

**DBDF: AN IMPLICIT NUMERICAL DIFFERENTIATION ALGORITHM
FOR INTEGRATED CIRCUIT SIMULATION**

by

Luoan Hu

A Thesis Submitted to the Faculty of the
COMMITTEE ON ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

**MASTER OF SCIENCE
WITH A MAJOR IN ELECTRICAL ENGINEERING**

In the Graduate College

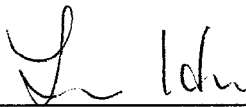
THE UNIVERSITY OF ARIZONA

1991

STATEMENT BY AUTHOR

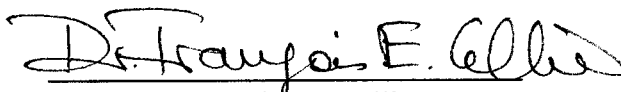
This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

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: 

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:



Francois E. Cellier
Professor of Electrical
and Computer Engineering

July 4, 1991
Date

ACKNOWLEDGEMENTS

I wish to express my deepest appreciation to professor Francois E. Cellier, my thesis advisor, for his valuable advice toward this research and his time and patience in reviewing this thesis. Special thanks also go to professor Olgierd C. Palusinski and Brenton LeMesurier for their constructive criticism of the thesis and presentation. I also wish to express my appreciation to Burr-Brown Corp. for providing financial support for this project. Finally, I would like to acknowledge my wife, Xiangyang Shen, for her tireless assistance and significant contributions in many ways to this research project.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS	6
LIST OF TABLES	7
ABSTRACT	8
1 INTRODUCTION AND DEFINITIONS	9
1.1 Description of ODES and Stiff Systems	9
1.2 Absolute Stability Region of Some Numerical Methods	10
1.3 $A(\alpha)$ -stability, Stiff Stability and Gear's Method	16
2 IMPLICIT INTEGRATION VS DIFFERENTIATION	18
2.1 SPICE: An IC Simulation Program	18
2.2 Transient Analysis: Implicit Integration vs Implicit Differentiation	20
2.3 Does SPICE Really Employ Gear's Methods?	22
2.4 Motivation for Developing the DBDF Method	24
3 DEVELOPMENT OF DBDF METHOD	26
3.1 A Multistep Numerical Scheme	26
3.2 DBDF Method for Solving Implicit I.V.P.	29
3.3 General Formulation for Solving Inverse Systems	35
4 ERROR ANALYSIS OF DBDF METHOD	40
4.1 LTE Analysis of DBDF	40
4.2 A Practical Algorithm for LTE Estimation	44
5 IMPLEMENTATION OF DBDF METHOD IN CTRL-C	46
5.1 Control of Step size and Order	47

5.2 Startup and Convergence of Newton-Raphson Iteration	49
5.3 Construction of DBDF Program	51
5.4 Numerical Test	57
CONCLUSION	71
APPENDIX SOURCE CODE (IN CTRLC)	72
REFERENCES	86

LIST OF ILLUSTRATIONS

Figure	page
1.1 Stability Regions	12
1.2 Stability Regions (continued)	15
2.1 Differential Pair Circuit	18
2.2 p-n Junction	20
5.1 Structure of DBDF Program	52
5.2 Flow Chart of MAIN	53
5.3 Flow Chart of DBDF	54
5.4 Flow Chart of SDBDF	55
5.5 Flow Chart of NR	56
5.6 Case 4 (R-C Circuit)	60
5.7.1 Numerical Solution(x_j) and Global Error(E_g) for Case 1	63
5.7.2 Global Error(E_g) for Case 1	64
5.8.1 Numerical Solution(x_j) and Global Error(E_g) for Case 2	65
5.8.2 Global Error(E_g) for Case 2	66
5.9.1 Numerical Solution(x_j) and Global Error(E_g) for Case 3	67
5.9.2 Global Error(E_g) for Case 3	68
5.10.1 Numerical Solutions(Q_c , I_c and V_c) for a R-C Circuit (Case 4)	69
5.10.2 Global Error(E_g) for a R-C Circuit (Case 4)	70

LIST OF TABLES

Table	page
3.1	27
5.1	58
5.2	59
5.3	60
5.4	61

ABSTRACT

Frequently, the design of integrated circuits cannot be accomplished by purely analytical techniques. Accurate and efficient algorithms for numerical circuit simulation are important tools. Several circuit simulators, such as SPICE, have been made available for this task.

Contrary to many other applications of numerical system simulation, integrated circuit problems don't lend themselves to a formulation of state-space models, since the space charge in a p-n junction is a nonlinear and noninvertible function of the voltage across the junction. Therefore, it is necessary to employ numerical differentiation instead of numerical integration in this type of simulation study.

The numerical algorithms employed in today's circuit simulators are fairly primitive. SPICE, for example, offers only two very simple implementations of the trapezoidal rule and of the backwards differentiation formula.

This thesis describes the design and implementation of DBDF, a specification of a numerical method in Nordsieck format for solving circuit simulation problems. A formal stability and truncation error analysis are included.

CHAPTER 1

INTRODUCTION AND DEFINITIONS

1.1 Description of ODEs and Stiff Systems

In continuous system simulation, we are usually faced with a system of m simultaneous first-order ordinary differential equations (ODEs) represented as:

$$\dot{X} = F(X,t), \quad X(0) = X_0. \quad (1.1)$$

with $t \in [0, T]$, $X \in \mathbb{R}^m$. This is called the initial value problem for first-order systems. We want to find the solution $X(t)$ at time $t \in [0, T]$. The system can be linear ($\dot{X} = A(t)X + B(t)$), but in general it is nonlinear. A subclass of the initial value problems, known as stiff systems, is of interest since it arises in many application areas such as electronic circuit analysis, most chemical kinetics problems and the method-of-lines solution to parabolic PDE problems. Stiff differential equations contain time constants with greatly different values. The *time constants* refer to the rate of decay. In other words, for stiff systems, the eigenvalues of their Jacobian matrices exhibit a widespread range in the complex root plane. Let us consider the following linear time invariant initial value problem:

$$\begin{aligned} \dot{X} &= AX, \\ X(0) &= [1, 0, -1]^T \end{aligned}$$

Where

$$X = [u(t), v(t), w(t)],$$

$$A = \begin{bmatrix} -21 & 19 & -20 \\ 19 & -21 & 20 \\ 40 & -40 & -40 \end{bmatrix}.$$

In this case, the Jacobian $\partial F/\partial X$ is the constant matrix A whose eigenvalues λ_i can be found to be $-2, -40 \pm 40j$. The theoretical solution is given as:

$$u(t) = \frac{1}{2}e^{-2t} + \frac{1}{2}e^{-40t}(\cos 40t + \sin 40t),$$

$$v(t) = \frac{1}{2}e^{-2t} - \frac{1}{2}e^{-40t}(\cos 40t + \sin 40t),$$

$$w(t) = -e^{-40t}(\cos 40t - \sin 40t).$$

There exist, however, several different definitions of stiff systems. One of them requires:

(i) $\text{Re}(\lambda_i) < 0$, and (ii) $\max |\text{Re}(\lambda_i)| \gg \min |\text{Re}(\lambda_i)|$, where $\lambda_i, i = 1, 2, \dots, m$, are the eigenvalues of the Jacobian $\partial F/\partial X$. Also $\max |\text{Re}(\lambda_i)| : \min |\text{Re}(\lambda_i)|$ is called *stiffness ratio* of the system. The example given above thus has a stiffness ratio of 20. Stiffness ratios of the order of 10^6 arise frequently in practical problems. Attempts to apply some of the most commonly used numerical integration methods, such as Runge-Kutta or Adams techniques, to solve such problems encounter substantial difficulties because of the stability properties of those methods. This statement will be substantiated in due course.

1.2 Absolute Stability Region of Some Numerical Methods

Since the problem of stability is only related to a method, [Gear, 1971a] let us consider the single initial value ODE:

$$\dot{x} = \lambda x, \quad x(0) = 1. \quad (1.2)$$

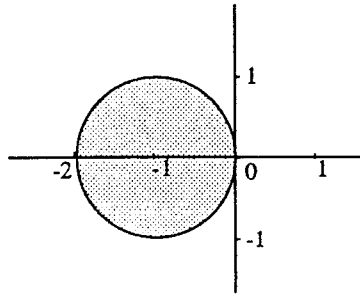
All comments apply equally to the system:

$$\dot{X} = AX, \quad X(0) = X_0.$$

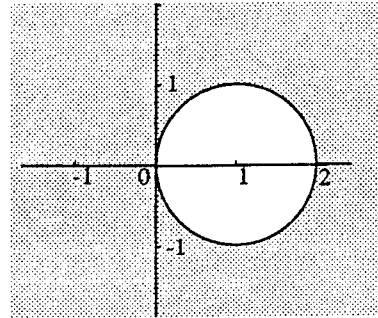
where A is a constant, diagonalizable matrix.[Dahlquist,1974] The region of absolute stability for a numerical method with fixed stepsize h can be defined as the set of values of h (real, nonnegative) and λ_i for which a perturbation in a single value x_n will produce a change in subsequent values which does not increase from step to step.[Gear, 1971a] First, let us look at the explicit Euler rule applied to the test ODE:

$$\begin{aligned} x_{n+1} &= x_n + h\dot{x}_n \\ &= x_n + \lambda h x_n \\ &= (1 + \lambda h)x_n . \end{aligned} \quad (1.3)$$

A stable solution requires $|1 + \lambda h| < 1$ which means that λh must lie in a unit circle around -1 in the complex λh plane (Fig. 1.1-a). In other words, the step size h must be chosen to keep the λh value within the absolutely stable region.

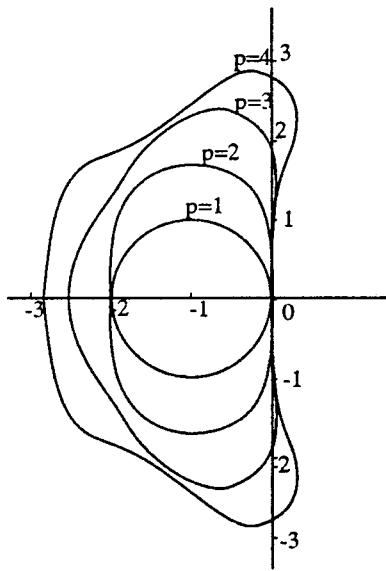


a) Euler (explicit)

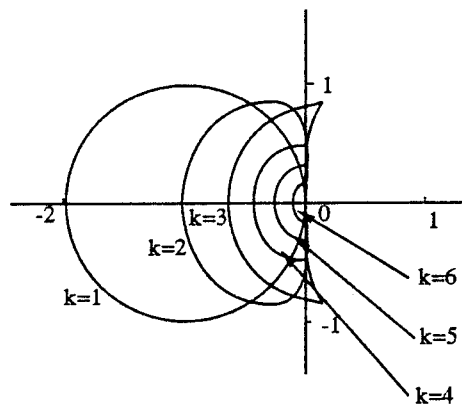


b) Euler (implicit)

λh



c) Runge-Kutta



d) Adams Bashforth

Fig. 1.1 Stability Regions

Then let us look at the implicit Euler rule:

$$\begin{aligned}
 x_{n+1} &= x_n + h\dot{x}_{n+1} \\
 &= x_n + \lambda h x_{n+1} \\
 &= (1 - \lambda h)^{-1} x_n .
 \end{aligned} \tag{1.4}$$

Here this method has a absolutely stable region outside the unit circle around 1. (Fig. 1.1-b)

For the purpose of further analysis and discussion, the stability regions of some most commonly used numerical algorithms are given as follows:

For a p-stage Runge-Kutta method (p = 1,2,3,4 and accordingly order 1 to 4), the algorithm is stable as long as the λh values lie inside the closed contours. (Fig. 1.1-c)

Now let us turn to the general linear multistep method which may be written as

$$\sum_{j=0}^k \alpha_j x_{n-j+1} = h \sum_{j=0}^k \beta_j f_{n-j+1} \tag{1.5}$$

One of the most commonly used multistep method "families" are the Adams methods which have the coefficients $\alpha_0 = 1$, $\alpha_1 = -1$ and $\alpha_2 = \alpha_3 = \dots = \alpha_k = 0$. Fig. 1.1-d gives the stability domains (inside the closed regions) of the k-step, kth order Adams-Bashforth formulae(ABF) (as in (1.6) with $\beta_0 = 0$, thus they are explicit methods).

$$x_{n+1} = x_n + h \sum_{j=1}^k \beta_j f_{n-j+1} \tag{1.6}$$

Fig. 1.2-a shows the stability domains of the k -step, $(k+1)$ th order [Lambert,1973] Adams-Moulton formulae(AMF) (as in (1.7) with $\beta_0 \neq 0$, thus they are implicit algorithms).

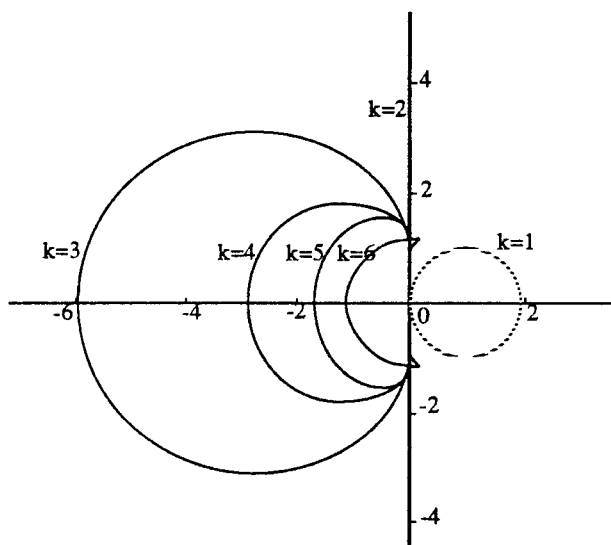
$$x_{n+1} = x_n + h \sum_{j=0}^{k-1} \beta_j f_{n-j+1} \quad (1.7)$$

Note that for $k = 3, 4, 5, 6$, these methods are stable inside the regions indicated; For $k = 2$, known as the Trapezoidal rule, the method is stable throughout the left-hand complex half-plane. (called A-stability[Dahlquist, 1963, Lambert,1973]) For $k=1$, the method is stable outside the unit circle around 1.

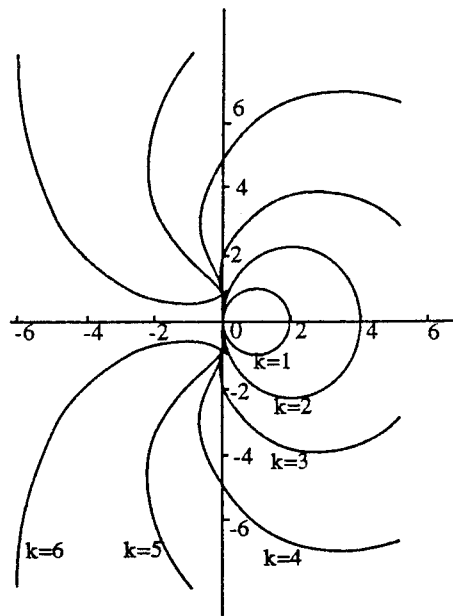
Another set of widely used multistep numerical integration methods are the Backward Differentiation formulae(BDF) which have the coefficients $\alpha_j \neq 0, j = 0, 1, 2, \dots, k, \beta_0 \neq 0$ and $\beta_j = 0$ for $j = 1, 2, 3, \dots, k$, thus we know they are implicit algorithms.(1.8)

$$x_{n+1} = \sum_{j=1}^k \alpha_j x_{n-j+1} + h \beta_0 f_{n+1} \quad (1.8)$$

The stability regions of BDF for $k = 1, 2, 3 \dots, 6$ [Gear,1969,1971a] are sketched in Fig. 1.2-b. The BDF methods are stable outside of the closed contours. Also it is noted that for $k \leq 2$, the BDF methods are A-stable; for $k = 3, 4, 5, 6$, they are known as stiffly stable[section 1.3] and for $k > 7$, the methods are unstable. [Gear,1971a]

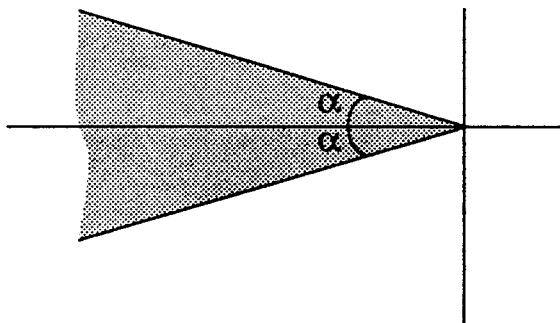


a) Adams Moulton

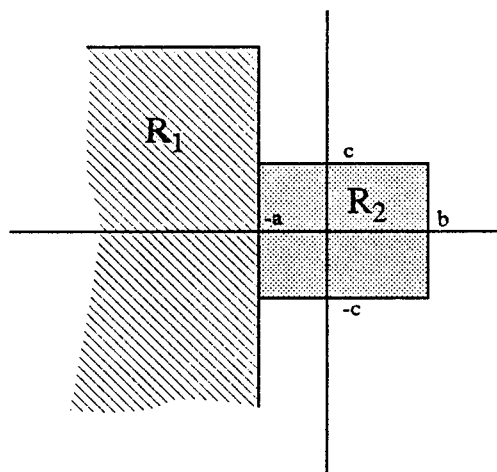


b) BDF

λh



c) $A(\alpha)$ Stability



d) Stiff Stability

Fig. 1.2 Stability Regions

1.3 $A(\alpha)$ -stability, stiff stability and Gear's method

We have seen in 1.1 and 1.2 that a stiff system requires very small step sizes to be taken to keep all λh within the stable domain, except when an A-stable method is applied. It is noted that the difficulties due to the stiffness are also dependent on the simulation time interval $[0, T]$. As long as the T is sufficiently small, we will not see any problem at all in selecting the stepsize for a numerical solution. Let us introduce an "additional eigenvalue", $\lambda_{m+1} = -1/T$, to the set of eigenvalues in section 1.1 if $T > 10/|\lambda_{\min}|$. With this addition, the applicable scope of definition for stiffness in section 1.1 will be extended to cover also first order systems and systems with a mismatch between eigenvalues and simulation run length. From 1.2 it is noted that all higher order methods are not A-stable. (implicit Euler and Trapezoidal rule are first and second order respectively) Indeed, Dahlquist [1963] proved that a multistep method that is A-stable cannot have an order greater than two and that the method of order two with the smallest error constant (0.5) is the Trapezoidal rule. [Gear, 1971a] Since the restriction on order for an A-stable method to solve stiff systems is a severe one, two less demanding stability definitions have been proposed: i) $A(\alpha)$ -stable [Lambert, 1973] if it is absolute stable for some (sufficiently small) $\alpha \in (0, \pi/2)$, (Fig. 1.2-c) [Lambert, 1973] and ii) Stiffly stable [Gear, 1967, Lambert, 1973] if in the region \mathbf{R}_1 ($\text{Re}(\lambda h) \leq -a$) it is absolutely stable, and in \mathbf{R}_2 ($-a < \text{Re}(\lambda h) < b$), ($|\text{Im}(\lambda h)| < c$) it is accurate. (Fig. 1.2-d)

Let us now consider the methods which are not necessarily A-stable but are $A(\alpha)$ or stiffly stable. It can be seen that among the methods considered in section 1.2, the class

which has these stability properties is BDF. In fact its use for stiff systems goes back to Curtis and Hirschfelder in 1952 [Lambert, 1973]. The problem of stiffness has been known for some time and has attracted the attention of many numerical analysts. One of the most important contributions in this area was made by Gear in 1971 [Lambert, 1973]. He derived the stability domain for the k -step BDF methods with k up to 6 and gave the definition of stiff stability. He then developed sophisticated numerical integration software by combining two options (Adams and BDF methods, both were implemented as predictor corrector pairs). [Lambert, 1973] For the automatic solution of initial value problems, it is highly desirable to adjust the integration step size and order in order to minimize the computation and at the same time, achieve a given error bound. Gear's method employed a technique originally due to Nordsieck[Lambert,1973], which is in fact a kind of "transformation" from a multistep method to a "single" step format. The idea is that, instead of storing a number of back values of x_i and f_i (for a multistep method), we can store up to the k th derivative to construct the so-called Nordsieck vector at a single point of the local polynomial interpolation which represents the solution. [Cellier, 1988, Lambert, 1973] For this single step vector format, the step size and order control, as well as the startup procedure can be more conveniently derived.

Until now, we talked about basic numerical concept for solving initial value problems. In the next chapter it is shown how these concepts are related to electronic circuit simulation techniques, one of the most important applications in electrical engineering.

CHAPTER 2

IMPLICIT INTEGRATION VS DIFFERENTIATION:

Problems with Integrated Circuit Simulation

2.1 SPICE: An IC Simulation Program

An electronic circuit simulation program that characterizes the performance of a circuit is one important computer aid in the circuit design process. The need for accurate and efficient circuit simulation has prompted the development of many circuit programs as well as the advancement of the associated numerical methods. Here let us consider SPICE, one of the most commonly used simulation programs by electronics industry.

Like most available network simulation software, SPICE, instead of using state-space representations, employs the topological description for network modelling due to the frequent algebraic loops inherent in most practical electrical circuits [Cellier, 1991] and for its simplicity and efficiency. As an example, [Nagel, 1975] let us look at the following simple electronic network. (Fig. 2.1)

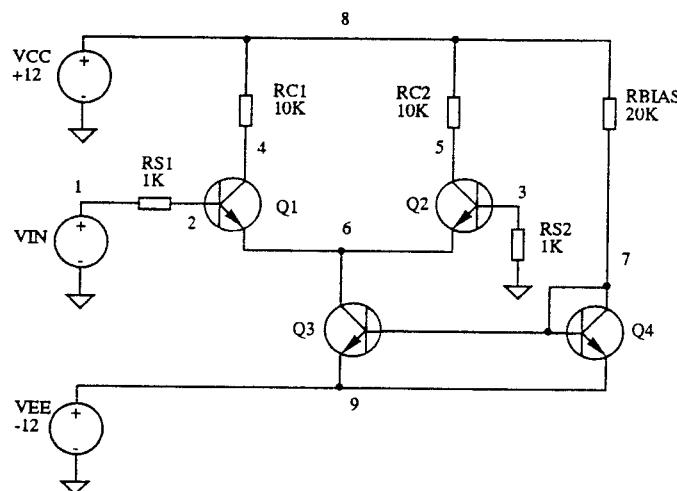


Fig. 2.1 Differential Pair Circuit

It can be modeled in SPICE as follows:

DIFFPAIR CKT - SIMPLE DIFFERENTIAL PAIR

```

* AC DEC 10 1 10GHZ
* DC VIN -0.25 0.25 0.005
* TRAN 5NS 500NS
  .TRAN 5NS 500NS
VIN 1 0 SIN(0 0.1 5MEG 5NS) AC 1
VCC 8 0 12
VEE 9 0 -12
Q1 4 2 6 QNL
Q2 5 3 6 QNL
RS1 1 2 1K
RS2 3 0 1K
RC1 4 8 10K
RC2 5 8 10K
Q3 6 7 9 QNL
Q4 7 7 9 QNL
RBIAS 7 8 20K
  .OUTPUT V4 4 0 PRINT DC TRAN
  .OUTPUT V5 5 0 PRINT MEG PHASE DC TRAN PLOT MEG
+ PHASE DC TRAN
  .MODEL QNL NPN (BF=80 RB=100 CCS=2PF TF=0.3NS TR=6NS
+ CJE=3PF CJC=2PF VA=50)
  .END

```

To accomplish the transition from the physical circuit to a mathematical system of equations, each element in the circuit is represented by a mathematical model implemented inside SPICE. The system of equations is determined by the model equations for each element and topological constraints. There are three basic analyses available in SPICE: dc analysis, time domain transient analysis and small signal ac analysis. Here let us restrict the discussion to the transient analysis which is most closely related to the subjects of this thesis.

2.2 Transient Analysis: Implicit Integration vs Implicit Differentiation

Transient analysis determines the time domain response of the circuit over a specified time interval $[0, T]$. The initial value is either specified by the user or determined by a dc operating point analysis. At the discrete time points within the time interval $[0, T]$, a numerical integration (differentiation) algorithm is employed to transform the differential model equations of each energy-storage element into equivalent algebraic equations. Then the time point solution is determined by Newton-Raphson iteration.

Numerical integration algorithms can be either explicit or implicit. For the reason of stability properties (section 1.2) the implicit methods are far superior for circuit simulation. Also it is noted that very frequently the circuits of interest today are heavily nonlinear due to the increasing application of integrated circuits (IC). Let us now look at a p-n junction, one of the very basic units in IC design. We know it could be modeled as a diode and a nonlinear capacitor connected in parallel (Fig. 2.2). [Cellier, 1991]

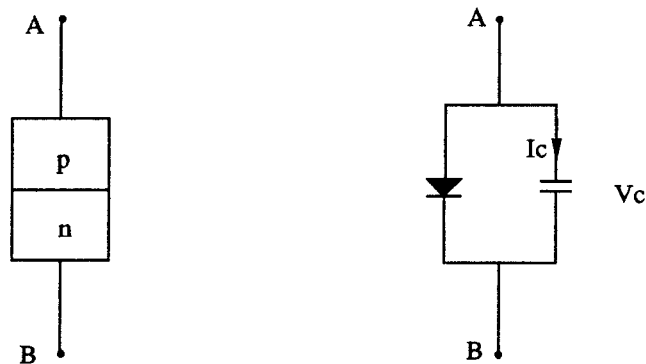


Fig. 2.2 p-n junction

For the nonlinear diffusion capacitor we have:

$$Q_c = f(V_c) = \tau I_s [e^{V_c/V_t} - 1],$$

$$I_c = \dot{Q}_c.$$

where I_s is the saturation current of the diode, τ is the time constant of the capacitor, and V_t is the thermal voltage. Then we use the k th order Backward Differentiation formula (BDF which is a stiffly stable method and was implemented in SPICE) to solve the above ODE:

$$x_{n+1} = \sum_{j=1}^k \alpha_j x_{n-j+1} + h\beta_0 \dot{x}_{n+1},$$

$$\Rightarrow \dot{Q}_{n+1} = \frac{1}{h\beta_0} (Q_{n+1} - \sum_{j=1}^k \alpha_j Q_{n-j+1}),$$

$$\Rightarrow I_{n+1} = \frac{1}{h\beta_0} \tau I_s (e^{\frac{V_{n+1}}{V_t}} - 1) - \frac{1}{h\beta_0} (\sum_{j=1}^k \alpha_j Q_{n-j+1}).$$

We would like to use the above BDF algorithm to solve this ODE, *i.e.* solve for Q_{n+1} . But the real situation is that V_c is already given or determined by previous calculation and $Q_c = f(V_c)$ does not have an analytical inverse ($V_c = \psi(I_c = \dot{Q}_c)$). Therefore, SPICE has no choice but to compute Q_c from $Q_c = f(V_c)$ and then numerically DIFFERENTIATE the result to compute dQ_c/dt . [Cellier, 1991] Thus we may consider that inside SPICE, instead of implicit numerical integration, the ODE is solved by "implicit numerical differentiation". Furthermore, we will define this type of ODE

formulated in the above example as an *inverse problem* which has the following general format:

$$\dot{x} = g(\dot{x}, t), \quad \dot{x}(t_0) = \dot{x}_0. \quad (2.1)$$

2.3 Does SPICE Really Employ Gear's Methods?

There is another problem worth to be mentioned here. In SPICE, there are two numerical methods available for solving ODE systems resulting from electronic circuits, one is the Trapezoidal method which, as we know, is a second order implicit method with an A-stability domain. The other is the so-called "Gear's algorithm" which is to be discussed in more detail: In SPICE, the "Gear's method"[Nagel,1975] starts from the rearranged BDF algorithm:

$$\dot{x}_{n+1} = \sum_{i=0}^k \alpha_i x_{n-i+1}. \quad (2.2)$$

Let us use the following kth order polynomial to do backward interpolation:

$$y(t) = c_0 + c_1(t_{n+1}-t) + \dots + c_k(t_{n+1}-t)^k. \quad (2.3)$$

The c_i can be obtained by fitting (2.3) to the $k+1$ values $(x_{n+1}, x_n, \dots, x_{n+1-k})$ to produce the equation:

$$\begin{bmatrix} x_{n+1} \\ x_n \\ \vdots \\ x_{n+1-k} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 1 & h & \dots & h^k \\ \vdots & \vdots & \ddots & \vdots \\ 1 & kh & \dots & (kh)^k \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_k \end{bmatrix}. \quad (2.4)$$

Substituting (2.4) into (2.2) yields the equation:

$$\begin{aligned}
 -c_1 = & \alpha_0 c_0 + \alpha_1 (c_0 + c_1 h + \dots + c_k h^k) \\
 & + \alpha_2 (c_0 + c_1 (2h) + \dots + c_k (2h)^k) \\
 & \vdots \\
 & + \alpha_k (c_0 + c_1 (kh) + \dots + c_k (kh)^k).
 \end{aligned} \tag{2.5}$$

Comparing the coefficients yields the system:

$$\begin{bmatrix} 1 & 1 & \dots & 1 \\ 0 & h & \dots & kh \\ 0 & h^2 & \dots & (kh)^2 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & h^k & \dots & (kh)^k \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_k \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{2.6}$$

Solving for α_i and substituting into (2.2) gives the BDF methods for fixed stepsize h . For variable stepsize, the results depend on h_1, h_2, \dots, h_k which might have different values.

So the equation with α_i becomes: [Nagel, 1975]

$$\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 0 & h_n & h_n + h_{n-1} & \dots & \sum_{i=1}^k h_{n+1-i} \\ 0 & h_n^2 & (h_n + h_{n-1})^2 & \dots & (\sum_{i=1}^k h_{n+1-i})^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & h_n^k & (h_n + h_{n-1})^k & \dots & (\sum_{i=1}^k h_{n+1-i})^k \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_k \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{2.7}$$

which means that the α_i ($i = 0, 1, \dots, k$ and $k = 1, 2, \dots, 6$) need to be recomputed for every time step in order to obtain \dot{x}_{n+1} and x_{n+1} .

One question arises here: Is this really Gear's method? In chapter 1 we mentioned that Gear's method is based on (originated from) the BDF (which is the same as in

SPICE) predictor-corrector pair and then employs the Nordsieck technique to transform the multistep method to a "single step" matrix format. One of the major advantages is that after this transformation, the order and step size control can be achieved much more easily (the quite expensive recomputation of α_i is no longer required here). Another advantage of the single step format comes to play when we try to upgrade the method for stability improvement. [Skeel, 1977] It is more convenient to be handled in the Nordsieck form than in the multistep format. (for instance, the blending technique discussed later) Although "Gear's method" as used in SPICE adjusts order and step size, this is what we see in other methods also (Trapezoidal). In fact, the "Gear's method" in SPICE is nothing more than basically a BDF method whose application for stiff systems goes back to Curtis and Hirschfelder (1952) [Lambert, 1973]

2.4 Motivation for Developing the DBDF Method

In 1.2 and 1.3 we talked about the stability regions of some numerical methods. We realized that the implicit methods are superior to the explicit methods for the circuit simulation due to their improved stability properties. Also we noted that for those implicit algorithms, all of them fall into either one of two categories: One (as the 4th order AMF method) has a closed stability domain within the left complex plane. The other (as the third order BDF algorithm) has a closed instability domain within the right half complex plane. If a system has eigenvalues close to the imaginary axis, neither type of those algorithms can ever represent the analytical stability domain correctly. (One would have the tendency to make it appear more stable and the other make it appear less

stable.) [Cellier, 1988]. There are several algorithms designed which try to handle this kind of situation. Currently one of the most promising technique is called Blended Linear Multistep Method derived by R. Skeel. He took a combination of 2 multistep formulas, AMF and Gear's methods, which are all in the Nordsieck matrix format with variable coefficients. The resulted blended formula works much better than either of the original methods. (Larger stable region with high accuracy)[SKEEL, 1977], [Cellier, 1988] We would like to try this blending technique for the Trapezoidal method (which is in fact a second order AMF) and BDF in SPICE. Unfortunately, this cannot be done easily because: 1) Both Trapezoidal and BDF in SPICE are not in matrix form which is necessary for the blending; 2) The blending technique derived by Skeel deals with the system (1.1) but the real systems we come up in circuit simulation basically are in the format (2.1), i.e., in the inverse formulation. It means we will need to rederive the blending algorithm for the inverse systems. 3) To achieve 2), we can neither use directly the existing TRAP and BDF in SPICE (which can deal with inverse systems, but not in matrix format), nor the widely used Gear's method (which is in the matrix format, but cannot handle inverse ODE systems). We really need to derive an implicit multistep numerical differentiation method which can solve the inverse initial value systems and is specified in the "single" step matrix format. The detailed derivation, analysis and implementation of a new method (we called it DBDF, which stands for Differentiation method based on BDF algorithm) will be the topic of the remaining chapters.

CHAPTER 3

DEVELOPMENT OF DBDF METHOD

From chapter 2 it is known that the most common problem raised in transient analysis of electronic circuits is how to solve an inverse initial value problem (I.V.P.) accurately and efficiently. Until now, a number of numerical methods have been proposed for solving the general form of I.V.P. represented as

$$\dot{x} = f(x, t), \quad x(t_0) = x_0. \quad (3.1)$$

But there are few methods available to deal with the inverse problem

$$x = g(\dot{x}, t), \quad \dot{x}(t_0) = \dot{x}_0. \quad (3.2)$$

In this chapter, we shall derive in much detail a numerical method for solving the inverse I.V.P. (3.2), which is stiffly stable and able to adjust the stepsize and order automatically. We call this new method DBDF method, which stands for Differentiation algorithm from Backward Difference Formula. The algorithm derivation will be given first, then it is to be expanded to a formulation for solving general inverse differential systems.

3.1 A Multistep Numerical Scheme

Because of the larger absolute stability domain compare to some other algorithms (chapter 1), let us once again consider the BDF scheme:

$$x_{n+1} = \sum_{i=1}^k \alpha_i x_{n-i+1} + h\beta_0 f_{n+1}. \quad (3.3)$$

The local truncation error of (3.3) is $c_{k+1}x^{(k+1)}(\xi)h^{k+1}$. The α_i , β_0 and c_{k+1} for $k = 1, 2 \dots 5$ are given in table 3.1.

Table 3.1 Coefficients and error constants of BDF scheme

k	1	2	3	4	5
β_0	1	2/3	6/11	12/25	60/137
α_1	1	4/3	18/11	48/25	300/137
α_2		-1/3	-9/11	-36/25	-300/137
α_3			2/11	16/25	200/137
α_4				-3/25	-75/137
α_5					12/137
c_{k+1}	-1/2	-2/7	-3/22	-12/125	-10/137

From there we try to develop a scheme for solving the inverse I.V.P. (3.2). First note that (3.3) can be written equally as

$$x_{n+1} = \sum_{i=1}^k \alpha_i x_{n-i+1} + h\beta_0 \dot{x}_{n+1}. \quad (3.4)$$

therefore

$$\dot{x}_{n+1} = \frac{1}{\beta_0 h} (x_{n+1} - \sum_{i=1}^k \alpha_i x_{n-i+1}). \quad (3.5)$$

Considering (3.2) it follows

$$\dot{x}_{n+1} = \frac{1}{\beta_0 h} (g(\dot{x}_{n+1}, t_{n+1}) - \sum_{i=1}^k \alpha_i x_{n-i+1}). \quad (3.6)$$

Note that (3.6) is an equation with respect to \dot{x}_{n+1} , which can be solved by Newton iteration. Let

$$\left\{ \begin{array}{l} \dot{x}_{n+1}^{(0)} = \dot{x}_n^{(m)} \\ \dot{x}_{n+1}^{(l)} = \dot{x}_{n+1}^{(l-1)} - \frac{g(\dot{x}_{n+1}^{(l-1)}, t_{n+1}) - \sum_{i=1}^k \alpha_i x_{n-i+1} - \beta_0 h \dot{x}_{n+1}^{(l-1)}}{\frac{\partial g}{\partial \dot{x}}(\dot{x}_{n+1}^{(l-1)}, t_{n+1}) - \beta_0 h} \\ l = 1, 2, \dots, m \end{array} \right. \quad (3.7)$$

A later discussion (chapter 4) will show that $m=3$ is sufficient for a given LTE bound.

Combining (3.7) and (3.4) gives

$$\left\{ \begin{array}{l} \dot{x}_{n+1}^{(0)} = \dot{x}_n^{(m)} \\ \dot{x}_{n+1}^{(l)} = \dot{x}_{n+1}^{(l-1)} - \frac{g(\dot{x}_{n+1}^{(l-1)}, t_{n+1}) - \sum_{i=1}^k \alpha_i x_{n-i+1} - \beta_0 h \dot{x}_{n+1}^{(l-1)}}{\frac{\partial g}{\partial \dot{x}}(\dot{x}_{n+1}^{(l-1)}, t_{n+1}) - \beta_0 h} \\ x_{n+1} = \sum_{i=1}^k \alpha_i x_{n-i+1} + \beta_0 h \dot{x}_{n+1}^{(m)} \\ l = 1, 2, \dots, m \end{array} \right. \quad (3.8)$$

As an example, apply (3.8) to the test equation

$$\dot{x} = \lambda x,$$

or

$$x = \frac{1}{\lambda} \dot{x}, \quad (\lambda \neq 0).$$

then

$$\begin{aligned} \dot{x}_{n+1}^{(l)} &= \dot{x}_{n+1}^{(l-1)} - \frac{\frac{1}{\lambda} \dot{x}_{n+1}^{(l-1)} - \sum_{i=1}^k \alpha_i x_{n-i+1} - \beta_0 h \dot{x}_{n+1}^{(l-1)}}{\frac{1}{\lambda} - \beta_0 h} \\ &= \frac{\sum_{i=1}^k \alpha_i x_{n-i+1}}{\frac{1}{\lambda} - \beta_0 h}. \end{aligned}$$

so

$$\begin{aligned} x_{n+1} &= \sum_{i=1}^k \alpha_i x_{n-i+1} + \beta_0 h \frac{\sum_{i=1}^k \alpha_i x_{n-i+1}}{\frac{1}{\lambda} - \beta_0 h} \\ &= \frac{\sum_{i=1}^k \alpha_i x_{n-i+1}}{1 - \lambda \beta_0 h}. \end{aligned}$$

Note that if, instead of (3.8), the BDF scheme (3.3) is applied to the above test equation, we will get exactly the same numerical result. This fact will lead to a conclusion that the scheme (3.8) has the same absolute stability domain (stiffly stable for order 1 ... 6) as (3.3) that was shown in chapter 1.

3.2 DBDF Method for Solving Implicit I.V.P.

In the last section, we have come out with a numerical scheme for solving inverse

I.V.P.. But (3.8) is neither convenient nor efficient to be used for the same reasons as mentioned in chapter 2. In order to get a highly automatic numerical method for solving inverse I.V.P. problems, the Nordsieck technique [chapter 1] is employed here to achieve our goal. The idea is that, instead of storing a number of back values of x_i , we store up to the k th derivative at a single point of a local interpolation polynomial which represents the solution. The key to adapting the Nordsieck technique is to create the right local interpolation polynomial for the numerical solution of the I.V.P..

Let us try to express (3.8) in a matrix format. First we define

$$D_{n+1}^{(l)} = \frac{\partial g}{\partial \dot{x}}(\dot{x}_{n+1}^{(l)}, t_{n+1}) - \beta_0 h.$$

then from (3.8)

$$\dot{x}_{n+1}^{(l)} = \left(1 + \frac{\beta_0 h}{D_{n+1}^{(l-1)}}\right) \dot{x}_{n+1}^{(l-1)} + \frac{\sum_{i=1}^k \alpha_i x_{n-i+1}}{D_{n+1}^{(l-1)}} - \frac{g(t_{n+1}, x_{n+1}^{(l-1)})}{D_{n+1}^{(l-1)}}, \quad (3.9)$$

which can be equivalently written in a matrix form as:

$$\begin{bmatrix} x_n \\ h\dot{x}_{n+1}^{(l)} \\ x_{n-1} \\ \vdots \\ x_{n-k+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \dots & \dots & 0 \\ \frac{h\alpha_1}{D_{n+1}^{(l-1)}} & 1 + \frac{h\beta_0}{D_{n+1}^{(l-1)}} & \frac{h\alpha_2}{D_{n+1}^{(l-1)}} & \dots & \dots & \frac{h\alpha_k}{D_{n+1}^{(l-1)}} \\ 0 & 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & \dots & 0 & 1 \end{bmatrix} \begin{bmatrix} x_n \\ h\dot{x}_{n+1}^{(l-1)} \\ x_{n-1} \\ \vdots \\ x_{n-k+1} \end{bmatrix}$$

$$+ \frac{hg(t_{n+1}, \dot{x}_{n+1}^{(l-1)})}{D_{n+1}^{(l-1)}} \begin{bmatrix} 0 \\ -1 \\ 0 \\ \vdots \\ \vdots \\ \vdots \\ 0 \end{bmatrix} \quad (3.10)$$

for $l = 1, 2, \dots, m$ and

$$\begin{bmatrix} x_n \\ h\dot{x}_{n+1}^{(0)} \\ x_{n-1} \\ \vdots \\ \vdots \\ x_{n-k+1} \end{bmatrix} = \begin{bmatrix} \alpha_1 & \beta_0 & \alpha_2 & \dots & \dots & \dots & \alpha_k \\ 0 & 1 & 0 & \dots & \dots & \dots & 0 \\ 1 & 0 & 0 & \dots & \dots & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & \dots & 0 \\ 0 & 0 & 0 & \dots & \dots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \dots & \dots & 0 & \vdots \\ 0 & 0 & 0 & \dots & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_{n-1} \\ h\dot{x}_n^{(m)} \\ x_{n-2} \\ \vdots \\ \vdots \\ \vdots \\ x_{n-k} \end{bmatrix} \quad (3.11)$$

for $l = 0$.

Let us define:

$$X_{n+1}^{(l-1)} = \begin{cases} [x_{n-1}, h\dot{x}_n^{(m)}, x_{n-2}, \dots, x_{n-k}]^T & l = 0 \\ [x_n, h\dot{x}_{n+1}^{(l-1)}, x_{n-1}, \dots, x_{n-k+1}]^T & l = 1, 2, \dots, (m+1) \end{cases}$$

$$L = [0, -1, 0, \dots, 0]^T,$$

$$P_{n+1}^{(l-1)} = \frac{h}{D_{n+1}^{(l-1)}}.$$

Let B_0, B_1 be the square matrices defined in (3.11), (3.10) and

$$B_2 = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ \alpha_1 & \beta_0 & \alpha_2 & \dots & \alpha_k \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix}$$

Then (3.8) can be rewritten equivalently as

$$\left\{ \begin{array}{l} X_{n+1}^{(0)} = B_0 X_{n+1}^{(-1)} \\ \quad = B_0 X_n^{(m)} \\ X_{n+1}^{(l)} = B_1 X_{n+1}^{(l-1)} + LP_{n+1}^{(l-1)} g(t_{n+1}, \dot{x}_{n+1}^{(l-1)}) \\ \quad = X_{n+1}^{(l-1)} + P_{n+1}^{(l-1)} [B_2 X_{n+1}^{(l-1)} + Lg(t_{n+1}, \dot{x}_{n+1}^{(l-1)})] \end{array} \right. \quad (3.12)$$

where $l = 1, 2, \dots, m$.

We have, in (3.12), a "one step" form of the predictor-corrector method (3.8). However, if we attempt to change the stepsize, the earlier difficulties are still present, since the vector of back values $X_{n-i+k}^{(0)}$, $i=1,2,\dots,k-1$, contains information computed at a number of different points. Now let us consider a k th order interpolation polynomial $W_k(t)$ which satisfies

$$\left\{ \begin{array}{l} W_k(t_i) = x_i \quad i = n, n-1, \dots, n-k+1 \\ \dot{W}_k(t_n) = \dot{x}_n^{(m)} \end{array} \right.$$

The $W_k(t)$ and its derivatives on t_n can be determined and if we define $U_{n+1}^{(0)}$ to be

$$U_{n+1}^{(0)} = [W_k(t_n), h\dot{W}_k(t_n), \frac{h^2}{2}\ddot{W}_k(t_n), \dots, \frac{h^k}{k!}W_k^{(k)}(t_n)]^T, \quad (3.13)$$

it turns out that the relation between $U_{n+1}^{(0)}$ and $X_{n+1}^{(0)}$ satisfies

$$U_{n+1}^{(0)} = QX_{n+1}^{(0)}. \quad (3.14)$$

where Q is a constant matrix determined by the interpolation conditions. These Q matrices corresponding to $k = 1, 2, \dots, 5$ are given as follows:

$$Q_{k=1} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}; \quad Q_{k=2} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 1 & 0 \end{bmatrix}; \quad Q_{k=3} = \frac{1}{4} \begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 \\ -7 & 6 & 8 & -1 \\ -3 & 2 & 4 & -1 \end{bmatrix};$$

$$Q_{k=4} = \frac{1}{36} \begin{bmatrix} 36 & 0 & 0 & 0 & 0 \\ 0 & 36 & 0 & 0 & 0 \\ -85 & 66 & 108 & -27 & 144 \\ -60 & 36 & 90 & -36 & 216 \\ -11 & 6 & 18 & -9 & 2 \end{bmatrix};$$

$$Q_{k=5} = \frac{1}{288} \begin{bmatrix} 288 & 0 & 0 & 0 & 0 & 0 \\ 0 & 288 & 0 & 0 & 0 & 0 \\ -830 & 600 & 1152 & -432 & 128 & -18 \\ -755 & 420 & 1248 & -684 & 224 & -33 \\ -238 & 120 & 432 & -288 & 112 & -18 \\ -25 & 12 & 48 & -36 & 16 & -3 \end{bmatrix}.$$

Combining (3.12) and (3.14) gives

$$QX_{n+1}^{(1)} = QB_1Q^{-1}U_{n+1}^{(0)} + QLP_{n+1}^{(0)}g(t_{n+1}, \frac{u_2^{(0)}}{h}), \quad (3.15)$$

where $u_2^{(0)}$ is the second element of $U_{n+1}^{(0)}$. Define

$$U_{n+1}^{(1)} = QX_{n+1}^{(1)},$$

then

$$U_{n+1}^{(1)} = QB_1 Q^{-1} U_{n+1}^{(0)} + QLP_{n+1}^{(0)} g(t_{n+1}, \frac{u_2^{(0)}}{h}).$$

Similarly

$$\begin{aligned} U_{n+1}^{(l)} &= QB_1 Q^{-1} U_{n+1}^{(l-1)} + QLP_{n+1}^{(l-1)} g(t_{n+1}, \frac{u_2^{(l-1)}}{h}) \\ &= U_{n+1}^{(l-1)} + P_{n+1}^{(l-1)} [QB_2 Q^{-1} U_{n+1}^{(l-1)} + QLg(t_{n+1}, \frac{u_2^{(l-1)}}{h})]. \end{aligned} \quad (3.16)$$

$l = 1, 2, \dots, m$ and $u_2^{(l-1)}$ is the second element of $U_{n+1}^{(l-1)}$. Note that

$$U_{n+1}^{(0)} = QX_{n+1}^{(0)} = QB_0 X_n^{(m)} = QB_0 Q^{-1} U_n^{(m)},$$

Thus we have obtained a method with one-step format:

$$\left\{ \begin{array}{l} U_{n+1}^{(0)} = QB_0 Q^{-1} U_n^{(m)} \\ U_{n+1}^{(l)} = U_{n+1}^{(l-1)} + P_{n+1}^{(l-1)} [QB_2 Q^{-1} U_{n+1}^{(l-1)} + QLg(t_{n+1}, \frac{u_2^{(l-1)}}{h})] \end{array} \right. \quad (3.17)$$

$$l = 1, 2, \dots, m.$$

Comparing with the original equation (3.8), equation (3.17) has the advantage that the vector $U_{n+1}^{(0)}$ defined by (3.13) contains only information calculated at point t_n . As

a single step numerical formulation, if, for instance, we want to adjust the stepsize from h to rh , all we have to do is to multiply the i th component of $U_{n+1}^{(0)}$ by r^i , $i = 0, 1, 2, \dots, k$.

We will call the scheme (3.17) the DBDF method, which stands for a numerical Differentiation method from the Backward Difference Formula.

3.3 General Formulation for Solving Inverse Systems

The DBDF algorithm developed in 3.1 and 3.2 can only be applied for solving a single inverse O.D.E.. In this section, we will extend the DBDF to a general purpose numerical method for solving inverse initial value systems (I.V.S.). A similar approach as in the previous sections is to be followed here for the derivation purpose. Suppose we are given

$$\begin{cases} \dot{x} = g(t, \dot{x}), \\ \dot{x}(t_0) = \dot{x}_0. \end{cases} \quad (3.18)$$

where

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_s \end{bmatrix}, \quad g(t, \dot{x}) = \begin{bmatrix} g_1(t, \dot{x}_1, \dot{x}_2, \dots, \dot{x}_s) \\ g_2(t, \dot{x}_1, \dot{x}_2, \dots, \dot{x}_s) \\ \vdots \\ g_s(t, \dot{x}_1, \dot{x}_2, \dots, \dot{x}_s) \end{bmatrix}, \quad \dot{x}_0 = \begin{bmatrix} \dot{x}_{1,0} \\ \dot{x}_{2,0} \\ \vdots \\ \dot{x}_{s,0} \end{bmatrix}.$$

Then the Jacobian matrix turns out to be

$$\frac{\partial g(t, \dot{x})}{\partial \dot{x}} = \begin{bmatrix} \frac{\partial g_1}{\partial \dot{x}_1} & \frac{\partial g_1}{\partial \dot{x}_2} & \cdots & \frac{\partial g_1}{\partial \dot{x}_s} \\ \frac{\partial g_2}{\partial \dot{x}_1} & \frac{\partial g_2}{\partial \dot{x}_2} & \cdots & \frac{\partial g_2}{\partial \dot{x}_s} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial g_s}{\partial \dot{x}_1} & \frac{\partial g_s}{\partial \dot{x}_2} & \cdots & \frac{\partial g_s}{\partial \dot{x}_s} \end{bmatrix}. \quad (3.19)$$

Let $x_{i,n}^{(l)}$ and $\dot{x}_{i,n}^{(l)}$ be the l th numerical approximation of $x_i(t_n)$ and $\dot{x}_i(t_n)$, it follows

$$\dot{x}_n^{(l)} = [\dot{x}_{1,n}^{(l)}, \dot{x}_{2,n}^{(l)}, \dots, \dot{x}_{s,n}^{(l)}]^T. \quad (3.20)$$

Let us define the $(k+1) \times s$ matrix

$$X_{n+1}^{(l)} = \begin{bmatrix} x_{1,n} & x_{2,n} & \cdots & x_{s,n} \\ h\dot{x}_{1,n+1}^{(l)} & h\dot{x}_{2,n+1}^{(l)} & \cdots & h\dot{x}_{s,n+1}^{(l)} \\ x_{1,n-1} & x_{2,n-1} & \cdots & x_{s,n-1} \\ \vdots & \vdots & \vdots & \vdots \\ x_{1,n-k+1} & x_{2,n-k+1} & \cdots & x_{s,n-k+1} \end{bmatrix}, \quad l = 0, 1, \dots, m. \quad (3.21)$$

Then from (3.4), (3.18), (3.20) and (3.21), we can obtain the following multistep format for solving the given system (3.18)

$$\begin{cases} X_{n+1}^{(0)} = B_0 X_n^{(m)}, \\ X_{n+1}^{(l)} = X_{n+1}^{(l-1)} + h(B_2 X_{n+1}^{(l-1)} + G_{n+1}^{(l-1)})[(D_{n+1}^{(l-1)})^{-1}]^T \end{cases} \quad (3.22)$$

where

$l = 1, 2, \dots, m.$

where $z_{n+1}^{(l-1)(2)}$ is the second row vector of matrix $Z_{n+1}^{(l-1)}$. Note that the l th column vector $z_{i,n+1}^{(l)}$ in $Z_{n+1}^{(l)}$ represents the l th approximation of the vector $[x_{i,n+1}, hx_{i,n+1}^{(1)}, \dots, h^k/k!x_{i,n+1}^{(k)}]^T$.

To avoid computing the inverse matrix $(D_{n+1}^{(l-1)})^{-1}$ which is quite expensive in (3.23), the following technique can be employed:

Let us define

$$V_{n+1}^{(l-1)} = Z_{n+1}^{(l)} - Z_{n+1}^{(l-1)}, \quad (3.24)$$

and

$$Y_{n+1}^{(l-1)} = [QB_2Q^{-1}Z_{n+1}^{(l-1)} + QG_{n+1}^{(l-1)}(t_{n+1}, \frac{z_{n+1}^{(l-1)(2)}}{h})]h. \quad (3.25)$$

It follows

$$V_{n+1}^{(l-1)} = Y_{n+1}^{(l-1)} [(D_{n+1}^{(l-1)})^{-1}]^T. \quad (3.26)$$

Taking transpose of both side gives

$$(V_{n+1}^{(l-1)})^T = (D_{n+1}^{(l-1)})^{-1} (Y_{n+1}^{(l-1)})^T. \quad (3.27)$$

or

$$D_{n+1}^{(l-1)} (V_{n+1}^{(l-1)})^T = (Y_{n+1}^{(l-1)})^T. \quad (3.28)$$

Solving (3.28) for $(V_{n+1}^{(l-1)})^T$ and taking the transpose again gives $V_{n+1}^{(l-1)}$. Consequently

we get (from 3.24)

$$Z_{n+1}^{(l)} = Z_{n+1}^{(l-1)} + V_{n+1}^{(l-1)}.$$

In practice we don't reevaluate the Jacobian matrix for every iteration step. We keep $D_{n+1}^{(l)}$ as a constant matrix D_{n+1} in (3.23) and (3.24) to reduce the computational overhead. thus if we define

$$A_0 = QB_0Q^{-1}, \quad A_1 = QB_1Q^{-1}, \quad D_{n+1} = D_{n+1}^{(l-1)}$$

for $l = 1, 2, \dots, m$. The practical format of the DBDF can be written as follows

$$\left\{ \begin{array}{l} Z_{n+1}^{(0)} = A_0 Z_n^{(m)} \\ Y_{n+1}^{(l-1)} = A_1 Z_{n+1}^{(l-1)} + QG_{n+1}^{(l-1)} \left(t_{n+1}, \frac{z_{n+1}^{(l-1)}(2)}{h} \right) \\ (Y_{n+1}^{(l-1)})^T = D_{n+1} (V_{n+1}^{(l-1)})^T \\ Z_{n+1}^{(l)} = Z_{n+1}^{(l-1)} + V_{n+1}^{(l-1)} \end{array} \right. \quad (3.29)$$

$l = 1, 2, \dots, m$.

CHAPTER 4

ERROR ANALYSIS OF DBDF METHOD

In this chapter we are first concerned about the theoretical analysis of the LTE for the DBDF method, then we incorporate a heuristic argument and experiences with our result to come up with a more practical algorithm for an error estimate and control.

4.1 LTE Analysis of DBDF

We give the conclusion first.

Conclusion:

Let us apply the DBDF scheme (3.8) to solve

$$x = g(\dot{x}, t), \quad \dot{x}(t_0) = \dot{x}_0.$$

with starting value x_i satisfying

$$|x(t_i) - x_i| \leq b_0 h^k, \quad i = 0, 1, \dots, k$$

where b_0 is a constant. Also we make the assumption that $g(\dot{x}, t)$ is sufficiently smooth with respect to \dot{x} . Let $\partial g / \partial \dot{x}$, $\partial^2 g / \partial^2 \dot{x}$ satisfy, for $t \in [t_0, T]$,

$$\left| \frac{\partial g}{\partial \dot{x}} \right| \geq \delta > 0, \quad \left| \frac{\partial^2 g}{\partial^2 \dot{x}} \right| < \mu, \quad (4.1)$$

where δ , μ are constants. Then the local truncation error (LTE)

$$\tau_{n+1} = x(t_{n+1}) - \left[\sum_{i=1}^k \alpha_i x_{n-i+1} + \beta_k h \dot{x}_{n+1}^{(m)} \right]$$

satisfies

$$|\tau_{n+1}| \leq ch^{k+1}.$$

This can be proven as follows:

We note that in (3.14), (3.16), $U=QX$, Q is a constant matrix; the first and second components of U are always identical with that in X , (section 3.1) respectively, hence we know (3.17) has the same error behavior as (3.8). Instead of (3.17), it is sufficient to estimate the LTE for (3.8). We approach that in two steps:

i) Consider the local error with the assumption that the Newton iteration is convergent to the exact solution of \dot{x}_{n+1} , i.e., consider

$$\begin{aligned} x_{n+1} &= \sum_{i=1}^k \alpha_i x_{n-i+1} + \beta_0 h \dot{x}_{n+1}, \\ \dot{x}_{n+1} &= \frac{1}{\beta_0 h} \left[g(\dot{x}_{n+1}, t_{n+1}) - \sum_{i=1}^k \alpha_i x_{n-i+1} \right]. \end{aligned} \quad (4.2)$$

We define

$$\tau_{n+1} = x(t_{n+1}) - \left[\sum_{i=1}^k \alpha_i x(t_{n-i+1}) + \beta_0 h \dot{x}_{n+1} \right]$$

which can be written as

$$\begin{aligned} \tau_{n+1} &= \left\{ x(t_{n+1}) - \left[\sum_{i=1}^k \alpha_i x(t_{n-i+1}) + \beta_0 h \dot{x}(t_{n+1}) \right] \right\} + \left\{ \beta_0 h [\dot{x}(t_{n+1}) - \dot{x}_{n+1}] \right\} \\ &= e_1 + e_2. \end{aligned} \quad (4.3)$$

From Henrici[1962], there is a constant b_1 such that

$$\begin{aligned} e_1 &= x(t_{n+1}) - [\sum_{i=1}^k \alpha_i x(t_{n-i+1}) + \beta_0 h \dot{x}(t_{n+1})] \\ &= b_1 h^{k+1}. \end{aligned} \quad (4.4)$$

To estimate e_2 , note that from (4.2) and the assumption (4.1), it follows

$$\begin{aligned} \dot{x}(t_{n+1}) - \dot{x}_{n+1} &= \dot{x}(t_{n+1}) - \frac{1}{\beta_0 h} [g(t_{n+1}, \dot{x}_{n+1}) - \sum_{i=1}^k \alpha_i x(t_{n-i+1})] \\ &= \frac{1}{\beta_0 h} [\beta_0 h \dot{x}(t_{n+1}) + \sum_{i=1}^k \alpha_i x(t_{n-i+1}) - g(t_{n+1}, \dot{x}_{n+1})] \\ &= \frac{1}{\beta_0 h} [\beta_0 h \dot{x}(t_{n+1}) + \sum_{i=1}^k \alpha_i x(t_{n-i+1}) - g(t_{n+1}, \dot{x}(t_{n+1}))] \\ &\quad + \frac{1}{\beta_0 h} [g(t_{n+1}, \dot{x}(t_{n+1})) - g(t_{n+1}, \dot{x}_{n+1})] \\ &= \frac{1}{\beta_0 h} b_1 h^{k+1} + \frac{1}{\beta_0 h} \frac{\partial g}{\partial \dot{x}}(t_{n+1}, \xi) [\dot{x}(t_{n+1}) - \dot{x}_{n+1}] \end{aligned}$$

where the Mean Value Theorem was applied. (ξ is a value between \dot{x}_{n+1} and $\dot{x}(t_{n+1})$) Thus

$$\begin{aligned} [1 - \frac{1}{\beta_0 h} \frac{\partial g}{\partial \dot{x}}(t_{n+1}, \xi)] [\dot{x}(t_{n+1}) - \dot{x}_{n+1}] &= \frac{b_1}{\beta_0} h^{k+1} \\ \dot{x}(t_{n+1}) - \dot{x}_{n+1} &= b_2 h^{k+2} \end{aligned} \quad (4.5)$$

$$e_2 = O(h^{k+2})$$

i.e.

$$\tau_{n+1} = O(h^{k+1}). \quad (4.6)$$

ii) Let us turn to the LTE analysis of (3.8). Assume

$$\begin{aligned}\dot{x}_i^{(m)} &= \dot{x}(t_i) \\ x_i &= x(t_i) \quad i = 1, 2, \dots, n.\end{aligned}\tag{4.7}$$

then the local truncation error of (3.8) turns out to be:

$$\begin{aligned}\epsilon_{n+1} &= x(t_{n+1}) - x_{n+1} \\ &= x(t_{n+1}) - \sum_{i=1}^k \alpha_i x(t_{n-i+1}) - \beta_0 h \dot{x}_{n+1}^m \\ &= x(t_{n+1}) - \sum_{i=1}^k \alpha_i x(t_{n-i+1}) - \beta_0 h \dot{x}_{n+1} + \beta_0 h (\dot{x}_{n+1} - \dot{x}_{n+1}^{(m)}),\end{aligned}$$

where \dot{x}_{n+1} is the exact solution of

$$\dot{x}_{n+1} = \frac{1}{\beta_0 h} [g(\dot{x}_{n+1}, t_{n+1}) - \sum_{i=1}^k \alpha_i x_{n-i+1}].$$

Considering the result (4.6) gives

$$\epsilon_{n+1} = O(h^{k+1}) + \beta_0 h (\dot{x}_{n+1} - \dot{x}_{n+1}^{(m)}).$$

Note that $\dot{x}_{n+1} - \dot{x}_{n+1}^{(m)}$ is the error caused by the Newton iteration. Let us define

$$d_l = \dot{x}_{n+1} - \dot{x}_{n+1}^{(l)}, \quad l = 0, 1, \dots, m.$$

According to Dahlquist[1974],

$$d_l = O(d_{l-1}^2),$$

where again we made use of assumption (4.1).

Since

$$\begin{aligned}
d_0 &= \dot{x}_{n+1} - \dot{x}_{n+1}^{(0)} \\
&= \dot{x}_{n+1} - \dot{x}(t_{n+1}) + \dot{x}(t_{n+1}) - \dot{x}_n^{(m)} \\
&= b_2 h^{k+1} + \dot{x}(t_{n+1}) - \dot{x}(t_n) \\
&= b_2 h^{k+1} + O(h) \\
&= O(h),
\end{aligned}$$

thus

$$d_1 = O(h^2), \quad d_2 = O(h^4), \quad d_3 = O(h^8).$$

For the derived scheme (3.8), $k=1,2,\dots, 5$, we know that it is sufficient to have $m=3$ to guarantee

$$\epsilon_{n+1} = O(h^{k+1}).$$

Now the proof is completed.

4.2 A Practical Algorithm for LTE Estimation

To satisfy a given error bound, let us consider the LTE for a k th order BDF scheme in a single step:

$$c_{k+1} h^{k+1} x^{(k+1)}(t_n) + O(h^{k+2}) \tag{4.8}$$

where the c_{k+1} depends on the method to be used and is given in table 3.1. The $O(h^{k+2})$ term can be neglected and the first term needs to be estimated so that the order k and stepsize h will vary accordingly in order to maximize the stepsize (efficiency consideration). At the same time, the LTE must be limited by a given error bound ϵ . The following approach of estimating the LTE and controlling stepsize and order for the

DBDF scheme mainly come from the strategy employed in Gear's method, [Gear, 1971a] with some modification according to our experimental testing process. To decide whether the computed value of $x(t_n)$ at each step can be accepted or a smaller stepsize or different order need to be selected, we need a algorithm to estimate the LTE first. From chapter 3 the resulted vector for a solution of a single ODE in the n th step has the form:

$$U_n = \left[x_n \quad h\dot{x}_n \quad \dots \quad h^k \frac{x_n^{(k)}}{k!} \right]^T.$$

The backward divided difference of the last component in each previous step leads to the estimation of $h^{k+1} x_n^{(k+1)} / k!$. Consider (4.8), if the given LTE bound is ϵ , then the chosen stepsize h for a k th order scheme must satisfy:

$$\begin{aligned} c_{k+1} k! |\nabla U_k| &= c_{k+1} k! \left| h^k \frac{x_n^{(k)}}{k!} - h^k \frac{x_{n-1}^{(k)}}{k!} \right| \\ &\approx c_{k+1} h^{k+1} |x_n^{(k+1)}| \\ &\leq \epsilon. \end{aligned} \tag{4.9}$$

A detailed strategy of implementation of the DBDF will be given in the next chapter.

CHAPTER 5

IMPLEMENTATION OF DBDF METHOD IN CTRL-C

Just like any other newly developed algorithm in numerical computation, DBDF needs to be implemented in a program for the purpose of experimental testing and evaluation. The program language used here is CTRL-C which is a high level matrix manipulation language and is more convenient to program than other general purpose program languages. The price paid for this convenience is a drastic reduction in execution speed.

In a highly accurate multistep numerical scheme, it is usually uneconomical to keep the stepsize or order constant. The automatic control of them is therefore an important and interesting part of a program for solving initial value systems. From a practical point of view, it is seldom reasonable to ask the user to specify stepsize and order. A method is incomplete unless it contains facilities for determining these parameters automatically. The primary point for the choice is to minimize the total computation to achieve a given error bound. In other word, based on the fact that the amount of work per step is relatively independent of the order, the order is to be chosen such as to maximize the stepsize so that the required accuracy is satisfied. Also as in all other multistep method implementations, a startup procedure must be devised for automatic computation. A detailed discussion is to be given in this chapter.

5.1 Control of Stepsize and Order

We rewrite (4.9) as follows

$$\begin{aligned} c_{k+1}k!|\nabla U_k| &= c_{k+1}k! \left| h^k \frac{x_n^{(k)}}{k!} - h^k \frac{x_{n-1}^{(k)}}{k!} \right| \\ &\approx c_{k+1}h^{k+1}|x_n^{(k+1)}| \\ &\leq \epsilon. \end{aligned}$$

What we really want is to find the largest stepsize for which the predicted local error is acceptable. The basic stepsize control strategies are as follows:

- 1) Equation (4.9) is tested for every computation step.
- 2) Step h is accepted if (4.9) is satisfied; then the new stepsize h_{new} for the next step is estimated.
- 3) Step h is rejected if (4.9) is not satisfied; then the current step is recomputed and retested as 2).

Suppose the new h for the next step or current rejected step is rh , the following equations are used for determining r :

$$\begin{aligned} c_{k+1}k!(1.2r_{k+1})^{k+1}|\nabla U_k| &= \epsilon, \\ r_{k+1} &= \frac{1}{1.2} \left[\frac{\epsilon}{c_{k+1}k!|\nabla U_k|} \right]^{\frac{1}{k+1}}. \end{aligned} \tag{5.1}$$

The coefficient 1.2 is used as a safety factor to keep the resulting LTE from reaching the error bound. There is also the possibility that we could use a larger stepsize with other orders. The following equations are used for determining r for order $k+1$ and $k-1$ respectively:

$$r_{k+2} = \frac{1}{1.4} \left[\frac{\epsilon}{c_{k+2} k! |\nabla^2 U_k|} \right]^{\frac{1}{k+2}}. \quad (5.2)$$

$$r_k = \frac{1}{1.3} \left[\frac{\epsilon}{c_k k! |U_k|} \right]^{\frac{1}{k}}. \quad (5.3)$$

where

$$\nabla^2 U_k \approx \frac{h^{k+2}}{k!} x^{(k+2)},$$

$$U_k \approx \frac{h^k}{k!} x^{(k)}.$$

Note that the different factors (1.2, 1.4, 1.3) in (5.1), (5.2) and (5.3) favor first not changing the order and second not increasing the order based on the work required for one step. Since the theory of the order and stepsize algorithms is quite incomplete, there is no known best way to proceed. The mechanism adopted here is based on tests quoted by Nordsieck(1962), Gear(1971), and Shampine(1975), also on our own experimental test. Some restrictions on the ratio of successive stepsizes are necessary to assure stability in a practical sense and the overhead is considerably reduced when a constant stepsize is used. The selection of a relatively "optimal" stepsize depends on the stepsize not varying too much locally. [Shampine 1975] For these reasons, we tend to choose steps in groups of constant stepsizes with occasional transitions from one stepsize to another. In addition to those basic strategies mentioned earlier, some technical considerations implemented into the CTRL-C program [Appendix] are as follows:

- 1) The new stepsize h_{new} will be permitted to vary between $0.5h$ to $2h$. (we accept r only in the range $[0.5, 2]$) The order is only considered changing by one.
- 2) If $1 \leq r \leq 1.1$, the stepsize is kept unchanged since the increase is not worth the computer time to perform it. If $0.9 \leq r \leq 1$, we make $r = 0.9$. This means that if we need to reduce the stepsize, we do so by a non-trivial amount.
- 3) The estimate of r is made $k+1$ steps after the last change in order or stepsize. (To reduce the overhead of testing too frequently, we will not recompute r for 10 steps if no step increase was made at that time)
- 4) Raising the order will not be considered unless the current step is successful and is unchanged from the last step.
- 5) Lowering the order is considered whether or not the step is successful.
- 6) When the step has failed, our approach is simply to half the stepsize and try again. (A failed step may be due to a somewhat abrupt change in the solution)
- 7) Repeated failures for a single step are a signal that the solution is discontinuous. [Shampine, 1975] We drop the order to one after three unsuccessful trials. This in effect discards all the information in previous steps which is assumed not to be correct and restarts the code.

5.2 Startup and Convergence of Newton-Raphson Iteration

For a multistep method there is always an initial phase during which a strategy is adopted to bring the procedure to a regular situation. Since our DBDF is an order and stepsize variable method and performs error bound testing for every step, we can initially

set the order to one and the stepsize to h_0 . For order one, U_0 is $[x_0 \ h_0 \dot{x}_0]^T$. Since \dot{x}_0 is given and x_0 can be computed from $g(\dot{x}_0, t_0)$, this gives enough information to allow a first order process to be used. From then on, the order and stepsize are chosen automatically.

Some codes require the user to choose an initial stepsize h_0 which is certainly inconvenient because most users have no idea of how to estimate a suitable value. Although what value of h_0 to choose seems not a critical matter for a self testing program, it would certainly effect the amount of computation work and efficiency. Our code selects h_0 according to the given error bound ϵ and the initial value \dot{x}_0 . [Shampine, 1975] The idea is as follows: If a zero order algorithm is considered, then the $x(t_0)$ and LTE are as follows:

$$\begin{aligned} x(t_0) &= x_0 + O(h) \\ &= x_0 + h\dot{x}_0 + O(h^2). \\ e_0 &= h\dot{x}_0 + O(h^2). \end{aligned}$$

For the first order formula, the LTE can roughly be estimated as

$$e_1 \approx h e_0 \approx h^2 \dot{x}_0.$$

To be conservative we want to initially satisfy

$$0.5\epsilon \approx h_0^2 |\dot{x}_0|$$

thus we select

$$h_0 \approx \left| \frac{0.5\epsilon}{\max(|\dot{x}_0|, 1)} \right|^{1/2}.$$

In case of small $|\dot{x}_0|$ (≤ 1), it gives

$$h_0 \approx (0.5\epsilon)^{1/2}.$$

There is one more thing we may need to consider: The convergence of the Newton-Raphson iteration to determine \dot{x}_n at every step. In 4.1. ii) it is shown that the LTE bound would be satisfied if the number of iterations m equals 3. A large number of numerical tests have indicated that is a good choice. There does exist the possibility when the solution does not converge within ϵ in three iterations. Should this happen, our code would reduce the current stepsize to a factor of 1/8 and recompute the solution again. Very likely that would bring the solution inside the LTE bound.

5.3 Construction of DBDF Program

As mentioned earlier the DBDF program is implemented in CTRL-C and consists of a calling program MAIN and five subroutines as shown in Fig. 5.1. MAIN (Fig. 5.2) contains the given system which is initialized by calling GXDS and solved by calling DBDF. MAIN also possesses the ability to give the final simulation results in both the form of values and curves of the solution and error(local and global). Fig. 5.3 gives the flow chart of DBDF which implements the derived multistep method by calling the subprogram SDBDF repeatedly. SDBDF (Fig. 5.4) performs a single step of the

numerical computation of DBDF and contains all the strategies of order and stepsize control, LTE and convergence test and control. The Newton Raphson iteration is accomplished by subroutine NR (Fig. 5.5) which calls GXDS to obtain the function evaluation and calls JXDS to compute the Jacobian. The numerical inversion block INV makes little sense for a program written in CTRL-C but will significantly save the computation effort later when the method is reimplemented in another general purpose computer language.

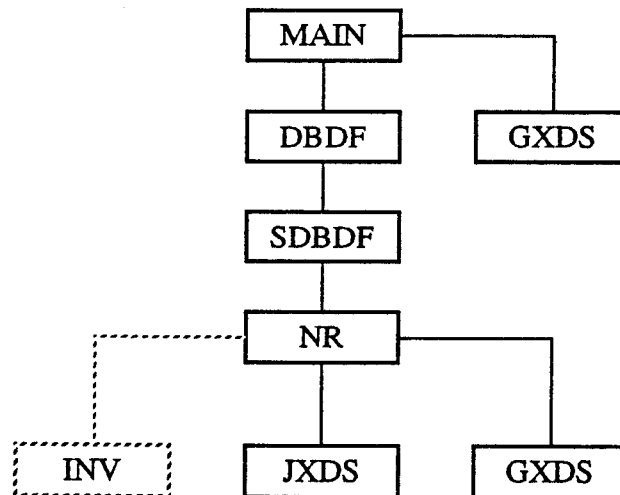


Fig 5.1 Structure of DBDF Program

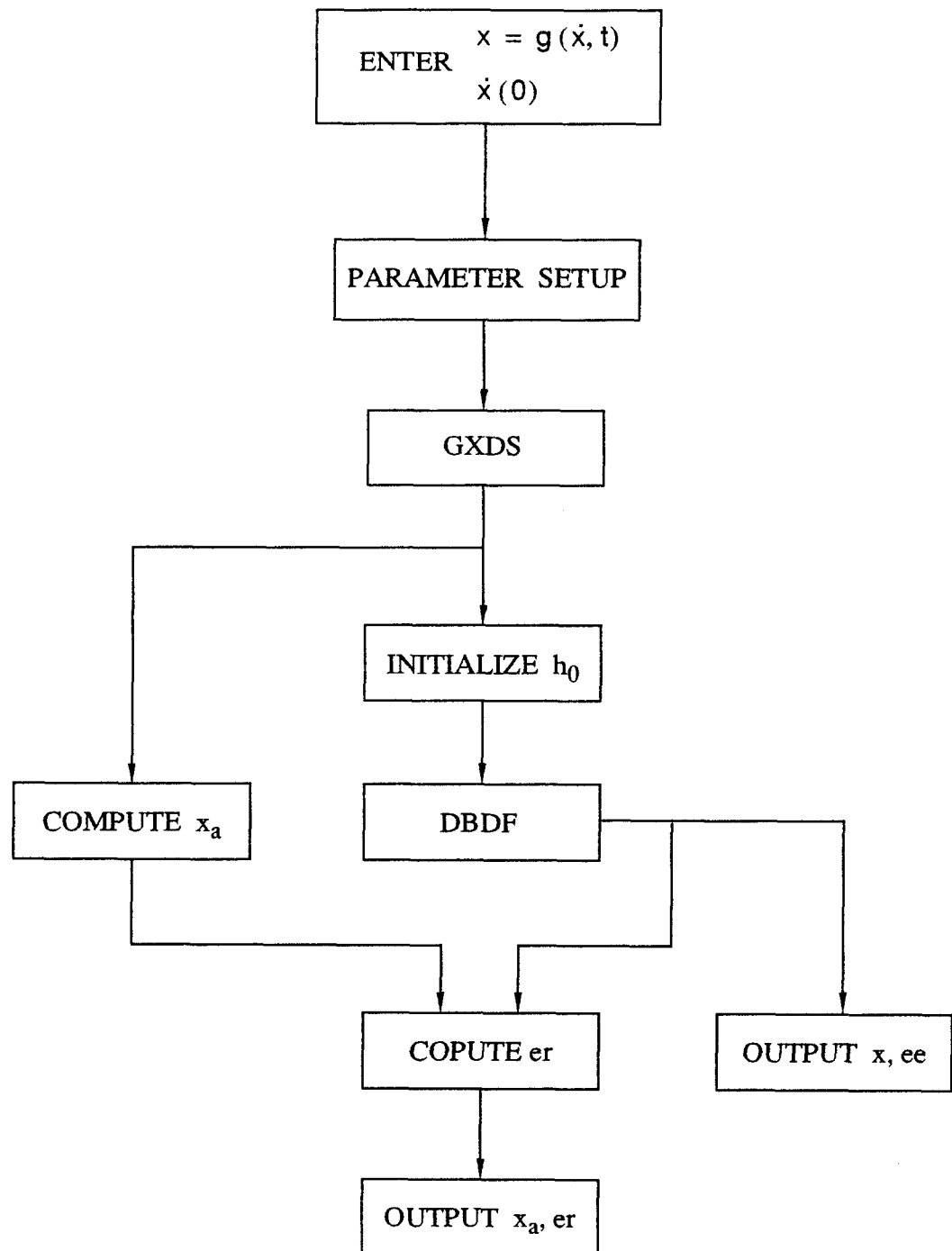


Fig 5.2 Flow Chart of MAIN

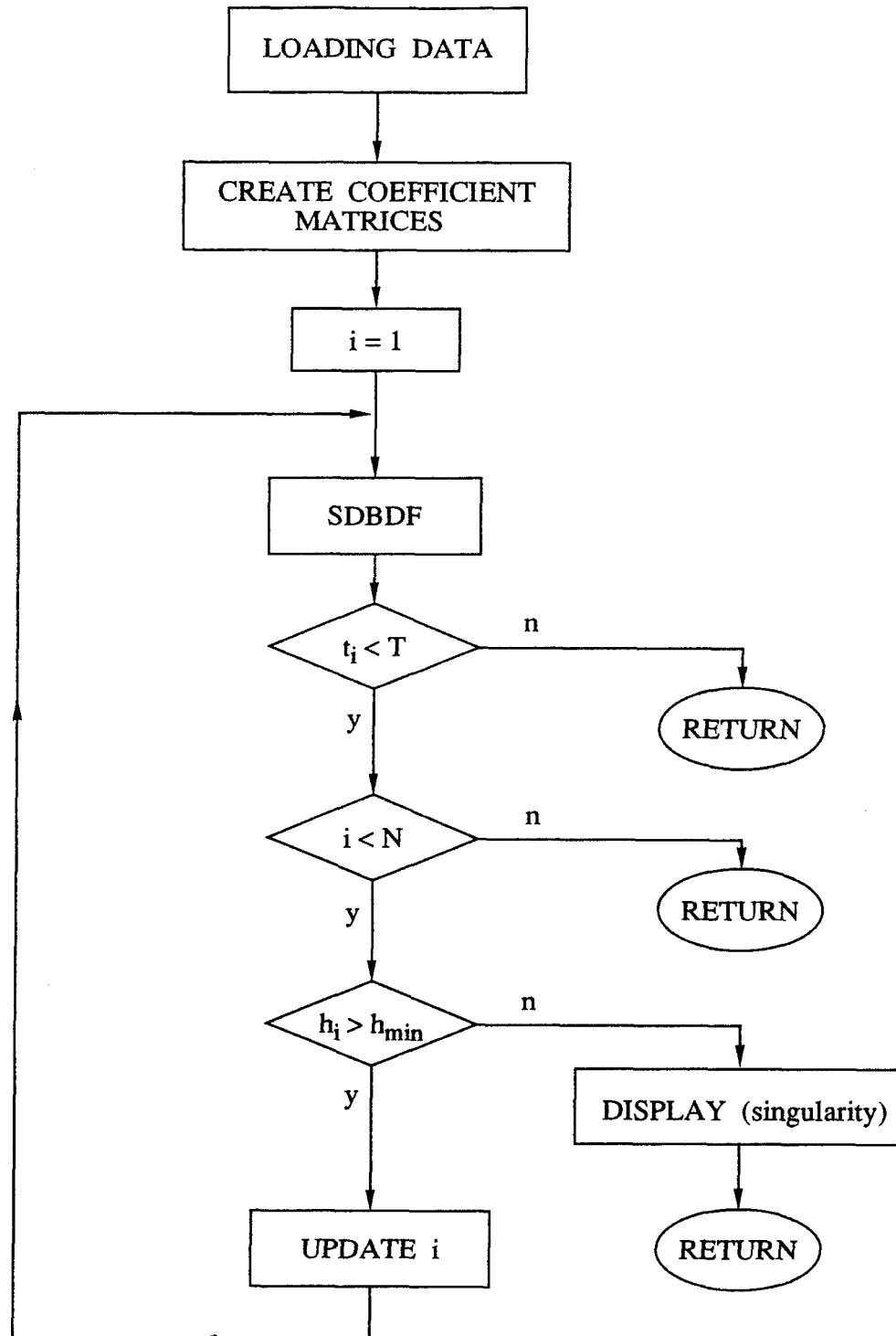


Fig 5.3 Flow Chart of DBDF

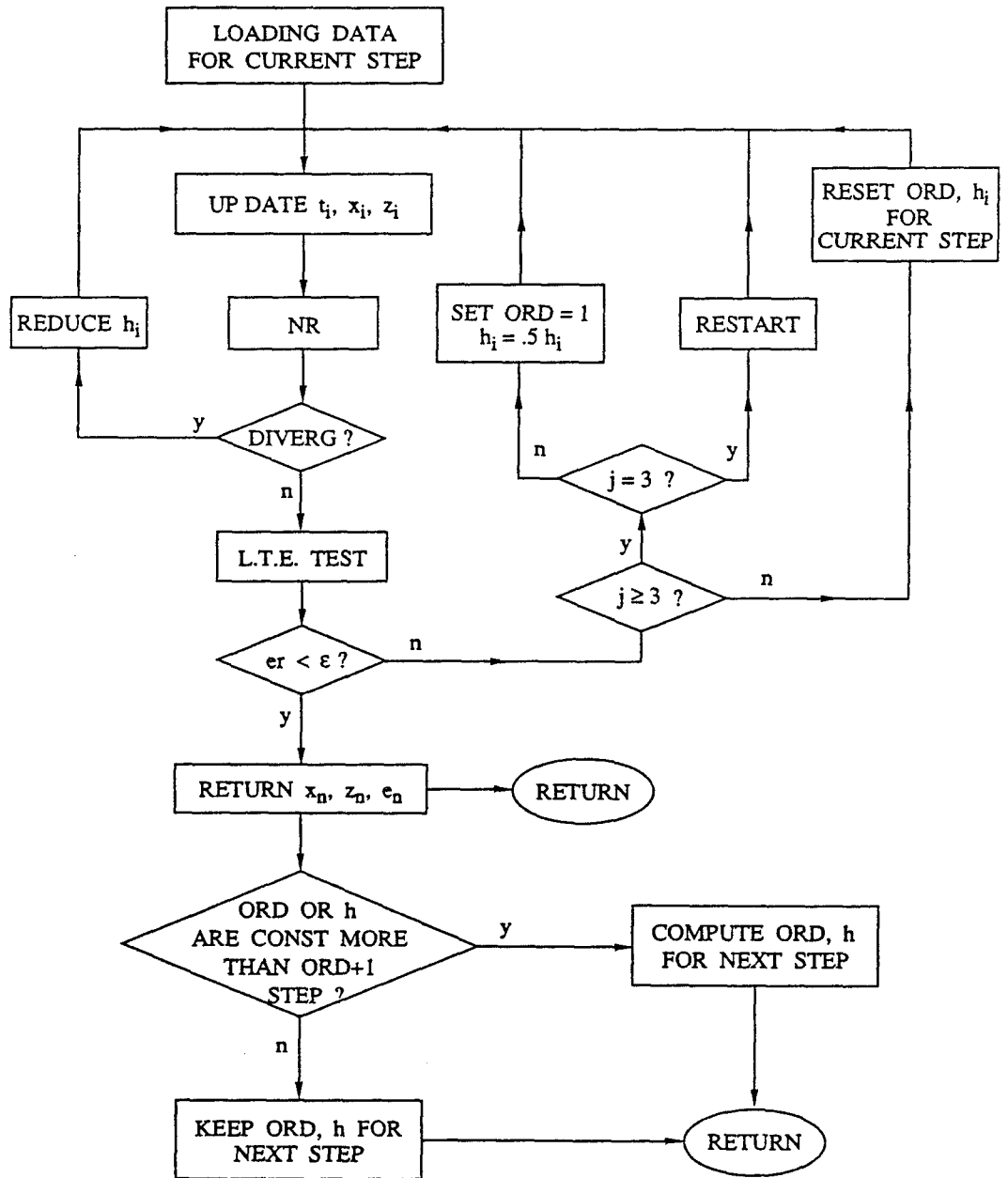


Fig 5.4 Flow Chart of SDBDF

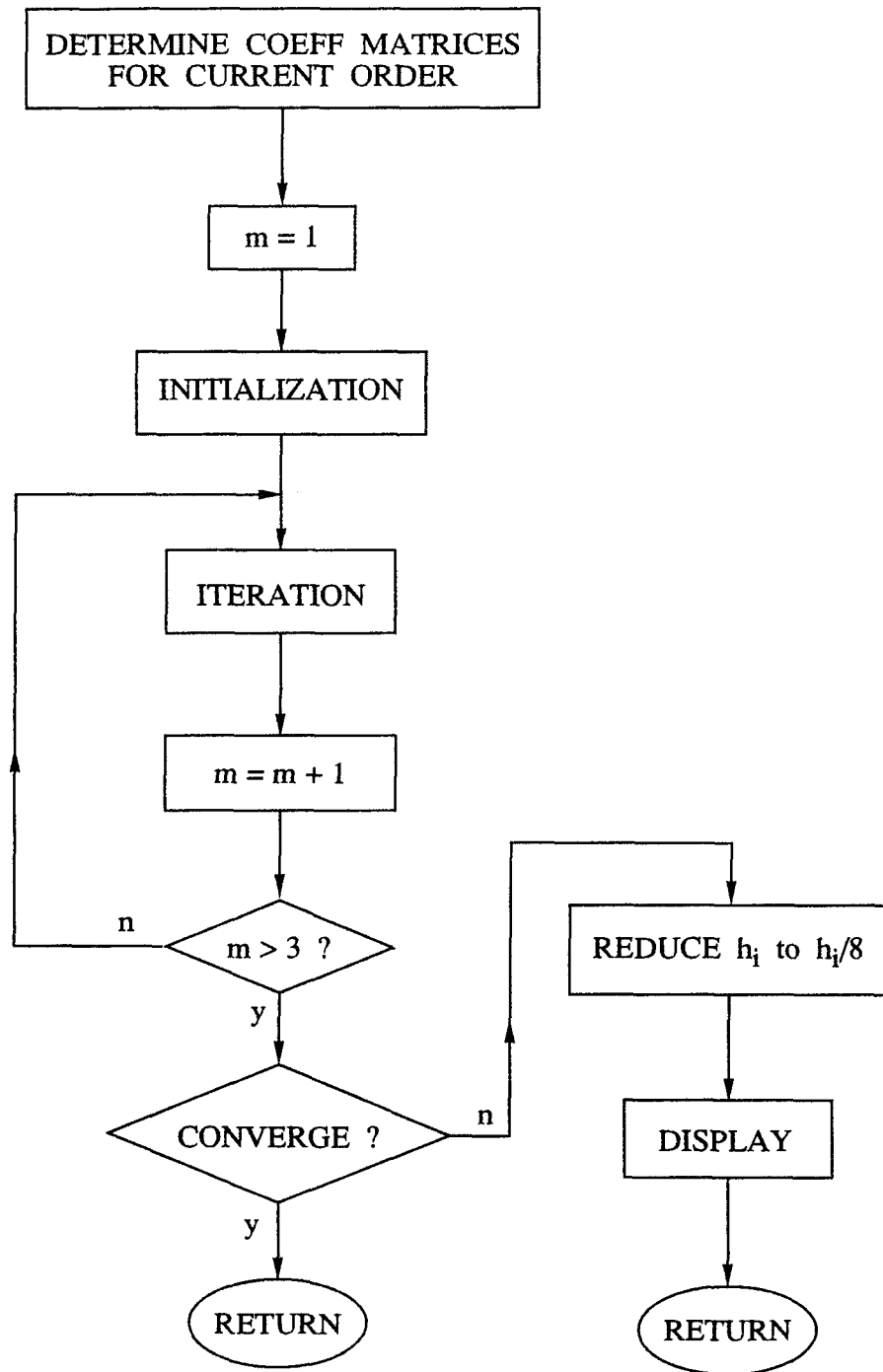


Fig 5.5 Flow Chart of NR

5.4 Numerical Test

A series of numerical tests have been made to compare our DBDF method with the Adams predictor corrector method and Gear's method in ACSL.(Advanced Continuous Simulation Language) We also implemented the Adams and Gear methods with adjustable stepsize and order in CTRLC for our comparison purpose.

As mentioned earlier, our motivation for deriving the DBDF method is to solve implicit ODEs (which may not be invertible to an explicit format). From the point of testing and comparison, we selected intentionally those cases which can be written in either formats. Moreover, in order to estimate the global error, problems with known analytical solution were emphasized. For the following testing cases, we set the LTE bound to $\epsilon=10^{-7}$ for all these methods. Let us first define:

E_1 : estimation of maximum LTE

E_g : estimation of maximum global error

T_c : end of simulation time

N_{int} : number of steps used for a simulation run

x_a : analytical solution

x_j : numerical solution $j=0, 1, \dots, n$

Case 1. Test equation: ($\lambda=-1$)

$$\dot{x} = -x$$

$$x(0) = 1$$

$$x_a = e^{-t}$$

The results of DBDF, Adams and Gear methods are shown in Table 5.1.

	<-----ADAMS----->			<-----GEAR----->	
	DBDF	CTRLC	ACSL	CTRLC	ACSL
* E_g	6.09	1.54	9.59	6.49	11.4
* E_l	0.98	0.67		0.95	
T_e	15	15	15	15	15
N_{int}	102	80		100	

* values $\times 10^{-7}$

It is noted for a test equation with $\lambda = -1$, all these methods give acceptable results. The Adams(CTRLC) is faster since the $(k+1)$ th order corrector allows larger stepsize with the same LTE bound. The solution curves are given in Fig. 5.7.1-2

Case 2. Stiff ODE:

$$\dot{x} = 100(\sin t - x)$$

$$x(0) = 0$$

$$x_a = \frac{1}{1.0001} [\sin t - 0.01 \cos t + 0.01 e^{-100t}]$$

This example is taken from Dahlquist[1974]. was shown that if a Runge-Kutta(R-K) method was employed, for $h=0.03$, the result turns out frightfully unstable Let us look at our results in Table 5.2 and Fig. 5.8.1-2

Table 5.2

	<-----ADAMS----->			<-----GEAR----->	
	DBDF	CTRLC	ACSL	CTRLC	ACSL
* E_g	5.08	490	11.5	3.37	3.17
* E_1	0.995	0.998		0.83	
T_o	5	5	5	5	5
N_{int}	119	546		269	

* values $\times 10^{-7}$

Both of the DBDF and Gear methods give higher accuracy and efficiency for the stiff ODE than the Adams algorithm(which even shows unstable tendency from the E_g curve in Fig. 5.8.2)

Case 3. Stiff system:

$$\dot{x}_1 = x_2$$

$$\dot{x}_2 = -1000x_1 - 1001x_2$$

$$x_1(0) = 1 \quad x_{1a} = e^{-t}$$

$$x_2(0) = -1 \quad x_{2a} = -e^{-t}$$

The eigenvalues turn out to be $s_1 = -1$, $s_2 = -1000$. Note that this system has stiffness ratio equals to 1000! When applying R-K method, it explodes for $h=0.0027$. [Dahlquist, 1974]

Our testing results are shown in Table 5.3 and Fig. 5.9.1-2.(the CTRLC version is not extended to solve systems, therefore only ACSL version available for this case)

Table 5.3

	<-----ADAMS----->			<-----GEAR----->	
	DBDF	CTRLC	ACSL	CTRLC	ACSL
* E_g	6.08		2430		79.2
* E_l	0.98				
T_e	15		15		15
N_{int}	102				

* values $\times 10^{-7}$

The Adams turns out to be entirely unstable. Even the Gear's method in ACSL does not give satisfy solution(we don't know why). The DBDF is successful and the recorded stepsize h_{max} is 0.74.

Case 4. Simple R-C circuit: (Fig. 5.10)

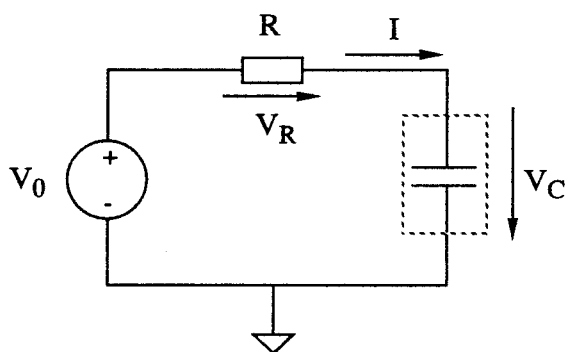


Fig. 5.6 Case 4. (R-C Circuit)

Here we assume the nonlinear capacitor lead to the relation:

$$\begin{aligned}
 Q_c &= g(V_c) = e^{9V_c} - e^{V_c} \\
 &= e^{9(V_0 - IR)} - e^{V_0 - IR} \\
 I &= \dot{Q}_c = \frac{V_0 - V_c}{R}
 \end{aligned}$$

with initial condition:

$$\begin{aligned}
 V_0 &= 1, \\
 V_c(0) &= 0, \\
 \dot{Q}_c(0) &= I(0) = \frac{V_0(0) - V_c(0)}{R} \\
 &= \frac{1}{R}, \quad (R=1k)
 \end{aligned}$$

and

$$\frac{\partial Q_c}{\partial I_c} = e^{9V_0} R e^{-9RI} - 9 R e^{9V_0} e^{-9RI}$$

Since the $Q_c = g(V_c, t)$ is not invertible, we will apply DBDF method to solve this problem.

Table 5.4

	<-----ADAMS----->			<-----GEAR----->	
	DBDF	CTRLC	ACSL	CTRLC	ACSL
* E_s					
* E_1	0.211				
T_c	10000				
N_{int}	224				

* values x 10^{-7}

It is noted that the E_g value is not provided since the analytical solution is not available. The E_1 has increasing tendency at the beginning but the algorithm can guarantee that E_1 will never reach the LTE bound. From this case we can see the DBDF methods are very efficient for solving this type of inverse problem.

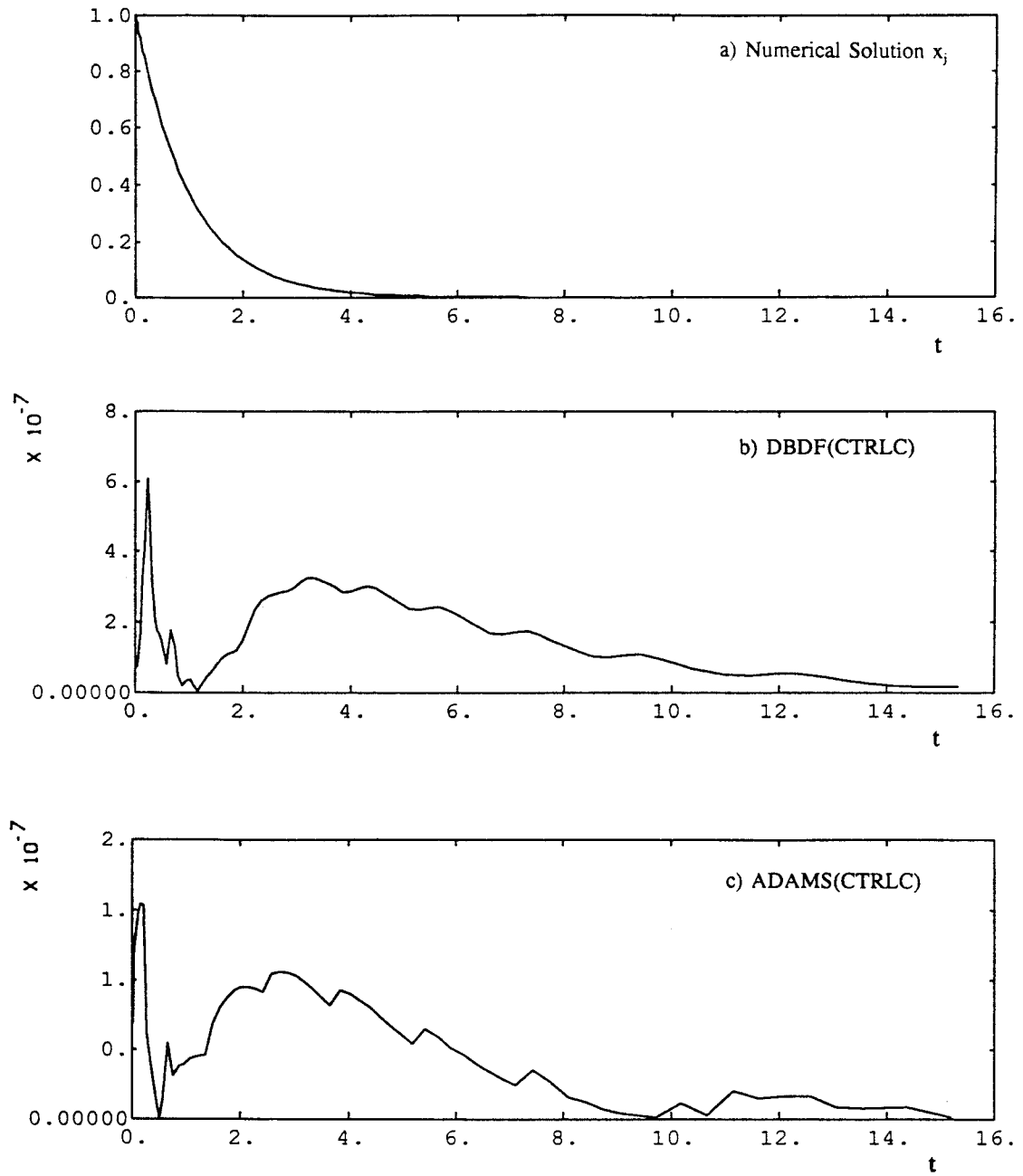


Fig. 5.7.1 Numerical Solution(x_i) and Global Error(E_g) for Case 1

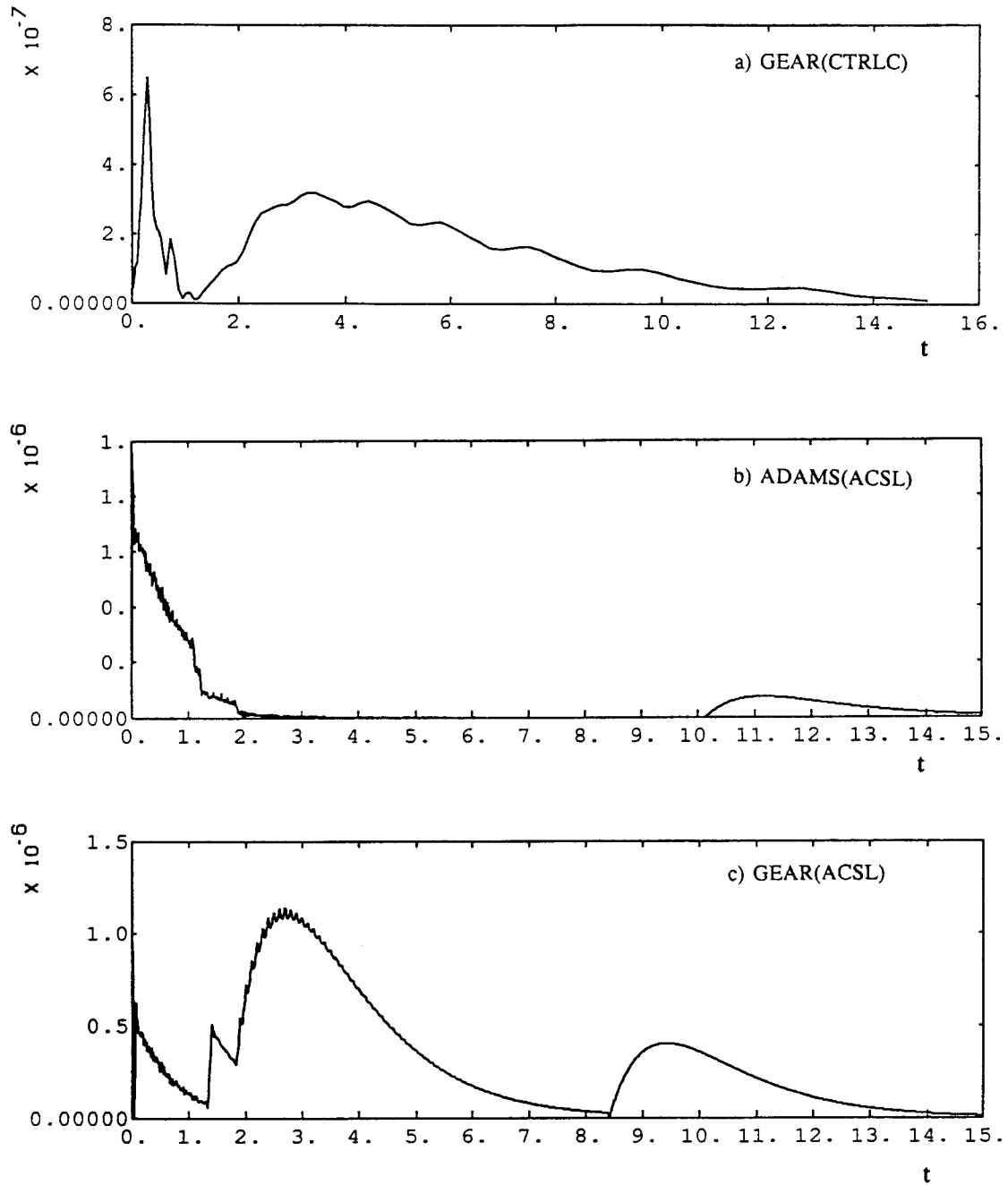


Fig. 5.7.2 Global Error(E_g) for Case 1

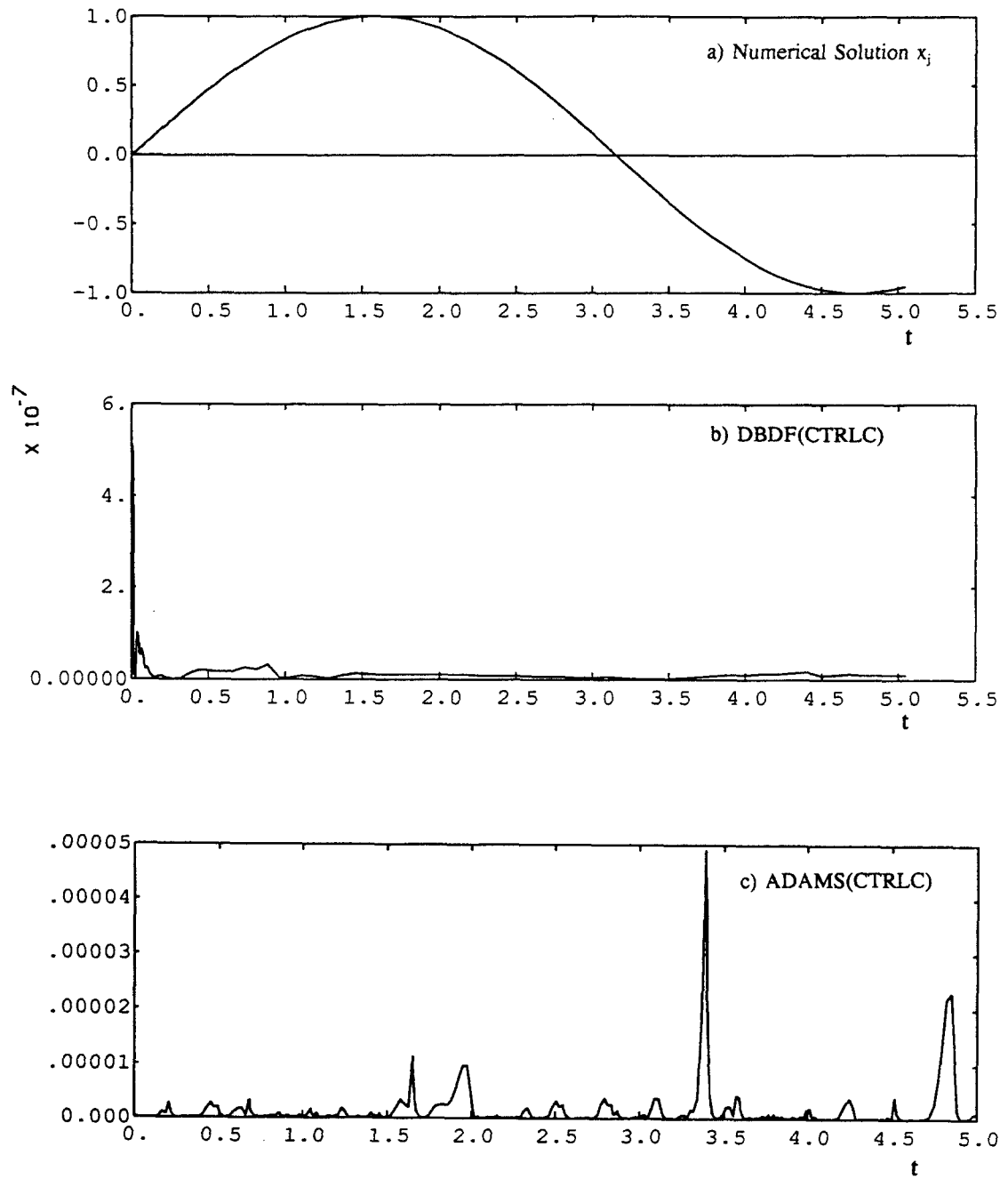
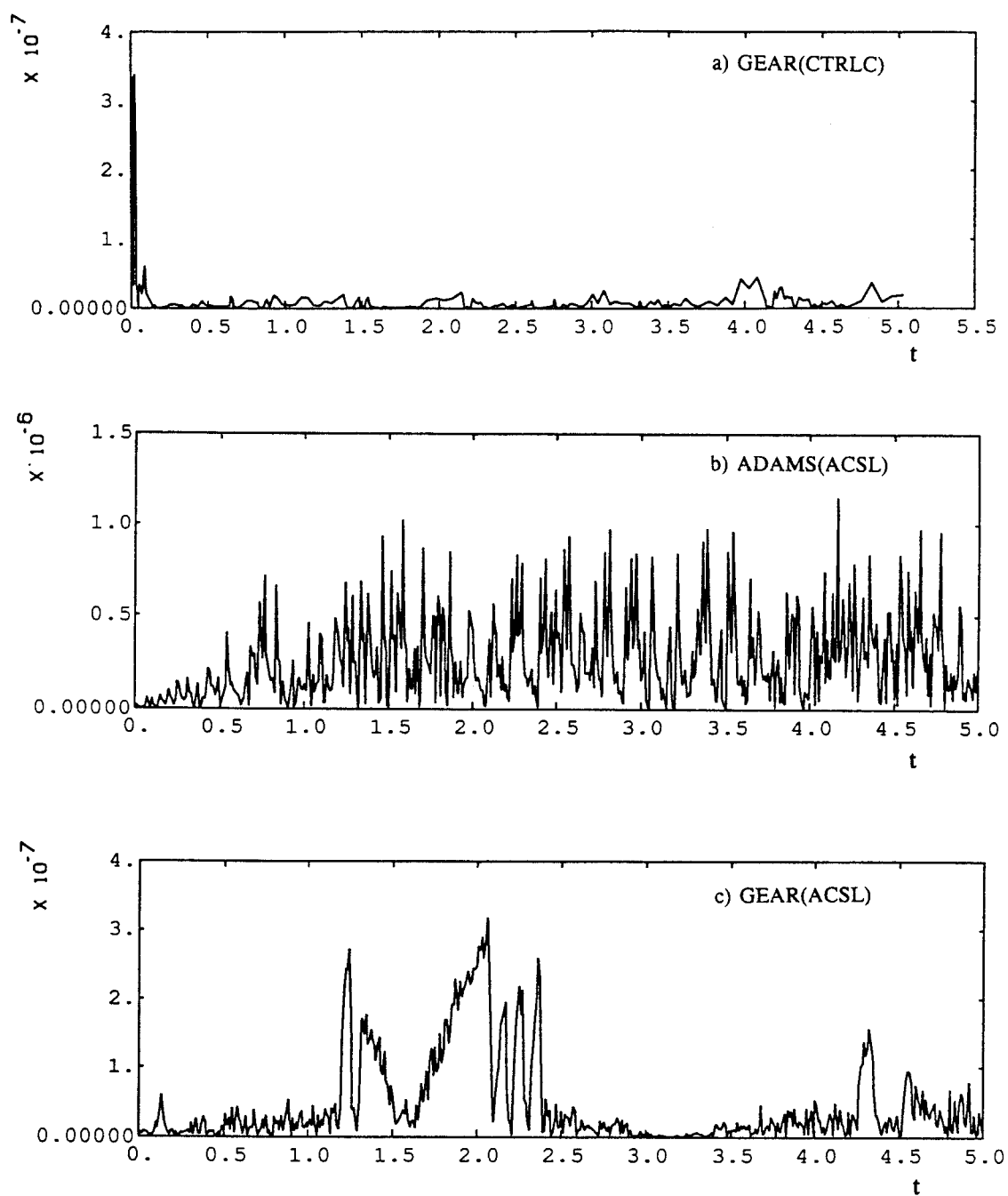


Fig. 5.8.1 Numerical Solution(x_i) and Global Error(E_g) for Case 2

Fig. 5.8.2 Global Error(E_g) for Case 2

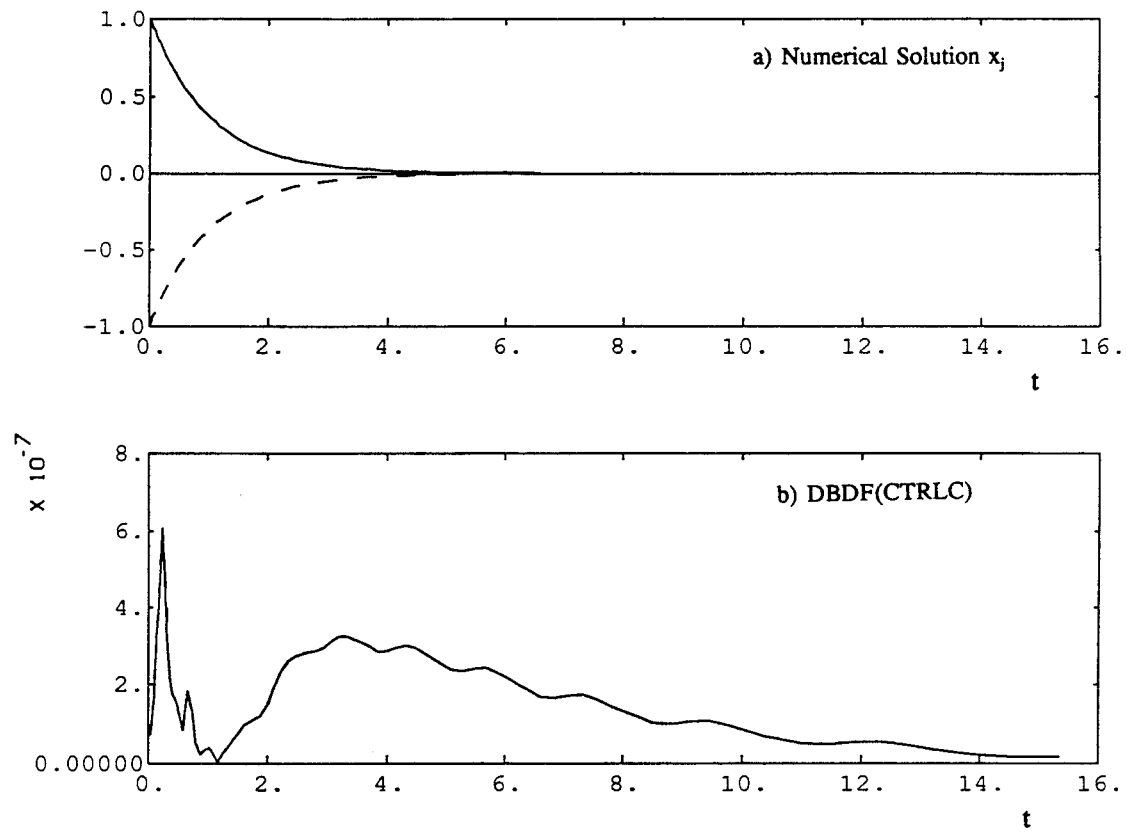


Fig. 5.9.1 Numerical Solution(x_j) and Global Error(E_g) for Case 3

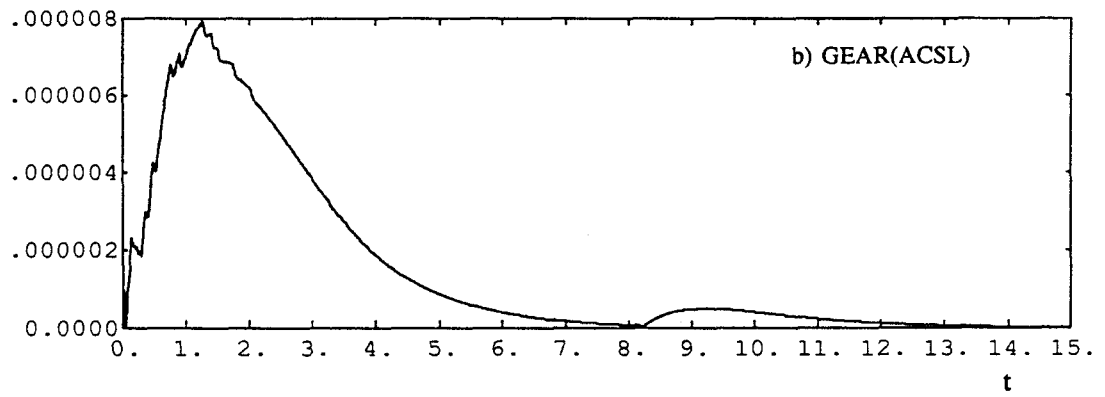
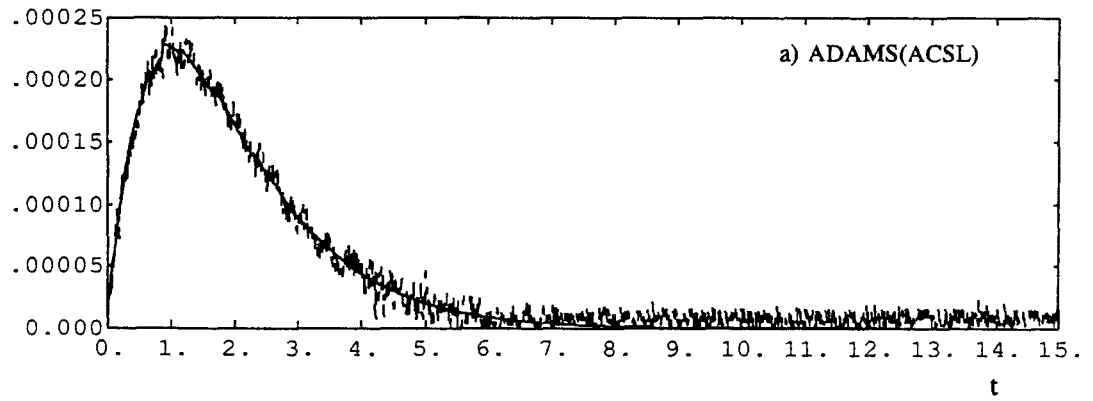


Fig. 5.9.2 Global Error(E_g) for Case 3

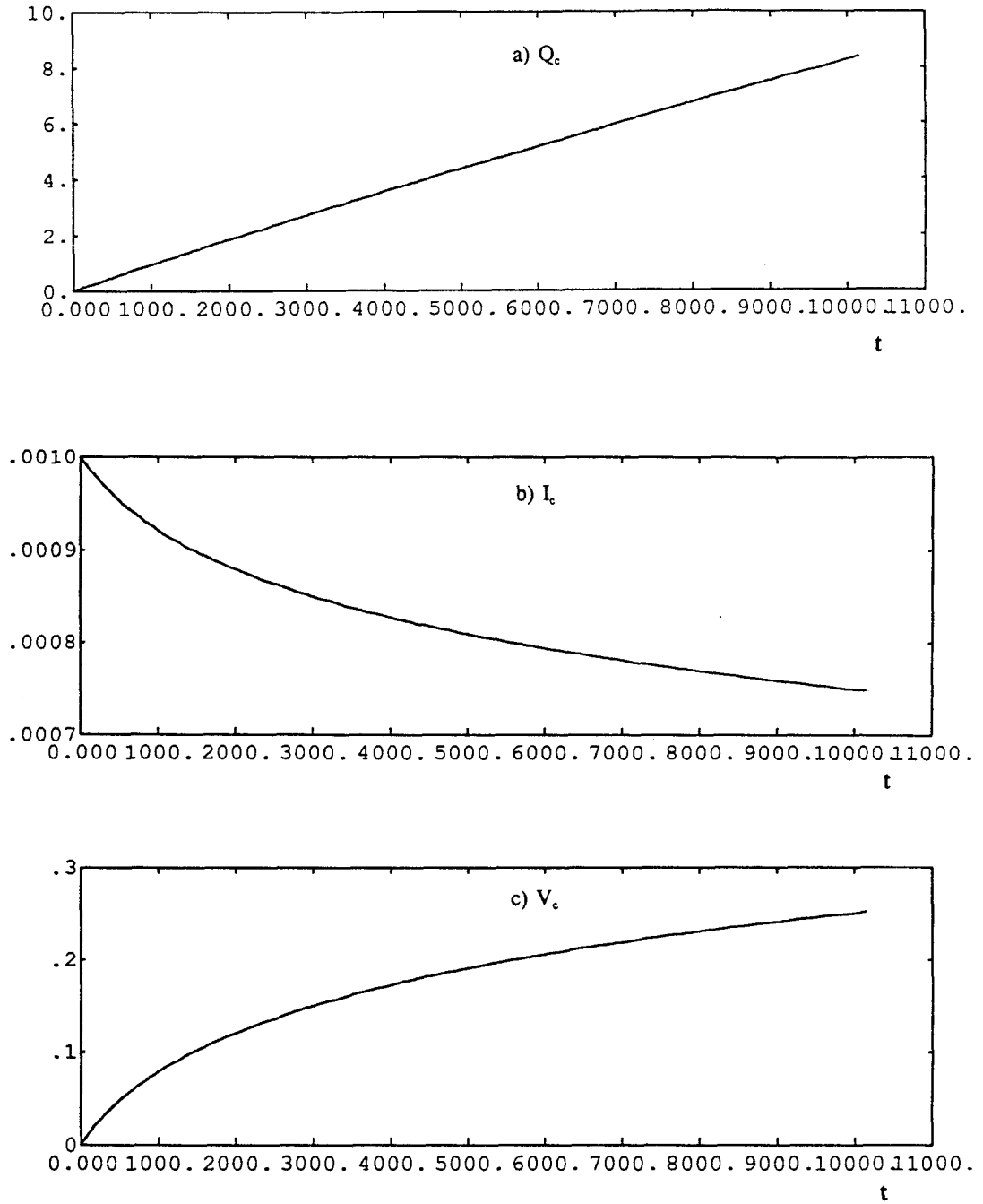


Fig. 5.10.1 Numerical Solutions(Q_c , I_c and V_c) for a R-C Circuit (Case 4)

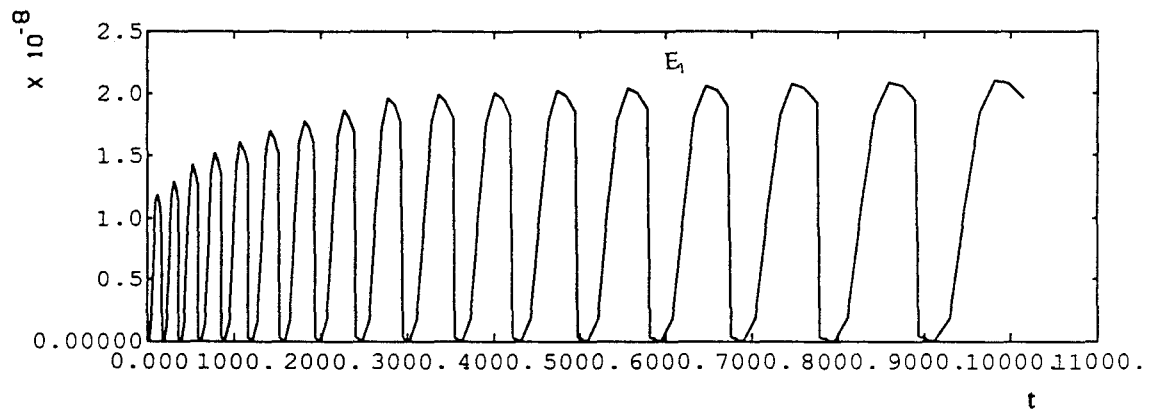


Fig. 5.10.2 LTE (E_1) for a R-C Circuit (Case 4)

CONCLUSION

The analysis and experimental testing given in the above text suggest that the DBDF method is very efficient for the solution of inverse differential systems. It can be incorporated into an highly automated simulation program for various applications. Furthermore, it can fairly easily be further upgraded, e.g. by incorporating the blending technique proposed by Skeel(1977) to achieve a larger stability region and higher accuracy. This is possible due to the consequent utilization of the Nordsieck technique.

Based on what has been discussed in this thesis, a number of further aspects of interest could be discussed:

- 1) We could attempt an explicit global error analysis. In this thesis, the global error was assumed to be approximately one order of magnitude larger than the LTE. This assumption was made by common sense reasoning, and is compatible with the experimental results presented in the thesis.

- 2) After a completed investigation and assessment of the DBDF algorithm, we are ready to derive the blending algorithm for inverse differential systems in accordance with the suggestion made by Skeel(1977), which could lead to a highly robust and efficient simulation software useful for general purpose integrated circuit design.

APPENDIX
SOURCE CODE (CTRLC)


```

//
//***** test file for DBDF algorithm *****
//***** (adjust h & ord) for SYSTEM: x=g(x',t), start from x'(0)
//
DIARY >main.dry
echo=1;
DEFF dbdf
DEFF Sdbdf
DEFF NR
DEFF GXDS
DEFF JXDS
//
long
ordmax=5,
ordset=5,
epsi=1.00-7,
epmin=epsi*1.00-7,
tmax=15,
num=10,
t0=0,
l=2,
//l=1,
//
//[B[Bxd0(1)=-0.01;
//
//xd0(1)=-100;
//xd0(1)=0.001;
//      xd0(1)=0;
//xd0(2)=1;
//
xd0(1)=-1;
xd0(2)=1;
//
x0=GXDS(xd0,t0),
//
//x0=1;
//x0=exp(-t0),
//x0=exp(-t0*(1+0.5*t0)),
//x0=1/(1+t0),
//x0=1/11,
//x0=SIN(t0),
//x0=1/(1+t0)*(3-EXP(-t0)*(2+t0));
//
//compute derivative xd0 as the initial value of 1st iter8.
//
//xd0=-exp(-t0),
//xd0=-(1+t0)*exp(-t0*(1+0.5*t0)),
//xd0=-1/((1+t0)**2),
//xd0=1/((11-t0)**2),
//xd0=COS(t0),
//xd0=0,
xd0m=MAX(ABS(xd0)),
IF xd0m>epsi, ...
  h0=(ABS((0.5*epsi)/xd0m))*0.5, ...
ELSE ...
  h0=(ABS(0.5*epsi))*0.5, ...
END
GLOB (epsi)
GLOB (epmin)
GLOB (l)

```



```

//title('                                glob ERR                                ')
//xlabel('                                T                                ')
ylabel('                                Rg                                ')
replot
erase
page
//***** plot estimated error
window('211')
plot(tt,ee)
//title('                                EST ER                                DBDF                                ')
//xlabel('                                Time                                ')
ylabel('                                E1                                ')
replot
erase
page
//***** plot xa
//window('222')
//plot(tt,xa)
//title('                                ACCURATE VALUE OF NONLINEAR ODE                                ')
//xlabel(' T ')
//ylabel(' Xa ')
//replot
//erase
//page
redhard -close
[nn,mm]=SIZE(tt);
tend=tt(1,mm),
[ee_max,id1]=MAX(ABS(ee)),
[ercolm]=MAX(ABS(er));
[er_max,id2]=MAX(ercolm),
DIARY -off

```

```

// [xxn,een,ttn]=DBDF(x0)
//
// Simulates an SYSTEM: ( x=g(x',t) ) with variable h & ord.
// by use of an DBDFnumerical integration
// algorithm (start-up automaticly from lower orders).
//
// Define the embedded functions
//
// Generate GENERAL coeffecient matrices for the k-step kth-order
// Gear-Nordsieck integration algrathm.(k=1..5)
//
z1=ZROW(1,2);
z2=ZROW(1,3);
z3=ZROW(1,4);
z4=ZROW(1,5);
z5=ZROW(1,6);
//
b1=[1 1;0 1];
b2=1/3*[4 2 -1;0 3 0;3 0 0];
b3=1/11*[18 6 -9 2;0 11 0 0;11 0 0 0;0 0 11 0];
b4=1/25*[48 12 -36 16 -3;0 25 0 0 0;25 0 0 0 0;0 0 25 0 0;0 0 0 25 0];
b5=1/137*[300 60 -300 200 -75 12; 0 137 0 0 0 0;137 0 0 0 0 0;0 0 137 0 0 0
0 0 0 137 0 0;0 0 0 0 137 0];
//
q1=EYE(2);
q2=[1 0 0;0 1 0;-1 1 1];
q3=1/4*[4 0 0 0;0 4 0 0;-7 6 8 -1;-3 2 4 -1];
q4=1/36*[36 0 0 0 0;0 36 0 0 0;-85 66 108 -27 4;-60 36 90 -36 6;-11 6 18 -9 2];
q5=1/288*[288 0 0 0 0 0;0 288 0 0 0 0;-830 600 1152 -432 128 -18
-755 420 1248 -684 224 -33;-238 120 432 -288 112 -18
-25 12 48 -36 16 -3];
//qbq1=b1;
//qbq2=1/3*[3 3 -1;0 3 0;0 0 1];
//qbq3=1/11*[11 11 -1 -7;0 11 0 0;0 0 -1 15;0 0 -2 8];
//qbq4=1/50*[50 50 2 -22 -46;0 50 0 0 0;0 0 -20 45 160;0 0 -20 20 160
0 0 -2 -3 46];
//qbq5=1/274*[274 274 34 -86 -206 -326;0 274 0 0 0 0;0 0 -176 147 744 1615
0 0 -170 19 756 2315;0 0 -30 -45 214 1295;0 0 -2 -3 -4 269 ];
//
qbq1=q1*b1/q1;
qbq2=q2*b2/q2;
qbq3=q3*b3/q3;
qbq4=q4*b4/q4;
qbq5=q5*b5/q5;
//
qb1q1=[z1;1 1];
qb1q2=1/3*[z2;3 3 -1;3 3 -1];
qb1q3=1/22*[z3;22 22 -2 -14;33 33 -3 -21;11 11 -1 -7];
qb1q4=1/150*[z4;150 150 6 -66 -138;275 275 11 -121 -253;150 150 6 -66 -138
25 25 1 -11 -23];
qb1q5=1/3288*[z5
3288 3288 408 -1032 -2472 -3912;6850 6850 850 -2150 -5150 -8150
4795 4795 595 -1505 -3605 -5705;1370 1370 170 -430 -1030 -1630
137 137 17 -43 -103 -163];
q11=[0 -1 0 0 0 0]';
q12=[0 -1 -1 0 0 0]';
q13=1/2*[0 -2 -3 -1 0 0]';
q14=1/6*[0 -6 -11 -6 -1 0]';
q15=1/24*[0 -24 -50 -35 -10 -1]';
c1=-1/2;

```

```

c2=-2/9;
c3=-3/22;
c4=-12/125;
c5=-10/137;
//
b9=[b1 [z3;z3];z5;z5;z5;z5; b2 [z2;z2;z2];z5;z5;z5; b3 [z1;z1;z1;z1];z5;z5
      b4 z4';z5; b5];
q9=[q1 [z3;z3];z5;z5;z5;z5; q2 [z2;z2;z2];z5;z5;z5; q3 [z1;z1;z1;z1];z5;z5
      q4 z4';z5; q5];
qbq9=[qbq1 [z3;z3];z5;z5;z5;z5; qbq2 [z2;z2;z2];z5;z5;z5; qbq3 [z1;z1;z1;z1]
      z5;z5; qbq4 z4';z5;qbq5];
qblq9=[qblq1 [z3;z3];z5;z5;z5;z5; qblq2 [z2;z2;z2];z5;z5;z5; qblq3 [z1;z1;z1
      z1];z5;z5; qblq4 z4';z5; qblq5];

q19=[q11 q12 q13 q14 q15];
gama=[c1 c2 c3 c4 c5];
GLOB (b9)
GLOB (q9)
GLOB (qbq9)
GLOB (qblq9)
GLOB (q19)
GLOB (gama)
//
i=0;
count=0,
hi=h0,
hil=hi;
ord=1;
xxo=x0,
xdo=xd0,
zxo1=ZROW(2,1);
zxo2=ZROW(2,1);
FOR i=1:1, ...
    zxo2(:,i)=[xxo(i);hi*xdo(i)]; ...
END,
tto=t0;
eeo=0;
// a1: count used to control the ord & h adjusting.
a1=0;
// pgxdo: old value of Jaco respect with xd.
pgxdo=0;
GLOB (hi)
GLOB (hil)
GLOB (tto)
GLOB (ord)
GLOB (xxo)
GLOB (eeo)
GLOB (count)
GLOB (a1)
GLOB (pgxdo)
i=0;
WHILE i<num, ...
    i=i+1; ...
    count=count+1, ...
    [zxn1,zxn2,een]=SDBDF (zxo1,zxo2); ...
    xt=CONJ([zxn2(1,1:1)]'); ...
    xxn=[xxo xt]; ...
    xxo=xxn; ...
    zxo1=zxn1; ...
    zxo2=zxn2; ...
    eeo=een; ...

```

```

[nt,mt]=SIZE(tto); ...
time=tto(mt), ...
ttn=[tto tto(mt)+hi]; ...
tto=ttn; ...
hi=hi1, ...
IF hi<epmin, ...
    DISPLAY('Solution may have singularaty here. '); ...
    i=num; ...
END, ...
IF tto(mt+1) > tmax, ...
    DISPLAY('The seting time is over. Simulation stop. '); ...
    i=num; ...
END, ...
...// IF ABS(een(mt+1)) < epmin, ...
...//     DISPLAY('The solution ---> linear as time ---> infinity. Simu stop. '); ...
...//     i=num; ...
...// END, ...
END

```

```

// [zzn1,zzn2,een]=sdbdf(zzo1,zzo2)
//
//*** single step of DBDF for inversed SYSTEM (vary h & ord) ***
//
// Calculate one step of integr8 by DBDF algorithm for a nonlinear
// system with adjustable order 1 .. ordset
//
// The computational coefficient matrices and variables are imported as globals
// ( b9,q9,qbq9,qblq9,q19,gama,ordset,ordmax,ord,epsi,epmin,tmax,num,t0,x0,xg0,
// h0,hi...) to reduce in/out arguments.
//
[n,m]=SIZE(xxo);
[nz,mz]=SIZE(zzo2);
xi=zzo2(1,1:1);
ti=tto(m);
til=ti+hi;
zi=zzo2;
e=eeo(m);
// r0=him1/hi
r0=1;
// if iter8 diverge-->r4=1/8, hi=r4*hi
r4=1;
// for some ODE, pgxd needn't be recomputed for each step.(flag=0)
// for other ODE, pgxd DO need recomputed even though flag=1.
flag=0;
GLOB(m),
GLOB(r4),
GLOB(flag),
IF m=1, ...
[zil,zit]=NR(zi,til,ord); ...
ELSE ...
ord=nz-1, ...
him1=ti-tto(m-1); ...
r0=hi/him1; ...
IF r0<>1, ...
FOR i=1:ord+1, ...
FOR j=1:1, ...
zi(i,j)=(r0**(i-1))*zi(i,j); ...
END, ...
END, ...
[zil,zit]=NR(zi,til,ord); ...
...// if iter8 diverge-->r4=1/8, hi=r4*hi, redo it ...
IF r4<0.8, ...
hi=r4*hi; ...
til=ti+hi; ...
al=0; ...
FOR i=1:ord+1, ...
FOR j=1:1, ...
zi(i,j)=(r4**(i-1))*zi(i,j); ...
END, ...
END, ...
[zil,zit]=NR(zi,til,ord); ...
END, ...
END
//
// Error test , stepsize and order control.
//
etest=0;
// r1 : if keep ord unchanged, hi=r1*hi

```

```

r1=1;
// r2 : if reduce ord by 1, hil=r2*hi
r2=0;
// r3 : if increas ord by 1, hil=r3*hi
r3=0;
facto=1;
FOR i=1:ord, ...
    facto=facto*i; ...
END,
// error estimate
FOR i=1:l, ...
    et(i)=gama(ord)*facto*(zil(ord+1,i)-zi(ord+1,i))+epmin; ...
END,
etest=NORM(et,'INF');
[em,idm]=MAX(ABS(et));
een1=[eeo etest];
// if etest<epsinon, accept solution. determine ordn & hil for next step.
IF etest<=epsi, ...
    r1=(1/1.2)*(epsi/etest)**(1/(ord+1)); ...
    IF a1<ord+1, ...
        a1=a1+1; ...
        ordn=ord; ...
        hil=hi; ...
    ELSE ...
        IF ord>=2, ...
            dn=ABS(gama(ord-1)*facto*zil(ord+1,idm)); ...
            r2=(1/1.3)*(epsi/dn)**(1/ord); ...
        END, ...
        IF ord<=ordset-1, ...
            ddz=zil(ord+1,idm)-2*zzo2((ord+1),idm)+zzo1((ord+1),idm); ...
            dn=ABS(gama(ord+1)*facto*ddz); ...
            r3=(1/1.4)*(epsi/dn)**(1/(ord+2)); ...
        END, ...
        rr=[r1,r2,r3]; ...
        [rmax,id]=MAX(rr); ...
        r=rmax; ...
        IF r>1.1, ...
            IF r>5, ...
                r=5; ...
            END, ...
            hil=r*hi; ...
            a1=-1; ...
        ELSE ...
            IF r<0.9, ...
                hil=r*hi; ...
                a1=-1; ...
            ELSE ...
                hil=hi; ...
            END, ...
        END, ...
    IF id=2, ...
        ordn=ord-1; ...
        a1=-1; ...
        flag=1; ...
    ELSE ...
        IF id=3, ...
            ordn=ord+1; ...
            FOR i=1:l, ...
                zb(i)=(zil(ordn,i)-zi(ordn,i))/ordn; ...
                zbb(i)=(zzo2(ordn,i)-zzo1(ordn,i))/ordn; ...
            END, ...
        END, ...
    END, ...

```



```

        END, ...
        zil=[zil;CONJ(zb')]; ...
        zzo1=[zzo1;ZROW(1,1)]; ...
        zzo2=[zzo2;CONJ(zbb')]; ...
        al=-1; ...
        flag=1; ...
    ELSE ...
        ordn=ord; ...
        IF hil=hi; ...
            al=al+1; ...
        END, ...
    END, ...
END, ...
...// xxn=[xxo,zil(1)]; ...
zxn1=zzo2(1:ordn+1,:); ...
zxn2=zil(1:ordn+1,:); ...
ELSE ...
...// otherwise, reject above solution,determine new hi, ordn for currend...
...// step, recompute solution until satisfied. ...
...// nf : # of times solution are rejected. ...
nf=0; ...
r=r0; ...
WHILE etest>epsi, ...
    nf=nf+1; ...
    r1=1; ...
    r2=0; ...
    FOR i=1:l; ...
        xd(i)=zi(2,i)/hi; ...
    END, ...
...// if fail more than 3 times, reduce ord=1. ...
IF nf>=3, ...
    ordn=1; ...
    flag=1; ...
    IF nf>3, ...
        hi=0.5*hi; ...
    ELSE ...
        hi=(ABS(0.5*epsi/(MAX(xd)))*0.5); ...
    END, ...
    FOR i=1:l, ...
        zzo2(2,i)=hi*xd(i); ...
    END, ...
    zi=zzo2(1:2,:); ...
    til=ti+hi; ...
    [zil,zit]=NRXDXS(zi,til,ordn); ...
...// otherwise, determine hi=r*hi, ordn for current step ...
ELSE ...
    r1=(1/1.2)*(epsi/etest)**(1/(ord+1));...
    IF ord>=2, ...
        dn=ABS(gama(ord-1)*facto*zil(ord+1,idm)); ...
        r2=(1/1.3)*(epsi/dn)**(1/ord); ...
    END, ...
    rr=[r1,r2]; ...
    [rmax,id]=MAX(rr); ...
    r=rmax;...
    hi=r*hi; ...
    til=ti+hi; ...
    IF id=2, ...
        ordn=ord-1; ...
        flag=1; ...

```

```

ELSE ...
  ordn=ord;...
END, ...
FOR i=1:ord+1, ...
  FOR j=1:l, ...
    zi(i,j)=(r**(i-1))*zi(i,j); ...
  END, ...
END, ...
[zil,zit]=NR(zi,til,ordn); ...
...// if iter8 diverge-->r4=1/8, hi=r4*hi, redo it ...
IF r4<0.8, ...
  hi=r4*hi; ...
  til=ti+hi; ...
  FOR i=1:ord+1, ...
    FOR j=1:l, ...
      zi(i,j)=(r4**(i-1))*zi(i,j); ...
    END, ...
  END, ...
  [zil,zit]=NR(zi,til,ordn); ...
END, ...
facto=1; ...
FOR i=1:ordn, ...
  facto=facto*i; ...
END, ...
...// test estimat error again, until satisfied ...
FOR i=1:l, ...
  et(i)=gama(ordn)*facto*(zil(ordn+1,i)-zi(ordn+1,i))+epmin; ...
END, ...
etest=NORM(et,'INF'); ...
[em,idm]=MAX(ABS(et)); ...
eenl=[eeo etest]; ...
al=0; ...
END, ...
...// get hil for next step ...
hil=hi; ...
END
// get ordn for next step.
ord=ordn;
//xxn=[xi zil(1)];
zzn1=zzo2(1:ordn+1,:); ...
zzn2=zil(1:ordn+1,:); ...
een=eenl;
RETURN

```

```

//[zil,zit]=NR(zi,tii,ord)
//
//Do Newton_Raphson iteration to get xd.
//
// Determine coefficient matrices for current order
nl=(ord-1)*(ordmax+1)+1;
nh=nl+ord;
b=b9(nl:nh, 1:ord+1);
q=q9(nl:nh, 1:ord+1);
qbq=qbq9(nl:nh, 1:ord+1);
qblq=qblq9(nl:nh, 1:ord+1);
q1=q19(1:ord+1, ord);
beta=b(1,2);
//
//doing N_R iter8 to get xd(i+1).(included in vector zit)
//
z=zi;
u=EYE(1);
//IF m=1, ...
// [pgxd]=JXDS(z(2,:)/hi,tii); ...
// p=pgxd-beta*hi*u; ...
// IF NORM(p)<=1.0D-15, ...
// DISPLAY('warning: (Jaco-beta*hi*1) is almost singular. '), ...
// END, ...
// pit=CONJ((u/p)'); ...
// pgxdo=pgxd; ...
//END,
////////////////////////////////////
// (section deleted put at the end of function. It may be reused after
// further consideration.)
////////////////////////////////////
IF flag >=0, ...
  [pgxd]=JXDS(z(2,:)/hi,tii); ...
  p=pgxd-beta*hi*u; ...
  IF NORM(p)<=1.0D-15, ...
    DISPLAY('warning: (Jaco-beta*hi*1) is almost singular. '), ...
  END, ...
  pit=CONJ((u/p)'); ...
  FOR i=1:3, ...
    g1=ZROW(ord+1,1); ...
    g1(2,:)=CONJ(GXDS(z(2,:)/hi,tii)'); ...
    dz=(qblq*z-q*g1)*pit*u*hi; ...
    z=z+dz; ...
  END, ...
  IF MAX(MAX(ABS(dz)))>epsi, ...
    r4=0.125; ...
    DISPLAY('Iteration not converg, set hi=(1/8)*hi. '); ...
  END, ...
  pgxdo=pgxd; ...
END,
zit=z;
//
//determine x(i+1). (in zil)
//
zil=qbq*zit;
flag=0;
RETURN
////////////////////////////////////
//
////////////////////////////////////

```

```

//[x]=GXDS(xd,t)
//known SYSTEM x=g(x',t)
//
//      e=exp(1);
//x(1)= e**(18)*e**(-9*1000*xd)-e**2*e**(-1000*xd),
//      x(1)= e**(9)*e**(-9*1000*xd)-e**1*e**(-1000*xd);
//x(1)=e**1*(e**(-100*xd)-e**(-900*xd)),
//
x(1)=-1.001*xd(1)-0.001*xd(2),
x(2)=xd(1),
//
//x(1)=-2*xd(1);
//x(2)=-3*xd(1)-0.5*xd(2);
//
//x(1)=xd(1)/(xd(2)**2);
//x(2)=xd(2)**2;
//
//x=-0.01*xd;
//x=-0.01*xd+sin(t);
//x=-xd;
//x=-100*xd;
//x=-(1+t)*xd;
//x=-1/(1+t)*xd;
//x=-(1+t)*(xd-exp(-t));
//x=sqrt(1/t*(xd+1/(1+t)));
//x=sqrt(abs((xd-1/(1-t))/(t-10)));
//x=-1/((1+t)**3*xd);
//x=sqrt(-xd);
//x=sin(2*t)/(2*xd);

```

```

//[pg]=JXDS(xd,t)
//partial derivative of x=g(x',t) respect to x'
//
//      e=exp(1);
//pg= 1000*e**2*e**(-1000*xd)-9000*e**(18)*e**(-9000*xd),
//      pg= 1000*e*e**(-1000*xd)-9000*e**(9)*e**(-9000*xd);
//pg=100*e**1*(9*e**(-900*xd)-e**(-100*xd));
//
pg(1,1)=-1.001;
pg(1,2)=-0.001;
pg(2,1)=1;
pg(2,2)=0;
//
//pg(1,1)=-2;
//pg(1,2)=0;
//pg(2,1)=-3;
//pg(2,2)=-0.5;
//
//pg(1,1)=1/(xd(2)**2);
//pg(1,2)=-2*xd(1)/(xd(2)**3);
//pg(2,1)=0;
//pg(2,2)=2*xd(2);
//
//      pg=-0.01;
//pg=-1;
//pg=-100;
//pg=-1/(1+t);
//pg=-(1+t);
//pg=.5/sqrt(t*(xd+1/(1+t)));
//pg=.5/sqrt(abs((t-10)*(xd-1/(1-t))));
//pg=1/((1+t)**3*xd**2);
//pg=-0.5/sqrt(-xd);
//pg=-sin(2*t)/(2*xd**2);
RETURN

```

REFERENCES

- Atkinson K.E., "An Introduction to Numerical Analysis", Jone Wiley and Sons, 1978.
- Cellier F.E., "Continuous System Modeling", Springer-Verlag, New York, 1991.
- Cellier F.E., "Simulation of Marginally Stable Circuits Using BBSPICE", Research Proposal, Funded by Burr-Brown Corporation, 1988.
- Cohen E., "Program Reference for SPICE-2", June 1976.
- Dahlquist G. and Björck Å., "Numerical Methods", Prentice-Hall, Englewood Cliffs, N. J., 1974.
- Gear C.W., "Numerical Initial Value Problems in Ordinary Differential Equations", Prentice-Hall, Englewood Cliffs, New Jersey, 1971a.
- Gear C.W., "The Automatic Integration of Ordinary Differential Equations", Comm. ACM, 14(3), 176-179, 1971b.
- Gear C.W., "The numerical Integration of Ordinary Differential Equations", Math. Comp., 21, 146-156, 1967
- Henrici P., "Discrete Variable Methods in Ordinary Differential Equations", John Wiley & Sons, Inc., New York, 1962
- Lambert J.D., "Computational Methods in Ordinary Differential Equations", Wiley, New York, 1973.
- Nagel L.W. "SPICE-2: A Computer Program to Simulate Semiconductor Circuits", May 1975.
- Skeel R.D. and Kong A.K., "Blended Linear Multistep Methods", ACM Trans. Math. Software, 3, 326-345, 1977.
- Shampine L.F. and Gordon M.K., "Computer Solution of Ordinary Differential Equations", W.H.Freeman & Co., San Francisco, 1975.