab(algor, alpha)
%  Find stability domain for BI
%
thetai = 0;
%
%  Loop over theta
%
hvec = [];
avec = [];
terminate = -1;
theta  = thetai;
dtheta = 5;
hmn = 0;
hmx = 100;
while terminate < 0,
  hmax = CalMaxH(alpha,algor,hmn,hmx,theta);
  if hmax >= 0.0,
    avec = [ avec ; theta ];
    hvec = [ hvec ; hmax ];
    hmx = 1.5*hmax;
    theta = theta + dtheta;
  else
    hmx = 2.5*hmx;
    if hmx > 100000
      disp('Check CalMaxH for Error !');
      end;
    end;
  if theta > 180
     terminate = 1;
     end,
  end,
avec = [ avec ;180];
hvec = [ hvec ; 0 ];
%
%  Convert to carthesian coordinates
%
radtheta = avec*pi/180;
x = + hvec .* cos(radtheta);
y = + hvec .* sin(radtheta);
%
%  Duplicate to get third quadrant also
%
ln = length(x);
xx = + x(ln:-1:1);
yy = - y(ln:-1:1);
x = [ x ; xx ];
y = [ y ; yy ];

end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   PlotStab.m                                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
algor=45;
al=[0.1,0.2,0.3,0.4]; %,0.45,0.49];
%al=[0.41,0.42,0.43,0.44,0.45]; %,0.45,0.49];
%al=[0.46,0.47,0.48,0.49,0.50]; %,0.45,0.49];
savefile=['save stab01t04' num2str(algor) ' al XX YY'];
%savefile=['save stab041t045' num2str(algor) ' al XX YY'];
%savefile=['save stab045t049' num2str(algor) ' al XX YY'];
num=length(al);

XX=[];
YY=[];
for i=1:num
    alpha=al(i);
    x=[]; y=[];
    [x,y]=stab(algor,alpha);
    XX=[XX,x];
    YY=[YY,y];
  end

xmax=max(max(XX));
ymax=max(max(YY))+10;

plot(XX,YY) %,[0 0], [-ymax ymax]);
title(['Stability Domain of BI' num2str(algor)]);
%axis([-5 xmax -ymax ymax]);
grid;
xlabel('Re(lambda*h)');
ylabel('Im(lambda*h)');
%gtext(['alpha=0.1,0.2,0.3,0.4']); % num2str(al(1))]);
%gtext(['alpha=0.41,0.42,0.43,0.44,0.45']); % num2str(al(1))]);
%gtext(['alpha=0.46,0.47,0.48,0.49,0.50']); % num2str(al(1))]);
gtext(['alpha=' num2str(al(1))]);
for i=2:num
    gtext(num2str(al(i)));
  end

eval(savefile);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    Accuracy.m                                  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [x,y]=Accuracy(algor, alpha, x0)
%  Find accuracy domain for BI
%
%  Loop over theta
%
hvec = [];
avec = [];
hmn = 0;
hmx = 1;
tol=10e-6;

for theta =0:5:180
  hmax = FindMaxH(alpha,algor,hmn,hmx,theta, x0, tol);
  avec = [ avec ; theta ];
  hvec = [ hvec ; hmax ];
  if hmax > 0
    hmx = 2*hmax;
  else
    hmx = 1;
```

```
  end;
end,

%
%  Convert to carthesian coordinates
%
radtheta = avec*pi/180;
x = + hvec .* cos(radtheta);
y = + hvec .* sin(radtheta);
%
%  Duplicate to get third quadrant also
%
ln = length(x);
xx = + x(ln:-1:1);
yy = - y(ln:-1:1);
x = [ x ; xx ];
y = [ y ; yy ];
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%    FindMaxH.m                                  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function hmax = FindMaxH(alpha,algor,hmn0,hmx0,theta, x0, tol)
%
%  This function computes the maximal step size for an
%  arbitrary algorithm, a given alpha value, a given
%  initial condition, and a given accuracy (tol).
%
radtheta = theta*pi/180;
cc=cos(radtheta);
ss=sin(radtheta);
A = [0 1;-1 2*cc];
tf=10;

if ss ~= 0
   c1=x0(1); c2=(x0(2)-x0(1)*cc)/ss;
   dc1=cc*c1+ss*c2; dc2=cc*c2-ss*c1;
else
   c1=x0(1); c2=x0(2)-x0(1)*cc;
   dc1=cc*c1+c2; dc2=cc*c2;
end

err = 100;
hmn = hmn0;
hmx = hmx0;
while err > 10e-6
  h = (hmn + hmx)/2;
  F = Fmat(A,h,algor,alpha);

  n=round(tf/h);
  MM=F;
  acc=0;
  for i=1:n
  if ss ~= 0
     et=exp(cc*h*i); ct=cos(ss*h*i); st=sin(ss*h*i);
  else
     et=exp(cc*h*i); ct=1; st=h*i;
  end

  anal=et*[c1*ct+c2*st; dc1*ct+dc2*st];
      ddd=norm(MM*x0 - anal);
```

```
%DDDDDDD=ddd
      if ddd > acc
         acc= ddd;
      end
      MM=MM*F;
   end;

  if acc > tol,
    hmx = h;
  else
    hmn = h;
  end;

% treat stepsize less than 10e-6 as zero.
  if h < 10e-6
     hmn=0; hmx=0;
  end;
  err=hmx-hmn;

end

hmax = (hmn + hmx)/2;
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%     PlotAcc.m                                   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


algor=55;
%al=[0.1];
al=[0.1,0.2,0.3,0.4]; %,0.45,0.49];
%al=[0.41,0.42,0.43,0.44,0.45]; %,0.45,0.49];
%al=[0.46,0.47,0.48,0.49,0.50]; %,0.45,0.49];
savefile=['save acc01t04_x001' num2str(algor) ' al XX YY'];
%savefile=['save stab041t045' num2str(algor) ' al XX YY'];
%savefile=['save stab045t049' num2str(algor) ' al XX YY'];
num=length(al);

x0=[0;1];
XX=[];
YY=[];
for i=1:num
    alpha=al(i);
    x=[]; y=[];
    [x,y]=Accuracy(algor,alpha,x0);
    XX=[XX,x];
    YY=[YY,y];
  end

xmax=max(max(XX));
ymax=max(max(YY))+10;

plot(XX,YY) %,[0 0], [-ymax ymax]);
title(['Accuracy Domain of BI' num2str(algor) ' (x0=(0,1), esp=10E-4)']);
%axis([-5 xmax -ymax ymax]);
grid;
xlabel('Re(lambda*h)');
ylabel('Im(lambda*h)');
%gtext(['alpha=0.1,0.2,0.3,0.4']); % num2str(al(1))]);
%gtext(['alpha=0.41,0.42,0.43,0.44,0.45']); % num2str(al(1))]);
%gtext(['alpha=0.46,0.47,0.48,0.49,0.50']); % num2str(al(1))]);
```

```
gtext(['alpha=' num2str(al(1))]);
for i=2:num
    gtext(num2str(al(i)));
  end

eval(savefile);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%     DampFreq.m                                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

R=20;
N=50;
algor=55;

ang=[90 75 45 15 0];
alp=[1 4];

for i=1:length(ang)
an=num2str(ang(i));
theta=ang(i)*pi/180;
for j=1:length(alp)
al=num2str(alp(j));
alpha=alp(j)/10;
[Sd,Wd,S,W,RR]=damp(algor, alpha, theta, R, N);

savefile=['save res55_' an '_0' al ' Sd Wd S W RR'];
eval(savefile);

end
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%     damp.m                                     %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [Sd,Wd,S,W,RR]=damp(algor, alpha, theta, R, N)

% damping and frequency plots for algorithm # "algor" with alpha
% for pole=r(cos(theta) + j sin(theta)), where 0 <= r <= R;
% N: discretizaation of [0, R]
% Note: we require 0 <= theta <= pi/2;

dr=R/N;
cc=cos(theta);
ss=sin(theta);

Sd=[]; Wd=[]; S=[]; W=[]; RR=[];
for i=0:N
r=i*dr;
s=r*cc; w=r*ss;
z=eig_of_F(algor, alpha, s, w);

sd = -log(abs(z));
wd = angle(z);

S=[S, s]; W=[W, w]; RR=[RR, r];
Sd=[Sd, sd]; Wd=[Wd, wd];

end

W=W - 2*pi*floor(W/(2*pi));
% transfer W to [0 2*pi] in order to compare with Wd.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%     eig_of_F.m                                  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function z = eig_of_F(algor,alpha,s,w)
%
%  This function computes the eigvalue of F;
%

A=[0 1;-s^2-w^2 -2*s];

h=1.0;
F = Fmat(A,h,algor,alpha);
zz = eig(F);
z = zz(1,:);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%     plotdamp.m                                  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


ang=[90 75 45 15 0];
alp=[1 4];

for i=1:length(ang)
an=num2str(ang(i));

loadfile1=['load res45_' an '_01'];
loadfile4=['load res45_' an '_04'];
dampname=['Damping Plot for BI45 (theta=' an ')'];
freqname=['Frequency Plot for BI45 (theta=' an ')'];
plotname=['print BI45_' an];
figname=['!lpr -PPS1 BI45_' an '.ps'];

eval(loadfile1);
Sd1=Sd;
Wd1=Wd;
eval(loadfile4);
Sd4=Sd;
Wd4=Wd;
subplot(1,2,1), plot(-RR,S,'-',-RR,Sd1,'--', -RR, Sd4,'-.');
title(dampname);
xlabel('r');
ylabel('damping');

subplot(1,2,2), plot(-RR,W,'-',-RR,Wd1,'--', -RR, Wd4,'-.');
title(freqname);
xlabel('r');
ylabel('frequency');
eval(plotname);
eval(figname);

end

end
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%        ode45.m                                  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [tout, yout] = ode45(FunFcn, t0, tfinal, y0, tol, trace)
%ODE45  Integrate a system of ordinary differential equations using
%       4th and 5th order Runge-Kutta formulas.  See also ODE23 and
%       ODEDEMO.M.
```

```
%       [T,Y] = ODE45('yprime', T0, Tfinal, Y0) integrates the system
%       of ordinary differential equations described by the M-file
%       YPRIME.M over the interval T0 to Tfinal and using initial
%       conditions Y0.
%       [T, Y] = ODE45(F, T0, Tfinal, Y0, TOL, 1) uses tolerance TOL
%       and displays status while the integration proceeds.
%
% INPUT:
% F     - String containing name of user-supplied problem description.
%         Call: yprime = fun(t,y) where F = 'fun'.
%         t      - Time (scalar).
%         y      - Solution column-vector.
%         yprime - Returned derivative column-vector; yprime(i) = dy(i)/dt.
% t0    - Initial value of t.
% tfinal- Final value of t.
% y0    - Initial value column-vector.
% tol   - The desired accuracy. (Default: tol = 1.e-6).
% trace - If nonzero, each step is printed. (Default: trace = 0).
%
% OUTPUT:
% T  - Returned integration time points (row-vector).
% Y  - Returned solution, one solution column-vector per tout-value.
%
% The result can be displayed by: plot(tout, yout).

%   C.B. Moler, 3-25-87.
%   Copyright (c) 1987 by the MathWorks, Inc.
%   All rights reserved.

% The Fehlberg coefficients:
alpha = [1/4  3/8  12/13  1  1/2]';
beta  = [ [    1      0      0     0      0    0]/4
          [    3      9      0     0      0    0]/32
          [ 1932  -7200   7296     0      0    0]/2197
          [ 8341 -32832  29440  -845      0    0]/4104
          [-6080  41040 -28352  9295  -5643    0]/20520 ]';
gamma = [ [902880  0  3953664  3855735  -1371249  277020]/7618050
          [ -2090  0    22528    21970    -15048  -27360]/752400 ]';
pow = 1/5;
if nargin < 6, trace = 0; end
if nargin < 5, tol = 1.e-6; end

% Initialization
t = t0;
hmax = (tfinal - t)/5;
hmin = (tfinal - t)/20000;
h = (tfinal - t)/100;
y = y0(:);
f = y*zeros(1,6);
tout = t;
yout = y.';
tau = tol * max(norm(y, 'inf'), 1);

if trace
   clc, t, h, y
end

% The main loop
   while (t < tfinal) & (h >= hmin)
      if t + h > tfinal, h = tfinal - t; end

      % Compute the slopes
      temp = feval(FunFcn,t,y);
      f(:,1) = temp(:);
      for j = 1:5
         temp = feval(FunFcn, t+alpha(j)*h, y+h*f*beta(:,j));
```

```
      f(:,j+1) = temp(:);
   end

   % Estimate the error and the acceptable error
   delta = norm(h*f*gamma(:,2),'inf');
   tau = tol*max(norm(y,'inf'),1.0);

   % Update the solution only if the error is acceptable
   if delta <= tau
      t = t + h;
      y = y + h*f*gamma(:,1);
      tout = [tout; t];
      yout = [yout; y.'];
   end
   if trace
      home, t, h, y
   end

   % Update the step size
   if delta ~= 0.0
      h = min(hmax, 0.8*h*(tau/delta)^pow);
   end
end;

if (t < tfinal)
   disp('SINGULARITY LIKELY.')
   t
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%           bi45.m                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [tout,xout]=bi45(FunFcn,t0,tf,x0,alpha,tol);
%
% BI45 with stepsize control (3.14) on my thesis
%

% The Fehlberg coefficients:
omega  = [1/4  3/8  12/13  1  1/2]';

beta   = [ [    1      0      0     0      0    0]/4
           [    3      9      0     0      0    0]/32
           [ 1932  -7200   7296     0      0    0]/2197
           [ 8341 -32832  29440  -845      0    0]/4104
           [-6080  41040 -28352  9295  -5643    0]/20520 ]';

gamma  = [ [902880  0  3953664  3855735  -1371249  277020]/7618050
           [ -2090  0    22528    21970    -15048  -27360]/752400 ]';

pow=1/5;
if nargin < 5, alpha = 0.45;  end
if nargin < 6, tol   = 1.e-6; end

% Initialization
x1 = x0(:);
f4 = x1*zeros(1,6);
hmax = (tf-t0)/16;
%hmin = hmax/64;
hmin=0.0;
h=hmax/8;
tout = [];
xout = [];
tout = [tout;t0];
```

```
xout = [xout;x0'];
t1 = t0;

% The main loop
   while (t1 < tf) & (h >= hmin)
      if t1 + h > tf, h = tf - t1; end

      ah0 = h*alpha;
      ah1 = h*(1-alpha);
% Calculate x(k+alpha);
      temp = feval(FunFcn,t1,x1);
      f4(:,1) = temp(:);
      for j = 1:5
          temp = feval(FunFcn, t1+omega(j)*ah0, x1+ah0*f4*beta(:,j));
          f4(:,j+1) = temp(:);
      end
      xa = x1 + ah0*f4*gamma(:,1);
      ta = t1+ah0;

% Calculate e(alpha*h)_left
      err_left = ah0*f4*gamma(:,2);

% Calculate the initial value of x(k+1)
      temp = feval(FunFcn,ta,xa);
      f4(:,1) = temp(:);
      for j = 1:5
          temp = feval(FunFcn, ta+omega(j)*ah1, xa+ah1*f4*beta(:,j));
          f4(:,j+1) = temp(:);
      end
      x20 = xa + ah1*f4*gamma(:,1);
      t2 = ta+ah1;

      x2=BroydenNewton(FunFcn,t2,ah1,xa,x20,tol);

      temp = feval(FunFcn,t2,x2);
      f4(:,1) = temp(:);
      for j = 1:5
          temp = feval(FunFcn, t2-omega(j)*ah1, x2-ah1*f4*beta(:,j));
          f4(:,j+1) = temp(:);
      end

% Calculate e(h-alpha*h)_right;
      err_right = ah1*f4*gamma(:,2);

% Calculate  error
      err =err_left - err_right;
      error  = norm(err,'inf');
      tau    = tol*max(norm(x2,'inf'),1.0);

% Update the solution only if the error is acceptable
      if error <= tau
         x1 = x2;
         t1 = t2;
         tout = [tout;t1];
         xout = [xout;x2'];
      end

% Update the step size
      if error ~= 0.0
         h = min(hmax, 0.8*h*(tau/error)^pow);
      end

end
% end of while loop

      if (t1 < tf)
```

```
         disp('Backward Interpolation Method Fails at')
         t1
      end

end



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%            sbi45.m                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [tout,xout]=sbi45(FunFcn,t0,tf,x0,alpha,tol);

%
% BI45 with stepsize control (3.15) on my thesis
%

% The Fehlberg coefficients:
omega  = [1/4  3/8  12/13  1  1/2]';

beta   = [ [    1      0      0     0      0    0]/4
           [    3      9      0     0      0    0]/32
           [ 1932  -7200   7296     0      0    0]/2197
           [ 8341 -32832  29440  -845      0    0]/4104
           [-6080  41040 -28352  9295  -5643    0]/20520 ]';

gamma  = [ [902880  0  3953664  3855735  -1371249  277020]/7618050
           [ -2090  0    22528    21970    -15048  -27360]/752400 ]';

pow=1/5;
pow1=0.3/5;
pow2=0.4/5;
pow=1/5;
if nargin < 5, alpha = 0.45;  end
if nargin < 6, tol   = 1.e-6; end

% Initialization
x1 = x0(:);
f4 = x1*zeros(1,6);
hmax = (tf-t0)/16;
%hmin = hmax/64;
hmin=0.0;
h=hmax/8;
tout = [];
xout = [];
tout = [tout;t0];
xout = [xout;x0'];
t1 = t0;
first_step=1;
% The main loop
   while (t1 < tf) & (h >= hmin)
      if t1 + h > tf, h = tf - t1; end

      ah0 = h*alpha;
      ah1 = h*(1-alpha);
% Calculate x(k+alpha);
      temp = feval(FunFcn,t1,x1);
      f4(:,1) = temp(:);
      for j = 1:5
          temp = feval(FunFcn, t1+omega(j)*ah0, x1+ah0*f4*beta(:,j));
          f4(:,j+1) = temp(:);
      end
      xa = x1 + ah0*f4*gamma(:,1);
      ta = t1+ah0;
```

```
% Calculate e(alpha*h)_left
      err_left = ah0*f4*gamma(:,2);

% Calculate the initial value of x(k+1)
      temp = feval(FunFcn,ta,xa);
      f4(:,1) = temp(:);
      for j = 1:5
          temp = feval(FunFcn, ta+omega(j)*ah1, xa+ah1*f4*beta(:,j));
          f4(:,j+1) = temp(:);
      end
      x20 = xa + ah1*f4*gamma(:,1);
      t2 = ta+ah1;

      x2=BroydenNewton(FunFcn,t2,ah1,xa,x20,tol);

      temp = feval(FunFcn,t2,x2);
      f4(:,1) = temp(:);
      for j = 1:5
          temp = feval(FunFcn, t2-omega(j)*ah1, x2-ah1*f4*beta(:,j));
          f4(:,j+1) = temp(:);
      end

% Calculate e(h-alpha*h)_right;
      err_right = ah1*f4*gamma(:,2);

% Calculate  error
      err    = err_left - err_right;
      error  = norm(err,'inf');
      tau    = tol*max(norm(x2,'inf'),1.0);

% Update the solution only if the error is acceptable
      if error <= tau
         x1 = x2;
         t1 = t2;
         tout = [tout;t1];
         xout = [xout;x2'];
      end

% Update the step size
      if error ~= 0.0
         if first_step
            h = min(hmax, 0.8*h*(tau/error)^pow);
            first_step=0;
         else
            h = min(hmax, h*(0.8*tau/error)^pow1*(error_last/error)^pow2);
         end
         error_last=error;
      end

end
% end of while loop

      if (t1 < tf)
         disp('Backward Interpolation Method Fails at')
         t1
      end

end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       BroydenNewton.m                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function xroot=BroydenNewton(Fun,t,h,xa,x0,tol,nn,B,InvB);

% Fun: function whose root are to be found;
%   t: time in the BI algorithm
%   h: step size in the BI algorithm
%  xa: target value of x (from forward step in BI algorithm)
%  x0: initial guess of zero;
% tol: accuracy specification;
%  nn: maximum number of iterations.
%   B: B is Broyden' maxtrix and InvB is the inverse of B;

% The Fehlberg coefficients:
omega  = [1/4  3/8  12/13  1  1/2]';

beta   = [ [    1      0      0     0      0    0]/4
           [    3      9      0     0      0    0]/32
           [ 1932  -7200   7296     0      0    0]/2197
           [ 8341 -32832  29440  -845      0    0]/4104
           [-6080  41040 -28352  9295  -5643    0]/20520 ]';

gamma  = [902880  0  3953664  3855735  -1371249  277020]'/7618050;

if nargin < 8, B   = eye(length(x0)); invB = B; end
if nargin < 7, nn  = 10000;           end
if nargin < 6, tol = 1.e-6;           end

N=0;
f = x0*zeros(1,6);
% Calculate f0:
       temp  = feval(Fun, t, x0);
       f(:,1) = temp(:);
       for j = 1:5
         temp = feval(Fun, t-omega(j)*h, x0-h*f*beta(:,j));
         f(:,j+1) = temp(:);
       end
f0 = x0 - h*f*gamma(:,1) - xa;

iteration = 1;

while (iteration & (N <= nn)) ,
N=N + 1;
% Calculate xroot:
xroot  = x0 - invB*f0;

% Calculate f1:
       temp  = feval(Fun, t, xroot);
       f(:,1) = temp(:);
       for j = 1:5
         temp = feval(Fun, t-omega(j)*h, xroot-h*f*beta(:,j));
         f(:,j+1) = temp(:);
       end
f1 = xroot - h*f*gamma(:,1) - xa;

% X-test and F-test
ds = xroot - x0;
xnorm=max(abs(x0));
dnorm=max(abs(ds));
fnorm=max(abs(f1));


iteration = (dnorm > tol*(1 + xnorm)) | (fnorm > tol);

% update everything
if iteration,
```

```
aa =1.0/norm(ds);
ds =ds*aa;
df =(f1 - f0)*aa;

B    = B    + (df -    B*ds)*ds';
%invB = invB + (df - invB*ds)*ds';
invB=inv(B);
end;

f0=f1;
x0=xroot;

% end of if

end;
% end of while loop;

 if (nn < N)
         disp('Broyden-Newton Method Fails at')
         t
      end

end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%      EulerMain55.m                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% this is the main program for the dynamic simulation of one-link flexible arm

global A_matrix b_vector t_duration amp_pulse
% A_matrix and b_vector will be used in DynOpen.m (impact.m, step.m, or ramp.m)

% define system parameters
eta=0.01;
mu=0.01;
kappa=0.01;
zeta=0.001;

% define number of segements
% n=10;

% define cross section
alpha=ones(n+1,1);
beta=ones(n+1,1);
delta=0.01*ones(n+1,1);

% calculate A_matrix and b_vector in the state-variable representation.
[A_matrix, b_vector]=ABmat(n, alpha, beta, delta, eta, mu, kappa, zeta);
freq=eig(A_matrix);

% state vector=[theta, displacement, their time derivatives];
% initial conditions
x0=zeros(2*(n+1),1);

% simulation period [t0, tf];
t0=0;
tf=2;
% pulse duration and amplitude
t_duration=0.5;
amp_pulse=0.1;

alpha=0.5;
tstart=cputime;
[tt,xx]=bi55('impact', t0, tf, x0,alpha);
```

```
texecution=cputime-tstart;
Step=length(tt);

theta=xx(:,1);
%dtheta=xx(:,n+2);
tipdef=xx(:,n+1)-theta;
%dtipdef=xx(:,2*(n+1))-dtheta;

% convert rad to degree
theta=theta*180/pi;
plot(tt, theta);
xlabel('time');
ylabel('theta (deg)');
pause

%plot(tt, dtheta);
%xlabel('time');
%ylabel('dot_theta');
%pause

plot(tt, tipdef);
xlabel('time');
ylabel('tip deflection');
pause

%plot(tt,dtipdef);
%xlabel('time');
%ylabel('dot_tip deflection');

savefile=['save BI1impact' num2str(n)];

eval(savefile);

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%          EulerMain45.m                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% this is the main program for the dynamic simulation of one-link flexible arm

global A_matrix

% define system parameters
eta=0.01;
mu=0.01;
kappa=0.01;
zeta=0.001;

% define number of segements
n=10;

% define cross section
alpha=ones(n+1,1);
beta=ones(n+1,1);
delta=0.01*ones(n+1,1);

% calculate A_matrix and b_vector in the state-variable representation.
[A_matrix, b_vector]=ABmat(n, alpha, beta, delta, eta, mu, kappa, zeta);

% feedback control
% tau=ktp*theta + ktv*dtheta + kwp*tipdef + kwv*dtipdef=kvector*x;
ktp=18.6550;
ktv=5.5285;
kwp=1.50;
kwv=0.005;
```

```
kvector=[ktp-kwp, zeros(1,n-1), kwp, ktv-kwv, zeros(1,n-1), kwv];
A_matrix=A_matrix - b_vector*kvector;
%pole=eig(A_matrix);

% state vector=[theta, displacement, their time derivatives];
% initial conditions

dn=1.0/n;
z0=dn:dn:1;
x0=[1; z0'; zeros(n+1,1)]*pi/2;

% simulation period [t0, tf];
t0=0;
tf=5;

alpha=0.47;
tstart=cputime;
[tt,xx]=bi45('DynClose', t0, tf, x0, alpha);
texecution=cputime-tstart;
Step=length(tt);

theta=xx(:,1);
%dtheta=xx(:,n+2);
tipdef=xx(:,n+1)-theta;
%dtipdef=xx(:,2*(n+1))-dtheta;

% convert rad to degree
theta=theta*180/pi;
plot(tt, theta);
xlabel('time');
ylabel('theta (deg)');
pause

%plot(tt, dtheta);
%xlabel('time');
%ylabel('dot_theta');
%pause

plot(tt, tipdef);
xlabel('time');
ylabel('tip deflection');
pause

%plot(tt,dtipdef);
%xlabel('time');
%ylabel('dot_tip deflection');

savefile=['save B2Close' num2str(n)];

eval(savefile);

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%           ABmat.m                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ABmat.m
% Euler-Bernoulli Model
function [A,B]=ABmat(n, alpha, beta, delta, eta, mu, kappa, zeta)

% this program gives A and B matrices for a flexible beam in state variable
% form:
%   dot(q) = A*q + B**(tau*L/D 0)
```

```
%   where q=[Z;dot(Z)]; Z=[theta; v_1; v_2; ...; v_n], n > 3.

% Note: alpha=[alpha(0); alpha(1); ...; alpha(n)];
%        beta=[ beta(0);  beta(1); ...;  beta(n)];
%       delta=[delta(0); delta(1); ...; delta(n)];

[mm,kk,bb]=MKBmat(n, alpha, beta, delta, eta, mu, kappa, zeta);

imm=inv(mm);
A=[zeros(n+1,n+1),eye(n+1,n+1);-imm*kk,zeros(n+1,n+1)];
B=[zeros(n+1,1);imm*bb];

% end of the subroutine

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%           MKBmat.m                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% MKBmat.m
% Euler-Bernoulli Model
function [mm,kk,bb]=MKBmat(n, alpha, beta, delta, eta, mu, kappa, zeta)

% this program gives mm, kk, and bb matrices for a flexible beam:
%   mm*ddot(Z) + kk*Z = bb*(tau*L/D_0)
%   where Z=[theta; v_1; v_2; ...; v_n], n > 3.
% Note: all difference equations have been divided by n*n.

% Note: alpha=[alpha(0); alpha(1); ...; alpha(n)];
%        beta=[ beta(0);  beta(1); ...;  beta(n)];
%       delta=[delta(0); delta(1); ...; delta(n)];

if n < 3,
   disp('number of segments is too few (n > 3)')
else
% calculate bb input vector;
bb=[1; zeros(n,1)];

n2=n*n;
n4=n2*n2;
in2=1.0/n2;

% calculate mm mass matrix;
% initialization
mm=[];
% i=0;
mm=[mm; eta, zeros(1,n)];
% i=1;
ad0=n2*alpha(1)*delta(1);
ad1=n2*alpha(2)*delta(2);
aa =alpha(2)+ad0+ad1;
mm=[mm; 0, aa, -ad1, zeros(1,n-2)];

% i=2 to n-2;
for i=2:(n-2)
ad0=n2*alpha(i)*delta(i);
ad1=n2*alpha(i+1)*delta(i+1);
aa =alpha(i+1)+ad0+ad1;
mm=[mm; zeros(1,i-1), -ad0, aa, -ad1, zeros(1,n-i-1)];
end

% i=n-1;
ad0=n2*alpha(n-1)*delta(n-1);
ad1=n2*(alpha(n)*delta(n)+n*kappa);
aa =alpha(n)+ad0+ad1;
mm=[mm; zeros(1,n-2), -ad0, aa, -ad1-n2*zeta*mu];
```

```
% i=n;
ad0=n*(alpha(n+1)*delta(n+1)+mu*zeta)+n2*kappa;
aa =ad0+n*mu*zeta+mu;
mm=[mm; zeros(1,n-1), -ad0, aa];


% calculate kk stiffness matrix;
% initialization
kk=[];
% i=0;
kk=[kk; n*beta(1), -n2*beta(1), zeros(1,n-1)];

% i=1;
b0=-n2*n*beta(1);
b1=n4*2*(beta(2)+beta(3));
b2=n4*(beta(1)+4*beta(2)+beta(3));
b3=n4*beta(3);
kk=[kk; b0, b2, -b1, b3, zeros(1,n-3)];

% i=2;
b0=n4*beta(2);
b1=n4*2*(beta(2)+beta(3));
b2=n4*(beta(2)+4*beta(3)+beta(4));
b3=n4*beta(4);
b4=n4*2*(beta(3)+beta(4));
kk=[kk; 0, -b1, b2, -b4, b3, zeros(1,n-4)];

% i=3 to n-2;
for i=3:(n-2)
b0=n4*beta(i);
b1=n4*2*(beta(i)+beta(i+1));
b2=n4*(beta(i)+4*beta(i+1)+beta(i+2));
b3=n4*beta(i+2);
b4=n4*2*(beta(i+1)+beta(i+2));
kk=[kk; zeros(1,i-2), b0, -b1, b2, -b1, b3, zeros(1,n-i-2)];
end

% i=n-1;
b0=n4*beta(n-1);
b1=n4*2*(beta(n-1)+beta(n));
b2=n4*(beta(n-1)+4*beta(n));
b3=n4*2*beta(n);
kk=[kk; zeros(1,n-3), b0, -b1, b2, -b3];

% i=n;
b0=n2*n*beta(n);
kk=[kk; zeros(1,n-2), b0, 2*b0, b0];

mm=mm*in2;
kk=kk*in2;
bb=bb*in2;
end
% end of if

end
% end of the subroutine


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%          impact.m                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function xdot=impact(t, x)
global A_matrix b_vector t_duration amp_pulse
```

```
tau=amp_pulse*(t < t_duration);

xdot=A_matrix*x + b_vector*tau;

end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%          step.m                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function xdot=step(t, x)
global A_matrix b_vector t_duration amp_pulse

tau=amp_pulse;

xdot=A_matrix*x + b_vector*tau;

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%          ramp.m                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function xdot=ramp(t, x)
global A_matrix b_vector t_duration amp_pulse

tau=amp_pulse*t;

xdot=A_matrix*x + b_vector*tau;

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%          DynClose.m                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function xdot=DynClose(t, x)
global A_matrix

xdot=A_matrix*x;

end
```

APPENDIX B

C Program

```
#include <stdio.h>
#include <math.h>
#define R sizeof(double)
#define L sizeof(tx_list)
#define MAXTX 2

typedef struct TTXX {double t; double *x; struct TTXX *p;} tx_list;

tx_list *tx;
double omega[5], beta[6][5], gama[6][2], *newv();
double *xa, *x20, *px1, *err_1, *err;
double *ds, *df, *f0, *f1, *f, *b, *invb, *temp;
int   mmn, n;

void main()
{ tx_list *sbi45(), *pp;
  double t0, tf, tol, alpha, nnorm();
  double x0[4], aalpha[17], ttol[17];      /* !!! x0(n) !!! */
  double eerr1[7][17], eerr2[7][17];
  double *p1, *p2, *err1, *err2;
  int i, j, nxout, step[7][17], data();


  data();
  t0 =  0.0;
  tf = 10.0;
  x0[0]= 0.0;  x0[1]= 1.0; x0[2]= 2.0; x0[3]= 0.0;
  for (i=0;i<17;i++) aalpha[i] = 0.1 + 0.025*i;
  tol=0.1;
  for (i=0;i<7;i++) {
    tol=tol/10.0;
    printf("\ntol=%e\n", tol);
    for (j=0;j<17;j++) {
      nxout=0;
      alpha = aalpha[j];
      printf("\n   alpha=%f", alpha);
      tx = sbi45(t0,tf,x0,alpha,tol);
      if (tx == NULL) {printf("\nprogram fails in sbi45..."); exit(1);}
      pp=tx;
      while(pp != NULL) {nxout++; pp = pp->p;}
      pp = tx;
      err1 = (double *)newv(nxout*R);
      err2 = (double *)newv(nxout*R);
      p1=err1; p2=err2;
      while(pp != NULL) {
        /*printf("\n       t(i)=%f", pp->t);*/
        *p1 = pp->x[0] - 2.0*sin(2*(pp->t));
        *p2 = pp->x[1] - cos(pp->t);
        pp=pp->p;p1++;p2++;}
      if((i == 4)&(j == 16)){
        printf("\nSome thing about X1 X2 E1 E2...");}
      printf("\n     Step (or SS)=%d,", nxout);
      printf("\n     ErrMax1=%e,", eerr1[i][j]=nnorm(nxout,err1));
      printf(" ErrMax2=%e.\n", eerr2[i][j]=nnorm(nxout,err2));
      step[i][j]=nxout;
      /*free(err1); free(err2);*/}
    ttol[i]=tol;
    printf("\n");
  }
}

int data()
```

```
{ int   i, j;
  long  bi[6][5];
  long  bd[5];
  long  gi[6][2];
  long  gd[2];

  omega[0]=0.25; omega[1]=0.375; omega[2]=12./13.; omega[3]=1.; omega[4]=0.5;
  bi[0][0]=1; bi[0][1]=3; bi[0][2]= 1932;  bi[0][3]= 8341;   bi[0][4]= -6080;
  bi[1][0]=0; bi[1][1]=9; bi[1][2]= -7200; bi[1][3]= -32832; bi[1][4]= 41040;
  bi[2][0]=0; bi[2][1]=0; bi[2][2]= 7296;  bi[2][3]= 29440;  bi[2][4]= -28352;
  bi[3][0]=0; bi[3][1]=0; bi[3][2]= 0;     bi[3][3]= -845;   bi[3][4]= 9295;
  bi[4][0]=0; bi[4][1]=0; bi[4][2]= 0;     bi[4][3]= 0;      bi[4][4]= -5643;
  bi[5][0]=0; bi[5][1]=0; bi[5][2]= 0;     bi[5][3]= 0;      bi[5][4]= 0;
  bd[0]=4;    bd[1]=32;   bd[2]=2197;      bd[3]=4104;       bd[4]=20520;
  gi[0][0]=  902880;      gi[0][1]= -2090;
  gi[1][0]=  0;           gi[1][1]= 0;
  gi[2][0]=  3953664;     gi[2][1]=  22528;
  gi[3][0]=  3855735;     gi[3][1]=  21970;
  gi[4][0]= -1371249;     gi[4][1]= -15048;
  gi[5][0]=  277020;      gi[5][1]= -27360;
  gd[0]   =  7618050;     gd[1]   = 752400;
  for(i=0; i<6; i++)
  { gama[i][0] = (double)gi[i][0]/(double)gd[0];
    gama[i][1] = (double)gi[i][1]/(double)gd[1];
    for(j=0; j<5; j++) beta[i][j] = (double)bi[i][j]/(double)bd[j];}
  n=4;
  mmn=MAXTX;
  px1   = (double *)newv(R*n*mmn);
  xa    = (double *)newv(R*n);
  x20   = (double *)newv(R*n);
  f     = (double *)newv(R*n*6);
  err_1 = (double *)newv(R*n);
  err   = (double *)newv(R*n);
  temp  = (double *)newv(R*n*n);
  f0    = (double *)newv(R*n);
  f1    = (double *)newv(R*n);
  ds    = (double *)newv(R*n);
  df    = (double *)newv(R*n);
  b     = (double *)newv(R*n*n);
  invb  = (double *)newv(R*n*n);
  tx    = (tx_list *)newv(L*mmn);
  return(0);
}

tx_list *sbi45(t0,tf,x0,alpha,tol)
  double x0[], alpha, tol, t0, tf;
{ tx_list *tp, *ptx, *append_tx();
  int     next_x(), pass_f(), broyden();
  double   *x1, *x2, error_last, norm();
  int     i, k, nn, first;
  double   pow,pow1,pow2,t1,ta,t2,ah0,ah1,h,hmax,hmin,xnorm,error,tau,c;

  ptx = tx;
  x1  = px1;
  x2  = x1 + n;
  pow = 0.2;
  pow1= 0.06;
  pow2= 0.08;
  nn  = 10000;
  for(k=0; k<n; k++)  x1[k] = x0[k];
  for(k=0;k<6*n;k++)  f[k] = 0.0;
  t1 = t0;
  hmax = (tf - t0)/16.0;
  hmin = 0.0;
  h = hmax/8.0;
  tx->t = t1;
```

```
  tx->x = x1;
  tx->p = NULL;
  first = 1;

  while((t1<tf) & (h>=hmin)) {
    /*printf("\n       t(i)=%f",t1);*/
    if(t1+h>tf) {h=tf-t1;return(tx);}
    if(mmn < 2) {
      x2  = px1 = (double *)newv(R*n*MAXTX);
      tp  = (tx_list *)newv(L*MAXTX);}
    ah0 = h*alpha;
    ah1 = h - ah0;
    pass_f(t1, x1, ah0, f);
    next_x(xa, x1, ah0, f, gama, 0, 6, 2);
    ta = t1 + ah0;
    next_x(err_1, NULL, ah0, f, gama, 1, 6, 2);
    pass_f(ta, xa, ah1, f);
    next_x(x20, xa, ah1, f, gama, 0, 6, 2);
    t2 = ta + ah1;
    if (broyden(t2, ah1, xa, x20, tol, nn, /*b,invb, */x2) )
      {printf("\nprogram fails in broydennewton..."); return(NULL);}
    pass_f(t2, x2, -ah1, f);
    next_x(err, err_1, -ah1, f, gama, 1, 6, 2);
    error = norm(err);
    xnorm = norm(x2);
    tau   = tol*((xnorm>1.0)?xnorm:1.0);

    if(error<=tau) {
      t1  = t2;
      if(--mmn < 1) {
        mmn = MAXTX;
        x1  = px1;
        tp->t = t1; tp->x = x1; tp->p = NULL;
        ptx = ptx->p = tp;}
      else {
        x1  = x2;
        ptx = append_tx(ptx, t1, x1);}
      x2 += n;}
    if(error != 0.0) {
      if(first) {
        c = pow*log(tau/error);
        c = 0.8*h*exp(c);
        h = (c<hmax)?c:hmax;
        first = 0;}
      else {
        c = pow1*log(0.8*tau/error) + pow2*log(error_last/error);
        c = h*exp(c);
        h = (c<hmax)?c:hmax;}
      error_last = error;
    }
  }
  if(t1 < tf) {
    printf("\nBackward Interpolation Method Fails at %f:\n",t1);
    return(NULL);}
  return(tx);
}

int broyden(t, h, xa, x0, tol, nn, /**b,invb, */xroot)
  int nn;
  double t, h, *xa, *x0, tol, /**b,*invb, */*xroot;
{ int   inv(), pass_f(), next_x();
  double *newv(), norm(), norm2();
  int   mn, i, j, k, iteration;
  double xnorm, dnorm, fnorm, aa;

  for (i=0;i<n*n;i++) {b[i]=0; invb[i]=0;}
```

```
  for (i=0;i<n;i++) {b[(n+1)*i]=1; invb[(n+1)*i]=1;}
  nn = 1000;
  /*tol=1.0e-6;*/
  mn = 0;

  for(k=0;k<6*n;k++)  f[k] = 0.0;
  pass_f(t, x0, -h, f);
  for(i=0;i<n;i++) f0[i] = x0[i] - xa[i];
  next_x(f0, f0, -h, f, gama, 0, 6, 2);
  iteration = 1;

  while(iteration & (mn++ <= nn)) {
    next_x(xroot, x0, -1.0, invb, f0, 0, n, 1);
    for(i=0;i<n;i++) {
      f1[i] = xroot[i] - xa[i];
      ds[i] = xroot[i] - x0[i]; }
    pass_f(t, xroot, -h, f);
    next_x(f1, f1, -h, f, gama, 0, 6, 2);
    xnorm = norm(x0);
    dnorm = norm(ds);
    fnorm = norm(f1);
    iteration = (dnorm > tol*(1 + xnorm)) | (fnorm > tol);
    dnorm = norm2(ds);
    if (iteration) {
      if(dnorm == 0) {printf("\ndigital error..."); return(1);}
      aa = 1.0/dnorm;
      for(i=0;i<n;i++) {
        ds[i] *= aa;
        df[i] = (f1[i] - f0[i])*aa;}
      for(i=0;i<n;i++)
        for(k=0;k<n;k++) df[i] -= b[i*n+k]*ds[k];
      for(i=0;i<n;i++)
        for(j=0;j<n;j++)
          b[i*n+j] += df[i]*ds[j];
      if(inv(b, invb)) {printf("\nfails to find inv b"); return(1);}
      printf(".");}
    for(i=0;i<n;i++) {
      f0[i] = f1[i];
      x0[i] = xroot[i];}
  }
  if(nn < mn) {
    printf("\nBroydenNewton Method Fails at %f:\n",t);
    /*return(1)*/;}
  return(0);
}

int next_x(xt, x, h, f, bg, y, mx, my)
  int y, mx, my;
  double *xt, *x, h, *f, *bg;
{ double xi;
  int i, k;

  for (i=0;i<n;i++) {
    xi = 0.0;
    for(k=0;k<mx;k++) xi = xi + f[mx*i+k]*bg[my*k+y];
    if(x) xt[i] = x[i] + h*xi; else xt[i] = h*xi;}
  return(0);
}

int pass_f(t, x, ah, f)
  double t, *x, ah, *f;
{ int i, j, next_x(), fun();
  double xx, *newv();

  fun(t, x, temp);
  for (i=0;i<n;i++) f[6*i] = temp[i];
```

```
  for(j=0; j<5; j++) {
    next_x(temp+n, x, ah, f, beta, j, 6, 5);
    xx = t + omega[j]*ah;
    fun(xx, temp+n, temp);
    for (i=0;i<n;i++) f[6*i+j+1] = temp[i];}
  return(0);
}

tx_list *append_tx(tl, t, x)
  tx_list *tl;
  double  t, *x;
{
  while(tl->p != NULL) tl = tl->p;
  tl = tl->p = tl + 1;
  tl->t = t; tl->x = x; tl->p = NULL;
  return (tl);
}

int inv(a,b)
  double *a, *b;
{ int i,k,addij(),onei(),zeroi();

  for (i=0;i<n*n;i++) {temp[i] = a[i]; b[i]=0;}
  for (i=0;i<n;i++)   b[(n+1)*i] = 1;
  for (i=0;i<n;i++)   {
    k=i;
    while(temp[n*k+i] == 0) {
      k++;
      if(k >= n) {printf("\nB matrix singular.\n"); return(1);}}
    if(k > i) addij(temp, b, i, k, 1.0);
    if(temp[n*i+i]!=1) onei(temp, b, i);
    zeroi(temp, b, i);}
  return(0);
}

int addij(b1, b2, i, j, d)
  double *b1, *b2, d;
  int   i,j;
{ int   k,in,jn;

  in=i*n; jn=j*n;
  for (k=0;k<n;k++) {
    b1[in]   += d*b1[jn];
    b2[in++] += d*b2[jn++];}
  return(0);
}

int onei(b1, b2, i)
  double *b1, *b2;
  int i;
{ int   k, in;
  double e;

  e = b1[i*(n+1)];
  if(e == 0) return(1);
  in = i*n;
  for (k=0;k<n;k++) {
    b1[in] /= e;
    b2[in++] /= e;}
  return(0);
}

int zeroi(b1, b2, i)
  double *b1, *b2;
  int   i;
{ int   k, in, addij();
```

```
  in=i*n;
  if(b1[in+i] == 0) return(1);
  if(b1[in+i]!=1) onei(b1,b2,i);
  for(k=0;k<i;k++)   addij(b1,b2,k,i,-b1[k*n+i]);
  for(k=i+1;k<n;k++) addij(b1,b2,k,i,-b1[k*n+i]);
  return(0);
}

double norm2(x)
  double *x;
{ int i;
  double n2x;

  n2x=0.0;
  for(i=0;i<n;i++)  n2x += x[i]*x[i];
  n2x = sqrt(n2x);
  return(n2x);
}

double nnorm(mn,x)
  int  mn;
  double *x;
{ int i;
  double mx;

  if(mn<0) return(1);
  mx=fabs(x[0]);
  for(i=0;i<mn;i++) {
    if(fabs(x[i])>mx) mx=fabs(x[i]); }
  return(mx);
}

double norm(x)
  double *x;
{ double nnorm();
  return(nnorm(n,x));
}

double *newv(d)
  int d;
{ double *p;

  if ((p = (double *)malloc(d)) == NULL) {
    printf("Not enough memory to allocate buffer\n");
    exit(1);  /* terminate program if out of memory */ }
  return(p);
}

int fun(t,x,fx)
  double t, *x, *fx;
{ int fun1();
  return(fun1(t,x,fx));
}

int fun1(t,x,fx)
  double t,*x,*fx;
{ double u;

  u=1.0;
  fx[0] =  4.0*x[2] -4.0*u;
  fx[1] = -4.0*x[3];
  fx[2] = -1.0*x[0];
  fx[3] = 0.25*x[1];
  return(0);
}
```