Stable High Order Methods for Circuit Simulation

by

© Yinghong Zhou

B.Eng., M.A.Sc.

A thesis submitted to the

Faculty of Graduate Studies and Research

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in Electrical Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

Department of Electronics

Faculty of Engineering

Carleton University

Ottawa, Ontario, Canada

May 2011

Your file Votre référence
ISBN: 978-0-494-81551-9
Our file Notre référence
ISBN: 978-0-494-81551-9

The undersigned recommend to
the Faculty of Graduate and Postdoctoral Affairs
acceptance of the thesis

Stable High Order Methods for Circuit Simulation

Submitted by
Yinghong Zhou, B.Eng., M. A. Sc.

In partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in Electrical Engineering

Professor Ram Achar, Co-Supervisor

Professor Michel Nakhla, Co-Supervisor

Professor Emad Gad, Co-Supervisor

Professor Greg Bridges, External Examiner

Professor Qi-Jun Zhang
Chair, Department of Electronics

Ottawa-Carleton Institute for Electrical and Computer Engineering
Carleton University
Department of Electronics
2011

Abstract

This thesis describes a new A-stable and L-stable integration method for simulating the time-domain transient response of stiff nonlinear circuits. The proposed method is based on the Obreshkov formula. It utilizes high-order derivatives at single step and can be made of arbitrary high order while maintaining the A-stability property. The new method allows for adoption of higher-order integration methods for transient analysis of electronic circuits while enabling them to take larger step sizes without violating stability, leading to faster simulations. Furthermore, the method can be run in an L-stable mode to handle circuits with extremely stiff equations. Necessary theoretical foundations, implementation details, error control mechanisms and validating computational results are presented.

Dedication

To my parents, my husband and my daughter

Acknowledgements

First, I thank Professor Michel Nakhla, whose guidance, advice, and knowledge were instrumental in the evolution of my research. I also thank Professor Emad Gad for his valuable suggestions, theoretical insights that form a large part of this thesis, tremendous involvements and encouragements. I appreciate the help and support from Professor Ram Achar. Their valuable suggestions guided me throughout the thesis. Without their support, this thesis would not have been possible.

I would also like to express my gratitude to my Master's thesis supervisor, Professor Ajoy Opal for his kindness and initialization of this research project. I thank Professor Rémi Vaillancourt, who spent time to proofread my thesis and pointed a few errors.

I am also grateful for both the contributions and friendship of my colleagues. In particular, I thank Dr. Changzhong Chen for his useful discussions and active participating in this research project. I am also indebted to Douglas Paul for the discussions and help on programming languages. I thank Ashok Narayanan, Dragan Trifkovic and Mina Adel for their help to proofread my thesis.

I give special thanks to my family, my husband Xiaoqiang Ma, my daughter Laura Ma, my parents Tijia Zhou and Youmei Luo, and my sister Yingxue Zhou for their endless love and understanding. Their encouragements helped me go through all the difficult times.

Finally, it is a pleasure to thank all my friends who always support me in all aspects of life. Thank you very much my friends.

Contents

List of Tables

List of Figures

List of Abbreviations

NR	Newton–Raphson	2
DE	Differential Equation	6
LMS	Linear Multi-Step	11
IVP	Initial Value Problem	10
ODE	Ordinary Differential Equation	12
LTE	Local Truncation Error	13
FE	Forward Euler	21
BE	Backward Euler	22
TR	Trapezoidal Rule	23
BDF	Backward Differentiation Formula	23
RK	Runge–Kutta methods	31
DIRK	Diagonally Implicit Runge–Kutta methods	37
GLM	General Linear Methods	37
DAE	Differential Algebraic Equation	50
MNA	Modified Nodal Analysis	67
AD	Automatic Differentiation	72
LU	Lower/Upper triangular matrix decomposition	79

FB	Forward/Backward substitution	79
KLU	A high-performance sparse linear system solver	86
UMFPACK	Unsymmetric multifrontal sparse LU factorization package	98
HSPICE®	A Circuit Simulator from Synopsys Inc.	102
OpAmp	Operational Amplifier	112

List of Symbols

$\mathbb{R}$, $\mathbb{C}$	Real, complex variable space
$\mathcal{R}(q)$	Amplification function to test stability
$\boldsymbol{x}(t)$	Unknown vector in the system of ordinary differential equations
$\boldsymbol{d}_n$	Local truncation error at time $t = t_n$
C	Local truncation error constant
$\boldsymbol{l}_n$	Local error at time $t = t_n$
$\alpha_{i,m}$	Integration coefficients in Obreshkov formula
α_i, β_i	Integration coefficients in modified Obreshkov formula
$\boldsymbol{M}(q)$	Stability matrix of general linear methods
$\Phi(z, q)$	Stability function of general linear methods and generalized Obreshkov formula
$\mathcal{R}_{(l,m)}(q)$	$[l/m]$ Padé approximant to the exponential function
$N_{lm}(q)$	Numerator of $\mathcal{R}_{(l,m)}(q)$
$N_{ml}(-q)$	Denominator of $\mathcal{R}_{(l,m)}(q)$
$C_{(l,m)}$	Local truncation error constant of modified Obreshkov formula
$\boldsymbol{x}_n$	Approximation of $\boldsymbol{x}(t)$ at $t = t_n$
h	Current step size

h_n	Step size between t_{n-1} and t_n
$\epsilon_{\text{tol}},\ \epsilon_{\text{tol1}},\ \epsilon_{\text{tol2}}$	User defined error tolerances
$\boldsymbol{z}_{n+1}$	Stage vector composed of present/past values/derivatives scaled by h
$\boldsymbol{y}_{n+1}$	Nordsieck vector composed of present/past values/derivatives scaled by h
$\boldsymbol{G},\ \boldsymbol{C}$	MNA matrices describing a circuit
$\boldsymbol{f}(\boldsymbol{x}(t))$	A vector containing nonlinear functions in MNA formulation
$\boldsymbol{b}(t)$	A vector containing independent stimuli in MNA formulation
$\tilde{\boldsymbol{G}},\ \tilde{\boldsymbol{C}},\ \tilde{\boldsymbol{f}},\ \tilde{\boldsymbol{b}},\ \boldsymbol{\xi}$	Matrices and vectors formulated by the modified Obreshkov formula when it is applied to circuit equations
$\tilde{\boldsymbol{J}},\ \hat{\boldsymbol{J}}$	Block Jacobian matrices
$\boldsymbol{J}_{i,j}$	$(i,j)^{\text{th}}$ block entry of matrix $\tilde{\boldsymbol{J}}$
$\boldsymbol{R}_u$	Matrix-valued u^{th} Taylor series coefficient of $\boldsymbol{J}_{0,0}$

Chapter 1

Introduction

1.1 Motivation

Circuit simulation plays a very important role in circuit analysis and design. A simulator is used to predict the behavior of electrical circuits without the need for physically constructing the circuits. Circuit level simulators are used to model the behavior of a circuit described in terms of transistors, capacitors, resistors and their interconnections. The simulator computes detailed analog waveforms (voltages and currents), which accurately model the behavior of the circuits to a given set of inputs. The results of analysis are either tabulated as a sequence of numbers, or plotted as waveforms.

One of the important time domain simulation methods is transient analysis. The first step in transient analysis is the formulation of a coupled set of nonlinear first-order differential equations representing the behavior of the circuit. The next step is to replace the time derivatives in the differential equations by finite difference approx-

imations (known as integration formulas) at discrete time instants. This step transforms the nonlinear ordinary differential equations, at each discretized time point, into a set of nonlinear algebraic equations.

The third step is to solve the nonlinear algebraic equations by Newton–Raphson (NR) method which approximates them with a linear set of equations based on an initial estimate.

Having arrived at a satisfactory solution at the time point, a trial time increment to advance to a new point is selected, and a prediction of the solution at the new point is made. This prediction is used as the initial estimate of the solution for Newton–Raphson method at the new time point. Thus simulation proceeds as a march-in-time through a sequence of discretized time points selected to achieve both convergence of the Newton iteration process and adequate accuracy of simulation.

Designers of electronic circuits typically expect transient analysis simulators to be of superior accuracy while requiring shorter execution time and limited computational resources. Typically, these requirements are often in conflict with each other for most circuit applications. The primary cause of such a conflict arises while solving underlying differential equations and is usually due to the relation between the order and stability of the integration methods being used.

1.2 Properties of integration methods

The order of the method reflects the computational effort needed to achieve a better accuracy. However, high-accuracy integration using a small-order method necessitates a significant reduction in the step size, causing solutions to be performed at a larger

number of time points. Another issue is the evaluation of nonlinear devices (and their derivatives) at each time step, with modern devices requiring more CPU time to evaluate. Typically the cost associated with the model evaluation dominates the computational efforts spent at each time step. Both issues often cause transient analysis time to run excessively longer. Hence, high-order methods which naturally lead to a reduction in the number of time steps would, therefore, be crucial to efficient simulations.

Nevertheless, numerical stability is a decisive factor in the choice of integration methods, since a lack of stability can cause the solution to grow without bounds over time, rendering the results of no use. In general, it is desirable that the integration method is guaranteed to be stable for all types of stable circuits (i.e., for all circuits with poles over the left-half plane of the complex domain). This concept is known in the mathematical literature as A-stability, and an integration method exhibiting this feature is called A-stable. Also, obtaining stiff decay at $q = \infty$ (methods possessing this characteristic, in addition to be A-stale, are called L-stable [1] methods) is a crucial requirement for handling very stiff circuits that are described by differential-algebraic equations (such as those arising in using the modified nodal admittance formulation [2]). Therefore, an ideal integration method should be A-stable (or L-stable) with high order, and should take minimal computational efforts.

1.3 Traditional methods

One of the major classes of integration methods that has been very popular in circuit simulation is the linear multistep (LMS). The known forward Euler (FE), backward

Euler (BE), trapezoidal rule (TR) and backward differentiation formula (BDF) belong to this class. Ideally, one hopes to find A-stable methods of significantly high order within the suite of LMS methods. Unfortunately, as has been established by the second Dahlquist barrier [3], the highest achievable order in this class of methods cannot exceed order 2, i.e., the trapezoidal rule is the A-stable method with highest order. As a result of this barrier, the quest for integration methods with better accuracy shifted to relax the A-stability requirement by considering the so-called $A(\alpha)$-stable methods, which are stable over limited portions of the complex domain (including $q = \infty$), but with stability domains increasingly shrinking for progressively increasing high order. The prime example of these approaches is BDF, or Gear's method [4], which has gained significant prominence due to its efficiency in handling very stiff circuits [5]. BDF uses past points to advance to future points and can achieve orders 1 to 6. However, there are several drawbacks to the notion of sacrificing stability in return for high-order integration methods, which represents the basic philosophy of BDF methods.

1. The theoretical stability regions for the high-order BDF methods, although smaller than the ideal A-stability, can only be maintained if one maintains a constant step size in marching through time. In fact, it was shown that these stability regions shrink even further if the ratio between two successive steps exceeds a certain threshold [6]. This is a serious disadvantage, noting that flexible step size adjustment is the key to follow the circuit waveforms efficiently.

2. $A(\alpha)$-stability methods are only sufficient for numerical integration of stiff differential systems in the absence of oscillating components. If the circuit has

highly oscillatory behavior with poles very close to the imaginary axis, the step size has to be reduced dramatically to guarantee a stable response.

3. In addition, when a variable step size implementation is used, the coefficients of the formula have to be updated to maintain the accuracy. Polynomial interpolation of the past points becomes necessary to estimate the local truncation error and to predict a new step size [7], [8]. However, polynomial interpolation always requires a certain degree of regularity in the waveforms being approximated, which may not be adequately satisfied if the nonlinear models exhibit strong nonlinear behavior.

The above reasons have prompted several mainstream circuit simulators to discourage the user from invoking methods of order higher than 2 in simulating highly nonlinear circuits. However, the available most accurate second-order A-stable LMS method, i.e., the trapezoidal rule, is not L-stable. This fact means that TR does not damp modes with large real parts [9], [1]. Backward Euler and second order BDF are L-stable methods. Unfortunately, they are not as accurate as trapezoidal rule.

It is important to note at this point, that A-stable (and also L-stable) methods with orders higher than 2 exist, but in forms different from the LMS, such as single-step multi-stage Runge–Kutta methods (RK) [9], [1]. However, for these methods, it is quite challenging to derive high-order formulas, in addition to the difficulties that arise from the computational aspects of their implementations.

1.4 Thesis objective

The main objective of this thesis is to develop a new integration method that addresses all of the above difficulties. In particular, the proposed algorithm will offer the following key advantages.

1. The new method is free from theoretical barriers that previously prevented the incorporation of high-order integrations, which are more accurate than low order methods for the solution of differential equations (DE).

2. Contrary to the existing high-order A-stable algorithms, the algorithms proposed here are readily available with *a priori* known coefficients for arbitrary high order with no need for derivation (chapter 2 briefly describes the existing methods such as [10], to highlight the main differences between them and the proposed algorithms).

3. The new method allows for efficient adoption of higher-order integration methods for transient analysis of electronic circuits while allowing them to take larger step sizes without violating stability. This leads to a smaller number of time point solutions for waveform evaluations and hence faster time-domain simulations.

The new method is based on a modified version of the Obreshkov formula which utilizes implicit high-order derivatives. Necessary theoretical foundations, implementation details, time-step error control mechanisms and validating computational results are presented.

1.5 Thesis contributions

This thesis proposes a modified version of Obreshkov formula. The properties of the modified Obreshkov formula, its application to circuit simulations and a step size control mechanism are presented. There are four major contributions in my thesis [11], [12], [13].

1. A class of integration methods are proposed based on the modified version of Obreshkov formula. The order of the modified Obreshkov formula can be arbitrarily high. Under certain conditions, the modified Obreshkov formula is A-stable and L-stable [11], [12].

2. A high-order circuit simulator is implemented based on the modified Obreshkov formula. In order to evaluate the high-order derivatives and partial derivatives of the nonlinear functions, tree-like structures are constructed and linear recursive relationships are formulated [11], [12].

3. A few specially tailored techniques are used to enable the high-order circuit simulator to reach its full potential. The implementations are improved in the following ways [13].

   (a) The evaluation time of the partial derivatives is reduced greatly by converting the nonlinear portion of Jacobian matrix from a two dimensional array into one dimensional array. In addition to the reduction of evaluation time of the nonlinear partial derivatives, the resulting Jacobian matrix enjoys a simple block Hessenberg matrix structure with repeated entries in super

diagonal and the main diagonal. This fact greatly facilitates the numerical LU factorization.

(b) A block version of KLU, which factorizes the high order matrices based on blocks instead of scalars, is implemented to efficiently factorize the high-order Jacobian matrices. Enjoying the almost symmetric sparse pattern and a almost zero-free diagonal, which are usually possessed by the circuit matrices, the block version of KLU reaches its best performance.

(c) The large step size afforded by the high-order methods typically requires more number of Newton iterations to obtain a converged solution. The computational burden is partially relieved by supplying a better initial guess, which is generated by a low-order method with a less accuracy requirement, to start the high-order Newton iterations.

4. To simulate the circuits efficiently, an adaptive variable step size control algorithm is implemented by using the Nordsieck method. During the fast changing phase, a small step size is used to capture the fine level of the response. When the waveform is smooth, a large step size is adopted to save computational cost [11].

1.6 Thesis organization

This thesis is organized into nine chapters. Following this chapter, chapter 2 reviews various integration methods for solving differential equations. Chapter 3 introduces the theoretical foundation of the proposed algorithms and their order, local truncation

error and stability issues. In chapter 4, the error estimation method and the step size control mechanism are discussed in detail. Chapter 5 describes implementation details on linear and nonlinear circuits. In chapter 6, the structural characteristics of the Jacobian matrix are discussed and a few improved implementation techniques are presented. Chapter 7 shows a few example circuits and their computational results. Chapter 8 presents more numerical examples to demonstrate the advantages introduced by the improved techniques proposed in chapter 6. Chapter 9 gives a brief summary of contributions and possible future work.

Chapter 2

Review of integration methods

This chapter presents a general background for various integration methods. It will also serve as a platform for highlighting the main theoretical aspects used in the proposed algorithm.

In general, the initial value problem (IVP) to be solved is written in the following form

$$\boldsymbol{x}' = \boldsymbol{f}(t, \boldsymbol{x}), \quad 0 \leq t \leq b, \tag{2.1}$$

with $\boldsymbol{x}(0) = \boldsymbol{x}_0$ given. For IVPs, one starts at the initial point with all the solution information and marches forward in time. We will assume $\boldsymbol{f}(t, \boldsymbol{x})$ be continuous for all $(t, \boldsymbol{x})$ in a region $\mathcal{D} = \{0 \leq t \leq b, \ |\boldsymbol{x}| < \infty\}$ and Lipschitz continuity in $\boldsymbol{x}$, i.e., there exists a constant L such that for all $(t, \boldsymbol{x})$ and $(t, \hat{\boldsymbol{x}})$ in $\mathcal{D}$,

$$|\boldsymbol{f}(t, \boldsymbol{x}) - \boldsymbol{f}(t, \hat{\boldsymbol{x}})| \leq L|\boldsymbol{x} - \hat{\boldsymbol{x}}|. \tag{2.2}$$

Then

1. For any $\boldsymbol{x}_0$, there exists a unique solution $\boldsymbol{x}(t)$ throughout the interval $[0, b]$ for the IVP (2.1). This solution is differentiable.

2. The solution $\boldsymbol{x}$ depends continuously on the initial data: if $\hat{\boldsymbol{x}}$ also satisfies the ODE (but not the same initial values) then

$$|\boldsymbol{x}(t) - \hat{\boldsymbol{x}}(t)| \leq e^{Lt}|\boldsymbol{x}0 - \hat{\boldsymbol{x}}(0)|. \tag{2.3}$$

3. If $\hat{\boldsymbol{x}}$ satisfies, more generally, a perturbed ODE

$$\hat{\boldsymbol{x}}' = \boldsymbol{f}(t, \hat{\boldsymbol{x}}) + \boldsymbol{r}(t, \hat{\boldsymbol{x}}),$$

where $\boldsymbol{r}$ is bounded on $\mathcal{D}$, $||\boldsymbol{r}|| \leq M$, then

$$|\boldsymbol{x}(t) - \hat{\boldsymbol{x}}(t)| \leq e^{Lt}|\boldsymbol{x}(0) - \hat{\boldsymbol{x}}(0)| + \frac{M}{L}\left(e^{Lt} - 1\right). \tag{2.4}$$

If $\boldsymbol{f}$ is differentiable, then the constant L can be taken as a bound on the first derivatives of $\boldsymbol{f}$ with respect to $\boldsymbol{x}$, i.e.,

$$L = \sup_{(t,\boldsymbol{x})\in\mathcal{D}} \left|\left| \frac{\partial \boldsymbol{f}(t, \boldsymbol{x})}{\partial \boldsymbol{x}} \right|\right|. \tag{2.5}$$

Thus, subject to both continuity and Lipschitz continuity of $\boldsymbol{f}$ in $\mathcal{D}$, (2.1) has a unique solution in $0 \leq t \leq b$ and is well-posed [1].

We will denote by $\boldsymbol{x}_n$ the approximation generated by an integration method to $\boldsymbol{x}(t)$ at time $t = t_n$ with $\boldsymbol{x}(t)\in \mathbb{R}^N$ being the vector of unknown variables and N the number of the variables. As suggested by Burrage [14], available integration methods for the solution of differential equations can be viewed on a two-dimensional plane as shown in Figure 2.1. On this plane, two main approaches can be seen to lie at opposite ends of the spectrum: the single-step multistage methods and the linear multistep (LMS) methods. Further approaches that combine aspects from these two approaches may be represented by points on that plane. The order of a method is indicated by the number between brackets.

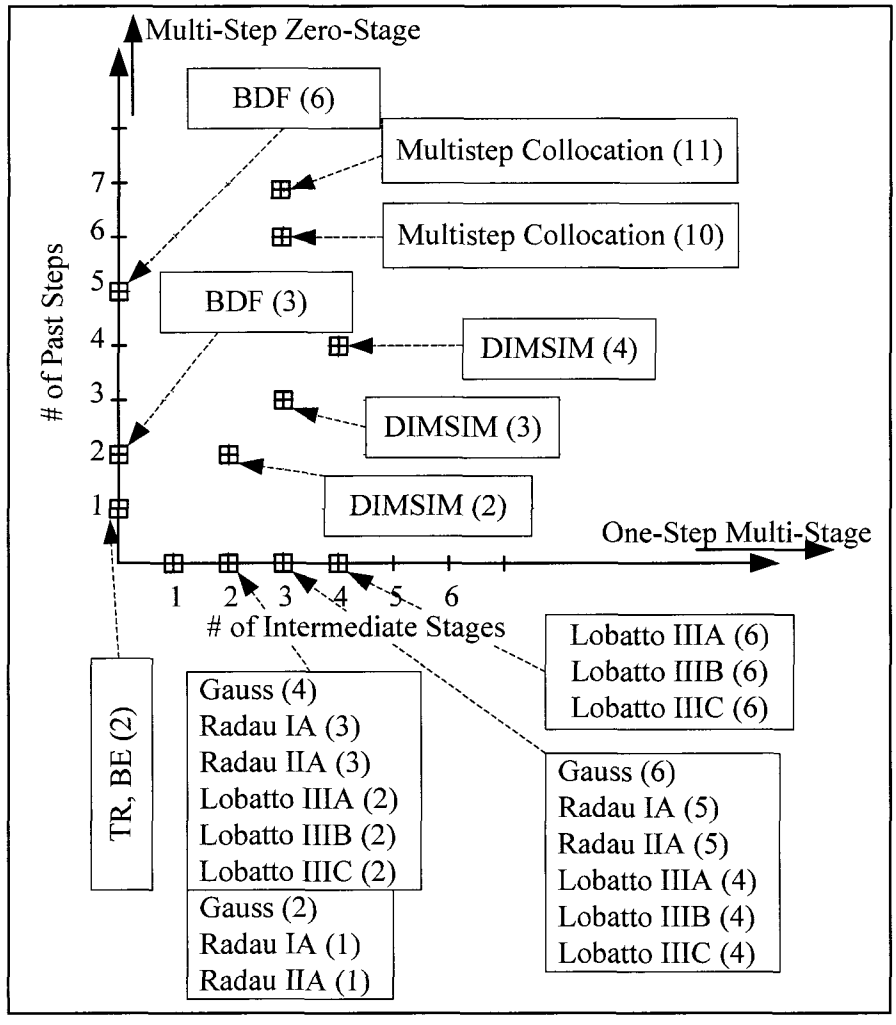

Figure 2.1: Representing integration methods on a plane as points that combine the number of stages and the number of previous step points. The numbers between brackets indicate the order of the methods.

2.1 Order and local error

Much of the study of numerical methods to solve ordinary differential equations (ODE) is concerned with the errors at each step that are due to the difference approximation, and how they accumulate over time.

A method is said to be of order p if

$$\boldsymbol{x}_n = \boldsymbol{x}(t_n) + Ch_n^{p+1}\frac{d^{p+1}}{dt^{p+1}}\boldsymbol{x}(t)\bigg|_{t=t_n} + O(h_n^{p+2}), \tag{2.6}$$

where $h_n = t_n - t_{n-1}$. C is called the error constant of the method. The magnitude of C indicates the accuracy of the results. $O(h^v)$ is an error term proportional to h^v. Typically, the error of a given integration method is defined using the so-called Local Truncation Error (LTE). LTE is defined as the $(p+1)^{\text{th}}$ term in (2.6), i.e.,

$$\boldsymbol{d}_n = Ch_n^{p+1}\frac{d^{p+1}}{dt^{p+1}}\boldsymbol{x}(t)\bigg|_{t=t_n}. \tag{2.7}$$

If a difference method starts from the exact solution and evolves only one step, the error between the exact solution and the approximation is LTE. Another important measure of the error made at each step, the local error, is defined as the amount by which the numerical solution $\boldsymbol{x}_n$ at each step differs from the exact solution $\bar{\boldsymbol{x}}(t_n)$ to the initial value problem

$$\begin{aligned} \bar{\boldsymbol{x}}'(t) &= \boldsymbol{f}(t, \bar{\boldsymbol{x}}(t)), \\ \bar{\boldsymbol{x}}(t_{n-1}) &= \boldsymbol{x}_{n-1}. \end{aligned} \tag{2.8}$$

Thus the local error is given by

$$\boldsymbol{l_n} = \bar{\boldsymbol{x}}(t_n) - \boldsymbol{x}_n. \tag{2.9}$$

It is shown in [1] that, for all the numerical ODE methods considered in our scope,

$$|\boldsymbol{d_n}| = |\boldsymbol{l_n}|(1 + O(h_n)). \tag{2.10}$$

The two local error indicators are thus often closely related. However, for stiff problems, the constant implied in this $O(h_n)$ may be quite large.

2.2 Stability

Stability is another crucial criterion in the choice of an integration method. The stability of a given integration method is usually studied through characterizing its behavior in approximating the scalar test equation [1]

$$x' = \lambda x, \tag{2.11}$$
$$x(0) = x_0,$$

where λ is a complex constant and $\Re e(\lambda) < 0$. Usually, λ represents an eigenvalue of a system's matrix. The exact solution for the scalar test equation is

$$x(t_n) = x_0 e^{\lambda t_n},$$

which is an exponentially decaying solution if $\Re e(\lambda) < 0$. Naturally, one would expects that the sequence of approximations to the scalar test equation, x_0, x_1, x_2, ..., x_n, should satisfy

$$|x_n| \leq |x_{n-1}|, \quad n = 1, 2, \ldots , \tag{2.12}$$

which is known as the absolute stability requirement.

For a given numerical method, the region of absolute stability is a region in the complex q-plane. When applying the method to the test equation (2.11), the $q = h\lambda$ within this region yields an approximate solution satisfying the absolute stability requirement (2.12). This means to approximate the naturally stable system (2.11), the step size h of an integration method must be selected to make $h\lambda$ lie on or inside the region of absolute stability.

For example, Figure 2.2 shows the region of absolute stability for the forward Euler method, which is a unit circle with center at (-1, 0). Thus if the step size h

used by the forward Euler method is such that $h\lambda$ lies inside the unit circle of Figure 2.2, then the integration is stable and the sequence of approximations satisfies (2.12).

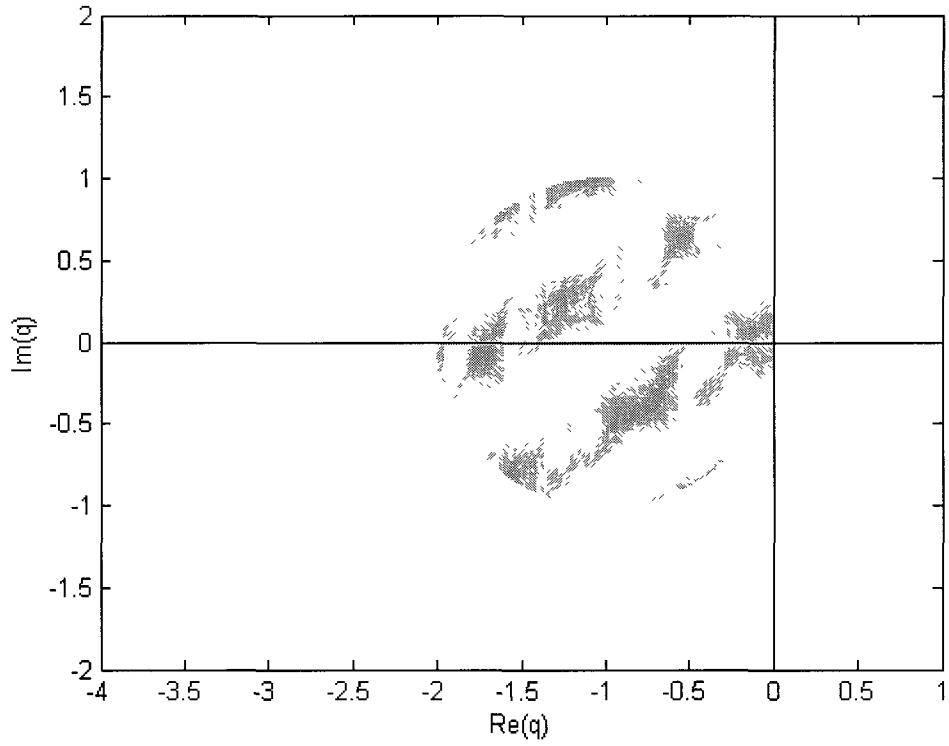

Figure 2.2: Absolute stability region for the forward Euler (shaded area).

Ideally, the choice of the step size h should be dictated by approximation accuracy requirements, not by the stability region. Loosely speaking, a system is referred as being stiff if the absolute stability requirements of some numerical methods dictate a much smaller step size than is needed to satisfy approximation requirements alone. Problems are characterized as stiff if the derivative depends strongly on the solution. That is, for the test equation (2.11), $|\lambda|$ is very large. Generalizing to multiple

dimensions, a system of linear equations of the form

$$\boldsymbol{x}' = \boldsymbol{J}\boldsymbol{x}$$

is stiff if the square matrix $\boldsymbol{J}$ has at least one eigenvalue, λ, for which $|\lambda|$ is very large. A nonlinear set of differential equations

$$\boldsymbol{x}' = \boldsymbol{f}(t, \boldsymbol{x})$$

is stiff if the Jacobian matrix

$$\boldsymbol{J}(\boldsymbol{x}) = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}$$

of $\boldsymbol{f}$ has at least one eigenvalue, λ, for which $|\lambda|$ is very large [6].

Usually, a stiff system has some eigenvalues close to the origin of coordinates and some eigenvalues very far away from them, all in the left half plane. In time domain simulations, the response of the system is mainly captured by the eigenvalues close to the origin. Large step size is sufficient because the solution corresponding to the these eigenvalues is relatively smooth. On the other hand, the solution components corresponding to the eigenvalues with large real parts generally decay rapidly [2]. With the forward Euler method, one has to adopt a very small step size h to maintain the stability of integration when some eigenvalues are far from the origin. This idea leads us to the concept of A-stability [3] defined next.

2.2.1 *A*-stability

An integration method is said to be A-stable if its region of absolute stability contains the entire left half-plane of $q = h\lambda$, i.e., an integration formula is said to be A-stable if it leads to a bounded solution of the test differential equation (2.11) for any step

size for all $\Re e(\lambda) < 0$. For example, both the backward Euler and the trapezoidal rule are A-stable since their regions of absolute stability cover the entire left half plane of the q-plane, as shown in Figures 2.3 and 2.4.

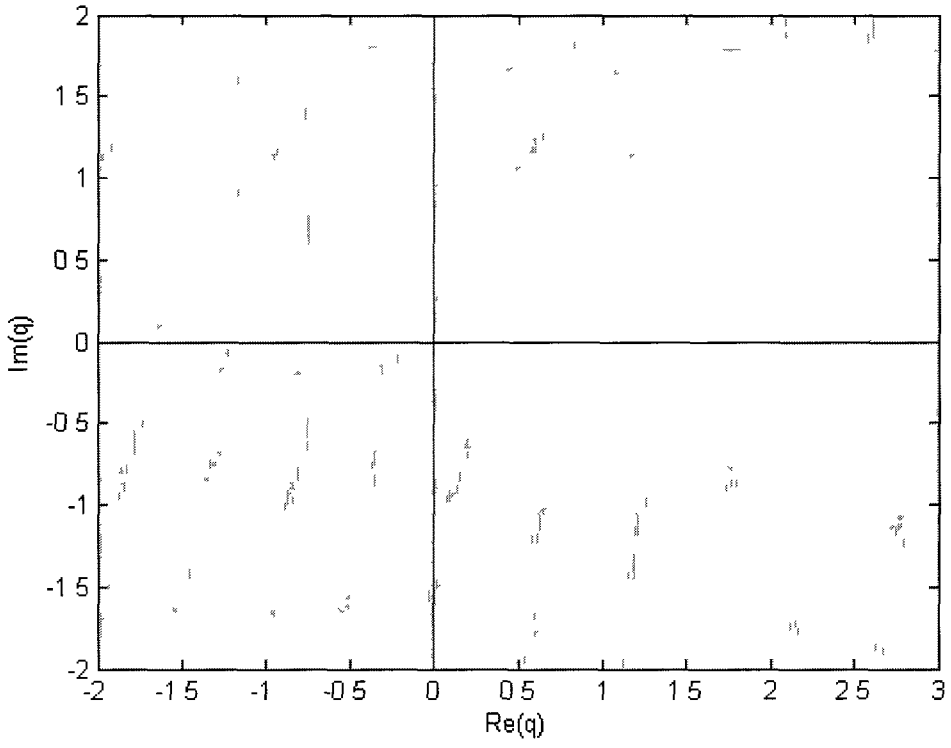

Figure 2.3: Absolute stability region for the backward Euler (shaded area).

2.2.2 *L*-stability

A further investigation into A-stability reveals two deficiencies. The first is that it does not distinguish between the cases

$$\Re e(\lambda) \to -\infty$$

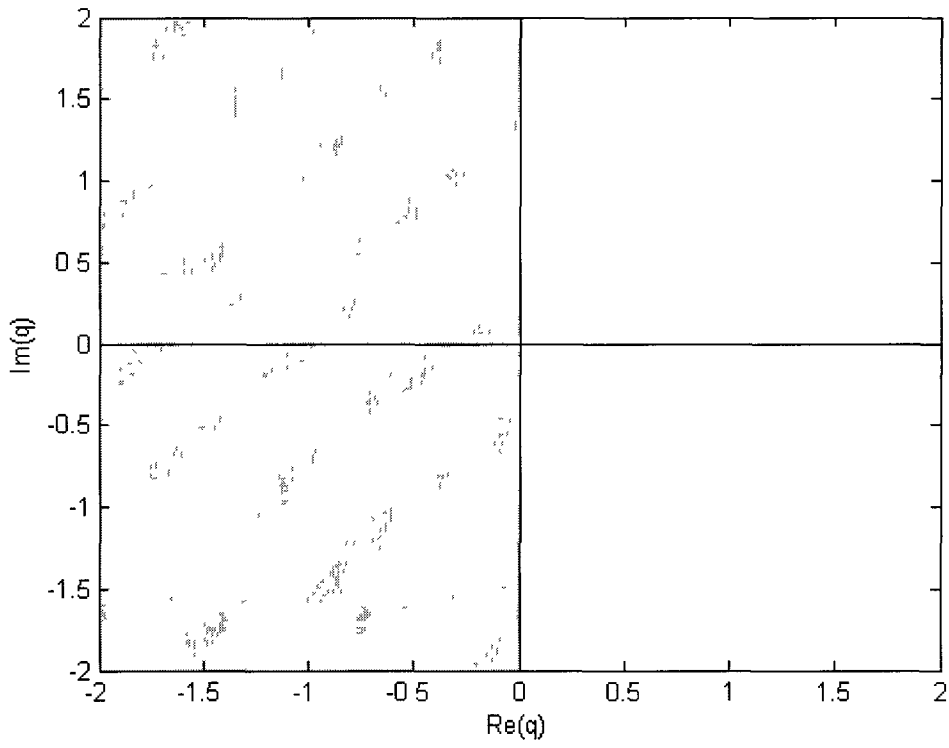

Figure 2.4: Absolute stability region for the trapezoidal rule (shaded area).

and

$$-1 \ll \Re e(\lambda) \leq 0, \quad |\Im m(\lambda)| \to \infty.$$

The latter case gives rise to a highly oscillatory exact solution, which does not decay much.

The second possible weakness of the A-stability definition arises from its exclusive use of absolute stability. In the very stiff limit, $h_n \Re e(\lambda) \to -\infty$, the exact solution of the test equation satisfies $|x(t_n)| = |x(t_{n-1})| e^{h_n \Re e(\lambda)} \ll |x(t_{n-1})|$. The corresponding absolute stability requirement, $|x_n| \leq |x_{n-1}|$, seems insufficient in comparison, since it does not exclude $|x_n| \approx |x_{n-1}|$. Here is where the stiff decay comes into the picture.

An integration method is said to have stiff decay if

$$\frac{x_{n+1}}{x_n} \to 0 \quad \text{as} \quad h\lambda \to -\infty.$$

The practical advantage of methods with stiff decay lies in their ability to skip fine-level (i.e., rapid varying) solution details and still maintain a decent description of the solution on a coarse level in the very stiff (not the highly oscillatory) case.

This leads us to the stronger concept of L-stability defined next. A method is said to be L-stable if it is A-stable with stiff decay [15], [1]. For example, the backward Euler is an L-stable method.

2.2.3 $A(\alpha)$-stability

Given the difficulty of constructing A-stable or L-stable methods with high order, the focus has been shifted to construct the so-called $A(\alpha)$-stable methods which are stable over limited portions of the left half plane of the q-domain.

A method is said to be $A(\alpha)$-stable if all the q in the wedge-shaped domain

$$u \leq 0 \text{ and } -\tan(\alpha)|u| \leq v \leq \tan(\alpha)|u|$$

is contained in the region of absolute stability [6], [9], where u and v are the real and imaginary parts of q, i.e.,

$$q = u + iv. \tag{2.13}$$

Of course, $A(\pi/2)$-stability is nothing else but A-stability, but decreasing $\alpha \geq 0$ allows for relaxed stability requirements. These are amply sufficient for the numerical integration of stiff differential systems in the absence of oscillating components [16].

2.3 Integration methods

According to Figure 2.1, existing integration methods can be viewed on the two-dimensional plane, where the x-axis represents single-step multistage methods and the y-axis represents linear multi-step methods. This section provides a basic outline for the general structure of these methods.

2.3.1 Linear multistep (LMS) methods

The general form of a linear k-step method is represented by the following formula,

$$\sum_{i=0}^{k} \mu_i \boldsymbol{x}_{n+k-i} = h \sum_{i=0}^{k} \zeta_i \boldsymbol{x}^{(1)}_{n+k-i}, \tag{2.14}$$

where $\boldsymbol{x}^{(i)}_n$ represents an approximation to $\frac{d^i}{dt^i}\boldsymbol{x}(t)$ at $t = t_n$, with $\boldsymbol{x}^{(0)}_n \equiv \boldsymbol{x}_n$. The coefficients μ_i and ζ_i are specific to each integration method.

Figure 2.5 describes graphically the use of the previous values and derivatives at the past k points to advance to the next step at t_{n+k}.

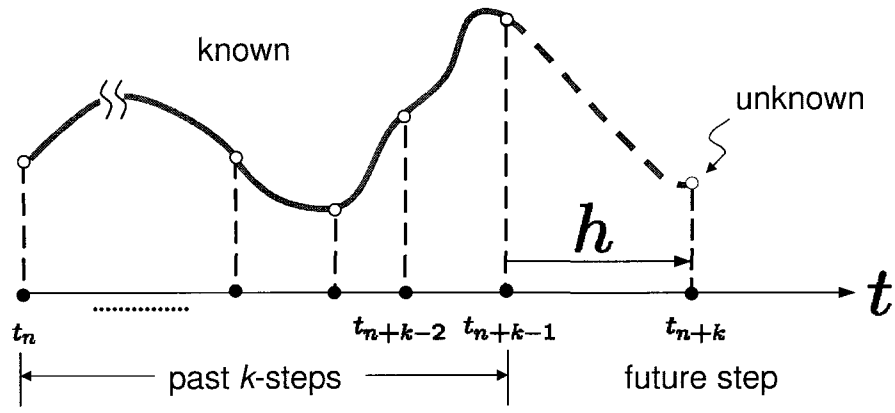

Figure 2.5: Time axis describing the use of the previous values and derivatives at the past k time points to advance linear multistep methods to the next time point, t_{n+k}.

In order to derive k-step LMS method with order p, one has to compute μ_i and ζ_i $(i = 0, \ldots, k)$ such that the first $p+1$ Taylor series coefficients in the operator

$$\boldsymbol{\mathcal{L}}(\boldsymbol{x}(t)) := \sum_{i=0}^{k} \mu_i \boldsymbol{x}(t_{n+k-i}) - h \sum_{i=0}^{k} \zeta_i \boldsymbol{x}'(t_{n+k-i}) \tag{2.15}$$

vanish identically. An LMS is said to be explicit if ζ_0 is set to 0, and implicit otherwise. An explicit method has the advantage that the unknown future point $(\boldsymbol{x}_{n+k})$ can be computed directly from the previous points and the derivatives of $\boldsymbol{x}$ at these past points. On the other hand, an implicit method $(\zeta_0 \neq 0)$ requires solving a system of nonlinear equations in order to obtain $\boldsymbol{x}_{n+k}$.

In the following, several examples of LMS methods are presented and their stability properties are described.

2.3.1.1 Single-step methods

Single-step methods are special cases of LMS methods obtained by setting $k = 1$.

Forward Euler method (FE)

Our first example of these methods is the forward Euler method obtained by setting $\zeta_0 = 0$, where (2.14) becomes

$$\mu_0 \boldsymbol{x}_{n+1} + \mu_1 \boldsymbol{x}_n = h\zeta_1 \boldsymbol{x}'_n. \tag{2.16}$$

To have a method of order 1 $(p = 1)$, we substitute $\boldsymbol{x}_{n+1}$, $\boldsymbol{x}_n$ and $\boldsymbol{x}'_n$ by $\boldsymbol{x}(t_{n+1})$, $\boldsymbol{x}(t_n)$ and $\frac{d\boldsymbol{x}}{dt}\big|_{t=t_n}$ in (2.16), expand each term in its Taylor series, and equate the terms up to the p^{th} derivative to zero. This results in the following method

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + h\boldsymbol{x}'_n, \tag{2.17}$$

and the values of μ_i and ζ_i are

$$\mu_0 = 1, \quad \mu_1 = -1 \quad \text{and} \quad \zeta_1 = 1.$$

The error constant is $C = 1/2$. which is the coefficient of the $(p+1)^{\text{th}}$ derivative that does not vanish in the Taylor expansion.

Since $\zeta_0 = 0$, the forward Euler method is an explicit method that does not require the expensive step of obtaining $\boldsymbol{x}$ of a system of nonlinear equations.

Backward Euler method (BE)

Backward Euler is obtained similarly by setting $\zeta_1 = 0$, where (2.14) becomes

$$\mu_0 \boldsymbol{x}_{n+1} + \mu_1 \boldsymbol{x}_n = h\zeta_0 \boldsymbol{x}'_{n+1}. \tag{2.18}$$

It is a method of order 1 ($p = 1$). To obtain the value of the coefficients in (2.18), we substitute $\boldsymbol{x}(t_{n+1})$, $\boldsymbol{x}(t_n)$ and $\frac{d\boldsymbol{x}}{dt}\big|_{t=t_{n+1}}$ for $\boldsymbol{x}_{n+1}$, $\boldsymbol{x}_n$ and $\boldsymbol{x}'_{n+1}$ in (2.16), expand each term in its Taylor series, and equate to zero the terms up to the p^{th} derivative. This results in the following method

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + h\boldsymbol{x}'_{n+1}, \tag{2.19}$$

and the values of μ_i and ζ_i are

$$\mu_0 = 1, \quad \mu_1 = -1 \quad \text{and} \quad \zeta_0 = 1.$$

The error constant is $C = 1/2$, which is the coefficient before the $(p+1)^{\text{th}}$ derivative that does vanish in the Taylor expansion.

Backward Euler is an implicit method because $\zeta_0 \neq 0$. In order to obtain $\boldsymbol{x}_{n+1}$, it requires the solution of a system of nonlinear equations.

Trapezoidal rule (TR)

The trapezoidal rule is obtained by weighting the derivatives at both $t = t_n$ and $t = t_{n+1}$ so that (2.14) becomes

$$\mu_0 \boldsymbol{x}_{n+1} + \mu_1 \boldsymbol{x}_n = h(\zeta_0 \boldsymbol{x}'_{n+1} + \zeta_1 \boldsymbol{x}'_n). \tag{2.20}$$

The order of the trapezoidal rule is 2 ($p = 2$). To obtain the value of the coefficients in (2.20), we substitute $\boldsymbol{x}(t_{n+1})$, $\boldsymbol{x}(t_n)$, $\frac{d\boldsymbol{x}}{dt}\big|_{t=t_{n+1}}$ and $\frac{d\boldsymbol{x}}{dt}\big|_{t=t_n}$ for $\boldsymbol{x}_{n+1}$, $\boldsymbol{x}_n$, $\boldsymbol{x}'_{n+1}$ and $\boldsymbol{x}'_n$ in (2.20), expand each term in its Taylor series, and equate to zero the terms up to the p^{th} derivative. This results in the following formula

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \frac{h}{2}(\boldsymbol{x}'_{n+1} + \boldsymbol{x}'_n), \tag{2.21}$$

and the values of μ_i and ζ_i are

$$\mu_0 = 1, \quad \mu_1 = -1 \quad \text{and} \quad \zeta_0 = \zeta_1 = \frac{1}{2}.$$

The error constant is $C = 1/12$, which is the coefficient of the $(p+1)^{\text{th}}$ derivative that does not vanish in the Taylor expansion.

The trapezoidal rule is an implicit method because $\zeta_0 \neq 0$. In order to obtain $\boldsymbol{x}_{n+1}$, it requires the solution of a system of nonlinear equations.

2.3.1.2 Backward differentiation formulas (BDF)

The well-known backward differentiation formulas (BDF) are obtained by setting $\zeta_1 = \zeta_2 = \cdots = \zeta_k = 0$, where (2.14) becomes

$$\sum_{i=0}^{k} \mu_i \boldsymbol{x}_{n+k-i} = h\zeta_0 \boldsymbol{x}'_{n+k}. \tag{2.22}$$

The k-step BDF is an order $p = k$ method. To obtain the coefficients μ_i, $i = 0, \ldots, k$ and ζ_0, in (2.22), we substitute $\boldsymbol{x}(t_{n+k-i})$, $i = 0, \ldots, k$, and $\frac{d\boldsymbol{x}}{dt}\big|_{t=t_{n+k}}$ for $\boldsymbol{x}_{n+k-i}$, $i = 0, \ldots, k$, and $\boldsymbol{x}'_{n+k}$ in (2.22), expand each term in its Taylor series, and equate to zero the terms up to the p^{th} derivative.

Applying the operator in (2.15), the order and error constants defined in (2.7) can be obtained for BDF, as shown in Table 2.1. Note in Table 2.1, the first order BDF is reduced to the backward Euler.

Table 2.1: Order and local truncation error constants of BDF.

Method	BDF					
p	1	2	3	4	5	6
$\lvert C\rvert$	0.5	0.2222	0.1364	0.0960	0.0730	0.0583

2.3.2 Stability of LMS methods

The stability of a general linear multistep method is usually studied by considering the scalar test equation (2.11). Thus, we substitute $x' = \lambda x$ into (2.14) and obtain

$$\sum_{i=0}^{k}(\mu_i - h\lambda\zeta_i)x_{n+k-i} = 0. \tag{2.23}$$

Since the test equation has the known solution

$$x(t) = x(0)e^{\lambda t}, \tag{2.24}$$

we anticipate that the solution of (2.23) has a similar form

$$x_{n+k-i} = x(0)e^{\lambda h(n+k-i)} = x(0)e^{q(n+k-i)}, \tag{2.25}$$

where $q = h\lambda$. Inserting this anticipated solution into (2.23), we obtain

$$x(0)e^{qn}\left\{\sum_{i=0}^{k}\mu_i e^{q(k-i)} - q\sum_{i=0}^{k}\zeta_i e^{q(k-i)}\right\} = 0. \tag{2.26}$$

The multiplicative factor can be removed. Introducing a new variable

$$z = e^q, \tag{2.27}$$

(2.26) simplifies to

$$\sum_{i=0}^{k} \mu_i z^{(k-i)} - q \sum_{i=0}^{k} \zeta_i z^{(k-i)} = 0. \tag{2.28}$$

It is usual to introduce new notations for the polynomials in z:

$$\rho(z) = \sum_{i=0}^{k} \mu_i z^{(k-i)}, \quad \sigma(z) = \sum_{i=0}^{k} \zeta_i z^{(k-i)}. \tag{2.29}$$

This simplifies (2.28) to

$$\rho(z) - q\sigma(z) = 0. \tag{2.30}$$

It is a polynomial equation of degree k in the variable z and must have k roots, which we designate as z_j. Assume that all of them are simple. Then

$$\rho(z) - q\sigma(z) = \gamma(z - z_1) \cdots (z - z_j) \cdots (z - z_k). \tag{2.31}$$

Insert the j^{th} root z_j into (2.25); at the $(n + k - i)^{\text{th}}$ step we have

$$(x_{n+k-i})_j = x(0) z_j^{n+k-i},$$

where $(x_{n+k-i})_j$ is the approximation to the j^{th} root. Now assume that we take a large number of steps, $n \to \infty$. To obtain a stable solution, we require

$$|z_j| \leq 1. \tag{2.32}$$

The above condition on the roots of $\rho(z) - q\sigma(z)$ is typically known as the root condition. In particular, it requires that all the roots of $\rho(z) - q\sigma(z)$ be on or inside the unit circle, with only simple roots allowed on the unit circle of the q-domain.

Thus. to characterize the stability of an LMS method, i.e., to find its absolute stability region in the q-domain, one must find the roots of $\rho(z) - q\sigma(z) = 0$ in terms of q. This would give the picture as to for which values of q, the roots will venture outside the unit circle.

Unfortunately, except for some simple cases, computing the roots of the polynomials (2.30) as explicit functions of q is very difficult.

This fact makes the study of stability properties of LMS methods a difficult task. Nonetheless, some simplified stability properties could be understood for two limiting cases. The first case happens when the step size h is so small that $h \to 0$, and (2.30) reduces to

$$\rho(z) = 0. \tag{2.33}$$

Thus, under this condition ($h \to 0$), the stability condition requires that the roots of $\rho(z)$ alone be on or inside the unit circle. Such stability condition is known as 0-stability.

The second case happens when the step size tends to be large, where the second term in (2.30) becomes dominant, thereby requiring that the roots of $\sigma(z)$ be inside or on the unit circle. Such stability is known as A_∞-stability.

For single-step methods, complete stability characteristics are very simple since one can obtain the roots of the polynomial $\rho(z) - q\sigma(z)$ as explicit functions of q. To illustrate this issue, we revisit the BE, FE and TR methods and deduce their regions of absolute stability.

2.3.2.1 Stability of the forward Euler method

We start with the forward Euler method. Inserting the scalar test equation $x' = \lambda x$ into the forward Euler formula (2.17) for the derivative, we obtain

$$x_{n+1} = (1 + h\lambda)x_n. \tag{2.34}$$

For the result to be finite for a stable differential equation ($\Re e(\lambda) < 0$), we must have

$$|1 + q| \leq 1, \tag{2.35}$$

where $q = h\lambda$. This is the condition for the forward Euler method to be stable. Insert $q = u + iv$ into (2.35). The result is

$$|1 + u + iv| \leq 1 \quad \text{or} \quad (1 + u)^2 + v^2 \leq 1.$$

This is the region inside a unit circle with center at (-1, 0) passing through the origin, as shown in Figure (2.2). The region of stability is inside the circle. For large $|\lambda|$, the step size h must be small to make $q = h\lambda$ inside the stability region.

2.3.2.2 Stability of the backward Euler method

The absolute stability region of the backward Euler is studied by inserting the scalar test equation (2.11) into the backward Euler formula (2.19) for x'. We obtain

$$x_{n+1} = x_n + h\lambda x_{n+1}$$

or

$$x_{n+1} = \frac{x_n}{1 - h\lambda} = \frac{1}{1 - q}x_n.$$

In order to get a stable solution for $n \to \infty$ we require

$$\left|\frac{1}{1 - q}\right| \leq 1.$$

Inserting $q = u + iv$, we obtain

$$(1-u)^2 + v^2 \geq 1.$$

If we take the equality in the above relation, we get a unit circle with center at (1, 0) passing through the origin, as shown in Figure (2.3). The inequality is satisfied outside this circle; thus, the backward Euler is stable for all λ in the left half plane. In the very stiff limit, when $h\lambda \to -\infty$, we have

$$\frac{x_{n+1}}{x_n} = \left.\frac{1}{1-q}\right|_{q\to-\infty} \to 0.$$

Thus we conclude that the backward Euler is an L-stable method.

2.3.2.3 Stability of the trapezoidal rule

Now consider the trapezoidal rule (2.21). Inserting the scalar test equation (2.11) for both x'_{n+1} and x'_n, we obtain

$$x_{n+1} = x_n + \frac{h\lambda}{2}(x_n + x_{n+1})$$

or

$$x_{n+1} = \frac{1 + \frac{h\lambda}{2}}{1 - \frac{h\lambda}{2}} x_n.$$

In order to obtain a stable solution in the limiting case, as $n \to \infty$, the stability requirement is

$$\left|\frac{2+q}{2-q}\right| \leq 1 \quad \text{or} \quad \left|\frac{2+u+iv}{2-u-iv}\right| \leq 1.$$

Simplifying, we obtain

$$4u \leq 0,$$

which shows that the boundary of the stability region is the imaginary axis. The trapezoidal rule is stable for any λ with $\Re e(\lambda) \leq 0$, as shown in Figure (2.4). This means it is an A-stable method.

The Dahlquist Barriers

Despite the difficulty in studying the stability characteristic of general LMS methods, Dahlquist [3] has established a very important property concerning the relation between their stability in the entire left half plane and their order. More specifically, Dahlquist proved that A-stable LMS methods cannot have order higher than 2, i.e., $p \leq 2$. Further, Dahlquist showed that the trapezoidal rule has the lowest possible truncation error for all A-stable LMS methods.

2.3.2.4 Stability of BDF methods

Gear [5] studied the stability problem and derived backward differentiation formula. He argued that the stability region of BDF, as shown in Figure 2.6, is better suited for practical problems. He suggested that whenever a system has desired as well as parasitic elements, there will be some desired poles in the region close to $q = 0$ and some parasitic poles that are far from the origin. All poles contribute to the response with components of the form $x' = \lambda x$ and must be located inside the region of stability. If a BDF is applied to a stiff problem, the stability of the solution can be secured even for large step sizes. Parasitic poles that have large magnitude and lie close to the imaginary axis require the formula to be A_∞ stable.

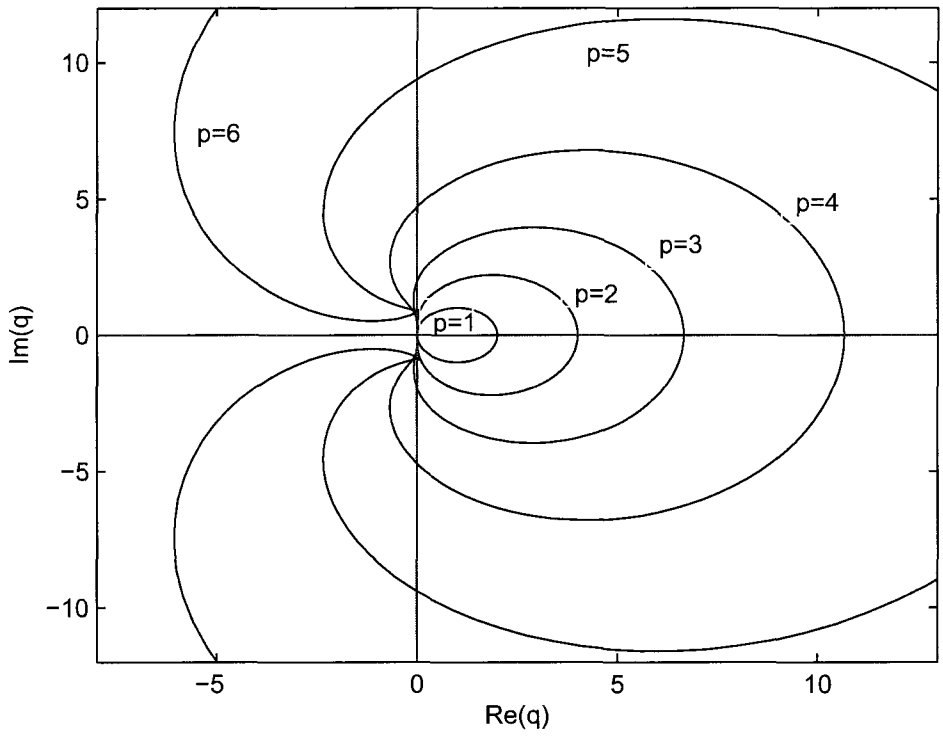

Figure 2.6: Stability regions of BDF of order 1 to 6. The curves close in the right half plane and the stable regions are outside the curves.

Applying the test equation (2.11) to BDF (2.22), we obtain the polynomial

$$\sum_{i=0}^{k} \mu_i z^{(k-i)} - q\zeta_0 z^k = 0, \tag{2.36}$$

where $q = h\lambda$ and $z = e^q$. When $h \to \infty$, (2.36) reduces to

$$q\zeta_0 z^k = 0. \tag{2.37}$$

All the roots of (2.37) are inside the unit circle, at the origin, i.e., BDF methods are A_∞ stable. The highest order BDF which is $A(\alpha)$-stable is of order 6, because for higher orders the boundaries protruding into the left half plane start cross the negative real axis and each other.

2.4 Single-step multistage methods

The single-step multistage methods do not use any information from previous steps, but use information at the intermediate stages within the current step. The Runge–Kutta (RK) methods [6], [1] are the best known examples of this class.

In general, an s-stage Runge–Kutta method for the ODE system

$$\boldsymbol{x}' = \boldsymbol{f}(t, \boldsymbol{x})$$

can be written in the form

$$\boldsymbol{X}_i = \boldsymbol{x}_{n-1} + h \sum_{j=1}^{s} a_{ij} \boldsymbol{f}(t_{n-1} + c_j h, \boldsymbol{X}_j), \quad 1 \leq i \leq s, \tag{2.38}$$

$$\boldsymbol{x}_n = \boldsymbol{x}_{n-1} + h \sum_{i=1}^{s} b_i \boldsymbol{f}(t_{n-1} + c_i h, \boldsymbol{X}_i). \tag{2.39}$$

The stage values $\boldsymbol{X}_i$'s are intermediate approximations to the solution at times $t_{n-1} + c_i h$, which may be correct to a lower order of accuracy than the solution $\boldsymbol{x}_n$ at the end of the step. $\boldsymbol{X}_i$ are local to the step from t_{n-1} to t_n, and the only approximation that the next step "sees" is $\boldsymbol{x}_n$. The coefficients of the method are chosen so that error terms cancel in such a way that $\boldsymbol{x}_n$ approaches $\boldsymbol{x}(t_n)$.

The method can be conveniently represented by the Butcher tableau

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & a_{22} & \dots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \end{array}$$

or in concise form

$$\begin{array}{c|c} \boldsymbol{c} & \boldsymbol{A} \\ \hline & \boldsymbol{b}^{\mathrm{T}} \end{array}.$$

We will always choose

$$c_i = \sum_{j=1}^{s} a_{ij}, \quad i = 1, \ldots, s. \tag{2.40}$$

A Runge–Kutta method is explicit iff $a_{ij} = 0$ for $j \geq i$, because then each $\boldsymbol{X}_i$ in (2.38) is given in terms of previous stages.

Some examples of explicit Runge–Kutta methods are given below:

Forward Euler, $s = 1$, $p = 1$,

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array}$$

One-parameter family of second-order methods, $s = 2$, $p = 2$,

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \alpha & \alpha & 0 \\ \hline & 1 - \frac{1}{2\alpha} & \frac{1}{2\alpha} \end{array}$$

The classical fourth-order method is written as

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

For an explicit s-stage Runge–Kutta method, the attainable order as a function of the number of stages is listed in Table 2.2.

Table 2.2: Attainable order of explicit RK as a function of the number of stages.

Number of stages	1	2	3	4	5	6	7	8	9	10
Attainable order	1	2	3	4	4	5	6	6	7	7

Compared to explicit Runge–Kutta methods, for implicit Runge–Kutta methods there are many more parameters to choose in (2.38). Thus, given same number of stages, higher order can be achieved by implicit Runge–Kutta methods. For implicit Runge–Kutta methods, the highest order attainable by s stages is $p = 2s$.

Many of the most commonly used implicit Runge–Kutta methods are based on quadrature methods, i.e., the points where the intermediate stage approximations are taken are the same points used in certain classes of integration formulas. There are several classes of these methods. The first two instances of each class are given below.

Gauss methods: These are the maximum order methods, i.e., an s-stage Gauss method has order $2s$:

$$\begin{array}{c|c} \frac{1}{2} & \frac{1}{2} \\ \hline & 1 \end{array} \qquad \text{implicit midpoint,} \qquad s = 1,\ p = 2,$$

$$\begin{array}{c|cc} \frac{3-\sqrt{3}}{6} & \frac{1}{4} & \frac{3-2\sqrt{3}}{12} \\ \frac{3+\sqrt{3}}{6} & \frac{3+2\sqrt{3}}{12} & \frac{1}{4} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} \qquad s = 2,\ p = 4.$$

Radau methods: These correspond to integration formulas where one end of the interval is included ($c_1 = 0$ or $c_s = 1$). They attain order $p = 2s - 1$. The choice $c_1 = 0$ makes no sense, so we consider only the case $c_s = 1$:

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array} \qquad \text{backward Euler} \qquad s = 1,\ p = 1,$$

$$\begin{array}{c|cc} \frac{1}{3} & \frac{5}{12} & -\frac{1}{12} \\ 1 & \frac{3}{4} & \frac{1}{4} \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array} \qquad s = 2,\ p = 3.$$

Lobatto methods: These correspond to integration formulas where the function is sampled at both ends of the interval. The order is $p = 2s - 2$.

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & \frac{1}{2} & \frac{1}{2} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} \qquad \text{trapezoidal rule} \qquad s = 2,\ p = 2,$$

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{5}{24} & \frac{1}{3} & -\frac{1}{24} \\ 1 & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \\ \hline & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{array} \qquad s = 3,\ p = 4.$$

The absolute stability region of a Runge–Kutta method is studied by inserting the scalar test equation $x' = \lambda x$ into the method and obtain the amplification function

$$x_n = \mathcal{R}(q)x_{n-1},$$

where $q = h\lambda$,

$$\mathcal{R}(q) = 1 + q\boldsymbol{b}^{\mathrm{T}}(1 - q\boldsymbol{A})^{-1}\mathbf{1}, \tag{2.41}$$

and $\mathbf{1} = (1, 1\ldots, 1)^{\mathrm{T}}$. The region of absolute stability is given by the set of values q such that

$$|\mathcal{R}(q)| \leq 1. \tag{2.42}$$

For an explicit method, $\mathcal{R}(q)$ is a polynomial. In particular, the amplification function

of an explicit p^{th}-order Runge–Kutta method for $s = p \leq 4$ is given by

$$\mathcal{R}(q) = 1 + h\lambda + \frac{(h\lambda)^2}{2} + \cdots + \frac{(h\lambda)^p}{p!}. \tag{2.43}$$

All p-stage explicit Runge–Kutta methods of order p have the same region of absolute stability, shown in Figure 2.7. For an s-stage method with order $p < s$, the absolute stability region is seen to depend somewhat on the method's coefficients.

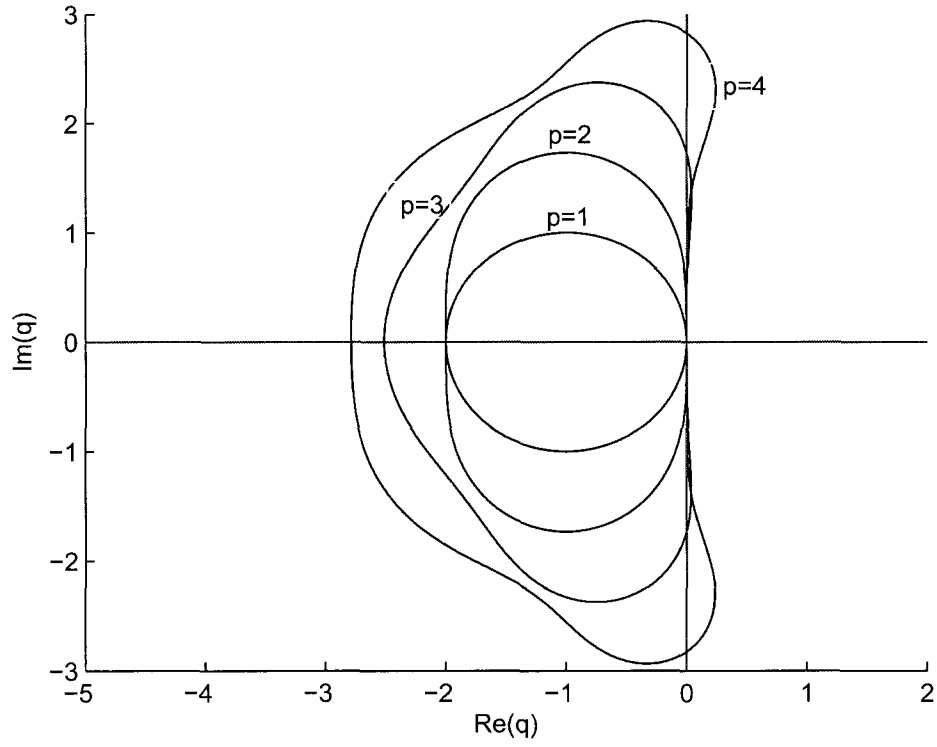

Figure 2.7: Absolute stability regions for the p-stage explicit RK of order p, $p = 1, 2, 3, 4$. The inner circle corresponds to forward Euler, $p = 1$. The absolute stability region is inside the circle.

For implicit Runge–Kutta methods, $\mathcal{R}(q)$ is a rational function,

$$\mathcal{R}(q) = \frac{P(q)}{Q(q)}. \tag{2.44}$$

A-stable implicit Runge–Kutta methods are abundant. Ehle [17] has shown that s-stage fully implicit Runge–Kutta methods of order $2s$ developed in [18] are all A-stable.

When $\Re e(q) \to -\infty$ we would also like a method to have stiff decay. For this we must have $\mathcal{R}(-\infty) = 0$ in (2.41), which is achieved if $P(q)$ in (2.44) has a lower degree than $Q(q)$. Note that by (2.41), if $\boldsymbol{A}$ is nonsingular, then $\mathcal{R}(-\infty) = 1 - \boldsymbol{b}^T\boldsymbol{A}^{-1}\mathbf{1}$, so $\mathcal{R}(-\infty) = 0$ if the last row of $\boldsymbol{A}$ coincides with $\boldsymbol{b}^{\mathrm{T}}$. In particular,

1. the Radau methods, extending the backward Euler, have stiff decay;

2. the Gauss and Lobatto methods, which extend midpoint and trapezoidal, do not have stiff decay, although they are A-stable.

One of the challenges for implicit Runge–Kutta methods is the development of efficient implementations. Consider again the general Runge–Kutta method

$$\boldsymbol{X}_i = \boldsymbol{x}_{n-1} + h\sum_{j=1}^{s} a_{ij}\boldsymbol{f}(t_{n-1} + c_j h, \boldsymbol{X}_j), \quad 1 \le i \le s$$

$$\boldsymbol{x}_n = \boldsymbol{x}_{n-1} + h\sum_{i=1}^{s} b_i\boldsymbol{f}(t_{n-1} + c_i h, \boldsymbol{X}_i),$$

and assume that Newton–Raphson method is used to solve for $\boldsymbol{X}_i$. For the v^{th} Newton iteration, let $\boldsymbol{\delta}_i = \boldsymbol{X}_i^{v+1} - \boldsymbol{X}_i^v$ and $\boldsymbol{r}_i = \boldsymbol{X}_i^v - \boldsymbol{x}_{n-1} - h\sum_{j=1}^{s} a_{ij}\boldsymbol{f}(\boldsymbol{X}_j^v)$. Then the Newton iteration takes the form

$$\begin{bmatrix} \boldsymbol{I} - ha_{11}\boldsymbol{J}_1 & -ha_{12}\boldsymbol{J}_2 & \dots & -ha_{1s}\boldsymbol{J}_s \\ -ha_{21}\boldsymbol{J}_1 & \boldsymbol{I} - ha_{22}\boldsymbol{J}_2 & \dots & -ha_{2s}\boldsymbol{J}_s \\ \vdots & \vdots & \ddots & \vdots \\ -ha_{s1}\boldsymbol{J}_1 & -ha_{s2}\boldsymbol{J}_2 & \dots & \boldsymbol{I} - ha_{ss}\boldsymbol{J}_s \end{bmatrix} \begin{bmatrix} \boldsymbol{\delta}_1 \\ \boldsymbol{\delta}_2 \\ \vdots \\ \boldsymbol{\delta}_s \end{bmatrix} = - \begin{bmatrix} \boldsymbol{r}_1 \\ \boldsymbol{r}_2 \\ \vdots \\ \boldsymbol{r}_s \end{bmatrix}, \tag{2.45}$$

where $\boldsymbol{J}_i = \partial \boldsymbol{f}/\partial \boldsymbol{x}$ is evaluated at $\boldsymbol{X}_i^v, i = 1, 2, \ldots, s$. We note that for a system of N differential equations, this is a strongly coupled equation and will require $(sN)^3$ operations to solve.

To overcome this, diagonally implicit Runge–Kutta methods (DIRK) are adopted. The coefficient matrix $\boldsymbol{A}$ of DIRK is lower triangular, with equal coefficients a along the diagonal. Thus, the Newton iteration equation (2.45) becomes a block lower triangular matrix and computational cost can be saved. The disadvantage is that the maximum attainable order of s-stage DIRK method cannot exceed $s + 1$, because by construction so many of the coefficients of DIRK methods have been specified to be zero. More seriously, DIRK suffer from a reduction in their order of convergence in the very stiff limit. At the very stiff limit, DIRK methods are only first-order accurate [1], [6].

2.5 General linear methods (GLM)

The work on GLM was motivated by the need to circumvent the existing barriers in LMS while addressing the computational difficulties of RK methods [19]. Examples of these methods are the diagonally-implicit multi-stage integration methods (DIMSIM) [20] and multistep collocation techniques [21]. However, they still inherit some of the difficulties of both LMS and RK such as the need for powerful optimization tools to construct the method [10].

GLM methods typically use the past r points along with s stages to advance to the next step. The formulation of a GLM method is usually represented by the four matrices $\boldsymbol{A} \in \mathbb{R}^{s\times s}$, $\boldsymbol{B} \in \mathbb{R}^{r\times s}$, $\boldsymbol{U} \in \mathbb{R}^{s\times r}$ and $\boldsymbol{V} \in \mathbb{R}^{r\times r}$. These can be written

together as a partitioned $(s+r) \times (s+r)$ matrix.

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{U} \\ \boldsymbol{B} & \boldsymbol{V} \end{bmatrix}.$$

The input vectors available from step $(n-1)$ will be denoted by $\boldsymbol{x}_1^{[n-1]}$, $\boldsymbol{x}_2^{[n-1]}$, $\ldots$, $\boldsymbol{x}_r^{[n-1]}$. During the computation at the next step, n, stage values $\boldsymbol{X}_1$, $\boldsymbol{X}_2$, $\ldots$, $\boldsymbol{X}_s$ are computed and derivatives values $\boldsymbol{F}_i = \boldsymbol{f}(\boldsymbol{X}_i)$, $i = 1, 2, \ldots, s$, are computed in terms of these. The output values are computed and denoted by $\boldsymbol{x}_i^{[n]}$, $i = 1, 2, \ldots, r$. The relationships between these quantities are defined in terms of the elements of $\boldsymbol{A}$, $\boldsymbol{U}$, $\boldsymbol{B}$ and $\boldsymbol{V}$ by the equations

$$\boldsymbol{X}_i = \sum_{j=1}^{s} a_{ij} h \boldsymbol{F}_j + \sum_{j=1}^{r} u_{ij} \boldsymbol{x}_j^{[n-1]}, \quad i = 1, 2, \ldots, s, \tag{2.46}$$

$$\boldsymbol{x}_i^{[n]} = \sum_{j=1}^{s} b_{ij} h \boldsymbol{F}_j + \sum_{j=1}^{r} v_{ij} \boldsymbol{x}_j^{[n-1]}, \quad i = 1, 2, \ldots, r. \tag{2.47}$$

It is convenient to use a more concise notation and we define vectors $\boldsymbol{X}$, $\boldsymbol{F} \in \mathbb{R}^{sN}$ and $\boldsymbol{x}^{[n-1]}$, $\boldsymbol{x}^{[n]} \in \mathbb{R}^{rN}$ as follows:

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{X}_1 \\ \boldsymbol{X}_2 \\ \vdots \\ \boldsymbol{X}_s \end{bmatrix}, \quad \boldsymbol{F} = \begin{bmatrix} \boldsymbol{F}_1 \\ \boldsymbol{F}_2 \\ \vdots \\ \boldsymbol{F}_s \end{bmatrix}, \quad \boldsymbol{x}^{[n-1]} = \begin{bmatrix} \boldsymbol{x}_1^{[n-1]} \\ \boldsymbol{x}_2^{[n-1]} \\ \vdots \\ \boldsymbol{x}_r^{[n-1]} \end{bmatrix}, \quad \boldsymbol{x}^{[n]} = \begin{bmatrix} \boldsymbol{x}_1^{[n]} \\ \boldsymbol{x}_2^{[n]} \\ \vdots \\ \boldsymbol{x}_r^{[n]} \end{bmatrix}.$$

Using these super-vectors, it is convenient to write (2.46) and (2.47) in the form

$$\begin{bmatrix} \boldsymbol{X} \\ \boldsymbol{x}^{[n]} \end{bmatrix} = \begin{bmatrix} \boldsymbol{A} \otimes \boldsymbol{I}_N & \boldsymbol{U} \otimes \boldsymbol{I}_N \\ \boldsymbol{B} \otimes \boldsymbol{I}_N & \boldsymbol{V} \otimes \boldsymbol{I}_N \end{bmatrix} \begin{bmatrix} h\boldsymbol{F} \\ \boldsymbol{x}^{[n-1]} \end{bmatrix}. \tag{2.48}$$

In this formulation, $\boldsymbol{I}_N$ denotes the $N \times N$ unit matrix and the Kronecker product

$\otimes$ is given by

$$\boldsymbol{A} \otimes \boldsymbol{I}_N = \begin{bmatrix} a_{11}\boldsymbol{I}_N & a_{12}\boldsymbol{I}_N & \dots & a_{1s}\boldsymbol{I}_N \\ a_{21}\boldsymbol{I}_N & a_{22}\boldsymbol{I}_N & \dots & a_{2s}\boldsymbol{I}_N \\ \vdots & \vdots & & \vdots \\ a_{s1}\boldsymbol{I}_N & a_{s2}\boldsymbol{I}_N & \dots & a_{ss}\boldsymbol{I}_N \end{bmatrix}.$$

When there is no possibility of confusion, the notation can be simplified by replacing

$$\begin{bmatrix} \boldsymbol{A} \otimes \boldsymbol{I}_N & \boldsymbol{U} \otimes \boldsymbol{I}_N \\ \boldsymbol{B} \otimes \boldsymbol{I}_N & \boldsymbol{V} \otimes \boldsymbol{I}_N \end{bmatrix} \quad \text{by} \quad \begin{bmatrix} \boldsymbol{A} & \boldsymbol{U} \\ \boldsymbol{B} & \boldsymbol{V} \end{bmatrix}.$$

To study the stability, we insert the scalar test equation $x' = \lambda x$ into (2.48) and rearrange terms, so that the stability matrix $\boldsymbol{M}(q)$ is obtained:

$$\boldsymbol{M}(q) = \boldsymbol{V} + q\boldsymbol{B}(\boldsymbol{I} - q\boldsymbol{A})^{-1}\boldsymbol{U},$$

where $q = h\lambda$. The stability function for a general linear method is the polynomial $\Phi(z, q)$

$$\Phi(z, q) = \det(z\boldsymbol{I} - \boldsymbol{M}(q))$$

and the absolute stability region is the subset of the complex plane such that if q is in this subset, then

$$\sup_{n=1}^{\infty} ||\boldsymbol{M}(q)^n|| < \infty.$$

A general linear method is A-stable if $\boldsymbol{M}(q)$ is power bounded for every q in the left half complex plane. To be A-stable, all the z-roots of $\Phi(z, q) = 0$ must lie inside the unit disk whenever $\Re e(q) \leq 0$, with only simple roots allowed on the unit circle [6]. The barrier on the highest order can be achieved by all the A-stable methods is given by the following theorem.

Theorem 2.1 *Let $\Phi(z,q)$ be a two-dimensional polynomial in z and q corresponding to an A-stable GLM method of order p. Then $p \leq 2s$. Furthermore, methods attaining the highest possible order, i.e., for $p = 2s$, will have an error constant satisfying the inequality*

$$|C| \geq \frac{s!s!}{(2s)!(2s+1)!}. \tag{2.49}$$

The theory of order stars provides the proof [16].

2.6 System of differential equations

In this section, we extend the test equation (2.11) to a system of ordinary differential equations [1], [22]. We first consider the following linear system

$$\boldsymbol{x}' = \boldsymbol{J}\boldsymbol{x}, \tag{2.50}$$

where $\boldsymbol{J}$ is a constant, diagonalizable, $N \times N$ matrix. Denote the eigenvalues of $\boldsymbol{J}$ by $\lambda_1, \lambda_2, \ldots, \lambda_N$, and let

$$\boldsymbol{\Lambda} = \text{diag}\{\lambda_1, \lambda_2, \ldots, \lambda_N\}$$

be the diagonal $N \times N$ matrix composed of these eigenvalues. We are only interested in the stable circuit, i.e., $\Re e(\lambda_j) \leq 0$, $j = 1, 2, \ldots, N$. The diagonalizability of $\boldsymbol{J}$ means the there is a nonsingular matrix $\boldsymbol{T}$, consisting the eigenvectors of $\boldsymbol{J}$, such that

$$\boldsymbol{T}^{-1}\boldsymbol{J}\boldsymbol{T} = \boldsymbol{\Lambda}.$$

Consider the following transformation of dependent variables,

$$\boldsymbol{w} = \boldsymbol{T}^{-1}\boldsymbol{x}.$$

Upon multiplying (2.50) by $\boldsymbol{T}^{-1}$ and noting that $\boldsymbol{T}$ is constant in t, we obtain the decoupled system for $\boldsymbol{w}(t)$

$$\boldsymbol{w}' = \boldsymbol{\Lambda}\boldsymbol{w}. \tag{2.51}$$

The components of $\boldsymbol{w}$ are separated, and for each component we get a scalar ODE in the form of the test equation $x' = \lambda x$ with $\lambda = \lambda_k$, $k = 1, 2, \ldots, N$.

The characteristics of a system of nonlinear differential equations

$$\boldsymbol{x}' = \boldsymbol{f}(\boldsymbol{x}, t) \tag{2.52}$$

are captured by analyzing its Jacobian matrix

$$\boldsymbol{J}(t) = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\Big|_{\boldsymbol{x}(t)}, \tag{2.53}$$

where $\boldsymbol{x}(t)$ is the particular solution of (2.52) which we attempt to compute. We are concerned with the solution of (2.52) in the neighborhood of $\boldsymbol{x}(t)$. This leads us to consider the variation system

$$\boldsymbol{x}' = \boldsymbol{J}(t)\boldsymbol{x} \tag{2.54}$$

associated with (2.52) and the particular solution. In a time interval Δt, chosen so that there is moderate change in the value of the solution to (2.52), and very little change in $\boldsymbol{J}(t)$, (2.54) simplifies to the system of linear differential equations (2.50) in a local sense.

Chapter 3

Single-step multi-derivative methods

This chapter proposes a new integration method for the transient simulation of electronic circuits. The basic idea in this chapter is that the search for A-stable methods should not be restricted to multistep and multi-stage paradigms. Instead, one can incorporate information from a third dimension, namely, using high-order derivatives. Such methods were first proposed by Obreshkov [23] but left untried due to the practical difficulties involved in handling high-order derivatives. Our goal in this chapter is to present this formula, illustrate its potentials for high-order A-stable solution of differential equations, and extend it to be L-stable to handle very stiff differential equations. Section 3.2 presents a general single-step multi-derivative formula, i.e., the modified Obreshkov formula, and shows the link between this formula and the Padé approximants. Section 3.3 shows that the Obreshkov formula is a special case in the general class and demonstrates its advantages and difficulties. Section 3.4 discusses

the stability properties of the modified Obreshkov formula. Section 3.5 extends the single-step Obreshkov formula to multi-step multi-derivative methods. The practical issues involved are left to chapter 5.

3.1 Introduction

Consider a scalar ordinary differential equation of the form

$$x' = f(t, x). \tag{3.1}$$

One potential way of approximating $x(t_{n+1})$ at time point, t_n, is through employing a Taylor series expansion and truncating it after $p+1$ terms

$$x_{n+1} \approx x_n + hx'_n + \frac{h^2}{2}x''_n + \cdots + \frac{h^p}{p!}x_n^{(p)}, \tag{3.2}$$

where

$$x_n^{(i)} = \left.\frac{d^i x}{dt^i}\right|_{t=t_n}, \tag{3.3}$$

and $h = t_{n+1} - t_n$.

Thus, if it is possible to evaluate the high-order derivatives of $x(t)$ at any time point, then one can use the above series to approximate $x(t)$ at any time point. One possible way to achieve that is through the definition of the scalar differential equation in (3.1), whereby differentiating both sides by the chain rule yields

$$\begin{aligned} x_n^{(2)} &= \frac{\partial f}{\partial x}x_n^{(1)} + \frac{\partial f}{\partial t} \\ x_n^{(3)} &= \frac{\partial f}{\partial x}x_n^{(2)} + 2\frac{\partial^2 f}{\partial x^2}\left(x_n^{(1)}\right)^2 + \frac{\partial^2 f}{\partial t^2} \\ &\vdots \end{aligned}$$

Although, the method by the Taylor series approximation will be explicit, thus requiring no matrix inversion, it will still have the following two difficulties.

1. The method can be shown not to be of the A-stable class, thus diminishing its practicality for circuit applications. The fact that the method is not A-stable can be demonstrated by using the scalar linear test problem

$$x' = \lambda x$$

where, in this case, the approximate x_{n+1} is given by

$$x_{n+1} = \left(\sum_{k=0}^{p} \frac{(h\lambda)^k}{k!} \right) x_n. \tag{3.4}$$

Since the term $\sum_{k=0}^{p} \frac{(h\lambda)^k}{k!}$ can be greater than unity for arbitrary large λ, the method then looses its A-stability outside a certain domain of the $h\lambda$-plane.

2. The second difficulty is actually related to the practical issue of computing high-order derivatives of $f(t, x)$. Beyond a few higher-order derivatives, this task becomes very cumbersome and may require the use of automatic differentiation tools, which could be a source of computational bottleneck.

The goal of this chapter is to use the concept of high-order derivatives and avoid the loss of A-stability, mentioned as the first difficulty. We will further show that one can preserve the stronger L-stability by choosing the number of high-order derivatives properly.

Subsequent chapters will address the second difficulty, namely, the implementation details for addressing the computing issues of high-order derivatives.

3.2 Modified single-step Obreshkov formula

The Taylor series approximation formula can be extended by employing the derivatives at the current point (the unknown derivatives) as follows

$$\sum_{i=0}^{m} \alpha_i h^i x_{n+1}^{(i)} = \sum_{i=0}^{l} \beta_i h^i x_n^{(i)}, \tag{3.5}$$

where m, l are integers. Called modified Obreshkov formula, (3.5) is a general form of the Obreshkov formula. In this section, we derive the coefficients α_i and β_i in the modified Obreshkov formula. Then in the next section, we show that the Obreshkov formula is a special case of the modified Obreshkov formula by setting $l = m$.

Computing the coefficients α_i and β_i in (3.5) is typically carried out by assuming that the values and derivatives obtained at the previous time point ($t = t_n$) are accurate, i.e.,

$$x_n^{(i)} = \left.\frac{d^i x}{dt^i}\right|_{t=t_n},$$

and then seeking to make x_{n+1} approximate the exact value $x(t_{n+1})$ as accurately as possible. To achieve this, one can substitute $x_{n+1}^{(i)}$ in (3.5) by $\left.\frac{d^i}{dt^i}x(t)\right|_{t=t_{n+1}}$ to obtain the following

$$\sum_{i=0}^{m} \alpha_i h^i \left.\frac{d^i x}{dt^i}\right|_{t=t_{n+1}} - \sum_{i=0}^{l} \beta_i h^i x_n^{(i)} = O(h^{p+1}), \tag{3.6}$$

where the term $O(h^{p+1})$ accounts for the error between using $x(t_{n+1})$ and x_{n+1} and the derivatives. To make the error as small as possible, one must try to make p as large as possible. This can be carried out by requiring that the coefficients α_i and β_i cancel as many terms on the left hand side of (3.6). To this end, we substitute $x^{(i)}(t_{n+1})$ in (3.6) by its Taylor series expansion around $x(t_n)$,

$$x(t_{n+1}) = \sum_{j=0}^{\infty} \frac{1}{j!} h^j x_n^{(j)},$$

$$\left.\frac{d^i}{dt^i}x(t)\right|_{t=t_{n+1}} = \sum_{j=i}^{\infty} \frac{1}{(j-i)!} h^{j-i} x_n^{(j)}.$$

Given that there are $l+m+1$ coefficients α_i and β_i (α_0 is normalized to one), we have $l+m+1$ degrees of freedom to cancel $l+m+1$ terms of powers of h. Thus, to make the powers of h^i ($i = 0, \ldots, l+m$) vanish, we will need to set the coefficients α_i and β_i to be

$$\alpha_i = (-1)^i \frac{(l+m-i)!}{(l+m)!} \frac{m!}{i!(m-i)!} \quad i = 0, \ldots, m, \tag{3.7}$$

$$\beta_i = \frac{(l+m-i)!}{(l+m)!} \frac{l!}{i!(l-i)!} \quad i = 0, \ldots, l. \tag{3.8}$$

In order to get more insight indicated by the coefficients α_i and β_i, we consider the exponential function $x(t) = e^t$. Replacing $x(t_{n+1})$ with e^{t_n+h}, $x(t_n)$ with e^{t_n} in (3.6) and noticing that

$$e^{t_n+h} = e^{t_n} + he^{t_n} + \frac{h^2}{2!}e^{t_n} + \cdots = \left(1 + h + \frac{h^2}{2!} + \ldots\right) e^{t_n}, \tag{3.9}$$

we obtain

$$\sum_{i=0}^{m} \alpha_i h^i \left(1 + h + \frac{h^2}{2!} + \ldots\right) e^{t_n} - \sum_{i=0}^{l} \beta_i h^i e^{t_n} = O(h^{p+1}). \tag{3.10}$$

Dividing both sides of (3.10) by $\sum_{i=0}^{m} \alpha_i h^i$, we obtain

$$\left(1 + h + \frac{h^2}{2!} + \ldots\right) e^{t_n} - \frac{\sum_{i=0}^{l} \beta_i h^i}{\sum_{i=0}^{m} \alpha_i h^i} e^{t_n} = O(h^{p+1}). \tag{3.11}$$

At $t_n = 0$, $e^{t_n} = 1$ in (3.11). Consider a general $[l/m]$ Padé approximant to the exponential function [24]

$$\mathcal{R}_{(l,m)}(q) = \frac{N_{lm}(q)}{N_{ml}(-q)}, \tag{3.12}$$

where $N_{lm}(q)$ is a polynomial of degree l,

$$N_{lm}(q) = \frac{l!}{(l+m)!} \sum_{i=0}^{l} \frac{(l+m-i)!}{i!(l-i)!} q^i. \tag{3.13}$$

Using the fact that $e^q = 1 + q + \frac{q^2}{2!} + \ldots$, we obtain

$$e^q - \frac{N_{lm}(q)}{N_{ml}(-q)} = O(q^{l+m+1})$$
$$\left(1 + q + \frac{q^2}{2!} + \ldots\right) - \frac{\sum_{i=0}^{l} \hat{\beta}_i q^i}{\sum_{i=0}^{m} \hat{\alpha}_i q^i} = O(q^{l+m+1}), \tag{3.14}$$

where $N_{lm}(q)$ and $N_{ml}(-q)$ are expressed in polynomials by $\hat{\alpha}_i$ and $\hat{\beta}_i$. Comparing (3.11) and (3.14), we conclude that the coefficients α_i and β_i defined in (3.7) and (3.8) are indeed the coefficients in the $[l/m]$ Padé approximant to the exponential function since the $[l/m]$ Padé approximant to the exponential function is unique. The order of approximation of the modified Obreshkov formula (3.5) is $p = m + l$. The error constant is defined as [24]

$$C_{(l,m)} = (-1)^m \frac{l!m!}{(l+m)!(l+m+1)!}. \tag{3.15}$$

3.3 Single-step Obreshkov formula

Setting $l = m$ in (3.7) and (3.8), we notice that the coefficients α_i and β_i in the modified Obreshkov formula simplify to

$$\alpha_i = (-1)^i \beta_i = (-1)^i \frac{(2m-i)!}{(2m)!} \frac{m!}{i!(m-i)!}, \quad i = 0, \ldots, m. \tag{3.16}$$

Define

$$\alpha_{i,m} = \frac{(2m-i)!}{(2m)!} \frac{m!}{i!(m-i)!}, \tag{3.17}$$

we obtain a special case of the modified Obreshkov formula

$$\sum_{i=0}^{m} \alpha_{i,m} (-1)^i h^i x_{n+1}^{(i)} = \sum_{i=0}^{m} \alpha_{i,m} h^i x_n^{(i)}. \tag{3.18}$$

(3.18) was proposed by Obreshkov in 1942 [23], thus, is called Obreshkov formula. In the following subsections, we discuss the advantages introduced by the Obreshkov formula (3.18) and the difficulties in the practical applications.

3.3.1 Stability properties

In studying the stability properties of the Obreshkov method (3.18), Ehle [17] noted that the coefficients $\alpha_{i,m}$ are the coefficients of the $[m/m]$-diagonal Padé approximant to the exponential function. Denoting such approximant by $\mathcal{R}_{(m,m)}(q)$, then

$$\mathcal{R}_{(m,m)}(q) = \frac{\sum_{i=0}^{m} \alpha_{i,m} q^i}{\sum_{i=0}^{m} \alpha_{i,m} (-q)^i} \tag{3.19}$$

and the approximation error to the exponential function is given by

$$e^q - \mathcal{R}_{(m,m)}(q) = \underbrace{(-1)^m \frac{m!m!}{(2m)!(2m+1)!}}_{C_{(m,m)}} q^{2m+1} + O(q^{2m+2}). \tag{3.20}$$

Based on the fact that the coefficients in the Obreshkov formula are the same as those of the $[m/m]$ diagonal Padé approximant, x_{n+1} can be written in terms of x_n as follows

$$x_{n+1} = \mathcal{R}_{(m,m)}(h\lambda) x_n,$$

when applying the test equation $x' = \lambda x$ to the Obreshkov formula.

Birkhoff and Varga established that all diagonal Padé approximants to the exponential function satisfy the so-called A-stability criteria [25], that is

$$|\mathcal{R}_{(m,m)}(h\lambda)| \leq 1 \quad \text{for all } \Re e(\lambda) \leq 0, \quad m = 1,\ 2 \ldots.$$

Consequently, it follows that

$$|x_{n+1}| \leq |x_n| \quad \text{for all } \Re e(\lambda) \leq 0.$$

Hence, it was concluded that the Obreshkov formula is indeed A-stable.

3.3.2 Order of the method

Another major advantage that is offered by the Obreshkov formula is that it can be made to yield approximation of arbitrary order. This fact can be observed by noting that the order of the error is always equal to $2m + 1$. Hence, the order of approximation ($p = 2m$) can be increased arbitrarily by setting m as large as desired.

In this regard, one also must stress that using high order does not conflict with the A-stability requirement. This fact should be contrasted to the LMS methods, where A-stable methods cannot exceed order 2.

In comparison to the RK methods, the Obreshkov method has its coefficients in closed-form analytical expressions. To find the Butcher Tableau for a RK method requires solving equations with extremely complex nonlinear behavior [6].

3.3.3 Accuracy properties

Another important remark to be noted when comparing the Obreshkov single-step method to the class of GLM methods introduced in chapter 2 is related to the truncation error constant. It is easy to show that such a truncation error constant is obtained from

$$x(t_{n+1}) - x_{n+1} = \underbrace{(-1)^m \frac{m!m!}{(2m)!(2m+1)!}}_{C_{(m,m)}} h^{2m+1} x^{(2m+1)}(t_{n+1}) + O(h^{2m+2}).$$

Now compare the error constant $C_{(m,m)}$ with the lower bound of the error constant in a GLM method introduced in chapter 2. This comparison should then lead to

the following conclusion: a single-step Obreshkov method employing m implicit (i.e., unknown) derivatives $x_{n+1}^{(i)}$, $i = 1, \ldots, m$, has an error constant identical to the lowest possible error constant achieved by a GLM or a RK method with m implicit (unknown) stages. In other words, single-step Obreshkov method attains the most accurate results that can be achieved theoretically by any existing single-step or multi-step method with an equal number of unknowns.

3.3.4 Difficulties in the Obreshkov formula

Despite the potential advantages mentioned above, Ehle [17] remarked that this integration formula appears to be "*largely of theoretical interest*" due to the work to deal with implicit high-order derivatives. Consequently, interest in pursuing practical implementation of the Obreshkov formula in general nonlinear system waned. In fact, Gear in his classical work on BDF methods [4], notes that due to such difficulty "*the application of these methods is a major computing task for large systems and is not generally practical*". Such an attitude towards using Obreshkov-like formulas persisted over three decades with only one exception; when Nørsett attempted to modify them in an effort to reduce the computational difficulties in solving linear ordinary differential equations. The modified method, however, lost the A-stability property for orders higher than 5. In addition, its generalization to nonlinear DAE-base systems was not obvious [26].

It should be further noted that A-stability alone is not sufficient to guarantee the efficiency of the transient time-domain analysis. L-stability is also a crucial requirement, since nonlinear circuits are often modeled by stiff systems of differential

equations. Unfortunately, in spite of being A-stable, the Obreshkov formula (3.18) is not L-stable. This result follows directly from the properties of the diagonal Padé approximants [24], where $\lim_{\lambda\to\infty} |\mathcal{R}_{(m,m)}(h\lambda)| \to 1$.

In the following section, we demonstrate the slightly modified version Obreshkov formula (3.5) is an L-stable method. In other words, we show the condition that can be used to run the Obreshkov formula in an L-stable mode.

The difficulties related to handling the high-order derivatives and the solution of DAE systems are addressed in the following chapters.

3.4 Stability of the modified Obreshkov formula

Substituting the coefficients of α_i and β_i found in (3.7) and (3.8) into (3.5) shows that for the scalar test problem,

$$x_{n+1} = \mathcal{R}_{(l,m)}(h\lambda)x_n,$$

where $\mathcal{R}_{(l,m)}(h\lambda)$, defined in (3.12), is the general $[l/m]$ Padé approximant to the exponential function. Hence, one can use the general properties of the $\mathcal{R}_{(l,m)}(h\lambda)$ to establish the stability properties of the modified single-step Obreshkov method in (3.5). In particular, one should search for values of the integers l and m that makes $|\mathcal{R}_{(l,m)}(h\lambda)| \leq 1$ for all $\Re e(\lambda) \leq 0$.

Historically, there has been a conjecture about the values for l and m that will maintain $|\mathcal{R}_{(l,m)}(h\lambda)| \leq 1$ for all $\Re e\lambda \leq 0$. The first conjecture in this regard was the Ehle conjecture [27], which identified that

$$|\mathcal{R}_{(l,m)}(h\lambda)| \leq 1 \quad \text{for} \quad \Re e(\lambda) \leq 0 \quad \text{iff} \quad m-2 \leq l \leq m.$$

This conjecture, however, was finally proved by the introduction of the theory of order stars [28], [16], [9].

Given that

$$\lim_{\lambda\to\infty} \mathcal{R}_{(l,m)}(h\lambda) \to 0, \qquad l < m,$$

the modified Obreshkov formula becomes L-stable when $m-2 \leq l < m$. The following theorem summarizes the above observations.

Theorem 3.1 *The modified Obreshkov formula (3.5) is*

1. *A-stable iff* $m - 2 \leq l \leq m$;

2. *L-stable iff* $m - 2 \leq l < m$.

Proof The theorem is proved as follows. Substitute the test equation $x' = \lambda x$ into (3.5) and rearrange terms to obtain the rational amplification relationship

$$x_{n+1} = \mathcal{R}_{(l,m)}(h\lambda)x_n, \tag{3.21}$$

where

$$\mathcal{R}_{(l,m)}(h\lambda) = \frac{N_{lm}(h\lambda)}{N_{ml}(-h\lambda)} = \frac{\sum_{i=0}^{l} \beta_i (h\lambda)^i}{\sum_{i=0}^{m} \alpha_i (h\lambda)^i},$$

is the $[l/m]$ rational Padé approximant to the exponential function, $e^{h\lambda}$. The necessity and sufficiency for the A-stability part follows from the fact that $|\mathcal{R}_{(l,m)}(h\lambda)| \leq 1$ for all the $\Re e(\lambda) \leq 0$ if and only if $m - 2 \leq l \leq m$. The proof is provided by the theory of order stars [16].

In addition to A-stability, L-stability follows as a consequence of the fact when $l < m$,

$$\lim_{\lambda\to\infty} |\mathcal{R}_{(l,m)}(h\lambda)| \to 0,$$

because the degree of $N_{lm}(h\lambda)$ is lower than that of $N_{ml}(-h\lambda)$ in $\mathcal{R}_{(l,m)}(h\lambda)$, i.e., $l < m$. Hence, the modified Obreshkov formula is L-stable iff $m - 2 \leq l < m$. ■

One can use a quick way to sketch the stability domains for the modified single-step Obreshkov formula by forming a grid over a large part of the complex plane. Subsequently, $|\mathcal{R}_{(l,m)}(q_{ij})| \leq 1$ can be evaluated for each point q_{ij} on the grid and for all possible values of l and m. Points on the grid for which $|\mathcal{R}_{(l,m)}(q_{ij})| \leq 1$ are given a dark shade to mark them as a stability region; whereas points for which $|\mathcal{R}_{(l,m)}(q_{ij})| > 1$ are left white.

In particular, when $l = m - 1$ and $l = m - 2$, the modified Obreshkov methods which extend the backward Euler method, are L-stable methods. Their absolute stability regions are outside circles that lie in the right half plane, see Figures 3.1 and 3.2. When $l = m$, the Obreshkov methods, which extend the trapezoidal rule, have symmetric stability region, with the imaginary axis being the boundary; thus, they are A-stable methods.

3.5 Generalized multistep Obreshkov formula

In this section, we investigate how the single-step Obreshkov formula is extended to a multi-step method of the form

$$\sum_{l=0}^{k} \sum_{i=0}^{k_l} \kappa_{li} h^i x_{n+k-l}^{(i)} = 0, \tag{3.22}$$

where k represents the number of past steps. In particular, we are concerned about whether A-stable (and perhaps L-stable) methods can be identified within this general class of methods. To examine the stability properties of such methods, we substitute

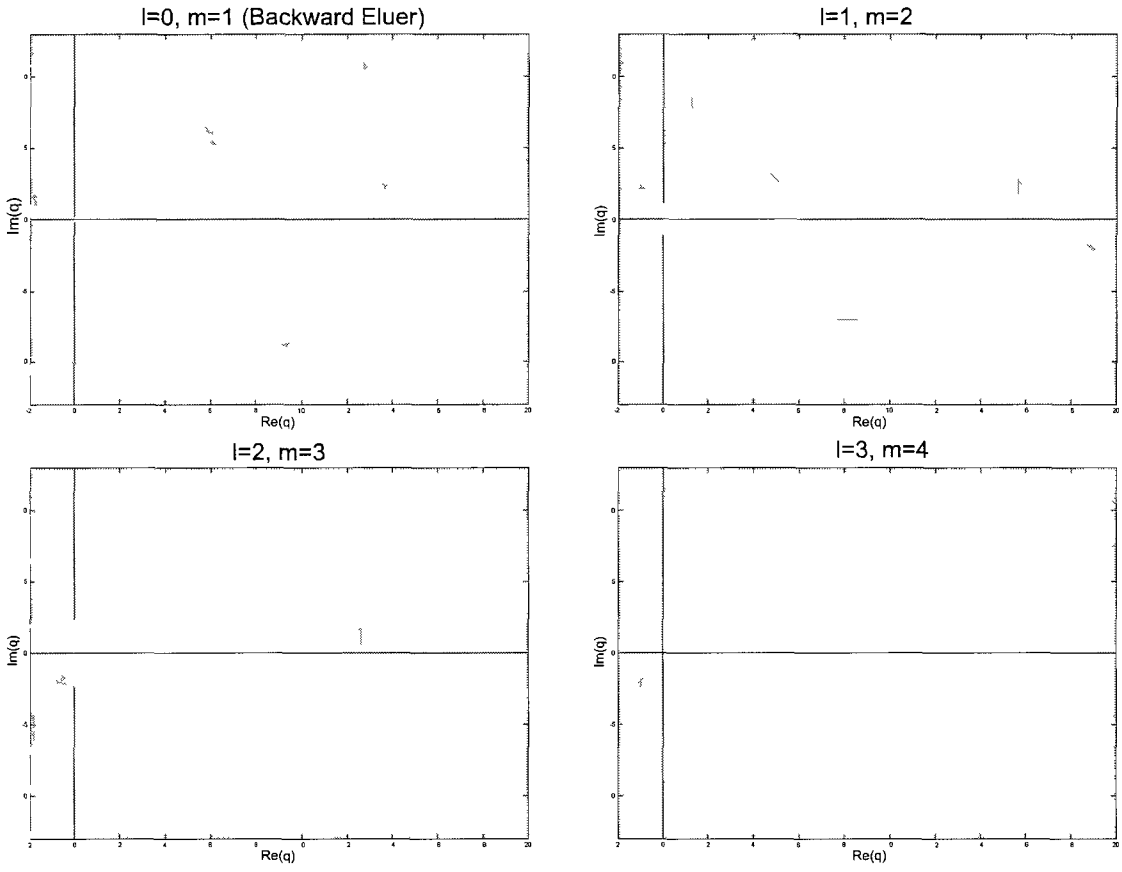

Figure 3.1: Absolute stability regions for the $l = m - 1$ Obreshkov methods (shaded area).

the scalar test problem $x' = \lambda x$ into (3.22) to obtain the following difference equation,

$$\sum_{l=0}^{k} \sum_{i=0}^{k_l} \kappa_{li} q^i x_{n+k-l} = 0, \tag{3.23}$$

where $q = h\lambda$. The solution of (3.23) is obtained by substituting $x_{n+k-l} = x(0)e^{q(n+k-l)}$ into (3.23) and looking for the z-roots of the characteristic polynomial

$$\Phi(z, q) \equiv \sum_{l=0}^{k} \sum_{i=0}^{k_l} \kappa_{li} q^i z^{(k-l)} = 0, \tag{3.24}$$

where $z = e^q$.

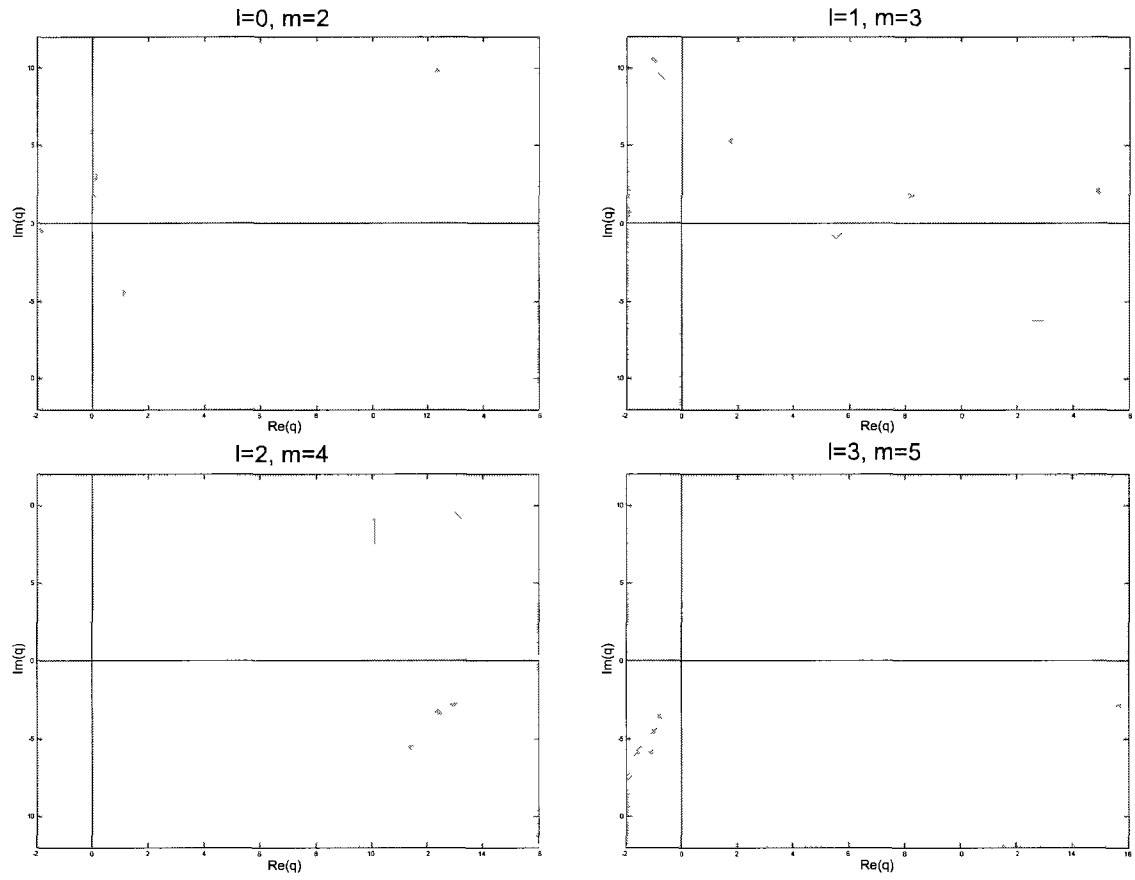

Figure 3.2: Absolute stability regions for the $l = m - 2$ Obreshkov methods (shaded area).

Therefore, a generalized multistep Obreshkov method (3.22) resembles a GLM method, in that it will be A-stable if all the z-roots of $\Phi(z, q)$ lie on or inside the unit circle for all $\Re e(\lambda) \leq 0$. Theorem 2.1 provides an even deeper insight as it shows that the highest attainable order by such an A-stable generalized Obreshkov method (if found) cannot exceed $2m$, with m being the number of implicit (unknown) high-order derivatives. Furthermore, the error constant by such an A-stable method (if it exists) cannot be lower than the limit in (2.49), which is equivalent to the error constant obtained from the single-step Obreshkov formula in (3.18).

Hence, we conclude that even though A-stable methods could potentially exist within the general class of multi-step Obreshkov formula (3.22), they will not offer higher order or better accuracy than what is offered by the single-step counterpart (3.18), with $l = m$.

It should be stressed that in the future, it may be possible to discover A-stable multi-step Obreshkov methods, whose error constants are smaller than those obtained from the single-step method (3.5) with $m - 2 \leq l < m$. These methods would therefore be expected to provide better accuracy (but not higher order) than a single-step having $m - 2 \leq l < m$. In that case, all the algorithm presented in this thesis would be carried to the general method with slight modifications.

Chapter 4

Step size and error control

This chapter presents the basic error control and step size adjustment mechanisms. The error estimation mechanism uses information obtained from the current point as well as from the past point to estimate the error and adjust the step size. This algorithm will be presented in section 4.2. To simplify notation, the following discussion is based on a single variable in the system.

4.1 General overview

Step size adjustment is necessary for efficient circuit simulation. Any integration method with a constant step size will perform poorly if the solution varies rapidly in some parts of the integration interval and slowly in other parts. When the waveform is smooth, a large step size is allowed to save computational cost. When the waveform changes fast, a smaller step size has to be used to keep the error within certain tolerance.

The integration process is kept local in time with all the information locally known.

Basically, given a user specified error tolerance ϵ_{tol}, the variable step size strategy attempts to keep the local truncation error (d_n) roughly equal to ϵ_{tol}

$$d_n \approx \epsilon_{\text{tol}}.$$

Recall from (2.10), the local error l_n is related to the local truncation error d_n by $|d_n| = |l_n|(1 + O(h_n))$. Thus, local error control and step size selection are sometimes viewed as controlling the local truncation error.

For the modified Obreshkov formula (3.5), the local truncation error is proportional to the $(l + m + 1)^{\text{th}}$ derivative,

$$d_n = C_{(l,m)} h_n^{l+m+1} x_n^{(l+m+1)}, \tag{4.1}$$

where h_n is the step size between time t_{n-1} and t_n, i.e.,

$$h_n = t_n - t_{n-1},$$

and $C_{(l,m)}$ is the error constant of the single-step modified Obreshkov method (3.5). This constant is given by

$$C_{(l,m)} = (-1)^m \frac{l!m!}{(l+m)!(l+m+1)!}.$$

It is obvious from (4.1) that any attempt to estimate the local truncation error must first be able to compute the $(l + m + 1)^{\text{th}}$ derivative. This issue will be discussed in section 4.2.

Thus, given a starting step size h_0, supplied by the user or determined heuristically, subsequent steps are adjusted according to the estimated local d_n.

After obtaining the local truncation error d_n. different actions are taken according to the user specified tolerance.

When the user specified tolerance ϵ_{tol1} is greater than the estimated d_n, i.e.,

$$\epsilon_{\text{tol1}} > d_n, \tag{4.2}$$

the algorithm accepts the current time step and moves to approximate the value of $x(t)$ at the future time point t_{n+1}.

If the waveform between the current time point t_n and the next time point t_{n+1} is smooth enough, the $(l+m+1)^{\text{th}}$ derivative will not be expected to change dramatically. Hence, d_{n+1}, the local error from t_n to t_{n+1} is expected to maintain the same value of d_n. This gives us the chance to find the step size to be taken from t_n to t_{n+1}, or $h_{n+1} = t_{n+1} - t_n$, which follows from

$$h_{n+1} = \frac{1}{\eta}\left(\frac{\epsilon_{\text{tol1}}}{d_n}\right)^{\frac{1}{l+m+1}} h_n, \tag{4.3}$$

where $\eta \geq 1$ is considered as a damping factor whose value is determined empirically.

Note here that since $\epsilon_{\text{tol1}} > d_n$, the next step size h_{n+1} will be generally larger than the previous step h_n. This is a desired behavior for a smooth waveform.

On the other hand, if the user specified tolerance ϵ_{tol2} is less than the estimated local error, i.e.,

$$\epsilon_{\text{tol2}} < d_n, \tag{4.4}$$

then the current step size is rejected. An alternative step size $\hat{h}_n$ is then considered,

$$\hat{h}_n = \frac{1}{\eta}\left(\frac{\epsilon_{\text{tol2}}}{d_n}\right)^{\frac{1}{l+m+1}} h_n. \tag{4.5}$$

The algorithm is run again to approximate the value of $x(t)$ at $t = t_{n-1} + \hat{h}_n$, instead of the one approximated at $t = t_{n-1} + h_n$.

Notice also that since $\epsilon_{\text{tol2}} < d_n$, we have $\hat{h}_n < h_n$, thus reducing the step size and controlling the error further by tightening h_n.

4.2 Computing the $(l+m+1)^{\text{th}}$ derivatives

The basic idea used to approximate the $(l+m+1)^{\text{th}}$ derivative can be summarized as follows. The $[l/m]$ Obreshkov method seeks to approximate the waveform $x(t)$ between t_n and t_{n+1} by a polynomial of degree $l+m$. This is the basis upon which we identified the Obreshkov method as a method of order $l+m$ with a truncation error proportional to h^{l+m+1}.

If the exact waveform $x(t)$ happened to be a polynomial of degree $l+m$, precisely for all value of t, then the method will obtain the exact waveform with no truncation error. Under this hypothetical situation, we must have

$$x^{(l+m+1)}(t) = 0 \text{ for all } t$$

and

$$x^{(l+m)}(t_i) = x^{(l+m)}(t_j) \text{ for all } t_i,\ t_j.$$

However, in practical circumstances $x(t)$ is not a polynomial of a fixed degree and thus

$$x^{(l+m+1)}(t) \neq 0$$

$$x^{(l+m)}(t_i) \neq x^{(l+m)}(t_j), \quad t_i \neq t_j.$$

Thus the only way we can approximate $x^{(l+m+1)}$ at $t = t_{n+1}$ is to use $x^{(l+m)}$ from two consecutive points in the following manner

$$x_{n+1}^{(l+m+1)} = \frac{x_{n+1}^{(l+m)} - x_n^{(l+m)}}{h_{n+1}}. \tag{4.6}$$

Indeed, for a hypothetical waveform given by a polynomial of degree $(l+m)$, the above formula should lead to a zero local truncation error indicating that the approximate x_{n+1} is the same as the exact one, $x(t_{n+1})$.

Hence, our goal in this section is to find $x^{(l+m)}$ at consecutive points.

Recall that the $[l/m]$ Obreshkov method needs l derivatives at the past point t_n and yields approximations to m derivatives at the current point t_{n+1}, in addition to the values at both points. Collect the two values and their derivatives in one vector, denoted by $\boldsymbol{z}_{n+1}$,

$$\boldsymbol{z}_{n+1} = [x_{n+1} \quad hx_{n+1}^{(1)} \quad \dots \quad h^m x_{n+1}^{(m)} \quad x_n \quad hx_n^{(1)} \quad \dots \quad h^l x_n^{(l)}]^{\mathrm{T}}.$$

and refer to this vector as the "stage vector". Our next goal is to use the components of $\boldsymbol{z}_{n+1}$ to approximate $x_{n+1}^{(l+m)}$, and then utilize two consecutive $\boldsymbol{z}_{n+1}$, $\boldsymbol{z}_n$ to approximate $x_{n+1}^{(l+m+1)}$.

Our approach is to take advantage of the fact that the method approximates $x(t)$ as an $(l+m)^{\text{th}}$-degree polynomial between t_n and t_{n+1}, uses this fact to compute the coefficients of the approximation polynomial and then uses these coefficients to approximate $x_{n+1}^{(l+m)}$ [2]. We show that, in this process, we do not need to explicitly compute the coefficients of the approximating polynomial.

To start the above process, define the approximating polynomial $x_p(t)$ of degree p between t_n and t_{n+1}

$$x_p(t) = \sum_{i=0}^{p} a_i \left(\frac{t_{n+1} - t}{h} \right)^i = \sum_{i=0}^{p} a_i \tau^i, \tag{4.7}$$

where h is defined as the current step size $h = t_{n+1} - t_n$, τ is the scaled time

$$\tau = \frac{t_{n+1} - t}{h}$$

and typically $p = l + m + 1$. We now need to find the coefficients a_i in (4.7) using the information available in the stage vector $\boldsymbol{z}_{n+1}$.

This can be done by simply taking the derivatives of $x_p(t)$ at time t,

$$x_p'(t) = -\frac{1}{h}\sum_{i=1}^{p} i a_i \left(\frac{t_{n+1}-t}{h}\right)^{i-1} = -\frac{1}{h}\sum_{i=1}^{p} i a_i \tau^{i-1}, \tag{4.8}$$

and in general,

$$x_p^{(k)}(t) = (-1)^k \frac{1}{h^k}\sum_{i=k}^{p} \frac{i!}{(i-k)!} a_i \left(\frac{t_{n+1}-t}{h}\right)^{i-k} = (-1)^k \frac{1}{h^k}\sum_{i=k}^{p} \frac{i!}{(i-k)!} a_i \tau^{i-k}, \tag{4.9}$$

where $k = 0, 1, 2, \ldots, p$. At the time of interest, $t = t_{n+1}$, where $\tau = 0$, we obtain

$$x_{n+1} = a_0, \tag{4.10}$$

and

$$h x'_{n+1} = -a_1, \tag{4.11}$$

and in general

$$h^k x^{(k)}_{n+1} = (-1)^k k! a_k, \quad k = 0, 1, \ldots, p. \tag{4.12}$$

Similarly, at $t = t_{n+1-j}$. where we define $\tau_j = (t_{n+1} - t_{n+1-j})/h$, we obtain

$$x_{n+1-j} = \sum_{i=0}^{p} a_i \tau_j^i, \tag{4.13}$$

and

$$h x'_{n+1-j} = -\sum_{i=1}^{p} i a_i \tau_j^{i-1}, \tag{4.14}$$

and in general

$$h^k x^{(k)}_{n+1-j} = (-1)^k \sum_{i=k}^{p} \frac{i!}{(i-k)!} a_i \tau_j^{i-k}, \quad k = 0, 1, \ldots, p. \tag{4.15}$$

Finding the coefficients of the approximating polynomial a_i can be done by arranging the equations from (4.10) to (4.12) and (4.13) to (4.15) for $j = 1$ in one system as

follows:

$$\begin{aligned}
a_0 &= x_{n+1} \\
-a_1 &= hx_{n+1}^{(1)} \\
(-1)^2 2! a_2 &= h^2 x_{n+1}^{(2)} \\
&\vdots \\
(-1)^m m! a_m &= h^m x_{n+1}^{(m)} \\
a_0 + a_1 + \cdots + a_{l+m+1} &= x_n \\
-a_1 - 2a_2 - \cdots - (l+m+1)a_{l+m+1} &= hx_n^{(1)} \\
&\vdots \\
(-1)^l l! a_l + (-1)^l (l+1)! a_{l+1} + \cdots + (-1)^l \tfrac{(l+m+1)!}{(m+1)!} a_{l+m+1} &= h^l x_n^{(l)}.
\end{aligned}$$

In matrix form, this is

$$\left[\begin{array}{ccccccc}
1 & 0 & \dots & & & & 0 \\
0 & -1 & 0 & \dots & & & 0 \\
0 & 0 & 2 & 0 & \dots & & 0 \\
\vdots & & \ddots & \ddots & \ddots & & \vdots \\
0 & \dots & & 0 & (-1)^m m! & 0 \dots & 0 \\
\hline
1 & 1 & \dots & & & & 1 \\
0 & -1 & -2 & \dots & & & -(l+m+1) \\
0 & 0 & 2 & 3 & \dots & & \frac{(l+m+1)!}{(l+m-1)!} \\
\vdots & & \ddots & \ddots & \ddots & & \vdots \\
0 & \dots & & 0 & (-1)^l l! & \dots & (-1)^l \frac{(l+m+1)!}{(m+1)!}
\end{array}\right]
\left[\begin{array}{c} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_m \\ \hline a_{m+1} \\ \\ \vdots \\ \\ a_{l+m+1} \end{array}\right]
=
\underbrace{\left[\begin{array}{c} x_{n+1} \\ hx_{n+1}^{(1)} \\ h^2 x_{n+1}^{(2)} \\ \vdots \\ h^m x_{n+1}^{(m)} \\ \hline x_n \\ hx_n^{(1)} \\ h^2 x_n^{(2)} \\ \vdots \\ h^l x_n^{(l)} \end{array}\right]}_{\boldsymbol{z}_{n+1}}, \tag{4.16}$$

or more concisely

$$\boldsymbol{V}\boldsymbol{a} = \boldsymbol{z}_{n+1}. \tag{4.17}$$

The solution is

$$\boldsymbol{a} = \boldsymbol{V}^{-1}\boldsymbol{z}_{n+1}. \tag{4.18}$$

Note that setting $k = l + m$ in (4.12)

$$h^{l+m}x_{n+1}^{(l+m)} = (-1)^{l+m}(l+m)!a_{l+m}$$

provides the key to find $h^{l+m}x_{n+1}^{(l+m)}$. Defining the vector

$$\boldsymbol{e}^{\mathrm{T}} = [0 \;\; 0 \;\; \ldots \;\; 0 \;\; (-1)^{l+m}(l+m)! \;\; 0], \tag{4.19}$$

and premultiplying $\boldsymbol{a}$ by $\boldsymbol{e}^{\mathrm{T}}$, we obtain

$$h^{l+m}x_{n+1}^{(l+m)} = (-1)^{l+m}(l+m)!a_{l+m} = \boldsymbol{e}^{\mathrm{T}}\boldsymbol{a} = \boldsymbol{e}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{z}_{n+1}. \tag{4.20}$$

Define in (4.20) an auxiliary vector $\boldsymbol{\varphi}$ as

$$(\boldsymbol{\varphi})^{\mathrm{T}} = \boldsymbol{e}^{\mathrm{T}}\boldsymbol{V}^{-1},$$

which can be rewritten as

$$\boldsymbol{V}^{\mathrm{T}}\boldsymbol{\varphi} = \boldsymbol{e}. \tag{4.21}$$

Solve (4.21) and insert the solution $\boldsymbol{\varphi}$ into (4.20) to obtain

$$h^{l+m}x_{n+1}^{(l+m)} = \boldsymbol{\varphi}^{\mathrm{T}}\boldsymbol{z}_{n+1}. \tag{4.22}$$

Thus, we see that $h^{l+m}x_{n+1}^{(l+m)}$ is a linear combination of all the known components in $\boldsymbol{z}_{n+1}$ with constant coefficients $\boldsymbol{\varphi}$. The local truncation error is computed by using

two consecutive estimates for $h^{l+m}x^{(l+m)}$ from two consecutive time points as follows

$$\begin{aligned} d_{n+1} &= C_{(l,m)}h^{l+m+1}x_{n+1}^{(l+m+1)} \\ &= C_{(l,m)}h^{l+m+1}\left(\frac{x_{n+1}^{(l+m)} - x_n^{(l+m)}}{h}\right) \\ &= C_{(l,m)}\left(h^{l+m}x_{n+1}^{(l+m)} - h^{l+m}x_n^{(l+m)}\right) \\ &= C_{(l,m)}\boldsymbol{\varphi}^{\mathrm{T}}\boldsymbol{z}_{n+1} - C_{(l,m)}\left(\frac{h}{h_n}\right)^{l+m}\boldsymbol{\varphi}^{\mathrm{T}}\boldsymbol{z}_n, \end{aligned} \tag{4.23}$$

where $C_{(l,m)}$ is the error constant of the integration method and $h_n = t_n - t_{n-1}$. The scaling factor $(h/h_n)^{l+m}$ is due to the fact that $\boldsymbol{z}_{n+1-j}$ is scaled by the step size h_{n+1-j} ($h_{n+1-j} = x_{n+1-j} - x_{n-j}$) at each time step.

The operation in (4.23) is repeated for all the variables in the system and a vector of local truncation error is formed. The step size is controlled by the norm of the local truncation error vector.

The Nordsieck representation

Note that in the above formulation, we did not have to explicitly compute the coefficients, a_0, a_1, ..., a_{l+m+1}, of the approximating polynomial, $x_p(t)$, to arrive at the evaluation of the local truncation error.

An alternative formulation could be achieved by using the notion of the Nordsieck vector [29], at a given point, which is defined by

$$\boldsymbol{y}_{n+1} = \left[x_{n+1} \quad hx_{n+1}^{(1)} \quad \cdots \quad \frac{h^{l+m+1}}{(l+m+1)!}x_{n+1}^{(l+m+1)}\right]^{\mathrm{T}}.$$

It is easy to deduce that the Nordsieck vector can be obtained from the stage vector via the following transformation,

$$\boldsymbol{y}_{n+1} = \boldsymbol{W}\boldsymbol{a} = \boldsymbol{W}(\boldsymbol{V})^{-1}\boldsymbol{z}_{n+1} = \boldsymbol{K}\boldsymbol{z}_{n+1}, \tag{4.24}$$

where $\boldsymbol{W}$ is a diagonal matrix having alternating ± 1 on the diagonal entries, with $+1$ being the first entry. The matrix $\boldsymbol{K}$ is a constant matrix and only the row before the last one, $\boldsymbol{K}_{l+m}$, is needed to obtain $h^{l+m}x_{n+1}^{(l+m)}$. $\boldsymbol{K}_{l+m}$ and $\boldsymbol{\varphi}$ have the following relationship

$$\boldsymbol{K}_{l+m} = \frac{\boldsymbol{\varphi}}{(l+m)!}.$$

Chapter 5

Implementation to circuits

The modified Obreshkov formula can be applied to both linear and nonlinear circuits. We first introduce its application to linear circuits. After we finish the fundamental discussion of the implementation to linear circuits, we proceed to more details on the implementation to nonlinear circuits and algorithm initialization.

5.1 Implementation to linear circuits

A general linear circuit can be described in the time domain using the Modified Nodal Analysis (MNA) [30],

$$\boldsymbol{G}\boldsymbol{x}(t) + \boldsymbol{C}\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{b}(t), \tag{5.1}$$

where $\boldsymbol{G}$ and $\boldsymbol{C} \in \mathbb{R}^{N\times N}$ are matrices describing the memoryless and memory elements in the circuit, respectively, $\boldsymbol{x}(t) \in \mathbb{R}^N$ is a vector of the node voltages appended by currents in inductors, independent voltage sources and controlled sources, and $\boldsymbol{b}(t) \in \mathbb{R}^N$ is the vector of independent stimuli to the circuit.

At time $t = t_n$, a step size h is taken to proceed to $t = t_{n+1}$. At $t = t_{n+1}$, discritizing the system of differential equations (5.1) and its derivatives w.r.t. t up to $(m-1)^{\text{th}}$ order by using the modified Obreshkov formula (3.5). we obtain a system of discretized difference equations as follows:

$$\begin{aligned}
\boldsymbol{G}\boldsymbol{x}_{n+1} + \tfrac{\boldsymbol{C}}{h}(h\boldsymbol{x}^{(1)}_{n+1}) &= \boldsymbol{b}_{n+1} \\
\boldsymbol{G}(h\boldsymbol{x}^{(1)}_{n+1}) + \tfrac{\boldsymbol{C}}{h}(h^2\boldsymbol{x}^{(2)}_{n+1}) &= h\boldsymbol{b}^{(1)}_{n+1} \\
&\vdots \\
\boldsymbol{G}(h^{m-1}\boldsymbol{x}^{(m-1)}_{n+1}) + \tfrac{\boldsymbol{C}}{h}(h^m\boldsymbol{x}^{(m)}_{n+1}) &= h^{m-1}\boldsymbol{b}^{(m-1)}_{n+1} \\
\alpha_0\boldsymbol{x}_{n+1} + \alpha_1(h\boldsymbol{x}^{(1)}_{n+1}) + \cdots + \alpha_m(h^m\boldsymbol{x}^{(m)}_{n+1}) &= \textstyle\sum_{k=0}^{l}\beta_k(h^k\boldsymbol{x}^{(k)}_n),
\end{aligned} \tag{5.2}$$

where $h = t_{n+1} - t_n$, $\boldsymbol{x}^{(k)}_{n+1}$ and $\boldsymbol{x}^{(k)}_n$ are the discretized values/derivatives at time $t = t_{n+1}$ and $t = t_n$. (5.2) can be summarized in the following matrix form

$$\begin{bmatrix}
\boldsymbol{G} & \boldsymbol{C}/h & & & \\
 & \boldsymbol{G} & \boldsymbol{C}/h & & \\
 & & \ddots & \ddots & \\
 & & & \boldsymbol{G} & \boldsymbol{C}/h \\
\alpha_0\boldsymbol{I} & \alpha_1\boldsymbol{I} & \dots & & \alpha_m\boldsymbol{I}
\end{bmatrix}
\begin{bmatrix}
\boldsymbol{x}_{n+1} \\ h\boldsymbol{x}^{(1)}_{n+1} \\ \vdots \\ h^{m-1}\boldsymbol{x}^{(m-1)}_{n+1} \\ h^m\boldsymbol{x}^{(m)}_{n+1}
\end{bmatrix}
=
\begin{bmatrix}
\boldsymbol{b}_{n+1} \\ h\boldsymbol{b}^{(1)}_{n+1} \\ \vdots \\ h^{m-1}\boldsymbol{b}^{(m-1)}_{n+1} \\ \sum_{k=0}^{l}\beta_k(h^k\boldsymbol{x}^{(k)}_n)
\end{bmatrix}, \tag{5.3}$$

where $\boldsymbol{I}$ is the identity matrix of size N and $\boldsymbol{b}^{(k)}_{n+1} = \frac{d^k}{dt^k}\boldsymbol{b}(t)$. More concisely, we have

$$(\tilde{\boldsymbol{G}} + \tilde{\boldsymbol{C}})\boldsymbol{\xi}_{n+1} = \tilde{\boldsymbol{b}}_{n+1}. \tag{5.4}$$

The matrices $\tilde{\boldsymbol{G}}$ and $\tilde{\boldsymbol{C}} \in \mathbb{R}^{(m+1)N\times(m+1)N}$ are obtained from $\boldsymbol{G}$ and $\boldsymbol{C}$ as follows

$$\tilde{\boldsymbol{G}} = \begin{bmatrix} \boldsymbol{G} & 0 & & & \\ 0 & \boldsymbol{G} & 0 & & \\ & \ddots & \ddots & \ddots & \\ & & 0 & \boldsymbol{G} & 0 \\ \alpha_0\boldsymbol{I} & \alpha_1\boldsymbol{I} & \dots & & \alpha_m\boldsymbol{I} \end{bmatrix} \quad \text{and} \quad \tilde{\boldsymbol{C}} = \begin{bmatrix} 0 & \boldsymbol{C}/h & 0 & & \\ & 0 & \boldsymbol{C}/h & 0 & \\ & & \ddots & \ddots & \ddots \\ & & & 0 & \boldsymbol{C}/h \\ 0 & 0 & \dots & & 0 \end{bmatrix}.$$

The vector $\boldsymbol{\xi}_{n+1}$ and $\tilde{\boldsymbol{b}}_{n+1} \in \mathbb{R}^{(m+1)N}$ are defined by

$$\boldsymbol{\xi}_{n+1} = \begin{bmatrix} \boldsymbol{x}_{n+1} \\ h\boldsymbol{x}^{(1)}_{n+1} \\ \vdots \\ h^{m-1}\boldsymbol{x}^{(m-1)}_{n+1} \\ h^{m}\boldsymbol{x}^{(m)}_{n+1} \end{bmatrix} \quad \text{and} \quad \tilde{\boldsymbol{b}}_{n+1} = \begin{bmatrix} \boldsymbol{b}_{n+1} \\ h\boldsymbol{b}^{(1)}_{n+1} \\ \vdots \\ h^{m-1}\boldsymbol{b}^{(m-1)}_{n+1} \\ \sum_{k=0}^{l}\beta_k(h^k\boldsymbol{x}^{(k)}_n) \end{bmatrix}.$$

The derivatives are scaled by the step size h. The system of difference equations (5.4) is linear and the solution can be obtained by inverting $(\tilde{\boldsymbol{G}} + \tilde{\boldsymbol{C}})$ directly,

$$\boldsymbol{\xi}_{n+1} = (\tilde{\boldsymbol{G}} + \tilde{\boldsymbol{C}})^{-1}\tilde{\boldsymbol{b}}_{n+1}. \tag{5.5}$$

The increase in the cost of the LU factorization in the resulting matrix (5.4) is typically offset by the expected savings in the number of time points needed as a result of using high-order method.

Theorem 5.1 *The proposed method (5.4) is*

1. *A-stable iff* $m - 2 \le l \le m$;

2. *L-stable iff* $m - 2 \le l < m$;

3. $\boldsymbol{x}_{n+1}^{(i)}$ *provides an order* $l+m$ *approximation to the derivatives* $\boldsymbol{x}^{(i)}(t)$, $i = 1, 2, \ldots, m$.

Proof For the scalar test problem, we have $\boldsymbol{C} = 1$, $\boldsymbol{G} = -\lambda$ and $\boldsymbol{b}(t) = 0$. Thus using (5.4) we obtain

$$x_{n+1} = \boldsymbol{d}^{\mathrm{T}} \boldsymbol{A}^{-1} \tilde{\boldsymbol{b}}_{n+1}, \tag{5.6}$$

where $\boldsymbol{d} \in \mathbb{R}^{m+1}$ is a vector with unity in its first component but zeros otherwise, and $\boldsymbol{A} = \tilde{\boldsymbol{G}} + \tilde{\boldsymbol{C}}$ with determinant given by

$$\det(\boldsymbol{A}) = \sum_{i=0}^{m} (-1)^m \alpha_i \lambda^i h^{i-m}, \tag{5.7}$$

whereas the $(m+1)^{\text{th}}$ column in its adjugate matrix is given by

$$(-1)^m [h^{-m} \ \lambda h^{-m+1} \ \ldots \ \lambda^{m-1} h^{-1} \ \lambda^m]^{\mathrm{T}}. \tag{5.8}$$

Noticing that

$$\tilde{\boldsymbol{b}}_{n+1} = \left[0 \ 0 \ldots \ 0 \ \sum_{k=0}^{l} \beta_k (h^k x_n^{(k)}) \right]^{\mathrm{T}},$$

we obtain

$$x_{n+1} = \mathcal{R}_{(l,m)}(h\lambda) x_n, \tag{5.9}$$

where $\mathcal{R}_{(l,m)}(h\lambda)$ defined in (3.12) is the $[l/m]$ Padé approximant to the exponential function. By Theorem 3.1, this proves the first and the second part of the theorem. Furthermore, the rest of the components in the solution vector $\boldsymbol{\xi}_{n+1}$ satisfy

$$x_{n+1}^{(i)} = \lambda^i \mathcal{R}_{(l,m)}(h\lambda) x_n, \tag{5.10}$$

thereby proving the third part of the theorem. ■

5.2 Implementation to nonlinear circuits

For nonlinear circuits, the MNA formulation is modified by appending the left side of (5.1) with a nonlinear vector $\boldsymbol{f}(\boldsymbol{x}(t)) \in \mathbb{R}^N$, which represents the currents of nonlinear conductances and the nonlinear entries corresponding to nonlinear capacitors and inductors:

$$\boldsymbol{G}\boldsymbol{x}(t) + \boldsymbol{C}\frac{d\boldsymbol{x}(t)}{dt} + \boldsymbol{f}(\boldsymbol{x}(t)) = \boldsymbol{b}(t). \tag{5.11}$$

Similarly, we discritize (5.11) and its derivatives w.r.t. t up to order $(m-1)$ by using the modified Obreshkov formula. Then the system of difference equations are shown in the following matrix form:

$$\begin{bmatrix} \boldsymbol{G} & \boldsymbol{C}/h & & & \\ & \boldsymbol{G} & \boldsymbol{C}/h & & \\ & & \ddots & \ddots & \\ & & & \boldsymbol{G} & \boldsymbol{C}/h \\ \alpha_0\boldsymbol{I} & \alpha_1\boldsymbol{I} & \dots & & \alpha_m\boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_{n+1} \\ h\boldsymbol{x}_{n+1}^{(1)} \\ \vdots \\ h^{m-1}\boldsymbol{x}_{n+1}^{(m-1)} \\ h^m\boldsymbol{x}_{n+1}^{(m)} \end{bmatrix} + \begin{bmatrix} \boldsymbol{f}_{n+1} \\ h\boldsymbol{f}_{n+1}^{(1)} \\ \vdots \\ h^{m-1}\boldsymbol{f}_{n+1}^{(m-1)} \\ 0 \end{bmatrix} = \begin{bmatrix} \boldsymbol{b}_{n+1} \\ h\boldsymbol{b}_{n+1}^{(1)} \\ \vdots \\ h^{m-1}\boldsymbol{b}_{n+1}^{(m-1)} \\ \sum_{k=0}^{l}\beta_k(h^k\boldsymbol{x}_n^{(k)}) \end{bmatrix}, \tag{5.12}$$

where $\boldsymbol{f}_{n+1}^{(k)} = \frac{d^k}{dt^k}\boldsymbol{f}(\boldsymbol{x}(t))\big|_{t=t_{n+1}}$. A more concise form of the difference equations is as follows:

$$(\tilde{\boldsymbol{G}} + \tilde{\boldsymbol{C}})\boldsymbol{\xi}_{n+1} + \tilde{\boldsymbol{f}}_{n+1} = \tilde{\boldsymbol{b}}_{n+1}, \tag{5.13}$$

where $\tilde{\boldsymbol{G}}$, $\tilde{\boldsymbol{C}}$, $\boldsymbol{\xi}_{n+1}$ and $\tilde{\boldsymbol{b}}_{n+1}$ are defined in (5.4), whereas $\tilde{\boldsymbol{f}}_{n+1}$ is given by

$$\tilde{\boldsymbol{f}}_{n+1} = \begin{bmatrix} \boldsymbol{f}_{n+1} \\ h\boldsymbol{f}_{n+1}^{(1)} \\ \vdots \\ h^{m-1}\boldsymbol{f}_{n+1}^{(m-1)} \\ 0 \end{bmatrix}.$$

5.3 Computing nonlinear derivatives

The solution of (5.13) requires Newton iterations because $\tilde{\boldsymbol{f}}_{n+1}$ is a nonlinear function of $\boldsymbol{\xi}_{n+1}$. This poses two major difficulties: evaluating $\tilde{\boldsymbol{f}}_{n+1}$ given a trial solution vector $\boldsymbol{\xi}_{n+1}$ and evaluating the Jacobian matrix consisting of the partial derivatives, $\partial\tilde{\boldsymbol{f}}_{n+1}/\partial\boldsymbol{\xi}_{n+1}$. It is possible to deploy some specialized automatic differentiation (AD) software for computing $\tilde{\boldsymbol{f}}_{n+1}$ [31]. However, AD becomes very cumbersome for computing high-order derivatives especially for the complex nonlinear expressions that are often used in modeling nonlinear devices. In addition, it cannot provide the elements of the Jacobian matrix. This section presents a rooted tree based approach to handle these tasks.

5.3.1 Rooted tree structure

The basic approach used here relies on representing each nonlinear expression entry in $\boldsymbol{f}(\boldsymbol{x}(t))$ as a rooted tree with the sub-expressions as children tree nodes. Each tree node represents a linear or nonlinear function. The leaf nodes in such a tree are nodes without descendants or children, which represent the MNA variables in the nonlinear

function (denoted by wave terms) or simple constant terms.

For example, consider that the q^{th} entry in $\boldsymbol{f}(\boldsymbol{x}(t))$ is due to a diode current, $f(\boldsymbol{x}(t)) = I_0\left(\exp\left(\frac{\boldsymbol{x}_q(t)-\boldsymbol{x}_p(t)}{V_T}\right) - 1\right)$, then such an expression is represented by the rooted tree shown in Figure 5.1.

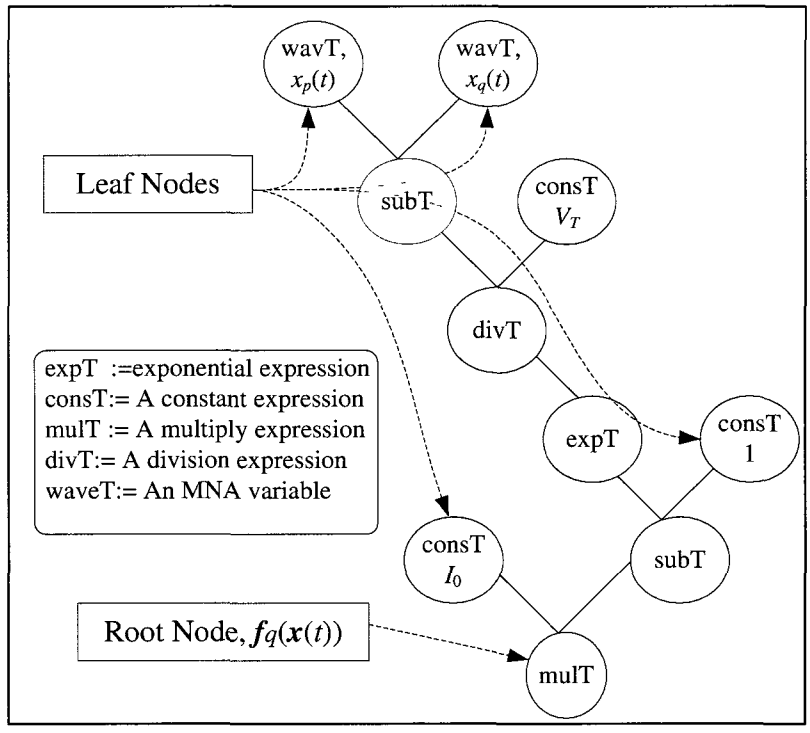

Figure 5.1: A rooted tree representing the diode current, wave terms and constant terms are the leaf nodes.

In our tree, the parent nodes represent basic operations, such as exponential function, logarithm, power, addition, subtraction, multiplication, etc. The parent nodes (including root node) have one or two child nodes. Nodes at the lowest level of the tree are called leaf nodes. In our tree, only wave terms (MNA variables) and constant terms (constants) are leaf nodes. They are also referred to as terminal nodes. A par-

ent node itself can be viewed as a rooted sub-tree. Thus, each parent node represents a linear or nonlinear sub-expression with the child node(s) being its argument(s).

5.3.2 Computing derivatives $\tilde{\boldsymbol{f}}_{n+1}$

The evaluation of the i^{th} derivative $\boldsymbol{f}_{n+1}^{(i)} = \frac{d^i}{dt^i}\boldsymbol{f}(\boldsymbol{x}(t))\big|_{t=t_{n+1}}$ is based on the idea that the derivative of each basic operation (parent tree node) can be obtained provided that knowledge of its lower order derivatives and the j^{th} derivatives ($j = 0,\ 1,\ \ldots,\ i$) of its children sub-expressions (arguments) are readily available. This fact can be demonstrated by assuming a simple exponential expression $f(x(t)) = \exp(x(t))$, at the k^{th} entry in $\boldsymbol{f}(\boldsymbol{x}(t))$. Expand both $x(t)$ and $f(x(t))$ into Taylor series around $t = t_{n+1}$,

$$x(t) = \sum_{i=0} u_i \tau^i, \quad u_i = \frac{1}{i!}\frac{d^i}{dt^i}x(t)\bigg|_{t_{n+1}}, \tag{5.14}$$

$$\exp(x(t)) = \sum_{i=0} v_i \tau^i, \quad v_i = \frac{1}{i!}\frac{d^i}{dt^i}\exp(x(t))\bigg|_{t_{n+1}}, \tag{5.15}$$

where $\tau = t - t_{n+1}$. Thus, $f_{n+1}^{(i)} = v_i i!$, $i = 0, 1, \ldots, m-1$. Taking the first order derivative of $f(x(t))$, we obtain

$$\frac{d}{dt}\exp(x(t)) = \frac{\partial(\exp(x))}{\partial x}\frac{dx(t)}{dt} = \exp(x(t))\frac{dx(t)}{dt}. \tag{5.16}$$

The result in (5.16) is based on the fact that $\frac{\partial(\exp(x))}{\partial x} = \exp(x)$. Substituting (5.14) and (5.15) into (5.16), we have

$$\begin{aligned}\left(\sum_{i=0} v_i\tau^i\right)' &= \left(\sum_{i=0} v_i\tau^i\right)\left(\sum_{i=0} u_i\tau^i\right)' \\ \sum_{i=1} i v_i \tau^{i-1} &= \left(\sum_{i=0} v_i\tau^i\right)\left(\sum_{i=1} i u_i \tau^{i-1}\right).\end{aligned} \tag{5.17}$$

Expanding both sides of (5.17) and equating the coefficients of τ with similar powers, we obtain the following recursive relationship

$$\begin{aligned} v_0 &= \exp(u_0) \\ v_i &= \tfrac{1}{i}\sum_{k=0}^{i-1} v_k u_{i-k}(i-k), \quad i \geq 1. \end{aligned} \tag{5.18}$$

Similar recursive formulas for other basic functions (operations) can be derived [32] and are listed in Table 5.1.

Table 5.1: Recursive formulas for the derivatives of simple functions.

Equation	Terms $x = \sum_{i=0} u_i\tau^i$, $y = \sum_{i=0} w_i\tau^i$, $f = \sum_{i=0} v_i\tau^i$
$f = \exp(x)$	$v_0 = \exp(u_0)$ $v_i = 1/i \sum_{k=0}^{i-1} v_k u_{i-k}(i-k)$
$f = \log(x)$	$v_0 = \log(u_0)$ $v_i = (1/u_0)(u_i - \sum_{k=1}^{i-1} v_{i-k}u_k((i-k)/i))$
$f = x^p$	$v_0 = u_0^p$ $v_i = (pv_0u_ii + \sum_{k=1}^{i-1} u_k v_{i-k}(k(p+1)-i))/iu_0$
$f = x + y$	$v_i = u_i + w_i$
$f = x - y$	$v_i = u_i - w_i$
$f = xy$	$v_i = \sum_{k=0}^{i} u_k w_{i-k}$
$f = x/y$	$v_i = (u_i - \sum_{k=0}^{i-1} v_k w_{i-k})/w_0$
$f = \tanh(x)$	$v_0 = \tanh(u_0)$ $v_i = u_i - w_{i-1}/i$ where $w_0 = v_0^2 u_1$ $w_i = \sum_0^{i-1}\sum_{r=1}^{i-k} v_r v_k(i+1-k-r)u_{i+1-k-r} + \sum_{k=0}^{i} v_0 v_k(i+1-k)u_{i+1-k}$

$\tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$

Therefore, the steps used to compute the i^{th} derivative corresponding to $f(x(t))$ at the root of the tree can be summarized by the following steps.

1. Assign the h-scaled derivatives to the leaf nodes. Note that the derivatives of the leaf nodes are trivial if the leaf nodes represent constant terms. For leaf nodes representing wave terms, i.e., $x_k(t)$, $k = 1, 2, \ldots, N$, those derivatives are

set to the initial guess of Newton iterations (usually values from previous time point are used as the initial guess) and updated after each Newton iteration.

2. Start with the root node, the j^{th} order ($j = 1, 2, \ldots, i$) scaled derivatives of each child node are first computed and stored locally within each node.

3. The derivatives of the parent node are next obtained from the derivatives of the children and those locally stored derivatives of lower order using recursive formulas listed in Table 5.1.

5.3.3 Computing the Jacobian matrix, $\tilde{\boldsymbol{J}} = \partial \tilde{\boldsymbol{f}}_{n+1}/\partial \boldsymbol{\xi}_{n+1}$

Computing the Jacobian matrix involves computing the partial derivative of $\boldsymbol{f}^{(i)}(x(t))$ w.r.t. $\boldsymbol{x}^{(j)}(t)$ at $t = t_{n+1}$ for $i = 0,\ 1,\ \ldots,\ m-1$ and $j = 0,\ 1,\ \ldots,\ m-1$. To illustrate this computation, we reconsider the hypothetical situation of the previous section where we had an exponential function at the k^{th} entry in $\boldsymbol{f}(\boldsymbol{x}(t))$. Clearly, we now seek the following partial derivatives

$$\left.\frac{\partial v_i}{\partial u_j}\right|_{t=t_{n+1}} \quad \text{for } i,\ j = 0,\ \ldots,\ m-1,\ j \le i. \tag{5.19}$$

Finding the above partial derivatives can be achieved by differentiating the terms obtained in (5.18) for the exponential function to have

$$\frac{\partial v_0}{\partial u_0} = \exp(u_0) \tag{5.20}$$

$$\frac{\partial v_i}{\partial u_j} = \frac{1}{i}\left(\sum_{k=j}^{i-1}(i-k)\frac{\partial v_k}{\partial u_j}u_{i-k} + j v_{i-j}\right),\ 1 \le i \le m-1,\ 0 \le j \le i. \tag{5.21}$$

The above partial derivatives should be evaluated in the overall Jacobian matrix at entries corresponding to the k^{th} entry in the manner explained in Figure 5.2. Notice

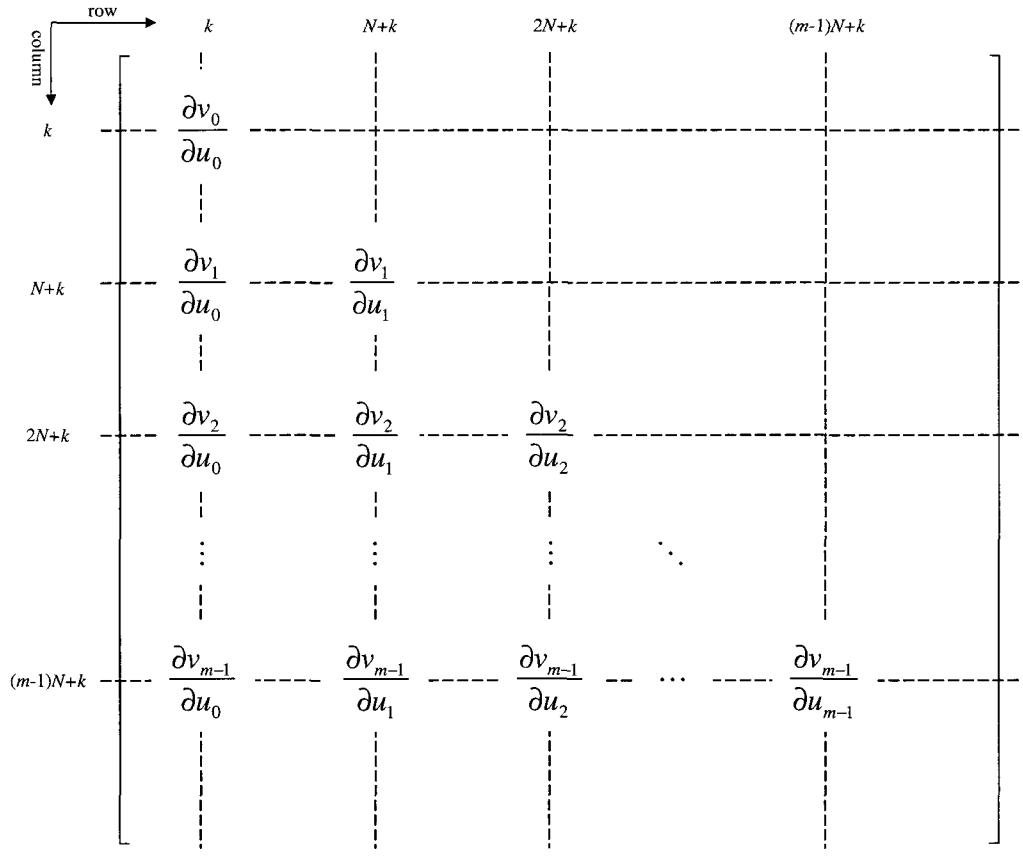

Figure 5.2: Jacobian entries contributed by the k^{th}-entry in $\boldsymbol{f}(\boldsymbol{x}(t))$.

that since $\boldsymbol{f}_{n+1}^{(i)}$ is only dependent on $\boldsymbol{x}_{n+1}^{(j)}$, where $j \leq i$, the resulting Jacobian entries contributed by the nonlinear functions $\boldsymbol{f}(\boldsymbol{x}(t))$ are always in lower block triangular matrix form, as shown in Figure 5.2. The underlying structural characteristics of the Jacobian matrix will be explored in chapter 6 to enable more efficient LU factorization techniques.

The notion of rooted tree was implemented using an object-oriented paradigm in C++. Each node in the tree is represented by an object of a certain class with local storage and suitable methods to handle its own derivatives as well as its children's.

We have also developed an automated tool with a front-end parser that reads in a script-like specification of the nonlinear model and translates it into the corresponding rooted tree. That automated tool is an essential component in making the proposed algorithm general enough to handle arbitrary nonlinear devices models. It also eliminates the inconvenient task of having to hard-code the computation of high-order derivatives for each user specified model.

5.4 Algorithm initialization

The solution of (5.4) and (5.13) requires the knowledge of $\boldsymbol{x}_0^{(i)}$, $i = 0, 1, \ldots, l$, in the vector $\tilde{\boldsymbol{b}}_1$ at $t = 0$. $\boldsymbol{x}_0$ is assumed to be available through a DC operating point analysis. The higher-order derivatives are computed as follows.

We first consider the situation where $\boldsymbol{C}$ matrix is invertible. Expanding terms in the MNA formulation (5.11) by their Taylor series provides the following recursive formula for computing $h^i \boldsymbol{x}_0^{(i)}$,

$$h^i \boldsymbol{x}_0^{(i)} = h\boldsymbol{C}^{-1}\left(h^{i-1}\boldsymbol{b}_0^{(i-1)} - \boldsymbol{G}(h^{i-1}\boldsymbol{x}_0^{(i-1)}) - h^{i-1}\boldsymbol{f}_0^{(i-1)}\right). \tag{5.22}$$

When the matrix $\boldsymbol{C}$ is not invertible, (5.22) can be partitioned into differential and algebraic sets of equations as follows,

$$\begin{bmatrix} \boldsymbol{C}_{11} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} h^i \boldsymbol{x}_{0,U}^{(i)} \\ h^i \boldsymbol{x}_{0,L}^{(i)} \end{bmatrix} = h \begin{bmatrix} h^{i-1}\boldsymbol{b}_{0,U}^{(i-1)} \\ h^{i-1}\boldsymbol{b}_{0,L}^{(i-1)} \end{bmatrix} - h \begin{bmatrix} \boldsymbol{G}_{11} & \boldsymbol{G}_{12} \\ \boldsymbol{G}_{21} & \boldsymbol{G}_{22} \end{bmatrix} \begin{bmatrix} h^{i-1}\boldsymbol{x}_{0,U}^{(i-1)} \\ h^{i-1}\boldsymbol{x}_{0,L}^{(i-1)} \end{bmatrix} - h \begin{bmatrix} h^{i-1}\boldsymbol{f}_{0\,U}^{(i-1)} \\ h^{i-1}\boldsymbol{f}_{0,L}^{(i-1)} \end{bmatrix}, \tag{5.23}$$

such that $\boldsymbol{C}_{11}$ and $\boldsymbol{G}_{22}$ are invertible [2], [33]. In the upper part of (5.23), $h^i \boldsymbol{x}_{0,U}^{(i)}$ is

linearly dependent on the lower-order derivatives. It is computed by inverting $\boldsymbol{C}_{11}$,

$$h^i \boldsymbol{x}_{0,U}^{(i)} = h\boldsymbol{C}_{11}^{-1}\left(h^{i-1}\boldsymbol{b}_{0,U}^{(i-1)} - \boldsymbol{G}_{11}(h^{i-1}\boldsymbol{x}_{0,U}^{(i-1)}) - \boldsymbol{G}_{12}(h^{i-1}\boldsymbol{x}_{0,L}^{(i-1)}) - h^{i-1}\boldsymbol{f}_{0,U}^{(i-1)}\right). \tag{5.24}$$

$\boldsymbol{C}_{11}$ is only inverted once to compute all the high-order derivatives. The lower part in (5.23) provides a nonlinear algebraic equation to compute the lower set of derivatives, $h^i \boldsymbol{x}_{0,L}^{(i)}$,

$$\boldsymbol{G}_{22}(h^i \boldsymbol{x}_{0,L}^{(i)}) + h^i \boldsymbol{f}_{0,L}^{(i)} = h^i \boldsymbol{b}_{0,L}^{(i)} - \boldsymbol{G}_{21}(h^i \boldsymbol{x}_{0,U}^{(i)}), \tag{5.25}$$

where $h^i \boldsymbol{x}_{0,U}^{(i)}$ is obtained from (5.24). As will be proved in section 6.4, (5.25) can be simplified as follows,

$$\left(\boldsymbol{G}_{22} + \tilde{\boldsymbol{J}}_{0,L}\right)(h^i \boldsymbol{x}_{0,L}^{(i)}) = h^i \boldsymbol{b}_{0,L}^{(i)} - \boldsymbol{G}_{21}(h^i \boldsymbol{x}_{0,U}^{(i)}) - \boldsymbol{\gamma}_{0,L}, \tag{5.26}$$

where $\tilde{\boldsymbol{J}}_{0,L} = \partial(h^i \boldsymbol{f}_{0,L}^{(i)})/\partial(h^i \boldsymbol{x}_{0,L}^{(i)}) = \partial \boldsymbol{f}_{0,L}^{(0)}/\partial \boldsymbol{x}_{0,L}^{(0)}$ is obtained during the DC operating point analysis. $\boldsymbol{\gamma}_{0,L}$ is contributed by the lower order terms in $h^i \boldsymbol{f}_{0,L}^{(i)}$ and is not dependent on $h^i \boldsymbol{x}^{(i)}$. Thus, only one LU factorization of $\boldsymbol{G}_{22} + \tilde{\boldsymbol{J}}_{0,L}$ and a few FB substitutions are needed to obtain the solution of (5.26) for all the high-order derivatives.

5.4.1 Numerical error in high-order derivatives

Although the above recursive formulas can be used to compute the first few (h-scaled) derivatives up to $i = 2, 3$, it quickly introduces huge numerical errors for progressively higher-order derivatives. The reason for that can be explained as follows. In the computation of low-order derivatives, i.e., when $i = 0, 1$, the finite machine precision typically adds a very small error due to the truncation of non-significant decimal

digits. However, this error, albeit being unnoticeable in low-order derivatives, gets amplified exponentially for increasingly higher-order due to the multiplication by $\boldsymbol{C}^{-1}$ (or $\boldsymbol{C}_{11}^{-1}$). The amplification factor gets even larger with the presence of parasitic memory elements whose relatively small values introduce very small eigenvalues in the matrix $\boldsymbol{C}$ ($\boldsymbol{C}_{11}$) causing the inverse to have large values. This situation is further illustrated by the following example.

Example The circuit shown in Figure 5.3 is a linear circuit with a sinusoidal input source. Therefore, its response at steady-state is also a sinusoidal waveform and can be computed using simple AC analysis at the stimulus frequency. To be able to compute the exact derivatives at any time t, we set the initial voltages/currents of the MNA variables to an arbitrary point on the steady-state trajectory. This setting allows computing the exact derivatives via the closed form Fourier series representation of the steady-state response.

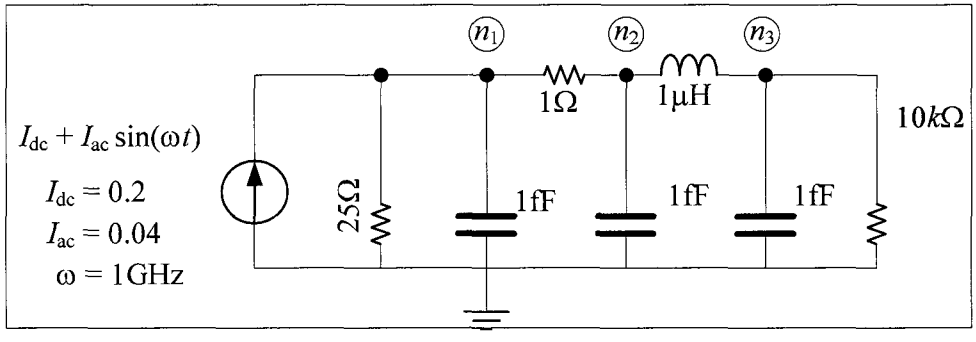

Figure 5.3: A schematic for a circuit used to illustrate the potential error in high-order derivatives.

Table 5.2 compares those exact derivatives (scaled by powers of h, with $h = 0.5$nS), to those obtained from the recursive formula in (5.22) for the voltage at node n_3,

indicating a huge error for high-order derivatives. To show the impact of the error in the initial stage vector on the algorithm performance, the proposed time-stepping algorithm was subsequently run for 50nS using a constant step size while disabling the error control mechanism. The resulting integration error is plotted by the upper curve in Figure 5.4. ■

Table 5.2: Comparing the h-scaled derivatives of V_{n_3} in Figure 5.3.

Order (i)	h-scaled derivatives $h^i x_0^{(i)}$ at n_3		
	Exact	From (5.22)	Proposed
0	**-4.99e+000**	-4.99e+000	**-4.99e+000**
1	**3.14e+000**	3.14e+000	**3.14e+000**
2	**3.94e-004**	3.94e-004	**3.94e-004**
3	**-3.10e+001**	-3.10e+001	**-3.10e+001**
4	**-3.89e-003**	1.74e+006	**-3.89e-003**
5	**3.06e+002**	-9.61e+013	**3.06e+002**
6	**3.84e-002**	5.28e+021	**3.84e-002**
7	**-3.02e+003**	-2.90e+029	**-3.02+003**

5.4.2 Proposed initialization procedure

One way to reduce the numerical error in the starting vector introduced by inverting $\boldsymbol{C}$ ($\boldsymbol{C}_{11}$) is to start the algorithm with a very small step size. Indeed, the step size will be automatically reduced when the step size control mechanism described in chapter 4 is being enabled. The error control mechanism effectively reduces the error amplification factor and could curb the propagation of error for higher order derivatives. Unfortunately, in many situations, it does not give satisfactory performance as it entails starting the time stepping algorithm with an excessively small size and the algorithm may never recover from.

To address this difficulty, a few options are available. The first option is to use an order ramping method to gradually establish the desired order $p = l+m$. For example, with the only information $\boldsymbol{x}_0$ available, we can choose $m = 2,\ l = 0$ to integrate the first time step at $t = h_1$. In the subsequent steps, the order is increased by setting $m = 4,\ l = 2$, and so forth until the desired integration order has been reached. One problem with this methodology is that during the order ramping process, the truncation error of the lower-order method is much larger than that of the high-order method at the same step size. Thus, a very small step size has to be chosen to get a matched truncation error.

Another possible method is to compute the scaled derivatives $h^i\boldsymbol{x}_0^{(i)}$ for $i = 1,\ 2$ by using the recursive formulas (5.22) or (5.23) at $t = 0$. With $\boldsymbol{x}_0$, $h\boldsymbol{x}_0^{(1)}$ and $h^2\boldsymbol{x}_0^{(2)}$ available, the order ramping method is adopted starting from $m = 4,\ l = 2$ to gradually increase the integration order.

It is possible to compute the desired high-order derivatives with considerable accuracy at the first time step $t = h_1$. This is achieved by integrating the first step using method $[(l + m)/0]$. The advantage is that method $[(l + m)/0]$ is as accurate as method $[l/m]$, in the sense of local truncation error because both methods are of order $p = l + m$. One issue with this approach lies in that method $[(l + m)/0]$ may not be an A-stable method. Noting that stability is a global property based on a period of integration, one step integration by a conditionally stable method should not affect the overall stability of method $[l/m]$.

To demonstrate the importance of the accurate initial vector, we reconsider the circuit shown in Figure 5.3. The last column in Table 5.2 shows the scaled derivatives at node n_3 obtained after the initial build-up and compares them with the exact

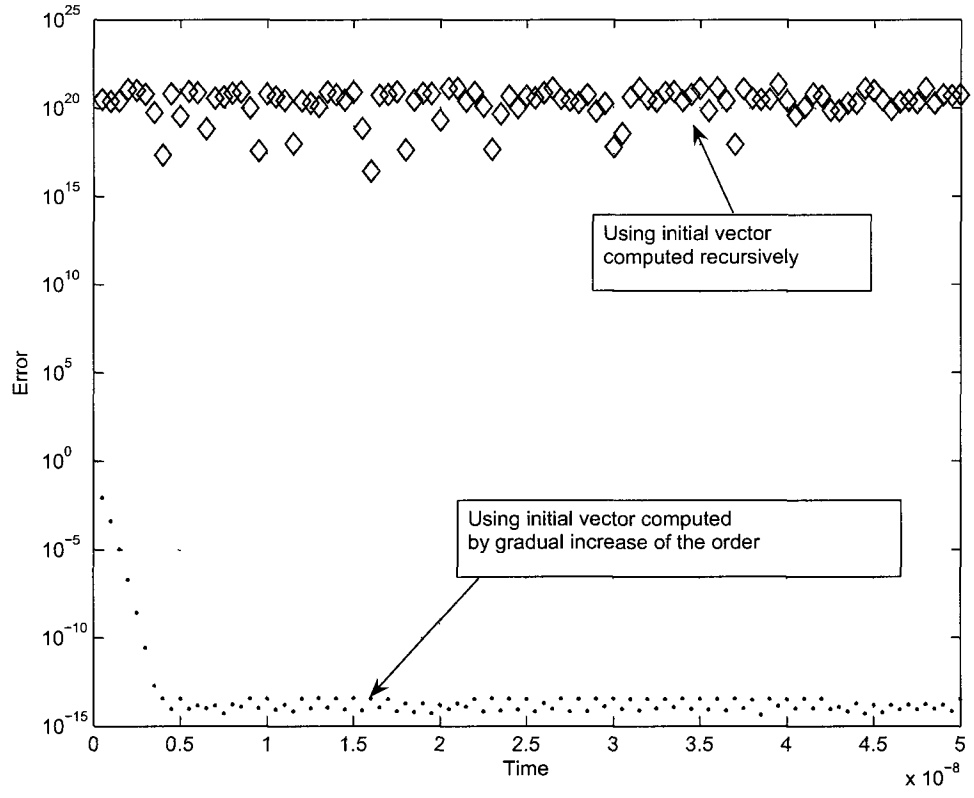

Figure 5.4: The integration error obtained with two different initial conditions.

scaled derivatives computed at the same time instant. As we can see from Table 5.2, the scaled derivatives obtained by the high order integration methods are accurate to at least 3 decimals. Also the lower curve in Figure 5.4 presents the error produced in the time-stepping starting with the initial vector computed using the proposed methodology.

5.5 Overall algorithm description

In this section, a piece of pseudo-code is provided to describe the general procedure of the proposed algorithm. A potential difficulty may arise if the input stimuli poses isolated points with discontinuous high-order derivatives, such as in the case of piecewise-linear functions. In such a situation, the proposed technique proceeds in a piecewise manner through implementation on the intervals where the input stimuli is locally smooth enough to provide the needed high-order derivatives. Terminal conditions within one interval provide the initial conditions to the subsequent interval. The algorithm in Figure 5.5 shows the computation procedure for one such interval.

TIME-STEPPING USING THE MODIFIED OBRESHKOV METHOD $[l/m]$

input: m, l, ϵ_{tol}, h_0, t_{final}, $\boldsymbol{x}_0$, $\boldsymbol{x}_0^{(1)}$, $\boldsymbol{x}_0^{(2)}$
initialization: $n \leftarrow 1$, $t_n \leftarrow 0$, $h \leftarrow h_0$.
if $l > 2$
  then
    comment: order ramping process
    repeat
      $\hat{m} \leftarrow 2$, $\hat{l} \leftarrow 0$
      repeat
        $\hat{l} \leftarrow \hat{m}$, $\hat{m} \leftarrow \hat{l} + 2$
        $t_{n+1} \leftarrow t_n + h$
        compute $\boldsymbol{\xi}_{n+1}$ in (5.13) using the modified Obreshkov method $[\hat{l}/\hat{m}]$
        $n \leftarrow n + 1$
      until $\hat{m} \geq l$
      estimate LTE d_{n+1} using methods in chapter 4
      if $d_{n+1} \leq \epsilon_{\text{tol}}$
        then update h using (4.3)
        else compute $\hat{h}$ using (4.5), $n \leftarrow 1$, $h \leftarrow \hat{h}$.
    until $d_{n+1} \leq \epsilon_{\text{tol}}$
    comment: end of order ramping process
while $t_{n+1} < t_{final}$
  do
    comment: start time stepping
    repeat
      $t_{n+1} \leftarrow t_n + h$
      compute $\boldsymbol{\xi}_{n+1}$ in (5.13) using the modified Obreshkov method $[l/m]$
      estimate LTE d_{n+1} using methods in chapter 4
      if $d_{n+1} \leq \epsilon_{\text{tol}}$
        then update h using (4.3), $n \leftarrow n + 1$
        else compute $\hat{h}$ using (4.5), $h \leftarrow \hat{h}$
    until $d_{n+1} \leq \epsilon_{\text{tol}}$
    comment: end of time stepping

Figure 5.5: A pseudo-code description for the time-stepping algorithm for nonlinear circuits.

Chapter 6

Improved implementation techniques

This chapter shows that the formulation of the high-order methods described in chapter 5 offers structural characteristics that, when exploited, can significantly improve the performance of the proposed method.

It is observed that the main computational overhead of high-order methods, as compared to the LMS methods such as BDF, is having to factorize a matrix that is $(m + 1)$ times larger than the Jacobian matrix of the LMS methods. In addition, the sparsity pattern encountered in the Jacobian matrix is different from the sparsity patterns arising in circuit simulations. Therefore, high-order Jacobian matrices do not lead themselves naturally to circuit-oriented sparse solver such as KLU [34].

Another computational burden in the proposed high-order methods is related to the number of iterations required by Newton–Raphson method in order to reach convergence. This is because the large step size afforded by the high-order methods

typically requires two or three iterations more than the number needed by the low-order small step size ones to converge to a solution. With the larger matrix size, these additional iterations become truly significant in the total computational efforts.

The goal of the following sections is to introduce more tailored techniques aimed at enabling high-order methods to attain their full potentials.

6.1 Structural characteristics

This section shows that the Jacobian matrix possesses a certain feature that considerably facilitates the LU factorization.

Solving the algebraic nonlinear equations in (5.13) is typically carried out through an iterative Newton-based technique, which requires factorize the Jacobian matrix at each iteration. For illustration purposes, the associated Jacobian matrix can be written as (6.1)

$$\tilde{\boldsymbol{J}} = \underbrace{\begin{bmatrix} \boldsymbol{G}+\boldsymbol{J}_{0,0} & \frac{1}{h}\boldsymbol{C}+\boldsymbol{J}_{0,1} & \boldsymbol{J}_{0,2} & \dots & \boldsymbol{J}_{0,m} \\ \boldsymbol{J}_{1,0} & \boldsymbol{G}+\boldsymbol{J}_{1,1} & \frac{1}{h}\boldsymbol{C}+\boldsymbol{J}_{1,2} & \dots & \boldsymbol{J}_{1,m} \\ \vdots & & \ddots & & \vdots \\ \boldsymbol{J}_{m-1,0} & \boldsymbol{J}_{m-1,1} & \dots & & \frac{1}{h}\boldsymbol{C}+\boldsymbol{J}_{m-1,m} \\ \alpha_0\boldsymbol{I} & \alpha_1\boldsymbol{I} & \dots & & \alpha_m\boldsymbol{I} \end{bmatrix}}_{(m+1)\times(m+1)\text{ blocks}}, \tag{6.1}$$

where $\boldsymbol{J}_{i,j} = \frac{\partial \boldsymbol{f}_{n+1}^{(i)}}{\partial \boldsymbol{x}_{n+1}^{(j)}} h^{i-j}$. It can easily be shown that

$$\boldsymbol{J}_{i,j} = 0 \quad \text{for} \quad i < j.$$

This fact makes the Jacobian matrix a lower block Hessenberg matrix.

We expand the Jacobian matrix $\boldsymbol{J}_{0,0} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\big|_{t=t_{n+1}}$ in a Taylor series in t at $t = t_{n+1}$. Hence,

$$\boldsymbol{J}_{0,0} = \sum_{u=0}^{\infty} \frac{1}{u!} \boldsymbol{R}_{n+1}^{(u)} \tau^u, \tag{6.2}$$

where $\tau = t - t_{n+1}$ and $\boldsymbol{R}_{n+1}^{(u)} = \frac{\partial^u \boldsymbol{J}_{0\,0}}{\partial t^u}\big|_{t=t_{n+1}}$. Define

$$\boldsymbol{R}_u = \frac{1}{u!} \boldsymbol{R}_{n+1}^{(u)} \tag{6.3}$$

as the matrix-valued u^{th} Taylor series coefficient of $\boldsymbol{J}_{0,0}$. Since $\boldsymbol{f}(\boldsymbol{x}(t))$ is continuously differentiable with respect to both t and $\boldsymbol{x}^{(\imath)}$, differentiation with respect to t and $\boldsymbol{x}^{(\imath)}$ commutes. We can therefore proceed with the following manipulation [35]:

$$\begin{aligned}
\boldsymbol{J}_{\imath,\jmath} &= \frac{\partial \boldsymbol{f}^{(\imath)}}{\partial \boldsymbol{x}^{(\jmath)}} h^{\imath-\jmath} \\
&= h^{\imath-\jmath} \frac{\partial}{\partial \boldsymbol{x}^{(\jmath)}} \left[\frac{\partial^\imath}{\partial t^\imath} \boldsymbol{f}(\boldsymbol{x}(t)) \right] \Bigg|_{t=t_{n+1}} \\
&= h^{\imath-\jmath} \frac{\partial^\imath}{\partial t^\imath} \left[\frac{\partial}{\partial \boldsymbol{x}^{(\jmath)}} \boldsymbol{f}(\boldsymbol{x}(t)) \right] \Bigg|_{t=t_{n+1}} \\
&= h^{\imath-\jmath} \frac{\partial^\imath}{\partial t^\imath} \left[\frac{\partial \boldsymbol{f}(\boldsymbol{x}(t))}{\partial \boldsymbol{x}(t)} \frac{\partial \boldsymbol{x}(t)}{\partial \boldsymbol{x}^{(\jmath)}} \right] \Bigg|_{t=t_{n+1}} \\
&= h^{\imath-\jmath} \frac{\partial^\imath}{\partial t^\imath} \left[\left(\sum_{u=0}^{\infty} \frac{1}{u!} \boldsymbol{R}_{n+1}^{(u)} \tau^u \right) \left(\frac{1}{\jmath!} \tau^\jmath \right) \right] \Bigg|_{t=t_{n+1}} \\
&= h^{\imath-\jmath} \binom{\imath}{\jmath} \left[\frac{\partial^{\imath-\jmath}}{\partial t^{\imath-\jmath}} \left(\sum_{u=0}^{\infty} \frac{1}{u!} \boldsymbol{R}_{n+1}^{(u)} \tau^u \right) \right] \Bigg|_{t=t_{n+1}} \\
&= h^{\imath-\jmath} \frac{\imath!}{\jmath!} \frac{1}{(\imath-\jmath)!} \boldsymbol{R}_{n+1}^{(\imath-\jmath)} \Bigg|_{t=t_{n+1}} = h^{\imath-\jmath} \frac{\imath!}{\jmath!} \boldsymbol{R}_{\imath-\jmath}.
\end{aligned}$$

The last two equations follow from Leibniz's rule and the fact that only the $\imath^{\text{th}}$ derivative of $t^\imath$ does not vanish at $t = t_{n+1}$. The above manipulation therefore demonstrates that the two-dimensional array of matrices $\boldsymbol{J}_{\imath,\jmath}$ can be replaced by the single-dimensional array $\boldsymbol{R}_\imath$ that represents the Taylor series coefficients matrices of the

Jacobian matrix used by the low-order method. Thus, the Jacobian matrix can be represented as follows

$$\tilde{\boldsymbol{J}} = \begin{bmatrix} \boldsymbol{G}+\boldsymbol{R}_0 & \frac{\boldsymbol{C}}{h} & & & \\ h\boldsymbol{R}_1 & \boldsymbol{G}+\boldsymbol{R}_0 & \frac{\boldsymbol{C}}{h} & & \\ \vdots & \ddots & \ddots & & \\ h^{m-1}(m-1)!\boldsymbol{R}_{m-1} & & \dots & \boldsymbol{G}+\boldsymbol{R}_0 & \frac{\boldsymbol{C}}{h} \\ \alpha_0\boldsymbol{I} & \alpha_1\boldsymbol{I} & \dots & & \alpha_m\boldsymbol{I} \end{bmatrix}. \tag{6.4}$$

In the modified implementation proposed in this section, the entries in the Jacobian matrix $\partial \boldsymbol{f}(\boldsymbol{x})/\partial \boldsymbol{x}$ are stored as nonlinear expressions represented by the rooted tree structures as described in chapter 5. Computing $\boldsymbol{R}_i$ is then carried out by assigning the appropriate $\boldsymbol{x}_{n+1}^{(j)}$ to the leaf nodes in the rooted tree and propagating the derivatives down to the root node in a manner similar to the description presented in chapter 5.

6.2 Block factorization

Although, the Jacobian matrix in (6.4) is very sparse, its sparse structure is not amenable to efficient LU factorization by circuit-oriented sparse LU factorization packages such as KLU [36], [34]. This is because sparse matrix solvers such as KLU, produce their best performance when the matrix to be factorized is symmetric and has a zero-free diagonal. Although matrices arising in circuit simulation are not exactly symmetric, their nonzero pattern is roughly symmetric. In addition, they have a zero-free or nearly zero-free diagonal.

Unfortunately, the Jacobian matrix produced by the high-order method and de-

picted by (6.4) is no longer nearly symmetric. Instead, it has lower block Hessenberg form. This fact can dramatically reduce the efficiency of circuit-oriented sparse solvers such as KLU.

It should be noted that there have been some recent attempts to factorize block Hessenberg matrices, e.g. [37] and [38]. However these techniques reach their peak efficiency when the blocks are full or sufficiently dense. In the present situation, however, the blocks have a circuit-like sparse structures.

The proposed idea to handle this issue takes advantage of the fact that each block is sparse and the blocks form a Hessenberg matrix. This idea can be explained by permuting the Jacobian matrix such that the h-scaled derivatives of the i^{th} variable are grouped together. This in effect makes the Jacobian matrix have the same circuit-like sparse structure, except that the entries become $(m+1)\times(m+1)$ matrices, instead of scalars. This idea is similar to the one proposed in [39] to handle the Jacobian matrix in the Harmonic Balance. However, the block entries arising in this situation enjoy unique structures that can be exploited to produce a more efficient factorization technique. Section 6.3 elaborates further on the structure of each block.

The proposed technique can be described in a two-stage process. In the first stage. a symbolic analysis is performed on an artificial $N \times N$ matrix whose sparsity pattern is identical to the sparsity pattern of the Jacobian matrix that arises in the course of using a low-order method such as the TR method. The goal of this stage is to obtain the optimal preordering techniques necessary to minimize the number of fill-ins. The symbolic analysis is carried out only once at the beginning of the simulation and its cost is identical to the one used for low-order methods.

The second stage is mainly numerical, where operations are performed on the

Jacobian matrix, whose entries are the $(m+1) \times (m+1)$ blocks, obtained by the permutation described above. In more precise terms, if $\hat{\boldsymbol{J}}$ is the $N \times N$ block matrix, whose entries are $(m+1) \times (m+1)$ blocks, then

$$\hat{\boldsymbol{J}} = \boldsymbol{P}\tilde{\boldsymbol{J}}\boldsymbol{P}^{\mathrm{T}}, \tag{6.5}$$

where $\boldsymbol{P}$ is an appropriate permutation operator. In this case, $\hat{\boldsymbol{J}}$ can be written as

$$\hat{\boldsymbol{J}} = [\hat{\boldsymbol{J}}_{i,j}]$$

where $i, j = 1, \ldots, N$ and $\hat{\boldsymbol{J}}_{i,j}$ is an $(m+1) \times (m+1)$ block, that occupies rows $(i-1)(m+1)+1$ to $i(m+1)$ and columns $(j-1)(m+1)+1$ to $j(m+1)$.

The numerical factorization typically proceeds using the left-looking LU factorization scheme with partial pivoting [36], [40], but with scalar floating-point operations extended to block operations. Figure 6.1 presents a pseudo-code illustration of the block version of the LU decomposition in [36].

BLOCK LEFT-LOOKING LU FACTORIZATION FOR AN $N \times N$ UNSYMMETRIC BLOCK MATRIX $\hat{\boldsymbol{J}}$

1 $\boldsymbol{L} = \boldsymbol{I}$
2 **for** $p = 1$ **to** N
do
3 solve the block lower triangular system $\boldsymbol{L}\boldsymbol{y} = \hat{\boldsymbol{J}}(:, p)$
4 $\boldsymbol{U}(1 : p-1, p) = \boldsymbol{y}(1 : p-1, 1)$
5 do partial pivoting on block vector $\boldsymbol{y}(p : N, 1)$
6 LU factorize $\boldsymbol{L}(p,p)\boldsymbol{U}(p,p) = \boldsymbol{y}(p,1)$
7 solve $\boldsymbol{L}(p+1 : N, p)\boldsymbol{U}(p,p) = \boldsymbol{y}(p+1 : N, 1)$ for $\boldsymbol{L}(p+1 : N, p)$

Figure 6.1: A pseudo-code description for the block LU factorization.

The pivoting step in the above algorithm is crucial to avoid numerical instability. However, given the block nature of the above algorithm, one needs to compare the

determinants of each block matrix in order to choose the block with the largest determinant as the pivot. As will be shown in section 6.3, the nature of these blocks makes computing their determinants trivial. However, due to fill-ins, full blocks can arise, in which case, computing the determinant can be expensive. Typically, computing the determinant of a matrix is carried out by performing the LU factorization and multiplying the diagonal elements of the resulting upper triangular matrix, i.e., given matrix $\boldsymbol{A}$ of size $(m+1) \times (m+1)$ and $\boldsymbol{A} = \boldsymbol{LU}$, the determinant of matrix $\boldsymbol{A}$ is obtained by

$$\begin{aligned} \det(\boldsymbol{A}) &= \det(\boldsymbol{L}) \det(\boldsymbol{U}) && (6.6) \\ &= \det(\boldsymbol{U}) && (6.7) \\ &= u_{1,1} u_{2,2} \cdots u_{m+1,m+1}, && (6.8) \end{aligned}$$

where equation (6.7) follows from the fact that $\det(\boldsymbol{L}) = 1$, and $u_{i,i}$, $i = 1, \ldots, m+1$, are the diagonal elements of $\boldsymbol{U}$ [41].

In order to relieve the computational burden of having to find the determinants of all the blocks, the proposed algorithm approximates the determinants by multiplying the diagonal elements of the matrices and selects the matrix with the largest product as the trial pivot. After performing the LU factorization of the trial pivot, the proposed algorithm computes the real determinant of the trial pivot by using (6.8). If the real determinant is close to the approximated determinant, the trial pivot is then accepted as the formal pivot. Otherwise, the proposed algorithm continues to seek new trial pivots in this fashion until a formal pivot is chosen.

6.3 Sparse reduced Jacobian formulation

This section presents an alternative formulation for the Jacobian matrix that leads to smaller matrix size without destroying the natural sparsity pattern of the Jacobian matrix. This formulation is based on the observation that $\boldsymbol{x}_{n+1}^{(m)}$ need not be computed explicitly as in system (5.13). Indeed, $\boldsymbol{x}_{n+1}^{(m)}$ can be written in terms of the lower-order derivatives $\boldsymbol{x}_{n+1}^{(i)}$, $i < m$, using the Obreshkov formula itself

$$h^m \boldsymbol{x}_{n+1}^{(m)} = \frac{1}{\alpha_m} \sum_{i=0}^{l} \beta_i (h^i \boldsymbol{x}_n^{(i)}) - \frac{1}{\alpha_m} \sum_{i=0}^{m-1} \alpha_i (h^i \boldsymbol{x}_{n+1}^{(i)}). \tag{6.9}$$

Substituting from (6.9) into (5.13) produces a new system

$$\begin{bmatrix} \boldsymbol{G} & \frac{\boldsymbol{C}}{h} & & \\ & \boldsymbol{G} & \frac{\boldsymbol{C}}{h} & \\ & & \ddots & \ddots \\ -\frac{\alpha_0}{\alpha_m}\frac{\boldsymbol{C}}{h} & \cdots & & \boldsymbol{G} - \frac{\alpha_{m-1}}{\alpha_m}\frac{\boldsymbol{C}}{h} \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_{n+1} \\ h\boldsymbol{x}_{n+1}^{(1)} \\ \vdots \\ h^{m-1}\boldsymbol{x}_{n+1}^{(m-1)} \end{bmatrix} + \begin{bmatrix} \boldsymbol{f}_{n+1} \\ h\boldsymbol{f}_{n+1}^{(1)} \\ \vdots \\ h^{m-1}\boldsymbol{f}_{n+1}^{(m-1)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{b}_{n+1} \\ h\boldsymbol{b}_{n+1}^{(1)} \\ \vdots \\ \hat{\boldsymbol{b}} \end{bmatrix}, \tag{6.10}$$

where $\hat{\boldsymbol{b}} = h^{m-1}\boldsymbol{b}_{n+1}^{(m-1)} - \frac{1}{\alpha_m}\frac{\boldsymbol{C}}{h}\sum_{k=0}^{l}\beta_k(h^k\boldsymbol{x}_n^{(k)})$. The corresponding Jacobian matrix is given in (6.11)

$$\tilde{\boldsymbol{J}} = \underbrace{\begin{bmatrix} \boldsymbol{G}+\boldsymbol{R}_0 & \frac{\boldsymbol{C}}{h} & & \\ h\boldsymbol{R}_1 & \boldsymbol{G}+\boldsymbol{R}_0 & \frac{\boldsymbol{C}}{h} & \\ 2h^2\boldsymbol{R}_2 & 2h\boldsymbol{R}_1 & \boldsymbol{G}+\boldsymbol{R}_0 & \frac{\boldsymbol{C}}{h} \\ \vdots & \ddots & \ddots & \\ h^{m-1}(m-1)!\boldsymbol{R}_{m-1} - \frac{\alpha_0}{\alpha_m}\frac{\boldsymbol{C}}{h} & h^{m-2}(m-1)!\boldsymbol{R}_{m-2} - \frac{\alpha_1}{\alpha_m}\frac{\boldsymbol{C}}{h} & \cdots & \boldsymbol{G}+\boldsymbol{R}_0 - \frac{\alpha_{m-1}}{\alpha_m}\frac{\boldsymbol{C}}{h} \end{bmatrix}}_{m\times m \text{ blocks}}. \tag{6.11}$$

It is important to note that the size of the new Jacobian matrix in (6.11) is $mN \times mN$ instead of $(m+1)N \times (m+1)N$ obtained earlier. It should also be

remarked that the sparsity pattern of the new Jacobian matrix is still similar to the sparsity pattern in the old one. This fact is advantageous since it maintains the same efficiency obtained from the block factorization algorithm presented in section 6.2.

After performing the permutation proposed in section 6.2, the blocks in the final Jacobian matrix can be one, or a combination of more than one of the following types of blocks.

1. A diagonal block, of the form $g\boldsymbol{I}$. This is the type of blocks that can arise if node i or node j, or both are connected to memoryless linear elements.

2. A block of the form $\frac{c}{h}\boldsymbol{B}$, where $\boldsymbol{B}$ is a companion matrix, i.e.,

$$\begin{bmatrix} 0 & 1 & & \\ \vdots & \ddots & \ddots & \\ 0 & \dots & 0 & 1 \\ -\frac{\alpha_0}{\alpha_m} & -\frac{\alpha_1}{\alpha_m} & \dots & -\frac{\alpha_{m-1}}{\alpha_m} \end{bmatrix}.$$

   This block arises if node i or node j, or both are connected to linear memory elements.

3. A lower triangular block of the form

$$\begin{bmatrix} r_0 & 0 & & \\ hr_1 & r_0 & 0 & \\ \vdots & & \ddots & \ddots \\ h^{m-1}(m-1)!r_{m-1} & & \dots & r_0 \end{bmatrix},$$

   which arises if node i or node j, or both are connected to nonlinear memoryless elements.

4. Naturally, if node i or node j or both are connected to more than one types of the above mentioned elements, therefore, in its most general form, a block can be a lower Hessenberg matrix of the form

$$\begin{bmatrix} g+r_0 & \frac{c}{h} & & \\ hr_1 & g+r_0 & \frac{c}{h} & \\ \vdots & \ddots & \ddots & \\ h^{m-1}(m-1)!r_{m-1} - \frac{\alpha_0}{\alpha_m}\frac{c}{h} & \cdots & & g+r_0 - \frac{\alpha_{m-1}}{\alpha_m}\frac{c}{h} \end{bmatrix}.$$

The nature of those blocks makes it easier to construct and store the Jacobian matrix using only information about the circuit elements connected at each node.

6.4 Reduced cost for Newton–Raphson method

Naturally, high-order methods can afford a larger time-step than low-order integration methods. However, a larger time-step entails a larger effort by the nonlinear iterative solver to converge to a solution to the algebraic nonlinear equations. This fact may become a significant computational burden, especially when the nonlinear system in (6.10) is m times the normal size of the circuit.

This section addresses the above problem by developing a technique that can reduce the number of iterations performed on the augmented system and replace them by iterations on the reduced system. The basic idea in the proposed technique relies on using a low-order method to obtain a low-accuracy solution. For illustration purpose, we assume here that the TR is used to obtain such low accuracy solution. Thus, denoting that solution by $\hat{\boldsymbol{x}}_{n+1}$, we have by TR

$$\hat{\boldsymbol{x}}_{n+1} = \boldsymbol{x}_n + \frac{h}{2}(\boldsymbol{x}'_n + \hat{\boldsymbol{x}}'_{n+1}). \tag{6.12}$$

Substituting (6.12) into (5.11) and rearranging terms gives

$$\left(\boldsymbol{G}+\frac{2}{h}\boldsymbol{C}\right)\hat{\boldsymbol{x}}_{n+1}+\boldsymbol{f}_{n+1}(\hat{\boldsymbol{x}}_{n+1})=\left(\frac{2}{h}\boldsymbol{C}-\boldsymbol{G}\right)\boldsymbol{x}_n-\boldsymbol{f}_n(\boldsymbol{x}_n)+\boldsymbol{b}_{n+1}+\boldsymbol{b}_n. \quad (6.13)$$

The above nonlinear system can be solved using the same iterative Newton–Raphson method. However, it should be noted here that the Jacobian matrix is only an $N\times N$ matrix, as opposed to the $mN\times mN$ matrix used in the high-order methods.

Upon convergence, the obtained $\hat{\boldsymbol{x}}_{n+1}$ represents an initial guess for the first N components in $\boldsymbol{\xi}_{n+1}$. Generating the initial guess for the remaining $(m-1)N$ components representing the h-scaled high-order derivatives in $\boldsymbol{\xi}_{n+1}$ is carried out through using the system in (6.13). More specifically, differentiating both sides of (6.13) w.r.t. t and multiplying by h, we obtain

$$\left(\boldsymbol{G}+\frac{2}{h}\boldsymbol{C}\right)h\hat{\boldsymbol{x}}^{(1)}_{n+1}+\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\Big|_{\hat{\boldsymbol{x}}_{n+1}}h\hat{\boldsymbol{x}}^{(1)}_{n+1}= \left(\frac{2}{h}\boldsymbol{C}-\boldsymbol{G}\right)h\boldsymbol{x}^{(1)}_n-\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\Big|_{\boldsymbol{x}_n}h\boldsymbol{x}^{(1)}_n+h\boldsymbol{b}^{(1)}_{n+1}+h\boldsymbol{b}^{(1)}_n. \quad (6.14)$$

Using the notation introduced earlier, whereby $\partial \boldsymbol{f}/\partial \boldsymbol{x}$ is expanded in a Taylor series as in (6.2), taking the $(i-1)^{\text{th}}$ derivative of $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}h\boldsymbol{x}^{(1)}_{n+1}$ w.r.t. t and multiplying by h^{i-1} yields

$$\begin{aligned}
h^i\boldsymbol{f}^{(i)}_{n+1} &= h^{i-1}\frac{\partial^{i-1}}{\partial t^{i-1}}\left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}h\boldsymbol{x}^{(1)}\right)\Big|_{t_{n+1}}\\
&= h^{i-1}\frac{\partial^{i-1}}{\partial t^{i-1}}\left(\left(\sum_{u=0}^{\infty}\frac{1}{u!}\boldsymbol{R}^{(u)}_{n+1}\tau^u\right)h\boldsymbol{x}^{(1)}\right)\Big|_{t_{n+1}}\\
&= h^i\sum_{j=0}^{i-1}\binom{i-1}{j}\frac{\partial^{i-1-j}}{\partial t^{i-1-j}}\left(\sum_{u=0}^{\infty}\frac{1}{u!}\boldsymbol{R}^{(u)}_{n+1}\tau^u\right)\boldsymbol{x}^{(j+1)}\Big|_{t_{n+1}}\\
&= h^i\sum_{j=0}^{i-1}\binom{i-1}{j}\boldsymbol{R}^{(i-1-j)}_{n+1}\boldsymbol{x}^{(j+1)}_{n+1}. \qquad (6.15)
\end{aligned}$$

The last two equations follow from Leibniz's rule and the fact that only the i^{th} derivative of t^i does not vanish at $t = t_{n+1}$. Thus, using the results obtained in (6.15), taking the $(i-1)^{\text{th}}$ derivative of (6.14) w.r.t. t and multiplying both sides by h^{i-1}, we obtain the following system for the higher-order derivatives of $\hat{\boldsymbol{x}}_{n+1}$

$$\left(\boldsymbol{G} + \frac{2}{h}\boldsymbol{C} + \boldsymbol{R}_{n+1}^{(0)}\right) h^i \hat{\boldsymbol{x}}_{n+1}^{(i)} = \left(\frac{2}{h}\boldsymbol{C} - \boldsymbol{G}\right) h^i \boldsymbol{x}_n^{(i)} + h^i \boldsymbol{b}_{n+1}^{(i)} + h^i \boldsymbol{b}_n^{(i)} - h^i \sum_{j=0}^{i-1} \binom{i-1}{j} \boldsymbol{R}_n^{(i-1-j)} \boldsymbol{x}_n^{(j+1)} - h^i \sum_{j=0}^{i-2} \binom{i-1}{j} \boldsymbol{R}_{n+1}^{(i-1-j)} \hat{\boldsymbol{x}}_{n+1}^{(j+1)}. \quad (6.16)$$

The above system can be solved recursively for $h^i \hat{\boldsymbol{x}}_{n+1}^{(i)}$ where the obtained solution is used as the starting initial guess for the augmented system in (6.10).

It should be remarked here that the additional cost in computing the $m-1$ higher-order derivatives will be simply another $m-1$ FB substitutions of the factorized Jacobian matrix obtained at the convergence of (6.13).

It should also be emphasized that the goal of the proposed technique is to shift a significant portion of the convergence effort from the $\mathbb{R}^{mN}$ space to the $\mathbb{R}^N$ space.

Chapter 7

Numerical experiments

In this chapter, we present a few numerical examples to demonstrate the advantages of the proposed algorithm. The implementation techniques introduced in chapter 5 were used to run the simulations. More examples will be given in chapter 8 to demonstrate the efficiency introduced by the techniques described in chapter 6. The algorithm was implemented using C++. Bison and flex were used to generate the parsing code and to interpret the netlists and nonlinear expressions. UMFPACK was used to perform LU decompositions and FB substitutions. All the tests were performed on Sun Microsystems, where the system clock frequency is 177MHZ and the memory size is 12GB.

7.1 Constant step size simulation

The proposed algorithm was tested on the linear circuit shown in Figure 7.1 to provide the initial proof-of-concept. The exact transient response for this circuit was obtained using a direct eigenvector-eigenvalue analysis, and is plotted in Figure 7.2 for node n_3.

This figure also shows the results obtained from the proposed algorithm after fixing the step size to 2nS and compares them with the results obtained by the trapezoidal rule (TR) for the same step size.

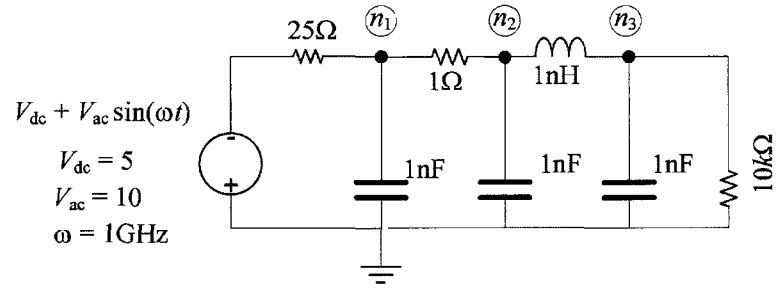

Figure 7.1: The linear test circuit used in section 7.1.

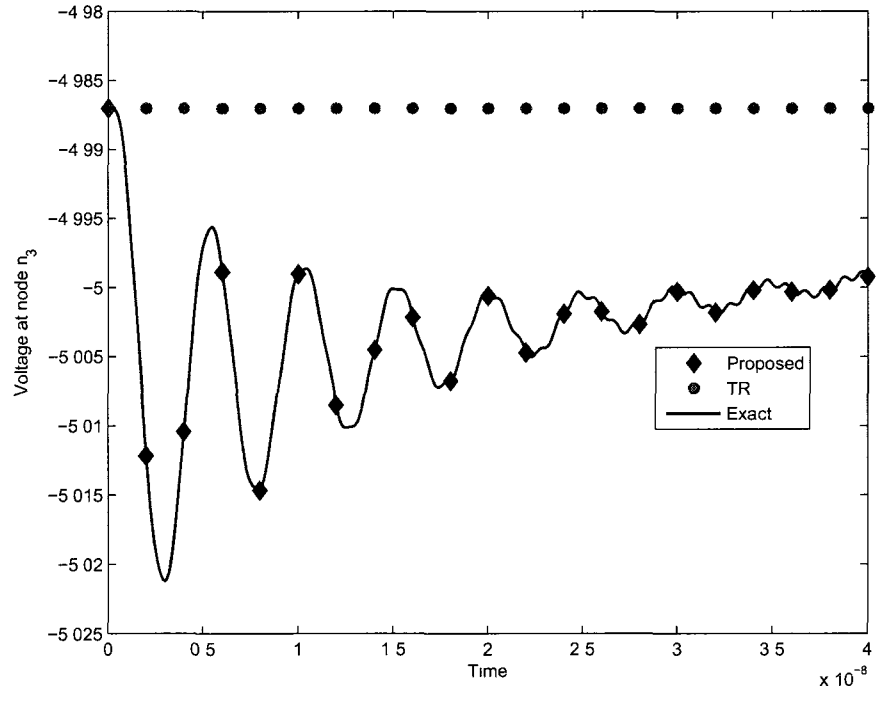

Figure 7.2: The waveform obtained in simulating the circuit of Figure 7.1.

This set of results demonstrate the performance of the proposed method when implemented using a constant size and for different values for the step sizes. Figure 7.3 shows the error obtained by the proposed method and compares it with the error resulting from using the Trapezoidal rule. The error shown in this figure is the difference between the exact and approximated solution averaged over the integration interval. The order used for the integration method in this experiment is 18.

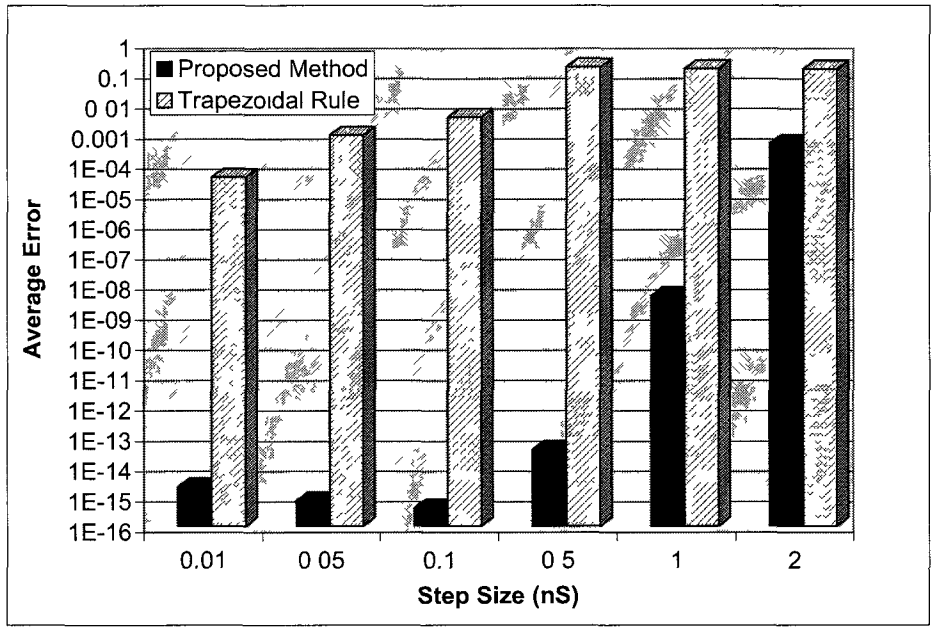

Figure 7.3: A comparison between the proposed method and TR for the average error vs. the step size.

7.2 Variable step size simulation

The proposed algorithm was tested on the linear circuit shown in Figure 7.4. The simulation result shown in Figure 7.5 was generated by using small step sizes.

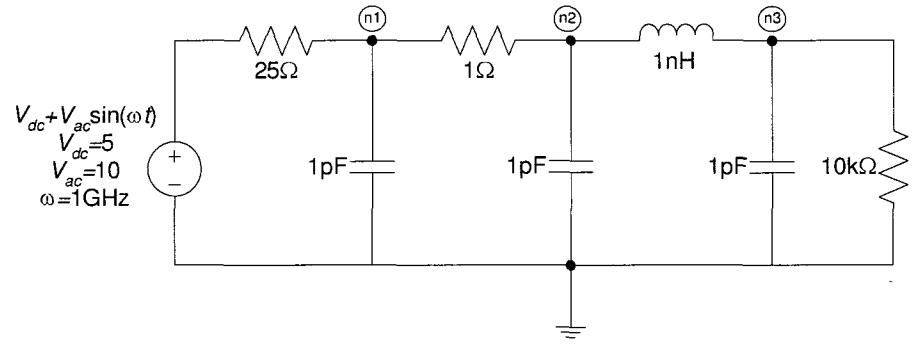

Figure 7.4: A linear RLC test circuit.

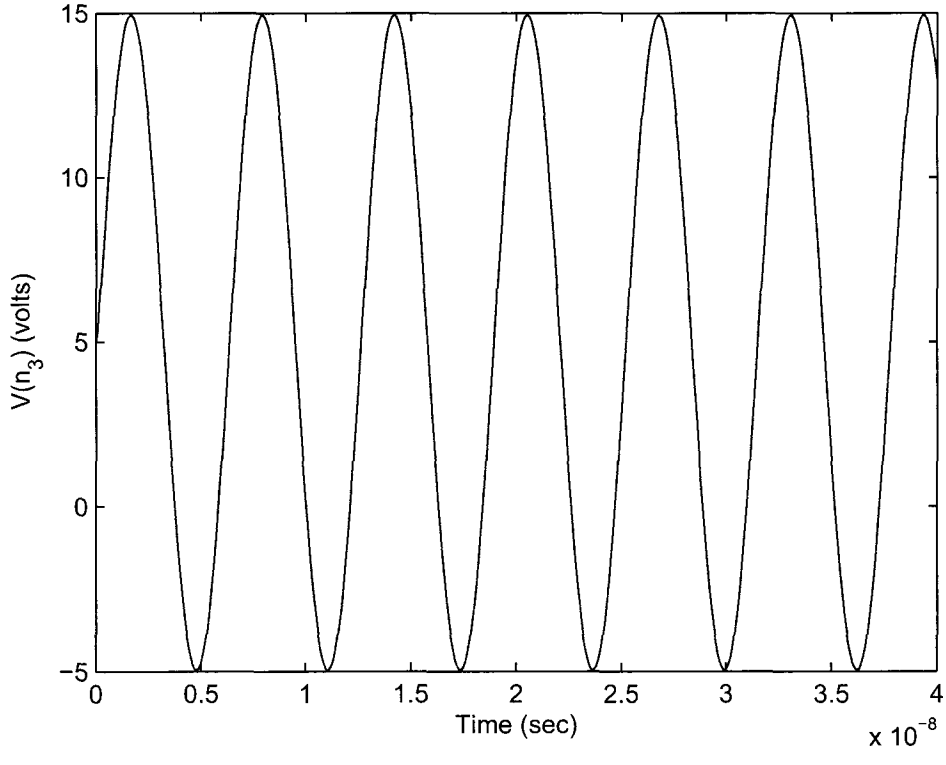

Figure 7.5: The time domain response of the circuit in Figure 7.4.

The step size was controlled to maintain the error below various error tolerances. Figure 7.6 displays the number of steps needed by the proposed method vs. the order of integration for different error tolerance levels ranging from $\epsilon_{\text{tol}} = 10^{-3}$ to 10^{-8} over an interval of 4ns.

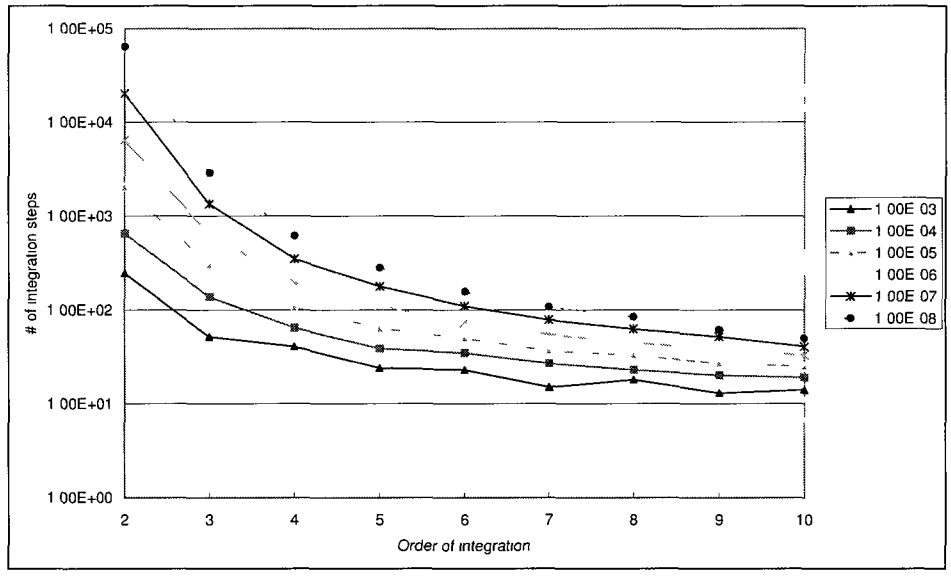

Figure 7.6: Integration steps vs. the order of integration for different tolerances.

This example was simulated using HSPICE® for 0.1ns, where the commercial simulator needed 53 time points. The proposed algorithm of order 6 required only 4 points for the same time-span and tolerance level.

7.3 A MOSFET amplifier circuit

The MOSFET amplifier circuit of Figure 7.7 is considered for this example, where the MOSFET transistor is modeled with the equivalent circuit in Figure 7.8. The

drain-source current I_{ds} is described by a single-piece C_∞ nonlinearity that models the different modes of operation including subthreshold conduction [42]. C_∞ models are often desirable in modeling device nonlinearities, but they often lead to complex nonlinear expressions. The C_∞ model and the netlist describing the circuit in Figure 7.7 are given in Appendix A.

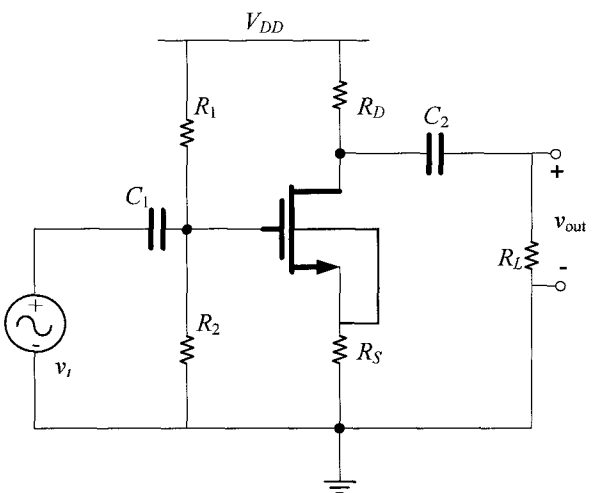

Figure 7.7: A MOSFET amplifier circuit. $R_1 = 99\text{k}\Omega$, $R_2 = 20\text{k}\Omega$, $R_D = 17.5\text{k}\Omega$, $R_s = 2\text{k}\Omega$, $R_L = 175\text{k}\Omega$, $C_1 = C_2 = 10\mu\text{F}$, $v_i = 1.6 \times \sin(\omega_0 t)$, $\omega_0 = 2\pi\text{kHz}$, $V_{DD} = 15\text{V}$.

The goal of this experiment is to demonstrate the performance of the high-order derivatives handling methodology in the proposed algorithm. Table 7.1 presents the CPU time spent in the evaluation of those derivatives needed to compute $\tilde{\boldsymbol{f}}$ and $\partial\tilde{\boldsymbol{f}}/\partial\boldsymbol{\xi}$ up to the 19^{th} derivative and compares it with the time taken by the AD package from Maple. The AD columns in the table show the construction time, that is the time needed to generate the symbolic derivatives, and the time needed to evaluate $\tilde{\boldsymbol{f}}$. In the proposed technique, the time needed to translate the nonlinear expression into the

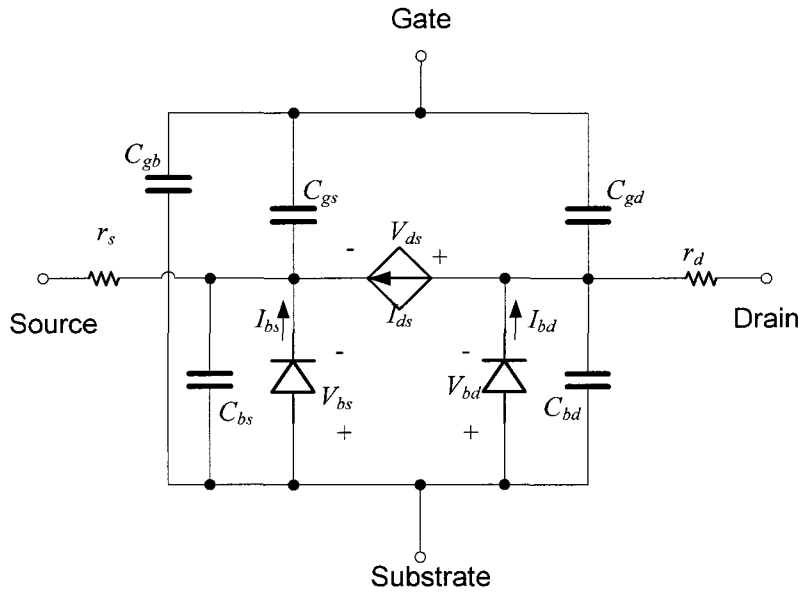

Figure 7.8: The MOSFET model used in Figure 7.7. $C_{gb} = G_{gd} = C_{bs} = C_{bd} = 20$fF, $C_{gs} = 12.9$fF, $r_d = r_s = 1\Omega$, $I_{bs} = I_{sbs}(\exp(v_{bs}/V_T) - 1)$, $I_{bd} = I_{sbd}(\exp(v_{bd}/V_T) - 1)$, $I_{sbs} = 10^{-16}$A, $I_{sbd} = 6.952 \times 10^{-14}$A, $V_T = 25e-3$, I_{ds} is modeled by a single-piece C_∞ model with equations described in [42].

rooted tree, which is independent of the derivative order, was equal to 0.046 seconds. It is to be noted also here that AD cannot compute the entries in the Jacobian matrix $\partial\tilde{\boldsymbol{f}}/\partial\boldsymbol{\xi}$ and therefore is not reported. Figure 7.9 shows the output waveform of this circuit, with the time points marked by squares. It was generated by the proposed algorithm with order 6 and $\epsilon_{\text{tol}} = 1e-4$.

Table 7.1: CPU time of the high-order derivative computations.

Order (i)	Proposed Technique		AD	
	$\tilde{\boldsymbol{f}}$	$\partial\tilde{\boldsymbol{f}}/\partial\boldsymbol{\xi}$	Construction	$\tilde{\boldsymbol{f}}$
1	9.68e-05	3.73e-04	1.60e-02	1.50e-2
5	3.94e-04	1.80e-03	1.87e-01	6.30e-02
10	7.97e-04	4.99e-03	3.94E+00	3.90e-01
15	1.26e-03	1.41e-02	3.40e+01	1.16e+00
19	1.70e-03	1.94e-02	2.14e+02	2.31e+00

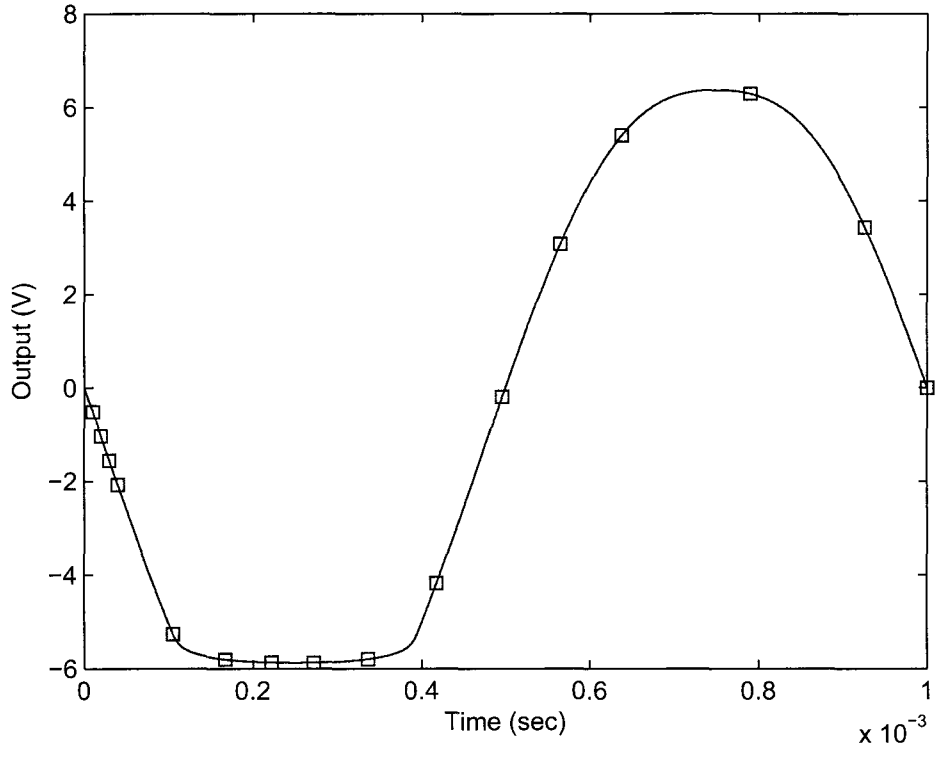

Figure 7.9: The output response of the MOSFET amplifier circuit. Exact response (represented by a solid line) was approximated using a very small step size. The response represented by the squared dots was obtained from the proposed algorithm.

7.4 An inverter circuit

The proposed algorithm was used to run the transient simulation of an inverter circuit shown in Figure 7.10, where the input stimulus is a piecewise-linear waveform represented by a trapezoidal waveform having a rise/fall time of 0.2ns, a pulse-width of 0.8ns and an amplitude of 1.2 volts.

The model used to represent the inverter is described by the behavioral model

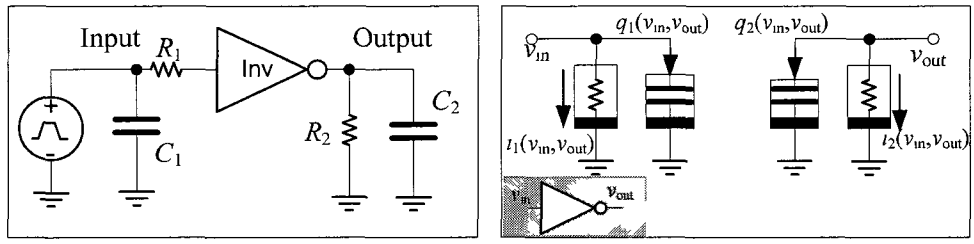

Figure 7.10: A digital circuit with an inverter and its modeling. $R_1 = 50\Omega$, $R_2 = 10\text{k}\Omega$, $C_1 = 0.2\text{fF}$, $C_2 = 0.2\text{fF}$.

illustrated in Figure 7.10(b). Such an approach to behavioral modeling of logic gates was proposed in [43], where a nonlinear capacitance and a nonlinear conductance are used at each port of the logical gate to model its dynamic behavior. The nonlinear functions in all the capacitors and conductors are functions of all port voltages and represented by sets of tabulated data providing the charge and current values at each port corresponding to a given configuration of port voltages. The tabulated data, which is generated by some independent simulations, is fitted to neural network having a single hidden layer employing a hyper-tangent activation function using a Matlab® neural network toolbox within 2 minutes. The detailed description of the neural network and the netlist of Figure 7.10 are given in Appendix B.

Figure 7.11 depicts the transient results obtained at the input and output nodes of the circuit. The TR simulation required 2859 points and lasted for 6 seconds, whereas the proposed algorithm (order 6) required 225 points and lasted 0.87 seconds for the same error tolerance level.

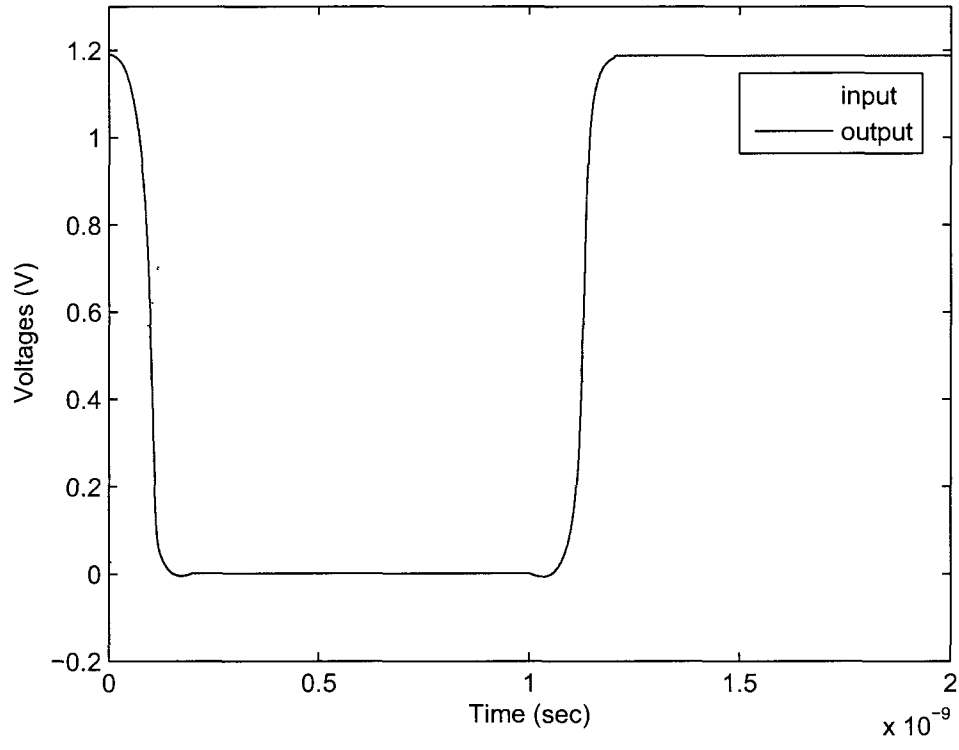

Figure 7.11: Transient simulation results for the inverter circuit.

7.5 A Cauer low-pass filter with MOSFET amplifiers

The proposed algorithm was tested on a Cauer low-pass filter consisting 8 OpAmps [2], where each OpAmp was constructed from a two-stage amplifier [44] as shown in Figure 7.12. The Cauer low-pass filter is shown in Figure 7.14. The MOSFET equivalent circuit illustrated in Figure 7.8 was used to model each transistor. The time domain response of the circuit is shown in Figure 7 13. Table 7.2-7.4 summarize the CPU simulation time along with the maximum error and number of computed time points for three different tolerance levels and for different orders. It is important

to note here that for order 2 ($l = 1,\ m = 1$), the modified Obreshkov formula reduces exactly to the simple trapezoidal rule.

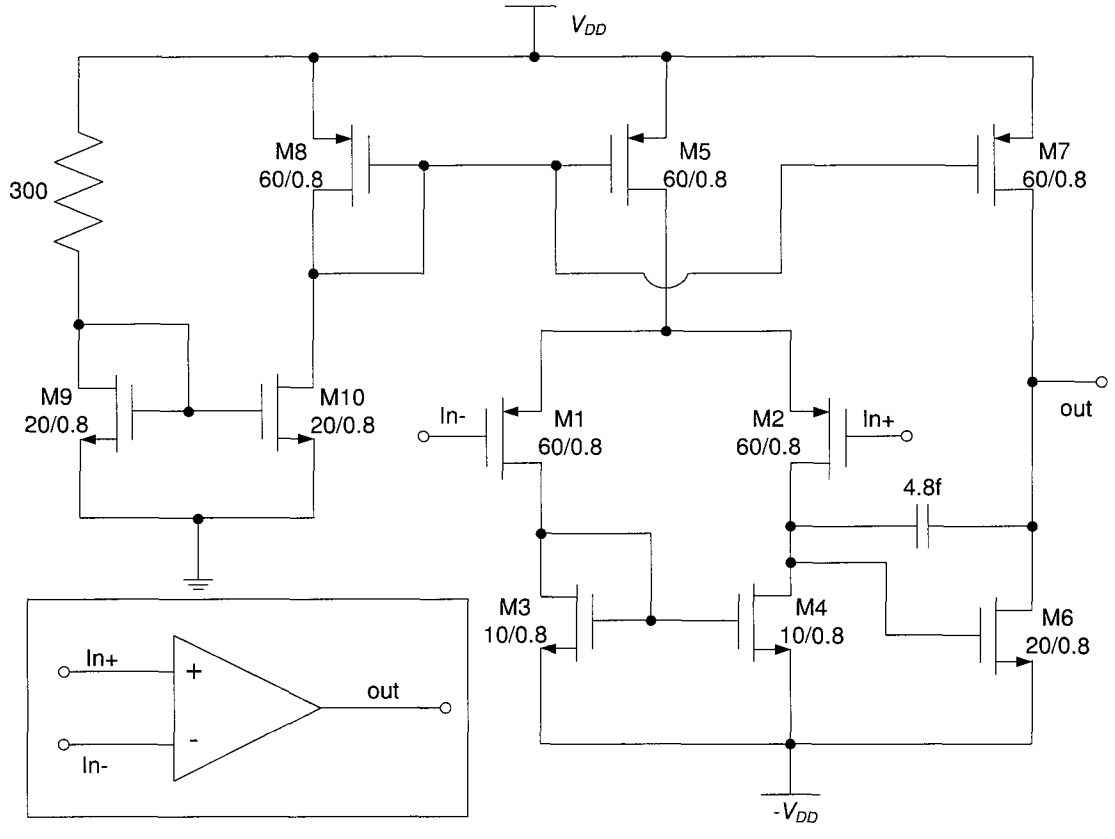

Figure 7.12: The two-stage OpAmp used in the Cauer low-pass filter. ($V_{DD} = 7$V)

Table 7.2: CPU cost for the Cauer low-pass filter with MOSFETs with $\epsilon_{\text{tol}} = 1e-2$.

Order	Error	Pts	#LUs	CPU			Speedup
				LU	Fn&Jn	Other	
TR (2)	8.74e-3	576	1815	7.71	241	1.5	1.0
3	8.88e-3	137	545	4.14	142	0.73	1.7
4	8.68e-3	60	252	2.10	67	0.34	3.6
5	8.70e-3	36	167	2.01	68	0.33	3.5
6	8.11e-3	26	145	2.53	60	0.29	3.9

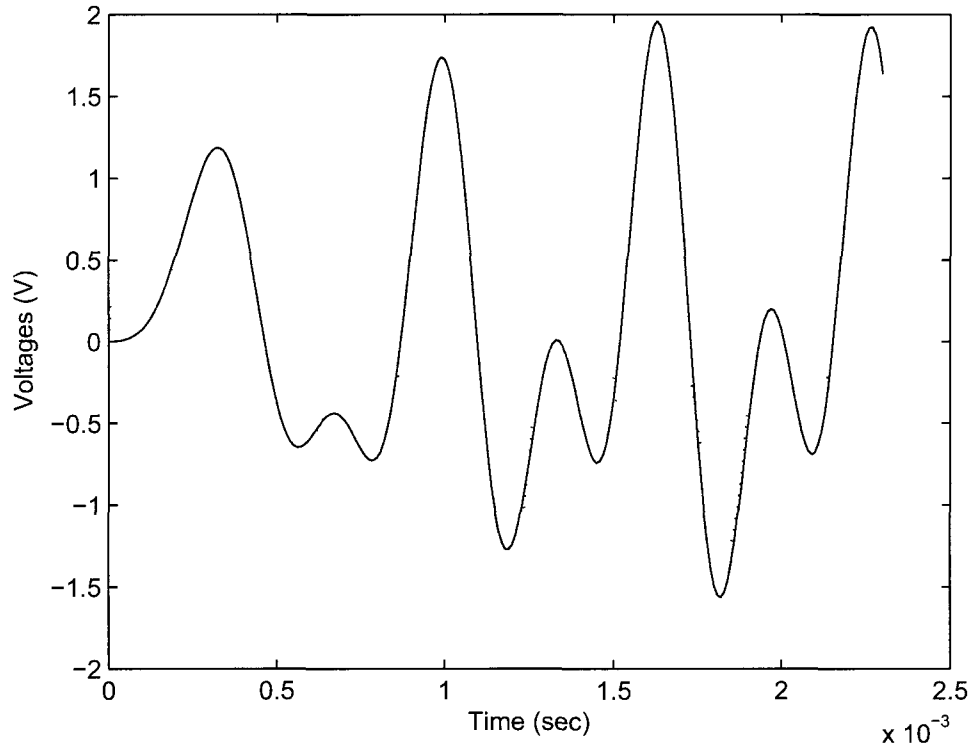

Figure 7.13: The time domain response of the Cauer low-pass filter with MOSFETs. The dotted line is the input and the solid line is the output.

Table 7.3: CPU cost for the Cauer low-pass filter with MOSFETs with $\epsilon_{\text{tol}} = 1e-3$.

Order	Error	Pts	#LUs	CPU			Speedup
				LU	Fn&Jn	Other	
TR (2)	9.23e-4	1771	5407	21.31	717	4.46	1.0
3	9.36e-4	289	1032	7.70	270	1.41	2.6
4	9.63e-4	101	417	3.35	110	0.565	6.5
5	9.92e-4	55	236	2.53	96	0.462	7.4
6	8.76e-4	36	168	2.07	69	0.328	10.3

The maximum errors shown in the second column have been computed by running the algorithm with very small step sizes and taking the resulting waveform as

Table 7.4: CPU cost for the Cauer low-pass filter with MOSFETs with $\epsilon_{\text{tol}} = 1e-4$.

Order	Error	Pts	#LUs	CPU			Speedup
				LU	Fn&Jn	Other	
TR (2)	8.74e-5	5751	17044	65.59	2261	14.38	1.0
3	8.57e-5	640	2023	14.90	528	2.80	4.2
4	8.89e-5	185	721	5.55	188	0.972	11.9
5	1.03e-4	90	380	4.24	153	0.743	14.7
6	9.16e-5	59	258	3.17	105	0.512	21.4

the reference for comparison for all orders and all tolerances. The third and fourth columns in these tables provide the number of computed time points and LU factorizations, respectively. The subsequent three columns provide a breakdown for the CPU time by the proposed algorithm, where the "Fn&Jn" denotes the total time for computing $\tilde{\boldsymbol{f}}_{n+1}$ and $\partial \tilde{\boldsymbol{f}}_{n+1}/\partial \boldsymbol{\xi}_{n+1}$. The last column in these tables provides the measured speedup relative to the CPU time of the trapezoidal rule which is reported at the first row in each table. As can be observed, the proposed algorithm offers a higher speed up for more accurate simulations.

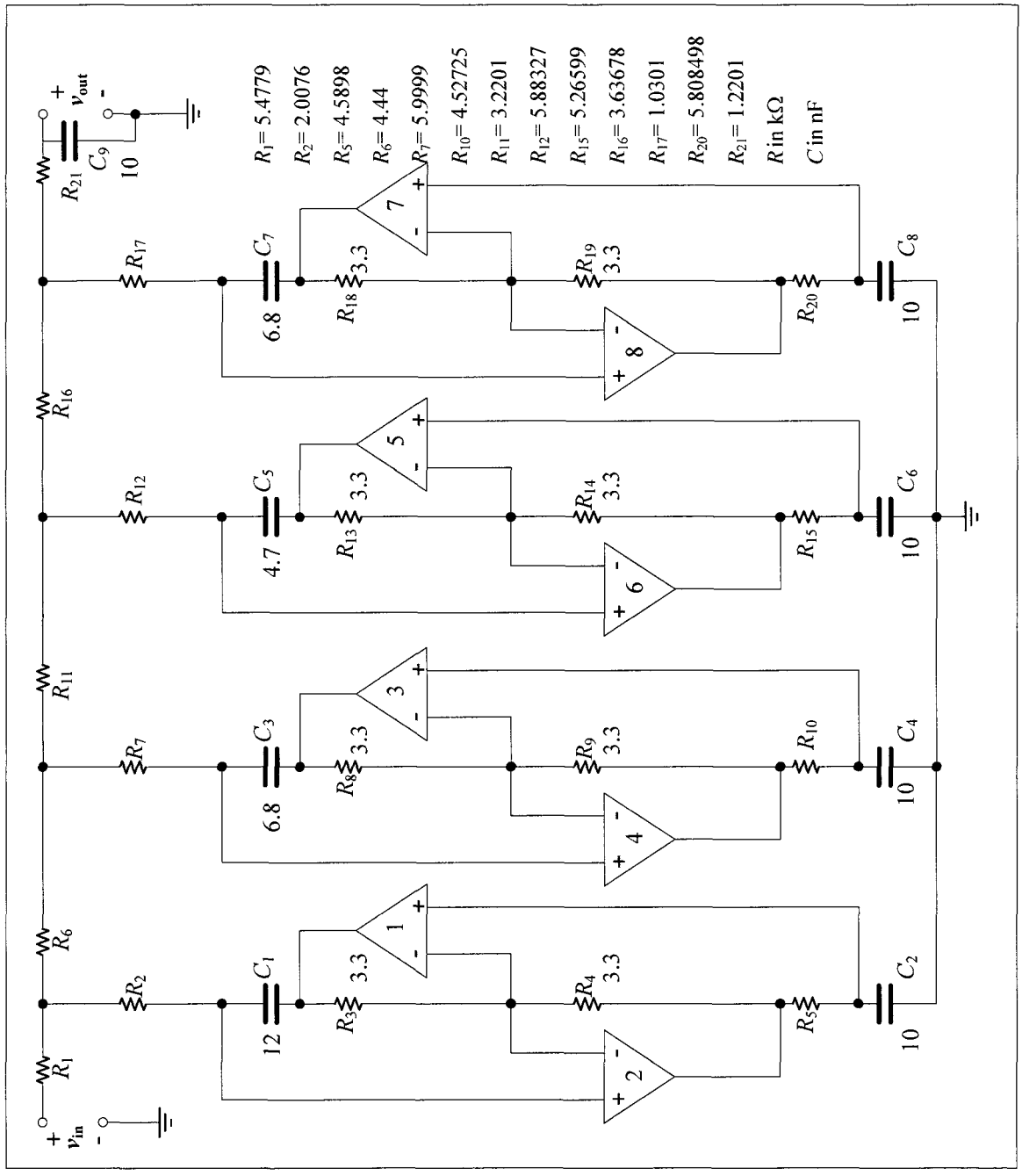

Figure 7.14: The Cauer-parameter low-pass filter used in section 7.5, pass-band from 0 to 3470Hz, 0.03dB ripple [2], $V_{in} = \sin(3200\pi t) + \sin(6200\pi t)$V.

7.6 A Cauer low-pass filter with μA-741 OpAmps

Instead of the two-stage MOSFET amplifier, the bipolar OpAmp μA-741 [44] was used to redesign the Cauer low-pass filter. The same simulation configuration shown in Figure 7.14 was used. The bipolar transistors were modeled by the Ebers-Moll models shown in Figure 7.15. Table 7.5 compares the performance of TR and that of the proposed algorithm of order 6.

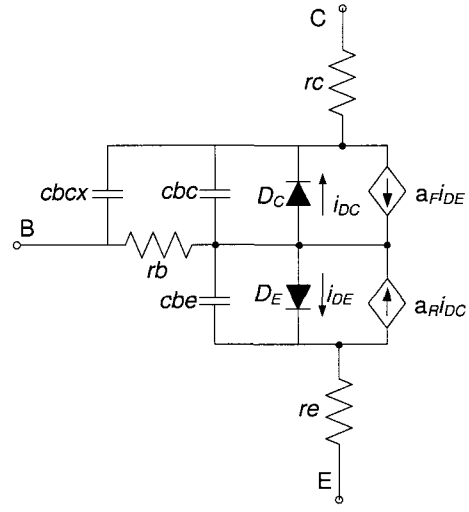

Figure 7.15: The Ebers-Moll model for NPN bipolar ($cbc = 0.8$pF, $cbe = 1$pF, $cbcx = 0.1$pF, $rb = 100\Omega$, $rc = 10\Omega$, $re = 1\Omega$, $i_{DE} = I_{SE}(\exp(V_{BE}/V_T) - 1)$, $i_{DC} = I_{SC}(\exp(V_{BC}/V_T) - 1)$, $\alpha_F I_{SE} = \alpha_R I_{SC} = I_S = 1e-15$A, $\alpha_F = 0.99$, $\alpha_R = 0.02$, $V_T = 25e-3$).

Table 7.5: CPU cost for the Cauer low-pass filter (μA-741 OpAmp) with $\epsilon_{\text{tol}} = 1e-4$.

Order	Pts	#LUs	CPU			Speedup
			LU	Fn&Jn	Other	
TR (2)	5751	17173	943	90.47	77.97	1.0
6	53	330	63.15	4.44	3.27	15.8

Chapter 8

Numerical examples by improved techniques

This chapter presents more numerical examples to show the influence of the proposed implementation improvements suggested in chapter 6.

8.1 Fill-in reduction in LU factorization

This example demonstrates the fill-in reduction in the LU factorization of the Jacobian matrix using the techniques proposed in section 6.2. Here, the Cauer low-pass filter with μA-741 OpAmps, which was presented in section 7.6, is considered again.

A sample Jacobian matrix arising in the course of using the high-order method was considered for factorization using two proposed techniques. The first is the classic KLU and the other one is the proposed block-form technique presented in section 6.2. The Jacobian matrix has the structure depicted in (6.11) and its size was $mN \times mN$, where N is the size of the problem MNA formulation, which is equal to 1601 for this

example. Table 8.1 compares the number of non-zero fill-ins obtained from the two techniques for various orders of m. It is obvious that the proposed technique can reduce the number of fill-ins by more than half as compared to the classic KLU.

Table 8.1: Number of nonzero fill-ins in the $\boldsymbol{L}$ and $\boldsymbol{U}$ factors.

m	1	2	3	4	5	6
Classic KLU	9300	36903	73682	145810	263614	560584
Proposed block LU	9300	30379	61785	104278	157591	221716

Figures 8.1 and 8.2 compare the sparsity pattern of the $\boldsymbol{L}$ and $\boldsymbol{U}$ factors obtained from the classic KLU and the proposed block factorization algorithm.

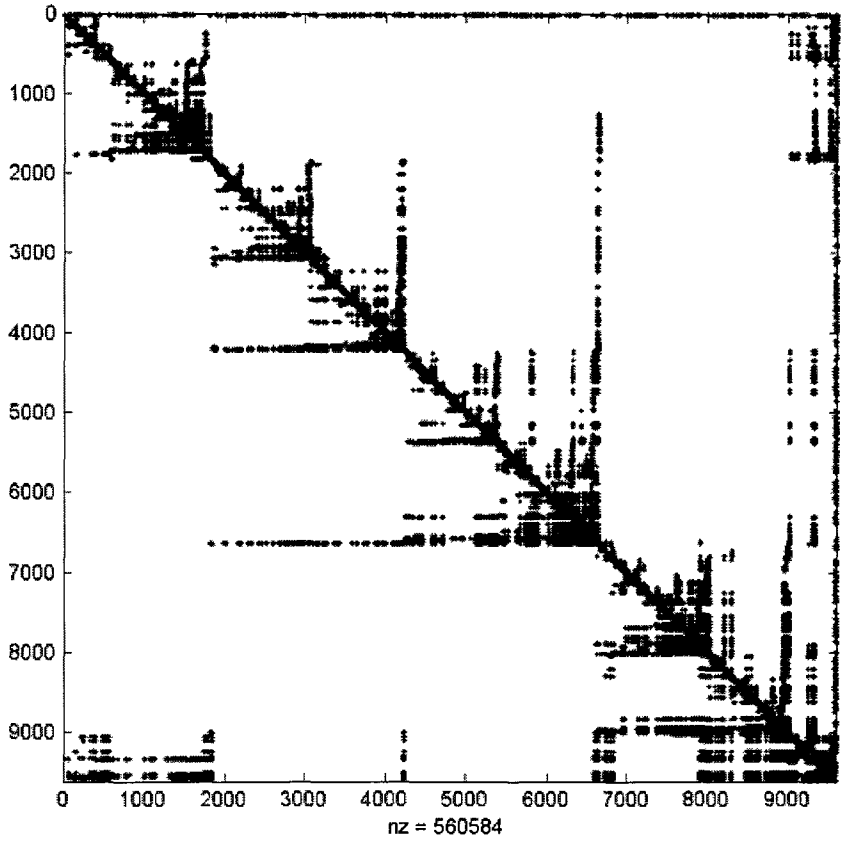

Figure 8.1: The LU sparsity pattern generated by the classic KLU.

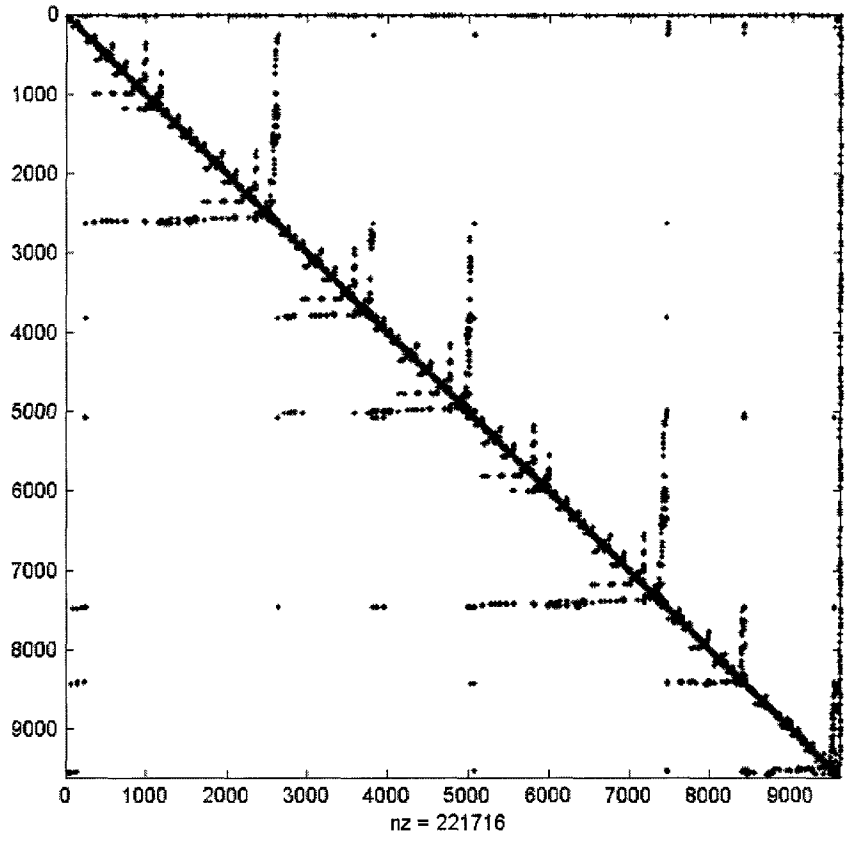

Figure 8.2: The LU sparsity pattern generated by the block factorization algorithm proposed in chapter 6.

8.2 Improved high-order Newton–Raphson convergence

This example demonstrates the savings in the computational efforts associated with the convergence to a solution in the course of Newton iterations. The Cauer low-pass filter in the previous example was simulated for different high orders, where at each step, the NR initial trial solution was generated using two different approaches. In

the first approach, the results obtained at $t = t_n$ were used as the initial guess of high-order NR iterations at $t = t_{n+1}$. In the second approach, before the high-order NR iterations, the proposed technique in section 6.4 was run first to generate the initial guess. For comparison purpose, other configurations, such as the step size, the total number of time points and the factorization algorithm were kept the same for the two situations.

Sample results reported in Table 8.2 show the total number of high-order NR iterations along with the total number of NR iterations used by TR to generate the initial guess.

Table 8.2: Number of Newton iterations for two initial condition configurations.

		Approach-1		Approach-2 (proposed)		
Order	Pts	#LU(I)	CPU(I)	#LU(II)	#LU(TR)	CPU(II)
3	1353	4299	100.15	2707	2704	88.46
4	329	1317	29.69	767	656	23.76
5	142	701	25.09	467	282	19.84
6	76	434	16.53	287	150	12.17

The order of the algorithm and the total number of time points are shown in the first and second columns. The third column shows the total number of high-order NR iterations in terms of the required LU factorizations in the first approach, followed by the overall CPU cost in seconds in this approach, indicated by "#LU(I)" and "CPU(I)", respectively. The following three columns show the reduced number of high-order NR iterations, the total number of NR iterations to obtain the initial guess based on the TR and the overall CPU time, indicated by "#LU(II)", "#LU(TR)" and "CPU(II)" respectively, as described in the second approach.

As shown in Table 8.2, the convergence characteristics of high-order methods are

improved when the proposed technique in section 6.4 is used to provide the initial guess for high-order methods.

8.3 Overall CPU efficiency

This example shows the overall improvement of the simulation time using the techniques proposed in chapter 6. The same Cauer low-pass filter with μA-741 OpAmps was simulated for 2.3msec using different orders, and the stimulus is a sinusoidal function consisting of two input frequencies, i.e., $f_1 = 1600$Hz and $f_2 = 3100$Hz. Figure 8.3 shows the transient waveforms generated by different orders of the pro-

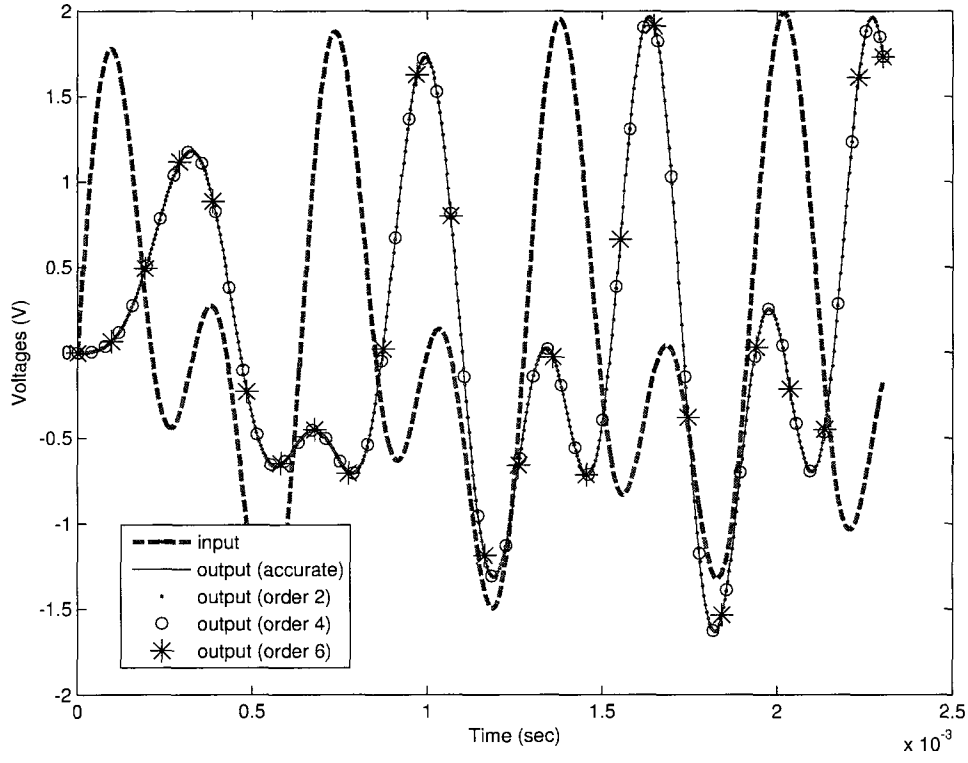

Figure 8.3: Transient simulation results for the example used in section 8.3.

posed algorithm. Tables 8.3 and 8.4 report the total CPU statistics resulting from the implementation in chapter 5 and from the proposed techniques in chapter 6, respectively.

Table 8.3 reports the results obtained by solving the augmented system (5.13) using techniques introduced in chapter 5. The maximum errors shown in the second column have been computed by running the algorithm with very small step sizes and taking the resulting waveforms as the references of comparison for all orders and all tolerances. The third and fourth columns in this table provide the total number of computed points and LU factorizations, respectively. The subsequent three columns provide a breakdown for the CPU execution times in seconds for the main tasks involved in the algorithm, where the Jacobian matrices were factorized by KLU and "Fn&Jn" denotes the total time for computing $\tilde{\boldsymbol{f}}_{n+1}$ and $\partial\tilde{\boldsymbol{f}}_{n+1}/\partial\boldsymbol{\xi}_{n+1}$. The last column provides the measured speedup relative to the CPU time of the TR, which is reported at the first row in Table 8.3.

The circuit was simulated again by solving the reduced system (6.11) of size $mN \times mN$, applying the proposed block algorithm and running TR first to enable better convergence characteristics of the high-order methods and the results are reported in Table 8.4. The same step size and number of computed points were used to maintain the same maximum error, thus are not reported. The second and third columns in this table provide the total number of LU factorizations used by block factorization algorithm to factorize the $(mN \times mN)$ Jacobian matrix (6.11), "#BLU", and the number of LU factorizations for the TR Jacobian matrix "#LU(TR)", respectively. The subsequent three columns provide a breakdown for the CPU execution times in seconds. The last column provides the measured speedup relative to the CPU time

of the TR. which is reported at the first row in Table 8.3.

Table 8.3: CPU cost for section 8.3 with ϵ_{tol} = 1e-4 (implementations in chapter 5).

Order	Error	Pts	#LU	CPU			Speedup
				LU	Fn&Jn	Other	
TR(2)	9.56e-5	5750	17173	81.87	58.69	29.98	1.00
3	9.33e-5	639	2495	33.91	14.66	8.35	3.08
4	9.96e-5	184	874	12.3	5.05	3.16	8.59
5	9.58e-5	89	511	13.96	4.4	2.32	8.56
6	9.46e-5	52	331	9.96	2.74	1.66	12.35

Table 8.4: CPU cost for section 8.3 with ϵ_{tol} = 1e-4 (implementations in chapter 6).

Order	#BLU	#LU(TR)	CPU			Speedup
			LU	Fn&Jn	Other	
3	1279	1276	23.21	11.14	6.11	4.19
4	471	490	8.46	4.2	2.24	11.45
5	294	290	8.46	3.45	1.46	12.95
6	206	190	5.86	2.25	1.13	18.67

8.4 Cascaded inverters

In this example, the circuit shown in Figure 8.4 is considered. The input stimulus is a piecewise linear waveform represented by a trapezoidal waveform having a rise/fall time of 40ns, a pulsewidth of 160ns and an amplitude of 1.2V. The model used to represent the inverters was presented in section 7.4. The transmission line is modeled by 1000 lumped RLC sections. The proposed algorithm with different orders was used to run the transient simulation and Figure 8.5 shows the transient waveforms obtained at the input and output nodes of the circuit.

In this example both TR and the proposed algorithm of order 8 were used to

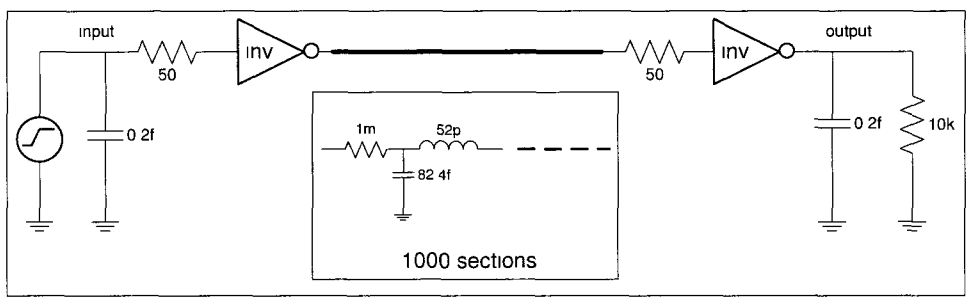

Figure 8.4: A digital circuit with two inverters connected by a transmission line.

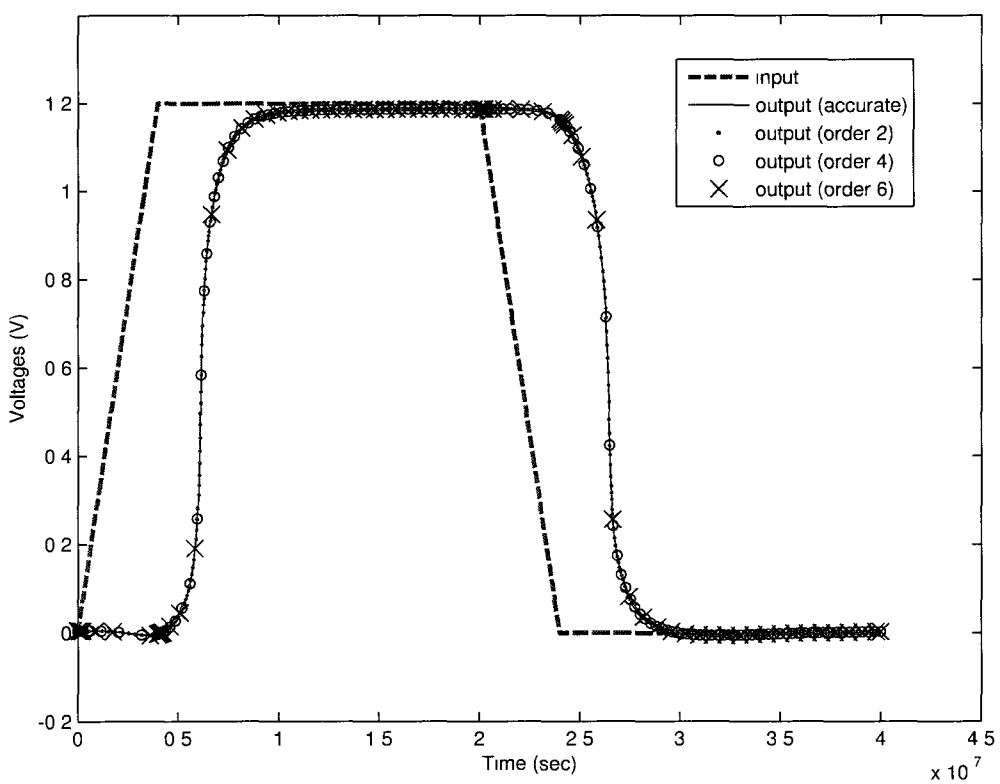

Figure 8.5: Transient simulation results for the example in section 8.4.

simulate the circuit in Figure 8.4. TR was implemented using the techniques in chapter 5, while the order 8 algorithm was implemented by adopting the proposed techniques in chapter 6.

The performance of the TR and the proposed techniques presented in chapter 6 are presented in Table 8.5 for tolerance level of 0.0001. The table also compares the performance of the proposed techniques in chapter 6 versus the performance obtained using techniques introduced in chapter 5.

Table 8.5: CPU cost for example in section 8.4 with $\epsilon_{\text{tol}} = 0.0001$.

Methods	Pts	#LU	#LU(TR)	CPU			Speedup
				LU	Fn&Jn	Other	
TR	16667	34197	0	293.62	48.98	122.84	1.00
Order 8 (chapter 5)	257	883	0	84.02	2.97	8.26	5.26
Order 8 (chapter 6)	257	664	255	30.09	2.2	5.08	12.56

It should be clear here that, with the cost of matrix factorization being the dominant component in the overall CPU, the improved techniques proposed in chapter 6 have reduced the high-order computational efforts significantly.

Chapter 9

Summary and future work

This thesis presents a new algorithm for A-stable and L-stable solutions of stiff differential algebraic equations arising from the circuit transient analysis. A brief summary and expected future works are described in the following two sections.

9.1 Summary

The new method, which is a single-step multi-derivative method, is based on Obreshkov integration formula. It utilizes high-order derivatives at the two time points of the current step. The integration formula can be arbitrarily high order by including arbitrarily high-order derivatives. The method is always A-stable if the number of the derivatives at two time points satisfies certain conditions. In addition to be A-stable, the method can be L-stable if the highest-order derivative at the past time point is one or two degree lower than the highest-order derivative at the current time point.

The Obreshkov formula is proved to have the smallest local truncation error constant among all A-stable method of the same order. This means a larger step size is

allowed to achieve the same accuracy when compared to traditional integration methods. By exploiting the effective error estimation and the step size control mechanism, the variable step size scheme is adopted to achieve better efficiency.

Implementation details to linear circuits and nonlinear circuits are discussed. Parser techniques are used to interpret the netlists and nonlinear expressions. It is not necessary to analytically differentiate the nonlinear functions before evaluating the high-order derivatives and the Jacobian matrix. Rooted trees are constructed to represent the nonlinear functions in the circuits. Each sub-tree of a rooted tree represents a simpler expression. A recursive relationship of the high order derivatives is exploited to compute the high-order derivatives and the Jacobian matrix. The properties of the rooted trees are discussed. An initialization process of the proposed algorithm is discussed and an overall algorithm is presented.

The structural characteristics of the underlying Jacobian matrix are exploited and a special tailored block factorization algorithm is discussed. A few improved implementation techniques are presented aiming at enabling the high-order methods to attain their full potentials.

Several numerical examples are given to validate the theoretical analysis and demonstrate the advantages of the proposed algorithm.

9.2 Future work

The proposed algorithm was carefully implemented and experiments showed that it is much more efficient than the traditional low-order methods. There is still room to improve the implementation. We expect to improve the implementation in the

following two ways.

Parallelism

It is possible to implement the proposed algorithm in parallel instead of computing everything by a single processor. Evaluating the nonlinear expressions and performing the LU factorizations are the two major time consuming tasks. Computing the nonlinear expressions in parallel is trivial because the nonlinear functions are independent of each other. While, in general, LU factorization is a sequential process because each step is based on previous steps; however, with the block factorization mechanism, parallelism within blocks can be implemented.

As illustrated in Figure 6.1, there are two major steps in the block left-looking factorization algorithm, i.e., block forward substitution (line 4) and $\boldsymbol{U}$-divide (line 8). Observing closely, we notice that at the block forward substitution stage, the columns in $\boldsymbol{y}$ can be computed independently of each other. Each column is only dependent on the known $\boldsymbol{L}$ and the corresponding column in $\hat{\boldsymbol{J}}(:, p)$. In the $\boldsymbol{U}$-divide stage, each block in $\boldsymbol{L}(p+1 : N, p)$ can be computed independently of other blocks after $\boldsymbol{U}(p, p)$ is obtained. Therefore, with a careful partition of the data, parallelism can be applied to the block factorization algorithm to achieve CPU speedup.

Piecewise nonlinear model

In current implementation, each nonlinear circuit element is represented by one expression in the nonlinear function vector $\boldsymbol{f}(\boldsymbol{x})$. For example, one-piece C_∞ MOSFET current model [42] is used as a substitute for the three-piece MOSFET current model

in order to make the current transition smooth. It is time consuming to evaluate the one-piece C_∞ MOSFET current model because it contains a complicated mathematical function.

The current implementation can be modified slightly to handle models with multiple functions, such as the three-piece MOSFET current model. In order to handle multiple functions, the grammar of the parser and the expression lexical analyzer have to be modified to read multiple expressions and their corresponding conditions. A new expression class, i.e., a logic term has to be created to represent the boundary conditions. The logic term class contains one operator (<, <=, ==, > or >=) and two operands (regular nonlinear expressions).

Two separate expression vectors in a single nonlinear element are created to store the multiple nonlinear expressions and their corresponding logic conditions. During computation, all the logic terms are evaluated to generate logic outputs, true or false. Only one nonlinear expression corresponding to the logic true condition is chosen to represent the current status of the nonlinear element.

Appendix A

C_∞ MOSFET source-drain current model

A.1 Model description

The single piece C_∞ MOSFET source-drain current model [42] is given as follows.

$$I_{DS} = \beta(1+\lambda(V_{DS}-V_{DS}^L))V_{DS}^L(V_{GS}^L - 0.5(1+\delta+\theta_C V_{GS}^L)V_{DS}^L)(1+\theta_S V_{GS}^L)^{-1}, \quad \text{(A.1)}$$

where

$$V_{GS}^L = \frac{1}{N_{ST}A_{AT}}\log(1+N_{ST}R_{ST}A_{ST}\exp(A_{ST}(V_{GS}-V_{TH}))+\exp(N_{ST}A_{ST}(V_{GS}-V_{TH}))), \quad \text{(A.2)}$$

$$V_{SAT} = \frac{V_{GS}^L}{1+\delta+\theta_C V_{GS}^L}, \quad \text{(A.3)}$$

$$V_{DS}^L = V_{SAT} - V_{SAT}\frac{\log(1+\exp(A_{TS}(1-V_{DS}/V_{SAT})))}{\log(1+\exp(A_{TS}))}, \quad \text{(A.4)}$$

and

$$\beta = \frac{W}{L}U_0 C_{OX}, \quad \lambda = 1.0e-3, \quad \theta_s = 0.1, \quad \delta = 0.52358, \quad \theta_C = 0.6,$$

$$A_{TS} = 7.7, \quad N_{ST} = 1.01, \quad A_{ST} = 13.6, \quad R_{ST} = 0.038.$$

V_{TH}, C_{OX}, U_0, W and L are device parameters.

A.2 Sample netlist

```
Vdd n01 gnd 15
Vin n07 gnd sin(1.6 1e3 0)
C1 n02 n03 10u
R1 n01 n03 99k
R2 n03 gnd 20k
Rd n01 n04 17.5k
Rs n05 gnd 2k
C2 n04 n06 10u
RL n06 gnd 175k
Rs_ac n02 n07 1

xm1  n04 n03 n05 n05 nmos length = 0.8u width=9.6148u
.subckt nmos nd ng ns nb length = 0.8u width=80u
    GDsub1 nb n01 cur = '1.0e-16*(exp((v(nb)-v(n01))/(25.0e-3)) - 1)'
    GDsub2 nb n02 cur = '6.952e-14*(exp((v(nb)-v(n02))/(25.0e-3)) - 1)'
    rs ns n01 1
    rd nd n02 1
    cgb ng nb 2.0e-14
    cgs ng n01 1.293e-14
    cgd ng n02 2.0e-14
    cbs n01 nb 2.0e-14
    cbd n02 nb 2.0e-14

    Gids n02 n01 cur='
    @ VGS = V(ng) -v(n01),
    @ VDS = V(n02) - v(n01),
    @ COX = 3.453e-3,
    @ U0 = 0.03,
    @ beta = width/length*U0*COX,
    @ lambda = 8.0e-4,
    @ theta_S = 0,
    @ delta = 0.277,
    @ theta_C = 0,
    @ VTH = 8.183e-01,
```

```
    @ ATS = 7.7,
    @ NST = 1.01,
    @ AST = 13.6,
    @ RST = 0.038,
    @ V_GS_L = 1/(NST*AST)*log(1 +NST*RST*AST*exp( AST * (VGS - VTH))
                + exp(NST * AST * (VGS - VTH))),
    @ V_SAT = V_GS_L/(1+delta+theta_C*V_GS_L),
    @ V_DS_L = V_SAT-V_SAT*log(1+exp(ATS*(1-VDS/V_SAT)))*1/(log(1+exp(ATS))),
    @ beta*(1+lambda*(VDS-V_DS_L))*V_DS_L * (V_GS_L-0.5*(1+delta+theta_C*V_GS_L)*V_DS_L)
                *1/(1 + theta_S* V_GS_L)
    ,
.ends
.end
```

Appendix B

Neural network description

B.1 Structure of a neural network

As indicated in section 7.4, the inverter model is approximated by a neural network. Figure B.1 shows the structure of the neural network. The hyperbolic tangent function, $\tanh(x)$, as described in Table 5.1, is chosen to be the neuron layer because it resembles the inverting function at the origin. As a result, the outputs of the neural network are weighted sum of tanh terms, $h_i \tanh(u_i)$, where the argument u_i of each tanh term is a weighted sum of v_{in} and v_{out}. w_i and h_i are weights from the neural network training.

B.2 Netlist of the inverter circuit in Figure 7.10

```
Vin1 1 gnd pwl 0 0, 0.2n 1.2, 1n 1.2, 1.2n 0, 2.0n 0
C1  1 gnd 2.0e-16
R1  1  2  50

xinv 2 3 InvGate
```

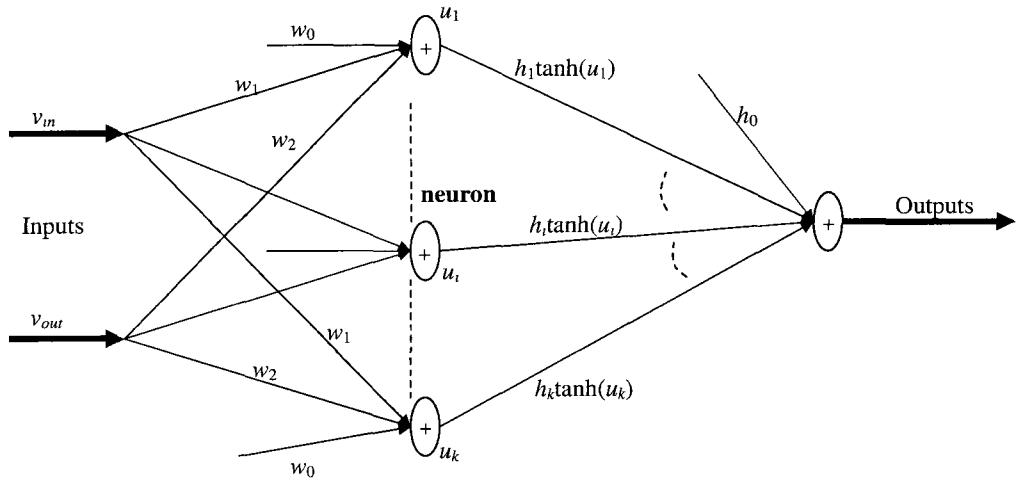

Figure B.1: Three-layer structure of the neural network.

```
C2  3 gnd 2.0e-16
R2  3 gnd 10000

.subckt InvGate n1  n2
G1  n1  GND cur = '
@   h0 = 5.059e-08,
@   w10 =     5.537, w11 =   -4.7754, w12 =   0.54448, h1 =-1.4914e-07,
@   w20 =    2.6313, w21 =   -3.1243, w22 =   -1.5827, h2 =-1.4096e-07,
@   w30 =     2.481, w31 =   -2.2587, w32 =    -2.317, h3 = 1.2026e-07,
@   w40 =   0.27459, w41 =    4.4161, w42 =   -0.1731, h4 = 1.0507e-07,
@   u1=w10+w11*v(n1)+w12*v(n2),
@   u2=w20+w21*v(n1)+w22*v(n2),
@   u3=w30+w31*v(n1)+w32*v(n2),
@   u4=w40+w41*v(n1)+w42*v(n2),
@   h0+h1*tanh(u1)+h2*tanh(u2)+h3*tanh(u3)+h4*tanh(u4)
'
G2  n2  GND cur = '
@   h0 =0.00032043,
@   w10 =   -1.2765, w11 =   0.82261, w12 =   -2.6917, h1 =-0.0016002,
@   w20 =   0.68567, w21 =  0.079837, w22 =  -0.65683, h2 = -0.004855,
@   w30 =  0.035186, w31 =  -0.95869, w32 =   0.92001, h3 =-0.0031312,
@   w40 =   0.65338, w41 =    -4.406, w42 =  -0.26359, h4 =0.00012864,
@   u1=w10+w11*v(n1)+w12*v(n2),
@   u2=w20+w21*v(n1)+w22*v(n2),
```

```
@   u3=w30+w31*v(n1)+w32*v(n2),
@   u4=w40+w41*v(n1)+w42*v(n2),
@   h0+h1*tanh(u1)+h2*tanh(u2)+h3*tanh(u3)+h4*tanh(u4)
'
C1  n1  GND q = '
@   h0 =1.4008e-15,
@   w10 =    3.6336, w11 =   -3.8301, w12 =    1.2266, h1 =-1.1078e-15,
@   w20 =    3.6513, w21 =  -0.64846, w22 =   -4.2748, h2 = 2.3035e-17,
@   w30 =    1.3685, w31 =   -3.4794, w32 =    1.0823, h3 =-1.1817e-15,
@   w40 =   0.75558, w41 =    3.6342, w42 =   -1.2274, h4 = 1.1764e-15,
@   u1=w10+w11*v(n1)+w12*v(n2),
@   u2=w20+w21*v(n1)+w22*v(n2),
@   u3=w30+w31*v(n1)+w32*v(n2),
@   u4=w40+w41*v(n1)+w42*v(n2),
@   h0+h1*tanh(u1)+h2*tanh(u2)+h3*tanh(u3)+h4*tanh(u4)
'
C2  n2  GND q = '
@   h0 = 2.126e-15,
@   w10 =   -1.7109, w11 =     3.927, w12 =   -2.8764, h1 =-2.3048e-16,
@   w20 =  -0.12504, w21 =   -1.0807, w22 =    4.1871, h2 = 5.2438e-16,
@   w30 =    1.1112, w31 =   -4.0169, w32 =   -2.2233, h3 = 2.2872e-16,
@   w40 =  -0.79794, w41 =  -0.74631, w42 =    1.3802, h4 = 3.7939e-15,
@   u1=w10+w11*v(n1)+w12*v(n2),
@   u2=w20+w21*v(n1)+w22*v(n2),
@   u3=w30+w31*v(n1)+w32*v(n2),
@   u4=w40+w41*v(n1)+w42*v(n2),
@   h0+h1*tanh(u1)+h2*tanh(u2)+h3*tanh(u3)+h4*tanh(u4)
'
.ends

.end
```

Bibliography

[1] U. M. Ascher and L. R. Petzold, *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations.* SIAM books, December 1997.

[2] J. Vlach and K. Singhal, *Computer Methods for Circuit Analysis and Design.* New York : Van Nostrand Reinhold, second ed., 1994.

[3] G. Dahlquist, "A special stability problem for linear multistep methods," *BIT*, vol. 3, pp. 27–43, 1963.

[4] C. Gear, "The simultaneous numerical solution of differential-algebraic equations," *IEEE Transactions Circuit Theory*, vol. 18, pp. 89–95, Jan. 1971.

[5] C. W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations.* Prentice-Hall, 1971.

[6] J. C. Butcher, *Numerical Methods for Ordinary Differential Equations.* John Wiley, second ed., 2008.

[7] R. K. Brayton, F. G. Gustavson, and G. D. Hachtel, "A new efficient algorithm for solving differential-algebraic systems using implicit backward differentiation formulas," *Proceedings of IEEE*, vol. 60, pp. 98–108, Jan 1972.

[8] W. van Bokhoven, "Linear implicit differentiation formulas of variable step and order," *IEEE Trans. Circ. and Sys.*, vol. 22, pp. 109–115. Feb 1975.

[9] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II, Stiff and Differential-Algebraic Problems.* Springer, second ed., 1996.

[10] J. C. Butcher, Z. Jackiewicz, and H. D. Mittelmann, "A nonlinear optimization approach to the construction of general linear methods of high order," *J. Comput. Appl. Math.*, vol. 81, no. 2, pp. 181–196, 1997.

[11] E. Gad, M. Nakhla, R. Achar, and Y. Zhou, "A-Stable and L-Stable high-order integration methods for solving stiff differential equations," *IEEE Trans. Computer-Aided Design of Integrated Circ. Sys.*, vol. 28, pp. 1359–1372, Sep. 2009.

[12] E. Gad, M. Nakhla, R. Achar, and Y. Zhou, "An absolutely-stable arbitrarily high-order implicit numerical integration method and its application to the time-domain simulation of interconnect circuits," in *Signal Propagation on Interconnects, 2007. SPI 2007. IEEE Workshop on*, pp. 186–187, may 2007.

[13] Y. Zhou, E. Gad, M. Nakhla, and R. Achar, "Structural characterization and efficient implementation techniques for A-stable high-order integration methods (under review)," *IEEE Trans. Computer-Aided Design of Integrated Circ. Sys.*

[14] K. Burrage, *Parallel and Sequential Methods for Ordinary Differential Equations.* New York: Oxford, 1995.

[15] J. D. Lambert, *Computational Methods in Ordinary Differential Equations.* New York: Wiley, 1973.

[16] A. Iserles and S. P. Nørsett, *Order Stars.* Chapman & Hall, 1991.

[17] B. L. Ehle, "High order A-stable methods for the numerical solution of differential equations," *BIT*, vol. 8, pp. 276–278, 1968.

[18] J. C. Butcher, "Implicit Runge–Kutta process," *Math. Comp.*, vol. 18, pp. 50–64, 1964.

[19] J. C. Butcher, "General linear methods," *Acta Numerica*, vol. 15, pp. 157–256, 2006.

[20] J. C. Butcher, "Diagonally-implicit multi-stage integration methods," *Applied Numerical Mathematics*, vol. 11, pp. 347–363, March 1993.

[21] I. Lei and S. P. Nørsett, "Superconvergence for multistep collocation," *Math. Comp.*, vol. 52, pp. 65–79, Jan. 1989.

[22] W. Liniger and R. A. Willoughby, "Efficient integration methods for stiff systems of ordinary differential equations," *SIAM Journal on Numerical Analysis*, vol. 7, no. 1, pp. 47–66, 1970.

[23] N. Obreshkov, "Sur les quadratures mécaniques," *(Bulgarian, French Summary) Akad. Nauk.*, vol. 65, pp. 191–289, 1942.

[24] G. Baker and P. Graves-Morris, *Padé approximants.* Encyclopedia of Mathematics, New York, USA: Cambridge Univ Press, 2nd ed., 1996.

[25] G. Birkhoff and R. S. Varga, "Discretization errors for well-set Cauchy problems," *J. Math. and Physics*, vol. 44, pp. 1–23, 1965.

[26] S. Nørsett, "One-step methods of Hermite type for numerical integration of stiff systems," *BIT*, vol. 14, pp. 68–77, 1974.

[27] B. Ehle, *On Padé Approximations to the Exponential Function and A-stable Methods for the Numerical Solution of Initial Value Problems.* Ph.D. thesis, University of Waterloo, 1969.

[28] G. Wanner, E. Hairer, and S. Nørsett, "Order stars and stability theorems," *BIT*, vol. 18, pp. 475–489, 1978.

[29] A. Nordsieck, "On numerical integration of ordinary differentiation equations," *Math. Comp.*, vol. 16, pp. 22–49, 1962.

[30] C.-W. Ho, A. Ruehli, and P. Brennan, "The modified nodal approach to network analysis," *IEEE Transactions on Circuits and Systems*, vol. 22, pp. 504–509, June 1975.

[31] A. Griewank, *Evaluating Derivatives: Principles and Applications of Algorithmic Differentiation.* SIAM, 2000.

[32] E. Gad, R. Khazaka, M. Nakhla, and R. Griffith, "A circuit reduction technique for finding the steady-state solution of nonlinear circuits," *IEEE Trans. Microwave Theory Tech.*, vol. 48, pp. 2389–2396, Dec. 2000.

[33] P. K. Gunupudi and M. S. Nakhla, "Nonlinear circuit-reduction of high-speed interconnect networks using congruent transformation techniques," *IEEE Transactions on Advanced Packaging*, vol. 24, no. 3, pp. 317–325, August, 2001.

[34] T. A. Davis and E. Palamadai, *User Guide for KLU and BTF.* Dept. of Computer and Information Science and Engineering, University of Florida, 2009.

[35] G. Corliss, A. Griewank, P. Henneberger, G. Kirlinger, F. A. Potra, and H. J. Stetter, "High-order stiff ODE solvers via automatic differentiation and rational prediction," in *In Lecture Notes in Comput. Sci*, pp. 114–125, Springer, 1996.

[36] T. A. Davis, E. P. Natarajan, and A. Inc, "Algorithm 8xx: KLU, a direct sparse solver for circuit simulation problems," *ACM Transactions on Mathematical Software.*

[37] G. W. Stewart, "On the solution of block Hessenberg systems," *Numerical Linear Algebra with Applications*, pp. 287–296, 1995.

[38] P. Favati, G. Lotti, and O. Menchi, "Non-recursive solution of sparse block Hessenberg systems," *Numerical Linear Algebra with Applications*, vol. 11, no. 4, pp. 391–406, Apr 2004.

[39] A. Mehrotra and A. Somani, "A robust and efficient harmonic balance (HB) using direct solution of HB Jacobian," in *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, (New York, NY, USA), pp. 370–375, ACM, 2009.

[40] J. R. Gilbert and T. Peierls, "Sparse partial pivoting in time proportional to arithmetic operations," *SIAM J. Sci. Statist. Comput.*, vol. 9, pp. 862–874, 1988.

[41] G. H. Golub and C. F. Van Loan, *Matrix Computations (3rd ed.)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[42] C. C. McAndrew, B. K. Bhattacharyya, and O. Wing, "A single-piece C_∞-continuous MOSFET model including subthreshold conduction," *IEEE Electron Device Letters*, vol. 12, no. 10, pp. 565–567, 1991.

[43] C. Amin, C. Kashyap, N. Menezes, K. Killpack, and E. Chiprout, "A multi-port current source model for multiple-input switching effects in CMOS library cells," in *DAC'06: Proceedings of the 43rd annual conference on Design automation*, (New York, NY, USA), pp. 247–252, 2006.

[44] A. S. Sedra and K. C. Smith, *Microelectronic Circuits*. New York: Oxford, 2004.