Improvements in Obreshkov-based High-Order Circuit Simulation Method

by

Yaoyao Lin

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the M.A.Sc degree in
Electrical and Computer Engineering

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

© Yaoyao Lin, Ottawa, Canada, 2015

Abstract

The transient time-domain simulation, of the circuit response, is a fundamental component in the Computer-Aided Design tools of all integrated circuit and systems. It is typically desirable that a method adopted in the transient circuit simulator be of high-order and numerically stable. The two requirements, however, proved to be in conflict with each other, especially in the larger class of methods that were used in traditional circuit simulators.

Recent work based on utilizing the Obreshkov formula has proved that it is possible to combine the high order with the numerical stability.

The objective of this thesis is to show how the present implementation of the Obreshkov-based method can be improved and generalized to handle different types of circuits. The first aspect of improvement targets the computation of the high-order derivatives required by the Obreshkov formula. The second aspect of improvement, presented in the thesis, develops a generalized formulation that takes into account the presence of nonlinear memory elements, whose nonlinearity is based on a capacitive or inductive-based nonlinear model.

Acknowledgements

My deepest gratitude goes first and foremost to Dr. Emad Gad, my supervisor, for his constant encouragement and guidance. He has walked me through all the stages of the writing of my thesis. Without his illuminating instruction, this thesis could not have reached its present form.

My sincere thanks are also to Dr. Michel Nakhla from whose lectures I benefited greatly.

I should finally like to express my gratitude to my beloved parents and my wife who helped me out of difficulties and supported me without a word of complaint.

Contents

1 Introduction **3**

1.1 General Problem Description . . . 3

1.2 Objectives . . . 4

1.3 Thesis contributions . . . 5

1.4 Thesis organization . . . 5

2 Review of Classical Transient Simulation **6**

2.1 Main Motivation . . . 6

2.2 Order of the method . . . 7

2.3 Stability Domain . . . 8

2.3.1 Examples of Stability Characteristics . . . 9

2.4 Transient Circuit Simulation . . . 14

2.5 A-stability vs L-stability . . . 16

2.6 Order vs Stability . . . 17

2.7 Numerical Verification of the Order . . . 18

3 Review of the Obreshkov-based Method **25**

3.1 Order and Stability Domain of the Obreshkov-based Method . . . 26

3.2 Obreshkov-based Circuit Simulation . . . 27

3.3 Efficient Factorization of the Jacobian Matrix . . . 30

3.4 Evaluating nonlinear vector and derivatives 35
3.5 Discussion 39

4 Improved Processing of Rooted Tree 41
4.1 Illustration of the Operation 41
4.2 Observations on the basic operation 44
4.2.1 Background to Rooted Trees 44
4.3 Improved Computation 46
4.4 Validation Example 48

5 Formulation of the Obreshkov-based Transient Circuit Simulation in the Presence of Nonlinear Memory Elements 54
5.1 Obreshkov-based Formulation for Circuits with Nonlinear Capacitors . . 55
5.2 Formulation of the Jacobian matrix 59
5.3 Implementation and Computational Complexity 62
5.3.1 Notes on Implementation 62
5.3.2 Analysis of Computational Complexity 63
5.3.3 Numerical Stability 66
5.4 Numerical Examples 66
5.4.1 Example 1: Circuit with Nonlinear Capacitor 67
5.4.2 Example 2: Large Circuit Example 69
5.4.3 Example 3: Large Circuit with Nonlinear Capacitors Example . . 73

6 Conclusion and Future Work 75
6.1 Concluding Remarks 75
6.2 Future Work 76

A Netlist of the Double-Balanced Mixer Circuit 77

A.1 Level 1 77
A.2 Level 2 80
A.3 Level 3 83

List of Tables

2.1 Coefficients of the Gear's method 14

3.1 Some Formulas for the Derivatives of Simple Functions 39

4.1 Parameters in the above definition expression 51

4.2 Comparison of the value in nonlinear vector $\tilde{\boldsymbol{\rho}}$ 53

5.1 CPU comparison between the Obreshkov-based and Gear's methods for the TL circuit of Figure 5.6 73

List of Figures

2.1 Absolute stability region for Forward Euler (pink area) 10

2.2 Absolute stability region for Backward Euler (pink area) 11

2.3 Absolute stability region for Trapezoidal Rule (pink area) 12

2.4 Absolute stability region of Gear's method from order 1 to 6 (pink area) 13

2.5 A simple RLC circuit 20

2.6 Behavior of the actual error (versus the step size h) for approximation generated by the Gear's method with $s = 1$ 22

2.7 Behavior of the actual error (versus the step size h) for approximation generated by the Gear's method with $s = 2$ 22

2.8 Behavior of the actual error (versus the step size h) for approximation generated by the Gear's method with $s = 3$ 23

2.9 Behavior of the actual error (versus the step size h) for approximation generated by the Gear's method with $s = 4$ 23

2.10 Behavior of the actual error (versus the step size h) for approximation generated by the Gear's method with $s = 5$ 24

2.11 Behavior of the actual error (versus the step size h) for approximation generated by the Gear's method with $s = 6$ 24

3.1 Rooted tree representing the diode current 36

4.1 JFET transient analysis model. 49

4.2 Schematic of the double-balanced mixer. 49

4.3 CPU Cycles of Evaluating the Nonlinear Vector 52

4.4 CPU Cycles of Evaluating the Jacobian Matrix 52

4.5 Comparison of the output of node Vout1 53

5.1 Schematic for the circuit used to obtain the results of Figure 5.2 64

5.2 Illustrating the growth in the complexity of matrix factorization versus the circuit MNA size, N, for the low-order TR and high-order Obreshkov-based methods 65

5.3 Illustrating the growth in the complexity of triangular (F/B) solution factorization versus the circuit MNA size, N, for the low-order TR and high-order Obreshkov-based methods 65

5.4 A circuit example used in numerical validation. $V_S = 1.25 \times sin(2000\pi t)$, $I = 10^{-15} \times (\exp((V_{n_1} - V_{n2})/0.025) - 1)$, $R = 1\Omega$, $C = 10^{-3} \times e^{V_{n_2}}$ . . 68

5.5 The behavior of the error computed by (5.32) vs. the step size h_1, for different values of k and m. 69

5.6 A TL-based circuit with $N = 18,241$ variables. 70

5.7 Transient response at the far-end of the 2nd line of segment 2 in the circuit shown in Figure 5.6. 71

5.8 Relative error in the transient response computed using the Gear's method, orders 4,5, and 6. 72

5.9 A TL-based circuit with nonlinear capacitors. 73

5.10 Transient response at the far-end of the 2nd line of segment 2 in the circuit shown in Fig. 5.9 74

Nomenclature

α_i, β_i	Integration coefficients in modified Obreshkov formula
$\alpha_{i,m}$	Integration coefficients in Obreshkov formula
$\mathbb{R}, \mathbb{C}$	Real,comoplex variable space
$\tilde{\boldsymbol{G}}, \tilde{\boldsymbol{C}}, \tilde{\boldsymbol{f}}, \tilde{\boldsymbol{b}}, \boldsymbol{\xi}$	Matrices and vectors describing the circuit when applied on modified Obreshkov formula
$\tilde{\boldsymbol{J}}$	Block Jacobian matrix
$\boldsymbol{u}(t)$	A vector consists of independent stimuli in MNA formulation
$\boldsymbol{f}(\boldsymbol{x}(t))$	A vector consists of nonlinear function in MNA formulation
$\boldsymbol{G}, \boldsymbol{C}$	MNA matrices describing a circuit
$\boldsymbol{J_{i,j}}$	The entry (i, j) of matrix $\tilde{\boldsymbol{J}}$
$\boldsymbol{R}_u$	Matrix-valued u^{th} Taylor series coefficients of $\boldsymbol{J}_{0,0}$
$\boldsymbol{x}_n$	Approximation of $\boldsymbol{x}(t)$ at $t = t_n$
h	Current step size
h_n	Step size between t_{n-1} and t_n

BE	Backward Euler
DE	Differential Equations
MNA	Modified Nodal Analysis
NR	Newton-Raphson method

Chapter 1

Introduction

1.1 General Problem Description

The circuit time-domain transient simulator is an important component in the design automation tools and is essential for the analysis and validation of electronic circuits. Transient simulation of electronic circuits is essentially the process of solving the system of differential equations (DEs) that model a given circuit, numerically. One of the main problems that impeded progress on this front was the inherent conflict between the order and the stability of the methods used in the DEs numerical solution in the traditional SPICE-based simulators [1].

Reference [2] has addressed this problem through presenting a new method, based on the Obreshkov formula [3], that does not suffer from this conflict. It was shown that this method allows combining the desired property of A-stability (or L-stability) with high-order solution of DEs. As a result, this method enabled transient simulation to run faster than the classical low-order methods used in the SPICE-based conventional simulators such as the Trapezoidal rule or the Gear's method. Moreover, recent advances reported in [4–6] have also consistently demonstrated that this method yields increasing speedup

for the time-domain transient simulation. It is also worth noting the important recent efforts to break the barrier of order/stability, e.g., based on the Runge-Kutta method in [7] and [8], the adaptive grid multistep method in [9], or the backward scaled method of [10].

1.2 Objectives

This thesis has been motivated by two main objectives. The first objective was to explore whether there is room in the current implementation procedure of the Obreshkov-based method for further improvement. More particularly, the work in this thesis take a closer look at one of the basic components in the computational steps presently used in the implementation of the Obreshkov-based method. The component considered in this study is related to the computation of the high-order derivatives, which is the essential part that gives the Obreshkov method its high-order and stability properties. This computation is carried out using the notion of rooted tree, where the nonlinear expressions representing the circuits nonlinear devices are represented by a graph-based rooted tree. This thesis considers a specific aspect of the way rooted trees are being utilized and suggests an alternative utilization technique with the objective of reducing the amount of nonessential overhead that is not related to the basic mathematical computation.

The second objective is to generalize the current formulation of the Obreshkov-based method so that it can handle circuits with nonlinear memory elements (i.e. capacitors or inductors) whose nonlinearity is based on describing the capacitance or the inductance as a nonlinear function expression of the circuit variables. This is because the published formulation assumes that the memory elements are either linear, or, in case they are nonlinear, have a charge-based or flux-based nonlinear expression that provides the charge of the capacitor or the flux of the inductor as a nonlinear function expression of the circuit variables. The work under this second objective expands on the guiding principles

that enabled the application of the Obreshkov-based method to circuits whose nonlinear memory elements are capacitance-based or inductance-based elements, as opposed to flux-based or charge-based elements.

1.3 Thesis contributions

The contributions of this thesis flow from the objectives outlined in the previous section.

The first contribution presents an alternative approach to handle the rooted tree that manages to reduce the overhead in computing the high-order derivatives. Numerical experiment showed that a reduction of 20-25% can be achieved. This work will be presented in Chapter 4.

The second contribution provides the derivations required to generalize the formulation of the Obreshkov method so that it can handle circuits whose nonlinear memory elements are given by the capacitance or the inductance-based nonlinearity. The contribution from this work was published in [11], and will be described in Chapter 5.

1.4 Thesis organization

This thesis consists of 6 chapters, organized as follows. Chapter 2 reviews the common integration methods and Chapter 3 reviews the high order method based on Obreshkov formula. Chapter 4 presents the improved processing of rooted tree. Chapter 5 presents the Obreshkov-based formulation for circuits with nonlinear capacitors and corresponding numerical experiments. Additionally, The last chapter makes a conclusion of the whole thesis.

Chapter 2

Review of Classical Transient Simulation

This chapter presents the necessary background on the transient time-domain simulation methods utilized by the classical SPICE-based engine. Particular emphasis will be placed on two main aspects of those methods, namely stability and order of the method.

2.1 Main Motivation

To motivate the discussion presented in this chapter, I consider a system of nonlinear differential equations of the form

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(x, t) \tag{2.1}$$

where $\boldsymbol{x} \in \mathbb{R}^N$ and $\boldsymbol{f}(x, t) : \mathbb{R}^{N+1} \to \mathbb{R}^N$ is a nonlinear mapping. The main goal in any method employed to solve the above differential equation is to approximate the exact solution $\boldsymbol{x}(t)$ at discrete time points $t_0, t_1, t_2, \ldots$ The approximations generated by the method will be denoted generally by $\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots$, whereas the exact values of the solution will be expressed by $\boldsymbol{x}(t_0), \boldsymbol{x}(t_1), \boldsymbol{x}(t_2), \ldots$

There are two important concepts that characterize any method being used to solve the above system. These are the order and stability domain of the method.

2.2 Order of the method

A method is said to be of order p when the following condition is satisfied.

$$\boldsymbol{x}_n = \boldsymbol{x}(t_n) + C h_n^{p+1} \frac{d^{p+1}}{dt^{p+1}} \boldsymbol{x}(t) \Big|_{t=t_n} + O(h_n^{p+2}). \tag{2.2}$$

where $h_n = t_n - t_{n-1}$, C is a constant called the error constant [12] of the method whose magnitude indicates the accuracy of the method, and $O(h^v)$ refers to the terms that are proportioned to powers of h that are larger than or equal to v.

One possible way to estimate the error of the method is via the Local Truncation Error (LTE) which is defined as the term in the above expansion given by

$$\text{LTE} = C h_n^{p+1} \frac{d^{p+1}}{dt^{p+1}} \boldsymbol{x}(t) \Big|_{t=t_n}. \tag{2.3}$$

LTE captures the asymptotic behavior of the approximation versus the step size h_n. For example, for increasingly smaller step size h_n, the error in the approximation tends to become proportional to h_n^{p+1}, for a method of order p, which matches the order of h_n in the LTE term above.

It should be obvious that a method of high-order is more desirable than a method of low-order, the reason being that a high-order method is capable of increasing the step size without having to incur a large error.

However, it is not just the order of the method that determines the efficiency of the method. Other important factors such as the stability of the method must be considered to evaluate the efficiency of the method.

2.3 Stability Domain

To study the concept of the stability domain of any given method, I consider the scalar test problem defined by

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \lambda x(t) \tag{2.4}$$
$$x(0) = x_0$$

where $\lambda \in \mathbb{C}$ is a complex parameter [13].

The stability domain of a given method refers to the values of λ in the complex plane, where the approximations generated by the method (for the scalar test problem) of (2.4) satisfy

$$x_m < x_n \quad \text{when} \qquad m > n \tag{2.5}$$

A method is said to be A-stable if its approximation to (2.4) satisfies (2.5) for all value of $\lambda \in \mathbb{C}^-$, where $\mathbb{C}^-$ is the left side of the complex plane.

Naturally, if $\lambda \in \mathbb{C}^-$, then the exact solution of the scalar test problem (2.4) is monotonically decreasing function of time. Therefore, it is reasonable to demand that a method deployed to approximate that solution must also exhibit the same property of monotonically decreasing function of time, and hence (2.5). In other words, the property of A-stability is a desirable property that one would like to be available in a given method used to approximate solution of the DE.

In the following sections, I consider several examples of common integration methods and study their stability characteristics.

2.3.1 Examples of Stability Characteristics

Stability Characteristics of Forward Euler Method

The Forward-Euler integration method (FE) is defined by

$$x_{n+1} = x_n + hx'_n \tag{2.6}$$

where h is the length of the time step, $h = t_{n+1} - t_n$. In other words, approximation for $x(t)$ at $t = t_{n+1}$ is obtained from the approximation of $x(t)$ and the approximation of its derivatives, both calculated at the previous time point, i.e., at $t = t_n$.

To study the stability characteristics of the FE method, I assume that it has been used to approximate the solution to the scalar test problem in (2.4). In this case, the approximation at $t = t_{n+1}$, or x_{n+1}, will be expressed in terms of x_n in the following manner

$$x_{n+1} = (1 + h\lambda)x_n \tag{2.7}$$

which can be used recursively to express x_{n+1} in terms of the initial solution x_0 in the following manner

$$x_{n+1} = (1 + h\lambda)^{n+1}x_0 \tag{2.8}$$

Thus considering (2.7) in the light of the definition of the domain of stability (2.5), it is easy to deduce the domain of stability of the FE method by determining the values of λ in the complex domain for which

$$|1 + h\lambda| < 1 \tag{2.9}$$

To delineate this domain, I substitute $h\lambda$ using the complex polar form, $x + jy$, in (2.9), which leads to

$$|1 + x + iy| < 1 \quad \text{or} \quad (1 + x)^2 + y^2 < 1. \tag{2.10}$$

The above relation describes a circle in the complex plane whose center is at $x = -1, y = 0$ and its radius is equal to 1, as shown by the circle in Figure 2.1

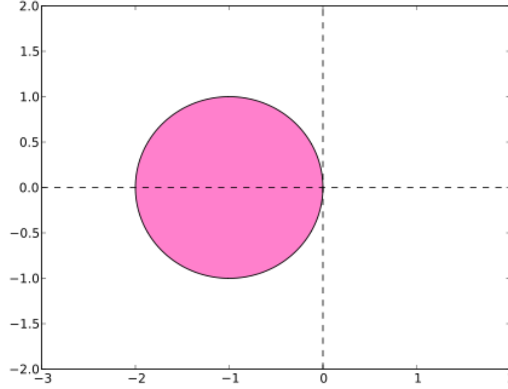


Figure 2.1: Absolute stability region for Forward Euler (pink area)

The above discussion, therefore, shows that in order for the FE to yield decaying approximation to the solution of the scalar test problem in (2.4) the value of $h\lambda$ must lie in the unit circle shown in Figure 2.1. This observation must be contrasted to the exact solution of the scalar test problem, which is given by the closed form expression:

$$x(t_{n+1}) = e^{\lambda t_{n+1}} x(0) \tag{2.11}$$

which is a decaying solution for all values of λ in the left-half plane of the complex plane.

Therefore, if the approximations resulting from the FE are to be at least *qualitatively* similar to the exact solution, the step size h must be reduced to make $h\lambda$ fall inside the unit circle. It should be obvious that h may have to be excessively reduced if the value of λ is very large in magnitude, so that the approximations follow a decaying curve similar the exact solution.

Another evidently obvious fact is that FE is not A-stable, since approximation x_m

and x_n do not necessarily satisfy (2.5) for all $\lambda \in \mathbb{C}^-$. Indeed, depending on the value of h, (2.5) may or may not be satisfied for all values of $\lambda \in \mathbb{C}^-$.

Stability Characteristics of the Backward Euler (BE) Method

The BE method is defined as follows

$$x_{n+1} = x_n + hx'_{n+1} \tag{2.12}$$

using the scalar test problem in (2.12) yields the following result for x_{n+1} in terms of x_n

$$x_{n+1} = (1 - h\lambda)^{-1}\, x_n \tag{2.13}$$

Proceeding in similar steps to delineate the stability domain for the BE, I consider the values of $h\lambda$ for which

$$|(1 - h\lambda)^{-1}| < 1 \tag{2.14}$$

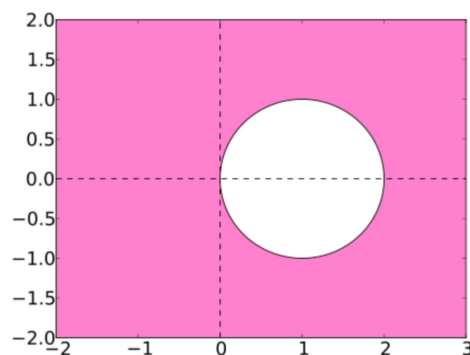


Figure 2.2: Absolute stability region for Backward Euler (pink area)

It should be straightforward to note here that the domain of stability extends over all the complex plane with the exception of those values of $h\lambda$ that fall in the unit circle centered at $x = 1$ and $y = 0$, and shown by the white circle of Figure 2.2.

A noteworthy observation here is that BE yields decaying approximations to the scalar test problem for all values of λ in the left half plane of the complex domain , and regardless of the step size h. This method is therefore A-stable.

Stability Characteristics of the Trapezoidal Rule

The Trapezoidal Rule (TR) method is defined as follows.

$$x_{n+1} = x_n + \frac{h}{2}(x'_n + x_{n+1}) \tag{2.15}$$

Applying the TR to approximate the solution of the scalar test problem (2.15) yields the following result for x_{n+1} in terms of x_n

$$x_{n+1} = \left(1 - \frac{h\lambda}{2}\right)^{-1} \left(1 + \frac{h\lambda}{2}\right) x_n \tag{2.16}$$

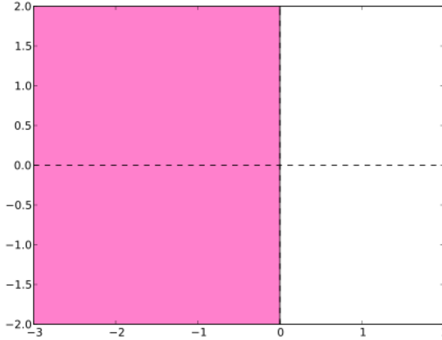


Figure 2.3: Absolute stability region for Trapezoidal Rule (pink area)

Proceeding in the same steps as with BE to delineate the stability domain for the TR, I consider the values of $h\lambda$ for which $\Re e(h\lambda) < 0$, i.e, for values of $\lambda \in \mathbb{C}^{-}$, where it can be easily seen that the domain for which the TR is stable, is all the values in the

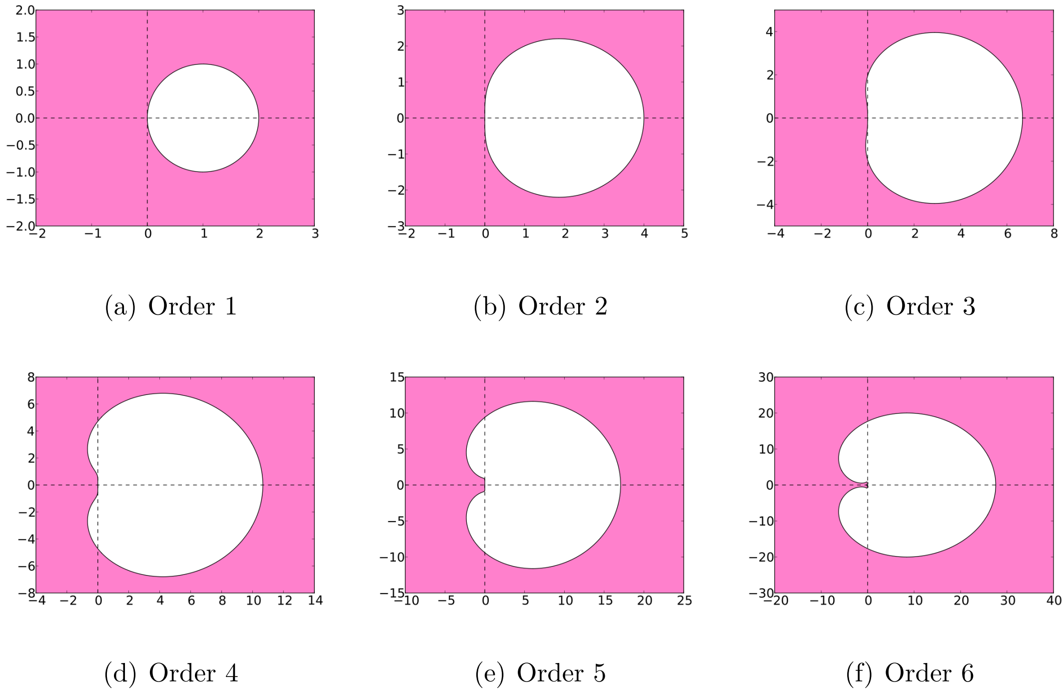
left-half plane of the complex domain, i.e. $\mathbb{C}^-$. This fact is indicated by the pink area in Figure 2.3.

Stability Characteristics of the Gear's Method

The Gear's method [14] is a family of methods which can be described as follows

$$\sum_{k=0}^{s} \alpha_k x_{n+k} = h\beta x'_{n+s} \tag{2.17}$$

where α_k and β are the coefficients which are chosen to make the method achieve order s. The k-step Gear's method is an order k method.



(a) Order 1 (b) Order 2 (c) Order 3

(d) Order 4 (e) Order 5 (f) Order 6

Figure 2.4: Absolute stability region of Gear's method from order 1 to 6 (pink area)

To obtain the coefficients $\alpha_k, k = 0, 1, \ldots, s$ and β in (2.17), I substitute $x(t_{n+s-k})$, $k = 0, \ldots, s$, and $\left.\frac{\mathrm{d}x}{\mathrm{d}t}\right|_{t=t_{n+s}}$ for x_{n+s-k}, $k = 0, \ldots, s$, and x'_{n+s} in (2.17), expand each term in its Taylor series, and equate to zero the terms up the sth derivative.

For example, when I set $s = 2$, the Gear's method of order 2 can be described as follows [15]

$$x_{n+2} - \frac{4}{3}x_{n+1} + \frac{1}{3}x_n = \frac{2}{3}hx'_{n+2} \tag{2.18}$$

Table 2.1 lists the values of the coefficients α_k and β for the values of $s = 1, 2, 3, 4, 5, 6$

s	α_0	α_1	α_2	α_3	α_4	α_5	α_6	β
1	-1	1						1
2	$\frac{1}{3}$	-$\frac{4}{3}$	1					$\frac{2}{3}$
3	-$\frac{2}{11}$	$\frac{9}{11}$	-$\frac{8}{11}$	1				$\frac{6}{11}$
4	$\frac{3}{25}$	-$\frac{16}{25}$	$\frac{36}{25}$	-$\frac{48}{25}$	1			$\frac{12}{25}$
5	-$\frac{12}{137}$	$\frac{75}{137}$	-$\frac{200}{137}$	$\frac{300}{137}$	-$\frac{300}{137}$	1		$\frac{60}{137}$
6	$\frac{10}{147}$	-$\frac{72}{147}$	$\frac{225}{147}$	-$\frac{400}{147}$	$\frac{450}{147}$	-$\frac{360}{147}$	1	$\frac{60}{147}$

Table 2.1: Coefficients of the Gear's method

Figure 2.4 shows the stability domains for the Gear's method for orders $s = 1, 2, 3, 4, 5$ and 6. It should be noted from these figures that the Gear's method ceases to be A-stable when the order s exceed 2. In fact this is one example that shows that the high-order in the method comes at the expense of the loss of A-stability.

2.4 Transient Circuit Simulation

The previous discussion assumed that the system of differential equations is either in the form of Ordinary Differential Equations such as in (2.1) or the scalar test problem in (2.4). General nonlinear circuits, however, are described by a system of mixed set of differential and algebraic equation (DAE) that result from applying the Modified Nodal Analysis (MNA) approach [16]. The MNA approach leads to the following system of

equations

$$\boldsymbol{G}\boldsymbol{x}(t) + \boldsymbol{C}\frac{\mathrm{d}\boldsymbol{x}(t)}{\mathrm{d}t} + \boldsymbol{f}(\boldsymbol{x}(t)) = \boldsymbol{u}(t), \tag{2.19}$$

where $\boldsymbol{C}$ and $\boldsymbol{G} \in \mathbb{R}^{N\times N}$ are matrices expressing the memory and memoryless elements in the circuits. Additionally, $\boldsymbol{x}(t) \in \mathbb{R}^N$ is the vector of unknown node voltage waveforms, appended with the waveforms of currents in inductors and voltage sources, $\boldsymbol{f}(\boldsymbol{x}(t))$ is a vector of nonlinear currents of the memoryless elements, $\boldsymbol{u}(t) \in \mathbb{R}^N$ is the vector representing the independent stimuli, and N is the total size of the MNA circuit formulation.

In this section, I would like to illustrate the application of a given method to approximate the solution to the MNA differential equations in (2.19). For that purpose, I will consider the BE method as a vehicle to illustrate the process of the transient time-domain simulation.

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + h\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t}\bigg|_{t=t_{n+1}} \tag{2.20}$$

Substituting from the MNA(2.19) in the above system leads to the following system of equations

$$(\boldsymbol{C} + h\boldsymbol{G})\boldsymbol{x}_{n+1} + h\boldsymbol{f}(\boldsymbol{x}_{n+1}) = h\boldsymbol{u}(t_{n+1}) + \boldsymbol{C}\boldsymbol{x}_n \tag{2.21}$$

The above system is nonlinear system of equations in $\boldsymbol{x}_{n+1}$ that can be solved using the Newton method. The Newton method proceeds from an initial guess, say $\boldsymbol{x}_{n+1}^{(0)}$, and iteratively updates it using a correction $\Delta\boldsymbol{x}$ given by

$$\Delta\boldsymbol{x} = \boldsymbol{J}^{-1}\Big|_{\boldsymbol{x}=\boldsymbol{x}_{n+1}^{(0)}}\boldsymbol{\Phi}(\boldsymbol{x}_{n+1})\Big|_{\boldsymbol{x}_{n+1}=\boldsymbol{x}_{n+1}^{(0)}} \tag{2.22}$$

where $\boldsymbol{J} \in \mathbb{R}^{N\times N}$ is the Jacobian matrix given by

$$\boldsymbol{J} = \boldsymbol{C} + h\boldsymbol{G} + h\frac{\partial\boldsymbol{f}}{\partial\boldsymbol{x}} \tag{2.23}$$

and $\boldsymbol{\Phi}(\boldsymbol{x}_{n+1})$ is the error in satisfying the system of equations defined by

$$\boldsymbol{\Phi}(\boldsymbol{x}_{n+1}) = (\boldsymbol{C} + h\boldsymbol{G})\boldsymbol{x}_{n+1} + h\boldsymbol{f}(\boldsymbol{x}_{n+1}) - \boldsymbol{C}\boldsymbol{x}_n - h\boldsymbol{u}(t_{n+1}) \tag{2.24}$$

Typically, the Newton method converges in two-three iterations if the initial guess $\boldsymbol{x}_{n+1}^{(0)}$ is taken from the previous time point, i.e., if $\boldsymbol{x}_{n+1}^{(0)} = \boldsymbol{x}_n$. The Jacobian matrix is typically very sparse and can be factorized efficiently using sparse factorization techniques.

2.5 A-stability vs L-stability

The presentation in the previous section of some integration methods and their stability characteristics highlighted the importance of stability in circuit simulation. For example, if a method whose stability domain does not cover the entire left half plane of the complex domain (non A-stable), then the approximations generated by the method are bound to become unstable if the value of $h\lambda$ lies outside the stability domain. This may be the case even though the exact solution is characteristically stable, as in the case of the scalar test problem with $\lambda \in \mathbb{C}^-$.

Thus, it may be necessary to identify the ideal integration method as a method that is A-stable, so that approximations obtained from the method are stable whenever the circuit modeled by the differential equation is naturally stable. This is indeed the case in circuit simulation , since circuits are designed to be stable, and simulations of a stable circuit must also be stable.

Although A-stability is by far the most important criteria in characterizing the stability of a given integration method, another concept of stability, known as the L-stability, occupies an equally important place in the study of the numerical methods used in solving differential equations.

To illustrate the concept of L-stability, I consider again the scalar test problem in

(2.4). Here a method is said to be L-stable if its successive approximations satisfy[17]

$$\lim_{|\lambda|\to\infty}\left|\frac{x_{n+1}}{x_n}\right| = 0 \tag{2.25}$$

According to the above definition, the BE is L-stable, whereas the TR is not.

The importance of L-stability can be appreciated by assuming $\lambda = x + jy$ to lie deep in the left half plane with large negative real part, i.e., $x \to -\infty$. In this case, the exact solution to the scalar test problem decays very fast. By contrast, the TR approximations to the scalar test problem tend to be sustained for over a large number of time steps, since the ratio between successive approximation become in the limit as $|\lambda| \to \infty$

$$\lim_{|\lambda|\to\infty}\left|\frac{x_{n+1}}{x_n}\right| = 1 \tag{2.26}$$

Here, although TR is A-stable, its lack of L-stability can cause numerical troubles in simulating the so-called stiff circuits, which are characterized by eigenvalues falling deep in the left half plane of the complex domain.

2.6 Order vs Stability

Stability of the method describes the qualitative aspect of the numerical method used to approximate the solution to a given system of differential equations. Qualitative in this sense means that if the exact solution to the differential equations is stable, then the approximation must also be stable, i.e., the method should also preserve the stability of the system that the method is supposed to simulate [18].

The order of the method, on the other hand, is what may be considered as the quantitative aspect, in the sense that it quantifies how good an approximation is the value obtained by the method to the exact solution.

A full discussion on the order of the method is beyond the scope of this thesis. However it will be stated here the order of the few examples presented earlier. The order p for the FE and BE is equal to 1, whereas for the TR it is equal to 2.

In an ideal world, the method used to numerically approximate the solution to system of differential equation must be A-stable or L-stable with an arbitrary high-order.

However, the combination of stability and high-order has always been problematic to say the least. Recent work on the Obreshkov-based method has proved that the tension between the order and stability can be eliminated if the integration formula adopts the high-order derivatives of $x(t)$ in a prescribed manner. The goal of the next chapter is to review the method based on the Obreshkov formula.

2.7 Numerical Verification of the Order

The fact that a given method for numerical approximation of the solution of DEs exhibit an order, say p, is typically demonstrated through standard mathematical procedure. Consider, for example, the Gear's method with $s = 3$, which is given by the following formula

$$x_{n+3} = \frac{18}{11}x_{n+2} - \frac{9}{11}x_{n+1} + \frac{2}{11}x_n + \frac{6}{11}x'_{n+3} \tag{2.27}$$

To demonstrate mathematically that approximations x_{n+3} generated for $x(t)$ at $t = t_{n+3}$ using the above formula indeed satisfy

$$x_{n+3} - x(t_{n+3}) \approx O(h^4) \tag{2.28}$$

(thereby making the method order equal to 3), I first assume that points x_{n+2}, x_{n+1} and x_n are all exact, that is

$$\begin{aligned} x_{n+2} &= x(t_{n+2}) \\ x_{n+1} &= x(t_{n+1}) \\ x_n &= x(t_n) \end{aligned}$$

Next, I expand both $x(t_{n+2})$ and $x(t_{n+1})$ as a Taylor series around $t = t_n$, and substitute in (2.27) to obtain the following series expansion for the approximated value

$$x_{n+3} = x(n) + 3hx^{(1)}(n) + \frac{9}{2}h^2x^{(2)}(n) + \frac{9}{2}h^3x^{(3)}(n) + \frac{309}{88}h^4x^{(4)}(n) + O(h^5) \quad (2.29)$$

To determine the approximation order that x_{n+3} exhibit when compared with $x(t_{n+3})$, I expand the latter in a Taylor series around $t = t_n$

$$x(n+3) = x(n) + 3hx^{(1)}(n) + \frac{9}{2}h^2x^{(2)}(n) + \frac{9}{2}h^3x^{(3)}(n) + \frac{297}{88}h^4x^{(4)}(n) + O(h^5) \quad (2.30)$$

subtracting (2.29) from (2.30) yields

$$x_{n+3} - x(n+3) = \frac{3}{22}h^4x^{(4)}(n) + O(h^5) \quad (2.31)$$

The above result, therefore, demonstrate that the Gear's method with $s = 3$ produces an order 3 approximations to the exact solution.

In addition to the mathematical verification of the order based on formal Taylor series-based method, one can also numerically verify the approximation order. Numerical verification is important when the goal is to ensure that the particular implementation of the method is correct, in the sense that approximations generated from the wave

implementation do indeed tend to the exact solution with power of h that match the order established theoretically. Naturally, this would require that the exact solution be accessible, which is not always possible.

I present in this section a practical way to find the exact solution. For this purpose, I consider the circuit shown in Figure 2.5. The basic idea in generating the accurate or

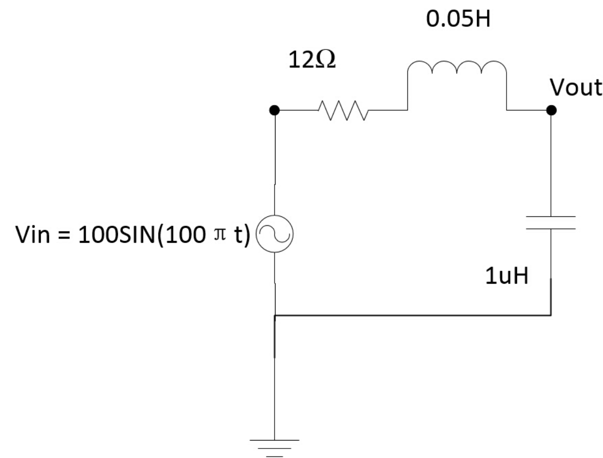


Figure 2.5: A simple RLC circuit

exact transient trajectory for given linear circuit, such as the one shown in Figure 2.5, are based on two main facts.

1. The steady-state response for a linear circuit when stimulated by sinusoidal sources is also sinusoidal, and can be computed by simple AC analysis at the frequency of the sources.

2. If a transient simulation is started with an initial condition constructed from an arbitrary point on the steady-state trajectory, then the transient solution will be identical to the steady-state solution

Clearly, for small step sizes (i.e., as $h \to 0$), the difference between an exact solution and a numerical approximation obtained by DE solver will be dominated by the truncated

term with the lowest power of h, which has been defined earlier as the LTE. Therefore, when plotted on a log-log scale versus the step size $\log h$, this difference will appear as a line the slope of which will be one integer higher than the actual order of convergence of the method.

The above idea are used to numerically verify the order of the Gear's method for $s = 1, 2, 3, 4, 5, 6$. Figures 2.6 through 2.11 plot the actual value of the error versus the step size h used in the formula (2.17). In all those figures, a straight line whose slope matches the theoretical order of the error term (i.e., $s + 1$) is drawn to enable a visual comparison between the actual numerical error behavior (versus the step size h) and its theoretical counterpart.

This technique will be used in Chapter 5 to verify that the proposed formulation for the Obreshkov method yields approximations with order matching the theoretical order of the method.

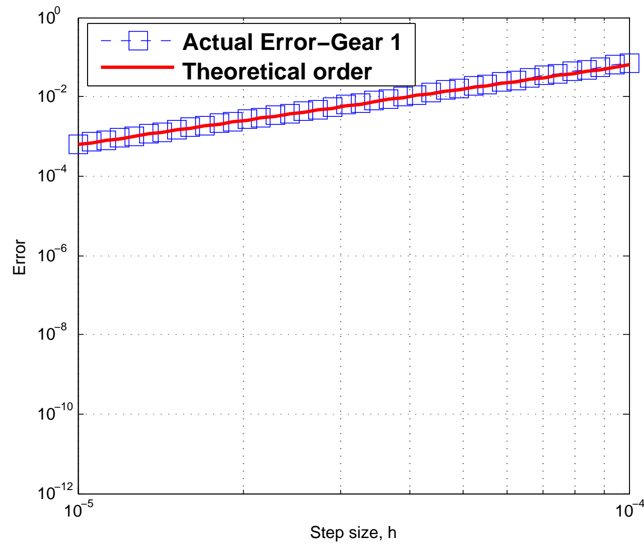


Figure 2.6: Behavior of the actual error (versus the step size h) for approximation generated by the Gear's method with $s = 1$

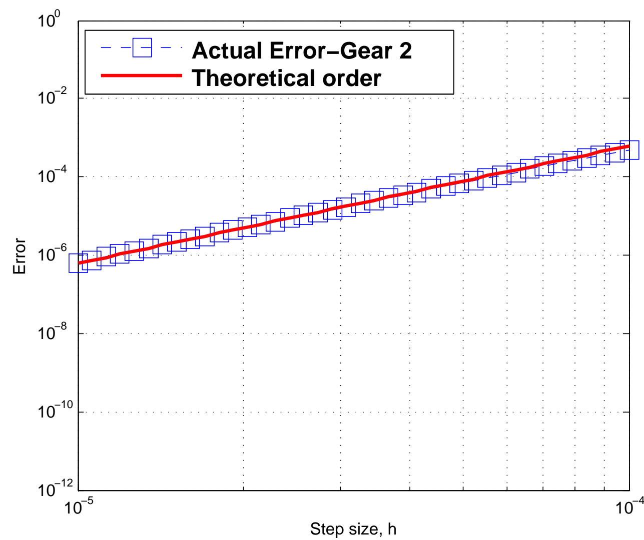


Figure 2.7: Behavior of the actual error (versus the step size h) for approximation generated by the Gear's method with $s = 2$

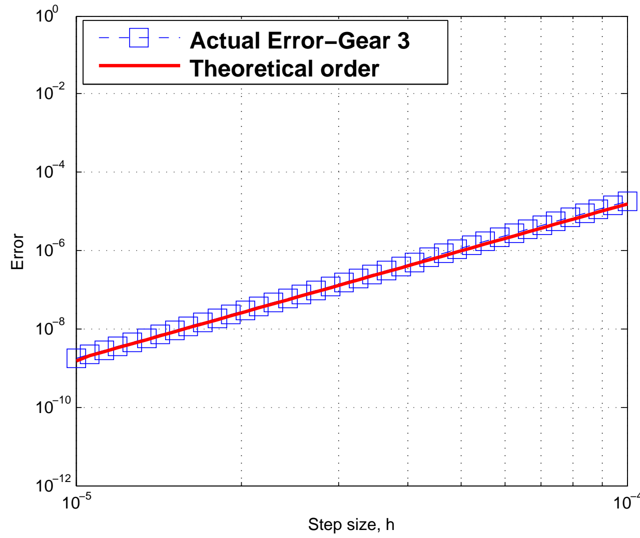


Figure 2.8: Behavior of the actual error (versus the step size h) for approximation generated by the Gear's method with $s = 3$

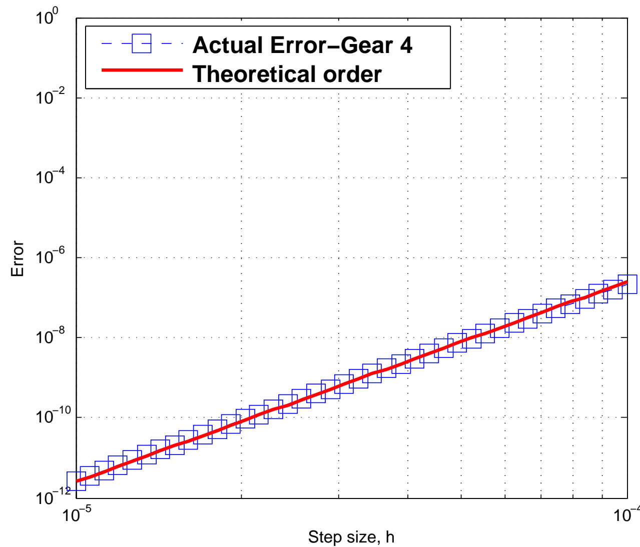


Figure 2.9: Behavior of the actual error (versus the step size h) for approximation generated by the Gear's method with $s = 4$

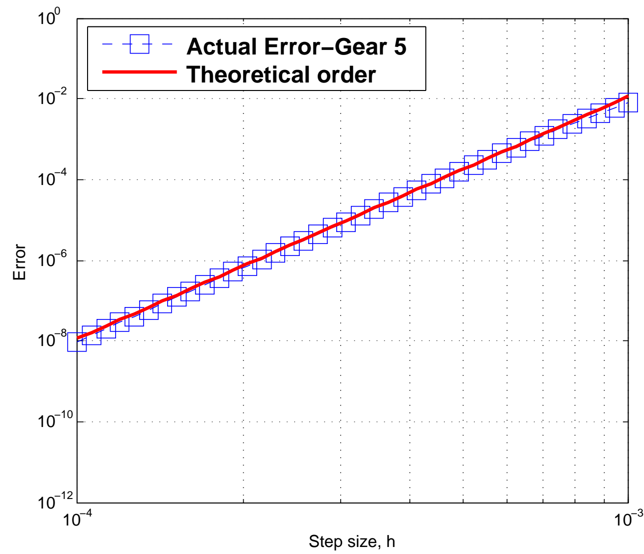


Figure 2.10: Behavior of the actual error (versus the step size h) for approximation generated by the Gear's method with $s = 5$

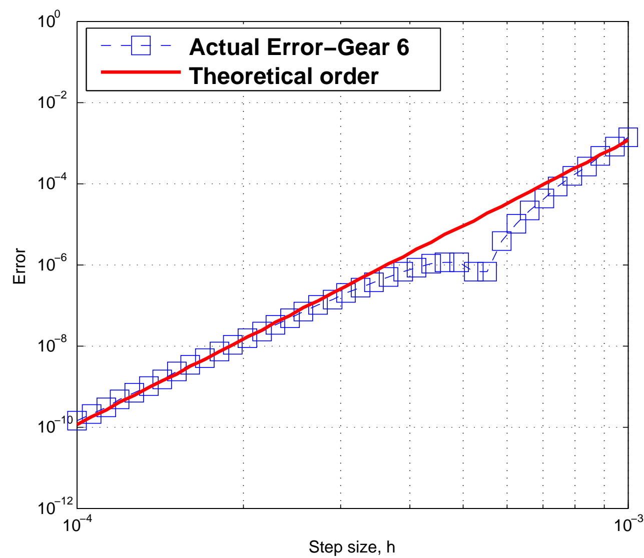


Figure 2.11: Behavior of the actual error (versus the step size h) for approximation generated by the Gear's method with $s = 6$

Chapter 3

Review of the Obreshkov-based Method

The recent work [2] presents a new method, based on the Obreshkov formula [3, 19], that does not suffer from the inherent conflict between the order and the stability. It was shown that this method allows combining the desired property of A-stability (or L-stability) with high-order numerical approximation for the solution of DEs. As a result, this method enabled transient simulation to run much faster than the classical low-order methods used in the SPICE-based simulators such as the Trapezoidal rule or the Gear's method.

The objective in this chapter is to present an overview of the transient simulation based on the Obreshkov formula. The presentation in this chapter will start by first focusing on the basic definition of the Obreshkov formula in Section 3.1, illustrating its high-order and stability properties. Section 3.2 then illustrates, how this formula is adapted for the task of circuit simulation. Section 3.3 shows some ideas that have been used to factorize the Jacobian matrix arising associated with the formulation. Section 3.4 reviews the idea of using rooted trees to automate the computation of high-order

derivatives.

The important items that the reader needs to take from this chapter is the formulation of the problem in Section 3.2, and the utilization of rooted trees in computing high-order derivatives in Section 3.4; for those are the aspects of implementation that the thesis study and propose alternative techniques to improve them.

3.1 Order and Stability Domain of the Obreshkov-based Method

The Obreshkov-based circuit simulator, presented first in [2], is based on approximating the unknown waveforms in the vector $\boldsymbol{x}(t) \in \mathbb{R}^N$ in (2.19) at discrete time points $t_0, t_1, t_2, \cdots$.

This goal is achieved through forcing two successive approximations for each component in $\boldsymbol{x}(t)$[1] at two successive time points, e.g. t_n, t_{n+1}, to satisfy the Obreshkov formula [19,20],

$$\sum_{i=0}^{k}(-1)^i \alpha_{i,k,m} h^i x_{l,n+1}^{(i)} = \sum_{i=0}^{m} \alpha_{i,m,k} h^i x_{l,n}^{(i)} \tag{3.1}$$

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

- $x_{l,n+1}^{(i)}$ is an approximation of the lth component of $\frac{\mathrm{d}^i \boldsymbol{x}(t)}{\mathrm{d}t^i}$ at $t = t_{n+1}$, that is

$$x_{l,n+1}^{(i)} \approx \left.\frac{\mathrm{d}^i x_l(t)}{\mathrm{d}t^i}\right|_{t=t_n}, \; l = 1, \cdots, N \tag{3.2}$$

  with the tacit understanding that $i = 0$ refers to the approximation obtained for the actual value of the waveform, that is, $x_{l,n+1}^{(0)} \approx x_l(t)|_{t=t_{n+1}}$. Similarly, $x_{l,n}^{(i)}$ is an approximation to $\frac{\mathrm{d}^i \boldsymbol{x}(t)}{\mathrm{d}t^i}$ at $t = t_n$.

[1] The individual components in $\boldsymbol{x}(t)$ are denoted by "nonbold, l-subscripted" $x_l(t), l = 1, \cdots, N$, throughout the thesis.

- The triple indexed parameter $\alpha_{i,k,m}$ is defined by the following general expression

$$\alpha_{p,q,r} = \frac{(r+q-p)!q!}{p!(r+q)!(q-p)!} \tag{3.3}$$

The integers k, m used in (3.1) determine the stability and order characteristics of the method. The order of the method is characterized by examining how $\boldsymbol{x}_{n+1}^{(0)}$ approximates the exact $\boldsymbol{x}(t_{n+1})$. It has been proved in [2] that $\boldsymbol{x}_{n+1}^{(0)}$ approximates $\boldsymbol{x}(t_{n+1})$ up to order $k+m$, i.e., $\boldsymbol{x}_{n+1}^{(0)} - \boldsymbol{x}(t_{n+1}) \approx O(h^{k+m+1})$, where $O(h^p)$ denotes terms that are proportional to $h^q, q \geq p$. Moreover, it was proved that numerical stability is not tied to the order $k+m$ as long as $k-2 \leq m \leq k$. This is a very important fact since it enables the technique to take larger time steps without any risk of losing stability, and typically results in significant speedup of the overall simulation [2]. By contrast, Gear's method, which is widely used in classical simulators based on SPICE, exhibits loss of A-stability for orders higher than 2 [21].

3.2 Obreshkov-based Circuit Simulation

In order to describe the implementation of the Obreshkov-based method to simulate the transient time-domain response of circuits, one would have to describe the modeling of the circuits using systems of nonlinear differential equations.

This has already been done in Section 2.4 where it was shown that the MNA approach leads to the system described by (2.19).

It can be shown [4] that using the Obreshkov formula to approximate $\boldsymbol{x}(t)$ can be formulated as the solution to the following nonlinear algebraic equation

$$\tilde{\boldsymbol{C}}\boldsymbol{\xi}_{n+1} + \tilde{\boldsymbol{G}}\boldsymbol{\xi}_{n+1} + \tilde{\boldsymbol{\rho}}_{n+1} = \tilde{\boldsymbol{u}}_{n+1} \tag{3.4}$$

where the above matrices $\tilde{\boldsymbol{C}}$ and $\tilde{\boldsymbol{G}} \in \mathbb{R}^{(k+1)N\times(k+1)N}$ are constructed from $\boldsymbol{C}$ and $\boldsymbol{G}$ as follows :

$$\tilde{\boldsymbol{C}} = \frac{1}{h}\begin{bmatrix} \mathbf{0} & \boldsymbol{C} & \dots & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{C} & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \boldsymbol{C} \\ \mathbf{0} & \mathbf{0} & \dots & \dots & \mathbf{0} \end{bmatrix} \tag{3.5}$$

$$\tilde{\boldsymbol{G}} = \begin{bmatrix} \boldsymbol{G} & \mathbf{0} & \dots & & \mathbf{0} \\ \mathbf{0} & \boldsymbol{G} & \mathbf{0} & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \boldsymbol{G} & \mathbf{0} \\ \alpha_{0,k,m}\boldsymbol{I} & -\alpha_{1,k,m}\boldsymbol{I} & & \dots & (-1)^k\alpha_{k,k,m}\boldsymbol{I} \end{bmatrix} \tag{3.6}$$

where $\boldsymbol{I} \in \mathbb{R}^{N\times N}$ is an identity matrix, and the vector $\boldsymbol{\xi}_{n+1}$ is given by:

$$\boldsymbol{\xi}_{n+1} = \left[\boldsymbol{x}_{n+1}^{(0)\,\top} \quad h\boldsymbol{x}_{n+1}^{(1)\,\top} \quad \cdots \quad h^k\boldsymbol{x}_{n+1}^{(k)\,\top} \right]^\top \tag{3.7}$$

where $\boldsymbol{x}_{n+1}^{(i)} = [x_{1,n+1}^{(i)} \quad x_{2,n+1}^{(i)} \dots x_{N,n+1}^{(i)}]^\top$. The vector $\tilde{\boldsymbol{\rho}}_{n+1}$ is defined by

$$\tilde{\boldsymbol{\rho}}_{n+1} = \left[\boldsymbol{\rho}_{n+1}^{(0)\,\top} \quad \boldsymbol{\rho}_{n+1}^{(1)\,\top} \quad \cdots \quad \boldsymbol{\rho}_{n+1}^{(k-1)\,\top} \quad \mathbf{0}^\top \right]^\top \tag{3.8}$$

where $\boldsymbol{\rho}_{n+1}^{(i)} = h^i \frac{\mathrm{d}^i \boldsymbol{f}(\boldsymbol{x}(t))}{\mathrm{d}t^i}$ computed at the time point t_{n+1} and $\top$ denotes the transpose operator.

The source vector $\tilde{\boldsymbol{u}}_{n+1}$ is defined in terms of the h-scaled derivatives of the independent stimuli to the circuit and the previous ($t = t_n$) state variables in the following

manner

$$\tilde{\boldsymbol{u}}_{n+1} = \begin{bmatrix} \boldsymbol{u}_{n+1}^{(0)\top} & \boldsymbol{u}_{n+1}^{(1)\top} & \cdots & \boldsymbol{u}_{n+1}^{(k-1)\top} & \sum_{j=0}^{m} \alpha_{i,m,k} h^j \boldsymbol{x}_n^{(j)\top} \end{bmatrix}^\top \tag{3.9}$$

where $\boldsymbol{u}_{n+1}^{(i)} = h^i \frac{\mathrm{d}^i \boldsymbol{u}(t)}{\mathrm{d}t^i}$ computed at the time point t_{n+1}.

The nonlinear algebraic equations of (3.4) can be solved through an iterative Newton-based technique. The Jacobian matrix can be shown as follows:

$$\tilde{\boldsymbol{J}} = \begin{bmatrix} \boldsymbol{G} + \boldsymbol{J}_{0,0} & \frac{1}{h}\boldsymbol{C} + \boldsymbol{J}_{0,1} & \boldsymbol{J}_{0,2} & \cdots & \boldsymbol{J}_{0,k} \\ \boldsymbol{J}_{1,0} & \boldsymbol{G} + \boldsymbol{J}_{1,1} & \frac{1}{h}\boldsymbol{C} + \boldsymbol{J}_{1,2} & \cdots & \boldsymbol{J}_{1,k} \\ \vdots & & \ddots & & \vdots \\ \boldsymbol{J}_{k-1,0} & \boldsymbol{J}_{k-1,1} & \cdots & & \frac{1}{h}\boldsymbol{C} + \boldsymbol{J}_{k-1,k} \\ \alpha_{0,k,m}\boldsymbol{I} & -\alpha_{1,k,m}\boldsymbol{I} & \cdots & & (-1)^k \alpha_{k,k,m}\boldsymbol{I} \end{bmatrix} \tag{3.10}$$

where $\boldsymbol{J}_{i,j} = \frac{\partial \boldsymbol{f}_{n+1}^{(i)}}{\partial \boldsymbol{x}_{n+1}^{(j)}} h^{i-j}$. Clearly for algebraic nonlinearities

$$\boldsymbol{J}_{i,j} = 0 \quad \text{for} \quad i < j \tag{3.11}$$

According to the above condition, the Jacobian matrix becomes a lower block Hessenberg matrix.

It is important to compare the structure and size of the Jacobian matrix arising from the Obreshkov-based transient simulation with the Jacobian matrix that arises from the low-order methods used in the SPICE-based simulations, which were reviewed in the previous chapter. The Jacobian matrix of the latter methods is shown in (2.23), and when compared with the one given in (3.10), illustrates that the Jacobian matrix in the Obreshkov-based method is $(k+1)$ times larger than those methods. This fact shows that the cost per-time-step in the Obreshkov-based method is higher than the cost in the low-order method.

However, it is typically the savings in the number of time steps that results from increasing the step size due to the high order, that offsets this cost increase at each time step and provides an overall speedup over the low-order methods.

3.3 Efficient Factorization of the Jacobian Matrix

As mentioned above, the Obreshkov method cause an increase in the cost at each time step, (as compared to the low-order method), is the factorization of a matrix that is $(k+1)$ times larger than the Jacobian matrix of the low-order method presented in the previous chapter. Several ideas have been presented in [4] to mitigate the increasing cost of factorizing the Jacobian matrix arising from the Obreshkov-based method. Those ideas are briefly described in the following text.

A. Structural Characterization of the Jacobian Matrix

The first idea that can be used to reduce the computation cost of factorizing the Jacobian matrix can be gleaned from its structure. To reveal the repetitive structural pattern of this matrix, I expand the Jacobian matrix $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}$, of the nonlinear part in a Taylor series expansion at $t = t_{n+1}$

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} = \sum_{u=0}^{\infty} \frac{1}{u!} \boldsymbol{R}_{n+1}^{(u)} \tau^u \tag{3.12}$$

where $\tau = (t - t_{n+1})/h$ and $\boldsymbol{R}_{n+1}$ is the matrix-valued Taylor series coefficients of the expansion. It is obvious the $\boldsymbol{J}_{0,0}$ is the same matrix $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}$. The authors in [4] further demonstrate that

$$\boldsymbol{J}_{i,j} = h^{i-j} \frac{i!}{j!} \frac{1}{(i-j)!} \boldsymbol{R}_{n+1}^{(i-j)} \tag{3.13}$$

This fact makes it obvious that the 2D array of matrices in (3.10), i.e., $\boldsymbol{J}_{i,j}$ can be represented by a single dimension array of matrices $\boldsymbol{R}_{n+1}^{(u)}$, for $i = 0, 1, \ldots, k-1$, which are obtained from the Taylor series coefficients of $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}$ matrix[22]. (3.13) can be used to recast the Jacobian matrix in the following form

$$\tilde{\boldsymbol{J}} = \begin{bmatrix} \boldsymbol{G} + \boldsymbol{R}_{n+1}^{(0)} & \frac{\boldsymbol{C}}{h} & & & \\ h\boldsymbol{R}_{n+1}^{(1)} & \boldsymbol{G} + \boldsymbol{R}_{n+1}^{(0)} & \frac{\boldsymbol{C}}{h} & & \\ \vdots & & \ddots & & \\ & & \ldots & \boldsymbol{G} + \boldsymbol{R}_{n+1}^{(0)} & \frac{\boldsymbol{C}}{h} \\ \alpha_{0,k,m}\boldsymbol{I} & -\alpha_{1,k,m}\boldsymbol{I} & \ldots & & (-1)^k \alpha_{k,k,m}\boldsymbol{I} \end{bmatrix} \tag{3.14}$$

B. Reducing the Jacobian Matrix Size

The second idea presented in [4] is used to reduce the Jacobian matrix size from $(k+1)N \times (k+1)N$ to $kN \times kN$ which has little effect on the nonzero pattern of the original matrix.

$\boldsymbol{x}_{k,n+1}$ can be written according to the lower-order terms with the Obreshkov formula as follows :

$$h^k \boldsymbol{x}_{k,n+1} = \frac{1}{(-1)^k \alpha_{k,k,m}} \sum_{i=0}^{m} \alpha_{i,m,k} (h^i \boldsymbol{x}_n^{(i)}) - \frac{1}{\alpha_{k,k,m}} \sum_{i=0}^{k-1} \alpha_{i,k,m} (h^i \boldsymbol{x}_{n+1}^{(i)}) \tag{3.15}$$

Substituting from (3.15) into (3.4) leads to a reduced form of Jacobian matrix:

$$\tilde{\boldsymbol{J}} = \underbrace{\begin{bmatrix} \boldsymbol{R}_{n+1}^{(0)} + \boldsymbol{G} & \frac{\boldsymbol{C}}{h} & & \\ h\boldsymbol{R}_{n+1}^{(1)} & \boldsymbol{R}_{n+1}^{(0)} + \boldsymbol{G} & \frac{\boldsymbol{C}}{h} & \\ 2h^2 \boldsymbol{R}_{n+1}^{(2)} & 2h\boldsymbol{R}_{n+1}^{(1)} & \boldsymbol{R}_{n+1}^{(0)} + \boldsymbol{G} \;\; \frac{\boldsymbol{C}}{h} & \\ \vdots & \ddots & \ddots & \\ h^{k-1}(k-1)!\boldsymbol{R}_{n+1}^{k-1} + \beta_0 \frac{\tilde{\boldsymbol{C}}}{h} & h^{k-2}(k-1)!\boldsymbol{R}_{n+1}^{k-2} + \beta_1 \frac{\tilde{\boldsymbol{C}}}{h} & \cdots & \boldsymbol{G} + \boldsymbol{R}_{n+1}^{0} + \beta_{k-1} \frac{\tilde{\boldsymbol{C}}}{h} \end{bmatrix}}_{k \times k \quad \text{blocks}}. \tag{3.16}$$

where $\beta_i = (-1)^{k+i+1} \frac{\alpha_{i,k,m}}{\alpha_{k,k,m}}$.

C. Block Factorization for High-Order Formulation

The third idea presented in [4] is to produce a more efficient factorization technique. The proposed idea takes advantages of the fact that each block in (3.16) is sparse and the blocks form a Hessenberg matrix [23, 24].

This idea can be explained in a two-stage process. The first stage is to perform a symbolic analysis on an artificial $N \times N$ matrix. The sparsity pattern of this matrix is identical to the sparsity pattern of the Jacobian matrix which arises in the course of using a low-order method such as the TR method. This analysis is executed only once at the beginning of the simulation. In addition, the cost of this analysis is identical to the preordering techniques used for the low-order method.

The second stage is mainly the numerical operation which is performed on the Jacobian matrix, where the entries are the $k \times k$ blocks, obtained by the permutation described above. If $\hat{\boldsymbol{J}}$ is the $N \times N$ block matrix, where the entries are $k \times k$ blocks, then

$$\hat{\boldsymbol{J}} = \boldsymbol{P}\tilde{\boldsymbol{J}}\boldsymbol{P}^{\top} \tag{3.17}$$

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

$$\hat{\boldsymbol{J}} = [\hat{\boldsymbol{J}}_{a,b}] \tag{3.18}$$

where $a, b = 1, \ldots, N$ and $\hat{\boldsymbol{J}}_{a,b}$ is a $k \times k$ block, which is from $(a-1)k+1$ to ak rows and $(b-1)k$ to bk columns.

Algorithm 1 presents a pseudocode illustration of the block version of the LU factorization in [25].

Algorithm 1: BLU factorization

$\boldsymbol{L} = \boldsymbol{I}$;

for $p = 1 \quad to \quad N$ **do**

solve the block lower triangular system $\boldsymbol{Ly} = \hat{\boldsymbol{J}}(:, p)$;

$\boldsymbol{U}(1 : p-1, p) = \boldsymbol{y}(1 : p-1, 1)$;

do partial pivoting on block vector $\boldsymbol{y}$;

LU factorize $\boldsymbol{L}(p,p)\boldsymbol{U}(p,p) = \boldsymbol{y}(p, 1)$;

solve $\boldsymbol{L}(p+1 : N, p)\boldsymbol{U}(p,p) = \boldsymbol{y}(p+1 : N, 1)$ for

$\boldsymbol{L}(p+1 : N, p)$

According to the block nature of the above algorithm, one needs to find the block with the largest determinant as the pivot[26]. Instead of computing the determinants explicitly and expensively, the proposed algorithm multiplies the diagonal elements to obtain an approximate estimate of the block determinant. The following fact from the matrix theory [27] is used in this notion.

$$0 < \frac{\det(\boldsymbol{a_D}) - \det(\boldsymbol{a})}{\det(\boldsymbol{a_D})} \leq 1 \tag{3.19}$$

where $\boldsymbol{a_D}$ is a diagonal matrix of which entries are the diagonal elements of the matrix $\boldsymbol{a}$. According to the fact that the pivoting step requires only comparing the determinants in each block, an approximate estimate is sufficient to serve the purpose of this step.

After the permutation in (3.17), the blocks in the final Jacobian matrix can be one, or a combination of more than one, of the following types of blocks.

- A diagonal block, of the form $g\boldsymbol{I}$. If node a or node b or both are connected to memoryless linear elements, this type of blocks will arise.

- A block of the form $\frac{c}{h}\boldsymbol{B}$,where $\boldsymbol{B}$ is the matrix of the companion form, that is

$$\begin{bmatrix} 0 & 1 & & \\ \vdots & \ddots & \ddots & \\ 0 & \dots & 0 & 1 \\ \beta_1 & \beta_2 & \dots & \beta_{k-1} \end{bmatrix}$$

  If node i or node j or both are connected to linear memory elements, this block will arise.

- A lower triangular block of the form

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

  If node i or node j or both are connected to nonlinear memoryless elements, this block will arise.

- Naturally, if node i or node j or both are connected to more than one type of the above mentioned elements, then the block can be a lower Hessenberg matrix of the form

$$\begin{bmatrix} g+r_0 & \frac{c}{h} & & \\ hr_1 & g+r_0 & \frac{c}{h} & \\ \vdots & & \ddots & \ddots \\ & & \dots & g+r_0-\beta_{k-1}\frac{c}{h} \end{bmatrix}$$

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

3.4 Evaluating nonlinear vector and derivatives

As has been mentioned earlier, the system in (3.4) is a nonlinear algebraic system that can be solved using the iterative Newton method. The Newton method first starts by assigning an initial guess to the vector of unknowns, which in this case is given by $\boldsymbol{\xi}_{n+1}$, i.e., the kN values that capture 0 to $(k-1)$ order derivatives of the N circuit variables. The Newton method then requires computing the vector $\tilde{\boldsymbol{\rho}}_{n+1}$ and the Jacobian matrix $\tilde{\boldsymbol{J}}$.

For the sake of brevity, I focus on the computation of $\tilde{\boldsymbol{\rho}}_{n+1}$.

Here the vector $\tilde{\boldsymbol{\rho}}_{n+1}$ represents the 0 to $(k-1)$ order derivatives of the nonlinear function vector $\boldsymbol{f}(\boldsymbol{x}(t))$, scaled by powers of h^0 to h^{k-1} for all of the N circuit variables.

To illustrate the process of this computation, I assume, without loss of generality, that the qth entry in $\boldsymbol{f}(\boldsymbol{x}(t))$ corresponds to the current diode expression given by

$$f_q(\boldsymbol{x}(t)) = I_0(\exp(\frac{x_q(t) - x_p(t)}{V_T}) - 1) \tag{3.20}$$

For example, this would be the case if there is only one diode connected between nodes p and q in the circuit.

The diode expression can be represented by a rooted tree graph shown in Figure 3.1. A rooted tree is a directed graph whose nodes are one of the following types

1. Leaf Nodes: Those nodes represent the independent variables in the expression, such as $x_p(t)$ and $x_q(t)$. They are also used to represent the constant terms such as I_0 or V_T.

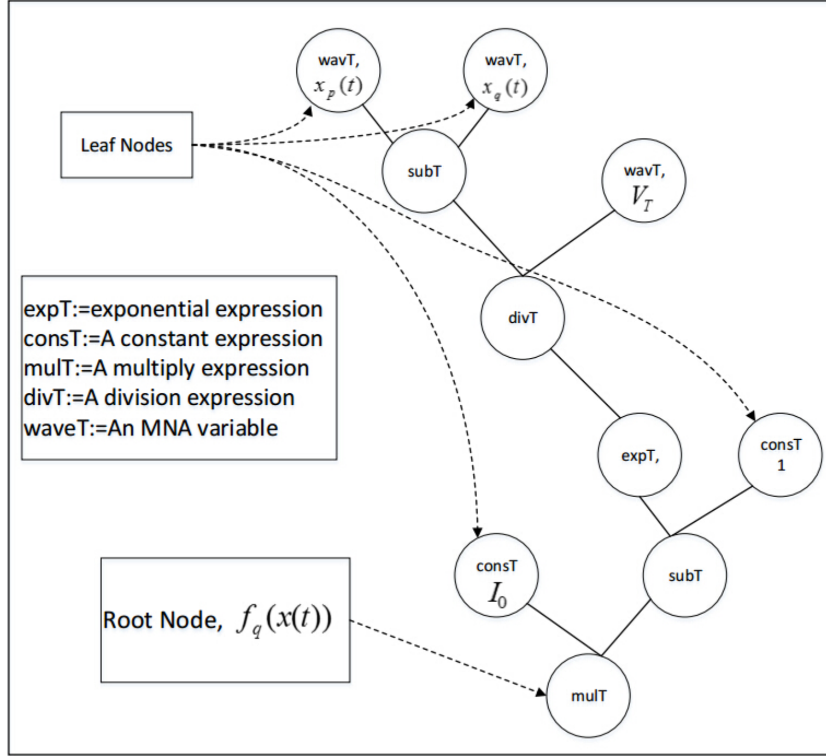


Figure 3.1: Rooted tree representing the diode current

2. Regular Nodes: Those nodes represent the intermediate functions such as the exponent function or the multiplication operation.

3. A Root Node: which represents the entire expression, $f_q(\boldsymbol{x}(t))$.

Edges between node i and node j exist if the expression represented by node i is a direct function of the expression represented by node j. In this situation, node i is called a child of node j and node j is called the parent node of i. Thus, leaf nodes are childless, whereas the root node is parentless.

The objective of using rooted trees to represent nonlinear expressions is to facilitate the computation of the derivatives of $\boldsymbol{f}(\boldsymbol{x}(t))$ with respect to t.

Expanding $f_q(\boldsymbol{x}(t))$ as a Taylor series in t around t_{n+1}, I get

$$\begin{aligned} f_{q,n+1} &= \sum_{i=0} \frac{1}{i!} f_{q,n+1}^{(i)} (t - t_{n+1})^i \\ &= \sum_{i=0} \frac{1}{i!} h^i f_{q,n+1}^{(i)} \left(\frac{t - t_{n+1}}{h} \right)^i \end{aligned} \tag{3.21}$$

where $f_{q,n+1}^{(i)} = \frac{\mathrm{d}^i f}{\mathrm{d}t^i}|_{t=t_{n+1}}$.

Thus, the components in $\tilde{\boldsymbol{\rho}}_{n+1}$ are the Taylor series coefficients of the individual components of $\boldsymbol{f}(\boldsymbol{x}(t))$ calculated at $t = t_{n+1}$ and scaled by powers of h. The computation of those coefficients using the notion of rooted trees is based on the following ideas.

- Computations of those coefficients is trivial for expressions represented by the leaf nodes. Indeed this is the case since leaf nodes either represent constant terms, (in which the derivatives with respect to time t are given by 0 for $i > 0$, or the constant itself for $i = 0$), or they represent the independent variables such as $x_q(t)$ or $x_p(t)$. The ith order derivatives for those terms (i.e., the independent variables) are simply the components of $\boldsymbol{\xi}_{n+1}$,which are supplied as an initial guess at the start of the iterative technique.

- Computation of those coefficients for expressions represented by second level nodes can be done based on the derivatives of the children expression (in this case leaf nodes) and specific formula tailored to the expression of the parent node. As an example, assume that the second level node represents a simple exponential function such as $\exp(x(t))$, and expand both $x(t)$ and $\exp(x(t))$ in their Taylor series around $t = t_{n+1}$

$$x(t) = \sum_{i=0} u_i \tau^i \tag{3.22}$$

$$\exp(x(t)) = \sum_{i=0} v_i \tau^i \tag{3.23}$$

with

$$u_i = \frac{1}{i!} h^i \frac{\mathrm{d}^i x(t)}{\mathrm{d}t^i} \tag{3.24}$$

$$v_i = \frac{1}{i!} h^i \frac{\mathrm{d}^i \exp(x(t))}{\mathrm{d}t^i} \tag{3.25}$$

$$\tau = \frac{t - t_{n+1}}{h} \tag{3.26}$$

It can be shown that the ith derivatives of the parent (exponential function) is given as follows [4]

$$v_i = \frac{1}{i} \sum_{m=0}^{i-1} (i - m) \underbrace{v_m}_{\text{parent derivative}} \times \underbrace{u_{i-m}}_{\text{child derivative}} . \tag{3.27}$$

Thus v_i, which represents the h^i-scaled derivative of the parent node, can be obtained from the h^i-scaled derivative of the children node and the h^p-scaled derivatives of the parent node for $p < i$. Similar expressions for the derivatives of parent nodes representing different types of functions can also be derived as shown in Table 3.1 [28].

- The process of computing the h-scaled derivatives can be propagated from the leaf nodes and down to the root node. The derivatives computed at the root node represent the values of $h^i f^{(i)}_{q,n+1}$ that is required to compute the entries in the vector $\tilde{\boldsymbol{\rho}}_{n+1}$.

The computation of the coefficients $h^i f^{(i)}_{q,n+1}$ proceeds by first prompting the root node to compute its own scaled derivative for $i = 0$. The root node then sends a request to its immediate children asking them to compute their own $i = 0$ derivative. The request is propagated all the way up to the leaf nodes. When the leaf nodes receive the request to compute their $i = 0$ derivative, they respond by releasing the derivatives collected from

Equation	$y = \sum_{i=0} c_i t^i, f = \sum_{i=0} d_i t^i, z = \sum_{i=0} e_i t^i$
$f = \exp(y)$	$d_0 = \exp(c_0)$ $d_n = \frac{1}{n} \sum_{i=0}^{n-1} d_i c_{n-i}(n-i)$
$f = \log(y)$	$d_0 = \log(c_0)$ $d_n = \frac{1}{c_0} \left(c_m - \sum_{i=1}^{n-1} d_{n-i} c_i((n-i)/n)\right)$
$f = y^p$	$d_n = \left(p d_0 c_n n + \sum_{i=1}^{n-1} c_i d_{n-i}(i(p+1)-n)\right)/n c_0$
$f = y + z$	$d_n = c_n + e_n$
$f = y - z$	$d_n = c_n - e_n$
$f = yz$	$d_n = \sum_{i=0}^{n} c_i e_{n-i}$
$f = y/z$	$d_n = (c_n - \sum_{i=0}^{n-1} d_i e_{n-i})/e_0$

Table 3.1: Some Formulas for the Derivatives of Simple Functions

the initial guess, if the leaf node represents an independent variable, or simply returning the value of the constant if they represent a constant term in the nonlinear expression. The parents of the leaf nodes, upon receiving the $i = 0$ derivative of the children (leaf node), also proceed to compute their own $i = 0$ derivative, using specific formula tailored to the type of each node. The process is then propagated all the way down to the root node, which uses the information just received to compute $h^0 f_{q,n+1}^{(0)}$. The above process is then repeated all the way for $i = 1, 2, \ldots, k-1$.

3.5 Discussion

This chapter presented a brief review of the transient circuit simulation based on the Obreshkov formula. The presentation of the Obreshkov method was meant to highlight those aspects that are targeted for improvement by this thesis.

The first aspect targeted for improvement is the technique used to evaluate the nonlinear function vector $\tilde{\boldsymbol{\rho}}_{n+1}$, which is also the same technique used to compute the entries in the Jacobian matrix $\tilde{\boldsymbol{J}}$.In each one of the Newton iteration , and for each order of the coefficients, the tree is traversed from the root node up to the leaf nodes, and then back down again to the root node.

The following chapter proposes an alternative approach to handle this aspect of the computation in a more efficient way that effectively reduces the computational time. The other aspect that the thesis addresses is motivated by the observation that the formulation leading up to the system of nonlinear algebraic equation in (3.4) is restricted to the special case where the memory matrix $\boldsymbol{C}(\boldsymbol{x}(t))$ is a constant matrix independent from $\boldsymbol{x}(t)$.

That limitation meant that the formulation can handle only circuits with linear memory elements, i.e., linear capacitors and inductors. It also implied that nonlinear memory elements can still be handled provided that their nonlinearity is described as a charge- or flux-based nonlinear function, since this description would still be represented by a constant $\boldsymbol{C}$ matrix using the charge-oriented formulation MNA [29]. Nevertheless, it leaves out those capacitors or inductors that the user or the vendor chooses to model using a capacitance- or inductance-based nonlinear functions, as those functions will be incorporated as entries in the $\boldsymbol{C}$ matrix. An example of this situation is encountered in modelling the diffusion capacitance of the MOS transistor [30]

The purpose of Chapter 5 is to bridge this gap by generalizing the formulation of the Obreshkov-based method so that it can handle the formulation with $\boldsymbol{x}(t)$-dependent $\boldsymbol{C}$ matrix.

Chapter 4

Improved Processing of Rooted Tree

In the previous chapter, the basic operation of traversing the rooted tree representing the nonlinear function is $\boldsymbol{f}(\boldsymbol{x}(t))$ or the Jacobian matrix $\tilde{\boldsymbol{J}}(\boldsymbol{x}(t))$ was briefly outlined. The main objective in this process is computing the high-order derivatives of those functions with respect to t. The orders of the derivatives that need to be computed in this context is for $i = 0, 1, \ldots, k-1$

4.1 Illustration of the Operation

In the basic operation scheme of the process, the tree is represented using Object-Oriented paradigm, where each node in the tree is represented by an object of a class that is derived from a generic abstract class called *iExpr*. For example, a node representing the multiplication operator, in a nonlinear function, is represented by an object of the class *mulTerm*, which is derived from *iExpr*. Each class derived from *iExpr* has member pointers that point to the children nodes objects. For example, the *mulTerm* class, has two children nodes that represent the two operands of the multiplication operation, and therefore requires two pointers (both of the same type *iExpr*) pointing to those nodes. By contrast, a class representing the exponential function has only one child node, since

the exponential function is a function of a single argument, e.g.,$\exp(x)$. That type of class is also derived from *iExpr*. However, it requires only one member pointer to point to the single child node. Leaf nodes, however, are childless, thereby requiring no such pointers.

The process of computing the high-order derivative is initiated from the root node by calling a member function, referred to as "*moment*". This member function is shared by all the classes of the tree. Using the Object-Oriented paradigm, "*moment*" is a method that is defined as pure virtual function in the base class *iExpr*. This is a feature of the Object-Oriented C++ language, that forces each class derived from *iExpr* to provide its own definition of the method "*moment*". The input argument of moment is an *integer* value which represents the value of the order of the derivative that needs to be computed. The return value of this function is the numerical value of the derivative, which is represented by the type *double* of the C-language.

The process then starts by calling the moment function of the root node, passing it the value of 0, to compute the 0th order derivative which represents the value of the function without any differentiations. The root node, then communicates this order to the children nodes by calling their *moment* function. For example, assume that the root node is of the *mulTerm* type with two children nodes, pointed two by two pointers, *arg1* and *arg2*, then the first thing that *mulTerm::moment* will do is to call *arg1* $\rightarrow$ *moment(0)* and *arg2* $\rightarrow$ *moment(0)* so that the 0th derivatives of the children nodes are computed first.

In a similar manner, each node in the tree sends a request to its children to compute their own 0th derivatives. Eventually, this request reaches the top or leaf nodes in the tree. The classes of the leaf nodes, having no children, respond by directly returning their 0th order derivative. For leaf nodes, the task of computing their derivative is quite trivial. For example, for node representing a constant (i.e. time-independent parameter)

in the nonlinear expression, the 0th order derivative is the value assigned to the constant parameter, whereas its high order derivatives are equal to zero identically.

The process is then reversed, with leaf nodes returning their 0th derivatives to their parents, and the parents using those just returned values to compute their own 0th order derivatives until the root node is reached. The above process is then repeated for the 1st order derivative, and then for $i = 2$ to compute the 2nd order derivative all the way to the $(k-1)^{\text{th}}$ order derivative. We show an example of the way this is done by providing the description of the method moment of the *mulTerm* class and *expTerm* class

```
1  double mulTerm::moment(int m)
2  {
3     double result = 0;
4     da1[m] = arg1->moment(m); // da1 and da2 are arrays to store the
5     da2[m] = arg2->moment(m); // calculated moments of the children nodes.
6     int i;
7     for (i = 0; i < = m; i++)  //The loop to calculate the moment for order m
8          result += da1[i]*da2[m-i];
9          //Calculate the moments according to the 7th row in Table 3.1
10    return result;
11 }
```

```
1  double expTerm::moment(int m)
2  {
3     double result = 0;
4     int i;
5     da[m] = arg->moment(m);
6     if (m==0){               // Zero order moment
7          result = exp(da[0]);
8     }
9     else{                    //The loop to calculate the moment for order m
10        for (i=0;i<m;i++)
11          result += dr[i]*da[m-i]*(m-i);
12       result /= m;
13       //Calculate the moments according to the second row in Table 3.1
14       }
15    return dr[m] = result;
16 }
```

4.2 Observations on the basic operation

In this section, I will register some observation on the process of traversing the rooted tree described by the previous section. The main observation stated here is that the process of traversing the tree in both directions, involves some fundamental components and some overhead operations. The fundamental component refers to the arithmetic operations. An example for that type of operation is the multiplication operation shown on line 8 of the *mulTerm* code. On the other hand, overhead operations are those non-arithmetic operations. Such operations are not fundamental part of the computational algorithm, but are nonetheless necessary for the arithmetic operation. Example of the overhead operations are plenty and include the calling of the moment function of the children nodes shown in line 4 and 5, or the setting up of the loop shown in line 7 of the *mulTerm* code. The second observation is that this overhead needs to be repeated for each order derivatives, from $i = 0, 1, \ldots, k - 1$.

The above two observations then instigated the question as to whether this overhead operation can be optimized. More specifically, the question was whether the overhead needs to be repeated for $i = 0, 1, \ldots, k-1$, or can be simply done once for all derivatives. Following the underlying reasoning behind the existing implementation would requite presenting the background in which the idea of rooted tree was first developed. This is done in the following subsection.

4.2.1 Background to Rooted Trees

The idea of rooted tree was first presented in [31]. The work in this paper was concerned with the following system of equations

$$\boldsymbol{G}\boldsymbol{x} + \boldsymbol{f}(\boldsymbol{x}(\alpha)) = \alpha \boldsymbol{b} \tag{4.1}$$

The objective in this work was the computation of the high order derivatives of $\boldsymbol{x}(\alpha)$ with respect to the parameter α.

The approach followed to achieve this objective is based on expanding $\boldsymbol{x}(\alpha)$, and $\boldsymbol{J}(\boldsymbol{x}(\alpha)) = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}$ in Taylor series in α as follows

$$\boldsymbol{x}(\alpha) = \sum_i \boldsymbol{a}_i \alpha^i \tag{4.2}$$

$$\boldsymbol{J}(\boldsymbol{x}(\alpha)) = \sum_i \boldsymbol{J}_i \alpha^i \tag{4.3}$$

It can be shown that, substituting from (4.2) and (4.3) into (4.1) yields the following

$$(\boldsymbol{G} + \boldsymbol{J}_0)\boldsymbol{a}_1 = \boldsymbol{b} \tag{4.4}$$

$$(\boldsymbol{G} + \boldsymbol{J}_0)\boldsymbol{a}_n = -\frac{1}{n}\sum_{j=1}^{n-1}(n-j)\boldsymbol{J}_i\boldsymbol{a}_{n-j} \tag{4.5}$$

The computation in this work proceeds by first finding a_1 from

$$(\boldsymbol{G} + \boldsymbol{J}_0)\boldsymbol{a}_1 = \boldsymbol{b} \tag{4.6}$$

Based on $\boldsymbol{a}_1$ the value of $\boldsymbol{J}_1$ can be computed. Next $\boldsymbol{a}_2$ is computed by

$$(\boldsymbol{G} + \boldsymbol{J}_0)\boldsymbol{a}_2 = -\frac{1}{2}\boldsymbol{J}_1\boldsymbol{a}_1 \tag{4.7}$$

From $\boldsymbol{a}_1$ and $\boldsymbol{a}_2$, then $\boldsymbol{J}_2$ can be computed. Next $\boldsymbol{a}_3$ is computed from

$$(\boldsymbol{G} + \boldsymbol{J}_0)\boldsymbol{a}_3 = -\frac{1}{3}\boldsymbol{J}_1\boldsymbol{a}_2 + \boldsymbol{J}_2\boldsymbol{a}_1 \tag{4.8}$$

The process continues in a similar manner for $\boldsymbol{a}_4, \boldsymbol{a}_5, \ldots$ Noting that $\boldsymbol{a}_i$ and $\boldsymbol{J}_i$ are

the ith order derivative of $\boldsymbol{x}(\alpha)$ and $\boldsymbol{J}(\boldsymbol{x}(\alpha))$ scaled by $\frac{1}{i!}$, i.e.

$$\boldsymbol{a}_i = \frac{1}{i!}\frac{\mathrm{d}^i \boldsymbol{x}}{\mathrm{d}\alpha^i} \tag{4.9}$$

$$\boldsymbol{d}_i = \frac{1}{i!}\frac{\mathrm{d}^i}{\mathrm{d}\alpha^i}\boldsymbol{J}(\boldsymbol{x}) \tag{4.10}$$

then it is possible to see that the notion of rooted trees can be used to compute those high-order derivatives. Indeed, this was the circumstances in which the idea of rooted trees was conceived.

The present situation in which the rooted trees are being used is different in one main aspect: the derivatives need not be computed successively. By contrast, in the situation of using the rooted tree in [31], derivatives had to be computed successively, starting with $\boldsymbol{a}_1$, then $\boldsymbol{a}_2$ and so on. On the other hand, the application of rooted tree for the Obreshkov-based method, starts by having all the derivatives of $\boldsymbol{x}(t)$ at $t = t_{n+1}$, as part of the initial guess. In other words, the derivatives of $\boldsymbol{x}(t)$ at $t = t_{n+1}$ are all available upfront.

4.3 Improved Computation

The above discussion (on the distinction between the original purpose for which the idea of rooted tree was utilized, and the underlying objective in using those trees for the Obreshkov-based transient simulation) suggests that there could be a room for further improvements in the implementation of rooted trees. More specifically, the suggested improvement aims at reducing the overhead by restricting the bottom-up traversing of the rooted tree to only one time traversal to compute all the derivatives $i = 0, 1, \ldots, k-1$.

To this end, this thesis proposes modifying the method *moment*, in all the classes of the tree, so that it receives two input arguments

1. The first argument specifies the maximum order of the desired derivatives, i.e. $k-1$.

2. The second argument is a pointer to an array of type *double*, which will receive the values.

The following codes show the prototype of the proposed moment method as it is implemented in the *mulTerm* and *expTerm*

```
1  void expTerm::new_moment(int order, double *p_output_moments)
2  {
3     int i,m;
4     arg->new_moment(da);
5     double result;
6
7     p_output_moments[0] = exp(da[0]); // Zero order moment
8
9     for ( m = 1;                  // Starting from the first order moment
10          m < order;              // Loop until the order required
11          m++)
12    {
13      result = 0;                 // Initialize an accumulator to zero
14      for (i = 0 ; i < m; i++)    // Internal loop to compute the moments of the
15                {                 // exponential term.
16                 result += p_output_moments[i]*da[m-i]*(m-i);
17                 }
18      result /= m;
19      p_output_moments[m] = result;
20    }
21 }
```

```
1  void mulTerm::new_moment(int order, double *p_output_moments)
2  {
3          arg1->yao_moment(da1); // da1 and da2 are arrays to store the
4          arg2->yao_moment(da2); // calculated moments of the children nodes.
5          int i,m;
6          double result;
7
8          for (m=0 ; m < order ; m++)
9          // The loop to calculate all the moments
10         {
11         result = 0;
12         for (i=0;i<=m;i++)
13                 result += da1[i]*da2[m-i];
14         p_output_moments[m] = result;
15         }
16 }
```

4.4 Validation Example

The idea presented in the precious section is put to test by considering the mixer circuit shown in Figure 4.2 and the netlist of this circuit is attached in Appendix-A. The circuit in this example has 6 JFET transistors which are modeled by the circuit shown in Figure 4.1.

The currents i_{gd}, i_{gs} and i_{ds} represent the main elements of nonlinearity in the circuit. The definitions of those nonlinearities are based on the level at which the device model is being used. We compared 3 levels of device modeling for which the definitions of

Figure 4.1: JFET transient analysis model.

Figure 4.2: Schematic of the double-balanced mixer.

nonlinearities is given as follows.

$$i_{gs} = 10^{-15}\left(\frac{\exp(v_G - v_S)}{0.025} - 1\right)$$
$$i_{gd} = 10^{-15}\left(\frac{\exp(v_G - v_D)}{0.025} - 1\right)$$

$$i_{ds} = \begin{cases} \text{Level 1} \begin{cases} 0, & v_{gst} < 0; \\ \beta eff \times {v_{gst}}^2 (1 + \lambda \times v_{ds}), & 0 < v_{gst} < v_{ds}; \\ \beta eff \times v_{ds}(2v_{gst} - v_{ds})(1 + \lambda \times v_{ds}), & 0 < v_{ds} < v_{gst}. \end{cases} \\ \\ \text{Level 2} \begin{cases} 0, & v_{gst} < 0; \\ \beta eff \times {v_{gst}}^2 \{1 + \lambda(v_{ds} - v_{gst}) \times (1 + LAM1 \times v_{gs})\}, & 0 < v_{gst} < v_{ds}, v_{gs} \geq 0; \\ \beta eff \times {v_{gst}}^2 \{1 - \lambda(v_{ds} - v_{gst}) \times (1 + \lambda \times \frac{v_{gst}}{VTO})\}, & 0 < v_{gst} < v_{ds}, v_{gs} \leq 0; \\ \beta eff \times v_{ds}(2v_{gst} - v_{ds}) \times (1 + \lambda \times v_{ds}), & 0 < v_{ds} < v_{gst}. \end{cases} \\ \\ \text{Level 3} \begin{cases} 0, & v_{gst} < 0; \\ \beta eff \times {v_{gst}}^{V_{GEXP}} (1 + \lambda \times v_{ds}) \times \tanh(\alpha \times v_{ds})), & v_{gst} \geq 0. \end{cases} \end{cases}$$

where

$$\begin{aligned} v_{gs} &= v_G - v_S \\ v_{ds} &= v_D - v_S \\ v_{gd} &= v_G - v_D \end{aligned}$$

and all the parameters are explained in Table 4.1.

The goal in this section is to compare the performance of the proposed method of traversing the rooted tree against the original method used previously.

Parameter	Description
VTO	Threshold voltage
λ	Channel length modulation parameter
LAM1	Channel length modulation gate voltage parameter
β	Transconductance parameter
β eff	$\beta\frac{Weff \times M}{Leff}$
Weff	Default FET width
Leff	Default FET length
M	Grading coefficient for gate-drain and gate-source diodes
α	Saturation factor
V_{GEXP}	Gate voltage exponent

Table 4.1: Parameters in the above definition expression

To this end, the circuit was excited by the following sources

$$\begin{aligned} V_{lo1} &= 0.125\text{sin}(2 \times 10^9 \pi t) \\ V_{lo2} &= 0.125\text{sin}(2 \times 10^9 \pi t + \pi) \\ V_{rf1} &= 0.0125\text{sin}(2 \times 9 \times 10^8 \pi t) \\ V_{rf2} &= 0.0125\text{sin}(2 \times 9 \times 10^8 \pi t + \pi) \end{aligned} \tag{4.11}$$

and the transient time-domain response was simulated for a period of time of $10n$ seconds. The transient time domain response was simulated using the Obreshkov method which was implemented using two different approaches: the first approach is based on the legacy processing of the rooted tree, whereas the second approach is based on the suggested processing based on a single traversal of the tree. In both approaches, the time spent on computing the nonlinear vector $\tilde{\boldsymbol{\rho}}_{n+1}$ and the Jacobian matrix $\tilde{\boldsymbol{J}}$ was measured using

the valgrind tool [32], which counts the number of the system clock cycles elapsed during this computation.

Figure 4.3 and Figure 4.4 show the number of system clock cycles required by both approaches indicating a reduction by about 20 to 25% using the proposed approach.

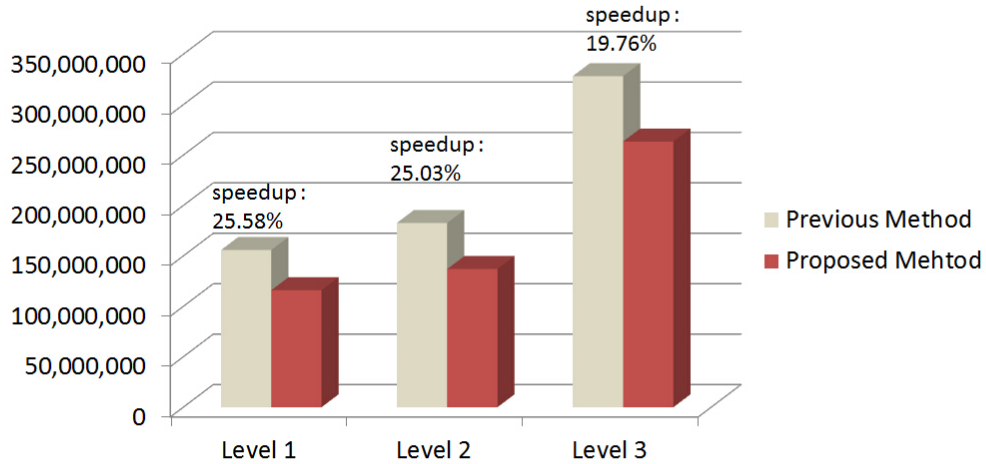


Figure 4.3: CPU Cycles of Evaluating the Nonlinear Vector

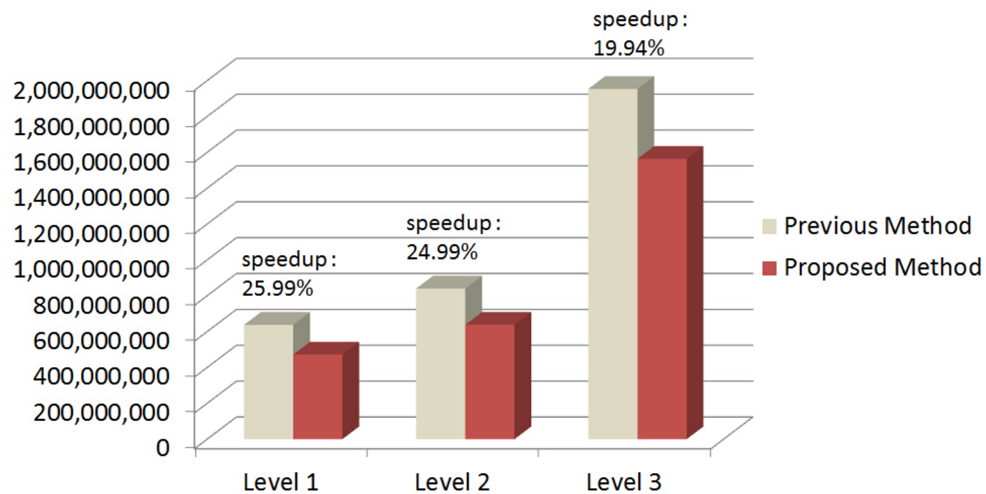


Figure 4.4: CPU Cycles of Evaluating the Jacobian Matrix

To show that this saving in the execution time occurs without loss of accuracy, I com-

pare the numerical values obtained by both approaches for a sample entry in the vector $\tilde{\boldsymbol{\rho}}_{n+1}$ (corresponding to the node $nd1$) at $t = 10ns$ for a time step of $h = 0.05ns$. This comparison, shown in Table 4.2 indicates that the proposed method produces numerical values exactly identical to the legacy method. Additionally, Figure 4.5 shows the steady state output voltage from 0 to $10ns$ of the node $Vout1$ using both the proposed method and the legacy method for a time step of $h = 0.05ns$ and they match exactly.

Order	Previous method	Proposed method
order 0	0.00401170239548004	0.00401170239548004
order 1	-0.000984052742629167	-0.000984052742629167
order 2	1.53496829106300e-05	1.53496829106300e-05
order 3	2.83910719324942e-05	2.83910719324942e-05
order 4	3.88795468503857e-05	3.88795468503857e-05

Table 4.2: Comparison of the value in nonlinear vector $\tilde{\boldsymbol{\rho}}$

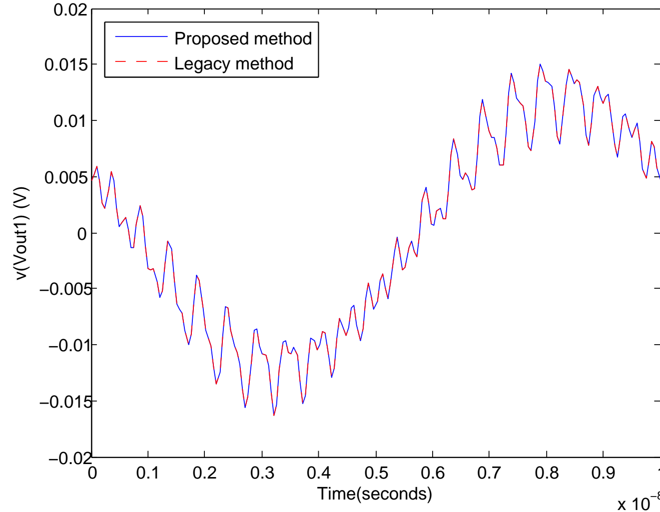


Figure 4.5: Comparison of the output of node Vout1

Chapter 5

Formulation of the Obreshkov-based Transient Circuit Simulation in the Presence of Nonlinear Memory Elements

The objective of this section is to formulate the nonlinear algebraic problem that results from using the Obreshkov formula to discretize the differential operator of (5.1) in the presence of the state-dependent memory matrix $\boldsymbol{C}(\boldsymbol{x}(t))$. To this end, the MNA formulation (2.19) is written in the following from to

$$\boldsymbol{C}(\boldsymbol{x}(t))\frac{\mathrm{d}\boldsymbol{x}(t)}{\mathrm{d}t} + \boldsymbol{i}(\boldsymbol{x}(t)) = \boldsymbol{u}(t). \tag{5.1}$$

to take into account the state dependent $\boldsymbol{C}$ matrix. Also, it is assumed that the memoryless linear part of the MNA formulation $\boldsymbol{G}\boldsymbol{x}(t)$ has been absorbed in the nonlinear term $\boldsymbol{i}(\boldsymbol{x}(t))$ to simplify the mathematical derivation. The next section will present the proposed the development of the formulation.

5.1 Obreshkov-based Formulation for Circuits with Nonlinear Capacitors

We first state the notations that will be used throughout this chapter. First, $\mathbf{I}_r$ is used to denote an identity matrix with size $r \times r$, whereas $\mathbf{e}_{m,r}$ ($\mathbf{e}_{m,r}^\top$) is used to denote its mth column (respectively, row), with $\top$ marking the transposition operator. The ensuing presentation will also rely on the Kronecker matrix product operator, denoted $\otimes$, to simplify the mathematical derivations [33].

The basic objective of utilizing the Obreshkov formula is to compute the circuit waveforms, through advancing from a past time point, say t_n, to a future time point t_{n+1}, where $t_{n+1} = t_n + h$. To do so, the approximation of the waveform and its derivatives at $t = t_n$ (i.e., $\boldsymbol{x}_n^{(i)}$, $i = 0, \cdots, m$) are assumed to be readily available either from past calculations or through some suitable initialization procedures [2]. The objective, therefore, becomes the approximation of $\boldsymbol{x}(t)$ and its derivatives at $t = t_{n+1}$, i.e., $\boldsymbol{x}_{n+1}^{(i)} \in \mathbb{R}^N$, $i = 0, \cdots, k$. As a result, the Obreshkov formula in (3.1) involves $(k+1)N$ unknowns but offers only N equations. Thus, to carry out the solution process, the problem needs to be balanced (i.e., have a number of equations equal to the number of unknowns) through generating an independent set of additional kN equations. The procedure for generating this set of equations is described next.

The first step in this procedure, similar to previous work, is based on expanding $\boldsymbol{x}(t)$, $\boldsymbol{i}(\boldsymbol{x}(t))$ and $\boldsymbol{u}(t)$ using their Taylor series around $t = t_{n+1}$, i.e.,

$$\boldsymbol{x}(t) = \sum_{i=0}^{\infty} \frac{1}{i!} \boldsymbol{a}_{i,n+1} (t - t_{n+1})^i \tag{5.2}$$

$$\boldsymbol{i}(\boldsymbol{x}(t)) = \sum_{i=0}^{\infty} \frac{1}{i!} \boldsymbol{b}_{i,n+1} (t - t_{n+1})^i \tag{5.3}$$

$$\boldsymbol{u}(t) = \sum_{i=0}^{\infty} \frac{1}{i!} \boldsymbol{u}_{i,n+1} (t - t_{n+1})^i \tag{5.4}$$

The departure point from previous work, however, is the expansion of the matrix $\boldsymbol{C}(\boldsymbol{x}(t))$ by its Taylor series around t_{n+1},

$$\boldsymbol{C}(\boldsymbol{x}(t)) \quad = \quad \sum_{i=0}^{\infty} \frac{1}{i!} \boldsymbol{C}_{i,n+1} \left(t - t_{n+1}\right)^{i} \tag{5.5}$$

where,

$$\boldsymbol{a}_{i,n+1} \quad = \quad \left. \frac{\mathrm{d}^{i} \boldsymbol{x}(t)}{\mathrm{d}t^{i}} \right|_{t=t_{n+1}}, \tag{5.6}$$

$$\boldsymbol{C}_{i,n+1} \quad = \quad \left. \frac{\mathrm{d}^{i} \boldsymbol{C}\left(\boldsymbol{x}(t)\right)}{\mathrm{d}t^{i}} \right|_{t=t_{n+1}}, \tag{5.7}$$

$$\boldsymbol{b}_{i,n+1} \quad = \quad \left. \frac{\mathrm{d}^{i} \boldsymbol{i}\left(\boldsymbol{x}(t)\right)}{\mathrm{d}t^{i}} \right|_{t=t_{n+1}}, \tag{5.8}$$

$$\boldsymbol{u}_{i,n+1} \quad = \quad \left. \frac{\mathrm{d}^{i} \boldsymbol{u}\left(t\right)}{\mathrm{d}t^{i}} \right|_{t=t_{n+1}} \tag{5.9}$$

For simplicity, $\boldsymbol{C}_{i}$ will be used whenever the goal is to refer to $\boldsymbol{C}_{i,n+1}$.

Typically, working directly with the derivatives leads to numerical ill-conditioning that arises from the exponential growth in their numerical values, for higher i. To avoid this problem, the change of variables $t \rightarrow \tau h_{n+1} + t_{n+1}$ is used in (5.2)-(5.5). The resulting expressions are subsequently substituted into (5.1) to yield

$$\frac{1}{h} \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} j \tilde{\boldsymbol{C}}_{i} \tilde{\boldsymbol{a}}_{j} \tau^{i+j-1} + \sum_{i=0}^{\infty} \tilde{\boldsymbol{b}}_{i} \tau^{i} = \sum_{i=0}^{\infty} \tilde{\boldsymbol{u}}_{i} \tau^{i} \tag{5.10}$$

where,

$$\tilde{\boldsymbol{a}}_{i,n+1} \quad = \quad h_{n+1}^{i} \boldsymbol{a}_{i,n+1}, \tag{5.11}$$

$$\tilde{\boldsymbol{C}}_{i} \quad = \quad h_{n+1}^{i} \boldsymbol{C}_{i}, \tag{5.12}$$

$$\tilde{\boldsymbol{b}}_{i,n+1} \quad = \quad h_{n+1}^{i} \boldsymbol{b}_{i,n+1}, \tag{5.13}$$

$$\tilde{\boldsymbol{u}}_{i,n+1} \quad = \quad h_{n+1}^{i} \boldsymbol{u}_{i,n+1}, \tag{5.14}$$

are the h_{n+1}^{i}-scaled i^{th}-order derivatives of the various terms in the MNA formulation. This scaling of the derivatives maintains a good numerical conditioning in the following problem formulation.

The smoothness of $\boldsymbol{x}(t)$ at $t = t_{n+1}$ entails the smoothness of $\boldsymbol{x}(\tau)$ at $\tau = 0$. Taking the derivatives of (5.10), say q times, w.r.t. τ results in the following,

$$\frac{1}{h}\sum_{i=0}^{\infty}\sum_{j=0}^{\infty}\frac{j\,(j+i-1)\cdots(j+i-q)}{i!j!}\tilde{\boldsymbol{C}}_i\tilde{\boldsymbol{a}}_j\tau^{i+j-q-1} + \sum_{i=0}^{\infty}\frac{1}{(i-q)!}\tilde{\boldsymbol{b}}_i\tau^{i-q} = \sum_{i=0}^{\infty}\frac{1}{(i-q)!}\tilde{\boldsymbol{u}}_i\tau^{i-q} \tag{5.15}$$

which is valid for arbitrary q and τ.

The process of generating the extra equations, required to balance the system of equations and unknowns, is based on sweeping $q = 0, 1, 2, \cdots, k-1$, while setting $\tau = 0$. This procedure leaves only the terms that are τ-independent (those that have τ raised to the power zero). We illustrate this process next.

- For $q = 0$, setting $\tau = 0$ in (5.15) yields,

$$\frac{1}{h}\tilde{\boldsymbol{C}}_0\tilde{\boldsymbol{a}}_{1,n+1} + \tilde{\boldsymbol{b}}_{0,n+1} = \tilde{\boldsymbol{u}}_{0,n+1} \tag{5.16}$$

- For $q = 1$, setting $\tau = 0$ in (5.15) yields,

$$\frac{1}{h}\tilde{\boldsymbol{C}}_1\tilde{\boldsymbol{a}}_{1,n+1} + \frac{1}{h}\tilde{\boldsymbol{C}}_0\tilde{\boldsymbol{a}}_{2,n+1} + \tilde{\boldsymbol{b}}_{1,n+1} = \tilde{\boldsymbol{u}}_{1,n+1} \tag{5.17}$$

- For $q = k-1$, and $\tau = 0$, only τ-independent terms in (5.15) are left. Hence, I look for terms in which the power of τ vanishes. Those terms are collected next,

$$\frac{1}{h}\sum_{j=1}^{k}\frac{(k-1)!}{(k-j)!(j-1)!}\tilde{\boldsymbol{C}}_{k-j}\tilde{\boldsymbol{a}}_j + \tilde{\boldsymbol{b}}_{k-1} = \tilde{\boldsymbol{u}}_{k-1} \tag{5.18}$$

The system of equations (5.16)-(5.18) is indeed a system of kN equations, in the $(k+1)N$ unknowns $\tilde{\boldsymbol{a}}_{i,n+1} \in \mathbb{R}^N$, $i = 0, 1, \cdots, k$.

Balancing the system is done by utilizing the N Obreshkov formulae (3.1) to substitute for $\tilde{\boldsymbol{a}}_{k,n+1}$ in terms of the rest of unknowns, $\tilde{\boldsymbol{a}}_{i,n+1}, i = 0, \cdots, k-1$. This is done by recasting (3.1) (using (3.2), (5.11)) as follows

$$\tilde{\boldsymbol{a}}_{k,n+1} = (-1)^k \left(\sum_{i=0}^{m} \frac{\alpha_{i,m,k}}{\alpha_{k,k,m}} \tilde{\boldsymbol{a}}_{i,n} - \sum_{i=0}^{k-1} (-1)^i \frac{\alpha_{i,k,m}}{\alpha_{k,k,m}} \tilde{\boldsymbol{a}}_{i,n+1} \right) \tag{5.19}$$

Multiplying both sides of (5.19) by $\frac{\tilde{\boldsymbol{C}}_0}{h_{n+1}}$ and substituting for its right-side in (5.18) enables formulating the system in the compact matrix form,

$$\bar{\boldsymbol{\mathcal{C}}} \left(\boldsymbol{\xi}_{n+1} \right) \boldsymbol{\xi}_{n+1} + \boldsymbol{\rho}_{n+1} \left(\boldsymbol{\xi}_{n+1} \right) = \boldsymbol{\iota}_{n+1} \tag{5.20}$$

where the matrix $\bar{\boldsymbol{\mathcal{C}}} \left(\boldsymbol{\xi}_{n+1} \right) \in \mathbb{R}^{kN \times kN}$ is a matrix whose structure is shown at the bottom of the next page with

$$\beta_i = (-1)^{k+i+1} \frac{\alpha_{i,k,m}}{\alpha_{k,k,m}}, \tag{5.21}$$

and $\boldsymbol{\xi}_{n+1}$, $\boldsymbol{\rho}_{n+1}$, and $\boldsymbol{\iota}_{n+1}$ are vectors in $\mathbb{R}^{kN}$ given by

$$\begin{aligned}
\boldsymbol{\xi}_{n+1} &= \left[\tilde{\boldsymbol{a}}_{0,n+1}{}^\top,\ \tilde{\boldsymbol{a}}_{1,n+1}{}^\top,\ \cdots\ \tilde{\boldsymbol{a}}_{k-2,n+1}{}^\top,\ \tilde{\boldsymbol{a}}_{k-1,n+1}{}^\top \right]^\top \\
\boldsymbol{\rho}_{n+1} &= \left[\tilde{\boldsymbol{b}}_{0,n+1}{}^\top,\ \tilde{\boldsymbol{b}}_{1,n+1}{}^\top,\ \cdots\ \tilde{\boldsymbol{b}}_{k-2,n+1}{}^\top,\ \tilde{\boldsymbol{b}}_{k-1,n+1}{}^\top \right]^\top \\
\boldsymbol{\iota}_{n+1} &= \left[\tilde{\boldsymbol{u}}_{0,n+1}{}^\top,\ \tilde{\boldsymbol{u}}_{1,n+1}{}^\top,\ \cdots\ \tilde{\boldsymbol{u}}_{k-2,n+1}{}^\top,\ \tilde{\boldsymbol{v}}_{n+1}{}^\top \right]^\top
\end{aligned}$$

with, $\tilde{\boldsymbol{v}}_{n+1} = \tilde{\boldsymbol{u}}_{k-1,n+1} + (-1)^{k+1} \sum_{i=0}^{m} \frac{\alpha_{i,m,k}}{h \alpha_{k,k,m}} \tilde{\boldsymbol{C}}_0 \tilde{\boldsymbol{a}}_{i,n}$

It is important to remark that the $\bar{\boldsymbol{\mathcal{C}}}$ is in fact the generalization of the corresponding matrix derived earlier in [2] for a constant memory matrix. It should be noted that (5.20)

is a nonlinear system in $\boldsymbol{\xi}_{n+1}$, in which $\bar{\boldsymbol{\mathcal{C}}}\left(\boldsymbol{\xi}_{n+1}\right)$ is a matrix-valued nonlinear function in $\boldsymbol{\xi}_{n+1}$, whereas $\boldsymbol{\rho}_{n+1}$ is a vector-valued nonlinear function in $\boldsymbol{\xi}_{n+1}$.

Direct solution of this system requires the utilization of the Newton-Raphson iterative scheme which necessitates evaluating the Jacobian matrix of (5.20). This is discussed in the next section.

5.2 Formulation of the Jacobian matrix

The Jacobian matrix, denoted $\boldsymbol{J}_{n+1}$, can be expressed as the sum of two terms,

$$\boldsymbol{J}_{n+1} = \frac{\partial}{\partial \boldsymbol{\xi}_{n+1}} \left(\bar{\boldsymbol{\mathcal{C}}}\left(\boldsymbol{\xi}_{n+1}\right) \boldsymbol{\xi}_{n+1} \right) + \frac{\partial \boldsymbol{\rho}_{n+1}}{\partial \boldsymbol{\xi}_{n+1}} \tag{5.22}$$

The derivation and computation of the second term was detailed in previous work [2,4]. The objective in the remainder of this section, however, is focused on deriving and computing the first term in (5.22). In general, the derivative of a matrix with respect to a vector is a three-dimensional matrix, or a tensor, in which each "layer" is the derivative of

---

$$\bar{\boldsymbol{\mathcal{C}}}\left(\boldsymbol{\xi}_{n+1}\right) =$$

$$\frac{1}{h}\begin{bmatrix} \mathbf{0} & \tilde{\boldsymbol{C}}_0 & \mathbf{0} & \mathbf{0} & . & \mathbf{0} \\ \mathbf{0} & \tilde{\boldsymbol{C}}_1 & \tilde{\boldsymbol{C}}_0 & \mathbf{0} & . & \mathbf{0} \\ \mathbf{0} & \tilde{\boldsymbol{C}}_2 & 2\tilde{\boldsymbol{C}}_1 & \tilde{\boldsymbol{C}}_0 & . & \mathbf{0} \\ . & . & & & & . \\ . & . & & . & & \tilde{\boldsymbol{C}}_0 \\ \beta_0 \tilde{\boldsymbol{C}}_0 & \tilde{\boldsymbol{C}}_{k-1} + \beta_1 \tilde{\boldsymbol{C}}_0 & (k-1)\tilde{\boldsymbol{C}}_{k-2} + \beta_2 \tilde{\boldsymbol{C}}_0 & \frac{(k-1)!}{2!(k-3)!}\tilde{\boldsymbol{C}}_{k-3} + \beta_3 \tilde{\boldsymbol{C}}_0 & . & (k-1)\tilde{\boldsymbol{C}}_1 + \beta_{k-1}\tilde{\boldsymbol{C}}_0 \end{bmatrix}$$

the matrix w.r.t. a scalar component in that vector. This tensor can be more conveniently represented as a two-dimensional matrix with the help of the Kronecker product. For example, assuming that the entries of a matrix $\boldsymbol{A} \in \mathbb{R}^{K\times K}$ are functions of a vector $\boldsymbol{y}$, then the derivative $\partial\boldsymbol{A}/\partial\boldsymbol{y}$ may be expressed as a matrix in $\mathbb{R}^{K\times K^2}$ given by

$$\frac{\partial \boldsymbol{A}}{\partial \boldsymbol{y}} = \sum_{l=1}^{K} \mathbf{e}_{l,K}^{\top} \otimes \frac{\partial \boldsymbol{A}}{\partial y_l} \tag{5.23}$$

where y_l is the lth component in $\boldsymbol{y}$, and $\frac{\partial \boldsymbol{A}}{\partial y_l}$ is the *entry-wise* derivative of the matrix $\boldsymbol{A}$ with respect to the scalar y_l.

Using this idea, I can then write the first term in the Jacobian matrix in (5.22) as follows,

$$\begin{aligned} &\frac{\partial}{\partial \boldsymbol{\xi}_{n+1}} \left(\bar{\boldsymbol{\mathcal{C}}} \left(\boldsymbol{\xi}_{n+1} \right) \boldsymbol{\xi}_{n+1} \right) = \\ &\sum_{i=0}^{k-1} \underbrace{\left(\mathbf{e}_{i+1,k}^{\top} \otimes \frac{\partial \bar{\boldsymbol{\mathcal{C}}}}{\partial \tilde{\boldsymbol{a}}_{i,n+1}} \right)}_{\in \mathbb{R}^{kN\times k^2N^2}} \underbrace{\left(\mathbf{I}_{kN} \otimes \boldsymbol{\xi}_{n+1} \right)}_{\in \mathbb{R}^{k^2N^2\times kN}} + \bar{\boldsymbol{\mathcal{C}}} \end{aligned} \tag{5.24}$$

In a similar manner, I have

$$\frac{\partial \bar{\boldsymbol{\mathcal{C}}}}{\partial \tilde{\boldsymbol{a}}_{i,n+1}} = \sum_{l=1}^{N} \mathbf{e}_{l,N}^{\top} \otimes \frac{\partial \bar{\boldsymbol{\mathcal{C}}}}{\partial \tilde{a}_{l,i,n+1}} \tag{5.25}$$

where $\tilde{a}_{l,i,n+1}$ is the lth component in $\tilde{\boldsymbol{a}}_{i,n+1}$.

It should be noted that the Jacobian matrix is a matrix of size $kN \times kN$, notwithstanding the fact that it is represented as the Kronecker product of very sparse matrices of larger dimensions.

Therefore, (5.25) shows that computing the first term of the Jacobian matrix can be carried out by computing the derivatives of the individual blocks of $\bar{\boldsymbol{\mathcal{C}}}$, i.e. $\tilde{\boldsymbol{C}}_j, j = 0, \ldots, k-1$, with respect to $\tilde{a}_{l,i,n+1}$. The following lemma aims at simplifying this computation.

Lemma 1 *Assume that the memory matrix* $\boldsymbol{C}(\boldsymbol{x}(t))$ *can be differentiated with respect to the* l^{th} *component in* $\boldsymbol{x}$*,* x_l*. Furthermore, assume that such derivative is sufficiently smooth to enable the expansion in the Taylor series format,*

$$\frac{\partial \boldsymbol{C}(\boldsymbol{x}(t))}{\partial x_l} = \sum_{p=0}^{\infty} \frac{1}{p!} \boldsymbol{L}_l^{(p)} \left(t - t_{n+1}\right)^p \tag{5.26}$$

where

$$\boldsymbol{L}_l^{(p)} = \frac{\mathrm{d}^p}{\mathrm{d}t^p} \left(\frac{\partial \boldsymbol{C}(\boldsymbol{x}(t))}{\partial x_l} \right) \Bigg|_{t=t_{n+1}} . \tag{5.27}$$

It then follows that

$$\frac{\partial \tilde{\boldsymbol{C}}_j}{\partial \tilde{a}_{l,i,n+1}} = \begin{pmatrix} j \\ i \end{pmatrix} \tilde{\boldsymbol{L}}_l^{(j-i)} \tag{5.28}$$

where $\tilde{\boldsymbol{L}}_{n+1}^{p} = h_{n+1}^{p} \boldsymbol{L}_{n+1}^{(p)}$

Proof: Proceeding from the definition of $\tilde{\boldsymbol{C}}_j$ in (5.11), I can write

$$\begin{aligned} \frac{\partial \tilde{\boldsymbol{C}}_j}{\partial \tilde{a}_{l,i,n+1}} &= \frac{\partial}{\partial \tilde{a}_{l,i,n+1}} \left[h_{n+1}^{j} \frac{\mathrm{d}^j \boldsymbol{C}(\boldsymbol{x}(t))}{\mathrm{d}t^j} \right] \Bigg|_{t=t_{n+1}} \\ &= h_{n+1}^{j} \frac{\mathrm{d}^j}{\mathrm{d}t^j} \left[\frac{\partial \boldsymbol{C}(\boldsymbol{x}(t))}{\partial \tilde{a}_{l,i,n+1}} \right] \Bigg|_{t=t_{n+1}} \\ &= h_{n+1}^{j} \frac{\mathrm{d}^j}{\mathrm{d}t^j} \left[\frac{\partial \boldsymbol{C}(\boldsymbol{x}(t))}{\partial x_l} \frac{\partial x_l}{\partial \tilde{a}_{l,i,n+1}} \right] \Bigg|_{t=t_{n+1}} \end{aligned} \tag{5.29}$$

By definition, we have

$$x_l(t) = \sum_{v=0}^{\infty} \frac{1}{v!} \tilde{a}_{l,v,n+1} \frac{(t - t_{n+1})^v}{h^v} \tag{5.30}$$

Hence, substitution from (5.30) and (5.26) into (5.29) and proceeding with the following

manipulation yields,

$$\begin{aligned}
\frac{\partial \tilde{\boldsymbol{C}}_j}{\partial \tilde{a}_{l,i,n+1}} &= h^{(j-i)} \frac{\mathrm{d}^j}{\mathrm{d}t^j} \left[\sum_{p=0}^{\infty} \frac{1}{p!} \boldsymbol{L}_l^{(p)} (t - t_{n+1})^{p+i} \right] \Bigg|_{t=t_{n+1}} \\
&= \frac{j!}{(j-i)!i!} h^{j-i} \boldsymbol{L}_l^{(j-i)} \\
&= \begin{pmatrix} j \\ i \end{pmatrix} \tilde{\boldsymbol{L}}_l^{(j-i)}
\end{aligned} \tag{5.31}$$

which proves the lemma.

5.3 Implementation and Computational Complexity

5.3.1 Notes on Implementation

Lemma 1 makes it clear that computing the Jacobian term due to the nonlinear capacitor requires only computing the h_{n+1}^p-scaled p^{th}-order derivatives of the matrix $\frac{\partial \boldsymbol{C}(\boldsymbol{x}(t))}{\partial x_l}$ at $t = t_{n+1}$, i.e., the $\tilde{\boldsymbol{L}}_l^{(p)}$ matrix. Such a matrix will have a non-zero entry only where the corresponding entry in the matrix $\boldsymbol{C}(\boldsymbol{x}(t))$ is a function of the lth component of $\boldsymbol{x}(t)$. Given that $\boldsymbol{C}(\boldsymbol{x}(t))$ is structurally sparse, with each entry being a function of at most two or three components in $\boldsymbol{x}(t)$, the matrices $\tilde{\boldsymbol{L}}_l^{(p)}$ will have very few non-zero entries. It should also be obvious that $\tilde{\boldsymbol{L}}_l^{(p)}$ depends, although not explicitly, on $\tilde{\boldsymbol{a}}_{i,n+1}$, i.e., the components of $\boldsymbol{\xi}_{n+1}$, the vector of unknowns in the system (5.20).

Typical execution of the NR iterative solution starts with an initial guess for the vector of unknowns, $\boldsymbol{\xi}_{n+1}$. This initial guess is used to assign values for the individual subvector components or $\tilde{\boldsymbol{a}}_{i,n+1}, i = 0, \cdots, k-1$. Using those subvectors, the corresponding values for $\tilde{\boldsymbol{L}}_l^{(p)}, p = 0, \cdots, k-1$ can be computed using the concept of rooted trees, which has been detailed in [2]. The nonlinear expression entries in the matrix $\frac{\partial \boldsymbol{C}(\boldsymbol{x}(t))}{\partial x_l}$ are calculated based on the new rooted tree-based method which has been explained in Chapter 4.

5.3.2 Analysis of Computational Complexity

Facrotization of the Jacobian matrix $\boldsymbol{J}_{n+1} \in \mathbb{R}^{kN \times kN}$ represents an important part of the computational effort spent per a single time step. The structure of this matrix can be best described by considering the $N \times N$ Jacobian matrix arising from the classical low-order methods used by SPICE, while replacing each scalar entry by a $k \times k$ block. The structurally zero entries in the Jacobian of low-order methods are kept as zero blocks in the high-order Jacobian, whereas the non-zero entries in the low-order becomes a non-zero block in the high-order. Such a block can be diagonal, lower triangular, lower Hessenberg, or a full block depending on the circuit topology.

It was shown in [4] that the ideal approach to factorize the high-order Jacobian matrix is through extending the matrix factorization packages suitable for circuit simulation, e.g. the KLU [34], to block-oriented version. In that sense, the scalar arithmetic operations, e.g. multiplications and divisions, that are performed by those packages are replaced by block operations, i.e. $(k \times k)$ matrix inversions and multiplications. This part has been explained in detail in Section 3.3.

It has been empirically observed [35] that the number of arithmetic operations involved in factorizing a sparse matrix of size $(N \times N)$ grows with factor proportional to N^{α}, denoted $O\left(N^{\alpha}\right)$, where α is parameter that is typically in the range $1.1 \leq \alpha \leq 1.2$, and approaches 1 for very large and sparse matrices.

Therefore, the number of arithmetic operations arising in the course of factorizing the $kN \times kN$ Jacobian matrix (using the block-oriented version) will be scaled by a k-dependent factor. Such a factor could range from k to k^3 depending on the type of the block operands. Thus, it is possible to estimate the computational cost of factorizing the Jacobian matrix $\boldsymbol{J}_{n+1}$ to be within the range of $O\left(kN^{\alpha}\right)$ to $O\left(k^3 N^{\alpha}\right)$.

There are several additional remarks related to the computational complexity that should be made at this point.

1. The block-oriented factorization algorithm [4] maintains the same computational complexity in regards to the circuit size N as it is for the low-order methods. This statement can be empirically verified by examining the increase in the CPU time versus circuit size N, for the high-order Obreshkov-based method, and comparing it with the growth of the CPU time of low-order methods such as the Gear's or the TR method. In fact, the circuit shown in Figure 5.1 as been used to serve this purpose. This circuit represents transmission lines modeled with various number of sections to produce circuits with different sizes in its MNA formulation, N, where the CPU matrix factorization can be measured for different integration methods. The results obtained from this experiment are shown in Figure 5.2. Figure 5.3 shows the results corresponding to the triangular solution, i.e., forward/backward (F/B) substitution.

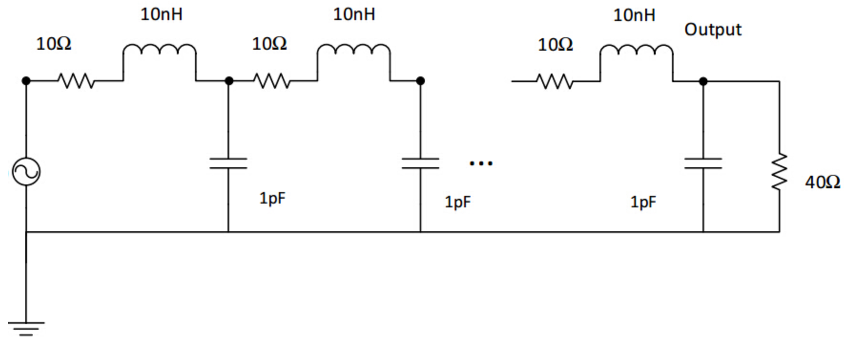


Figure 5.1: Schematic for the circuit used to obtain the results of Figure 5.2

2. The above results indicate that the growth in the size of the circuit formulation, N, affects, almost in the same manner, the CPU time of factorizing the matrix of the high-order Obreshkov-based method, as it does the matrix factorization of the low-order TR, at each time step. This feature is attributed to the success of the block factorization algorithm in keeping the fill-ins to the same order-of-magnitude, in terms of N, as it is for low-order methods [4].

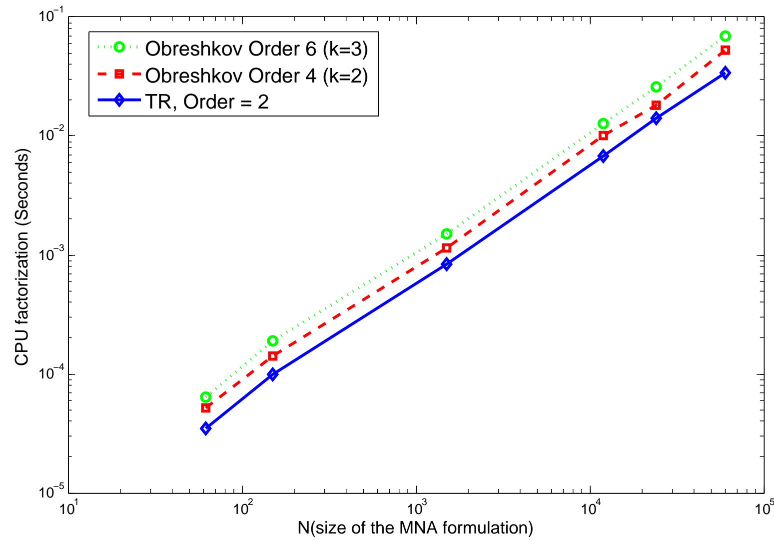


Figure 5.2: Illustrating the growth in the complexity of matrix factorization versus the circuit MNA size, N, for the low-order TR and high-order Obreshkov-based methods

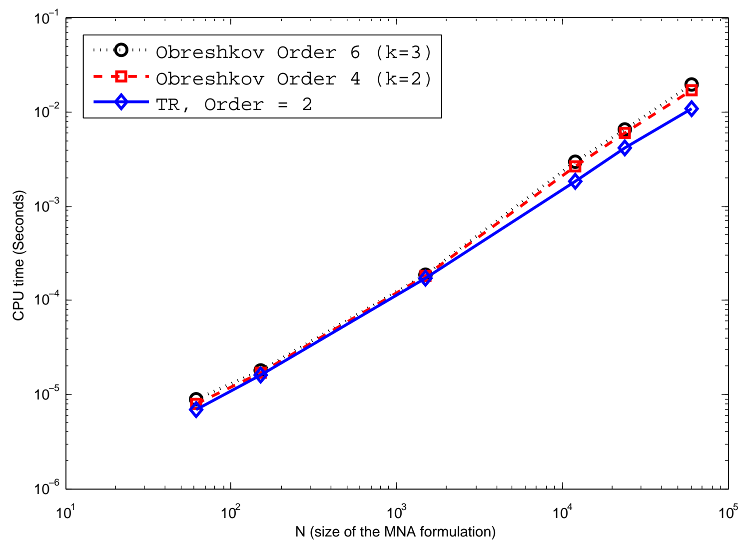


Figure 5.3: Illustrating the growth in the complexity of triangular (F/B) solution factorization versus the circuit MNA size, N, for the low-order TR and high-order Obreshkov-based methods

3. The increase in the computational efforts in factorization the Jacobian matrix at each time step is typically offset by the significant reduction in the number of time points afforded by the high-order approximation of the Obreshkov formula. It should also be stressed that numerical stability inherent in the Obreshkov formula is the crucial factor without which the high-order approximation loses its effectiveness.

Similar arguments can be made in regards to the complexity of the memory usage of the Obreshkov-based algorithm. It should be obvious that the required memory storage will depend on the block size, k, and the number of fill-in blocks resulting from the factorization. As stated earlier, the number of fill-in blocks in the block version is identical to the fill-in of the regular version of the KLU[36], which is typically proportional to N^{α}, with α factor that approaches unity from above. Thus, a realistic estimate for the memory storage requirements should range from $O(kN^{\alpha})$ to $O(k^2N^{\alpha})$.

5.3.3 Numerical Stability

As mentioned earlier, the numerical stability of the method can be proved by studying the successive approximations it produces for scalar test problem $\dot{x} = \lambda x(t)$. This was achieved in Theorem 3 in [2], where it has been proved that the Obreshkov-based method is A-stable if and only if $k - 2 \leq m \leq k$, and L-stable if and only if $k - 2 \leq m < k$.

5.4 Numerical Examples

Circuit examples are considered to demonstrate the validity and efficiency of the Obreshkov-based method. The first example provides the necessary proof of concept required to validate the computational procedure described in Sections 5.1 and 5.2.

5.4.1 Example 1: Circuit with Nonlinear Capacitor

The purpose of this example is to numerically validate the computational derivations presented above for a circuit with a nonlinear capacitor. The methodology used to validate the proposed computational approach is based on verifying the practical results obtained through contrasting it with the anticipated theoretical results. The underlying theory of the Obreshkov method [2] states that a solution obtained based on the Obreshkov formula (3.1) with general k, m at $t = t_1$, denoted here by $\boldsymbol{z}_1^{(k,m)}$, approximates the exact solution, denoted $\boldsymbol{z}_{\text{exact}}(t)$, so that the approximation error is given by

$$\text{Error} = \left|\boldsymbol{z}_1^{(k,m)} - \boldsymbol{z}_{\text{exact}}(t_1)\right| = Ch_1^{k+m+1} + O\left(h^{k+m+2}\right) \tag{5.32}$$

when $\boldsymbol{z}_0^{(k,m)} = \boldsymbol{z}_{\text{exact}}(t_0)$, where $h_1 = t_1 - t_0$, and C is a constant [2]. (5.32) shows that as $h \to 0$, the error becomes asymptotically proportional to h_1^{k+m+1}. That fact represents the main idea used in the verification of the proposed approach, where the behaviour of the error computed above is examined versus the step size h_1, and compared versus the asymptotic error behaviour predicted by theory (5.32).

The only difficulty to this approach, however, is that finding the exact solution is not feasible in general systems of nonlinear differential equations, since such systems do not have analytical solutions that can serve as the exact solution in the error analysis of (5.32). This difficulty is overcome by proceeding in the following steps.

In the first step, the circuit is stimulated using a sinusoidal source with period T. This causes the steady-state response of the circuit to be periodical (with period T). Such a steady-state response (denoted $x_{\text{ss}}(t)$) is typically represented by a Fourier series, in $(2\pi/T)t$, whose coefficients that can be computed using standard steady-state analysis such as the Harmonic Balance (HB) approach [37].

In the second step, the Fourier series representing the steady-state response is eval-

uated at an arbitrary time instant, e.g. $t = 0$. The vector $x_{\text{ss}}(0)$ computed this way is then used as the the initial value $t = 0$ for the DAE of the MNA system in (5.1). It is obvious that, under this initial condition, the *exact* transient solution of the DAE is indeed the steady-state solution. Hence $x_{\text{ss}}(t)$ can serve as the reference $\boldsymbol{z}_{\text{exact}}(t)$ in the error measurement of (5.32).

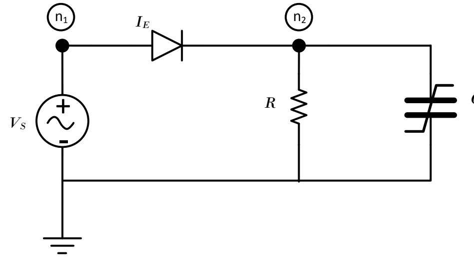


Figure 5.4: A circuit example used in numerical validation. $V_S = 1.25 \times sin(2000\pi t)$, $I = 10^{-15} \times (\exp((V_{n_1} - V_{n2})/0.025) - 1)$, $R = 1\Omega$, $C = 10^{-3} \times e^{V_{n_2}}$

Figure 5.4 shows the circuit used in the validation. This circuit represents a simple rectifier circuit constructed with a nonlinear capacitor and excited by a sinusoidal source. The HB technique was invoked to compute the steady-state waveform, $x_{\text{ss}}(t)$, where this response, and its derivatives, at $t = 0$ were used to supply the initial condition for the Obreshkov method. The solution obtained from the Obreshkov method at $t = h$, was then used as the value of $\boldsymbol{z}_1^{(k,m)}$ in (5.32), whereas $\boldsymbol{z}_{\text{exact}}(t_1)$ was substituted for by $x_{\text{ss}}(h_1)$.

Figure 5.5 presents a log-scale graph for the error computed by (5.32) versus the step size, h_1, for several values of k and m. The solid-lines in those graphs show the expected asymptotical behaviour of the error versus the step size h_1, which is given by a line with a slope equal to $k + m + 1$, based on (5.32).

As can be observed from those graphs, the actual error calculated from (5.32) does agree with the predictions based on the theory, thereby confirming the validity of the

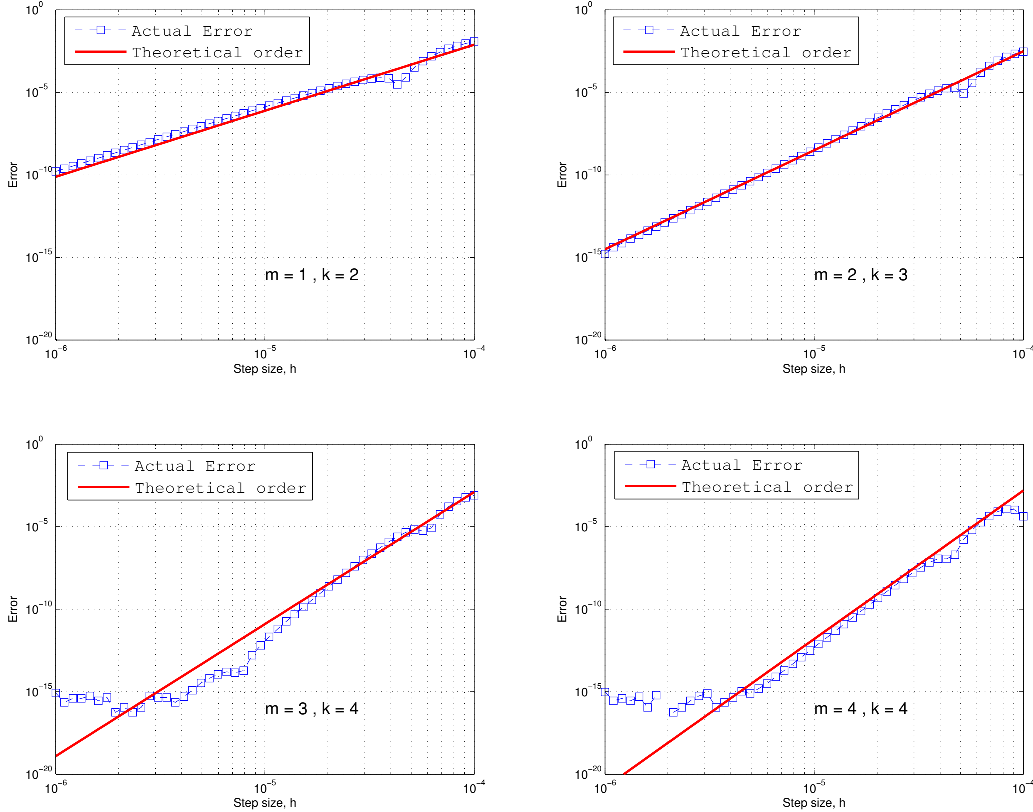


Figure 5.5: The behavior of the error computed by (5.32) vs. the step size h_1, for different values of k and m.

computational approach derived in this work.

A noteworthy observation pertaining to the lower two panels of Figure 5.5 is the "freezing" of the error around the level of machine precision, i.e., 10^{-15}, which indicates that the approximation obtained from the high order has become almost identical to the exact solution.

5.4.2 Example 2: Large Circuit Example

The circuit considered for this example is a Transmission-Line (TL)-based circuit that consists of three segments, each with a 64 coupled conductors. The TLs were represented

by lumped sections of RLCG elements using the techniques illustrated in [38]. The size of MNA formulation for the circuit, N, was 18,241 variables. Figure 5.6 presents the circuit schematic of this example.

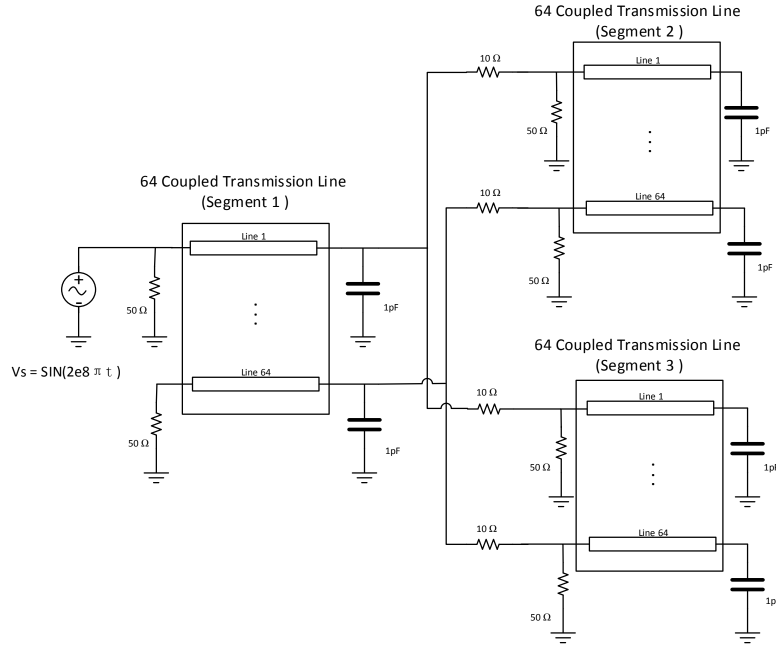


Figure 5.6: A TL-based circuit with $N = 18,241$ variables.

The circuit was stimulated with a sinusoidal source with frequency 0.2GHz and 1V amplitude. The transient response of the circuit was simulated using the Obreshkov-based method and the Gear's method [15] for about 10 periods of $10T$, where T is the period of the stimulus.

Figure 5.7 shows a snapshot of the results of the transient simulation for the voltage of the node at the far-end of Line 6 in the second TL segment. This figure compares the waveforms obtained from the Gear's method (Orders 2 and 3) method with the Obreshkov method (with $m = n = 3$, i.e., for order 6). A noteworthy observation related to those results (besides the reduction in the number of points produced by the Obreshkov method) is that the Order-3 Gear's method started to exhibit numerical instability. In

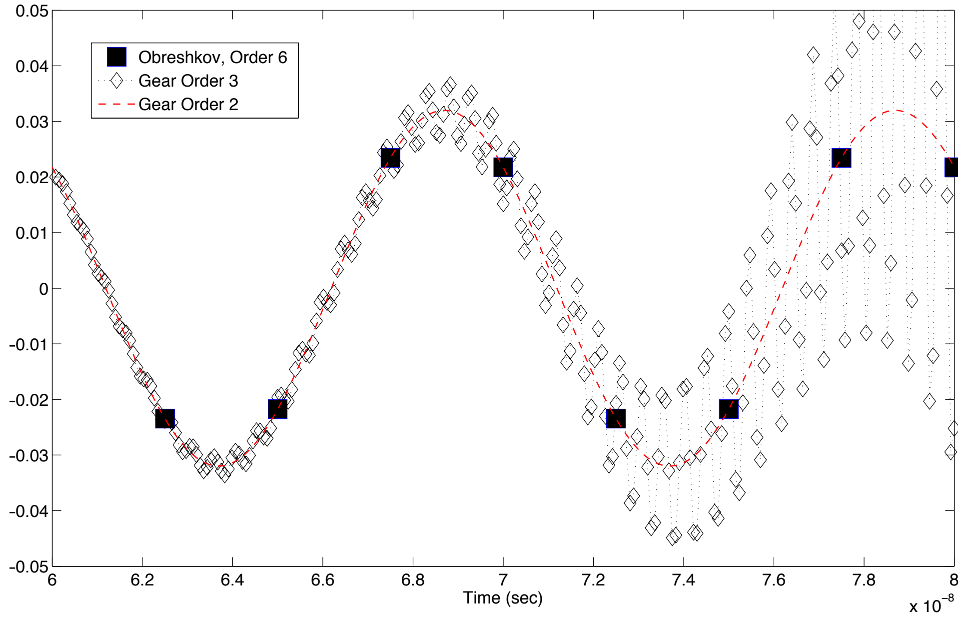


Figure 5.7: Transient response at the far-end of the 2$^{\text{nd}}$ line of segment 2 in the circuit shown in Figure 5.6.

fact, the Gear's method for orders higher than 3 have all become numerically unstable after few cycles of the transient simulation.

Figure 5.8 shows this instability by depicting the relative error obtained from the Gear's method, for the voltage at the same node, and indicating its exponential growth. It is worth noting here that although the Gear's method is $A(\alpha)$-stable [39] (for orders 3-6), the lack of A-stability in the high order is what causes the numerical instability illustrated in this example.

Table 5.4.2 summarizes the CPU time taken by both the Obreshkov and Gear's method (order 2), which has the same A-stability characteristic. The second column in the table shows the maximum % relative error observed in both methods, where the error in this case is calculated using the method utilized in the previous example. The third column shows the number of time steps that each method needed to take. The fourth and fifth columns indicate the CPU time taken in a single matrix factorization and single

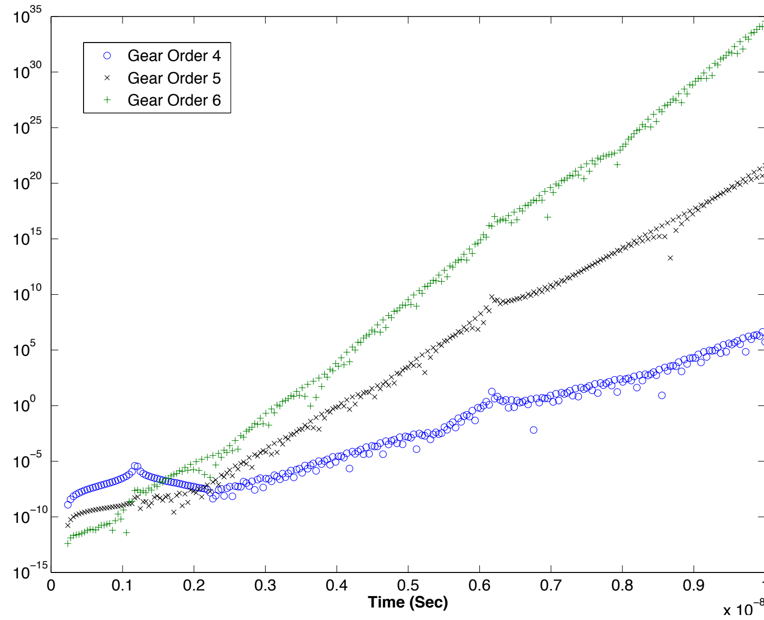


Figure 5.8: Relative error in the transient response computed using the Gear's method, orders 4,5, and 6.

triangular (F/B). The results in this example suggest An overall speedup of 14.5 times.

Worthy of note here is the fact that the speedup of the Obreshkov method arises mainly from the savings in computed time points. the speedup factor, however, depends on the computational effort at each time step, and therefore could vary from circuit to circuit. For example, a linear circuit, such as the one used in this example, when simulated with constant step size will require only one LU factorization at the initial point, $t = 0$, and a triangular solution at each time, in which case the total time of the initial LU factorization will have a significant weight, effectively influencing the speedup factor.

Method	Max Relative Error	# of computed points	CPU time (seconds)		Speedup
			One LU	One F/B	
Gear (Order 2)	6.25×10^{-2}	1281	1.2	0.03	-
Obreshkov (Order 6)	1.57×10^{-2}	20	5.3	0.05	14.5

Table 5.1: CPU comparison between the Obreshkov-based and Gear's methods for the TL circuit of Figure 5.6

5.4.3 Example 3: Large Circuit with Nonlinear Capacitors Example

The circuit considered is the same with the above example except that we change all the capacitors to nonlinear ones which is shown in Figure 5.9.

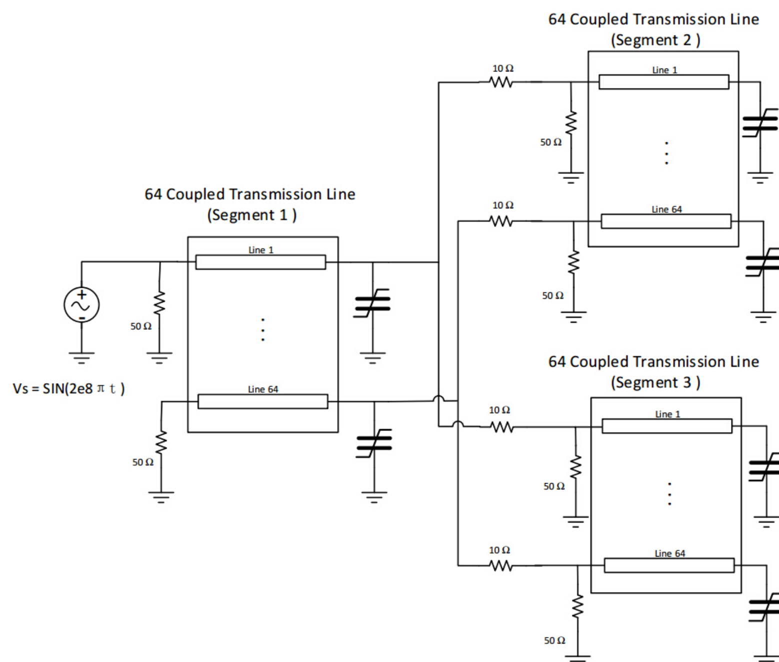


Figure 5.9: A TL-based circuit with nonlinear capacitors.

where the capacitors become

$$C = 1 \times 10^{-12} \exp(v_{node}) \tag{5.33}$$

and the v_{node} means the voltage of node connected with the capacitor as its the other side is always connected to the ground.

We use the same simulation environment with the second example and get the result which is shown in Figure 5.10. We can see that the proposed technique applied on the large circuit with nonlinear capacitors is stable, while the one based on the Gear's method with order 3 has become unstable.

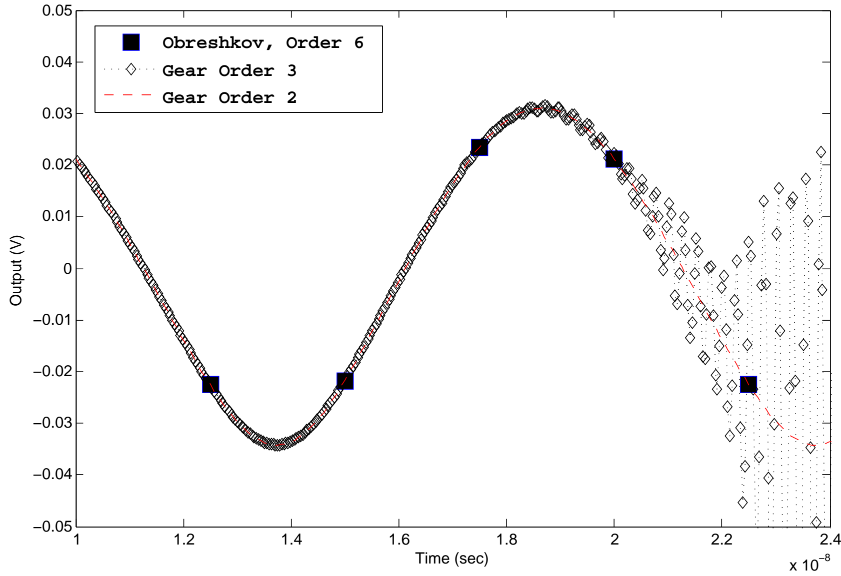


Figure 5.10: Transient response at the far-end of the 2$^{\text{nd}}$ line of segment 2 in the circuit shown in Fig. 5.9

Chapter 6

Conclusion and Future Work

6.1 Concluding Remarks

The goal of this thesis is to improve the Obreshkov-based high order method and generalize it to handle different types of circuits.

The first aspect is to present a new method to improve the efficiency of computing high order derivatives in the Obreshkov-based method. Because of the situation that rooted tree method was conceived, the tree is traversed from the root node up to the leaf nodes and then back down again to the root node for each order of the derivatives. The proposed method handles this aspect of the computation in a more efficient way where the tree is traversed only one time, which reduces the computational cost effectively.

The second aspect of this thesis is to present a generalization of the Obreshkov-based formulation. Previous Obreshkov-based discretizations were only formulated to handle those elements based on the charge-based MNA formulation. The presented generalization is aimed at enabling the Obreshkov-based transient simulator to handle nonlinear circuits in which the nonlinear memory elements such as capacitors or inductors can be represented by their capacitance or inductance. The formulation and ensuing

computational steps were verified by several numerical validation examples.

6.2 Future Work

This thesis has generalized the Obreshkov-based high order method to handle memory elements that are described by nonlinear constituent equations. In this generalization, the most important change is that the constant $\boldsymbol{C}$ becomes $\boldsymbol{x}(t)$-dependent matrix. We expect to extend this generalization to the Harmonic Balance(HB) analysis, which is the most widely used frequency domain method for calculating the steady state response of nonlinear circuits [40, 41]. The HB method can also handle only circuits in which the nonlinearity of memory elements is described as a charge- or flux-based nonlinear function. It is possible to transfer the technique in this thesis to the HB method, which will make the $\boldsymbol{C}$ become $\boldsymbol{x}(t)$-dependent matrix. Furthermore, in the processing of Jacobian matrix in HB method, it will utilize the same technique (5.23) to calculate the derivative of the matrix ($\boldsymbol{C}(\boldsymbol{x}(t))$) with respect to the vector ($\boldsymbol{x}(t)$).

Appendix A

Netlist of the Double-Balanced Mixer Circuit

This appendix provides the netlist for the mixer circuit that was used in Figure 4.2 from level 1 to level 3.

A.1 Level 1

```
Vdd1 ndd1 gnd 15
Vdd2 ndd2 gnd 15
Io n7 gnd 10m
Vlo1 nVlo1 gnd SIN( 0.125 1G 0)
Vlo2 nVlo2 gnd SIN( 0.125 1G 180)
Vrf1 nVrf1 gnd SIN( 0.0125 900E6 0)
Vrf2 nVrf2 gnd SIN( 0.0125 900E6 180)
rd1 ndd1 n3 1000
rd2 ndd2 n4 1000
ri nVout1 nVout2 1000

rkk1 n1 gnd 1G
```

```
rkk2 n2 gnd 1G
rkk3 nvout1 gnd 1G
rkk4 nvout2 gnd 1G

xT1 nVlo1 n5 n7 JFET
xT2 nVlo2 n6 n7 JFET
xT3 nVrf1 n3 n5 JFET
xT4 nVrf2 n4 n5 JFET
xT5 nVrf1 n4 n6 JFET
xT6 nVrf2 n3 n6 JFET

xa1 n1 n3 crys lm=0.190985740724534 cm=1.326289866142595e-17
xa2 nVout2 n1 crys lm=0.190985740724534 cm=1.326289866142595e-17
xa3 nVout1 n2 crys lm=0.190985740724534 cm=1.326289866142595e-17
xa4 n2 n4 crys lm=0.190985740724534 cm=1.326289866142595e-17

xb1 nVout1 n1 crys lm =0.190986122696397 cm= 1.326292518724980e-17
xb2 n1 n4 crys lm =0.190986122696397 cm= 1.326292518724980e-17
xb3 nVout2 n2 crys lm =0.190986122696397 cm= 1.326292518724980e-17
xb4 n2 n3 crys lm =0.190986122696397 cm= 1.326292518724980e-17

.subckt crys nA nB lm=0.190985740724534 cm=1.326289866142595e-17
rm nA nC 120
co nA nB 5e-12
lm1 nC nD lm
cm1 nD nB cm
.ends

.subckt JFET ng nd ns
GDsub1 ng ns1 cur = '1e-15*(exp((v(ng)-v(ns))/0.025)-1)'
GDsub2 ng nd1 cur = '1e-15*(exp((v(ng)-v(nd))/0.025)-1)'
rs ns1 ns 1
```

rd nd1 nd 1

Gids nd1 ns1 cur='

@ VTO = -1, @ SCALM = 1, @ M = 1, @ L = 1u, @ LAMBDA = 0.333, @ BETA = 0.1, @ W = 1u, @ WDEL = 0, @ LDEL = 0, @ WDELeff = WDEL*SCALM, @ Weff = W*SCALM + WDELeff, @ LDELeff = LDEL*SCALM, @ Leff = L*SCALM + LDELeff, @ BETAeff = BETA*(Weff*M)/Leff, @ VGS = V(ng) - V(ns1), @ VGST = VGS - VTO, @ VDS = V(nd1) - V(ns1), @VGD = V(ng)-v(nd1),

@ IF (VGST < 0)

@ {0.0}

@ ELSE

@ {

@ IF (VGST < VDS)

@ {BETAeff*VGST*VGST*(1+LAMBDA*VDS)}

@ ELSE

@ {BETAeff*VDS*(2*VGST-VDS)*(1+LAMBDA*VDS)}

@ }'

CgD1 ng nd1 c='

@ M =0.5, @ BETAeff = BETA*(Weff*M)/Leff, @ VGS = V(ng) - V(ns1), @ VGST = VGS - VTO, @ VDS = V(nd) - V(ns), @VGD = V(ng)-v(nd1), @cgd=0.27p, @N=1, @vt=2.56929E-2, @PB=0.8, @AREA = Weff/Leff, @AREAeff=M*AREA, @IS=1E-14, @ISeff=IS*AREAeff, @CGDeff = CGD*AREAeff,

@ IF (VGD < 0)

@ {CGDeff*1/POW((1-VGD/PB),M)}

@ ELSE

@ {CGDeff*(1+M*VGD/PB)}'

CgS1 ng ns1 c ='

@ M = 0.5, @ L = 1u, @ BETAeff = BETA*(Weff*M)/Leff, @ VGS = V(ng) - V(ns1), @ VGST = VGS - VTO, @ VDS = V(nd) - V(ns), @VGD = V(ng)-v(nd1), @cgs=0.27p, @N=1, @vt=2.56929E-2, @PB=0.8, @AREA = Weff/Leff, @AREAeff=M*AREA, @IS=1E-14, @ISeff=IS*AREAeff, @CGSeff = CGS*AREAeff,

```
@ IF (VGS < 0 )
@ {CGSeff*1/POW((1-VGS/PB),M)}
@ ELSE
@ {CGSeff*(1+M*VGS/PB)}'
.ends
.end
```

A.2 Level 2

```
Vdd1 ndd1 gnd 15
Vdd2 ndd2 gnd 15
Io n7 gnd 10m

Vlo1 nVlo1 gnd SIN( 0.125 1G 0)
Vlo2 nVlo2 gnd SIN( 0.125 1G 180)
Vrf1 nVrf1 gnd SIN( 0.0125 900E6 0)
Vrf2 nVrf2 gnd SIN( 0.0125 900E6 180)
rd1 ndd1 n3 1000
rd2 ndd2 n4 1000
ri nVout1 nVout2 1000

rkk1 n1 gnd 1G
rkk2 n2 gnd 1G
rkk3 nvout1 gnd 1G
rkk4 nvout2 gnd 1G

xT1 nVlo1 n5 n7 JFET
xT2 nVlo2 n6 n7 JFET
xT3 nVrf1 n3 n5 JFET
xT4 nVrf2 n4 n5 JFET
xT5 nVrf1 n4 n6 JFET
```

```
xT6 nVrf2 n3 n6 JFET

xa1 n1 n3 crys lm=0.190985740724534 cm=1.326289866142595e-17
xa2 nVout2 n1 crys lm=0.190985740724534 cm=1.326289866142595e-17
xa3 nVout1 n2 crys lm=0.190985740724534 cm=1.326289866142595e-17
xa4 n2 n4 crys lm=0.190985740724534 cm=1.326289866142595e-17

xb1 nVout1 n1 crys lm =0.190986122696397 cm= 1.326292518724980e-17
xb2 n1 n4 crys lm =0.190986122696397 cm= 1.326292518724980e-17
xb3 nVout2 n2 crys lm =0.190986122696397 cm= 1.326292518724980e-17
xb4 n2 n3 crys lm =0.190986122696397 cm= 1.326292518724980e-17

.subckt crys nA nB lm=0.190985740724534 cm=1.326289866142595e-17
rm nA nC 120
co nA nB 5e-12
lm1 nC nD lm
cm1 nD nB cm
.ends

.subckt JFET ng nd ns
GDsub1 ng ns1 cur = '1e-15*(exp((v(ng)-v(ns))/0.025)-1)'
GDsub2 ng nd1 cur = '1e-15*(exp((v(ng)-v(nd))/0.025)-1)'
rs ns1 ns 1
rd nd1 nd 1

Gids nd1 ns1 cur='
@ VTO = -1, @ SCALM = 1, @ M = 1, @ L = 1u, @ LAMBDA = 0.333, @ BETA = 0.1, @ W = 1u,
@ WDEL = 0, @ LDEL = 0, @ WDELeff = WDEL*SCALM, @ Weff = W*SCALM + WDELeff, @
LDELeff = LDEL*SCALM, @ Leff = L*SCALM + LDELeff, @ BETAeff = BETA*(Weff*M)/Leff, @
VGS = V(ng) - V(ns1), @ VGST = VGS - VTO, @ VDS = V(nd1) - V(ns1),
@ IF (VGST < 0)
@ {0.0}
```

```
@ ELSE
@ {
@ IF (VGST < VDS )
@ {
@ IF ( VGS ≤ 0 )
@ { BETAeff*VGST*VGST*(1+LAMBDA*(VDS-VGST))}
@ ELSE
@ {BETAeff*VGST*VGST*(1-LAMBDA*(VDS-VGST)*(1+LAMBDA*VGST/VTO))}
@ }
@ ELSE
@ {BETAeff*VDS*(2*VGST-VDS)*(1+LAMBDA*VDS)}}'

CgD1 ng nd1 c='
@ M =0.5, @VGD = V(ng)-v(nd1), @cgd=0.27p, @N=1, @vt=2.56929E-2, @PB=0.8, @AREA = Wef-
f/Leff, @AREAeff=M*AREA, @IS=1E-14, @ISeff=IS*AREAeff, @CGDeff = CGD*AREAeff,
@ IF (VGD < 0 )
@ CGDeff/POW((1-VGD/PB),M)
@ ELSE
@ CGDeff*(1+M*VGD/PB)'

CgS1 ng ns1 c ='
@ M = 0.5, @CGSeff = CGS*AREAeff,
@ IF (VGS < 0 )
@ {CGSeff/POW((1-VGS/PB),M)}
@ ELSE
@ {CGSeff*(1+M*VGS/PB)}'
.ends
.end
```

A.3 Level 3

```
Vdd1 ndd1 gnd 15
Vdd2 ndd2 gnd 15
Io n7 gnd 10m

Vlo1 nVlo1 gnd SIN( 0.125 1G 0)
Vlo2 nVlo2 gnd SIN( 0.125 1G 180)
Vrf1 nVrf1 gnd SIN( 0.0125 900E6 0)
Vrf2 nVrf2 gnd SIN( 0.0125 900E6 180)

rd1 ndd1 n3 1000
rd2 ndd2 n4 1000
ri nVout1 nVout2 1000
rkk1 n1 gnd 1G
rkk2 n2 gnd 1G
rkk3 nvout1 gnd 1G
rkk4 nvout2 gnd 1G
xT1 nVlo1 n5 n7 gnd JFET
xT2 nVlo2 n6 n7 gnd JFET
xT3 nVrf1 n3 n5 gnd JFET
xT4 nVrf2 n4 n5 gnd JFET
xT5 nVrf1 n4 n6 gnd JFET
xT6 nVrf2 n3 n6 gnd JFET

xa1 n1 n3 crys lm=0.190985740724534 cm=1.326289866142595e-17
xa2 nVout2 n1 crys lm=0.190985740724534 cm=1.326289866142595e-17
xa3 nVout1 n2 crys lm=0.190985740724534 cm=1.326289866142595e-17
xa4 n2 n4 crys lm=0.190985740724534 cm=1.326289866142595e-17
xb1 nVout1 n1 crys lm =0.190986122696397 cm= 1.326292518724980e-17
xb2 n1 n4 crys lm =0.190986122696397 cm= 1.326292518724980e-17
xb3 nVout2 n2 crys lm =0.190986122696397 cm= 1.326292518724980e-17
```

xb4 n2 n3 crys lm =0.190986122696397 cm= 1.326292518724980e-17

.subckt crys nA nB lm=0.190985740724534 cm=1.326289866142595e-17

rm nA nC 120

co nA nB 5e-12

lm1 nC nD lm

cm1 nD nB cm

.ends

.subckt JFET ng nd ns nb

GDsub1 ng ns1 cur = '1e-15*(exp((v(ng)-v(ns))/0.025)-1)'

GDsub2 ng nd1 cur = '1e-15*(exp((v(ng)-v(nd))/0.025)-1)'

rs ns1 ns 1

rd nd1 nd 1

Gids nd1 ns1 cur='

@ VTO = -1, @ SCALM = 1, @ M = 1, @ L = 1u, @ LAMBDA = 0.333, @ BETA = 0.1, @ W = 1u, @ WDEL = 0, @ LDEL = 0, @ WDELeff = WDEL*SCALM, @ Weff = W*SCALM + WDELeff, @ LDELeff = LDEL*SCALM, @ Leff = L*SCALM + LDELeff, @ BETAeff = BETA*(Weff*M)/Leff, @ vT = 26e-3, @ beteff = BETAeff,@ VGS = V(ng) - V(ns1), @ VGST = VGS - VTO, @ VDS = V(nd1) - V(ns1), @ Vsb = V(ns) - V(nb), @ ND = 1, @ ni = 1.45*1e10, @ qqq = 1.60212e-19, @ epo = 11.7*8.854e-14, @ Na = ni*ni/ND, @ Cox = 69.03e-8, @ num = qqq*epo*Na/(vT*log(Na/ni)), @ kk = pow(num,0.5)/(2*Cox), @ Ids0 = BETAeff * vT*vT*exp(1.8), @ VGEXP = 2, @ ALPHA = 2,

@ IDS = Ids0 * exp((VGS - VTO + 0.1*VDS -kk*Vsb)/(1.7*vT)) * (1 - EXP(-VDS/vT)), (IDS in this example is equal to 0)

@ IF (VGST < 0)

@ {ids}

@ ELSE

@ { beteff*(pow(vgst,VGEXP))*(1+LAMBDA*VDS)*tanh(ALPHA*VDS) + ids}'

CgD1 ng nd1 c='

@ M =0.5, @ VGD = V(ng)-v(nd1), @ VGS = V(ng) - V(ns1), @ VGST = VGS - VTO, @ VDS =

```
V(nd1) - V(ns1),
@ veff = 0.5*(VGS + VGD + POW((VDS*VDS + 1/(ALPHA*ALPHA)),0.5)),
@ vnew = 0.5*(veff + VTO + POW(((veff-VTO)*(veff-VTO) + 0.04),0.5)),
@cgs=0.27p, @cgd=0.27p, @N=1, @vt=2.56929E-2, @PB=0.8, @AREA = Weff/Leff, @AREAeff=M*AREA,
@IS=1E-14, @ISeff=IS*AREAeff, @CGDeff = CGD*AREAeff,
@ cgs/(4*POW((1 - vnew/PB),0.5)) * (1 + (veff - VTO)/POW(((veff-VTO)*(veff-VTO)+0.04),0.5)) *
@ (1 - VDS/POW((VDS*VDS + 1/(ALPHA*ALPHA)), 0.5)) + CGD/2*(1+VDS/POW((VDS*VDS
+ 1/(ALPHA*ALPHA)),0.5))'

CgS1 ng ns1 c ='
@ M = 0.5, @CGSeff = CGS*AREAeff,
@ cgs/(4*POW((1 - vnew/PB),0.5)) * (1 + (veff - VTO)/POW(((veff-VTO)*(veff-VTO)+0.04),0.5)) *
@ (1 + VDS/POW((VDS*VDS + 1/(ALPHA*ALPHA)), 0.5)) + CGD/2*(1-VDS/POW((VDS*VDS
+ 1/(ALPHA*ALPHA)),0.5))'
.ends
.end
```

Bibliography

[1] K. S. Kundert. *The designer's guide to SPICE and Spectre.* Kluwer Academic Publishers, 1995.

[2] E. Gad, M. Nakhla, R. Achar, and Y. Zhou. A-stable and L-stable high-order integration methods for solving stiff differential equations. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(9):1359–1372, Sept 2009.

[3] G. Wanner, E. Hairer, and S. P. Nørsett. Order stars and stability theorems. *BIT Numerical Mathematics*, 18(4):475–489, 1978.

[4] Y. Zhou, E. Gad, M. Nakhla, and R. Achar. Structural characterization and efficient implementation techniques for A-stable high-order integration methods. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(1):101–108, 2012.

[5] M.A. Farhan, E. Gad, M. Nakhla, and R. Achar. New method for fast transient simulation of large linear circuits using high-order stable methods. *IEEE Transactions on Components, Packaging and Manufacturing Technology*, 3(4):661–669, 2013.

[6] M.A. Farhan, E. Gad, M. Nakhla, and R. Achar. Fast simulation of microwave circuits with nonlinear terminations using high-order stable methods. *IEEE Transactions on Microwave Theory and Techniques*, 61(1):360–371, 2013.

[7] P. Maffezzoni, L. Codecasa, and D. D'Amore. Time-domain simulation of nonlinear circuits through implicit Runge–Kutta methods. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 54(2):391–400, 2007.

[8] P. Maffezzoni. A versatile time-domain approach to simulate oscillators in RF circuits. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 56(3):594–603, 2009.

[9] H. G. Brachtendorf and K. Bittner. Grid size adapted multistep methods for high Q oscillators. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(11):1682–1693, 2013.

[10] J. Dobes, A. Yadav, and D. Cerny. Efficient algorithm for solving systems of circuit differential-algebraic equations with reliable divergence suppression in DC and time domains. In *Circuits and Systems (MWSCAS), 2011 IEEE 54th International Midwest Symposium on*, pages 1–4, 2011.

[11] Y. Lin and E. Gad. Formulation of the Obreshkov-based transient circuit simulator in the presence of nonlinear memory elements. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(1):86–94, 2015.

[12] G. A. Baker and P. Graves-Morris. Padé approximants. 1996. *Encyclopedia of Mathematics and its Applications*, 1996.

[13] U. M. Ascher and L. R. Petzold. *Computer methods for ordinary differential equations and differential-algebraic equations*, volume 61. Siam, 1998.

[14] C. Gear. *Numerical initial value problems in ordinary differential equations*, volume 59. Prentice-Hall Englewood Cliffs, NJ, 1971.

[15] C. Gear. Simultaneous numerical solution of differential-algebraic equations. *Circuit Theory, IEEE Transactions on*, 18(1):89–95, 1971.

[16] Chung-Wen Ho, Albert E. Ruehli, and Pierce A. Brennan. The modified nodal approach to network analysis. *Circuits and Systems, IEEE Transactions on*, 22(6):504–509, Jun 1975.

[17] J. D. Lambert. Computational methods in ordinary differential equations. 1973.

[18] J. C. Butcher. *Numerical methods for ordinary differential equations*. John Wiley, 2008.

[19] N. Obreshkov. Sur le quadrature mecaniques. *Spisanie Bulgar. Akad. Nauk.(Journal of the Bulgarian Academy of Sciences)*, 65:191–289, 1942.

[20] 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*, pages 186–187, 2007.

[21] T. L. Quarles. *The SPICE3 implementation guide*. Electronics Research Laboratory, College of Engineering, University of California, 1989.

[22] G. F. 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*. Springer, 1997.

[23] P. Favati, G. Lotti, and O. Menchi. Non-recursive solution of sparse block Hessenberg systems. *Numerical Linear Algebra with Applications*, 11(4):391–409, 2004.

[24] G. W. Stewart. On the solution of block Hessenberg systems. *Numerical linear algebra with applications*, 2(3):287–296, 1995.

[25] J. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM Journal on Scientific and Statistical Computing*, 9(5):862–874, 1988.

[26] T. A. Davis and E. P. Natarajan. Algorithm 8xx: KLU, a direct sparse solver for circuit simulation problems. *ACM Trans. Math. Softw*, 5:1–17, 2011.

[27] I. C. Ipsen and D. J. Lee. Determinant approximations. *Numeric. Linear Algebra Applicat*, 2005.

[28] E. Gad, R. Khazaka, M. Nakhla, and R. Griffith. A circuit reduction technique for finding the steady-state solution of nonlinear circuits. *Microwave Theory and Techniques, IEEE Transactions on*, 48(12):2389–2396, 2000.

[29] U. Günther, M.and Feldmann. The DAE-index in electric circuit simulation. *Mathematics and Computers in Simulation*, 39(5):573–582, 1995.

[30] D. Weste, N.and Harris. *CMOS VLSI design: a circuits and systems perspective.* Addison-Wesley Publishing Company, 2010.

[31] E. Gad and M. Nakhla. Model reduction for DC solution of large nonlinear circuits. In *Computer-Aided Design, 1999. Digest of Technical Papers. 1999 IEEE/ACM International Conference on*, pages 376–379, 1999.

[32] J. Seward, N. Nethercote, and J. Weidendorfer. *Valgrind 3.3-Advanced Debugging and Profiling for GNU/Linux applications.* Network Theory Ltd., 2008.

[33] Charles F. Van Loan. The ubiquitous Kronecker product. *Journal of Computational and Applied Mathematics*, 123(1):85–100, 2000.

[34] T. A. Davis and E. Palamadai Natarajan. Algorithm 907: KLU, a direct sparse solver for circuit simulation problems. *ACM Transactions on Mathematical Software (TOMS)*, 37(3):36, 2010.

[35] J. Vlach and K. Singhal. *Computer methods for circuit analysis and design*. Springer, 1983.

[36] T. A. Davis and E. P. Natarajan. User Guide for KLU and BTF. Dept. of Computer and Information Science and Engineering, University of Florida, 2009.

[37] K. S. Kundert and A. Sangiovanni-Vincentelli. Simulation of nonlinear circuits in the frequency domain. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 5(4):521–535, 1986.

[38] C. R. Paul. *Analysis of multiconductor transmission lines*. John Wiley, 2008.

[39] G. Wanner and E. Hairer. Solving ordinary differential equations II, 1991.

[40] R. J. Gilmore and M. B. Steer. Nonlinear circuit analysis using the method of harmonic balance. *International Journal of Microwave and Millimeter-Wave Computer-Aided Engineering*, 1(1):22–37, 1991.

[41] M. Nakhla and J. Vlach. A piecewise harmonic balance technique for determination of periodic response of nonlinear systems. *Circuits and Systems, IEEE Transactions on*, 23(2):85–91, 1976.