Abstract

VIJAYCHAND , SHUBHA,. Time Step Control in Transient Analysis. (Under the direction of Michael B. Steer)

The time-step control algorithm has a dramatic impact on the accuracy and simulation time in transient circuit simulation. A new time-step control algorithm is presented based on a novel estimation of the truncation error. The new truncation error estimation uses difference between Backward Euler and Trapezoidal numerical integration techniques. The results of this technique are compared with The traditional **SPICE**-like approach implemented in fREEDA. Results for the solution of a Soliton line and Mesfet circuit are presented.

Time Step Control in Transient Analysis

by

SHUBHA VIJAYCHAND

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

ELECTRICAL ENGINEERING

Raleigh

2002

APPROVED BY:

Chair of Advisory Committee

Biographical Summary

Shubha Vijaychand was born on 7th July, 1978 in Bangalore, India. She received a degree in Electronics and Instrumentation engineering in 2000 from the Sri Jayachamarajendra College of Engineering in Mysore, India. While pursuing the B.S. degree she worked as a co-op for National Aeronautics Laboratory, Bangalore, India in the summer of 1998. She was admitted to the Master's program at North Carolina State University in the Spring of 2001. Her interests are in the fields of analog , RF circuit design and computer-aided analysis of circuits.

Acknowledgments

I would like to express my sincere gratitude to my advisor, Dr. Michael Steer for giving me an opportunity to work with his research group. It has provided me with a great chance to explore and learn, for which I am grateful. I would also like to thank Dr. Griff Bilbro and Dr. Gianluca Lazzi for serving on my thesis committee. I would also like to thank Dr. Carlos Christofferson, whose knowledge and patience have been instrumental in my work. I would also like to thank parents for all their love, support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The accuracy and speed of a simulation is dependant on the accuracy of the models, the error injected by the simulator algorithms and the circuit itself. If the models do not accurately reflect the physical effects that are important to the circuit the answer will be wrong. Similarly, if the model does not match the physical device because the model parameters are badly chosen, again, the answer will be in error. However, even if the model does a good job of reproducing the important characteristic of the device, the simulator itself can inject error into this solution.

These errors, if not well controlled can result in significant error. The circuit itself can either magnify errors made by the simulator and the models. Conversely, it can be very tolerant of such errors. Circuits that can be very sensitive to simulator errors include oscillators and charge storage circuits such as switched-capacitor circuits.

The earlier implementation of transient analysis in the object-oriented circuit simulator fREEDA did not have error control or test for convergence. The

aim of the project reported here was implementation of an error control technique in fREEDA.

Two types of error control techniques were implemented and the results compared. The new time step algorithm developed was seen to be faster with high accuracy as compared to the other transient algorithms.

fREEDA is a circuit simulator that has the capability to support many types of elements, like electrical, electromagnetic and thermal. It supports various types of analyses such as dc, transient and harmonic balance. The greatest difference between fREEDA and other modern commercial simulators is that it uses an object-oriented approach for analyzing circuits. Elements, be they electrical thermal or electromagnetic, can be considered as objects and all these elements are linked or connected to each other at nodes and by edges. This structure maps smoothly on to the concept of classes. Hence the concepts of OO (Object-oriented) programming maps cleanly onto circuit simulation. This makes the process of transient analysis for fREEDA relatively straightforward as the elements and sources need not be polled for breakpoints. The value of the derivatives are also stored which eliminates the need to recalculate when a solution fails.

1.2 Thesis Overview

Chapter 2 deals with local truncation error and time step control in **SPICE**. It also gives a brief overview about **fREEDA**, the circuit simulator in the analysis types have been implemented and some of its features that simplifies the

analysis routine.

Chapter 3 deals with the transient analysis algorithms. The **SPICE** like algorithm and the new time step algorithm are explained in detail accompanied by the formulae

Chapter 4 presents some results of testing the analysis types using the Non-linear Transmission Line(NLTL) and a mesfet circuit. A comparison between the analysis types, their merits and demerits has been discussed.

Chapter 5 presents the conclusions and the future research in this area.

Chapter 2

Literature Review

2.1 Introduction

The accuracy of a simulation is dependant on three factors, the accuracy of the models, the error injected by the simulator algorithms and the circuit itself. Errors due to the models are present when the models do not accurately reflect the physical effects that are important to the circuit behaviour. The choice of model parameters also have an important effect on the accuracy of the solution. These errors need to be minimised.

2.2 Time Step Algorithm in SPICE

2.2.1 Introduction

SPICE has become the standard computer program for electrical circuit simulation. Its practical use is however limited by nonconvergence failures, time step control errors and numerical integration failures. Because of these failures results are either difficult to obtain or contain inaccuracies.

2.2.2 Description and Formulae

Transient analysis in **SPICE** computes the time response of a circuit for any arbitrary input signal. Using numerical integration methods it solves the circuit equations at time intervals whose size depends on the circuit activity, i.e. on the rate of transitions of voltages or currents. This is known as dynamic time step control and ensures accuracy and convergence for circuits with large and rapid voltages and current transitions. During times of low circuit activity the time step will be increased to reduce simulation time.

Sinusoidal signals are characterized by a permanent change of the derivative dv/dt or di/dt. This forces **SPICE** to continously change the time step. Failure mechanisms within the time step control algorithm may therefore introduce significant error during transient analysis of sinusoidal sigals

2.3 Error Control Techniques

2.3.1 Introduction

The most wide spread method of nonlinear circuit analysis is time-domain analysis using numerical integration to determine the circuit response at one instance of time given the circuit's response at a previous instance of time. This section reviews some of the numerical integration methods.

Consider the following diffrential equation

$$x' = f(x,t) \tag{2.1}$$

where x is the unknown function, t is the time and $f(x,t)$ is a given function. In the case of fREEDA $f(x,t)$ is the state variable at a given instant of time say

t_0, x is the value of the state variable at $t_1 = t_0 + t$, where t is the time step. The different integration methods predict different values of x. Backward Euler tends to overdamp the solution where as Trapezoidal tends to underdamp.

The task of an error correcting algorithm is to minimize the error by finding an optimum time step.

The equivalent integral function of (2.1) is

$$x(t_1) = x(t_0) + \int_{t_0}^{t_1} f(x,t)dt \tag{2.2}$$

where t_0 and t_1 are two time points as defined above. The difference between the two gives the time step which is very small and the integral equation can be discretized as

$$x(t_1) \approx x(t_0) + x'(t_1 - t_0) \tag{2.3}$$

$$x_1 \approx x(t_1) \tag{2.4}$$

and $x_0 = x(t_0)$. Which gives

$$x_1 = x_0 + hx^{'} \tag{2.5}$$

where h=$t_1 - t_0$ is the size of the timestep. Therefore the generic expression is

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

This indicates that the future value can be computed based on the current

value. The different integration methods differ in the method used to estimate $x^{'}$. The generic integration formula used by the different integration formulae can be reduced to

$$x'_n = ax_n + b_{n-1} \tag{2.7}$$

where a is a constant and b_{n-1} depends on the previous values of x.

2.3.2 Backward Euler

This is the simplest implicit method. From Equation (2.7)

and setting $x^{'} = x^{'}_n$ in

$$x_n = x_{n-1} + hx^{'}_n, \tag{2.8}$$

the coefficients of the generic integration formula are

$$a = \frac{1}{h} \tag{2.9}$$

$$b_{n-1} = \frac{-1}{h} x_{n-1} \tag{2.10}$$

It is a first order differential method as the value of the state variable at any instant depends only on the value of the state variable at the previous instant.

2.3.3 Trapezoidal

From Equation (2.7) and setting

$$x^{'} = (x^{'} + x^{'}_{n-1})/2, \tag{2.11}$$

The discretized numerical integration becomes

$$x_n = x_{n-1} + h/2(x^{'}_{n-1} + x^{'}_{n}) \tag{2.12}$$

Now the coefficients of the generic integration formulae are

$$a = \frac{2}{h} \tag{2.13}$$

$$b_{n-1} = -\frac{2}{h}x_{n-1} - x^{'}_{n-1} \tag{2.14}$$

This a second order differential method as the value of the state variable at any instant depends on the value of the state variables at previous two instances.

2.3.4 Gear

The Gear integration method is different from the traditional integration methods. It estimates the value of the function at time t_{n+1} with the information from the present solution point and past solution points, The previous solution points that are used determines the order of the Gear method. The formula for the Gear-2-method is

$$x_{n+1} = \frac{4}{3}x_n - \frac{1}{3}x_{n-1} + \frac{2h}{3}x^{'}_{n+1} \tag{2.15}$$

2.3.5 Stability and Accuracy

Due to the difference in the integration formulas, each method will produce a different result when used to discretize a given function. The performance of a

method is determined by its *accuracy* and *stability*. Since the numerical integration solution is only an approximation to the exact solution, a finite amount of error may be introduced at each time point. This error is known as the local truncation error **(LTE)**. How the LTE accumulates over a large number of time points is a measure of the stability of an integration method. If a method is unstable it will diverge from the exact solution over a large number of timepoints.

The accuracy and stability of an integration method depends on the function it is applied to. The accuracy is also determined by the time step used. Decreasing the step size of any integration method improves the accuracy of the solution. It forces the simulator to solve more points which consequently results in longer simulation time. Decreasing the time step also increases the chance of stepping into or close to a model discontinuity and failing to converge.

The Gear integration method tends to be the most stable because of its averaging formula. It however yields extremely poor estimates of future solution values on highly nonlinear circuits. This averaging formula is the reason for a failure mechanism called *Gear Overshoot* ([1]). For switching waveforms and piecewise linear waveforms, the Gear method may overshoot the correct value.

Trapezoidal integration suffers from a failure mechanism called *trapezoidal overshoot*. Trapezoidal oscillation occurs when the integration step size is too large to follow the curvature of a given function. The result is a predicted solution that appears to oscillate around the correct solution from one time point solution to the next.

All integration methods suffer from a failure mechanism called *accumulated error*. It usually occurs in periodic circuits and long transient simulations. If the overestimate/underestimate errors do not cancel each other out, the accu-

mulated error tends to increase with each new time point.

2.4 Predictors

The Backward Euler and Trapezoidal discretization formulae use the derivative at the next point. But at the beginning of the analysis this value is not known and cannot be reasonably extimated. Therefore in order to start the calculations an approximate value must be computed. This is done in various ways, the simplest being the result of the previous step. This is used in the implementation.

Another possibility for initialising analysis at the next time step is to use Forward Euler formula.

The formula is

$$x'(t_0) = \frac{x_1 - x_0}{h} \tag{2.16}$$

or

$$x_1 = x_0 + hx_0^{'} \tag{2.17}$$

Once the predicted value is inserted into the corrector (say Backward Euler or Trapezoidal), iteration is performed to correct the mismatch. This iteration is usually performed using Newton's iteration.

Predictors are however not absolutely necessary, a better prediction will result in fewer iterations. Since predictors are very simple to use , their application is highly desirable. They also help in estimating the errors committed and in

controlling the step size h.

2.5 Local Truncation Error

The LTE-Timestep control algorithm is the default time step control algorithm in most **SPICE** versions. The LTE-algorithm uses a formula that predicts the magnitude of the error computed in the numerical integration calculations of the previous time point. The time step is therefore adjusted by evaluating the error generated in the numerical integration routine.

Using the SPICE parameters RELTOL and ABSTOL an upper bound E for the LTE can be defined as

$$\mathrm{E} = \mathrm{RELTOL.max}(|\mathrm{x}^{'}_{\mathrm{n+1}}|, |\mathrm{x}^{'}_{\mathrm{n}}|) + \mathrm{ABSTOL} \tag{2.18}$$

where $\mathrm{x}^{'}_{n+1}$ and $\mathrm{x}^{'}_{n}$ represent the current of capacitors or the voltage across inductors. RELTOL is the relative error tolerance within which voltages and device currents are required to converge. ABSTOL is called the absolute current tolerance, it represents the smallest current that can be monitored. These are user defined parameters and determine how accurately SPICE calculates the solution. ABSTOL and RELTOL can have direct impact on convergence and simulation time and have to be chosen carefully.

An important measure of the accuracy of a numerical integration method is the local truncation error, LTE, evaluated at each time point t_n. The local truncation error at t_n is an estimate of the difference between the value computed by the simulator and the exact solution of the function at this time point. When the trapezoidal method is used

$$\mathrm{LTE_{ESTIM}} = \left| \frac{\mathrm{TRTOL}}{12} \mathrm{h^3 x_n'''} \right| \tag{2.19}$$

where x_n''' is the third derivative of the state variable x computed at the time point t_n, h is the time step used and TRTOL is a user defined constant which underestimates the $\mathrm{LTE_{ESTIM}}$ value. The third derivative is approximated by divided difference formulae.

If x is the charge of a capacitor the estimated LTE for the capacitor current (x') is given by

$$\mathrm{LTE_{ESTIM}} = \left| \frac{\mathrm{TRTOL}}{6} \mathrm{h^2 x_n'''} \right| \tag{2.20}$$

Equation 2.19 is used in the LTE-time step control algorithm to calculate an upper limit for LTE. A weighted summation of the LTE's at all previous time points yields a global error "e".

A recursive form of the global error "e" at time point t_{n+1} can be written as

$$e(t_{n+1}) = a.e(t_n) + LTE_{n+1} \tag{2.21}$$

where the constant a is the amplification factor of the integration method used. ($a < 1$, otherwise the global error increases over the simulation time (accumulated error) and the system is unstable). The magnitude of the LTE is determined by the time step h. As mentioned in the earlier section the accuracy of the numerical integration generally improves when the time step is reduced, but when simulating sinusoidal circuits like oscillators the trapezoidal method introduces a frequency error $\triangle f$ in addition to the amplitude error LTE.

2.6 fREEDA

2.6.1 Introduction

The rapid rate of innovation of microwave and millimeter wave systems requires the development of an easily extensible and modifiable computer aided engineering (CAE) environment. While great strides have been made in the flexibility of commercial CAE tools, these sometimes prove inadequate in modeling advanced systems. As with virtually all aspects of electronic engineering the abstraction level of RF and microwave theory and techniques has increased dramatically. In particular, large systems are being designed with attention given to the interaction of components at many levels. One of the most significant developments relevant to computer aided engineering is the rise of object oriented (OO) design practice [10]. While it is normal to think of OO-specific programming languages as being the main technology for implementing OO design, good OO practice can be implemented in more conventional programming languages such as C. However OO-specific languages foster code reuse and have constructs that facilitate object manipulation. The OO abstraction is well suited to modeling electronic systems, for example, circuit elements are already viewed as discrete objects and at the same time as an integral part of a (circuit) continuum. The OO view is a unifying concept that maps extremely well onto the way humans perceive the world around them.

Non-OO circuit simulators always become complicated with many layers of special cases. Referring to circuit elements again, traditional simulation implementations have many if-then like statements and individually identify every element in many places for special handling. An integral part of the various high performance computing initiatives is the separation of the core components em-

bodying numerical methods from the modeling and solver formulation process with the result that numerical techniques developed by computer scientists and mathematicians can be formulated using formal correctness procedures. The circuit abstraction is adapted so that highly reliable and efficient pre-developed libraries can be used. C++ was once considered slow for scientific applications. Advances in compilers and programming techniques, however, have made this language attractive and in some benchmarks C++ outperforms Fortran. Several OO numerical libraries have been developed . Of great importance to the work described here is the incorporation of the standard template library (STL). The STL is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. The STL is a generic library, meaning that its components are heavily parameterized: almost every component in the STL is a template. The current ISO/ANSI C++ standard has not been fully implemented and C++ compilers support a variable subset of the standard. The biggest areas of noncompliance being the templates and the standard library. The goal in design was to obtain speed in development, to use off the shelf advanced numerical techniques, and to allow easy expansion and testing of new models and numerical methods. The circuit simulator implementing these ideas is fREEDA. fREEDA is the first circuit simulator to use recent OO techniques. The design intent was to to combine the advantages of previous OO circuit simulators with these new developments as well as expanding capability. fREEDA uses C++ libraries and some written in C or Fortran.

2.6.2 Support Libraries

Solution of Sparse Linear Systems

Sparse 1.3 2 is a flexible package of subroutines written in C used to numerically solve large sparse systems of linear equations. The package is able to handle arbitrary real and complex square matrix equations. Besides being able to solve linear systems, it is also able to quickly solve transposed systems, find determinants, and estimate errors due to ill-conditioning in the system of equations and instability in the computations. Sparse also provides a test program that is able to read matrix equation from a file, solve it, and print useful information (such as condition number of the matrix) about the equation and its solution. Sparse was originally written for use in circuit simulators and is well adapted to handling nodal- and modified-nodal admittance matrices.

SuperLU is used in the wavelet and time marching transient analyses. It contains a set of subroutines to numerically solve a sparse linear system $\mathbf{Ax} = \mathbf{b}$. It uses Gaussian elimination with partial pivoting (GEPP). The columns of $\mathbf{A}$ may be pre-ordered before factorization; the pre-ordering for sparsity is completely separate from the factorization. SuperLU is implemented in ANSI C. It provides support for both real and complex matrices, in both single and double precision.

Vectors and Matrices

Most of the vector and matrix handling in fREEDA uses MV++4. This is a small set of vector and simple matrix classes for numerical computing written in C++. It is not intended as a general vector container class but rather designed specifically for optimized numerical computations on RISC and pipelined

architectures which are used in most new computer architectures. The various MV++ classes form the building blocks of larger user-level libraries. The MV++ package includes interfaces to the computational kernels of the Basic Linear Algebra Subprograms package (BLAS) which includes scalar updates, vector sums, and dot products. The idea is to utilize vendor-supplied, or optimized BLAS routines that are fine-tuned for particular platforms.

The Matrixtemplate Library (MTL5) is a high-performance generic component library that provides comprehensive linear algebra functionality for a wide variety of matrix formats. It is used in the wavelet and time marching transient analyses. As with the STL, MTL uses a five-fold approach, consisting of generic functions, containers, iterators, adaptors, and function objects, all developed specifically for high performance numerical linear algebra. Within this framework, MTL provides generic algorithms corresponding to the mathematical operations that define linear algebra. Similarly, the containers, adaptors, and iterators are used to represent and to manipulate matrices and vectors.

Solution of Non-Linear systems

Nonlinear systems of equations in fREEDA are solved using the NNES6 library. This package is written in Fortran and provides Newton and quasi- Newton methods with many options including the use of analytic Jacobian or forward, backwards or central differences to approximate it, different quasi- Newton Jacobian updates, or two globally convergent methods, etc. This library is used through an interface class (NLSInterface), so it is possible to install a different routine to solve nonlinear systems if desired by just replacing the interface (four different nonlinear solvers have already been used). The Fortran routine NLEQ1

(Numerical solution of nonlinear (NL) equations (EQ)7) can also be used as a compile option.

Fourier Transform

Fourier transformation is implemented in fREEDA using the FFTW8 library. FFTW is a C subroutine library for computing the Discrete Fourier Transform (DFT) in one or more dimensions, of both real and complex data, and of arbitrary input size. Benchmarks, performed on a variety of platforms show that FFTW's performance is typically superior to that of other publicly available FFT software. Moreover, FFTW's performance is portable: the program performs well on most computer architectures without modification.

Automatic Differentiation

Most nonlinear computations require the evaluation of first and higher derivatives of vector functions with m components in n real or complex variables. Often these functions are defined by sequential evaluation procedures involving many intermediate variables. By eliminating the intermediate variables symbolically, it is theoretically always possible to express the m dependent variables directly in terms of the n independent variables. Typically, however, the attempt results in unwieldy algebraic formulae, if it can be completed at all. Symbolic differentiation of the resulting formulae will usually exacerbate this problem of expression swell and often entails the repeated evaluation of common expressions.

An obvious way to avoid such redundant calculations is to apply an optimizing compiler to the source code that can be generated from the sym-

bolic representation of the derivatives in question. Given a code for a function $F : \Re^n \rightarrow \Re^m$, automatic differentiation (AD) uses the chain rule successively to compute the derivative matrix. AD has two basic modes, forward mode and reverse mode. The difference between these two is the way the chain rule is used to propagate the derivatives.

A versatile implementation of the AD technique is Adol-C 9, a software package written in C and C++. The numerical values of derivative vectors (required to fill a Jacobian for solving non-linear elements using Newton's method) are obtained free of truncation errors at a small multiple of the run time required to evaluate the original function with little additional memory required. It is important to note that AD is not numerical differentiation and the same accuracy achieved by evaluating analytically developed derivatives is obtained. The eval() method of the nonlinear element class is executed at initialization time and so the operations to calculate the currents and voltages of each element are recorded by Adol-C in a tape which is actually an internal buffer. After that, each time that the values or the derivatives of the nonlinear elements are required, an Adol-C function is called and the values are calculated using the tapes. This implementation is efficient because the taping process is done only once (this almost doubles the speed of the calculation compared to the case where the functions are taped each time they are needed). When the Jacobian is needed, the corresponding Adol-C function is called using the same tape. In the case of Harmonic Balance simulations, the program has been tested with large circuits with many tones, and the function or Jacobian evaluation times are always very small compared with the time required to solve the matrix equation (typically some form of Newton's method) that uses the Jacobian. The

conclusion is that there is little detriment to the performance of the program introduced by using automatic differentiation. However the advantage in terms of rapid model development is significant. The majority of the development time in implementing models in simulators, is in the manual development of the derivative equations. Unfortunately the determination of derivatives using numerical differences is not sufficiently accurate for any but the simplest circuits and in any event, is computationally intensive. With Adol-C full 'analytic' accuracy is obtained and the **implementation of new analysis is dramatically simplified**. From experience the average time to develop and implement a transistor model is an order of magnitude less than deriving and coding the derivatives manually. Note that time differentiation, time delay and transformations are left outside the automatic differentiation block. The calculation speed achieved is approximately ten times faster than the speed achieved by including time differentiation, time delay and transformations inside the block.

Chapter 3

Time Step Control in fREEDA

3.1 Fixed Time Step Algorithm

3.1.1 Introduction

The fixed time-step marching Algorithm in fREEDA is state-variable based and is considerably faster than the convolution based transient analysis scheme. The time-marching techniques are more efficient than the conventional SPICE analysis as will be shown in the next section. This technique was implemented by Dr Carlos E. Christofferson in fREEDA.

As shown in [9] the speed of the convolution based transient approach limits the use of this technique for nearly all transient circuit simulations. The simulation of a Non Linear Transmission Line takes about 5 hours of computer time using convolution based analysis as compared to 10 minutes using the state-variable based transient analysis.

3.1.2 Description and Formula

The formulation of the system equations begins with partitioning the network into linear and nonlinear subnetworks. The nonlinear elements are replaced by variable voltage or current sources. For each nonlinear element one terminal is taken as the reference and the element is replaced by a set of sources connected to the reference terminal. Both voltage and current sources are valid replacements for the nonlinear elements but current sources are more convenient because they yield a smaller *modified nodal admittance matrix* (MNAM). The basic idea is to convert the differenatial equations in an algebraic system of nonlinear equations using time marching integration methods.

The MNAM formulation of a linear subcircuit begins with the definition of two matrices $\mathbf{G}$ and $\mathbf{C}$ of equal size n_m. Where n_m is the number of non reference nodes in the circuit. A vector $\mathbf{s}$ of size n_m is also defined. The contributions of the fixed sources and the non linear elements which are dependant on the time are entered in this vector. All conductors and frequency independant MNAM stamps arising in the formulation are entered in $\mathbf{G}$. The values of capacitors and inductors and other values that are associated with dynamic elements are stored in $\mathbf{C}$. The linear system obtained is

$$\mathbf{Gu(t)} + \mathbf{C}\frac{d\mathbf{u}}{dt} = \mathbf{s}(t) \tag{3.1}$$

where $\mathbf{u}$ is the vector of nodal voltages and required currents. The vector $\mathbf{s}$ consists of an independant component $\mathbf{s}_f$ and a component $\mathbf{s}_v$ that depends on the state variable.

$$\mathbf{s}(t) = \mathbf{s}_f(t) + \mathbf{s}_v(t) \tag{3.2}$$

The vector $\mathbf{s}_f$ vector is due to independant sources in the circuit, where as $\mathbf{s}_v$ is the contribution of the currents injected into the circuit by the nonlinear network.

The non linear subcircuit is represented as

$$\mathbf{v}_{NL}(\mathbf{x}_n) = u[\mathbf{x}_n, \mathbf{x}_n^{'}, ..., \mathbf{x}_n^{(m)}, \mathbf{x}_{D,n}] \tag{3.3}$$

$$\mathbf{i}_{NL}(\mathbf{x}_n) = w[\mathbf{x}_n, \mathbf{x}_n^{'}, ..., \mathbf{x}_n^{(m)}, \mathbf{x}_{D,n}] \tag{3.4}$$

where $\mathbf{x}_n = \mathbf{x}(t_n), \mathbf{x}_n^{'} = \mathbf{x}^{'}(t_n), (\mathbf{x}_{D,n})_i$, and t_n is the current time.

If $\mathbf{T}$ is the incident matrix whose size is determined by the number of state variables, then by discretizing the above

$$\mathbf{G}\mathbf{u}_n + \mathbf{C}\mathbf{u}_n^{'} = \mathbf{s}_{f,n} + \mathbf{T}^T\mathbf{i}_{NL}(\mathbf{x}_n) \tag{3.5}$$

The approximation of the time marching integration is applied by using general integration representation of a variable for $\mathbf{u}$.

$$\mathbf{u}_n^{'} = a\mathbf{u}_n + \mathbf{b}_{n-1} \tag{3.6}$$

where, $\mathbf{b}_{n-1}$ is a vector of the same dimension as $\mathbf{u}_n$. Substituting the value

of $\mathbf{u}_n^{'}$ in the above equation (3.5),

$$\mathbf{G}\mathbf{u}_n + \mathbf{C}[a\mathbf{u}_n + \mathbf{b}_{n-1}] = \mathbf{s}_{f,n} + \mathbf{T}^T\mathbf{i}_{NL}(x_n) \tag{3.7}$$

Solving for the nodal voltages

$$\mathbf{u}_n = [\mathbf{G} + a\mathbf{C}]^{-1}[\mathbf{s}_{f,n} - \mathbf{T}^T\mathbf{i}_{NL}(\mathbf{x}_n)] \tag{3.8}$$

The error function $\mathbf{f}(t) = \mathbf{v}_L(t) - \mathbf{v}_{NL}(t) = 0$

$\mathbf{v}_L$(t) = $\mathbf{Tu}$(t), then

therefore $\mathbf{f}$(t)=$\mathbf{Tu}$(t)-$\mathbf{v}_{NL}$(t)

Discretizing the above equation and replacing the value of the nodal voltages $\mathbf{u}_n$, we have the following

$$\mathbf{f}(\mathbf{x}_n) = \mathbf{s}_{sv,n} + \mathbf{M}_{sv}\mathbf{i}_{NL}(\mathbf{x}_n) - \mathbf{v}_{NL}(\mathbf{x}_n) = 0 \tag{3.9}$$

where $\mathbf{s}_{sv,n}$ and $\mathbf{M}_{sv}$ are defined as

$$\mathbf{s}_{sv,n} = \mathbf{T}[\mathbf{G} + a\mathbf{C}]_{-1}[\mathbf{s}_{f,n} - \mathbf{C}\mathbf{b}_{n-1}] \tag{3.10}$$

$$\mathbf{M}_{sv} = \mathbf{T}[\mathbf{G} + a\mathbf{C}]^{-1}\mathbf{T}^T \tag{3.11}$$

The error function is however not minimized to arrive at an optimal solution.

3.1.3 Merits and Demerits

The major disadvantage of this technique is the absence of any error reduction technique. The analysis computes the values of the matrices at various time points as specified by the netlist. The technique informs the user when the solution has failed to converge but does not attempt to correct it. The method might not be accurate and depends heavily on the time-step specified in the netlist. The accuracy of this technique is user dependant and relies on the user to identify an incorrect solution and make modifications in the netlist to rectify the error.

The main advantage of this method is the speed of the solution. An approximate solution can be found before doing a detailed analysis which has higher accuracy.

3.2 SPICE-like Time Step Algorithm

3.2.1 Introduction

The **SPICE**-like time-step algorithm was implemented here for state-variable based transient analysis. An error predictor corrector method is incorporated in the analysis. The method is called the **SPICE**-like time step algorithm because the error predictor and corrector algorithm is as used in Berkely **SPICE3**.

3.2.2 Description and Formulae

The analysis uses the same formula as does the fixed time step algorithm. Error check and convergence check routines is incorporated and the matrices are recalculated when the time step is changed. The algorithm uses straight line

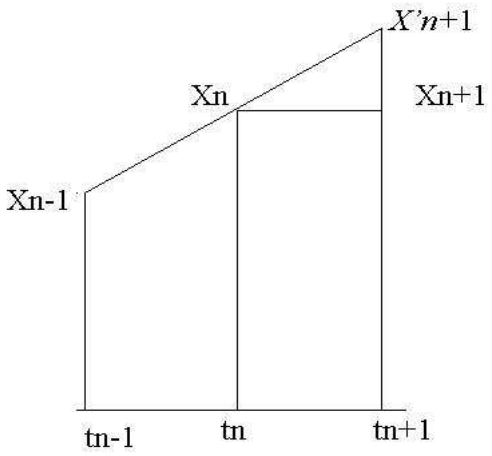

Figure 3.1: SPICE like linear interpolation

extrapolation to calculate the value of the state variable at the next time step.

From Figure 3.1, the difference between the value of state variable at t_{n+1} and the derivative at t_n is the error.

Convergence is achieved in DC and Transient solutions when

1. The nonlinear branch currents converge to within a tolerance of 0.1 percent or 1 picoamp (1.0E-12 Amp), whichever is larger

2. The node voltages converge to within a tolerance of 0.1 percent or 1 microvolt (1.0E-6 Volt), whichever is larger.

The **SPICE**-like algorithm checks for convergence. The output of the nonlinear solver interface has to be less than 1 for convergence. A combination of relative tolerance and absolute tolerance gives the error limit. The values of

M_{sv} matrix, non-linear voltages and currents have to be recalculated every time the time step changes. These calculation do not take too much computation time as the derivatives have already been calculated by the **ADOL-C** routine.

3.2.3 Merits and Demerits

The algorithm implementing the **SPICE**-like approach checks for error and non-convergence and varies the time step and attempts to correct the same to keep the truncation error within tolerance. The results using this method have a high accuracy.

The main disadvantage of this method is its inability to follow curves. The algorithm takes more time steps while following curves as the time step is reduced, as the straight line interpolation does not make an accurate guess for the next time step.

3.2.4 Comments

The algorithm deviates in minor respects from the original **SPICE3**. The original **SPICE3** algorithm cuts the time step to 1/8th of its previous value when the error is more than a combination of the relative error and the absolute error. The algorithm used in **fREEDA** cuts the time step by 1/2 when error requirements are not met. The algorithm used here also does not check for DC convergence. **f** uses state-variables to solve the matrices. The state-variables are set by the model and can be either voltage, current or charge. Hence in the current implementation a DC solution cannot be found prior to transient analysis.

3.3 New Time Step Algorithm

3.3.1 Introduction

The new time step algorithm to uses the difference in the values of backward euler and trapezoidal derivatives to predict the value of the state variables at the next time point. This algorithm makes a better prediction than the conventional **SPICE** like approach and therefore is able to follow curves with less time points.

3.3.2 Description and Formulae

The New Time Step Algorithm uses the value of the difference between derivatives to predict the value of the state variable at the next time point. The two different integration techniques give results which are close when the algorithm approaches the solution. The figure illustrates this fact.

As in the **SPICE** like algorithm the values of the M_{sv} vector and the values of the non linear voltages and currents have to be recalculated when the time step changes. As in the previous case these calculations do not take too much computation as the derivatives have already been calculated by the **ADOL-C** routine.

3.3.3 Merits and Demerits

The Algorithm checks for error and non-convergence and varies the time step and attempts to correct the solution when the criteria are not met. The results using this method have a high accuracy. The elements and voltage sources need not be polled to check for break points. This considerably increases the speed of simulation.

The main disadvantage of this method is problems faced while following

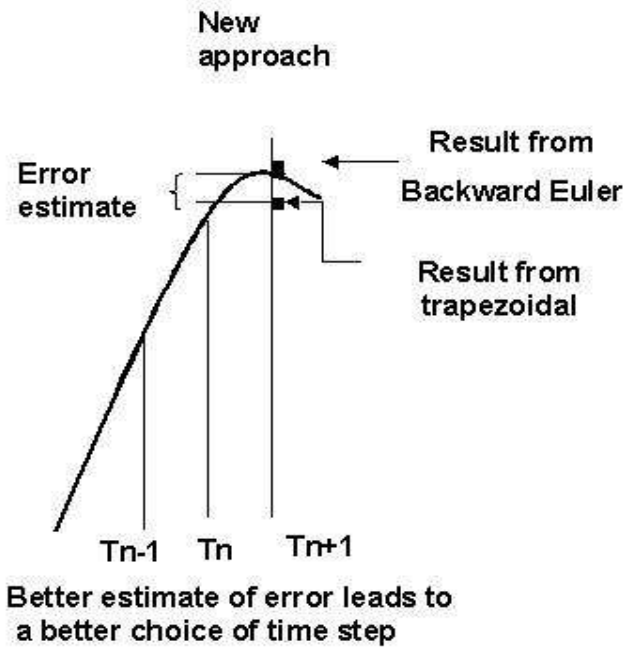

Figure 3.2: New Time Step Algorithm

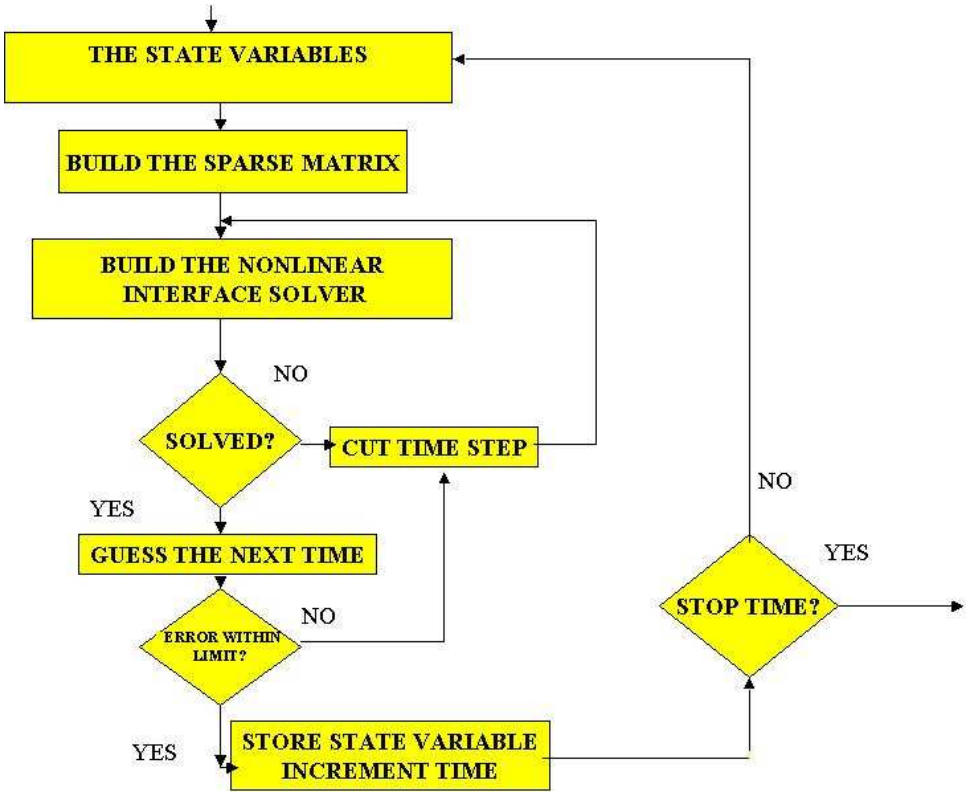

Figure 3.3: Flow Chart

sharp changes in input. The time step reduces to its minimum value to incorporate these changes. However, this problem can be overcome by setting realistic rise and fall times for square waves, impulse etc.

The algorithm cuts the time step by 1/2 when error limits or convergence has not been satisfied. The time step increases by 2 when the solution has been met for 3 consecutive time points. This reduces the number of time steps that have to be discarded.

3.3.4 Comments

The New time step Algorithm does not use the derivatives stored in the **ADOL-C** routine and calculates the values by calculating the derivatives of the state variables. This is done so as to avoid error while calculating the first time step. **FREEDA** sets the values of all state variables and its derivatives as zero. The first time step is therefore calculated by extrapolation. The same routine is used for all elements and sources. The elements and voltage sources are not polled for break points which results in a considerable reduction in computation as compared to traditional **SPICE**.

Chapter 4

Simulation and Results

The purpose of this chapter is to present some of the results the analysis types implemented in **fREEDA**. The analysis was tested on a 47 diode soliton line and a MESFET circuit. The results of the three analysis types are presented. A comparison between the results on the basis of speed and accuracy are presented in Sections 4.1.3 and 4.2.3.

4.1 Soliton Line

A nonlinear transmission line is regarded by many in the field as an extreme test of the performance of circuit simulators. Nonlinear transmission lines (NLTLs) find applications in a variety of high speed, wide bandwidth systems including picosecond resolution sampling circuits, laser and switching diode drivers, test waveform generators, and mm-wave sources. They have three fundamental characteristics: nonlinearity, dispersion and dissipation. The NLTL consist of coplanar waveguides (CPWs) periodically loaded with reverse biased Schottky diodes. Diode-based NLTL used for pulse generation are extremely nonlinear circuits and are being used to test the robustness of circuit simulators. The

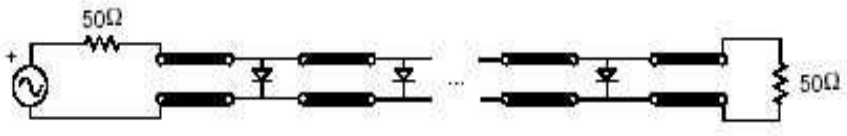

Figure 4.1: 47 diode soliton line

NLTL considered here was designed with a balance between the nonlinearity of the loaded nonlinear elements and the dispersion of the periodic structure which results in the formation of a stable soliton . The nonlinearity of NLTLs is principally due to the voltage dependent capacitance of the diodes and the dissipation is due to the conductor losses in the CPWs.

The NLTL was modeled using generic transmission lines with frequency-dependent loss and Schottky diodes. Skin effect was taken into account in the modelling of the transmission lines. The NLTL model is shown in Figure 4.3 and is excited by a 9 GHz sinusoid. The NLTL was designed for a 24 GHz initial Bragg frequency, 225 GHz final Bragg frequency, 0.952097 tapering rule, and 120 ps total compression [9]. It contains 48 sections of CPW transmission lines and 47 diodes. The drive is a 27 dBm sine wave with -3 V dc bias.

4.1.1 Results

The results of nonlinear transmission lines(NLTLs) are presented in this section using three simulation algorithms.

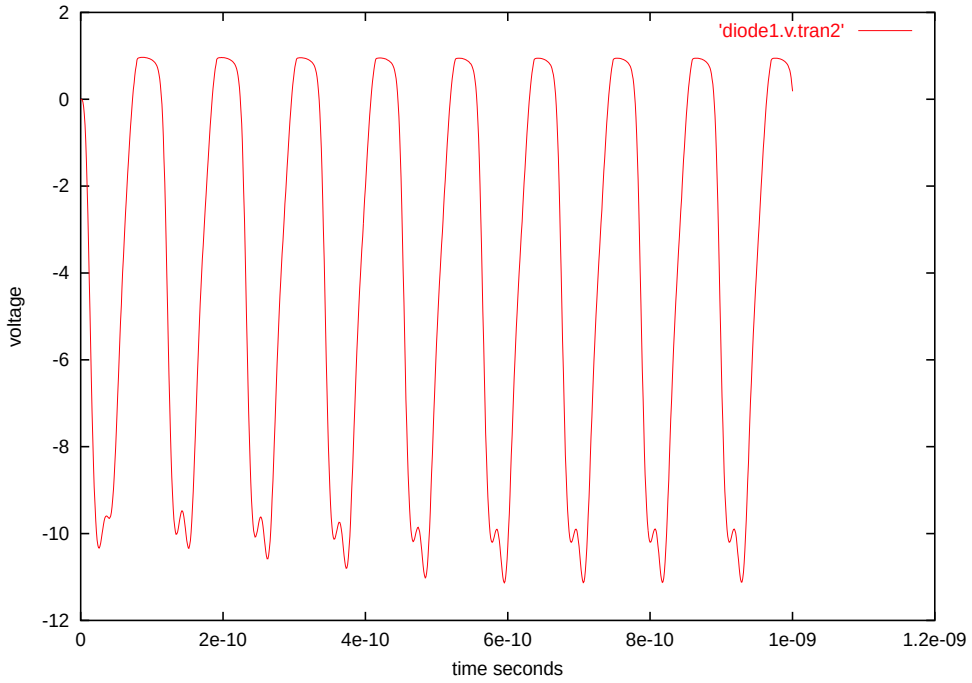

Figure 4.2: Voltage across diode D1 of soliton line with fixed time step Algorithm at 9Ghz. The method does not check for error or convergence

- Fixed Time Step algorithm solves the entire circuit without any error calculation or convergence check as described in Section 3.1.2. The simulated results are as shown in Figure 4.2

- SPICE like Algorithm solves the entire circuit using linear interpolation as described in Section 3.2.2. The simulated results are as shown in Figure 4.3

- New Time Step Algorithm solves the circuit using the algorithm described in Section 3.3.2 The simulated results are shown in Figure 4.4

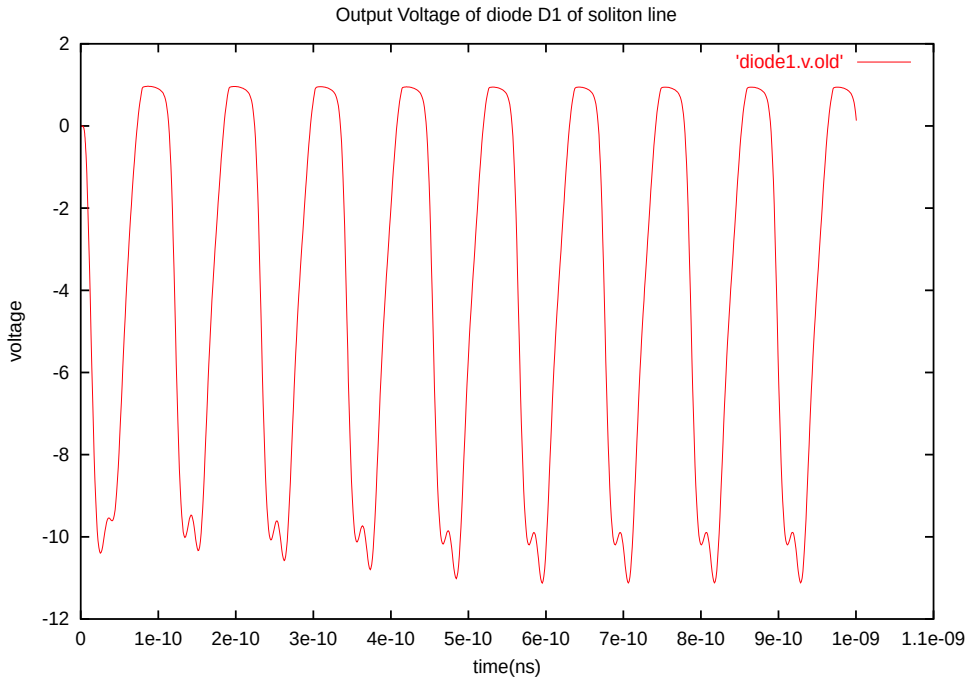

Figure 4.3: Voltage across diode D1 of soliton line with SPICE-like time step Algorithm. The value of state-variables are predicted using linear interpolation.

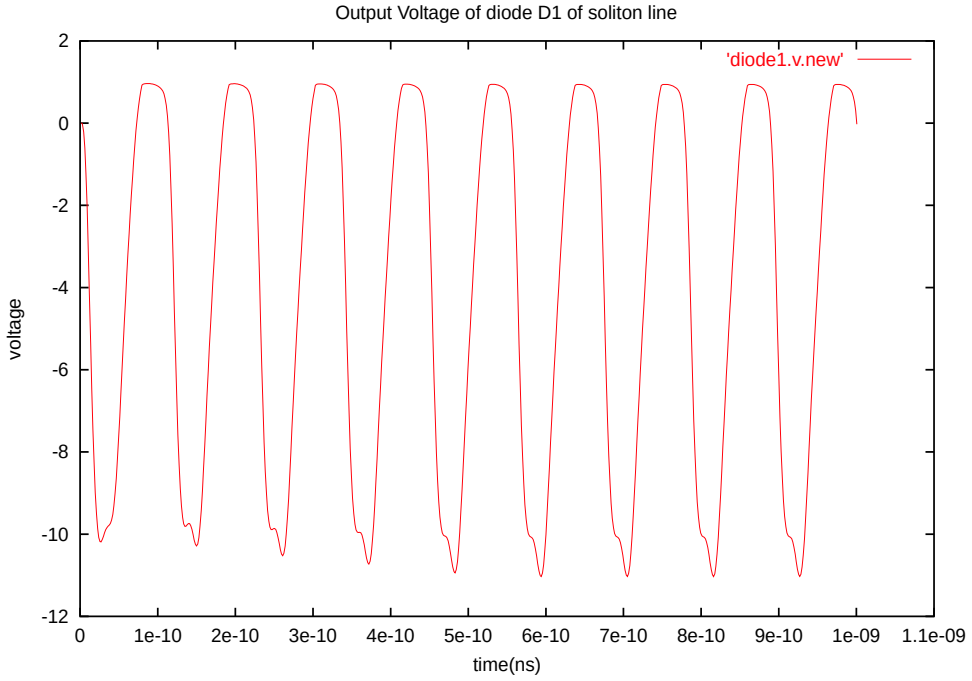

Figure 4.4: Voltage diode D1 of soliton line with new time-step algorithm. The value of state-variables are predicted using a combination of Backward Euler and Trapezoidal derivatives

The new time step is considerably faster than the **SPICE** like representation. The **SPICE** like analysis takes 1030 steps to reach the final stop time where as the new time step analysis takes only 670 time steps to complete it, which is about 34.95% reduction in computation time.

Method	**Number of time steps**
`Fixed time Step Algorithm`	1000
`SPICE like time step Algorithm`	1030
`New Time step Algorithm`	670

Table 4.1: Results of Soliton Line

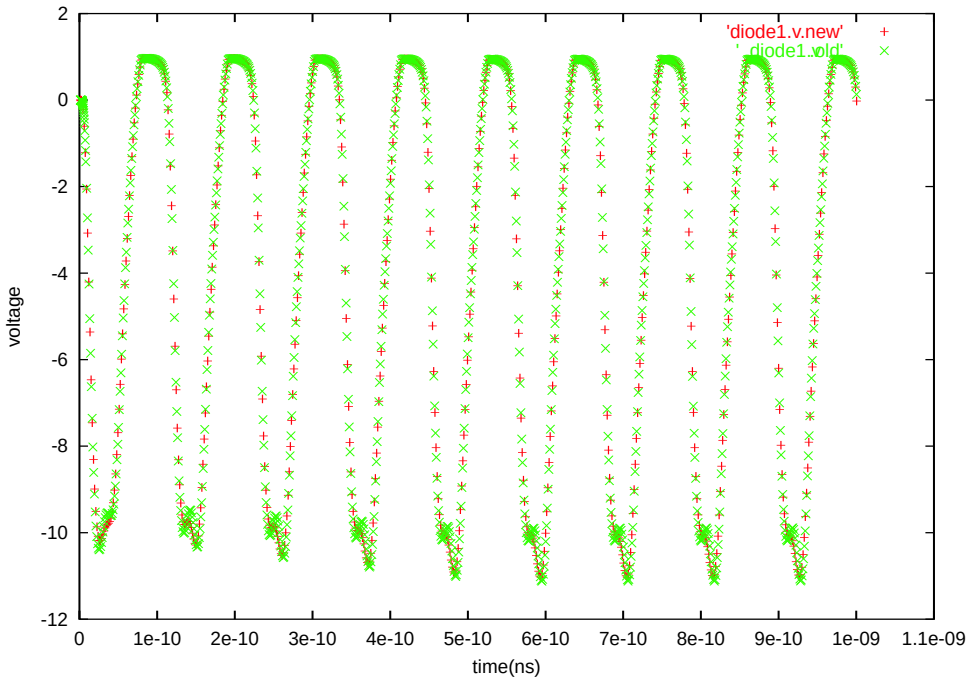

Figure 4.5: Bunching of time points along curves are demonstrated. The SPICE-like approach takes more time steps to solves curves as compared to the new time-step approach.

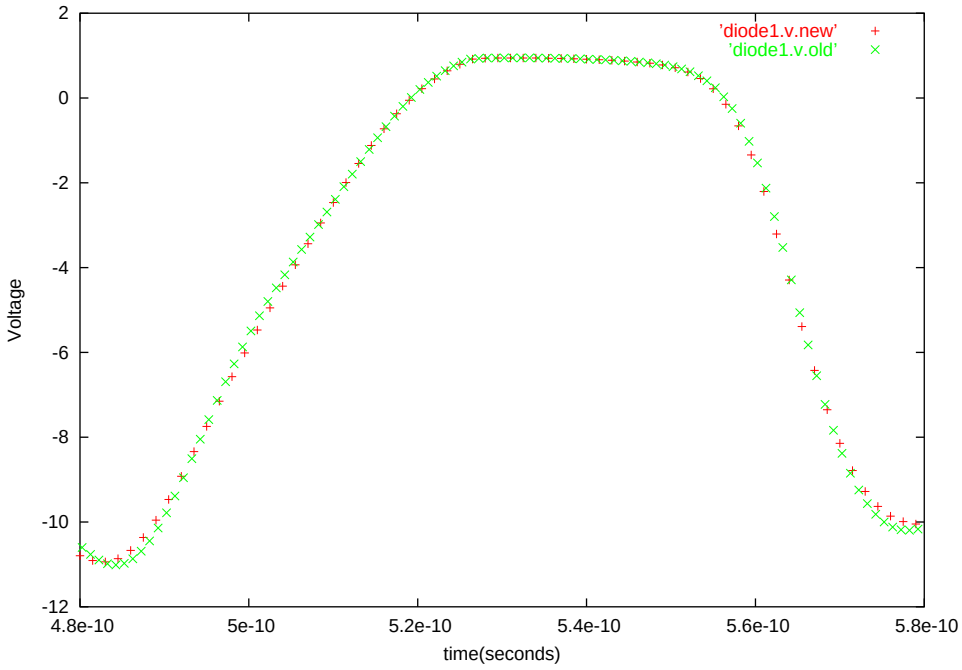

Figure 4.6: Bunching effect of time points .

4.2 MESFET

The MESFET implements the Materka-Kacprzac model for a MESFET. It requires two state variables, but only one of them needs to be delayed.A MESFET Amplifier is shown in the circuit.

4.2.1 Results

The results of the Materka-Kacprzac MESFET circuit is presented here using three simulation algorithms

- Fixed Time Step Algorithm solves the entire circuit without any error calculation or convergence check as described in Section 3.1.2.

  The simulated results are shown in Figure 4.8

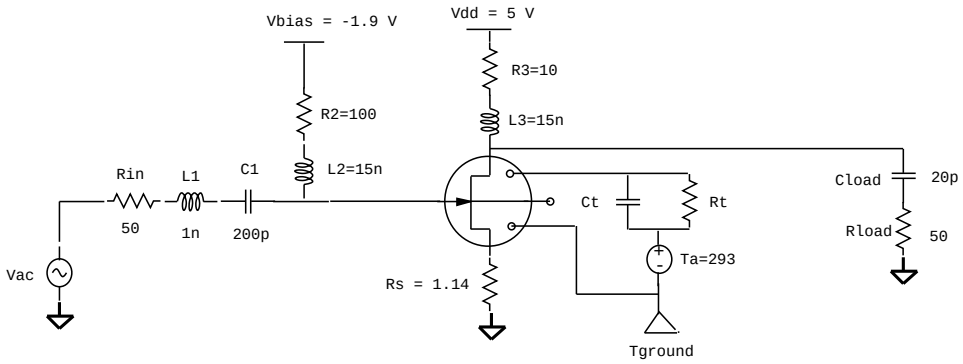

Figure 4.7: MESFET Circuit

- SPICE like Algorithmsolves the entire circuit using linear interpolation as described in Section 3.2.2. The simulated results are as shown in Figure 4.9

- New Time Step Algorithm solves the circuit using the algorithm described in Section 3.3.2. The simulated results are shown in Figure 4.4.

The New time step algorithm is faster as compared to the **SPICE** like algorithm. The New time step algorithm takes about 3338 time steps to reach the final stop time whereas the **SPICE** like algorithm takes about 5009 time steps to reach the final stop time which is about 33.33% reduction in computation time.

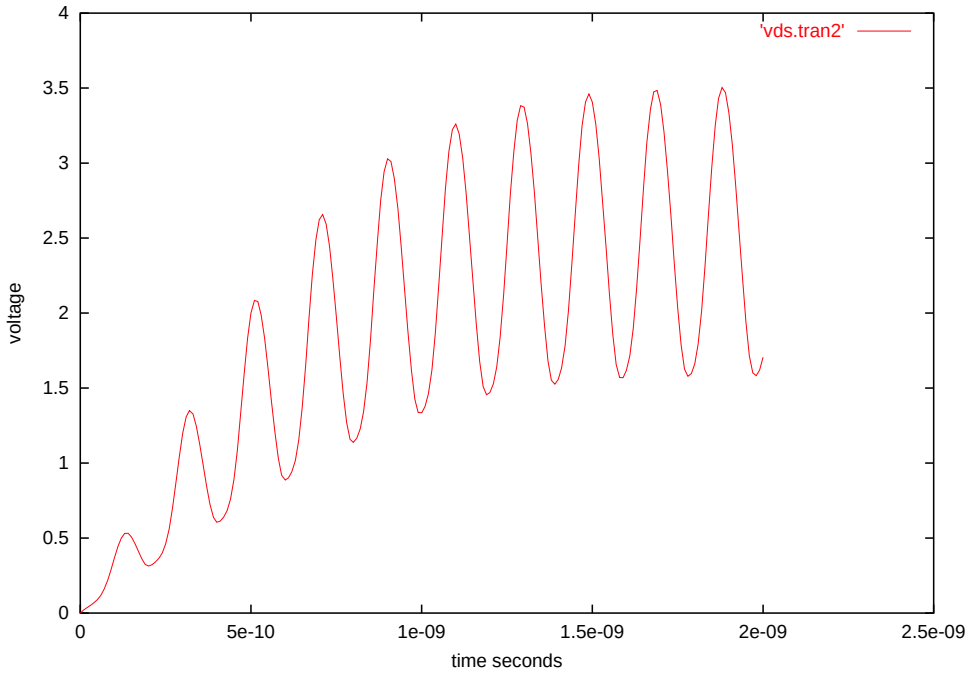

Figure 4.8: Output of the MESFET circuit with fixed time step analysis.

Method	**Number of time steps**
`Fixed time Step Algorithm`	5000
`SPICE like time step Algorithm`	5009
`New Time step Algorithm`	3338

Table 4.2: Results of MESFET Circuit

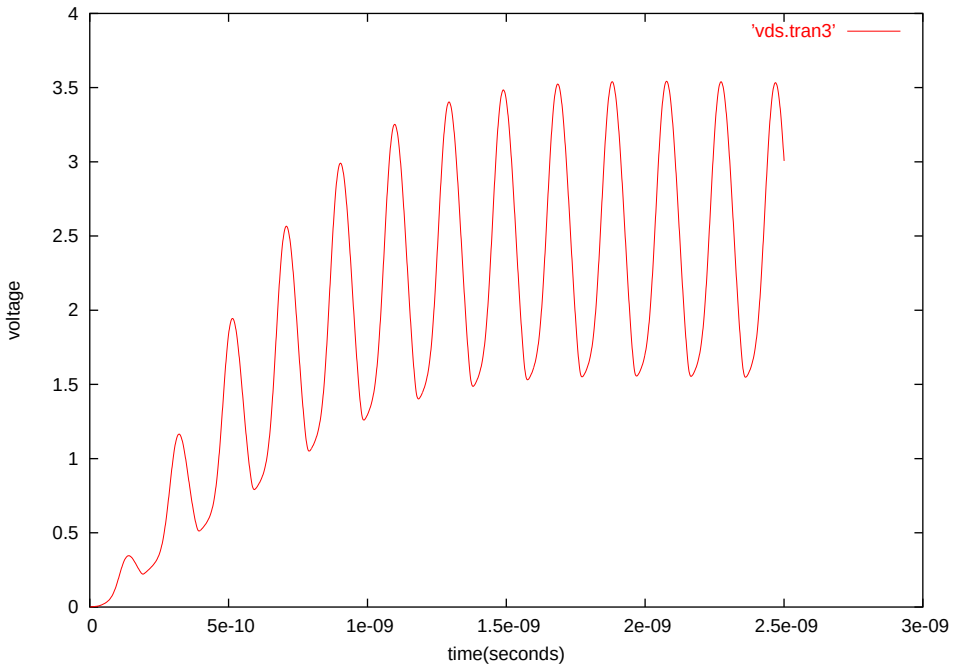

Figure 4.9: Output of a Mesfet circuit with SPICE like time step analysis. Linear Interpolation is used to predict the values of state-variables.

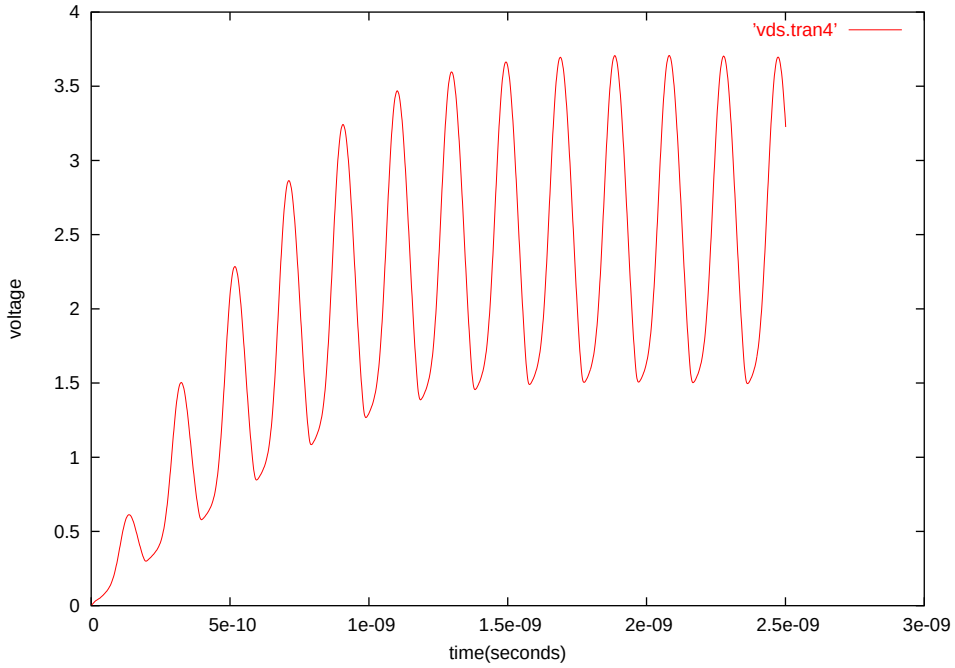

Figure 4.10: Output of a MESFET Circuit with New time step analysis. A combination of Backward Euler and Trapezoidal derivatives are used to predict the value of state variables

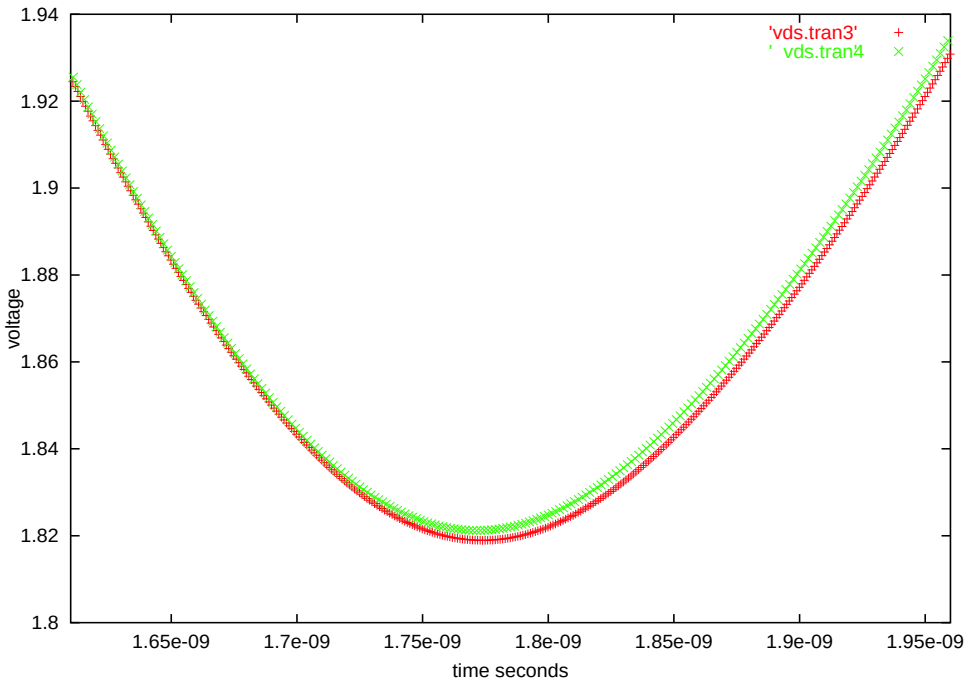

Figure 4.11: Bunching of time points at the Curves in Mesfet Circuit. The SPICE-like approach takes more time points to solve curves as compared to the new time-step approach

4.3 Conclusions

The amount of time taken by either the new time step algorithm or the **SPICE** like algorithm to compute the matrices is the same in both the cases. The reduction in computation time is achieved by making a better estimate of the values of the state variables at the next time step.

The two important sources of error i.e. the errors due to the models, errors due to non ideal convergence and truncation error are taken into account during the analysis.

4.4 Effect of Tolerance

Tolerance is one of the deciding factors while calculating the acceptable time step during analysis. In the New time step algorithm error increases linearly with increase in tolerance where as in the SPICE-like approach the behavior is non-linear.

The linear behavior of the New Time Step Algorithm helps in making a better estimate of the time step.

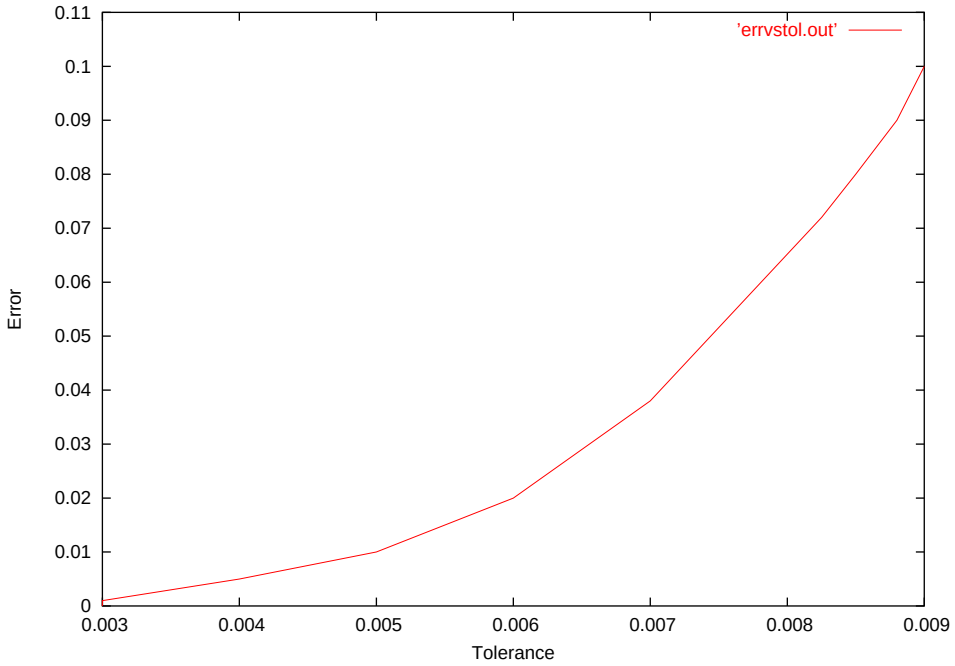

Figure 4.12: Effect of error vs. tolerance using the New Time Step Algorithm.

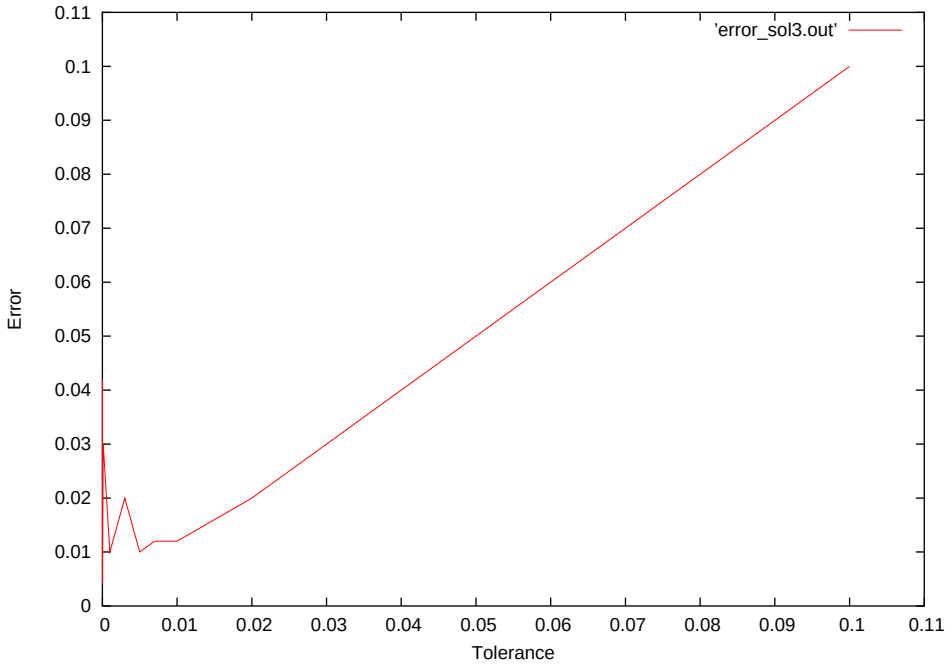

Figure 4.13: Effect of error vs. tolerance using the SPICE-like Algorithm.

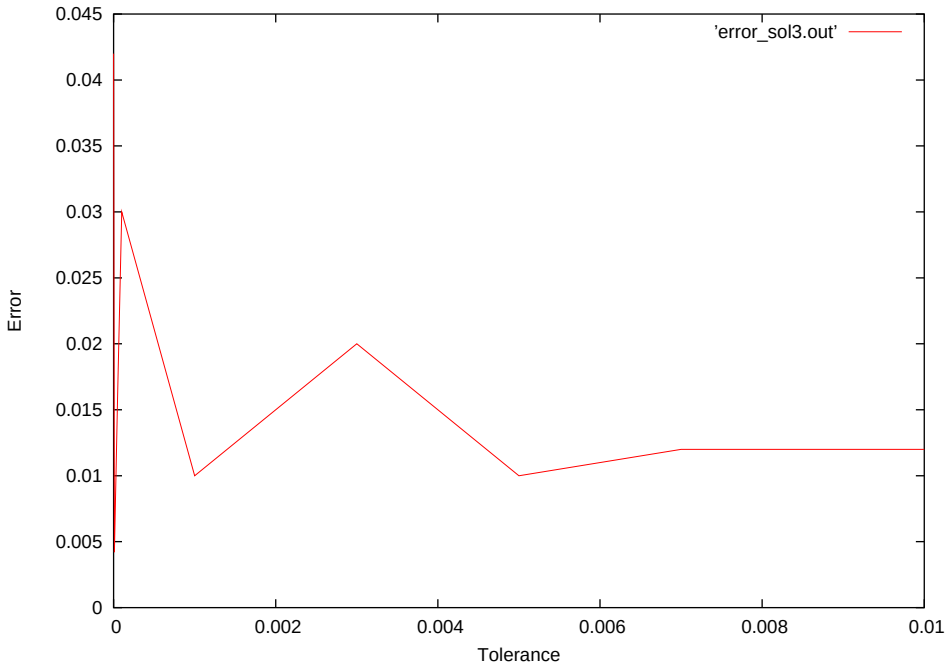

Figure 4.14: Discontinuities in error vs. tolerance using the SPICE-like Algorithm.

Chapter 5

Conclusions and Future Research

5.1 Conclusions

As mentioned in Chapter 1 the difference between the derivatives of Backward Euler and Trapezoidal is a good measure of error. Trapezoidal derivatives are underdamped where as Backward Euler is overdamped. The exact solution lies in between the two predicted values.

Circuits vary on how much they accumulate error. Circuits that tend to be very sensitive to simulator errors include charge storage circuits such as switched capacitor circuits and memories-chaotic circuits, such as oversampled analog/digital convertors and autonomous circuits such as oscillators. Analog circuits with long time constants tend to retain errors for a long time and subsequent errors tend to accumulate.

5.2 Future Research

• The analysis types have to be tested on other models such **MOS** and **BJT**

• The effect of change in tolerance on the results of the circuits such as oscillators etc have to be studied.

Appendix A

Source code

This section contains the C++ code of the SPICE like Algorithm.

A.1 C++ code of SPICE like Algorithm

```
#include "SVTran3.h" #include "TimeMNAM.h" #include
"TimeDomainSV.h" #include "NLSInterface.h" #include "Euler.h"
#include "Trapezoidal.h" #include  MV_P"/mvblasd.h" extern int
superLU_print_enable ;


extern "C" { #include "../inout/ftvec.h" #include
"../inout/report.h"

}

void buildTIncidence(ElemFlag mask, Circuit*& my_circuit,
             IntMatrix& T, ElementVector& elem_vec,
             int& n_states, int& max_n_states);

// Static members
const unsigned SVTran3::n_par = 12;

// Element information
ItemInfo SVTran3::ainfo = {
  "SVTran3",
  "State- variable based time marching transient with variable time step ",
  "Shubha Vijaychand",
  DEFAULT_ADDRESS
};

// Parameter information
ParmInfo SVTran3::pinfo[] = {
  {"tstop", "Stop time (s)", TR_DOUBLE, true},
  {"tstep", "Time step (s)", TR_DOUBLE, true},
  {"nst", "No save time (s)", TR_DOUBLE, false},
  {"deriv", "Approximate derivatives or use automatic diff.", TR_INT, false},
  {"msv", "Use Msv flag", TR_BOOLEAN, false},
  {"im", "Integration method", TR_INT, false},
  {"savenode", "Save node voltages", TR_BOOLEAN, false},
  {"permc_spec", "Permutation ordering to factor Msv (0, 1 or 2)",
   TR_INT, false},
  {"out_steps", "Number of steps skipped for output simulation progress",
   TR_INT, false},
  {"gcomp", "Compensation network conductance (S)", TR_DOUBLE, false},
  {"abstol", " absolute tolerance of the circuit", TR_DOUBLE, false},
```

```
  {"reltol", "relative tolerance of the circuit", TR_DOUBLE, false}

};


SVTran3::SVTran3() : Analysis(&ainfo, pinfo, n_par), ls_size(0),
  superLU(false)
{
  // Parameter stuff
  paramvalue[0] = &(tf);
  paramvalue[1] = &(h);
  paramvalue[2] = &(nst = zero);
  paramvalue[3] = &(deriv = 0);
  paramvalue[4] = &(use_msv = true);
  paramvalue[5] = &(int_method = 1);
  paramvalue[6] = &(savenode = true);
  paramvalue[7] = &(permc_spec = 2);
  paramvalue[8] = &(out_steps = 200);
  paramvalue[9] = &(gcomp = 1e-4);
  paramvalue[10] = &(abstol = 0.05);
  paramvalue[11]= &(reltol = 0.05);

}


SVTran3::~SVTran3() {
  assert(!superLU);

}

void SVTran3::run(Circuit* cir) {
  this->cir = cir;
  char msg[80];

  // Build time domain MNAM
  ElemFlag mnam_mask(LINEAR);
  TimeMNAM mnam(cir, mnam_mask);
  // Supress the output in SuperLU
 superLU_print_enable = 0;
  // Build T matrix and nonlinear element vector
  n_states = 0;
  int max_n_states = 0;
  ElemFlag mask(NONLINEAR);
  buildTIncidence(mask, cir, T, elem_vec, n_states, max_n_states);
 if (gcomp)
    // Add the compensation resistors to the MNAM
    for (int i = 0; i < n_states; i++) {
      mnam.setMAdmittance(T(0,i), T(1,i) , gcomp);
    }
  // Setup simulation variables (use circular vectors)
  ls_size = mnam.getDim();
  list_size = int  (((tf - nst) / h + 2)*50);
  if (savenode)
    cU = new CircVector(list_size, ls_size);
  else
    cU = new CircVector(51, ls_size);
  cX = new CircVector(list_size, n_states);
  cVnl = new CircVector(list_size, n_states);
  cInl = new CircVector(list_size, n_states);
  cTime = new CircVector(list_size, 1);
  DenseVector x_v(n_states);
  // debug
  DenseVector f_v(n_states);
  // debug
  ssv = DenseVector(n_states);
  DenseVector sf(ls_size);
  vl1 = DenseVector(n_states);
  itmp2 = DenseVector(n_states);
```

```
tmp_v = DenseVector(ls_size);
rhs_v = DenseVector(ls_size);
c_v = DenseVector(ls_size);
r_v = DenseVector(ls_size);
rhs_p = &(rhs_v[0]);
r = &(r_v[0]);
c = &(c_v[0]);
// Choose an integration method and combine M and M'
LIntegMethod *l_im;
NLIntegMethod *nl_im;
if (int_method == 1) {
  l_im = new LTrapezoidal(cU, &mnam);
  nl_im = new NLTrapezoidal(cX,cTime, h);
}
else {
  l_im = new LEuler(cU, &mnam);
  nl_im = new NLEuler(cX,  h);
}
// Create Matrix
SparseMatrix M(ls_size, ls_size, 6 * ls_size);


// decalaration  of the variables used
double error;
// store the value of the timestep specified in the netlist
double net_timestep = h;
double prev_timestep = 0;

// Create the interface class
tdsv = new TimeDomainSV(nl_im, max_n_states);

// Setup nonlinear solver
NLSInterface* nlsi = new NLSInterface(this, n_states, deriv);

// Begin main loop
double residual = zero;
report(MESSAGE, "--- Starting transient simulation ...\n");
sepLine();
report(MESSAGE, "| Step \t| Time (s) \t| Residual (V)\t|");
sepLine();
double ctime = zero;
// Set zero initial conditions
DenseVector::iterator ii = x_v.begin();
DenseVector::iterator iend = x_v.end();
for (; ii != iend; ++ii)
  *ii = zero;

// Build M = G + a C
l_im->buildMd(M, h);
// Factor M
superLUFactor(M);

// Create Msv if requested
if (use_msv)
  buildMsv();

//  NLEuler * euler = new NLEuler(cX , h);
// NLTrapezoidal* trap = new NLTrapezoidal(cX, h);

 nt = 0;
 double ctime1;
 ctime1=0;
cX ->getCurrent()=zero;
cX ->advance();
cX ->getCurrent() =zero;
cX -> advance();
cTime ->getCurrent()[0]=-h;
cTime->advance();
```

```
 cTime ->getCurrent()[0]=zero;
 cTime ->advance();
 while (ctime <tf)
 {
      nt++;
      error=0;
int changenstep=0;
int changestep=0;
      do{

          if (prev_timestep != h)
       {
         // euler ->changeStep(h);
         //trap ->changeStep(h);
         nl_im->changeStep(h);
         freeSuperLU();
              M = SparseMatrix (ls_size, ls_size, 6 * ls_size);
          l_im ->buildMd(M,h);
              // Factor M
              superLUFactor(M);
              // Create Msv if requested
              if (use_msv)
           buildMsv();
           }


 ctime1= ctime+h;
       cTime->getCurrent()[0] = ctime1;
           // Update time in interface class
           tdsv->setTime(&(cX->getCurrent()[0]),
         &(cVnl->getCurrent()[0]),
         &(cInl->getCurrent()[0]),
         nt, ctime1, h);
            // Update compressed source vector
            l_im->buildSf(sf, ctime1);
            f1(sf, ssv);


           // Solve the nonlinear equation (call the nonlinear solver).
              double this_res;
            if( prev_timestep >  h)
          {
              for(int i=0; i< n_states; i++)
          x_v[i]=cX->getPrevious(1)[i];
              }

       int RC= nlsi->solve(&(x_v[0]), this_res);
           if(RC<0)
       changenstep=1;
       else
       changenstep=0;
       residual += this_res * this_res;
           error = zero;


           DoubleVector XP(n_states);
           DoubleVector x1, x2, x3;
       double t1, t2, t3;
           DoubleVector diff;
           double k,absdiff, reldiff, factor2;

           // x1=xn-1 x2=xn, x3=xn+1



           x1= cX-> getPrevious(2);
           x2=cX-> getPrevious(1);
           x3=cX-> getCurrent();
```

```
      // t1=tn-1 t2=tn t3=tn+1
          t1=cTime-> getPrevious(2)[0];
          t2=cTime-> getPrevious(1)[0];
          t3=cTime-> getCurrent()[0];
          k= (t3-t1) / ( t2 - t1);
          // calculate the value  at the next timestep by extrapolating
          XP=0.;
      XP+= x2;
          XP -=x1;
          XP *=k;
          XP += x1;
          diff= XP- x3;
          absdiff= norm(diff);
          factor2= norm(x3);

          if(factor2< 1e-3)
          {

        error=   absdiff;
            if(error > abstol)
            changestep=1;
            else
            changestep=0;
          }
            else
          {
         reldiff= norm(diff)/factor2;
         error= reldiff;
              cout<<"error "<<error;
          if(error > reltol)
              changestep=1;
              else
              changestep=0;
          }

             prev_timestep=h;
       if (changestep || changenstep) {
         if (h > 1e-13)

       h *= 0.5;
       }
       else if(h<net_timestep)

        h =2 *h;
  //else if (h < net_timestep)
                  // h = min(h*2.0, net_timestep);
                  //h = ((2.0*h- net_timestep) < 0 ? 2.0*h : net_timestep);

    } while(h<prev_timestep);
    ctime= ctime1;
                  // Update the vectors with the results
   updateVInl(&(x_v[0]));
   updateU(sf);
            // Update previous derivatives in integration method (if required)
   nl_im->store();
   l_im->store();
   // Advance the circular lists for the next step.
   cU->advance();
   cX->advance();
   cVnl->advance();
   cInl->advance();
   cTime->advance();
   // Print table line
   if (!(nt % out_steps))
   {
     sprintf(msg, "| %d  \t| %g  \t| %g\t|", nt, ctime, sqrt(residual));
     report(MESSAGE, msg);
   }
```

```
  }
  residual = sqrt(residual);

  sprintf(msg, "--- Residual: %g", residual);
  sepLine();
  newLine();
  report(MESSAGE, msg);


  doOutput();
  // Erase allocated space
  delete nlsi;
  delete tdsv;
  delete nl_im;
  delete l_im;
  freeSuperLU();
  delete cInl;
  delete cVnl;
  delete cX;
  delete cU;

  return;
}


void SVTran3::superLUFactor(SparseMatrix& M) {
  // Call freeSuperLU to release memory

  assert(!superLU);
  double *tmp_p = &(tmp_v[0]);

  // Set source vector to zero since we are not interested in the result
  // in this routine (Make this better later).
  DenseVector::iterator ii = rhs_v.begin();
  DenseVector::iterator iiend = rhs_v.end();
  for (; ii != iiend; ++ii)
    *ii = zero;

  int nrhs = 1;
  elem_val = new double[M.nnz()];
  row_index = new int[M.nnz()];
  col_pointer = new int[ls_size + 1];
  // Until we find a better way, copy matrix into a separate structure
  SparseMatrix::iterator i, iend;
  SparseMatrix::OneD::iterator j, jend;
  int count = 0;
  int col_count = 0;
  i = M.begin();
  iend = M.end();
  for (; i != iend; ++i) {
    j = (*i).begin(); jend = (*i).end();
    col_pointer[col_count++] = count;
    for (; j != jend; ++j) {
      elem_val[count] = (*j);
      row_index[count] = j.row();
      count++;
    }
  }
  col_pointer[col_count] = count;
  assert(count == M.nnz());
  //  char msg[80];
  //  sprintf(msg, "Matrix size = %d", ls_size);
  //report(MESSAGE, msg);
  //sprintf(msg, "Matrix nnz = %d", count);
  //report(MESSAGE, msg);
  dCreate_CompCol_Matrix(&A, ls_size, ls_size, M.nnz(),
             elem_val, // elem_val pointer
             row_index, // row_index pointer
```

```
            col_pointer, // col_pointer pointer
            NC, _D, GE);

  dCreate_Dense_Matrix(&B, ls_size, nrhs, rhs_p, ls_size, DN, _D, GE);
  dCreate_Dense_Matrix(&X, ls_size, nrhs, tmp_p, ls_size, DN, _D, GE);
  perm_r = new int[ls_size];
  perm_c = new int[ls_size];

  /*
   * Get column permutation vector perm_c[], according to permc_spec:
   *   permc_spec = 0: use the natural ordering
   *   permc_spec = 1: use minimum degree ordering on structure of A'*A
   *   permc_spec = 2: use minimum degree ordering on structure of A'+A
   */
  get_perm_c(permc_spec, &A, perm_c);

  etree = new int[ls_size];
  factor_param_t *ftp = NULL;
  char fact, equed, trans, refact;
  fact      = 'E';
  equed     = 'B';
  trans     = 'N';
  refact    = 'N';
  double recip_pivot_growth;
  double rcond;
  // These are vectors of 1 element (nrhs elements)
  double ferr(0), berr(0);
  mem_usage_t mem_usage;
  int info;
  dgssvx(&fact, &trans, &refact, &A, ftp, perm_c, perm_r, etree, &equed, r, c,
     &L, &U, work, lwork, &B, &X, &recip_pivot_growth, &rcond,
     &ferr, &berr, &mem_usage, &info);
  // cout << "equed: " << equed << endl;
  //cout << "recip_pivot_growth = " << recip_pivot_growth << endl;
  //sprintf(msg, "1 / Condition number = %g", rcond);
  //report(MESSAGE, msg);
  //sprintf(msg, "info = %d", info);
  //report(MESSAGE, msg);
  //sprintf(msg, "ferr = %g", ferr);
  //report(MESSAGE, msg);
  //sprintf(msg, "berr = %g", berr);
  //report(MESSAGE, msg);

  if (info)
    report(FATAL, "There was a problem factoring the MNAM.");

  NCformat *Ustore;
  SCformat *Lstore;
  Lstore = (SCformat *) L.Store;
  Ustore = (NCformat *) U.Store;
  // sprintf(msg, "No of nonzeros in factor L = %d", Lstore->nnz);
  //report(MESSAGE, msg);
  //sprintf(msg, "No of nonzeros in factor U = %d", Ustore->nnz);
  //report(MESSAGE, msg);
  //sprintf(msg, "No of nonzeros in L+U = %d",
  //  Lstore->nnz + Ustore->nnz - ls_size);
//report(MESSAGE, msg);
  // sprintf(msg, "L\\U MB %.3f\ttotal MB needed %.3f\texpansions %d",
  //  mem_usage.for_lu/1e6, mem_usage.total_needed/1e6,
  //  mem_usage.expansions);
  //report(MESSAGE, msg);
  fflush(stdout);

  // This is not used anymore
  Destroy_SuperMatrix_Store(&X);
  superLU = true;

} void SVTran3::freeSuperLU() {
```

```
  assert(superLU);
  // Free vectors
  delete [] etree;
  delete [] perm_c;
  delete [] perm_r;
  // Free supermatrix structures
  Destroy_SuperMatrix_Store(&A);
  Destroy_SuperMatrix_Store(&B);
  // Free L and U matrices
  if ( lwork >= 0 ) {
    Destroy_SuperNode_Matrix(&L);
    Destroy_CompCol_Matrix(&U);
  }
  delete [] col_pointer;
  delete [] row_index;
  delete [] elem_val;
  superLU = false;
}

void SVTran3::f1(DenseVector& s1, DenseVector& v1l) {

  assert(superLU);
  //  do diag(r)*B
  ele_mult(s1, r_v, rhs_v);
  StatInit(sp_ienv(1), sp_ienv(2));
  // Solve using triangular solve for speed
  char trans = 'N';
  int info;
  dgstrs(&trans, &L, &U, perm_r, perm_c, &B, &info);
  if (info)
    cerr << "Warning: info = " << info << endl;

  // Premultiply rhs by T' (also premultiply by diag(c))
  for (int k=0; k < n_states; k++) {
    v1l[k] = zero;
    if (T(0,k)) {
      int tmpidx(T(0,k) - 1);
      v1l[k] += rhs_p[tmpidx] * c[tmpidx];
    }
    if (T(1,k)) {
      int tmpidx((T(1,k) - 1));
      v1l[k] -= rhs_p[tmpidx] * c[tmpidx];
    }
  }
  StatFree();
  return;
}


void SVTran3::buildMsv() {
  // This Msv is actually -Msv to avoid having to invert the currents sign
  Msv = DenseMatrix(n_states, n_states);
  // Clear input vector
  DenseVector::iterator ii = tmp_v.begin();
  DenseVector::iterator iend = tmp_v.end();
  for (; ii != iend; ++ii)
    *ii = zero;

  for (int i=0; i < n_states; i++) {
    // Build by columns
    if (T(0,i))
      tmp_v[T(0,i) - 1] -= one;
    if (T(1,i))
      tmp_v[T(1,i) - 1] += one;
    // Solve linear system and reduce
    f1(tmp_v, v1l);
```

```
    // Copy result to Msv column
    for (int j=0; j < n_states; j++)
      Msv(j,i) = vl1[j];
    // Clear vector for next iteration
    if (T(0,i))
      tmp_v[T(0,i) - 1] = zero;
    if (T(1,i))
      tmp_v[T(1,i) - 1] = zero;

  }
  // print_all_matrix(Msv);
  return;
}


void SVTran3::func_ev(double* X, double* F) {
  // Evaluate first the time-domain elements.
  updateVInl(X);
  // Tell tdsv that the first evaluation is already completed.
  tdsv->clearflag();
  // get nonlinear current and voltages vector
  DoubleVector& vnl = cVnl->getCurrent();
  DoubleVector& inl = cInl->getCurrent();
  if (use_msv) {
    ExtVector itmp1(&(inl[0]), n_states);
    ExtVector vtmp1(&(vnl[0]), n_states);
    add(itmp1, scaled(vtmp1, -gcomp), itmp2);
    mult(Msv, itmp1, vl1);
  }
  else {
    // Calculate error function
    DenseVector::iterator ii = tmp_v.begin();
    DenseVector::iterator iend = tmp_v.end();
    for (; ii != iend; ++ii)
      *ii = zero;
    for (int i=0; i < n_states; i++) {
     double itmp = inl[i] - gcomp * vnl[i];
      if (T(0,i))
    tmp_v[T(0,i) - 1] -= itmp;
      if (T(1,i))
    tmp_v[T(1,i) - 1] += itmp;
    }
    f1(tmp_v, vl1);
  }
  for (int i=0; i < n_states; i++)
    F[i] = ssv[i] + vl1[i] - vnl[i];
}

void SVTran3::jacobian(double* X, DoubleMatrix& J) {
  // Set current state variable values.
  for (int j=0; j < n_states; j++)
    cX->getCurrent()[j] = X[j];

  // Number of elements
  int n_elem = elem_vec.size();
  int i = 0;
  DoubleMatrix& Ju = tdsv->getJu();
  DoubleMatrix& Ji = tdsv->getJi();

  if (use_msv) {
    // Go through all the nonlinear elements
    for (int k = 0; k < n_elem; k++) {
      int cns = elem_vec[k]->getNumberOfStates();
      // Set base index in interface object
      tdsv->setIBase(i, cns);
      // Call element evaluation
      elem_vec[k]->deriv_svTran(tdsv);
```

```
      for (int l=0; l < n_states; l++)
    for (int j=0; j < cns; j++) {
      J(l, i+j) = zero;
      for (int n=0; n < cns; n++)
        J(l, i+j) += Msv(l, i+n) * (Ji(n, j)- gcomp * Ju(n,j)) ;
    }

      for (int j=0; j < cns; j++)
    for (int n=0; n < cns; n++)
      J(i+j, i+n) -= Ju(j, n);

      i += cns;
    }
  }
  else {
    // Go through all the nonlinear elements
    for (int k = 0; k < n_elem; k++) {
      int cns = elem_vec[k]->getNumberOfStates();
      // Set base index in interface object
      tdsv->setIBase(i, cns);
      // Call element evaluation
      elem_vec[k]->deriv_svTran(tdsv);

      for (int j=0; j < cns; j++) {
    // Evaluate the linear contribution
    DenseVector::iterator ii = tmp_v.begin();
    DenseVector::iterator iend = tmp_v.end();
    for (; ii != iend; ++ii)
      *ii = zero;
    for (int n=0; n < cns; n++) {
      int ipn = i + n;
      if (T(0,ipn))
        tmp_v[T(0,ipn) - 1] -= (Ji(n, j) - gcomp * Ju(n,j));
      if (T(1,ipn))
        tmp_v[T(1,ipn) - 1] += (Ji(n, j) - gcomp * Ju(n,j));
    }
    f1(tmp_v, vl1);
    for (int n=0; n < n_states; n++)
      J(n, i+j) = vl1[n];
    // add nonlinear contribution
    for (int n=0; n < cns; n++) {
      J(i+n, i+j) -= Ju(n, j);
    }
      }
      i += cns;
    }
  }
}


void SVTran3::updateVInl(double* x_p) {
  // Set current state variable values.
  for (int j=0; j < n_states; j++)
    cX->getCurrent()[j] = x_p[j];

  // Number of elements
  int n_elem = elem_vec.size();
  int i = 0;
  // Go through all the nonlinear elements
  for (int k = 0; k < n_elem; k++) {
    int cns = elem_vec[k]->getNumberOfStates();
    // Set base index in interface object
    tdsv->setIBase(i, cns);
    // Call element evaluation
    elem_vec[k]->svTran(tdsv);
    i += cns;
  }
```

```
}

void SVTran3::updateU(DenseVector& s1) {
  copy(s1, tmp_v);
  DoubleVector& inl = cInl->getCurrent();
  DoubleVector& vnl = cVnl->getCurrent();
  for (int i=0; i < n_states; i++) {
  double itmp = inl[i] - gcomp * vnl[i];
    if (T(0,i))
      tmp_v[T(0,i) - 1] -= itmp;
    if (T(1,i))
      tmp_v[T(1,i) - 1] += itmp;
  }

  assert(superLU);
  //  do diag(r)*B
  ele_mult(tmp_v, r_v, rhs_v);
  StatInit(sp_ienv(1), sp_ienv(2));
  // Solve using triangular solve for speed
  char trans = 'N';
  int info;
  dgstrs(&trans, &L, &U, perm_r, perm_c, &B, &info);
  if (info)
    cerr << "Warning: info = " << info << endl;

  ExtVector u_v(&(cU->getCurrent()[0]), ls_size);
  ele_mult(rhs_v, c_v, u_v);
}

void SVTran3::doOutput() {
  // First check if the result matrices contain any data
  assert(cX);

  report(MESSAGE, "--- Writing output vectors ...");
  int out_size;
  if(nt>list_size)
  out_size= list_size-1;
  else
  out_size= nt-1;


//   out_size = (nt < list_size) ? list_size - 1 : nt - 1;

  // Allocate and copy global time vector
  allocTimeV_P(out_size);
  int tindex;
  tindex=0;


  for ( tindex = 0; tindex < out_size; tindex++)
    TimeV_P[tindex] = cTime->getPrevious(out_size - tindex )[0];

  // Temporary vectors
  DoubleVector tmp_x(out_size);
  DoubleVector tmp_i(out_size);
  DoubleVector tmp_u(out_size);

  // Now fill currents and port voltages of time domain devices.
  int current_ns;
  int n_elem = elem_vec.size();   // Number of elements
  int i=0;
  for (int k=0; k < n_elem; k++) {

    current_ns = elem_vec[k]->getNumberOfStates();
    for(int j=0; j< current_ns; j++) {

      // For the current, decompensate while copying.
      for (int tindex=0; tindex < out_size; tindex++) {
```

```
    tmp_x[tindex] = cX->getPrevious(out_size - tindex)[j+i];
    tmp_i[tindex] = cInl->getPrevious(out_size - tindex)[j+i];
    tmp_u[tindex] = cVnl->getPrevious(out_size - tindex)[j+i];
      }
      elem_vec[k]->getElemData()->setRealX(j, tmp_i);
      elem_vec[k]->getElemData()->setRealI(j, tmp_i);
      elem_vec[k]->getElemData()->setRealU(j, tmp_u);
    }
    i += current_ns;
  }

  if (savenode) {
    // For each terminal, assign voltage vector
    Terminal* term = NULL;
    cir->setFirstTerminal();
    while((term = cir->nextTerminal())) {
      // Get MNAM index
      if (term->getRC()) {
    // Remember that MNAM indices begin at 1
    i = term->getRC() - 1;
    for (int tindex=0; tindex < out_size; tindex++)
      tmp_u[tindex] = cU->getPrevious(out_size - tindex)[i];
      }
      else
    // This is a reference terminal
    tmp_u = zero;

      // Set terminal vector
      term->getTermData()->setRealV(tmp_u);
    }
  }
}
```

A.2 Header File of SPICE like Algorithm

```
// This may look like C code, but it is really -*- C++ -*-
//
// time marching transient analysis with variable timestep
// Author:
//          Shubha Vijaychand
//

#ifndef SVTran3_h #define SVTran3_h 1


#include "Analysis.h" #include "OFunction.h" #include
"CircVector.h" #include "CircDouble.h" #include "transim_mtl.h"
extern "C" { #undef _S #undef _C #include SUPERLU_P"/dsp_defs.h"
#include SUPERLU_P"/util.h"
       }

// Main class definition follows
class SVTran3 : public Analysis, public OFunction { public:

  SVTran3();

  ~SVTran3();

  // (from Analysis)
  // The main analysis routine.
  virtual void run(Circuit* cir);

  // (from OFunction)
  // Evaluate error function vector F given X
  virtual void func_ev(double* X, double* F);
  // Evaluate Jacobian
  virtual void jacobian(double* X, DoubleMatrix& J);

  // Write output vectors
  void doOutput();

private:

  void superLUFactor(SparseMatrix& M);
  void freeSuperLU();
  void f1(DenseVector& s1, DenseVector& v1l);
  void updateVInl(double* x_p);
  void updateU(DenseVector& s1);
  void buildMsv();

  Circuit* cir;
  int n_states;
  IntMatrix T;
  ElementVector elem_vec;
  TimeDomainSV *tdsv;
  int ls_size;
  int nt;
  int n_tsteps;
  int list_size;

  DenseVector ssv, vl1, itmp2, tmp_v, rhs_v, c_v, r_v;
  CircVector *cU;
  CircVector *cX;
  CircVector *cVnl;
  CircVector *cInl;
  CircVector *cTime;
  DenseMatrix Msv;

  // SuperLU variables
  SuperMatrix A, L, U, B, X;
  int *perm_r;
```

```
  int *perm_c;
  int *etree;
  double *r, *c;
  double *rhs_p;
  void *work;
  int lwork;
  bool superLU;
  double* elem_val;
  int* row_index;
  int* col_pointer;

  // ------------------- Parameter-related variables
  // Analysis information
  static ItemInfo ainfo;
  // Number of parameters of this element
  static const unsigned n_par;
  // Analysis parameters
  double h, tf, nst,gcomp, abstol, reltol;
  int deriv, int_method, permc_spec, out_steps;
  bool use_msv, savenode;
  // Parameter information
  static ParmInfo pinfo[];
};


#endif
```

A.3 C++ code of New Time Step Algorithm

```
#include "SVTran4.h" #include "TimeMNAM.h" #include
"TimeDomainSV.h" #include "NLSInterface.h" #include "Euler.h"
#include "Trapezoidal.h" #include  MV_P"/mvblasd.h" extern int
superLU_print_enable ;


extern "C" { #include "../inout/ftvec.h" #include
"../inout/report.h"

}

void buildTIncidence(ElemFlag mask, Circuit*& my_circuit,
             IntMatrix& T, ElementVector& elem_vec,
             int& n_states, int& max_n_states);

// Static members
const unsigned SVTran4::n_par = 13;

// Element information
ItemInfo SVTran4::ainfo = {
  "SVTran4",
  "State- variable based time marching transient with variable time step  ",
  "Shubha Vijaychand",
  DEFAULT_ADDRESS
};
// Parameter information
ParmInfo SVTran4::pinfo[] = {
  {"tstop", "Stop time (s)", TR_DOUBLE, true},
  {"tstep", "Time step (s)", TR_DOUBLE, true},
  {"nst", "No save time (s)", TR_DOUBLE, false},
  {"deriv", "Approximate derivatives or use automatic diff.", TR_INT, false},
  {"msv", "Use Msv flag", TR_BOOLEAN, false},
  {"im", "Integration method", TR_INT, false},
  {"savenode", "Save node voltages", TR_BOOLEAN, false},
  {"permc_spec", "Permutation ordering to factor Msv (0, 1 or 2)",
   TR_INT, false},
  {"out_steps", "Number of steps skipped for output simulation progress",
   TR_INT, false},
```

```
  {"gcomp", "Compensation network conductance (S)", TR_DOUBLE, false},
  {"abstol", " absolute tolerance of the circuit", TR_DOUBLE, false},
  {"reltol", "relative tolerance of the circuit", TR_DOUBLE, false},
  {"tol", "tolerance of the circuit", TR_DOUBLE,false}


};


SVTran4::SVTran4() : Analysis(&ainfo, pinfo, n_par), ls_size(0),
  superLU(false)
{
  // Parameter stuff
  paramvalue[0] = &(tf);
  paramvalue[1] = &(h);
  paramvalue[2] = &(nst = zero);
  paramvalue[3] = &(deriv = 0);
  paramvalue[4] = &(use_msv = true);
  paramvalue[5] = &(int_method = 1);
  paramvalue[6] = &(savenode = true);
  paramvalue[7] = &(permc_spec = 2);
  paramvalue[8] = &(out_steps = 200);
  paramvalue[9] = &(gcomp = 1e-4);
  paramvalue[10] = &(abstol = 0.05);
  paramvalue[11]= &(reltol = 0.05);
  paramvalue[12]= &(tol= 1.);

}


SVTran4::~SVTran4() {
  assert(!superLU);

}

void SVTran4::run(Circuit* cir) {
  this->cir = cir;
  char msg[80];

  // Build time domain MNAM
  ElemFlag mnam_mask(LINEAR);
  TimeMNAM mnam(cir, mnam_mask);
  // Supress the output in SuperLU
  superLU_print_enable = 0;
  // Build T matrix and nonlinear element vector
  n_states = 0;
  int max_n_states = 0;
  ElemFlag mask(NONLINEAR);
  buildTIncidence(mask, cir, T, elem_vec, n_states, max_n_states);
if (gcomp)
    // Add the compensation resistors to the MNAM
    for (int i = 0; i < n_states; i++) {
      mnam.setMAdmittance(T(0,i), T(1,i) , gcomp);
    }
  // Setup simulation variables (use circular vectors)
  ls_size = mnam.getDim();
  list_size = int  (((tf - nst) / h + 2)*50);
  if (savenode)
    cU = new CircVector(list_size, ls_size);
  else
    cU = new CircVector(51, ls_size);
  cX = new CircVector(list_size, n_states);
  cVnl = new CircVector(list_size, n_states);
  cInl = new CircVector(list_size, n_states);
  cTime = new CircVector(list_size, 1);
  DenseVector x_v(n_states);
  // debug
  DenseVector f_v(n_states);
```

```
// debug
ssv = DenseVector(n_states);
DenseVector sf(ls_size);
vl1 = DenseVector(n_states);
itmp2 = DenseVector(n_states);
tmp_v = DenseVector(ls_size);
rhs_v = DenseVector(ls_size);
c_v = DenseVector(ls_size);
r_v = DenseVector(ls_size);
rhs_p = &(rhs_v[0]);
r = &(r_v[0]);
c = &(c_v[0]);
// Choose an integration method and combine M and M'
LIntegMethod *l_im;
NLIntegMethod *nl_im;
if (int_method == 1) {
  l_im = new LTrapezoidal(cU, &mnam);
  nl_im = new NLTrapezoidal(cX,cTime, h);
}
else {
  l_im = new LEuler(cU, &mnam);
  nl_im = new NLEuler(cX,cTime, h);
}
// Create Matrix
SparseMatrix M(ls_size, ls_size, 6 * ls_size);


// decalaration  of the variables used
double error;
// store the value of the timestep specified in the netlist
double net_timestep = h;
double prev_timestep = 0;

// Create the interface class
tdsv = new TimeDomainSV(nl_im, max_n_states);

// Setup nonlinear solver
NLSInterface* nlsi = new NLSInterface(this, n_states, deriv);

// Begin main loop
double residual = zero;
report(MESSAGE, "--- Starting transient simulation ...\n");
sepLine();
report(MESSAGE, "| Step \t| Time (s) \t| Residual (V)\t|");
sepLine();
double ctime = zero;

// Set zero initial conditions
DenseVector::iterator ii = x_v.begin();
DenseVector::iterator iend = x_v.end();
for (; ii != iend; ++ii)
  *ii = zero;

// Build M = G + a C
l_im->buildMd(M, h);
// Factor M
superLUFactor(M);

// Create Msv if requested
if (use_msv)
  buildMsv();

NLEuler * euler = new NLEuler(cX , h);
NLTrapezoidal* trap = new NLTrapezoidal(cX, h);

nt = 0;
DoubleVector xdot1(n_states);
DoubleVector xdot2(n_states);
```

```
DoubleVector XP(n_states);
DoubleVector x1, x2, x3;
double t1, t2, t3;
DoubleVector diff;
double k,absdiff, reldiff, factor2;
double  factor,error_prev,error_temp;
error_prev=0;
double ctime1;
ctime1=0;
cX ->getCurrent()=zero;
cX ->advance();
cX ->getCurrent() =zero;
cX -> advance();
cTime ->getCurrent()[0]=-h;
cTime->advance();
cTime ->getCurrent()[0]=zero;
cTime ->advance();

int counter = 0;
while (ctime <tf)
  {
    nt++;
    error=0;
    int changenstep=0;
    int changestep=0;
    double ooh;
    do{

  if (prev_timestep != h)
    {
      ooh = one / h;
      euler ->changeStep(h);
      trap ->changeStep(h);
      nl_im->changeStep(h);
      freeSuperLU();
      M = SparseMatrix (ls_size, ls_size, 6 * ls_size);
      l_im ->buildMd(M,h);
      // Factor M
      superLUFactor(M);
      // Create Msv if requested
      if (use_msv)
        buildMsv();
    }


  ctime1= ctime+h;
  cTime->getCurrent()[0] = ctime1;
  // Update time in interface class
  tdsv->setTime(&(cX->getCurrent()[0]),
            &(cVnl->getCurrent()[0]),
            &(cInl->getCurrent()[0]),
            nt, ctime1, h);
  // Update compressed source vector
  l_im->buildSf(sf, ctime1);
  f1(sf, ssv);


  // Solve the nonlinear equation (call the nonlinear solver).
  double this_res;
  if( prev_timestep >  h)
    {
      for(int i=0; i< n_states; i++)
        x_v[i]=cX->getPrevious(1)[i];
    }

  int RC= nlsi->solve(&(x_v[0]), this_res);
  if(RC<0)
    changenstep=1;
```

```
else
  changenstep=0;
residual += this_res * this_res;
error = zero;



// x1=xn-1 x2=xn, x3=xn+1

if(nt==1)
  {
        x1= cX-> getPrevious(2);
        x2= cX-> getPrevious(1);
        x3= cX-> getCurrent();
    // t1=tn-1 t2=tn t3=tn+1
        t1=cTime-> getPrevious(2)[0];
        t2=cTime-> getPrevious(1)[0];
        t3=cTime-> getCurrent()[0];
        k= (t3-t1) / ( t2 - t1);
        // calculate the value  at the next timestep by extrapolating
    XP = x2;
        XP -= x1;
        XP *= k;
        XP += x1;
        XP -= x3;
        absdiff= norm(XP);
        factor2= norm(x3);
    //xdot1 = x3;
     // xdot1 -= x2;
     // xdot1 *= ooh;
      xdot2 = x3;
     xdot2 -= x2;
     xdot2 *= ooh;

        if(factor2< 1e-3) {
    error =  absdiff;
    if(error > abstol)
      changestep=1;
        else
      changestep=0;
      }
    else {
    reldiff= absdiff / factor2;
    error= reldiff;
    if(error > reltol)
      changestep=1;
    else
      changestep=0;
      }
      }
else
  {
    diff = cX->getCurrent();
    diff -= cX->getPrevious(1);
    diff *= ooh;
    double nbed = norm(diff);
    diff -= xdot1;
    double ndiff = norm(diff);
    error =  ndiff / nbed;
    if(error>tol)
      changestep=1;
    else
      changestep=0;
  }


prev_timestep=h;
if (changestep || changenstep) {
```

```
     if (h > 1e-13) {
        counter = 5;
      h *= 0.5 ;
       }


  }
  else if(h<net_timestep) {
    if (counter == 0)
      h =  3 * h;
      //else
       counter--;
  }
  //else if (h < net_timestep)
  // h = min(h*2.0, net_timestep);
  //h = ((2.0*h- net_timestep) < 0 ? 2.0*h : net_timestep);

    } while(h<prev_timestep);
    ctime= ctime1;
    // Update the vectors with the results
    updateVInl(&(x_v[0]));

    updateU(sf);

    // Calculate trapezoidal derivatives
    // cout<<"xdot1 before deravative calucalation"<<xdot1;
    // cout<<"xdot2 before deravative calculation"<<xdot2;
    xdot2 = xdot1;
    xdot1 =cX->getCurrent();
    xdot1 -=cX->getPrevious(1);
    xdot1 *= 2. * ooh;
    xdot1 -= xdot2;
    // cout<<"xdot1 after deravative calucalation"<<xdot1;
    //cout<<"xdot2 after deravative calculation"<<xdot2;

    // Update previous derivatives in integration method
    nl_im->store();
    l_im->store();
    // Advance the circular lists for the next step.
    cU->advance();
    cX->advance();
    cVnl->advance();
    cInl->advance();
    cTime->advance();
    // Print table line
    if (!(nt % out_steps))
  {
    sprintf(msg, "| %d  \t| %g  \t| %g\t|", nt, ctime, sqrt(residual));
    report(MESSAGE, msg);
  }
  }
residual = sqrt(residual);

sprintf(msg, "--- Residual: %g", residual);
sepLine();
newLine();
report(MESSAGE, msg);


doOutput();
// Erase allocated space
delete nlsi;
delete tdsv;
delete nl_im;
delete l_im;
freeSuperLU();
delete cInl;
delete cVnl;
```

```
  delete cX;
  delete cU;
  return;
}


void SVTran4::superLUFactor(SparseMatrix& M) {
  // Call freeSuperLU to release memory

  assert(!superLU);
  double *tmp_p = &(tmp_v[0]);

  // Set source vector to zero since we are not interested in the result
  // in this routine (Make this better later).
  DenseVector::iterator ii = rhs_v.begin();
  DenseVector::iterator iiend = rhs_v.end();
  for (; ii != iiend; ++ii)
    *ii = zero;

  int nrhs = 1;
  elem_val = new double[M.nnz()];
  row_index = new int[M.nnz()];
  col_pointer = new int[ls_size + 1];
  // Until we find a better way, copy matrix into a separate structure
  SparseMatrix::iterator i, iend;
  SparseMatrix::OneD::iterator j, jend;
  int count = 0;
  int col_count = 0;
  i = M.begin();
  iend = M.end();
  for (; i != iend; ++i) {
    j = (*i).begin(); jend = (*i).end();
    col_pointer[col_count++] = count;
    for (; j != jend; ++j) {
      elem_val[count] = (*j);
      row_index[count] = j.row();
      count++;
    }
  }
  col_pointer[col_count] = count;
  assert(count == M.nnz());
  //  char msg[80];
  //  sprintf(msg, "Matrix size = %d", ls_size);
  //report(MESSAGE, msg);
  //sprintf(msg, "Matrix nnz = %d", count);
  //report(MESSAGE, msg);
  dCreate_CompCol_Matrix(&A, ls_size, ls_size, M.nnz(),
             elem_val, // elem_val pointer
             row_index, // row_index pointer
             col_pointer, // col_pointer pointer
             NC, _D, GE);

  dCreate_Dense_Matrix(&B, ls_size, nrhs, rhs_p, ls_size, DN, _D, GE);
  dCreate_Dense_Matrix(&X, ls_size, nrhs, tmp_p, ls_size, DN, _D, GE);
  perm_r = new int[ls_size];
  perm_c = new int[ls_size];

  /*
   * Get column permutation vector perm_c[], according to permc_spec:
   *   permc_spec = 0: use the natural ordering
   *   permc_spec = 1: use minimum degree ordering on structure of A'*A
   *   permc_spec = 2: use minimum degree ordering on structure of A'+A
   */
  get_perm_c(permc_spec, &A, perm_c);

  etree = new int[ls_size];
  factor_param_t *ftp = NULL;
  char fact, equed, trans, refact;
```

```
  fact      = 'E';
  equed     = 'B';
  trans     = 'N';
  refact    = 'N';
  double recip_pivot_growth;
  double rcond;
  // These are vectors of 1 element (nrhs elements)
  double ferr(0), berr(0);
  mem_usage_t mem_usage;
  int info;
  dgssvx(&fact, &trans, &refact, &A, ftp, perm_c, perm_r, etree, &equed, r, c,
     &L, &U, work, lwork, &B, &X, &recip_pivot_growth, &rcond,
     &ferr, &berr, &mem_usage, &info);
  // cout << "equed: " << equed << endl;
  //cout << "recip_pivot_growth = " << recip_pivot_growth << endl;
  //sprintf(msg, "1 / Condition number = %g", rcond);
  //report(MESSAGE, msg);
  //sprintf(msg, "info = %d", info);
  //report(MESSAGE, msg);
  //sprintf(msg, "ferr = %g", ferr);
  //report(MESSAGE, msg);
  //sprintf(msg, "berr = %g", berr);
  //report(MESSAGE, msg);

  if (info)
    report(FATAL, "There was a problem factoring the MNAM.");

  NCformat *Ustore;
  SCformat *Lstore;
  Lstore = (SCformat *) L.Store;
  Ustore = (NCformat *) U.Store;
  // sprintf(msg, "No of nonzeros in factor L = %d", Lstore->nnz);
  //report(MESSAGE, msg);
  //sprintf(msg, "No of nonzeros in factor U = %d", Ustore->nnz);
  //report(MESSAGE, msg);
  //sprintf(msg, "No of nonzeros in L+U = %d",
  //  Lstore->nnz + Ustore->nnz - ls_size);
//report(MESSAGE, msg);
  // sprintf(msg, "L\\U MB %.3f\ttotal MB needed %.3f\texpansions %d",
  //  mem_usage.for_lu/1e6, mem_usage.total_needed/1e6,
  //  mem_usage.expansions);
  //report(MESSAGE, msg);
  fflush(stdout);

  // This is not used anymore
  Destroy_SuperMatrix_Store(&X);
  superLU = true;

} void SVTran4::freeSuperLU() {


  assert(superLU);
  // Free vectors
  delete [] etree;
  delete [] perm_c;
  delete [] perm_r;
  // Free supermatrix structures
  Destroy_SuperMatrix_Store(&A);
  Destroy_SuperMatrix_Store(&B);
  // Free L and U matrices
  if ( lwork >= 0 ) {
    Destroy_SuperNode_Matrix(&L);
    Destroy_CompCol_Matrix(&U);
  }
  delete [] col_pointer;
  delete [] row_index;
  delete [] elem_val;
  superLU = false;
```

```
}

void SVTran4::f1(DenseVector& s1, DenseVector& v1l) {

  assert(superLU);
  //  do diag(r)*B
  ele_mult(s1, r_v, rhs_v);
  StatInit(sp_ienv(1), sp_ienv(2));
  // Solve using triangular solve for speed
  char trans = 'N';
  int info;
  dgstrs(&trans, &L, &U, perm_r, perm_c, &B, &info);
  if (info)
    cerr << "Warning: info = " << info << endl;

  // Premultiply rhs by T' (also premultiply by diag(c))
  for (int k=0; k < n_states; k++) {
    v1l[k] = zero;
    if (T(0,k)) {
      int tmpidx(T(0,k) - 1);
      v1l[k] += rhs_p[tmpidx] * c[tmpidx];
    }
    if (T(1,k)) {
      int tmpidx((T(1,k) - 1));
      v1l[k] -= rhs_p[tmpidx] * c[tmpidx];
    }
  }
  StatFree();
  return;
}


void SVTran4::buildMsv() {
  // This Msv is actually -Msv to avoid having to invert the currents sign
  Msv = DenseMatrix(n_states, n_states);
  // Clear input vector
  DenseVector::iterator ii = tmp_v.begin();
  DenseVector::iterator iend = tmp_v.end();
  for (; ii != iend; ++ii)
    *ii = zero;

  for (int i=0; i < n_states; i++) {
    // Build by columns
    if (T(0,i))
      tmp_v[T(0,i) - 1] -= one;
    if (T(1,i))
      tmp_v[T(1,i) - 1] += one;
    // Solve linear system and reduce
    f1(tmp_v, v1l);
    // Copy result to Msv column
    for (int j=0; j < n_states; j++)
      Msv(j,i) = v1l[j];
    // Clear vector for next iteration
    if (T(0,i))
      tmp_v[T(0,i) - 1] = zero;
    if (T(1,i))
      tmp_v[T(1,i) - 1] = zero;

  }
  // print_all_matrix(Msv);
  return;
}


void SVTran4::func_ev(double* X, double* F) {
  // Evaluate first the time-domain elements.
  updateVInl(X);
  // Tell tdsv that the first evaluation is already completed.
```

```
  tdsv->clearflag();
  // get nonlinear current and voltages vector
  DoubleVector& vnl = cVnl->getCurrent();
  DoubleVector& inl = cInl->getCurrent();
  if (use_msv) {
    ExtVector itmp1(&(inl[0]), n_states);
    ExtVector vtmp1(&(vnl[0]), n_states);
    add(itmp1, scaled(vtmp1, -gcomp), itmp2);
    mult(Msv, itmp2, vl1);
  }
  else {
    // Calculate error function
    DenseVector::iterator ii = tmp_v.begin();
    DenseVector::iterator iend = tmp_v.end();
    for (; ii != iend; ++ii)
      *ii = zero;
    for (int i=0; i < n_states; i++) {
      double itmp = inl[i] - gcomp * vnl[i];
      if (T(0,i))
    tmp_v[T(0,i) - 1] -= itmp;
      if (T(1,i))
    tmp_v[T(1,i) - 1] += itmp;
    }
    f1(tmp_v, vl1);
  }
  for (int i=0; i < n_states; i++)
    F[i] = ssv[i] + vl1[i] - vnl[i];
}

void SVTran4::jacobian(double* X, DoubleMatrix& J) {
  // Set current state variable values.
  for (int j=0; j < n_states; j++)
    cX->getCurrent()[j] = X[j];

  // Number of elements
  int n_elem = elem_vec.size();
  int i = 0;
  DoubleMatrix& Ju = tdsv->getJu();
  DoubleMatrix& Ji = tdsv->getJi();

  if (use_msv) {
    // Go through all the nonlinear elements
    for (int k = 0; k < n_elem; k++) {
      int cns = elem_vec[k]->getNumberOfStates();
      // Set base index in interface object
      tdsv->setIBase(i, cns);
      // Call element evaluation
      elem_vec[k]->deriv_svTran(tdsv);

      for (int l=0; l < n_states; l++)
    for (int j=0; j < cns; j++) {
      J(l, i+j) = zero;
      for (int n=0; n < cns; n++)
        J(l, i+j) += Msv(l, i+n) * (Ji(n, j)- gcomp * Ju(n,j));
    }

      for (int j=0; j < cns; j++)
    for (int n=0; n < cns; n++)
      J(i+j, i+n) -= Ju(j, n);

      i += cns;
    }
  }
  else {
    // Go through all the nonlinear elements
    for (int k = 0; k < n_elem; k++) {
      int cns = elem_vec[k]->getNumberOfStates();
      // Set base index in interface object
```

```
      tdsv->setIBase(i, cns);
      // Call element evaluation
      elem_vec[k]->deriv_svTran(tdsv);

      for (int j=0; j < cns; j++) {
    // Evaluate the linear contribution
    DenseVector::iterator ii = tmp_v.begin();
    DenseVector::iterator iend = tmp_v.end();
    for (; ii != iend; ++ii)
      *ii = zero;
    for (int n=0; n < cns; n++) {
      int ipn = i + n;
      if (T(0,ipn))
        tmp_v[T(0,ipn) - 1] -= (Ji(n, j)- gcomp * Ju(n,j));
      if (T(1,ipn))
        tmp_v[T(1,ipn) - 1] += (Ji(n, j)-  gcomp * Ju(n,j));
    }
    f1(tmp_v, vl1);
    for (int n=0; n < n_states; n++)
      J(n, i+j) = vl1[n];
    // add nonlinear contribution
    for (int n=0; n < cns; n++) {
      J(i+n, i+j) -= Ju(n, j);
    }
      }
      i += cns;
    }
  }
}



void SVTran4::updateVInl(double* x_p) {
  // Set current state variable values.
  for (int j=0; j < n_states; j++)
    cX->getCurrent()[j] = x_p[j];

  // Number of elements
  int n_elem = elem_vec.size();
  int i = 0;
  // Go through all the nonlinear elements
  for (int k = 0; k < n_elem; k++) {
    int cns = elem_vec[k]->getNumberOfStates();
    // Set base index in interface object
    tdsv->setIBase(i, cns);
    // Call element evaluation
    elem_vec[k]->svTran(tdsv);
    i += cns;
  }
}

void SVTran4::updateU(DenseVector& s1) {
  copy(s1, tmp_v);
  DoubleVector& inl = cInl->getCurrent();
  DoubleVector& vnl = cVnl->getCurrent();
  for (int i=0; i < n_states; i++) {
   double itmp = inl[i] - gcomp * vnl[i];
    if (T(0,i))
      tmp_v[T(0,i) - 1] -= itmp;
    if (T(1,i))
      tmp_v[T(1,i) - 1] += itmp;
  }

  assert(superLU);
  //  do diag(r)*B
  ele_mult(tmp_v, r_v, rhs_v);
  StatInit(sp_ienv(1), sp_ienv(2));
  // Solve using triangular solve for speed
```

```
  char trans = 'N';
  int info;
  dgstrs(&trans, &L, &U, perm_r, perm_c, &B, &info);
  if (info)
    cerr << "Warning: info = " << info << endl;

  ExtVector u_v(&(cU->getCurrent()[0]), ls_size);
  ele_mult(rhs_v, c_v, u_v);
}

void SVTran4::doOutput() {
  // First check if the result matrices contain any data
  assert(cX);

  report(MESSAGE, "--- Writing output vectors ...");
  int out_size;
  if(nt>list_size)
  out_size= list_size-1;
  else
  out_size= nt-1;


//   out_size = (nt < list_size) ? list_size - 1 : nt - 1;

  // Allocate and copy global time vector
  allocTimeV_P(out_size);
  int tindex;
  tindex=0;


  for ( tindex = 0; tindex < out_size; tindex++)
    TimeV_P[tindex] = cTime->getPrevious(out_size - tindex )[0];

  // Temporary vectors
  DoubleVector tmp_x(out_size);
  DoubleVector tmp_i(out_size);
  DoubleVector tmp_u(out_size);

  // Now fill currents and port voltages of time domain devices.
  int current_ns;
  int n_elem = elem_vec.size();   // Number of elements
  int i=0;
  for (int k=0; k < n_elem; k++) {

    current_ns = elem_vec[k]->getNumberOfStates();
    for(int j=0; j< current_ns; j++) {

      // For the current, decompensate while copying.
      for (int tindex=0; tindex < out_size; tindex++) {
    tmp_x[tindex] = cX->getPrevious(out_size - tindex)[j+i];
    tmp_i[tindex] = cInl->getPrevious(out_size - tindex)[j+i];
    tmp_u[tindex] = cVnl->getPrevious(out_size - tindex)[j+i];
      }
      elem_vec[k]->getElemData()->setRealX(j, tmp_i);
      elem_vec[k]->getElemData()->setRealI(j, tmp_i);
      elem_vec[k]->getElemData()->setRealU(j, tmp_u);
    }
    i += current_ns;
  }

  if (savenode) {
    // For each terminal, assign voltage vector
    Terminal* term = NULL;
    cir->setFirstTerminal();
    while((term = cir->nextTerminal())) {
      // Get MNAM index
      if (term->getRC()) {
    // Remember that MNAM indices begin at 1
```

```
    i = term->getRC() - 1;
    for (int tindex=0; tindex < out_size; tindex++)
      tmp_u[tindex] = cU->getPrevious(out_size - tindex)[i];
      }
      else
    // This is a reference terminal
    tmp_u = zero;

      // Set terminal vector
      term->getTermData()->setRealV(tmp_u);
    }
  }
}
```

A.4 Header File of New Time Step Algorithm

```
// This may look like C code, but it is really -*- C++ -*-
//
// time marching transient analysis with variable timestep
// Author:
//          Shubha Vijaychand
//

#ifndef SVTran4_h #define SVTran4_h 1


#include "Analysis.h" #include "OFunction.h" #include
"CircVector.h" #include "CircDouble.h" #include "transim_mtl.h"
extern "C" { #undef _S #undef _C #include SUPERLU_P"/dsp_defs.h"
#include SUPERLU_P"/util.h"
       }

// Main class definition follows
class SVTran4 : public Analysis, public OFunction { public:

  SVTran4();

  ~SVTran4();

  // (from Analysis)
  // The main analysis routine.
  virtual void run(Circuit* cir);

  // (from OFunction)
  // Evaluate error function vector F given X
  virtual void func_ev(double* X, double* F);
  // Evaluate Jacobian
  virtual void jacobian(double* X, DoubleMatrix& J);

  // Write output vectors
  void doOutput();

private:

  void superLUFactor(SparseMatrix& M);
  void freeSuperLU();
  void f1(DenseVector& s1, DenseVector& v1l);
  void updateVInl(double* x_p);
  void updateU(DenseVector& s1);
  void buildMsv();

  Circuit* cir;
  int n_states;
  IntMatrix T;
  ElementVector elem_vec;
  TimeDomainSV *tdsv;
  int ls_size;
  int nt;
  int n_tsteps;
  int list_size;

  DenseVector ssv, vl1,itmp2,  tmp_v, rhs_v, c_v, r_v;
  CircVector *cU;
  CircVector *cX;
  CircVector *cVnl;
  CircVector *cInl;
  CircVector *cTime;
  DenseMatrix Msv;

  // SuperLU variables
  SuperMatrix A, L, U, B, X;
  int *perm_r;
```

```
  int *perm_c;
  int *etree;
  double *r, *c;
  double *rhs_p;
  void *work;
  int lwork;
  bool superLU;
  double* elem_val;
  int* row_index;
  int* col_pointer;

  // ------------------- Parameter-related variables
  // Analysis information
  static ItemInfo ainfo;
  // Number of parameters of this element
  static const unsigned n_par;
  // Analysis parameters
  double h, tf, nst,gcomp, abstol, reltol,tol;
  int deriv, int_method, permc_spec, out_steps;
  bool use_msv, savenode;
  // Parameter information
  static ParmInfo pinfo[];
};


#endif
```

A.5 Euler Deratives

```
#include "Euler.h"

NLEuler::NLEuler(CircVector*& cX,CircVector*& cTime, double& h) {
  this->cX = cX;
  this->cTime=cTime;
  a = one / h;
  a2 = a * a;
}


NLEuler::NLEuler(CircVector*& cX, double& h) {
  this->cX = cX;
  this->cTime=NULL;
  a = one / h;
  a2 = a * a;
}


double NLEuler::derivX(const int& index) {
  return (cX->getCurrent()[index] - cX->getPrevious(1)[index]) * a;
}

double NLEuler::deriv2X(const int& index) {
  return (cX->getCurrent()[index] - 2. * cX->getPrevious(1)[index]
      + cX->getPrevious(2)[index]) * a2;
}

double NLEuler::getdx_dtFactor() {
  return a;
}

double NLEuler::getd2x_dt2Factor() {
  return a2;
}

double NLEuler::delayX(const int& index, const double& t) {
  if(cTime==NULL)
  {
```

```
  // First, find where to interpolate
  double delta_step = t * a;
  int time_idx = int(delta_step);
  delta_step -= double(time_idx);

  const double& v2 = cX->getPrevious(time_idx)(index);
  const double& v1 = cX->getPrevious(time_idx + 1)(index);
  // Interpolate between v1 and v2
  return v2 + (v1 - v2) * delta_step;
} else
  {
   double time_delay = cTime-> getCurrent()(0) ;
   //define some temporary variables
   double prev_temp, temp ;
   int time_idx;
   int  i=1;
  do
    {
      temp=cTime->getPrevious(i)(0);
      if(temp-time_delay>0)
      prev_temp=temp;
      i++;
    }
  while(temp-time_delay>0);
  time_idx = i;
  double delta_step=t/(temp-prev_temp);
  delta_step -=int(delta_step);
  const double& v2 = cX->getPrevious(time_idx)(index);
        const double& v1 = cX->getPrevious(time_idx + 1)(index);
  // Interpolate between v1 and v2

       return v2 + (v1 - v2) * delta_step;
  }
}

double NLEuler::getDelayXFactor(const double& t) {
  if(cTime==NULL)
    {
  // First, find where to interpolate
  double delta_step = t * a;
  int time_idx = int(delta_step);
  delta_step -= double(time_idx);
  if (time_idx)
    return zero;
  else
    return one - delta_step;
    }
else
  {
// First, find where to interpolate
double time_delay= cTime->getCurrent()(0) ;
  //define some temporary variables
  double prev_temp, temp ;
  int time_idx;
  int  i=1;
  do
    {
      temp=cTime->getPrevious(i)(0);
       if(temp-time_delay>0)
       prev_temp=temp;
       i++;
      }
  while(temp-time_delay>0);
  time_idx= i;
 double delta_step=t/(temp-prev_temp);
 delta_step -= int(delta_step);
 if (time_idx)
   return zero;
```

```
else
  return one- delta_step;
  }
}

void NLEuler::getNSamples(int &nx, int &ndx) {
  nx = 1;
  ndx = 0;
}

double NLEuler::deriv(double* x, double * dx) {
  return a * (x[0] - x[1]);
}

void NLEuler::changeStep(const double& h) {
  a = one / h;
  a2 = a * a;
}

LEuler::LEuler(CircVector*& cU, TimeMNAM* mnam) {
  this->cU = cU;
  this->mnam = mnam;

  size = mnam->getDim();
  s2 = DenseVector(size);
  a = zero;
}

void LEuler::buildMd(SparseMatrix& M, const double& h) {
  a = one/h;
  mnam->getMatrices(M1, M1p);
  copy(M1, M);
  add(scaled(M1p, a), M);
}

void LEuler::buildSf(DenseVector& s1, const double& ctime) {
  assert(a);
  mnam->getSource(ctime, s2);
  ExtVector u_n(&(cU->getPrevious(1)[0]), size);
  //  cout << cU->getPrevious(1) << endl;
  mult(M1p, scaled(u_n, a), s2, s1);
}



void LEuler::changeStep(const double& h) {
  a = one / h;
}
```

A.6 Trapezoidal Derivatives

```
#include "Trapezoidal.h"

NLTrapezoidal::NLTrapezoidal(CircVector*& cX,CircVector*& cTime,
double& h) {
  this->cX = cX;
  this->cTime=cTime;
  a = 2. / h;
  a2 = a * a;
  da = 2. * a;
  xpn = DoubleVector(cX->getCurrent().size(), zero);
  xsn = DoubleVector(cX->getCurrent().size(), zero);
}

NLTrapezoidal::NLTrapezoidal(CircVector*& cX, double& h) {
  this->cX = cX;
  this->cTime=NULL;
  a = 2. / h;
  a2 = a * a;
  da = 2. * a;
  xpn = DoubleVector(cX->getCurrent().size(), zero);
  xsn = DoubleVector(cX->getCurrent().size(), zero);
}

double NLTrapezoidal::derivX(const int& index) {
  return (cX->getCurrent()[index] - cX->getPrevious(1)[index]) * a
    - xpn[index];
}

double NLTrapezoidal::deriv2X(const int& index) {
  return (cX->getCurrent()[index] - cX->getPrevious(1)[index]) * a2
    - da * xpn[index] - xsn[index];
}

double NLTrapezoidal::getdx_dtFactor() {
  return a;
}

double NLTrapezoidal::getd2x_dt2Factor() {
  return a2;
}
//this is the new delay routine for the Non linear trapezoidal technique

double NLTrapezoidal::delayX(const int& index, const double& t) {
// first we need to find the where in the circular matrix the state variable required is stored
  //double time_delay;
  if(cTime==NULL)
    {
      double delta_step = t * a / 2.;
      int time_idx = int(delta_step);
      delta_step -= double(time_idx);
      const double& v2 = cX->getPrevious(time_idx)(index);
      const double& v1 = cX->getPrevious(time_idx + 1)(index);
      // Interpolate between v1 and v2
      return v2 + (v1 - v2) * delta_step;
    }
else {
   double time_delay = cTime-> getCurrent()(0) ;
   //define some temporary variables
   double prev_temp, temp ;
   int time_idx;
   int  i=1;
  do
    {
      temp=cTime->getPrevious(i)(0);
      if(temp-time_delay>0)
      prev_temp=temp;
```

```
      i++;
    }
  while(temp-time_delay>0);
  time_idx = i;
  double delta_step=t/(temp-prev_temp);
  delta_step -=int(delta_step);
  const double& v2 = cX->getPrevious(time_idx)(index);
        const double& v1 = cX->getPrevious(time_idx + 1)(index);
  // Interpolate between v1 and v2

       return v2 + (v1 - v2) * delta_step;
} }
      /*
      double NLTrapezoidal::delayX(const int& index, const double& t)
      {
  // First, find where to interpolate

      double delta_step = t * a / 2.;
      int time_idx = int(delta_step);
      delta_step -= double(time_idx);
      const double& v2 = cX->getPrevious(time_idx)(index);
      const double& v1 = cX->getPrevious(time_idx + 1)(index);
  // Interpolate between v1 and v2
       return v2 + (v1 - v2) * delta_step;
}
      */

double NLTrapezoidal::getDelayXFactor(const double& t) {
  if(cTime==NULL)
{
 // First, find where to interpolate
  double delta_step = t * a / 2.;
  int time_idx = int(delta_step);
  delta_step -= double(time_idx);
  if (time_idx)
    return zero;
  else
    return one - delta_step;
} else {
// First, find where to interpolate
double time_delay= cTime->getCurrent()(0) ;
  //define some temporary variables
  double prev_temp, temp ;
  int time_idx;
  int  i=1;
  do
    {
      temp=cTime->getPrevious(i)(0);
       if(temp-time_delay>0)
       prev_temp=temp;
       i++;
      }
  while(temp-time_delay>0);
  time_idx= i;
 double delta_step=t/(temp-prev_temp);
 delta_step -= int(delta_step);
 if (time_idx)
   return zero;
else
  return one- delta_step;
} }

/* double NLTrapezoidal::getDelayXFactor(const double& t) {
  // First, find where to interpolate
  double delta_step = t * a / 2.;
  int time_idx = int(delta_step);
  delta_step -= double(time_idx);
  if (time_idx)
```

```
    return zero;
  else
    return one - delta_step;
} */ void NLTrapezoidal::changeStep(const double& h) {
  a = 2. / h;
  a2 = a * a;
  da = 2. * a;
}

void NLTrapezoidal::getNSamples(int &nx, int &ndx) {
  nx = 1;
  ndx = 1;
}

double NLTrapezoidal::deriv(double* x, double* dx) {
  return a * (x[0] - x[1]) - dx[1];
}

void NLTrapezoidal::store() {
  // Update previous derivative vectors
  int iend = xpn.size();
  for (int i=0; i < iend; i++) {
    xsn[i] = (cX->getCurrent()[i] - cX->getPrevious(1)[i]) * a2
      - da * xpn[i] - xsn[i];
    xpn[i] = a * (cX->getCurrent()[i] - cX->getPrevious(1)[i]) - xpn[i];
  }
  return;
}

LTrapezoidal::LTrapezoidal(CircVector*& cU, TimeMNAM* mnam) {
  this->cU = cU;
  this->mnam = mnam;

  size = mnam->getDim();
  s2 = DenseVector(size);
  upn = DenseVector(size, zero);
  tmp1v = DenseVector(size);
  a = zero;
}

void LTrapezoidal::buildMd(SparseMatrix& M, const double& h) {
  a = 2. / h;
  mnam->getMatrices(M1, M1p);
  copy(M1, M);
  add(scaled(M1p, a), M);
}

void LTrapezoidal::buildSf(DenseVector& s1, const double& ctime) {
  assert(a);
  mnam->getSource(ctime, s2);
  ExtVector u_n(&(cU->getPrevious(1)[0]), size);
  add(upn, scaled(u_n, a), tmp1v);
  //  cout << cU->getPrevious(1) << endl;
  mult(M1p, tmp1v, s2, s1);
}

void LTrapezoidal::store() {
  // Update previous derivative vector
  for (int i=0; i < size; i++) {
    upn[i] = a * (cU->getCurrent()[i] - cU->getPrevious(1)[i]) - upn[i];
  }
  return;
}
```

Bibliography

[1] Vlach and Singhal, *Computer Methods for Circuit Analysis and Design*, Chapman and Hall,1994.

[2] Carsten Mundt, *Transient Analysis of Linear Circuits in SPICE- Limitations, Errors and Dynamic Range*, ECE 718 Project Report , 1995.

[3] Roldan Pozo, *MV++ User's Manual* ,1995.

[4] M. B. Steer, "Transient and Steady-State Analysis of Nonlinear RF and Microwave Circuits," ECE 718 class notes, 2001.

[5] Kenneth S.Kundert and Ian H. Clifford , *Achieving Accurate Results with a circuit simulator* , IEE Colloquim, 1993.

[6] Carlos E.Christofferson, *State varible harmonic balance of quasi optical power combining system*, Master thesis dissertation, 1998.

[7] Mete Ozkar ,*Transient Analysis of spatially disturbuted micorwave circuits using convolution and state variables*, Masters thesis dissertation, 1998.

[8] Griewank, Juedes and Utke, *Adol-C: A Package for the Automatic Differenciation of Algorithms Written in C/C++*, Version 1.7, Sep 1996.

[9] christoffersen, *Global modeling of nonlinear microwave circuits*, Phd thesis dissertation , 2001.

[10] Christoffersen, Usman Mughal and M.B. Steer *Object Oriented Microwave Circuit Simulation*, Int J on RF and Microwave Computer Aided Engineering, Volume 10, Issue 3,2000, pp.164-172.