The Hong Kong Polytechnic University

Department of Electronic and Information Engineering

MODELLING AND SIMULATION OF A NON-LINEAR CIRCUIT

by

Lei Weng Hong

A thesis submitted in partial fulfillment of the requirements

for the degree of

Master of Philosophy

2001

(Submitted July 2000)

Abstract of thesis entitled 'Modelling and Simulation of a Non-linear Circuit'
submitted by Lei Weng Hong
for the degree of Master of Philosophy
at The Hong Kong Polytechnic University in July 2000

ABSTRACT

Device Represented of Averaged Fourier Transforms (DRAFT) has been proved to be fruitful in generating SPICE macromodels for series, parallel and series-parallel resonant converters. However, The macromodels are inaccurate when state variables are not sinusoidal and the more accurate version of the models lead to convergence problems in a simulator. In this thesis, a novel macromodel and a new SPICE simulation program are presented. An algorithm of the new macromodel is extended from the averaged Fourier transforms. The algorithm uses actual solutions of state variables. A SPICE macromodel for a resonant converter is generated by the algorithm. The simulated results of the new macromodel are much more accurate than those given by the previous macromodels. A new nonlinear dependent source is proposed to enhance SPICE. The new source can solve equations of the form $f(x)=0$ with no convergence problems. The implementation of the new source is presented. A macromodel for a resonant converter is taken as an example to demonstrate the application of the new source. In addition, a new SPICE simulation program is written based on SPICE3 using object-oriented programming paradigms. The internal structure of the new SPICE simulation program is outlined. The object-oriented programming paradigms provide a set of features to simplify the internal structure of the new SPICE. When comparing with SPICE3, the new SPICE has many advantages for further development and maintenance.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I am deeply indebted to Dr. S. C. Wong, my supervisor, who has given me much insight into the area of my research work. Without his patient supervision and guidance, this thesis and all other related publications would not have been completed.

I appreciate the financial support of the Hong Kong Polytechnic University by way of Research Studentship. Also, I thank UC Berkeley for giving me free source codes of the SPICE3f3 program.

Besides, I would like to express my gratitude and sincere thanks to my co-supervisor, Professor Y. S. Lee, who has been a constant source of encouragement and support during these years of study. My grateful thanks are due to Mr. Shen Xu for his noteworthy amount of energy and contribution to the development of the new SPICE program.

Finally, I wish to thank my friends and my family who over the years have inspired and helped me.

Chapter 1

Introduction

Virtually all electronic products require power supplies. The demand for higher reliability and efficiency coupled with compact size has tightened the requirements on power supply circuit designs. To meet these challenges, the switching mode power supply (SMPS) makes use of electronic switches, inductors and capacitors to process electric power. Since ideal switches, inductors and capacitors do not dissipate power, the SMPS can be designed to have high efficiency and small size. The heart of a SMPS is a DC-DC converter, which accepts a DC input and produces a controlled DC output. The conventional topology of a switching-mode DC-DC converter is largely of the Pulse-Width-Modulation (PWM) type. The efficiency of a well-designed and well built PWM converter typically ranges from 80% to 85%. Recently, the development of high frequency resonant power converters has become more significant. The resonant type converters process energy in a sinusoidal fashion with the switches commutated under zero-current/voltage that results in a much higher internal converter frequency (50kHz to 3MHz), smaller LC tank circuit and higher efficiency (greater than 90%). However, the circuit complexity of the resonant converters makes them less attractive to designers. This is because of the highly nonlinear nature of the switching circuitry and the lack of the sophisticated support like computer-aided design tools for analysis and design. The periodic switching actions lead to abrupt changes in both voltage and current in the converters. Such nonlinear operation has invalidated the direct application of the general circuit simulator like SPICE [1] that is the industry-standard circuit simulation software. SPICE can only perform full circuit transient analysis. Thousands of time steps within a

single switching period will be analyzed by SPICE. However, circuit designers are often interested in the steady-state response of SMPS. This makes the simulation time hours or even days for a single transient analysis. To analyze fully the system response, it may be necessary to generate hundreds of transient runs. With this lengthy simulation time, simulation of system response using full-transient analysis is often considered impractical.

I. MODELLING, SIMULATION AND ANALYSIS OF RESONANT CONVERTERS

Over the past decade, a good deal of research has been devoted to "linearizing" the physical systems. The sampled-data method [2] is an algorithm to calculate numerically the transient as well as steady state waveforms of state variables. Within a switching cycle, the values of the state variables at the end of a stage are calculated from those at the start of the stage. The calculated values of the state variables at the end of the stage will be used as initial values for the next stage. The process is then continued forward. We thus have an exact large-signal sampled-data description of the dynamics of a power converter circuit. By using the sampled-data method, there is a huge speed advantage over the conventional full circuit simulation as the switching details are being omitted. However, it is too specific and difficult to use. In the method of complex analysis [3,11,19], the tank variables are assumed sinusoidal and hence most of the devices can be replaced by complex impedances. The tank circuit is then modelled by its equivalent circuit of complex impedances. Other circuit components are replaced by devices of their averaged impedances. The Fourier series [4,10] approach adopts the same

calculation as complex analysis at each harmonic frequency. Due to the complexity of calculating the overall effect of different harmonics, the analysis is limited to the point of setting up a set of general equations. Instead, complex impedance analysis is used to complete the calculation. The state-space approach [9,21] is an accurate method of calculating the steady state solution of the state variables. However, due to the complexity of the approach, not all closed-form solutions of state variables can be found, and therefore the problem must be solved by numerical methods. The state-plane technique [12,25,31-32] is a technique to map the results of the state-space calculation onto a two-dimensional graph of state variables. The resulting graph is a symmetrical piecewise connected arcs of circles. The number of connected arcs equals the number of stages of the converter. However, the state variables cannot be solved analytically. The approaches mentioned above cannot avoid the tedious mathematical manipulations. None of the methods can be easily implemented in a general circuit simulator like SPICE. Recently, the macromodel for resonant converters [5] is an averaged model that allows performing DC, AC and transient analyses in SPICE at a speed several orders of magnitude faster than the full-transient simulation. The model uses the sinusoidal averaging of state variables. Both the tank inductor current and tank capacitor voltage are assumed sinusoidal. This assumption allows the nonlinear time variant nature of the switching circuitry to be transformed into a time invariant equivalent circuit. The model that results in a set of "linearized" equations can be easily implemented in the circuit simulator with the SPICE arbitrary dependent source. The power circuit designers can analyze the complete system response quickly at this functional model. However, the functional model is inaccurate when the tank state variables are no longer sinusoidal or

lack of simulation features that simulate equations having no closed-form solutions obstacles more accurate models being implemented into SPICE.

Since functional models can be conveniently used by engineers, and can give a fast physical insight into the static operation and dynamic behavior of the converters. Many circuit designers can use SPICE to simulate their designs at functional model level before building breadboards to reduce design time, cost, and increase the insight into important design issues. However, using functional models in the existing SPICE program usually results in convergence problems. Also, the SPICE program does not have suitable devices/features for SMPS circuit simulation (both in full circuit level and in functional level). When SMPS circuit designers want refinement of the SPICE program for tackling their problems at programming level, they become power users (programmers) of SPICE. Unfortunately, the modification of the program is a hard job for someone who is not familiar with the SPICE program structure. It is because of the size and the complexity of the internal structure of the program.

II. THE NEW MACROMODEL AND THE NEW SPICE SIMULATION PROGRAM

The purpose of this project is to develop an improved macromodel that allows all system level DC, AC and transient analyses to be carried out using the entire resonant converter as a circuit primitive, and a new SPICE simulation program with functional level simulation enhancement. In addition, the new SPICE simulation program should allow future development and modification easily.

The new macromodel applies the actual solutions of the state variables instead of the sinusoidal approximations. This approach can overcome the limitation of the existing macromodels and gives a more accurate functional model for resonant converters. In addition, a new nonlinear dependent source which does not exist in SPICE (and its variants) is developed for inputting the algebraic equations having no closed-form solutions directly via circuit netlist file. The new nonlinear dependent source can be implemented into the new SPICE simulation program, as well as the UC Berkeley SPICE3 program [1]. Moreover, the new program is written in object-oriented programming paradigms rather than the traditional procedure programming paradigms. It gives a clear hierarchy internal structure.

III. THESIS ORGANISATION

This thesis is organized in six chapters. Chapter 2 reviews the existing macromodels for resonant converters, and discusses the current limitation. Chapter 3 introduces the new improved macromodel. An example implementation of the new macromodel for a half-bridge series resonant converter is derived and the verification of the new macromodel with the full circuit simulation is shown. Chapter 4 deals with the enhancement of the SPICE circuit simulator (macromodel simulator). A new nonlinear dependent source for functional level simulation is developed. An example of application of the new source arriving from a macromodel for a resonant converter is discussed. Chapter 5 introduces a new SPICE simulation program. An overview of the internal structure of SPICE3 is presented. An outline of the basic structure of the new SPICE is given. A comparison of SPICE3 program structure and the new program structure is discussed. In concluding the

thesis, Chapter 6 discusses some aspects of the improvement of the new SPICE simulator. A hint for future development of functional model simulation is also provided.

Chapter 2

Review of Existing Macromodels for Resonant Converters

This chapter will focus on the method of the Device Representation of Averaged Fourier Transforms (DRAFT) macromodelling for resonant converters. The macromodels for series resonant converter [6], parallel resonant converter [7] and series-parallel resonant converter [8] have been developed. Such models give excellent results when using the sinusoidal approximation for the state variables of resonant converters with the mentioned constraints. They also can be incorporated with a control circuit to complete the design. The models can simulate in a general circuit simulator SPICE using all high level analyses: AC, DC and transient. Thus, macromodelling is a very useful tool for fast simulation of resonant converters. However, when the state variables are deviated from sinusoidal, the output of the models will deviate from the actual output of the converters and make the solutions of the models inaccurate. Therefore, this chapter will study in detail the operation of resonant converters, different modes of operation and the mathematical expressions for the converters. Then we will have a review of the method of DRAFT macromodelling and point out its limitations.

Section I illustrates the circuit operation and waveforms for an ideal half bridge series resonant converter. The equivalent circuit for the converter and the state space equations are presented. Then, section II describes the Device Representation of Averaged Transforms macromodels for resonant converters and the limitation of the models.

I. RESONANT CONVERTER

A resonant converter contains a resonant tank, chopper, rectifier and an output filter. The resonant tank is used to convert a chopped DC input from the chopper to produce a piecewise sinusoidal current before the current is rectified.

A. Series Resonant Converter

A half-bridge series resonant converter will be studied in this section. The circuit schematic of a half-bridge series-resonant converter is shown in figure 2-1. As the name implies, the resonant inductor (L) and the resonant capacitor (C) are connected in series. The inductor current is rectified by the diode bridge and filtered by a larger capacitor before being delivered to the output load.

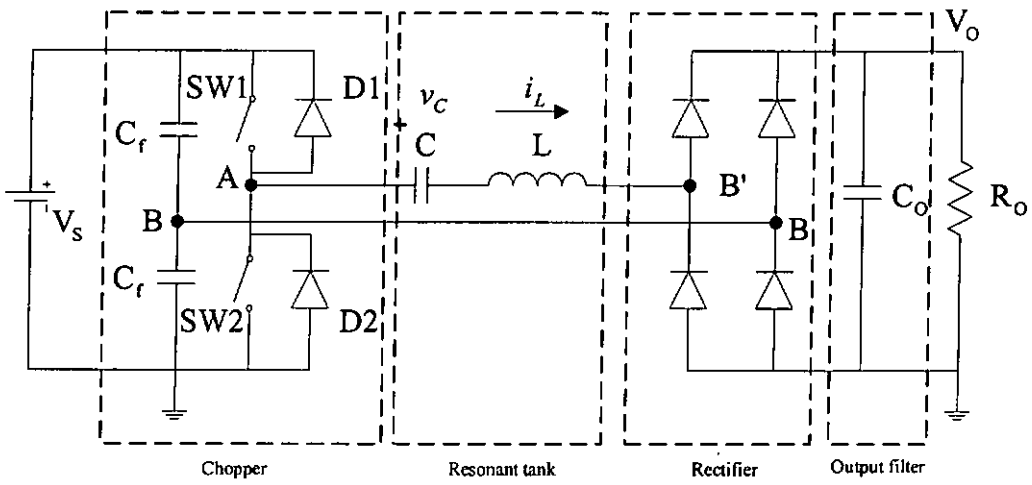


Figure 2-1, A half-bridge series resonant converter.

B. Circuit topologies

Firstly, the circuit can be divided into four parts: chopper, resonant tank, rectifier and output filter as shown in figure 2-1.

Chopper

The chopper shown in figure 2-1 is a half-bridge configuration with two switches which chop the line-input voltage (V_S), and two capacitors (C_f) which are used as a potential divider. The switches are normally power MOSFETS which are driven by two complementary square pulses with less than 50% duty cycle.

Resonant tank

The resonant tank is a series connection of the resonant inductor (L) and the resonant capacitor (C).

Rectifier and output filter

The AC current from the resonant tank shown in figure 2-1 is rectified by the full-wave bridge rectifier to a DC current followed by a filter capacitor (C_o) and then delivered to the load. The filter capacitor is usually very large. The load can be represented as a constant output voltage source (V_o). The RC time constant is chosen to be large enough that the capacitor voltage varies extremely slowly compared to the switching frequency.

C. Mode of operation

A typical output voltage versus switching frequency of the converter is given in figure 2-2, where f_o is the resonant frequency ($f_o = \frac{1}{2\pi\sqrt{LC}}$). In the steady state, the operation of the half-bridge series resonant converter basically has two modes depending on the switching frequency (f_s).

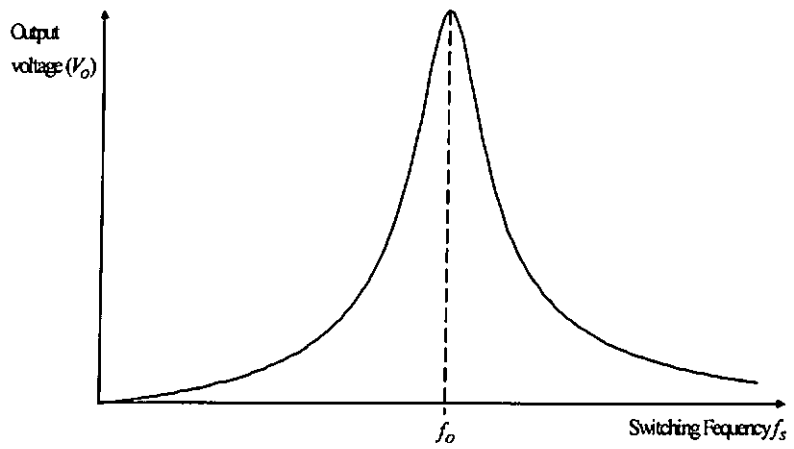


Figure 2-2, Output voltage versus switching frequency characteristic.

Below resonance ($f_o/2 < f_s < f_o$)

In this mode operation, state variables (v_C, i_L) have four stages of operation in one switching cycle.

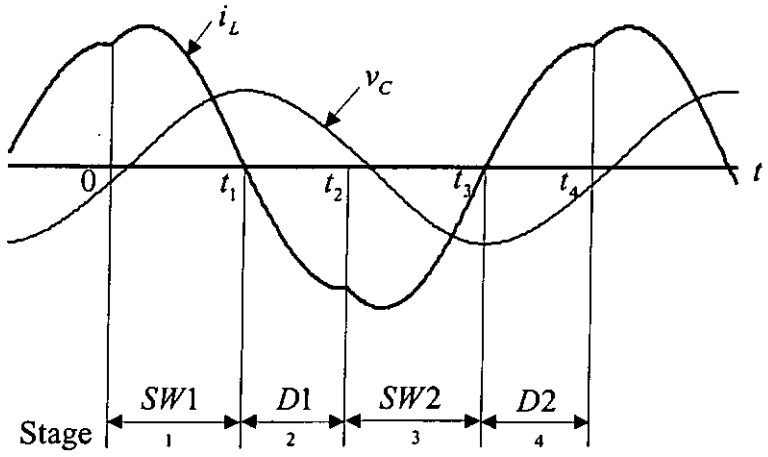


Figure 2-3, The inductor current and capacitor voltage waveform below resonance.

Stage 1 ($0 < t < t_1$)

The waveforms of the state variables are shown in figure 2-3. Before the start of a switching cycle, the resonant inductor current (i_L) is positive. When *SW1* is on at t=0, the

resonant inductor current (i_L) increases and reaches its peak amplitude. *SW1* remains conducting until it is naturally turned off when the resonant inductor (i_L) reduces to zero.

Stage 2 ($t_1 < t < t_2$)

When the resonant inductor current (i_L) reverses its direction as it feeds energy back to the input source ($V_S/2$), the anti-parallel diode (D_1) conducts. At the same time, $V_{B'B}$ reverses its polarity.

Stage 3 ($t_2 < t < t_3$)

When *SW2* switches on at time t_2 (the resonant inductor current (i_L) transfers from D_1 to *SW2*) and remains conducting until it is naturally turned off at t_3.

Stage 4 ($t_3 < t < t_4$)

When the resonant inductor current (i_L) reverse its direction and continuous to flow through D_2 , the converter enters stage 4. Stage 4 ends when *SW1* switches on again.

Above resonance ($f_s > f_o$)

There are four stages of operation in one switching cycle in this mode, as shown in figure 2-5.

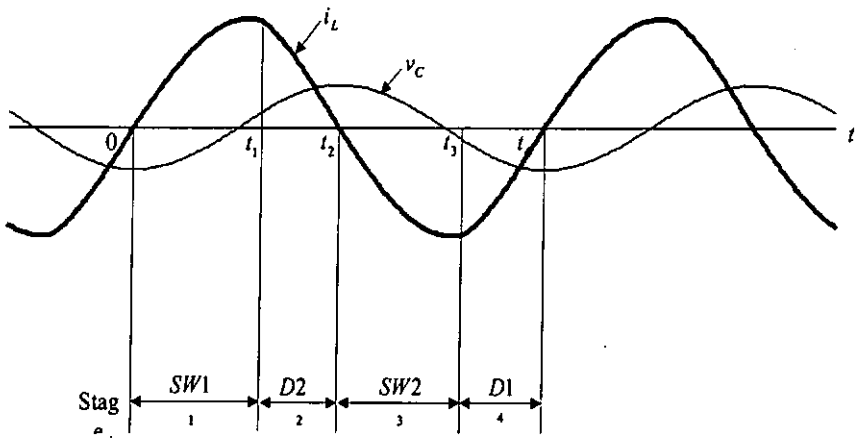


Figure 2-4, The inductor current and capacitor voltage waveform above resonance.

Stage 1 ($0< t <t_1$)

The waveforms of the state variables are shown in figure 2-4. The resonant inductor current (i_L) begins at zero, increases and reaches its peak amplitude where *SW1* is on.

Stage 2 ($t_1< t <t_2$)

When *SW1* is forced to switch off at time t_1, the resonant inductor current (i_L) continues to follow through D_2. D_2 starts to conduct as it is forward biased. Because of the large negative DC voltage applied across the *LC* resonant tank ($V_{AB'} = -\frac{V_S}{2} - V_O$) as shown in figure 2-5, the resonant inductor current (i_L) through D_2 goes to zero quickly.

Stage 3 ($t_2< t <t_3$)

SW2 is turned on just before time t_2, and begins to conduct when i_L reverses its direction and feeds energy back to the input source (V_S).

Stage 4 ($t_3 < t < t_4$)

When *SW2* is forced to switch off at time t_3, the resonant inductor current (i_L) continues to flow through the path along D_1. D_1 starts to conduct as it is forward biased until i_L goes back to zero.

D. Equivalent circuit of the half-bridge series-resonant converter

Direct analysis (transient, AC and DC analyses) of such a complicated circuit as shown in Figure 2-1 is almost impossible [5-9,21,26]. Therefore, we must make some simplifying assumptions first. The half-bridge series resonant converter with ideal devices is assumed. The output voltage across the capacitor (C_o) is assumed to be a DC voltage (V_o) without any ripple. The voltage drop of rectified diode is negligible. For the tank circuit, the output voltage (V_o) is reflected back to the rectifier input as $V_{B'B}$ ($V_{B'B} = \pm V_o$) where $V_{B'B}$ is equal to V_o if the tank circuit current (i_L) is positive, and $V_{B'B}$ is equal to $-V_o$ if i_L is negative. The input voltage (V_S) is divided equally between the two symmetrical input capacitors (C_f). They serve as the input sources during each half switching cycle. Moreover, the tank circuit current i_L and the tank capacitor voltage (v_C) are time varying at a speed comparable to the switching frequency. Let us refer to these tank circuit variables as "fast" variables. The output voltage V_O and input voltage V_S are varying much more slowly (more than an order of magnitude in terms of frequency) than the fast variables. They are referred to as "slow" variables. For the input circuit, the fast tank circuit current i_L is reflected back to the line input as a slow variable given by the averaging current $\overline{i_L}$. Similarly, the fast i_L is rectified to the output

filter and can be averaged to $\overline{|i_L|}$ at the output circuit. Therefore, the half-bridge series resonant converter can be represented by an equivalent circuit of figure 2-5. It is still a non-linear time-variant equivalent circuit.

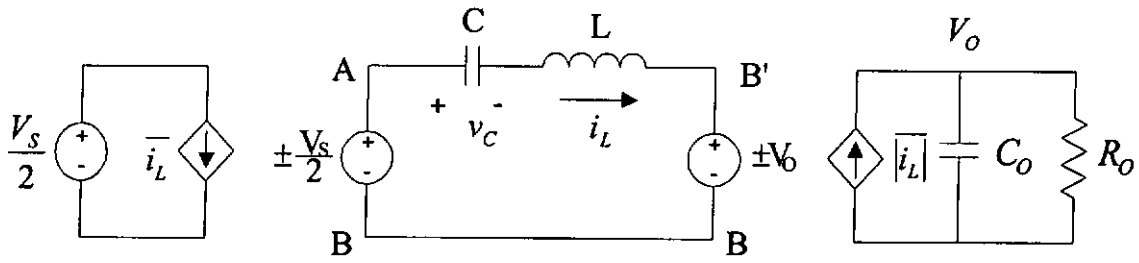


Figure 2-5, An equivalent circuit of a series resonant converter.

1. State-space equation

Using the equivalent circuit model shown in figure 2-5, the tank circuit state-space equations are given by

$$\frac{di_L}{dt} = \frac{-v_C + v_S - \text{sgn}(i_L)V_O}{L} \tag{2-1}$$

$$\frac{dv_C}{dt} = \frac{i_L}{C} \tag{2-2}$$

where $v_S = \pm\frac{V_S}{2}$.

In matrix form, the state-space equation is given by

$$\begin{pmatrix} \frac{di_L}{dt} \\ \frac{dv_C}{dt} \end{pmatrix} = \begin{pmatrix} -\frac{1}{L} & 0 \\ 0 & \frac{1}{C} \end{pmatrix} \begin{pmatrix} v_C \\ i_L \end{pmatrix} + \begin{pmatrix} \frac{-V_O \,\text{sgn}(i_L) + v_S}{L} \\ 0 \end{pmatrix} \tag{2-3}$$

Equation (2-3) can be rewritten in short form as follows. (The matrix and vector variables are typed in **boldface** and their elements are in regular font.)

$$\frac{d\mathbf{x}}{dt} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} \tag{2-4}$$

where $\mathbf{x} = (v_C \quad i_L)^T$ is the state vector,

$\mathbf{A} = \begin{pmatrix} -\frac{1}{L} & 0 \\ 0 & \frac{1}{C} \end{pmatrix}$ is the state matrix,

and $\mathbf{B}\mathbf{u} = \begin{pmatrix} \frac{-V_O \operatorname{sgn}(i_L) + v_S}{L} \\ 0 \end{pmatrix}$ is the input vector.

Since $v_S = \pm\frac{V_S}{2}$ and $\operatorname{sgn}(i_L) = \pm 1$ are discontinuous binary functions of time, equation (2-4) is a *highly nonlinear time-varying state-space equation* of the converter and it is difficult to solve analytically.

II. DEVICE REPRESENTATION OF AVERAGED FOURIER TRANSFORMS (DRAFT) MACROMODEL

This section presents an overview of Device Representation of Averaged Fourier Transforms (DRAFT) macromodelling [5-8]. The theory of the method is derived from the extended describing function method [20,23]. After extension of the extended describing function method, the macromodel allows AC, DC and transient analyses to be carried out in a fast, easy and familiar manner in a general circuit simulator SPICE. In section A, we will describe the extended describing function first.

A. Extended describing function

In this method, the tank state variables in equation (2-5a) are represented by a power series of sine function (Fourier series) with respect to an integral multiple of switching frequency. In equation (2-5a), I is the set of all integers. $x(t)$ is the tank state variable. ω_S is the angular switching frequency and $\langle x \rangle_k(t)$ is the k-th coefficient and is determined by equation (2-5b).

$$x(t) = \sum_{k \in I} \langle x \rangle_k(t) \exp(jk\omega_S t) \tag{2-5a}$$

$$\langle x \rangle_k(t) = \frac{\omega_S}{2\pi} \int_t^{t+\frac{2\pi}{\omega_S}} x(\tau) \exp(-jk\omega_S \tau) d\tau \tag{2-5b}$$

As mentioned in section I, the state-space equation of a resonant converter is given by

$$\frac{d\mathbf{x}}{dt} = \mathbf{Ax} + \mathbf{Bu} \tag{2-6}$$

where **x** is the state vector, **A** the state matrix, **u** the excitation vector and **B** the excitation matrix.

Using equation (2-5), the time-varying state-space equation (2-6) can be transformed to a *time invariant state-space equation* (2-7) for each multiple of switching frequency. This is done by applying the Averaged Fourier Transforms of the state-space equation for each harmonic component of the Fourier series. Equation (2-7) is complex. There are actually two real variable equations (2-8) that describe the system dynamics of the converter at each harmonic frequency.

$$\frac{d\langle \mathbf{x} \rangle_k}{dt} = -jk\omega_S \langle \mathbf{x} \rangle_k + \langle \mathbf{Ax} + \mathbf{Bu} \rangle_k \tag{2-7}$$

$$\frac{d\langle \mathbf{x} \rangle_k^r}{dt} = k\omega_S \langle \mathbf{x} \rangle_k^i + \langle \mathbf{Ax} + \mathbf{Bu} \rangle_k^r$$
$$\frac{d\langle \mathbf{x} \rangle_k^i}{dt} = -k\omega_S \langle \mathbf{x} \rangle_k^r + \langle \mathbf{Ax} + \mathbf{Bu} \rangle_k^i \quad (2\text{-}8)$$

To simplify the calculation, only the fundamental term of equation (2-8) is retained to complete the analysis for the converter.

B. DRAFT macromodel

The DRAFT macromodel makes use of equation (2-8) at the fundamental frequency to implement the macromodel in SPICE by using the dependent sources. The macromodel is a time-invariant averaged model and can perform all high level simulation provided by SPICE. It gives excellent results for state variables being almost sinusoidal. However, when the state variables deviate from sinusoidal waveforms, The DRAFT macromodel will give results that also deviate from the actual calculation [6-8].

C. The current limitations

Since the state variables are abrupt functions that should be fully described by the Fourier series with an infinite number of harmonics, the accuracy and order of the model depend on the number of harmonics retained in the averaged method [8]. An approach that is more general is to increase the number of harmonics used in the DRAFT macromodel. However, each harmonic term requires a separate implementation of equations (2-8). The contributions of all harmonics result in a complicated mathematical equation which gives rise to convergence problems. Therefore, the optimized result of the macromodel for resonant converters [5-8] is that only fundamental sinusoidal

approximation is used. In summary, the approach of DRAFT macromodelling based on the extended describing function method, has two main problems:

1. The macromodel is not general enough to be suitable in all operation conditions for resonant converters.
2. SPICE (macromodel simulator) does not have enough robust tools for simulating the model of non-linear circuit.

D. The new macromodel and new SPICE

Therefore, the following chapters in this thesis are devoted to develop a new macromodel and a new circuit simulation program (an object-SPICE). The new macromodel will allow all system level DC, AC and transient analyses to be carried out using the entire resonant converter as a circuit primitive. It can be applied accurately to the conventional frequency controlled resonant converters operated in all conduction modes. The object-SPICE accepts a description of a circuit and determines the quiescent operation point of the circuit, the time-domain response of the circuit and the frequency-domain response of the circuit. It can be used not only for the circuit-level simulation but also for the functional-mode level simulation, including some difficult functions for non-linear circuit modelling. In addition, the object-SPICE should allow future development and modification easily.

Chapter 3

The New Macromodel

The new macromodel applies the actual solutions of the state variables instead of the sinusoidal approximations. This approach can overcome the limitation of the existing macromodels and gives a more accurate functional model for resonant converters. A new transform (Averaged Piecewise Transform) will be developed in this chapter.

I. AVERAGED PIECEWISE TRANSFORM MACROMODEL

The transformation is based on three assumptions, namely piecewise approximation, symmetrical behavior and slow-fast assumption. We will describe them one by one.

1. Piecewise approximation

The tank state variables ($x(t)$) of a resonant power converter can be generally represented by a piecewise differentiable sinusoidal function with a natural frequency (ω_O) within a switching period ($\frac{2\pi}{\omega_S}$).

$$x(t) = X_i \sin[\omega_O(t-\tau_i)] \qquad t \in (t_{i-1}, t_i] \tag{3-1}$$

where $i = 1 \cdots 2n$ is the i-stage of the converter, $t_0 = 0$, $t_n = \frac{\pi}{\omega_S}$ and $t_{2n} = \frac{2\pi}{\omega_S}$.

X_i and τ_i have a period $\frac{\pi}{\omega_S}$, $x(t)$ has a period $\frac{2\pi}{\omega_S}$.

2. Symmetrical behavior

The new transform can be simplified by using the half-cycle symmetrical behavior of the state variables [2]. Therefore, only the first half switching cycle is considered and transformed.

3. Slow-fast assumption

The slow-fast assumption [5-8] of the state-space equation of the resonant converter is used.

- Slow variables are treated as constants with respect to the fast variables.
- Averaged fast variables can replace fast variables when solving for slow variables.
- Averaged fast variables are slow functions.

Therefore, the slow averaged input current ($\overline{i_S}$) of a converter shown in figure 2-5 is given by integrating the fast tank state variables (equation (3-1)) as follows:

$$\begin{aligned}\overline{i_S} &= \frac{\omega_S}{\pi}\int_0^{\frac{\pi}{\omega_S}} x(t)dt \\ &= \frac{\omega_S}{\omega_O \pi}\left\{\begin{array}{l} X_n + X_1 - X_1\cos(\tau_1\omega_O) - X_n\cos(\tau_n\omega_O - \dfrac{\omega_O\omega_S}{\pi}) \\ -\sum_{k=2}^{n-1} X_k + \sum_{l=2}^{n-1} X_l \cos((\tau_{l-1} - \tau_l)\omega_O) \end{array}\right\}\end{aligned} \tag{3-2}$$

where $\overline{i_S}$ is a function of X_i and τ_i. Both X_i and τ_i are slow. $x(t)$ is the tank circuit current i_L.

The slow averaged output current of the converter is given by equation (3-3).

$$\begin{aligned}\overline{|i_o|} &= \frac{\omega_S}{\pi}\int_0^{\frac{\pi}{\omega_S}}|x(t)|dt \\ &= \frac{\omega_S}{\omega_O\pi}\left\{\begin{aligned} &X_n - X_1 + X_1\cos(\tau_1\omega_O) - X_n\cos(\tau_n\omega_O - \frac{\omega_O\omega_S}{\pi}) \\ &-\sum_{k=2}^{n-1}X_k + \sum_{l=2}^{n-1}X_l\cos((\tau_{l-1}-\tau_l)\omega_O)\end{aligned}\right\}\end{aligned} \tag{3-3}$$

where $\overline{|i_o|}$ is a function of X_i and τ_i. $x(t)$ is the tank circuit current.

A. Theory of Averaged Piecewise Transform

Based on the above assumptions, the tank state variables ($x(t)$) are extended from the Fourier series (equation 2-5) by using the natural frequency (ω_O) instead of the switching frequency (ω_S). Since the state variables having displacement symmetry, the transform taken in the first half period is identical to the transform taken in the second half period. Thus the tank state variables are represented in the form

$$x(t) = \begin{cases}\sum_{k=\pm1}\langle x\rangle_k(t)\exp(jk\omega_O t) & t\in(t_0,t_n] \\ \sum_{k=\pm1}\langle x\rangle_k\left(t-\frac{\pi}{\omega_S}\right)\exp\left[jk\omega_O\left(t-\frac{\pi}{\omega_S}+\frac{\pi}{\omega_O}\right)\right] & t\in(t_n,t_{2n}]\end{cases} \tag{3-4}$$

where $\langle x\rangle_k(t)$ is a slow function, and also called the coefficient of the averaged piecewise transform. It has a period $\frac{\pi}{\omega_S}$ and is determined by

$$\langle x\rangle_k(t) = \frac{\omega_S}{\pi}\int_t^{t+\frac{\pi}{\omega_S}}x(\tau)\exp(-jk\omega_O\tau)d\tau \tag{3-5}$$

Similarly, the differentiation of the coefficient (3-5) is computed as

$$\left\langle \frac{dx}{dt} \right\rangle_k (t) = \frac{\omega_S}{\pi} \int_{t}^{t+\frac{\pi}{\omega_S}} \frac{dx(\tau)}{d\tau} \exp(-jk\omega_O\tau) d\tau$$

Integrating by parts gives

$$\left\langle \frac{dx}{dt} \right\rangle_k (t) = \frac{\omega_S}{\pi} \left\{ \left(\int_{t}^{t_1} + \int_{t_1}^{t_2} + \ldots\ldots \int_{t_{n-1}}^{t_n} + \int_{t_n}^{t+\frac{\pi}{\omega_S}} \right) \frac{dx(\tau)}{d\tau} \exp(-jk\omega_O\tau) d\tau \right\}$$

$$= \frac{\omega_S}{\pi} \left\{ \begin{array}{l} \left[x(\tau)\exp(-jk\omega_O\tau) \right]_t^{t_1} + jk\omega_O \int_{t}^{t_1} x(\tau)\exp(-jk\omega_O\tau) d\tau \\ + \left[x(\tau)\exp(-jk\omega_O\tau) \right]_{t_1}^{t_2} + jk\omega_O \int_{t_1}^{t_2} x(\tau)\exp(-jk\omega_O\tau) d\tau \\ + \ldots\ldots \\ + \left[x(\tau)\exp(-jk\omega_O\tau) \right]_{t_n}^{t+\frac{\pi}{\omega_S}} + jk\omega_O \int_{t_n}^{t+\frac{\pi}{\omega_S}} x(\tau)\exp(-jk\omega_O\tau) d\tau \end{array} \right\}$$

Hence, collecting the term $\exp(-jk\omega_O t_\bullet)$ and rewriting the terms

$$\left[x\left(t+\frac{\pi}{\omega_S} \right) \exp\left(-jk\omega_O \left(t+\frac{\pi}{\omega_S} \right) \right) \right] - \left[x(t)\exp(-jk\omega_O t) \right]$$

by $\frac{d}{d\tau} \int_{t}^{t+\frac{\pi}{\omega_S}} x(\tau)\exp(-jk\omega_O\tau) d\tau$ gives

$$\left\langle \frac{dx}{dt} \right\rangle_k (t) = \frac{\omega_S}{\pi} \left\{ \begin{array}{l} [x(t_1-) - x(t_1+)]\exp(-jk\omega_O(t_1)) \\ +[x(t_2-) - x(t_2+)]\exp(-jk\omega_O(t_2)) \\ +..... \\ +[x(t_{n-1}-) - x(t_{n-1}+)]\exp(-jk\omega_O(t_{n-1})) \\ +x(t_n-)\exp(-jk\omega_O(t_n)) - x(t_n+)\exp(-j\pi) \\ +\frac{d}{d\tau}\int_t^{t+\frac{\pi}{\omega_S}} x(\tau)\exp(-jk\omega_O\tau)d\tau + jk\omega_O \int_t^{t+\frac{\pi}{\omega_S}} x(\tau)\exp(-jk\omega_O\tau)d\tau \end{array} \right\}$$

$$= \frac{d}{dt}\langle x \rangle_k + jk\omega_O \langle x \rangle_k$$

$$-\frac{\omega_S}{\pi}\{x(t_n+)\exp(-j\pi) - x(t_n-)\exp(-jk\omega_O(t_n))$$

$$+\sum_{i=1}^{n-1}[x(t_i+) - x(t_i-)]\exp(-jk\omega_O(t_i))\}$$

(3-6)

where n is the number of stages within a half switching cycle and t_i is the end time of the stage i. $x(t_i+)$ is the right hand side limits of the state variable $x(t)$ at t_i ($x(t_i+) = \lim_{t \to t_i+} x(t)$) and $x(t_i-)$ is the left hand side limits of the state variable $x(t)$ at t_i ($x(t_i-) = \lim_{t \to t_i-} x(t)$), as shown in figure 3-1. It should be noted that $\exp(-jk\omega_O(t_n+))$ is equal to $\exp(-j\pi)$ according to equation (3-4).

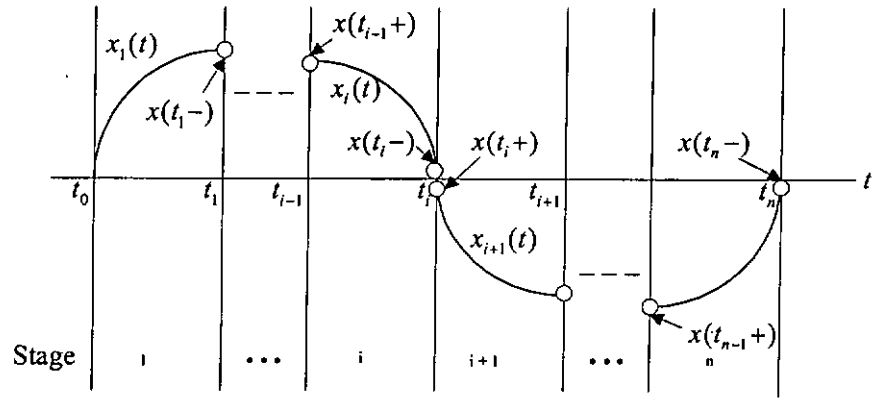


Figure 3-1, The steady state waveform of a state variable.

B. Derivation of the time invariant equations

In the previous sub-section, the theory of Averaged Piecewise Transform has been described. In this sub-section, the application of Averaged Piecewise Transform will be illustrated. The state-space equation of the converter for the tank circuit is given by

$$\frac{d\mathbf{x}}{dt} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} \tag{3-7}$$

where **x** is the state vector, **A** the state matrix, **u** the excitation vector and **B** the excitation matrix. It is a time varying state-space equation. There are a number of stages within a switching cycle. Each stage is governed by an individual set of state-space equations. In each stage, the state-space equation (3-7) is solved to give the solution of tank state variables. The tank state variables are in the form

$$x_i(t) = X_i \sin[\omega_O(t - \tau_i)] \qquad t \in (t_{i-1}, t_i] \tag{3-8}$$

where $i = 1 \cdots 2n$; is the *i*-stage of the converter, $2n$ is the number of stage in a switching period. X_i, τ_i are the function of excitation variables, tank element values and the initial values of the state variables at stage *i*. Different stages have different excitations and initial values of the state variables on the state space equation. Therefore, there are a total of $2n$ different X_i and τ_i in a switching cycle. To apply the Averaged Piecewise Transform to the converter, the tank state variables equation (3-8) is substituted into equation (3-6) to give

$$\left\langle \frac{dx}{dt} \right\rangle_k (t) = \frac{d}{dt}\langle x \rangle_k + jk\omega_O \langle x \rangle_k - \frac{\omega_S}{\pi} \left\{ \begin{array}{l} -X_n \sin(B - t_{n-1}\omega_O + \dfrac{\pi\omega_O}{\omega_S}) \left[1 - \exp(-\dfrac{jk\pi\omega_O}{\omega_S}) \right] \\ -\displaystyle\sum_{i=1}^{n-1} (X_i - X_{i+1}) \sin(B) \exp(-jkt_i\omega_O) \end{array} \right\}$$

(3-9)

where B is a constant which is the difference between τ_i and t_i at stage i. For the tank circuit current, B is equal to zero. For the tank capacitor voltage, B is equal to $\frac{\pi}{2}$. Note that it is difficult to find all X_i and τ_i. However, the values of the state variables at the end of the previous stage (stage i-1) can be used as the initial values of the state variables of the stage i. Thus, X_i can be in terms of X_{i-1} and τ_i can be in terms of τ_{i-1}. The substitution continues backward. X_i is eventually a function $k_i(X_1, u)$, and τ_i is a function $g_i(\tau_1, u)$. We have

$$X_i = k_i(X_1, u)$$

$$\tau_i = g_i(\tau_1, u) \tag{3-10}$$

where u is the excitation variable of the state space equation. Substituting equation (3-10) into (3-8) gives

$$x_i(t) = f_i(X_1, \tau_1, u, t) \tag{3-11}$$

A set of state variables in terms of only X_1 and τ_1 can be obtained.

1. Transformation of the derivative of the k-th coefficient

Substituting equation (3-11) into the derivative equation (3-9) gives

$$\left\langle \frac{dx}{dt} \right\rangle_k (t) = \frac{d}{dt}\langle x \rangle_k + jk\omega_o \langle x \rangle_k + h(X_1, \tau_1, u) \tag{3-12}$$

where h is a function of X_1 and τ_1, and u is the excitation variable only.

2. Derivation the averaged input current and output current/voltage

Rewriting equations (3-2) and (3-3) according to equation (3-11) gives

$$\overline{i_S} = v(X_1, \tau_1, u) \tag{3-13}$$

$$\overline{|i_o|} = q(X_1, \tau_1, u) \tag{3-14}$$

where both v and q are functions of X_1, τ_1 and u.

3. Solving X_1 and τ_1

By using the theory in section I-A, equation (3-11) is computed as

$$\langle x \rangle_k(t) = \frac{\omega_S}{\pi} \int_t^{t+\frac{\pi}{\omega_S}} x(\tau) \exp(-jk\omega_O\tau) d\tau$$

Since $x(\tau)$ is a piecewise function, the integration can be partitioned as

$$\langle x \rangle_k(t) = \frac{\omega_S}{\pi} \left(\int_t^{t_1} x_1(\tau) + \int_{t_1}^{t_2} x_2(\tau) + \cdots + \int_{t_n}^{\frac{\pi}{\omega_S}} x_n(\tau) \right) \exp(-jk\omega_O\tau) d\tau$$

Hence, we have

$$\langle x \rangle_k(t) = g(X_1, \tau_1, u, t) \tag{3-15}$$

where g is a function of X_1, τ_1, u and t. $t_i (i = 1 \cdots n)$ equals to τ_i in (3-9) or the switching period. Therefore, equation (3-15) becomes a function of X_1, τ_1 and u only. Since equation (3-15) is a complex equation, there are two corresponding equations in real variables for equation (3-15) (one for the real part and the other for the imaginary part). Hence, there are two coefficient equations (3-15) with two unknowns (X_1, τ_1). X_1 and τ_1 can be solved in term of the real and imaginary parts of the coefficients of the state variable ($\mathrm{Re}\{\langle x \rangle_k(t)\}$ and $\mathrm{Im}\{\langle x \rangle_k(t)\}$). It is shown that equations (3-12), (3-13)

and (3-14) are time-invariant equations in terms of X_1 and τ_1 only. Therefore, using equation (3-15), a complete set of time-invariant equations is established.

C. Averaged Piecewise Transform macromodel for a resonant converter

This section presents the implementation of a SPICE macromodel using the time invariant equation (3-15). The derivative of *k*-th coefficient equation (3-12) can be written as

$$\left\langle \frac{dx}{dt} \right\rangle_k (t) = \frac{d}{dt}\langle x \rangle_k + jk\omega_O \langle x \rangle_k + h(X_1, \tau_1, u) \tag{3-16}$$

where x is the tank state variable.

Equation (3-16) can be written by considering the tank inductor and capacitor in the form of

$$\langle v \rangle_k = L\left\{ \frac{d}{dt}\langle i \rangle_k + jk\omega_O \langle i \rangle_k + h_v(X_1, \tau_1, u) \right\} \tag{3-17}$$

$$\langle i \rangle_k = C\left\{ \frac{d}{dt}\langle v \rangle_k + jk\omega_O \langle v \rangle_k + h_i(X_1, \tau_1, u) \right\} \tag{3-18}$$

and equations (3-17) and (3-18) are rearranged to give

$$L\frac{d}{dt}\langle i \rangle_k = -L\left\{ jk\omega_O \langle i \rangle_k + h_v(X_1, \tau_1, u) \right\} + \langle v \rangle_k \tag{3-19}$$

$$C\frac{d}{dt}\langle v \rangle_k = -C\left\{ jk\omega_O \langle v \rangle_k + h_i(X_1, \tau_1, u) \right\} + \langle i \rangle_k . \tag{3-20}$$

Actually, both equations (3-19) and (3-20) are complex and four corresponding differential equations in real variables are given below.

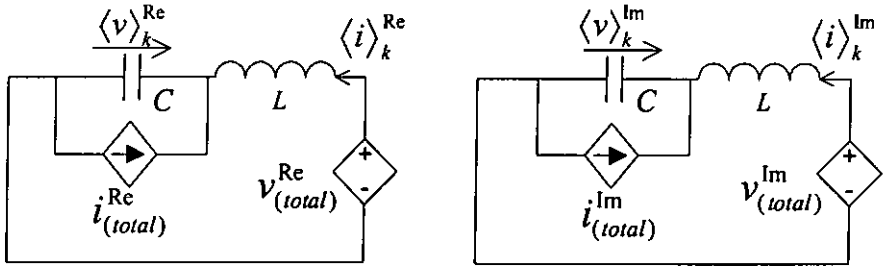
$$L\frac{d}{dt}\langle i \rangle_k^{\mathrm{Re}} = -L\left\{ k\omega_O \langle i \rangle_k^{\mathrm{Im}} + h_v^{\mathrm{Re}}(X_1, \tau_1, u) \right\} + \langle v \rangle_k^{\mathrm{Re}} \tag{3-21}$$

$$L\frac{d}{dt}\langle i\rangle_k^{\mathrm{Im}} = -L\left\{-k\omega_O\langle i\rangle_k^{\mathrm{Re}} + h_v^{\mathrm{Im}}(X_1,\tau_1,u)\right\} + \langle v\rangle_k^{\mathrm{Im}} \tag{3-22}$$

$$C\frac{d}{dt}\langle v\rangle_k^{\mathrm{Re}} = -C\left\{k\omega_O\langle v\rangle_k^{\mathrm{Im}} + h_i^{\mathrm{Re}}(X_1,\tau_1,u)\right\} + \langle i\rangle_k^{\mathrm{Re}} \tag{3-23}$$

$$C\frac{d}{dt}\langle v\rangle_k^{\mathrm{Im}} = -C\left\{-k\omega_O\langle v\rangle_k^{\mathrm{Re}} + h_i^{\mathrm{Im}}(X_1,\tau_1,u)\right\} + \langle i\rangle_k^{\mathrm{Im}} \tag{3-24}$$

Thus equations (3-21), (3-22), (3-23), (3-24) can be modelled by two equivalent circuit models with four dependent sources for the resonant tank of the converter shown in figure 3-2.



Real part resonant tank model

Imaginary part resonant tank model

where

$$i_{(total)}^{\mathrm{Re}} = C\left\{k\omega_O\langle v_C\rangle_k^{\mathrm{Im}} + h_i^{\mathrm{Re}}(X_1,\tau_1,u)\right\}$$

$$i_{(total)}^{\mathrm{Im}} = C\left\{-k\omega_O\langle v_C\rangle_k^{\mathrm{Re}} - h_i^{\mathrm{Im}}(X_1,\tau_1,u)\right\}$$

$$v_{(total)}^{\mathrm{Re}} = L\left\{k\omega_O\langle i_L\rangle_k^{\mathrm{Im}} - h_v^{\mathrm{Re}}(X_1,\tau_1,u)\right\}$$

$$v_{(total)}^{\mathrm{Im}} = L\left\{-k\omega_O\langle i_L\rangle_k^{\mathrm{Re}} - h_v^{\mathrm{Im}}(X_1,\tau_1,u)\right\}$$

Figure 3-2, The SPICE macromodel for the resonant tank circuit.

To complete the macromodel for a resonant converter, two additional equations (3-13) and (3-14) are used for the model of the input and output equivalent circuits. Thus, the

complete SPICE macromodel derived by Averaged Piecewise Transform is shown in figure 3-3.

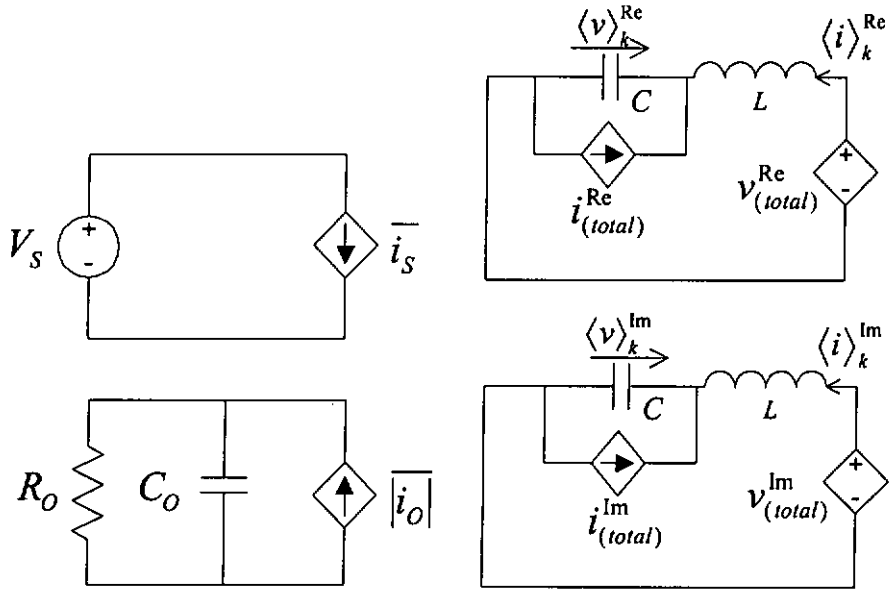


Figure 3-3, The complete SPICE macromodel for a Series Resonant Converter.

II. AVERAGED PIECEWISE TRANSFORM (APT) MACROMODEL FOR A SERIES RESONANT CONVERTER

This section presents an example application of the APT macromodel for a half-bridge series resonant converter. The circuit of the half-bridge series resonant converter was described in Chapter 2. The detailed operation and equivalent circuit for the converter were also presented. We now recall the circuit shown in figure 3-4.

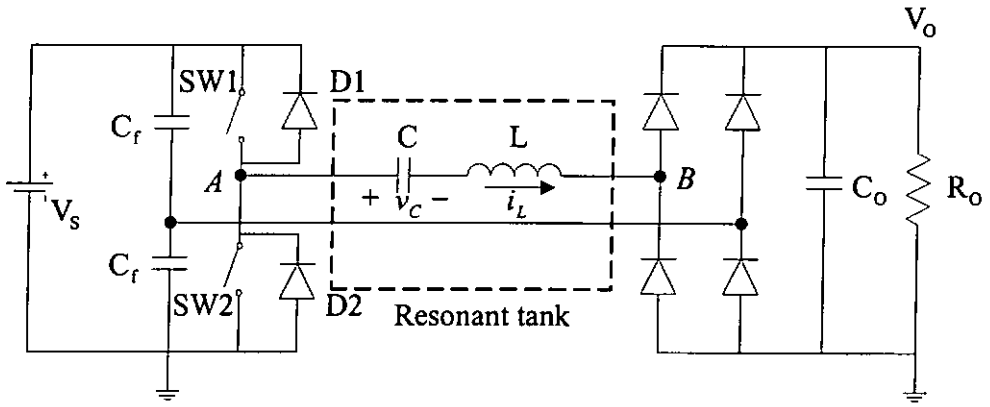


Figure 3-4, A half-bridge series resonant converter.

The steady state waveforms of the tank inductor current (i_L) and the tank capacitor voltage (v_C) for the converter operating above resonant mode are shown in figure 3-5 (detailed in Chapter 2 Section B-2). Using the assumption in section I (equation (3-1)), the solutions of i_L and v_C at the first two stages in a period are written in a piecewise sinusoidal function with the angular resonance frequency (ω_o).

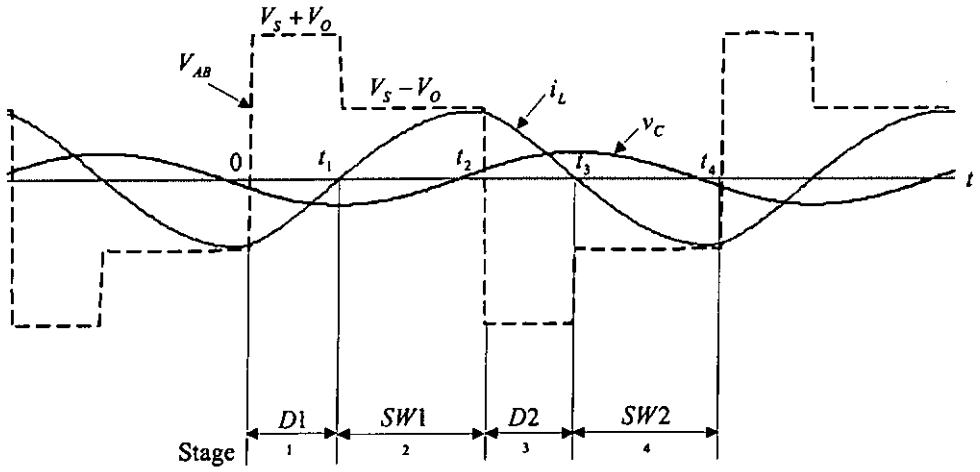


Figure 3-5, The above resonance waveforms of the tank inductor current (i_L) and the tank capacitor voltage (v_C).

At stage 1 ($0< t < t_1$)

$$i_{L1}(t) = X_1 \sin(\omega_O(t-\tau_1)) \tag{3-25a}$$

$$v_{C1}(t) = -X_1 Z_O \cos(\omega_O(t-\tau_1)) \tag{3-25b}$$

At stage 2 ($t_1< t < t_2$)

$$i_{L2}(t) = -\frac{(2V_O - X_1 Z_O)}{Z_O}\sin(\omega_O t) \tag{3-26a}$$

$$v_{C2}(t) = (2V_O - X_1 Z_O)\cos(\omega_O t) \tag{3-26b}$$

$$\text{The Characteristic impedance} = Z_O = \sqrt{\frac{L}{C}}\ \Omega$$

$$\text{Angular resonance frequency} = \omega_O = \frac{1}{\sqrt{LC}}$$

where the amplitude and phase of $i_{L1}(t)$ in equation (3-25a) are represented by the variables X_1 and τ_1 (Actually X_1, τ_1 are the functions of V_O, V_S, Z_O and the initial values of v_C and i_L at stage 1). Based on equation (3-25a), the amplitudes of $v_{C1}(t)$ in

equations (3-25b) and (3-26) can be written in terms of X_1, V_O, Z_O by replacing the initial value at each stage.

A. Transformation of the derivatives of $\langle i_L(t) \rangle$ and $\langle v_C(t) \rangle$

For $t_0 = 0, t_1 = \tau_1$ and $t_2 = \frac{\pi}{\omega_S}$, substituting them into equation (3-25) and (3-26) gives

$$
\begin{aligned}
&i_L(t_1-) = i_{L1}(\tau_1) = 0 \\
&v_C(t_1-) = v_{C1}(\tau_1) = -X_1 Z_O \\
&i_L(t_1+) = i_{L2}(0) = 0 \\
&v_C(t_1+) = v_{C2}(0) = 2V_O - X_1 Z_O \\
&i_L(t_2-) = i_{L2}\left(\frac{\pi}{\omega_S} - \tau_1\right) = -\frac{(2V_O - X_1 Z_O)\sin\left(\omega_O\left(\frac{\pi}{\omega_S} - \tau_1\right)\right)}{Z_O} \\
&v_C(t_2-) = v_{C2}\left(\frac{\pi}{\omega_S} - \tau_1\right) = (2V_O - X_1 Z_O)\cos\left(\omega_O\left(\frac{\pi}{\omega_S} - \tau_1\right)\right) \\
&i_L(t_2+) = i_{L2}\left(\frac{\pi}{\omega_S} - \tau_1\right) = -\frac{(2V_O - X_1 Z_O)\sin\left(\omega_O\left(\frac{\pi}{\omega_S} - \tau_1\right)\right)}{Z_O} \\
&v_C(t_2+) = v_{C2}\left(\frac{\pi}{\omega_S} - \tau_1\right) = (2V_O - X_1 Z_O)\cos\left(\omega_O\left(\frac{\pi}{\omega_S} - \tau_1\right)\right) + 2V_S
\end{aligned}
\tag{3-27}
$$

The calculation is not simple. However, we can use Mathematica [35] to do the calculation in a computer. (Mathematica is a powerful tool (mathematical software) to compute the engineering mathematics described in Appendix II). All the following results are generated by Mathematica.

Putting equation (3-27) into equation (3-6) and using the first coefficient (k=1) we get

$$\left\langle \frac{di_L}{dt} \right\rangle^r (t) = \frac{d}{dt}\langle i_L \rangle^r - \omega_O \langle i_L \rangle^i - \frac{2\omega_S(-2V_O + X_1 Z_O)}{\pi Z_O} \cos^2\left(\frac{\pi\omega_O}{2\omega_S}\right) \sin\left(\omega_O\left(\tau_1 - \frac{\pi}{\omega_S}\right)\right)$$

$$\left\langle \frac{di_L}{dt} \right\rangle^i (t) = \frac{d}{dt}\langle i_L \rangle^i + \omega_O \langle i_L \rangle^r - \frac{\omega_S(2V_O - X_1 Z_O)}{\pi Z_O} \sin\left(\frac{\pi\omega_O}{2\omega_S}\right) \sin\left(\omega_O\left(\tau_1 - \frac{\pi}{\omega_S}\right)\right)$$

$$\left\langle \frac{dv_C}{dt} \right\rangle^r (t) = \frac{d}{dt}\langle v_C \rangle^r - \omega_O \langle v_C \rangle^i + \frac{2\omega_S}{\pi} \left\{ \begin{array}{l} V_S - V_O \cos(\omega_O \tau_1) \\ + (2V_O - X_1 Z_O) \cos^2\left(\frac{\pi\omega_O}{2\omega_S}\right) \cos\left(\omega_O\left(\tau_1 - \frac{\pi}{\omega_S}\right)\right) \end{array} \right\}$$

$$\left\langle \frac{dv_C}{dt} \right\rangle^i (t) = \frac{d}{dt}\langle v_C \rangle^i \omega_O \langle v_C \rangle^r + \frac{\omega_S}{2\pi} \left\{ \begin{array}{l} (2V_O + X_1 Z_O) \sin(\omega_O \tau_1) \\ + (2V_O - X_1 Z_O) \sin\left(\omega_O\left(\tau_1 - \frac{2\pi}{\omega_S}\right)\right) \end{array} \right\}$$

(3-28)

B. Transformation of the input current and the output current

Substituting equations (3-25a) and (3-26a) into equation (3-2) becomes

$$\begin{aligned} \overline{|i_o|} &= \frac{\omega_S}{\pi} \int_0^{\frac{\pi}{\omega_S}} |i_L(t)| dt \\ &= \frac{\omega_S}{\pi} \left(\int_0^{\tau_1} i_{L1}(t) dt + \int_0^{\frac{\pi}{\omega_S} - \tau_1} i_{L2}(t) dt \right) \\ &= \frac{\omega_S}{\pi \omega_O Z_O} \left\{ \begin{array}{l} -2V_O + X_1 \cos(\omega_O \tau_1) \\ + (2V_O - X_1 Z_O) \cos\left(\omega_O\left(\tau_1 - \frac{\pi}{\omega_S}\right)\right) \end{array} \right\} \end{aligned} \quad (3\text{-}29)$$

Similarly, the average value of the input current $\overline{i_S}$ is given by

$$\begin{aligned}\overline{i_S} &= \frac{\omega_S}{\pi}\int_0^{\frac{\pi}{\omega_S}} i_L(t)dt \\ &= \frac{\omega_S}{\pi}\left(\int_0^{\tau_1} i_{L1}(t)dt - \int_0^{\frac{\pi}{\omega_S}-\tau_1} i_{L2}(t)dt\right) \\ &= \frac{\omega_S}{\pi\omega_O Z_O}\left\{\begin{matrix}-2V_O + 2X_1Z_O - X_1Z_O\cos(\omega_O\tau_1) \\ +(2V_O - X_1Z_O)\cos\left(\omega_O\left(\tau_1 - \frac{\pi}{\omega_S}\right)\right)\end{matrix}\right\}\end{aligned} \tag{3-30}$$

C. Transformation of the tank state variables

Substituting equations (3-25) and (3-26) into (3-5), we have

$$\begin{aligned}\langle i_L\rangle_k(t) &= \frac{\omega_S}{\pi}\int_t^{t+\frac{\pi}{\omega_S}} i_L(\tau)\exp(-jk\omega_O\tau)d\tau \\ \langle v_C\rangle_k(t) &= \frac{\omega_S}{\pi}\int_t^{t+\frac{\pi}{\omega_S}} v_C(\tau)\exp(-jk\omega_O\tau)d\tau\end{aligned} \tag{3-31}$$

For t starting at 0 (t=0) and k=1, equation (3-31) can be rewritten as

$$\begin{aligned}\langle i_L\rangle(t) &= \frac{\omega_S}{\pi}\int_0^{\frac{\pi}{\omega_S}} i_L(\tau)\exp(-j\omega_O\tau)d\tau \\ \langle v_C\rangle(t) &= \frac{\omega_S}{\pi}\int_0^{\frac{\pi}{\omega_S}} v_C(\tau)\exp(-j\omega_O\tau)d\tau\end{aligned} \tag{3-32}$$

As mentioned in section I-B, the integration is computed as follow

$$\begin{aligned}\langle i_L\rangle(t) &= \frac{\omega_S}{\pi}\left\{\int_0^{\tau_1} i_{L1}(t)\exp(-j\omega_O\tau)d\tau + \int_{\tau_1}^{\frac{\pi}{\omega_S}} i_{L2}(t-\tau_1)\exp(-j\omega_O\tau)d\tau\right\} \\ \langle v_C\rangle(t) &= \frac{\omega_S}{\pi}\left\{\int_0^{\tau_1} v_{C1}(t)\exp(-j\omega_O\tau)d\tau + \int_{\tau_1}^{\frac{\pi}{\omega_S}} v_{C2}(t-\tau_1)\exp(-j\omega_O\tau)d\tau\right\}\end{aligned} \tag{3-33}$$

Substituting equations (3-25) and (3-26) into (3-33) and writing the result by equating the real and imaginary parts, the following equations are obtained.

$$\begin{aligned}\langle i_L\rangle^r &= \frac{1}{4\pi\omega_O Z_O}(C+D)\\ \langle i_L\rangle^i &= \frac{1}{4\pi\omega_O Z_O}(A+B)\\ \langle v_C\rangle^r &= \frac{1}{4\pi\omega_O}(A-B)\\ \langle v_C\rangle^i &= \frac{1}{4\pi\omega_O}(C-D)\end{aligned} \tag{3-34}$$

where

$$C = (-2V_O\omega_S + X_1\omega_S Z_O)\cos(\omega_O\tau_1) + \omega_S(2V_O - X_1 Z_O)\cos\left(\omega_O\tau_1 - \frac{2\pi\omega_O}{\omega_S}\right)$$

$$D = 2\omega_O(2\pi V_O - 2\tau_1 V_O\omega_S - \pi X_1 Z_O)\sin(\omega_O\tau_1)$$

$$A = 2\omega_O(2\pi V_O - 2\tau_1 V_O\omega_S - \pi X_1 Z_O)\cos(\omega_O\tau_1)$$

$$B = (2V_O\omega_S + X_1\omega_S Z_O)\sin(\omega_O\tau_1) + (2V_O - X_1 Z_O)\sin\left(\omega_O\tau_1 - \frac{2\pi\omega_O}{\omega_S}\right)$$

Solving equation (3-34) for X_1 and τ_1 gives

$$X_1 = \frac{1}{Z_O}\left(\sqrt{Q^2+P^2} + 2V_O\left(1+\frac{\tau_1\omega_S}{\pi\omega_O}\right)\right) \tag{3-35}$$

$$\tau_1 = \tan^{-1}\left(\frac{Q}{P}\right) \tag{3-36}$$

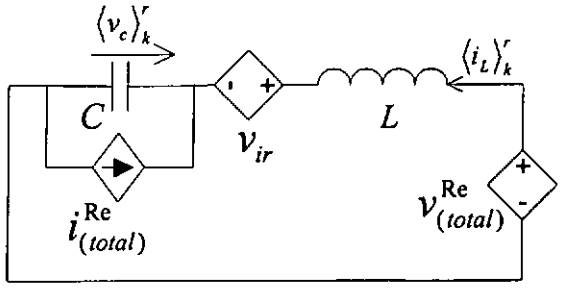
where $Q = Z_O\langle i_L\rangle^r - \langle v_C\rangle^i$, $P = Z_O\langle i_L\rangle^i + \langle v_C\rangle^r$

D. Creating the APT macromodel for the series resonant converter

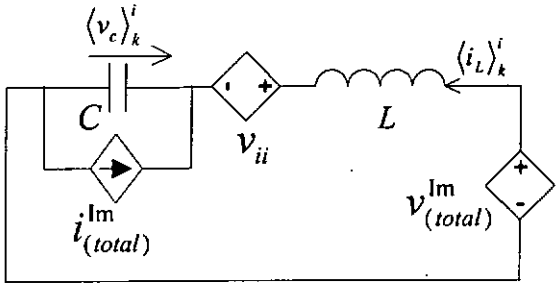
The APT macromodel contains two parts: the resonant tank model and the input/output model. They are illustrated as follows.

1. The series resonant tank model

Using the method in section I-C, equation (3-28) can be modelled by two equivalent circuits for the series resonant tank shown in figure 3-6, where V_{ir} and V_{ii} are used as the current sensors in SPICE simulations



Real part resonant tank model



Imaginary part resonant tank model

Figure 3-6, The resonant tank model of the series resonant converter for the Averaged Piecewise Transform.

$$v_{(total)}^{\mathrm{Re}} = \omega_O L\langle i_L\rangle^i + \frac{2\omega_S L(-2V_O + X_1 Z_O)}{\pi Z_O}\cos^2\left(\frac{\pi\omega_O}{2\omega_S}\right)\sin\left(\omega_O\left(\tau_1 - \frac{\pi}{\omega_S}\right)\right)$$

$$i_{(total)}^{\mathrm{Re}} = \omega_O C\langle v_C\rangle^i - \frac{2\omega_S C}{\pi}\left\{\begin{array}{l} V_S - V_O\cos(\omega_O\tau_1) \\ +(2V_O - X_1 Z_O)\cos^2\left(\frac{\pi\omega_O}{2\omega_S}\right)\cos\left(\omega_O\left(\tau_1 - \frac{\pi}{\omega_S}\right)\right)\end{array}\right\} \quad (3\text{-}37)$$

$$v_{(total)}^{\mathrm{Im}} = -\omega_O L\langle i_L\rangle^r + \frac{\omega_S L(2V_O - X_1 Z_O)}{\pi Z_O}\sin\left(\frac{\pi\omega_O}{2\omega_S}\right)\sin\left(\omega_O\left(\tau_1 - \frac{\pi}{\omega_S}\right)\right)$$

$$i_{(total)}^{\mathrm{Im}} = -\omega_O C\langle v_C\rangle^r - \frac{\omega_S C}{2\pi}\left\{\begin{array}{l} (2V_O + X_1 Z_O)\sin(\omega_O\tau_1) \\ +(2V_O - X_1 Z_O)\sin\left(\omega_O\left(\tau_1 - \frac{2\pi}{\omega_S}\right)\right)\end{array}\right\} \quad (3\text{-}38)$$

2. The input and output macrmodel of the series resonant converter

The input macromodel of the series resonant converter is shown in figure 3-7. Using the equation (3-30), the circuit of figure 3-7 is obtained.

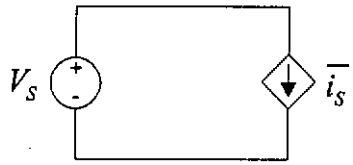


Figure 3-7, The input model of the series resonant converter.

Note that $\overline{i_S} = \frac{\omega_S}{\pi\omega_O Z_O}\left\{\begin{array}{l} -2V_O + 2X_1 Z_O - X_1 Z_O\cos(\omega_O\tau_1) \\ +(2V_O - X_1 Z_O)\cos\left(\omega_O\left(\tau_1 - \frac{\pi}{\omega_S}\right)\right)\end{array}\right\}$, V_S is the line input voltage of the converter.

Similarly, the output model of the series resonant converter is shown in figure 3-8 and equation (3-29) is used.

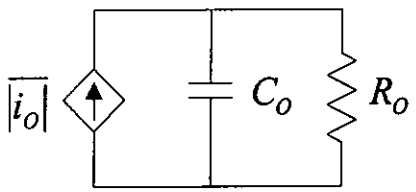


Figure 3-8, The output model of the series resonant converter.

In figure 3-8, $\overline{|i_o|} = \frac{\omega_S}{\pi\omega_O Z_O}\left\{\begin{matrix}-2V_O + X_1\cos(\omega_O\tau_1) \\ +(2V_O - X_1 Z_O)\cos\left(\omega_O\left(\tau_1 - \frac{\pi}{\omega_S}\right)\right)\end{matrix}\right\}$, C_O is the output filter of the converter and R_O is the load resistor.

3. The complete Averaged Piecewise Transform macromodel of the series resonant converter

Collecting figures 3-6, 3-7 and 3-8, the complete macromodel of the converter is shown in figure 3-9. It is noticed that equations (3-29), (3-30), (3-37) and (3-38) govern the voltage and current sources (except V_S) in the macromodel. There are two unknown variables X_1 and τ_1 in all equations. Hence, we can use equations (3-35) and (3-36) to complete the implementation of the macromodel.

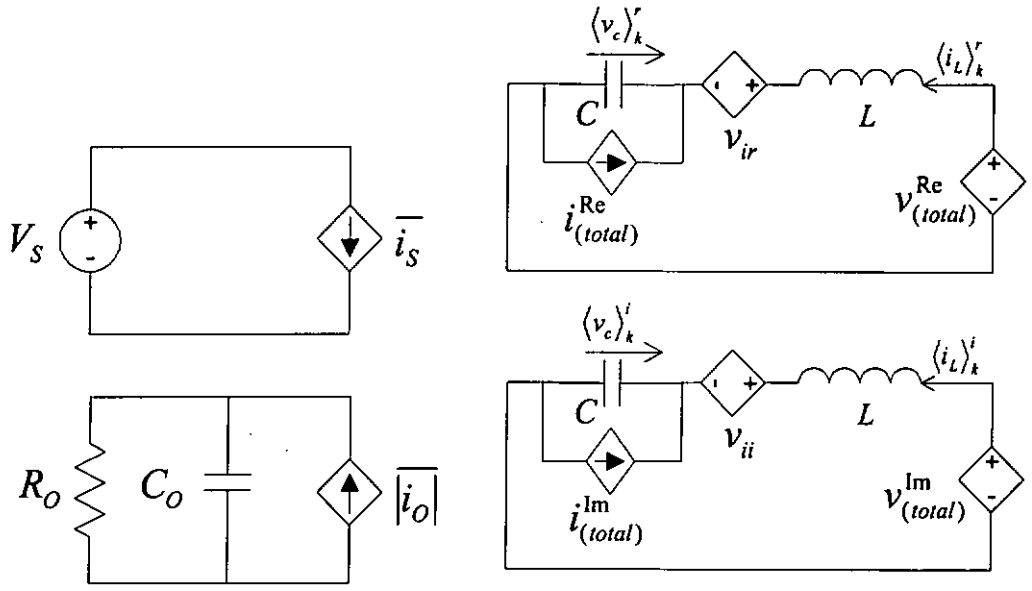


Figure 3-9, The complete Averaged Piecewise Transform macromodel of the series resonant converter.

In this section, we have developed the APT macromodel for a series-resonant converter. In section III, we will verify its validity and compare it with the DRAFT macromodel.

III. COMPARISON OF THE APT MACROMODEL AND THE DRAFT MACROMODEL

This section presents the accuracy of the APT macromodel. A comparison of the results from the DRAFT macromodel and the APT macromodel is shown in the following sections. The full circuit models are also taken to verify the accuracy of the APT macromodel. As the DRAFT macromodel is inaccurate when the circuit is operated with a large $\frac{R_O}{Z_O}$ (>0.5) ratio for a series resonant converter [6], the following experiments are mainly carried out with $\frac{R_O}{Z_O}$=2.

A. Circuit

The idealized full circuit models shown in figure 3-10 are taken from established designs [6,20] to verify the corresponding APT macromodel developed in section II. The component values used are shown in table 3-1. The driving source (V_A) is generated by a voltage controlled Wien-bridge oscillator (detailed in Appendix I). All the diodes in the circuit are idealized by setting the emission coefficient n=1 μ in the SPICE diode model.

The simulation results of DRAFT and APT macromodels can be generated quickly and easily from a single .DC, .AC and .TRAN command in SPICE. However, the full circuit simulation can only be performed for transient analysis. The DC analysis results are taken from the steady state of the transient analysis at different switching frequencies. The AC analysis are measured by the .FOUR command in the SPICE at the different modulation frequencies (described in Appendix I).

All results in the following section are generated by SPICE3f3 [1] of U.C. Berkeley.

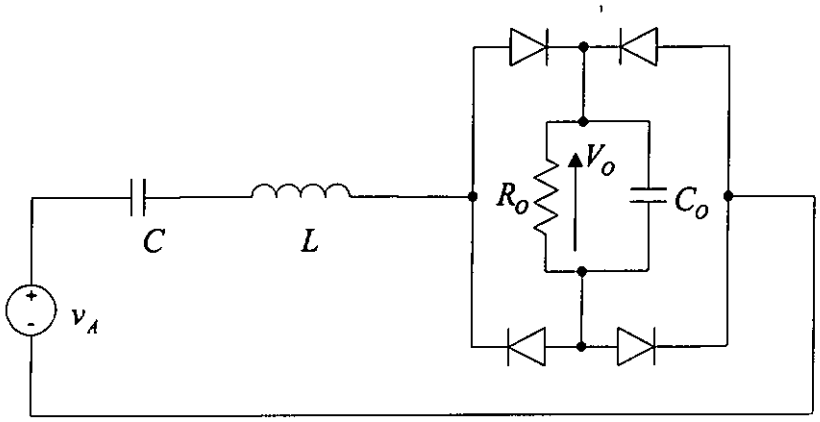


Figure 3-10, Full SPICE model for a series resonant converter.

Parameter	Value
L	$197\,\mu H$
C	$100nF$
C_O	$14.2\,\mu F$
V_S	$14V$
R_O	88.769Ω

Table 3-1, Components used in the series resonant converter.

Characteristic parameter	Value
Z_O	44.385Ω
f_O	$35.86kHz$

Table 3-2, Characteristic parameters calculated.

B. DC analysis

A comparison of simulation results with $\frac{R_O}{Z_O}$ =0.1, 0.5, 1.0 and 2.0 is shown in figure 3-11. The results are accurate at all the $\frac{R_O}{Z_O}$ ratios for the APT macromodel. However, the results from the DRAFT macromodel are accurate only for $\frac{R_O}{Z_O}$ =0.1.

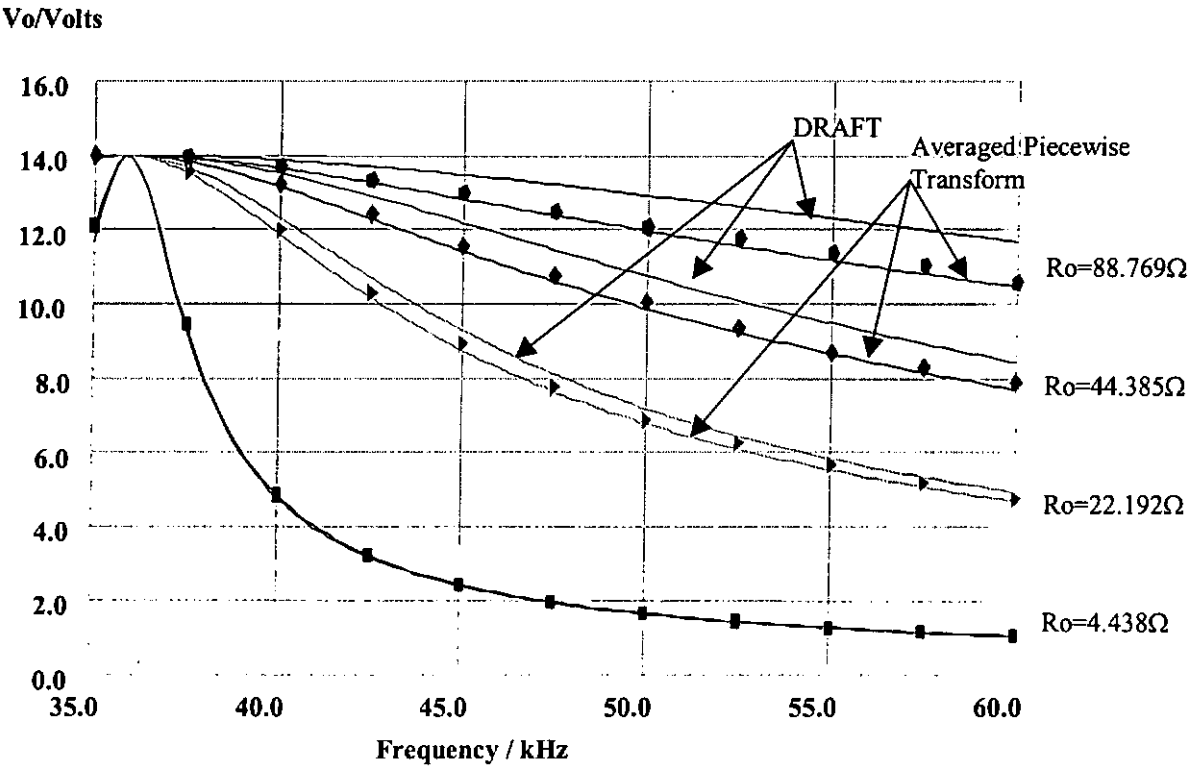


Figure 3-11, Comparison of simulation results for the series resonant converter with $\frac{Ro}{Zo}$ = 0.1, 0.5, 1.0 and 2.0 (Zo =44.3847Ω). The results from DC analyses (full lines) of the DRAFT macromodel, the Average Piecewise Transform macromodel and from steady state transient simulations (polygons) of a full model.

C. Transient analysis

The comparison of simulation results from DRAFT, APT macromodels and the full simulation is shown in figure 3-12. The response firstly shows that the converter starts up from 'cold' and reaches the steady state, then the drive frequency is abruptly changed between 50kHz and 40kHz. The result is excellent from the APT macromodel while that from the DRAFT macromodel deviates from the full transient simulation.

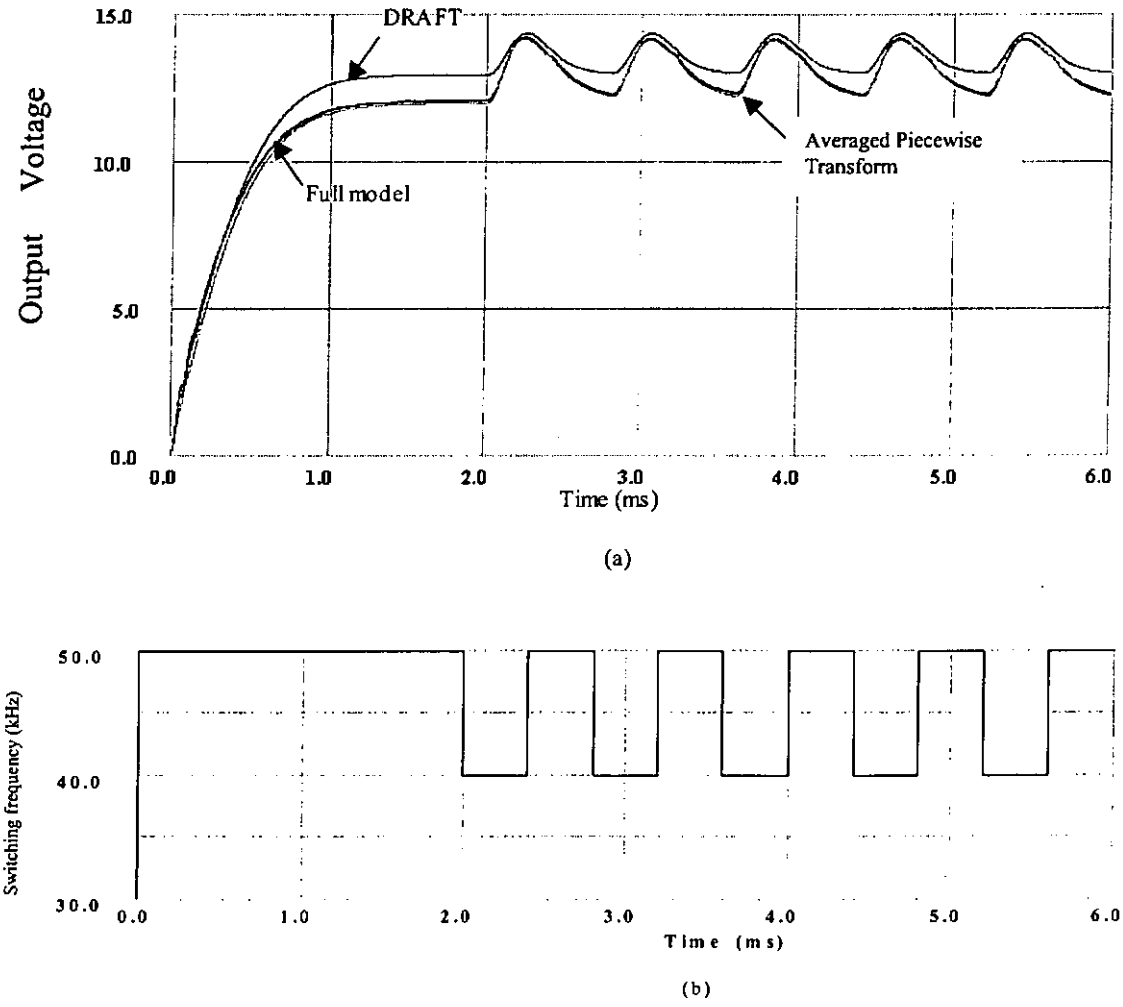
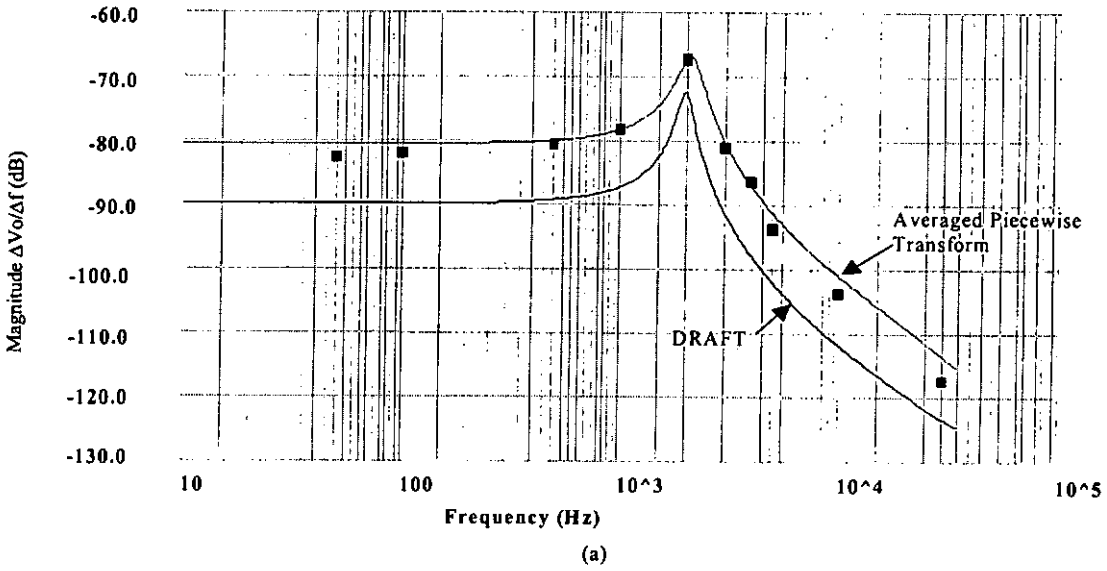


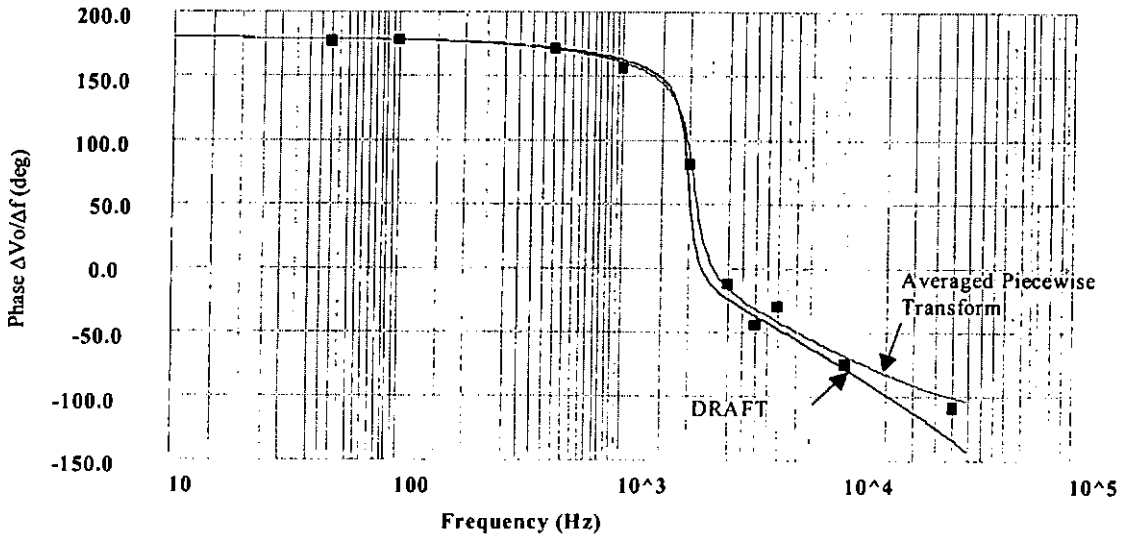
Figure 3-12, Comparison of results from transient analyses of the DRAFT macromodel, the Average Piecewise Transform macromodel and from transient simulation of a full model. A step change in switching frequency is applied.

D. AC analysis

The result for the control-to-output small signal responses of the converter at a switching frequency of 38kHz is shown in figure 3-13. The comparison of simulation results from DRAFT, APT macromodels and the full simulation is demonstrated. Also, both magnitude and phase curves from the APT macromodel in the response are in good agreement with the simulation results from the full circuit model. However, the results from the DRAFT macromodel is in poor agreement for both curves with those generated from the full circuit model.



(a)



(b)

Figure 3-13, Comparison of results from AC small signal analyses (full lines) of the DRAFT macromodel, the Averaged Piecewise Transform macromodel and from transient simulations (squares) of a full model.

E. Conclusion

This chapter has presented a novel SPICE macromodel for resonant converters. Averaged Piecewise Transform is used to generate the time invariant equations which are transformed to the circuit-oriented SPICE macromodel. When comparing with the DRAFT macromodel, the APT macromodel is much more accurate. Although the equivalent circuit model derived in this chapter is applied to the series resonant converter only, it can be extended to other resonant converters (parallel and series-parallel resonant converters) using the same basic transform.

Chapter 4

Enhancement of A Macromodel Simulator

In chapter 2, for the previous (DRAFT) macromodel for resonant converters, the tank state variables were actually described by Fourier series with an infinite number of harmonic components. The most accurate macromodel for resonant converters requires an infinite number of harmonics tank-circuit models. However, the macromodel incorporating higher harmonics results in a set of complex analytical equations. Some analytical equations that cannot be simulated directly and effectively using available dependent sources in a SPICE family simulator (macromodel simulator) require indirect and complex inputs. This approach often leads to convergence problems [5-8]. To overcome the problems, the Averaged Piecewise Transform macromodel developed in chapter 3 provides more accurate results than the DRAFT. However, Averaged Piecewise Transform requires a complex mathematical manipulation. In this chapter, a new SPICE nonlinear dependent source is developed for the input of analytical equations having no closed-form solutions. The new source is added to SPICE3 by modifying the program of the arbitrary source device package. Actually, the new source is not specific to the problem of the DRAFT macromodel but general as a new feature of simulation of "in-line" equations for SPICE.

Section I introduces the new source and describes the implementation of the new dependent source into U. C. Berkeley SPICE3f3 [1] (SPICE3) at programming level. Section II uses an example of a DRAFT macromodel for a resonant converter with higher harmonics (up to 25 harmonics) to illustrate the usefulness of the new nonlinear

dependent source. A further example is given in Appendix VI for reference. Finally, section III discusses the simulation results by using the new dependent source.

I. THE NEW SPICE NON-LINEAR DEPENDENT SOURCE

Suppose that x is a root of a nonlinear equation of the form $f(x) = 0$ for a given function f. The nonlinear equation can easily be solved with a new implementation of SPICE3 nonlinear dependent source (B) with a new option called FZERO. Thus, the general form for B becomes

BXXXXXXX N+ N- <FZERO=EXPR>

where N+ is the positive node, and N- is the negative node. The values of the FZERO parameters determine the voltages across the device. If FZERO is given then the device is a voltage source with a root finding feature. The <EXPR> expressions given for FZERO may be any function of voltages and currents. These functions can be: *abs, acos, acosh, asin, asinh, atan, atanh, cos, cosh, exp, ln, log, sin, sinh, sqrt,* and *tan.* For example, the root of the equation:

$$f(x) = [\ln(\cos(x))]^2 - [\sin(\ln(x))]^{1/2} - 0.3799 = 0 \qquad (4\text{-}1)$$

given in [14] can be calculated by using the new FZERO option (similar to voltage V and current I parameters):

B1 3 0 FZERO = [ln(cos(v(3))]^2 – sqrt[sin(ln(v(3)))] – 0.3799 (4-2)

SPICE will calculate v(3) such that $[\ln(\cos(v(3))]^2 - [\sin(\ln(v(3)))]^{1/2} - 0.3799 = 0$. Hence, x is the voltage at node 3.

A. Numerical method

For solving the nonlinear equation in the form of $f(x) = 0$, a numerical method called Tensor method [15-16,30] is used by the new nonlinear dependent source. The algorithm of Tensor method is as follows. Given $f : R \rightarrow R$, find $x \in R$ such that

$$f(x) = 0 \tag{4-3}$$

can be computed using the iteration process

$$x_{k+1} = g(x_k, x_{k-1}) \tag{4-4}$$

with the Tensor algorithm given by

$$g(x, x_{-1}) = x - \frac{s^2 f'(x)}{2[f(x_{-1}) - f(x) - f'(x)s]} \tag{4-5}$$

where $s = x_{-1} - x$. Beginning with an initial guess x_0 and x_1, (4-6) is used to generate subsequent iterations until the n^{th} iteration satisfies $|x_n - x_{n-1}| < \varepsilon$, where $\varepsilon > 0$ is a specified bound on the acceptable accuracy of the solution. By using x_0, x_1 can be calculated by the well known Newton-Raphson method [34] and is given by

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{4-6}$$

We will discuss the implementation of the new nonlinear dependent source to SPICE3f3 (SPICE3) in the next section.

B. Implementation of the new source

When adding a new device in SPICE3, we can add the new device programming codes to the existing routines of the SPICE3 program [33]. Each device is described by

functions that provide the device specific operation and data that describes the parameters of the devices [1].

For new devices, it is necessary to define their own internal data structures and operating functions. To simplify the implementation of the new source, we can modify the existing SPICE3 arbitrary source. The arbitrary source consists of data structures and functions which are also suitable for the new source. The modification of the arbitrary source can be divided into two areas and five steps:

1. Data structure area

- The first area is to modify the data structure of the arbitrary source. An additional type name (ASRC_FZERO) is created and implemented in the C programming language header file:

  asrcdef.h

  in the arbitrary source. When the program calls the arbitrary source, all relative functions and data provided by the new source are identified by the ASRC_FZERO type name.

- A parameter table - ASRCptable[] in asrc.c file stores all necessary parameters for the arbitrary source. A new set of parameters is added into the table and given below.

  IP("fzero", ASRC_FZERO, IF_PARSTREE, "Voltage souce")

  where IP is an input parameters macro. "fzero" is the parameter name that the user uses to refer to the new source. ASRC_FZERO is a type name. IF_PARSTREE is a constant to indicate that the parameter requires a parse tree representing an expression. The "Voltage source" is the description of the new source.

2. Functional package area

- We then add the type name ASRC_FZERO into all necessary functions given as follows:

  ASRCset()

  ASRCpzset()

  ASRCpzld()

  ASRCconvTest()

  ASRCacld()

  where *ASRCset()* in asrcset.c file is used for parameter preprocessing. *ASRCpzset()* in asrcpzset.c file is almost exactly the same as *ASRCset()* and is called during the pole-zero analysis. *ASRCconvTest()* in asrcconvtest.c file performs the convergence test. *ASRCacld()* in asrcacld.c file is used for the AC analysis to generate a sparse matrix in SPICE3.

  All decision trees in these functions are added with ASRC_FZERO option.

- In *ASRCparm()* in asrcparam.c file, the type name ASRC_FZERO is added to an input parser for the root finding option by using the programming statement:

  here->ASRCtype = ASRC_FZERO

  where here->ASRCtype is a device pointer to set options for the arbitrary source. When the new source is requested by the user, the pointer points to ASRC_FZERO.

- Finally, the function *ASRCload()* (implemented in asrcload.c file) is rewritten in the following way. A flow chart identifying the original and modified routines is shown in figure 4-1. Since this routine is used in the inner iteration loop of the DC and transient analyses to load the sparse matrix, at each iteration we have the previous

solutions as the input. With reference to figure 4-1, the routine starts by filling vector values from the previous solutions. A new option ASRC_FZERO has been added to the original decision tree of V and I. If the new option is detected, the root of the expression is calculated using the Tensor method mentioned in the previous section. Otherwise, the calculation follows the original design. The solution obtained is put into the right hand side of the sparse matrix for subsequent calculations.

After the modified SPICE3 program is recompiled, the new nonlinear dependent source is created. The application of the new source is demonstrated in section II.

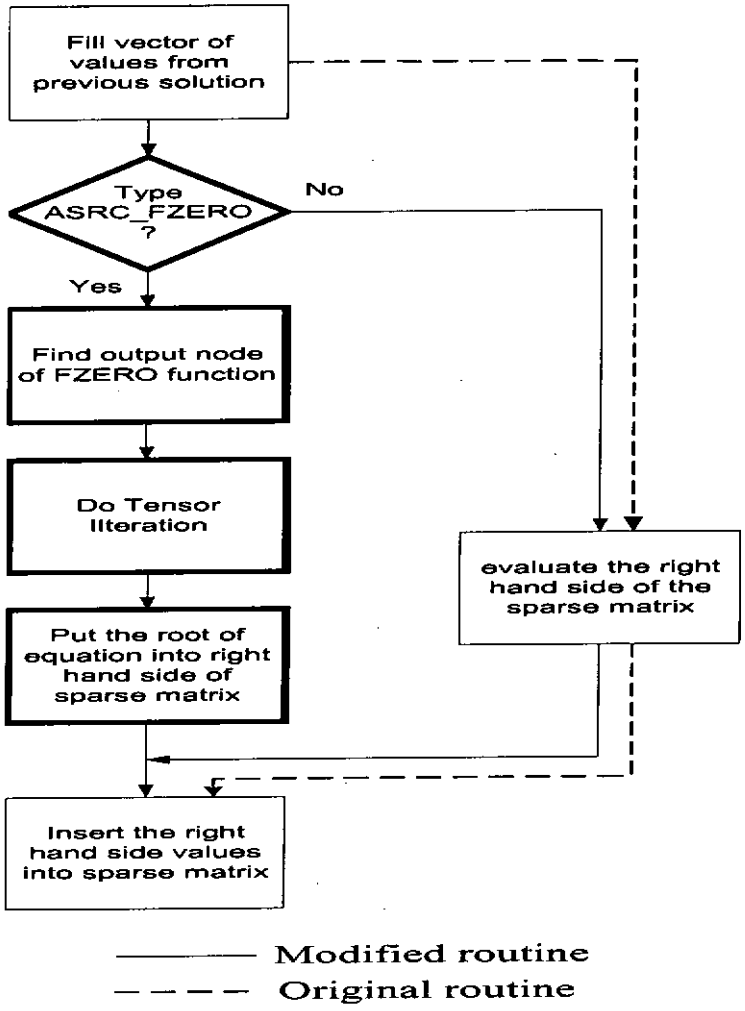


Figure 4-1, Flowchart of the modified *ASRCload()* routine.

II. APPLICATION OF THE NEW NONLINEAR DEPENDENT SOURCE

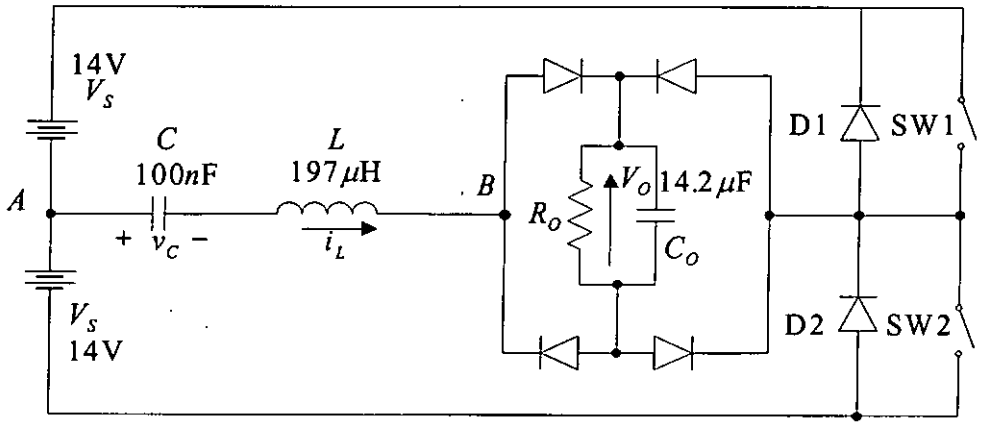


Figure 4-2, The power stage of a series resonant converter.

The DRAFT macromodel [6] for a half-bridge series resonant converter shown in figure 4-2 is used to illustrate the usefulness of the new nonlinear source. The macromodel obtained by transforming the state-space equation of the converter to an (almost) time-invariant state-space equation (equation (4-7)) by using the Fourier series has been discussed in chapter 2.

$$\begin{bmatrix} \langle v_C \rangle_k' \\ \langle i_L \rangle_k' \end{bmatrix} = \begin{bmatrix} -jk\omega_S \langle v_C \rangle_k + \dfrac{\langle i_L \rangle_k}{C} \\ -jk\omega_S \langle i_L \rangle_k - \dfrac{\langle v_C \rangle_k}{L} + \dfrac{j\dfrac{2}{k\pi}(V_S + V_O \exp(jk\rho))}{L} \end{bmatrix} \tag{4-7}$$

where $\langle \bullet \rangle_k$ denotes the kth coefficient of the Fourier series. ω_S is the switching frequency.

The phase angle, ρ, can be calculated by

$$0 = \sum_{l=1}^{\infty} I_l \sin[l\rho - \theta_l] \tag{4-8}$$

for each harmonic $l = 1,3,5\cdots$,

where $I_k = 2\sqrt{\langle i_L \rangle_{r,k}^2 + \langle i_L \rangle_{i,k}^2}$, $\theta_k = \tan^{-1}\left(\frac{\langle i_L \rangle_{r,k}}{\langle i_L \rangle_{i,k}}\right)$,

($\langle \bullet \rangle_{r,k}$ and $\langle \bullet \rangle_{i,k}$ are the real and imaginary parts of the kth coefficient. k=1,3,5,7...)

The averaged input current ($\overline{i_S}$) and output current ($\overline{|i_O|}$) are given by summing the contributions of all harmonics $l = 1,3,5\cdots$

$$\overline{i_S} = \frac{2}{\pi}\sum_{l=1}^{\infty}\frac{I_l}{l}\cos(\theta_l) \tag{4-9}$$

and

$$\overline{|i_O|} = \frac{2}{\pi}\sum_{l=1}^{\infty}\frac{I_l}{l}\cos(l\rho - \theta_l) \tag{4-10}$$

In [6], only the fundamental component is used for SPICE calculation for the reason that firstly, the fundamental approximation gives satisfactory results, and secondly, there is no such feature provided by SPICE to input an in-line equation in the form of equation (4-8). In addition, the incorporation of higher harmonics using the indirect input via the original dependent sources causes convergence problems. This prevents the incorporation of higher harmonics. The accuracy of the DRAFT macromodel is limited by the macromodel simulator SPICE. In the following section, equation (4-8) can be inputted directly via the circuit netlist file to SPICE3 with the new nonlinear dependent source. A comparison of simulation results of the fundamental macromodel and one with higher harmonics will be shown.

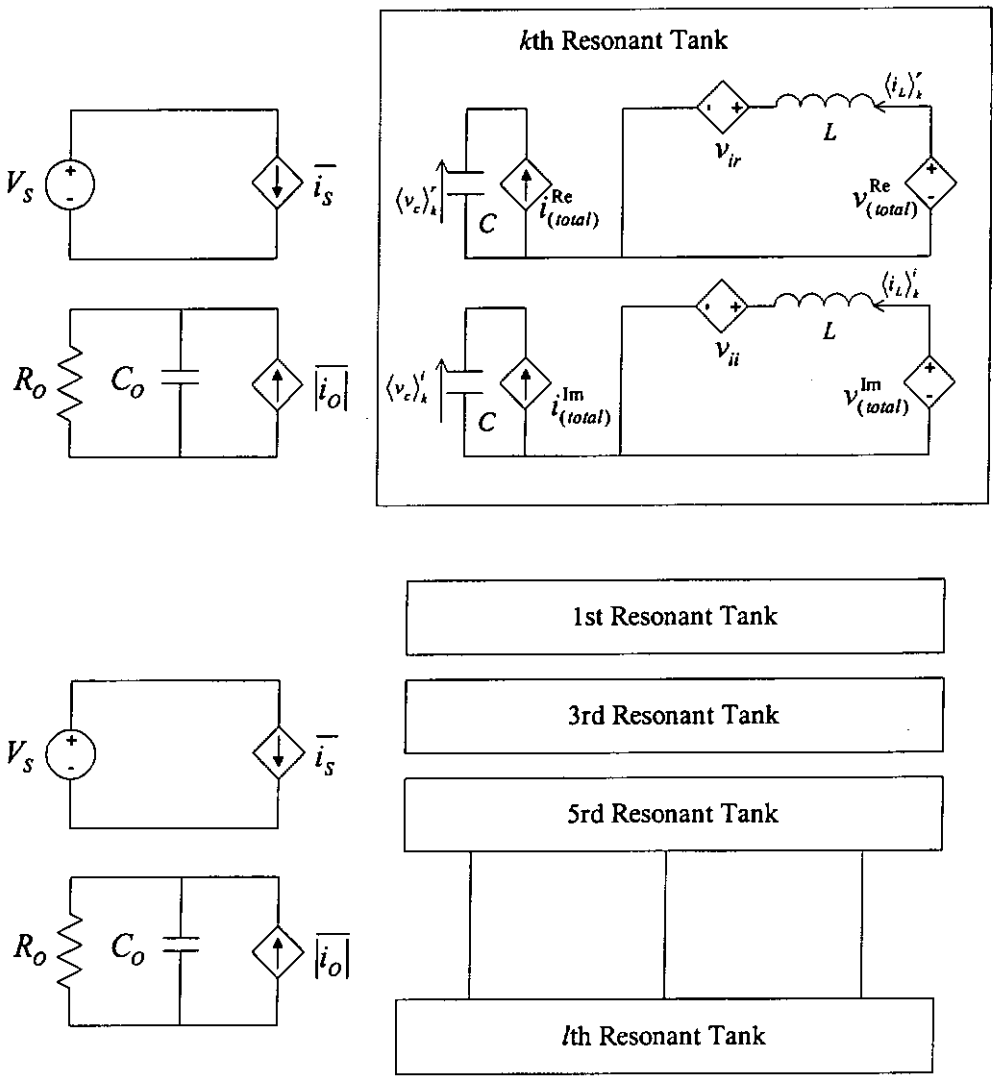
III. SIMULATION RESULTS

Equations (4-7), (4-8), (4-9) and (4-10) are directly implemented into the modified SPICE3. Therefore, the macromodel of the series resonant converter with up to 25^{th} harmonics can be simulated in SPICE3, as shown in figure 4-3. The component values of the series resonant converter are taken from [6]. The simulated results of the steady-state output voltage as a function of switching frequency above resonance for different R_O / Z_O ratios (0.1, 0.5 and 2.0) are shown in figure 4-4. It is shown that when compared with the full circuit simulation, the fundamental approximation deviates much when R_O / Z_O is greater than 0.5. The addition of the 3^{rd} harmonics gives better results. The addition of $3^{rd}+5^{th}+7^{th}+9^{th}+11^{th}+13^{th}+15^{th}$ harmonics shows excellent agreement even for $R_O = 2Z_O$. The open loop Bode response of the series resonant converter at a switching frequency of 38kHz for $R_O = 2Z_O$ is shown in figure 4-5. The fundamental approximation model deviates from the full transient simulations, and the $1^{st}+3^{rd}+5^{th}+7^{th}+9^{th}+11^{th}+13^{th}+15^{th}$ harmonics model gives better results. Moreover, the results of the macromodel in $1^{st}+3^{rd}+5^{th}+7^{th}...+19^{th}+21^{th}+23^{th}+25^{th}$ harmonics and the full model are in good agreement.

IV. CONCLUSION

A new nonlinear dependent source has been developed for SPICE3 to simulate equations having no closed-form solutions. An example using the new source is illustrated by a macromodel of the series resonant converter. An additional example other than resonant

converters is given in Appendix VII. On the other hand, the implementation of the new source into the SPICE3 program reveals that the main problem in attempting to understand and modify SPICE3 is the sheer size of the program and the complexity of the program organization. Therefore, we will discuss how to simplify the whole internal structure of SPICE3 in chapter 5.



where

$$i^{\mathrm{Re}}_{(total)} = \mathrm{Re}\{-jk\omega_s\langle i_L\rangle_k L - \langle v_C\rangle_k + j\frac{2}{k\pi}(V_S + V_O\exp(jk\rho))\}$$

$$i^{\mathrm{Im}}_{(total)} = \mathrm{Im}\{-jk\omega_s\langle i_L\rangle_k L - \langle v_C\rangle_k + j\frac{2}{k\pi}(V_S + V_O\exp(jk\rho))\}$$

$$v^{\mathrm{Re}}_{(total)} = \mathrm{Re}\{-jk\omega_s\langle v_C\rangle_k C + \langle i_L\rangle_k\}$$

$$v^{\mathrm{Re}}_{(total)} = \mathrm{Im}\{-jk\omega_s\langle v_C\rangle_k C + \langle i_L\rangle_k\}$$

Figure 4-3, A macromodel for the series resonant converter incorporating with higher harmonics.

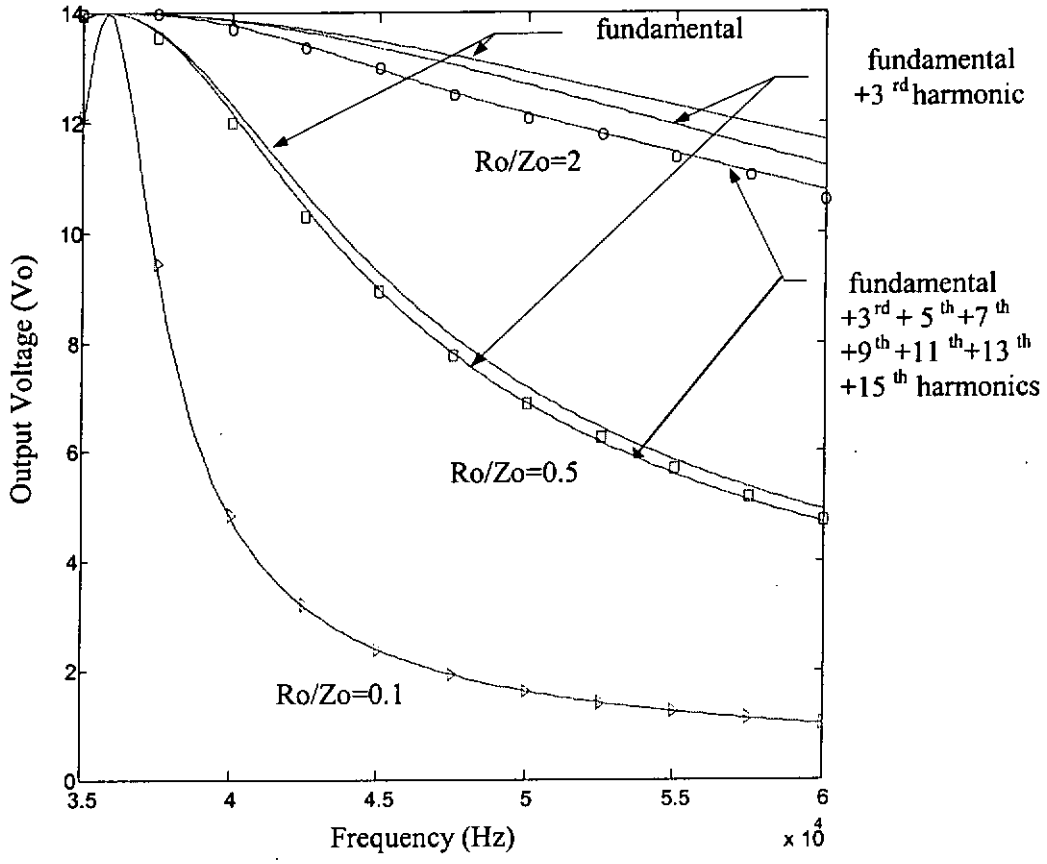


Figure 4-4, Comparsion of simulation result for the series resonant converter with Ro/Zo=0.1, 0.5 and 2.0 (Zo=44.3847Ω). The results from DC analyses (full lines) of the macromodel with fundamental average, the macromodel with fundamental + 3rd harmonic, the macromodel with fundamental +3rd +5th +7th +9th +11th +13th + 15th harmoincs and from steady state transient simulations (polygons) of a full model.

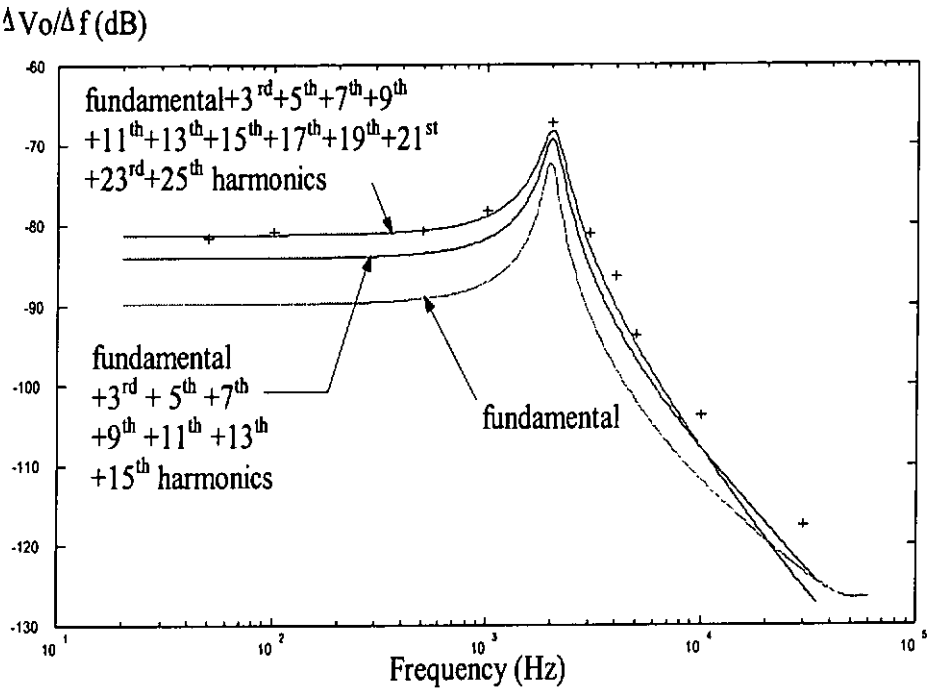


Figure 4-5, Comparison of results from AC small signal analyses (full lines) of the fundamental averaged macromodel, the fundamental $+3^{rd}+5^{th}+7^{th}+9^{th}+11^{th}+13^{th}+15^{th}$ harmonics macromodel, the fundamental$+3^{rd}+5^{th}+7^{th}+9^{th}+11^{th}+13^{th}+15^{th}+17^{th}+19^{th}+21^{st}+23^{rd}+25^{th}$ harmonics macromodel and from transient simulations (pluses) of a full model.

Chapter 5

The New Macromodel Simulator, Object-SPICE

Although circuit simulation programs differ considerably in size and capability, most simulation programs consist of four major subprograms: input, circuit setup, analysis and output subprograms. The input and output subprograms determine how easy the simulation program is to use. The circuit setup and analysis are the major parts of the program to perform circuit analyses and provide outputs of the simulation. They are the core of the simulation program and play a dominant role in enhancement, modification and future development of the simulator. For this reason, the largest portion of the development effort for simulation programs is devoted to the implementation of an efficient and clear internal structure.

Many simulators including SPICE2 [18], SPICE3 [1] and many commercial version of SPICE program (i.e. INTUSOFT SPICE, PSPICE) which use the SPICE2, SPICE3 as the engine cannot give a clear detailed internal structure. SPICE3 was rewritten from SPICE2 using the same basic algorithms. SPICE3 has made it as simple as possible by partitioning the program into packages. However, both SPICE2 and SPICE3 using the conventional procedure-programming paradigm inevitably produce a complex internal structure. The reason is that procedural program defines data and functions separately. It makes the user difficult to identify an individual package which contains both functions and data structures. It is even more difficult to understand the overall program from the isolated package.

To solve this problem, a new SPICE will be rewritten from SPICE3. The new SPICE program is written by using C++ object-oriented programming language [22] and therefore called object-SPICE.

Section I in this chapter presents an overview of the internal structure of SPICE3 and illustrates the difficulties faced by a programmer (power user) who wants to modify and enhance SPICE3. Section II gives the outline of the whole structure of object-SPICE. The detailed structure of object-SPICE is described in Appendix IV for someone who needs to modify the simulation program. In section III, a comparison between SPICE3 and object-SPICE is given. The advantages of using object-oriented programming paradigm in object-SPICE are mentioned. Finally, the modification of object-SPICE and conclusion are presented in section IV and V respectively.

I. OVERVIEW OF INTERNAL STRUCTURE OF SPICE3

The internal structure of SPICE3 is shown in figure 5-1. The program contains six major functional packages and two categories of data structures. A brief overview of these packages and their corresponding data structures is presented below. The detailed description of the structures can be found in [1].

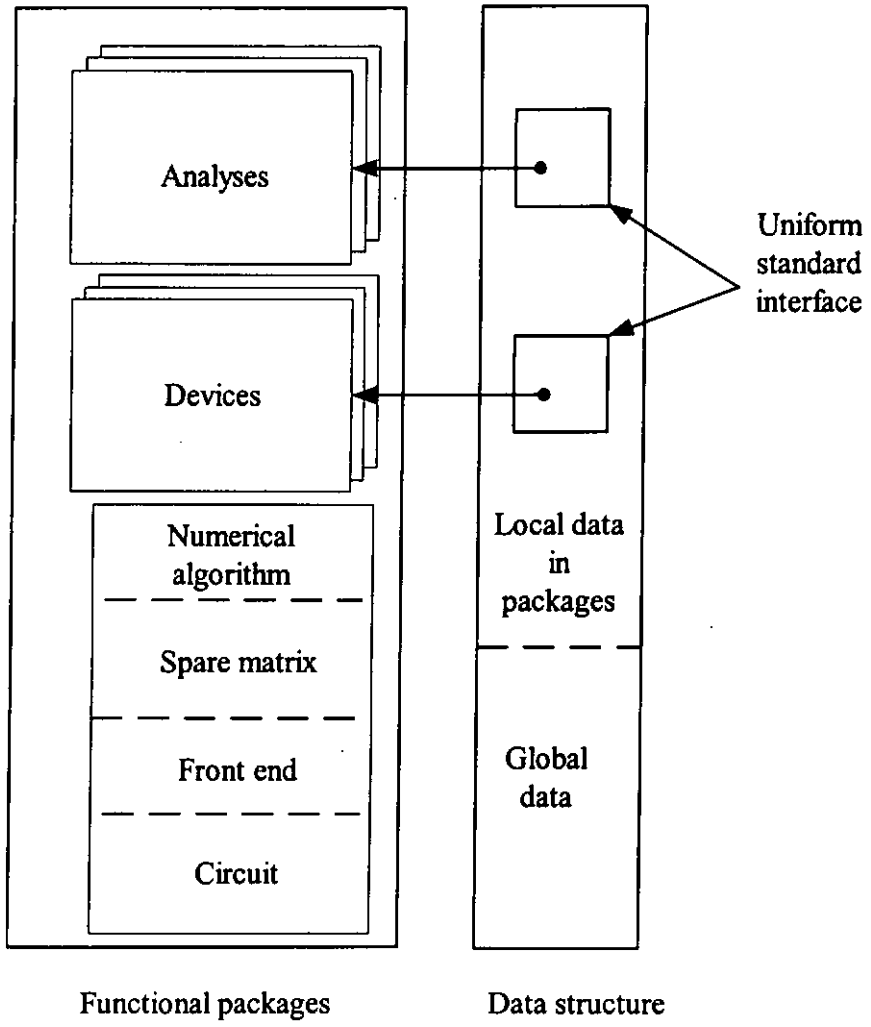


Figure 5-1, The internal structure of the SPICE3.

A. Functional packages

- "Sparse matrix" package

This package handles the sparse matrix generated by the application of the modified nodal analysis algorithm to the simulated circuit.

- "Device" package

This package contains all devices used by SPICE3. It performs all the necessary set-up operations as well as initializes the sparse matrix for the simulation. It then uses the

sparse matrix package to collect pointers to specific matrix locations where they will be referenced regularly.

- "Analysis" package

This package contains all basic algorithms for circuit analyses used by SPICE3. It performs the simulation by calling on device functions, matrix operations and generates the results.

- "Numerical algorithm" package

This package handles the basic numerical algorithms used by the analysis package. It includes numerical integration functions and Newton-Raphson iteration loops.

- "Front end" package

This package handles the input language, graphical output, output data management and user interface.

- "Circuit" package

This package is used to hold other packages together. It guides the sequence of analyses and loops through the various devices, and provides an interface to the front end package.

B. Data structure

The data structure of SPICE3 can be broken into two major categories: global and local. Global data can be used by all functional packages in the program. The most frequently used data structure is the CKTcircuit structure that knows all details of circuit simulation in the program. Local data is used within each functional package.

C. Uniform standard interface

The largest part of SPICE3 is the "device" and "analysis" packages. There are a total of 26 devices and 11 analyses provided by SPICE3. To cater for further addition of new devices and new analyses, template function pointers are defined. Two sets of template functions have been used for devices and analyses. Each set of template functions is collected into the data structures shown in figure 5-1. The data structure serves as a uniform standard interface to be used by other packages. This structure contains a set of function pointers to functions used to implement a particular device or analysis. For example, the function pointer *DEVsetup* in a device interface will point to *RESsetup* or *CAPsetup* to deal with a resistor or a capacitor respectively. Similarly, an analysis interface is a data structure that describes all necessary functions for any analysis. Other packages can use the interface to perform an analysis without knowing the type of analysis.

D. Execution sequence of functional packages

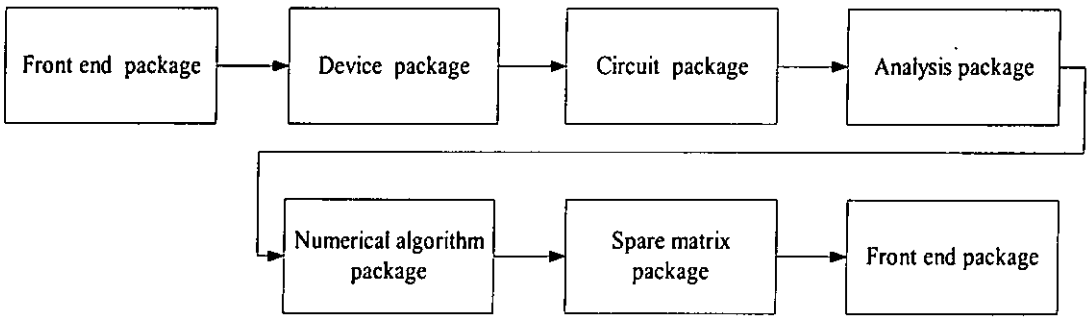


Figure 5-2, The sequence of execution of functional packages.

During the simulation, the major sequence of execution of functional packages is shown in figure 5-2. The front end package reads a circuit netlist and parse into a internal data structure. The data structure is passed to the device package for setting up devices used by the analysis package. The data structure is then manipulated by the circuit package for guiding a sequence of analysis loop. The analysis package calls functions from the numerical and sparse matrix packages to perform analyses. Finally, the results of analyses are plotted by the front end package to complete the simulation.

E. Current limitations of the SPICE3 program

The SPICE3 program consists of six functional packages, the modification of the program can focus on an individual package for specific functions and data. Each package contains data and functions manipulating in declaration files (header files) and implementation files (routine files) respectively. However, the following difficulties have been found.

1. Although SPICE3 is divided into six major functional packages, the source code of the program is physically divided into packages by the header files. In order to be consistent and easy to maintain, each package has a header file which declares all functions for that package. Since there is no explicit way to declare public functions or private functions, it is hard to identify an entry point of a package.
2. Some functions from different packages are grouped into a single header (e.g. Circuit package, Numerical package). Consequently, the programmer can only use the function name to guess the properties of those functions as SPICE3 sometimes uses a

specific naming system for functions. For example, all functions in the device package uses leading characters – DEV, while functions in the front end package uses other leading characters – FTE. For functions that do not follow this naming system, one can refer to the SPICE3 document. However, some functions are not documented, their role in SPICE3 becomes difficult to guess.

3. The uniform standard interfaces which are used for device and analysis packages are all recognized by other functional packages. However, it is hard to find where the function pointers point to. The actual function becomes "hidden" in the program. A programmer has to find the interface, then the linking table. The linking table is the only place to find the actual function.
4. Functions in a package use data structure declared outside the package. When a programmer traces the definition of the structure, the structure has sub-structures which are declared inside other packages. Sooner or later, the programmer becomes exhausted.

To solve the above problems, SPICE3 has been rewritten as object-SPICE. Object-SPICE utilizes the features of object oriented programming paradigms [22]. The fundamental concepts of object programming are:

1. "Object" which allows some data and functions to be isolated from other data and functions,
2. "Inheritance among objects" which can reduce the duplication of programming codes, and
3. "Polymorphism" which allows removing function pointers for the uniform standard interfaces.

Based on the object programming, we have developed a novel SPICE simulation program – object-SPICE by using the same basic algorithm of SPICE3.

II. OBJECT-SPICE SIMULATION PROGRAM

The six packages in SPICE3 have been replaced by six objects in object-SPICE. The details will be explained in the following sub-sections.

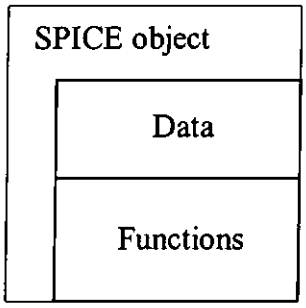


Figure 5-3, The object diagram in object-SPICE.

A. Objects in the program

Object-SPICE is implemented with objects. An "object" is a structure that consists of data and functions, as shown in figure 5-3. The data and functions inside an object can be declared as public to allow access by other objects. They can also be declared as private to be isolated from other objects. Object-SPICE can be developed into six major isolated objects shown in figure 5-4. A detailed description of each object is given in Appendix IV-II.

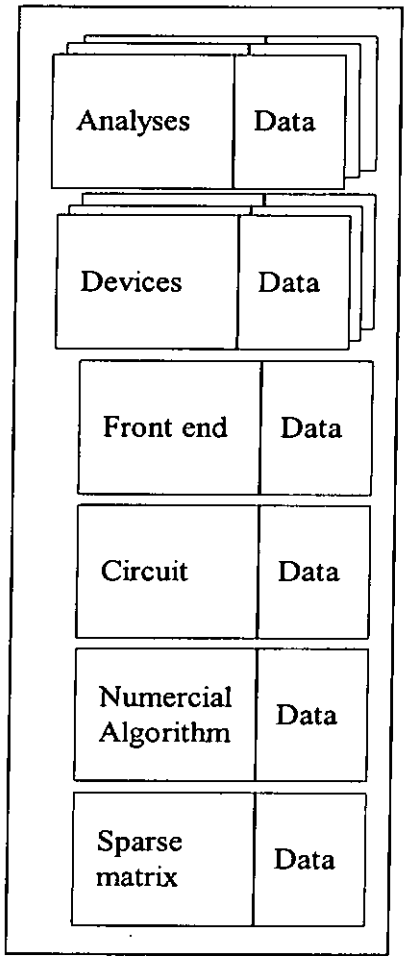


Figure 5-4, Isolated objects in object-SPICE.

- "Devices" object handles all types of devices provided by object-SPICE. For each type of devices, there is a group of necessary models and functions for performing all necessary jobs required by other objects.
- "Analyses" object handles all types of analyses provided by object-SPICE. For each type of analyses, there is a basic algorithm for calling on device functions, numerical methods, matrix operations and output operations as needed to perform the type of analysis.

- "Sparse matrix" object handles sparse matrices in the program. The object provides the matrix allocation, matrix element creation and calculation of the solution of circuit equations.
- "Numerical algorithm" object is responsible for the basic numerical iterations and integration methods used by object-SPICE as well as inserts devices equations into matrices.
- "Circuit" object provides the basic functionality of the circuit simulation. It establishes the initial condition, determines the operating point of the simulated circuit and drives the various devices to setup their parameters for circuit simulations.
- "Front end" object handles the front end command loop of the program. It reads the input circuits files, runs the simulation, and collects the output results.

B. Inheritance among the objects

Functionally, there is not much difference between a package and an object. However, both data and functions can be declared explicitly as private and public. Public functions and data can be accessed outside the object while private functions and data can only be accessed within the object. Programmers can easily identify public and private functions and data via the header files.

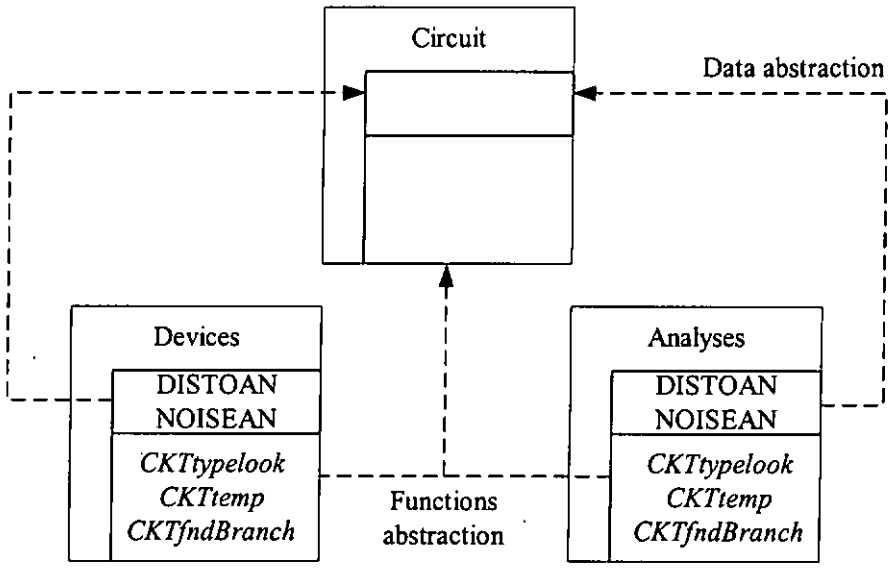


Figure 5-5, Abstraction of the common properties (data structure and functions) from devices and analyses object into circuit object.

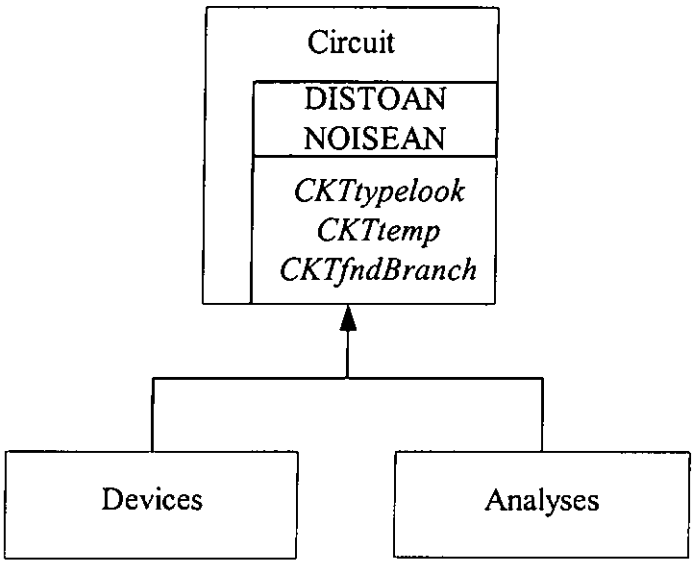


Figure 5-6, The hierarchy structure of circuit, devices and analyses objects.

Another advantage of using object programming is that we can have inheritance among the objects. Inheritance can reduce the size of program codes and increase readability. The inheritance of objects in object-SPICE can be illustrated using the devices and analyses objects.

1. Code reuse

In object-SPICE, both the analyses and devices objects have the common properties (data structure and functions) shown in figure 5-5. There are two common data structures (DISTOAN and NOISAN) in both objects. Similarly, both objects encapsulate three common functions (*CKTtypelook, CKTtemp and CKTfndBranch*). All these common properties are then abstracted into a higher level object - circuit object. This results in a hierarchy with circuit object and its two child objects, devices and analyses, as shown in figure 5-6. In other words, the circuit is a more general object, while the devices and analyses represent specific objects of the circuit. The properties of the circuit can be inherited or overridden by its child objects. Therefore, inheritance among devices and analyses objects can reduce duplication of programming codes.

2. The hierarchy tree

The same mechanism is applied to all objects in object-SPICE. A single hierarchy tree has been established as shown in figure 5-7. The bottom level of the tree is the devices and analyses object. The circuit object is at a higher level of these two objects. The circuit object is abstracted into general numerical and sparse matrix objects which are the elementary objects to perform simulation by the program. Finally, the common

properties in both sparse matrix and front end objects are abstracted into a higher level of the hierarchy tree, common root object. The common root object is the most basic object in object-SPICE. It provides the basic properties for the whole program. The program is developed into the single hierarchy tree. The detailed hierarchy tree of the program is described in Appendix IV.

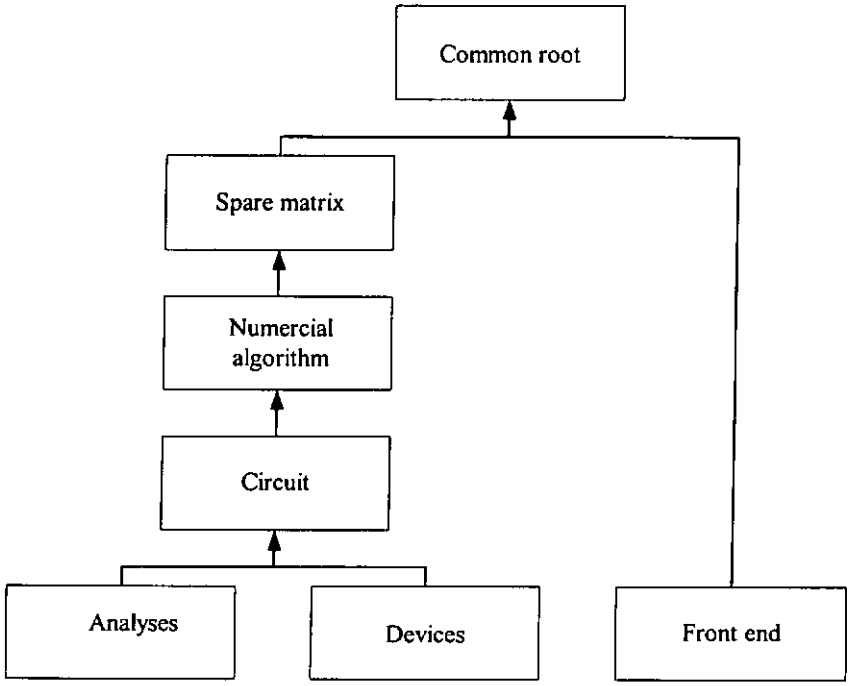


Figure 5-7, The simplified hierarchy tree of object-SPICE.

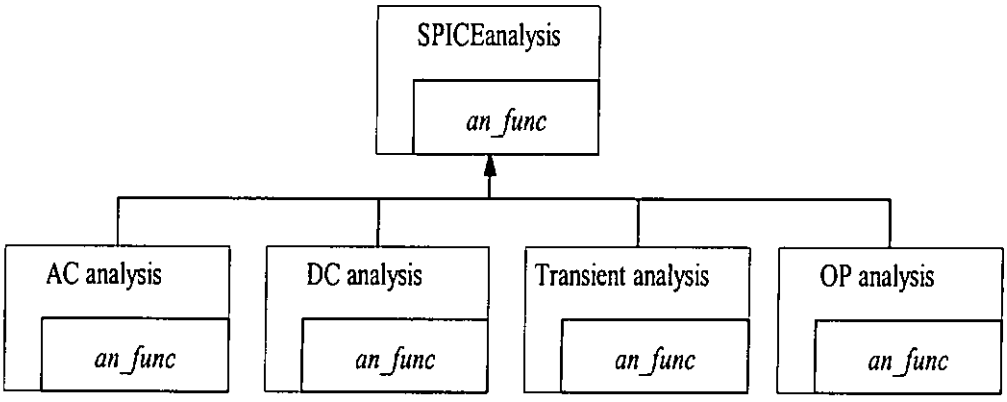


Figure 5-8, The function *an_func* in SPICEanalysis object.

C. Polymorphism

Section I-C has described SPICE3 using function pointers to select the appropriate device or analysis functions. In this section, we will present the mechanism used in object-SPICE. Instead of function pointers, object-SPICE contains a new generic interface in both analyses and devices objects. Figure 5-8 shows the interface for the analyses object. SPICEanalysis (a general analyses object) provides a generic interface for all specific analysis objects. Each type of analyses inherits all features from the common object – SPICEanalysis, and adds its own properties to form a specific analysis object. However, each specific analysis object in figure 5-8 implements its own version of function *an_func*. Other objects in the program can call or send messages to a different analysis function in the program using the same function *an_func*. Object-SPICE will automatically perform the analysis function that is specified by the user. For the devices object, each device inherits all data and functions in the common basic SPICEdev object and adds its own features to form a specific object of that type. Each device has a common interface (a common function name). The detailed mechanism of the generic interfaces for both analyses object and devices object is given below.

D. Analysis generic interface

Figure 5-9 shows the mechanism of the generic interface provided by SPICEanalysis.

1. *setParms* function is called for setting the necessary parameters for a specific analysis. The *setParms* function in a TRANinfo object (an object performing transient analysis in

the program) will override the SPICEanalysis *setParms* function for storing the necessary parameter values.

2. *askQuest* function is called for querying the parameters used by the specific analysis. In transient analysis, the *askQuest* function in the TRANinfo object will override the *askQuest* contained in the SPICEanalysis.

3. *an_init* function is called for initializing the specific analysis. The *an_init* member function in the TRANinfo object will override the SPICEanalysis *an_init* function for setting up the transient analysis.

4. *an_func* function is called for actually performing specific analysis. The *an_func* function in the TRANinfo object will also override the *an_func* function contained in the SPICEanalysis object for carrying out the analysis.

In addition to overriding the SPICEanalysis base object functions, the TRANinfo object has three additional properties - time point cleanup, breakpoint cleanup and calculation of truncation error, which are mediated by the additional member functions: *CKTaccept*, *CKTclrBreak* and *CKTtrunc*.

Similarly, the ACinfo object carries out the AC analysis by replacing all functions provided by SPICEanalysis except the *an_init* function. The ACinfo object uses the *an_init* function provided by the SPICEanalysis.

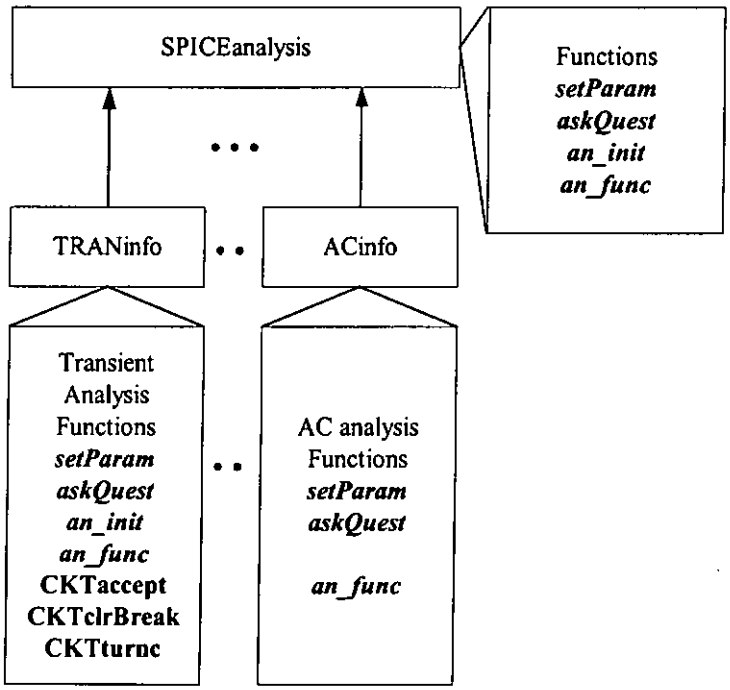


Figure 5-9, The generic interface in the SPICEanalysis object.

E. Devices generic interface

The SPICEdev object provides the generic interface for all device objects. Each device object inherits or overrides the base object functions: *DEVparam, DEVload, ... , DEVpzload*, as shown in figure 5-10. When the arbitrary dependent source is called, functions in SPICEdev are replaced by functions in the dependent source object (WinASRC object). Likewise, when the constant voltage source is called, functions in SPICEdev are overridden by functions in the voltage source object (WinVSRC object).

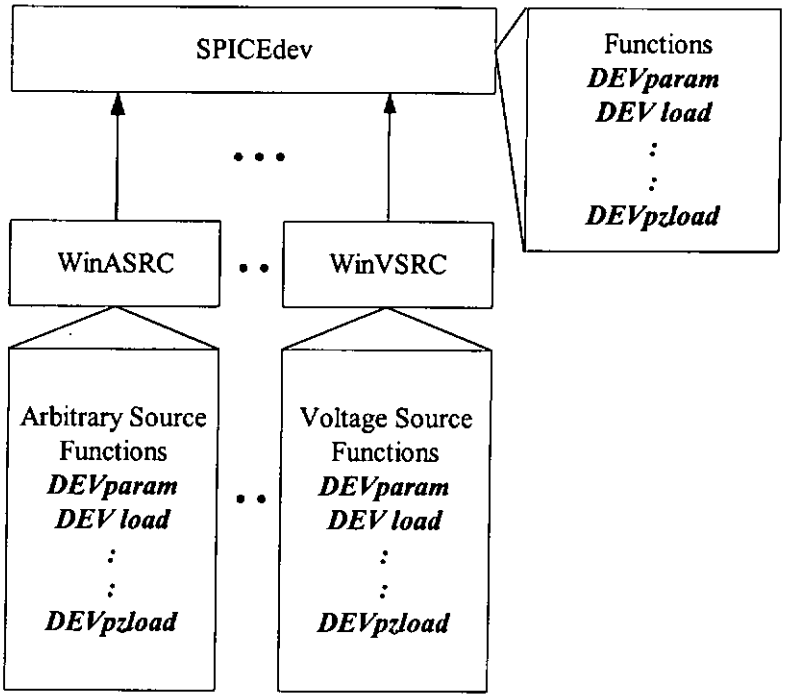


Figure 5-10, The generic interface in the SPICEdev object.

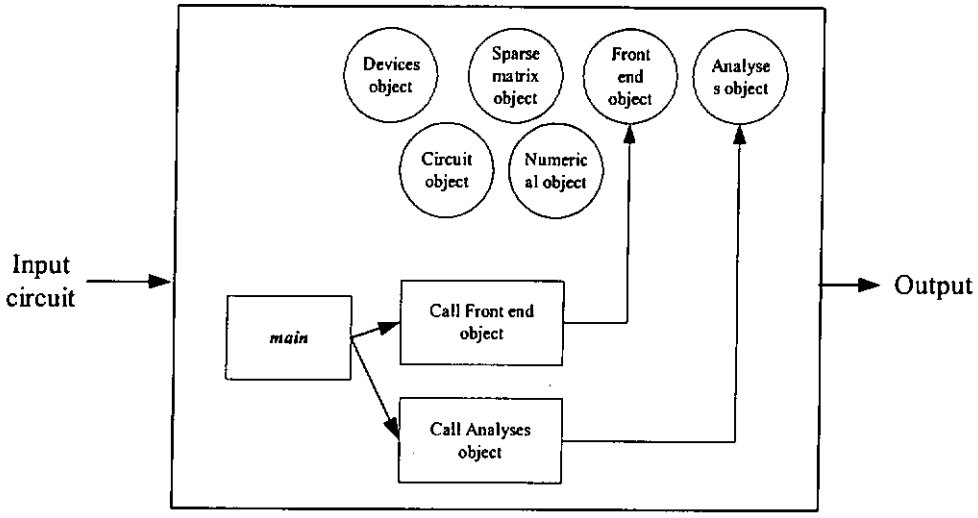


Figure 5-11, The overall operation of object-SPICE.

F. The overall operation of object-SPICE

Object-SPICE will create all individual objects during circuit simulation. All these objects perform operations isolated from each other. In figure 5-11, the *main* function in

the program calls the front end object to setup the simulation. It then sends the circuit analysis message to the analyses object and requests a service for performing analyses. The analyses object then invokes the devices object and the numerical algorithm object for further activities to complete the simulation and output required results.

III. DISCUSSION

An overview of the two programs has been presented in previous sections. By using object-SPICE, the limitations of SPICE3 mentioned in section I-E have been eliminated. The reasons are illustrated in subsection A. However, the improvement of readability does have speed penalty. A comparison of the speed of object-SPICE and SPICE3 is given in subsection B.

A. Internal structures

1. The first problem in section I-E is solved by having the explicit way of declaring public and private functions and data.
2. The second problem is that some header files contain functions from different packages. No separation is provided for each package in SPICE3. However, in object-SPICE, all functions have been arranged into their own objects. No combination of such functions exists.
3. For the third problem, object-SPICE uses the function polymorphism to realize the generic interface. No function pointer and the linking table exist anymore.
4. Finally, a data structure declared outside a package in SPICE3 makes the programmer difficult to trace. In object-SPICE, objects are related in the form of a hierarchy tree. Functions manipulate a data structure outside an object. Such the data

structure is declared inside an object which is at a higher level of that object. The programmer finds the structure consistently from the tree.

B. Speed of object-SPICE

The speed of object-SPICE has been measured using circuits from the SPICE3C1 user guide [1,18]. The result is generated by a 166MHz P5 personal computer and shown in table 5-2. Object-SPICE is about 13% slower than SPICE3. However, the simplicity of the internal structure of object-SPICE simulation program leads to greater code maintainability.

Circuit name	Run time using SPICE3 in second (R_3)	Run time using Object-SPICE in second (R_o)	$\frac{R_O - R_3}{R_3} * 100\%$
BSIM1TST	0.631	0.651	3.169
BJTNOISE	0.070	0.100	42.857
BSIM2TST	0.391	0.451	15.345
DIODISTO	0.061	0.0699	14.590
LTRA_1	3.365	3.655	8.618
LTRA_2	25.236	25.938	2.781
LTRA_3	7.040	7.361	4.559
MOSAMP2	0.912	0.941	3.179
MOSMEM	0.130	0.160	23.076
PZ2	0.020	0.020	0.000
PZT	0.030	0.040	33.333
RCA3040	0.431	0.481	11.600
RTLINV	0.080	0.090	12.500
RTL_INVERTER	0.04	0.050	25.000
SCHMITT	0.100	0.110	10.000
DIFFPAIR	0.200	0.230	15.000

Note: The average value of $\frac{R_O - R_3}{R_3} * 100\% = 13.271\%$

Table 5-2, The total analysis time for benchmark circuits for SPICE3 and object-SPICE.

IV. MODIFICATION OF OBJECT-SPICE

When enhancement of the program is required, the modification of the new program can be realized in several ways: A new class[1] (e.g. new device) can inherit from the existing class of object-SPICE and can add the specific data and functions. The second way is to copy the existing class and modify the code for particular functions and data structures of that class. The new class is then added to the hierarchy tree of the program. The easiest way is the direct modification of the existing classes of object-SPICE. An example of the modification of the arbitrary source developed in Chapter 4 is presented. The modification can also be done on object-SPICE. It is outlined as follows:

- The data structures and operation functions are modified in the class of arbitrary source only.
- First, the data structure that is specific for the dependent source is added to the data declaration of the class of arbitrary source.
- Next, the operation functions in the class of arbitrary source (*DEVacload, DEVask, DEVconvTest, DEVload, DEVparam, DEVpzLoad, DEVpzSetup, DEVsetup*) is modified. The detailed modification of each function is the same as that mentioned in Chapter 4.
- Finally, object-SPICE is recompiled.

It is shown that the benefits of the implementation of a dependent source using object-SPICE are: (1) The implementation is localized in the class of arbitrary source. (2) There

---

[1] A class is a language construct to implement an object in a program.

is no linking table and function pointers. (3) The user can directly modify the function in the arbitrary source class but in SPICE3 the user needs to deal with pointers.

For users who are familiar with SPICE3, a cross-reference table of the functions used by SPICE3 and object-SPICE is given in Appendix V.

V. CONCLUSION

Object-SPICE presented in this chapter utilizes features of object-oriented programming (such as object, inheritance and polymorphism). Object-SPICE yields a much clearer structure when compared to the internal structure of SPICE3 by (1) removing the function pointer reference, (2) declaring private and public parts of each object, (3) combining data and functions to form an isolated object. Object-SPICE is about 13% slower than SPICE3.

Chapter 6

Conclusion

I. SUMMARY

In this thesis, a new macromodel for resonant converters, a new dependent source for SPICE simulation and a new SPICE have been presented. The new macromodel was extended from the macromodel using Device Representation Averaged Fourier Transform (DRAFT). Improved results of the new macromodel over the existing macromodel have been illustrated. The new dependent source has been implemented in both SPICE3 and the new SPICE for simulating equations having no closed-form solutions. An example of using the new source was given. Finally, the new SPICE has been developed in a clear hierarchical structure, allowing easy further development and maintenance. Some of the research works on SPICE macromodel has been published in international conference proceedings and is attached in Appendix VIII. A summary of the important results from each chapter is presented below.

Chapter 2 reviewed the operation of a resonant converter, the equivalent circuit and state-space equations of the converter. A description of the existing (DRAFT) macromodel for resonant converters has been given. DRAFT is developed based on the Extended Describing Function Method. The Extended Describing Function Method represents the state variable by Fourier series. This approach limits the performance of the DRAFT macromodel.

In chapter 3, a new averaging method has been proposed. The new averaging method has been extended from DRAFT. The development of the new macromodel for resonant converters has been divided into two parts. The first part is to develop a set of time-invariant equations. The second is to develop an equivalent circuit for the time-invariant equations. Then the equivalent circuit has been implemented in SPICE to perform all high level analyses: such as AC, DC, and transient analyses. An example of the new macromodel (APT macromodel) for a half-bridge series resonant converter has been developed. A comparison of the simulation results generated from the new macromodel with that from the DRAFT macromodel has shown that the APT macromodel can give a much more accurate result.

In order to enhance SPICE to simulate macromodels of converters, chapter 4 has introduced a new tool – a new nonlinear dependent source for simulating equations having no closed-form solutions. The implementation of the new source into SPICE3 at programming level has been given. A numerical method called Tensor method has been proposed for solving such equation in SPICE3. An example of the DRAFT macromodel for a resonant converter incorporating with higher harmonics has demonstrated the robustness and usefulness of the new nonlinear dependent source. An additional example other than resonant converters is given in Appendix VII.

While the new source has been developed, the main problem in attempting to understand and modify SPICE3 is the sheer size of the program and the complexity of the program organization. Thus, a new SPICE simulation program has been developed with a clear

internal structure for power users of SPICE in chapter 5. An overview of the internal structure of SPICE3 program has been given. The internal structure of SPICE3 can be broken into six major functional packages and the corresponding data structures. Problems of modification and understanding the programming structure have been addressed. An alternative approach for writing the SPICE program has been proposed. The new SPICE program (object-SPICE) has been rewritten from SPICE3 by using object oriented programming paradigms. An outline of the internal structure of object-SPICE has been presented. A comparison of SPICE3 program and object-SPICE program was made. Advantages for further development and maintenance of object-SPICE have been addressed.

Someone who wishes to implement the object-SPICE program should refer to the document in the Appendixes. The details of each object of object-SPICE and the hierarchy tree are described in Appendix IV.

II. FURTHER DEVELOPMENT

In this section, we will discuss the limitations of the Averaged Piecewise Transform, the new nonlinear dependent source and the object-SPICE and then give some suggestions for the direction of the further development.

1. The limitation of the Averaged Piecewise Transform (APT)

Although the equivalent circuit model derived in this thesis is applied to the series resonant converters, it can be extended to other converters. For example, in PWM type converters, the natural frequency of state variables is extremely slower than the switching frequency of converters. The APT can also be used by applying the solutions

of state variables of that converters and then deriving time invariant equivalent circuits. Similarly, for another example, in phase shift resonant converters operating below resonant frequency, the natural frequency of the state variable is extremely higher than the switching frequency of the converters. At this mode of operation, another set of the state variables is applied to the APT. By using the same mechanism, a time-invariant equivalent model can be derived. The APT can basically apply to any kind of converters with actual solutions of state variables. However, the APT requires very complicate mathematical-manipulations and easily causes convergence problems in SPICE simulations. Therefore, the further development of the APT is to develop a new transform. The new transform should not require the actual solutions of the state variables in calculations in order to simplify the mathematical manipulations.

2. The limitation of the new nonlinear dependent source

Convergence problem arises from multiple solutions. When performing macromodel simulation in SPICE, the macromodel requires the new nonlinear dependent source. However, SPICE and the new dependent source use their own local convergence method to calculate solutions. It is difficult for the system to handle the multiple solution problem. Therefore, further work of the new source should be devoted to find or develop a global convergence method for the simulator to deal with the multiple solutions problem in simulations.

3. The limitation of object-SPICE

Totally, 50 classes have been developed inside object-SPICE. Almost all classes are derived from a single base class. A single hierarchy tree has been established for increasing both extendibility and readability. Four sets of function polymorphism have

been implemented into object-SPICE to reduce complexity. However, due to time constraint, the front end object reuses the codes from SPICE3. Therefore, it is difficult to understand and modify the program. Further work should be done to implement the front end object. In object-SPICE, the front end object contains function pointers to manipulate all user commands (e.g. RUN, PLOT, PRINT, SOURCE, AC, DC, TRAN, FOURIER). A linking table still exists and contains functions to implement the corresponding user commands. This affects the consistency and readability of the program. The direct way to replace the function pointers and the linking table is to inherit a set of new objects from the front end object in order to form a function polymorphism. The new set of objects contains all these functions and corresponding data structures. All functions then use a common function name (e.g. *COMM_func*). Each function in a new object implements its own version of *COMM_func*. Consequently, all indirect pointer references and linking table are removed. Such modification extends the hierarchy tree of object-SPICE at the front end object. All these functions are then included in the hierarchy tree of object-SPICE and can be found directly from the tree.

Appendix I

SPICE Simulation Techniques

I. INTRODUCTION

SPICE simulation techniques used in this thesis are described in the following section.

II. DESIGN OF A VOLTAGE-CONTROLLED-OSCILLATOR

In the full-transient simulation of resonant converters, the driving voltage is generated by switches controlled by a voltage-controlled oscillator (VCO). A simplified VCO using a Wien bridge oscillator has been proposed [5,6]. The circuit in SPICE is shown below:

```
*Varying Pulse generator
.SubCkt VCO nfin no
ba no 0 V=3*V(nc)
bR1 no n1 I=1e-6*V(nfin)*V(no,n1)
Rr1 no n1 1T
C1 n1 nc 159.1549431n ic=1
*{1u/Twopi}
bR2 nc 0 I=1e-6*V(nfin)*V(nc)
Rr2 nc 0 1T
C2 nc 0 159.1549431n ic=0
*{1u/Twopi}
.ends VCO
```

III. THE TECHNIQUE OF AC EQUIVALENT FULL-TRANSIENT ANALYSIS

As only transient analysis can be performed in the full-transient simulation, the small signal response is extracted from the steady state of the transient simulation [25]. The control-to-output response will be described by using the following procedures.

1. Modifying the control signal with a single-frequency FM in the SPICE lists below

```
va 8 0 sffm(0 700 38k 76m 5k)
rs 8 7 100k
d5 6 7 dz
d6 6 0 dz
```

where 700 is the magnitude of modulated signal, 38k is the switching frequency and 76m is modulation index and 5k is the small signal frequency.

2. Make sure that the converter has entered steady-state before measuring the output response, as the example below:

```
.tran 0.1us 12m 6m 0.1us uic
```

where 6m is the time to allow the converter to enter steady state. The simulated results is record from 6m to 12m (it should be more than the period of 5k small signal).

3. The output response can be extracted by .FOUR command in the SPICE:

```
.four 5k v(Ro) v(vf)
```

Where 5k is the small signal frequency, v(Ro) is the output voltage of the converter and v(vf) is the reference voltage. The .FOUR command is used to extract DC component of the fundamental frequency component of the Fourier analysis of v(Ro) and v(vf). It should be notice that the ratio of fundamental and DC magnitudes of v(Ro). The ratio is larger than 3% to compensate for the effects of switching noise but smaller than 10% to avoid nonlinearity [5].

IV. METHODS OF HANDLING CONVERGENCE PROBLEMS

The convergence problems in SPICE simulations usually occur in the bias point calculation or transient analysis. These problems are usually caused by two main factors: 1) discontinuities in the device equations that make solving the matrices difficult, and 2) constraints on the analysis technique that prevent the SPICE algorithms from solving the equations. To overcome convergence problems during the bias point calculation, we can use the following options:

- Setting the initial voltage at designed nodes at the beginning of the analysis procedure (`.nodeset v(no)=0`) allows SPICE to converge on difficult circuits.
- And using the option control (`.option itl1=1e6`) increase the maximum number of DC iterations.

SPICE simulations suffer convergence problems in a transient analysis. The following method is adopted to handle these problems.

- Increasing the maximum number of iterations (`.option itl4=500`).
- Reducing the maximum simulation time step in a transient option.

Appendix II

Mathematica Techniques

I. INTRODUCTION

Mathematica is a software program basically combines symbolic and numerical calculations, plots, graphics programming and interactive environment. Generally, the program can be divided into two parts: The Mathematica kernel and the front end. The Mathematica kernel is the part of the program which actually performs the calculations (consists of several 100000 lines of source code written in an object oriented extension of C). Standard C code is produced by a precompilation. Since there is no assumptions about the target computer are made, the same source code can be compiled essentially on all systems. Therefore, the same kernel program can be used on a large variety of computers. The front ends is used to handle the interaction between the user and the kernel. It is called notebooks. These notebooks consist of Mathematica input and output, text and graphics. The two-dimensional typeset input and output is used. The Mathematica techniques used in this report are illustrated in follows.

II. TECHNIQUES

1. A function called `f`, which is a sine function with amplitude (Ao) and phase (th), is defined. The Mathematica command to define this function is

```
f[t_] = Ao Sin[t-th];
```

2. Using the command `ExpToTrig[expr]` converts exponential form in *expr* to trigonometric functions. For example,

```
ExpToTrig[i3 Exp[-I Wo]]
```

Results

```
i3 (Cos[Wo]-I Sin[Wo])
```

3. Getting the real and imaginary part of the expression, we use the commands `ComplexExpand[re[expr]]` and `ComplexExpand[Im[expr]]`. Using the equation in 2 gives

```
ComplexExpand[Re[i3 (Cos[Wo]-I Sin[Wo])]]

ComplexExpand[Im[i3 (Cos[Wo]-I Sin[Wo])]]
```

Resuts

```
i3 Cos[Wo]

-i3 Sin[Wo]
```

4. Calculating the integration, the two-dimensional form can be used to enter the expression. The integrate are entered in two-dimensional form by using a palette in Mathematica.

$$\int_{a}^{b} \mathtt{Sin[x]}\,dx$$

Results

```
Cos[a]-Cos[b]
```

Appendix III

The Netlist file of Averaged Piecewise Macromodel for the Series Resonant Converter

```
Spice3 modeling of series resonant converter
*.Param L = 197u
*.Param C = 100n
*.Param Vs = 14
*.Param wo = {1/SQRT(L*C)}= 225302.9545
*.Param 1_wo = {SQRT(L*C)}=
*.Param Zo = {SQRT(L/C)}=44.38468204
*.Param 1_Zo = {SQRT(C/L)}=
*.Param Rs = 1u
*
Vs ns 0 14
bis ns 0 i=v(nido)+v(niqo)
*
bio 0 no i=v(niqo)-v(nido)
Co no 0 14.2u

Ro no 0 88.76936408
*
bvsor nr1 nr5 v=44.38468204*i(vii)
bchr nr5 0 v=-(-2*v(no)+v(nao))*(1+cos(v(ntw)))*v(nsin1)/(v(ntw))
Rr nr1 nr2 1u
Lr nr2 nr3 197e-6
Vir nr3 nr4 0
Cr nr4 0 100e-9
bvr 0 nr4 i=v(ni4)/44.38468204
bstr 0 nr4 i=-2*100e-9*v(nws)*(v(ns)-v(no)*v(ncos)+(2*v(no)-
v(nao))*(1+cos(v(ntw)))*v(ncos1)/2)/(3.14159265359)
*
bvsoi ni1 ni5 v=-44.38468204*i(vir)
bchi ni5 0 v=(-2*v(no)+v(nao))*sin(v(ntw))*v(nsin1)/v(ntw)
Ri ni1 ni2 1u
Li ni2 ni3 197e-6
Vii ni3 ni4 0
Ci ni4 0 100e-9
bvi 0 ni4 i=-v(nr4)/44.38468204
bsti 0 ni4 i=-100e-9*v(nws)*((2*v(no)+v(nao))*v(nsin)-(2*v(no)-
v(nao))*v(nsin2))/(2*3.14159265359)
*
bas nas 0 v=44.38468204*i(vir)-v(ni4)
Ras nas 0 1e6
bac nac 0 v=i(vii)*44.38468204+v(nr4)
Rac nac 0 1e6
*
bthe nthe 0 v=atan(v(nas)/v(nac))
Rthe nthe 0 1e6
bsin nsin 0 v=sin(v(nthe))
Rsin nsin 0 1e6
bcos ncos 0 v=cos(v(nthe))
Rcos ncos 0 1e6
bntw ntw 0 v=3.14159265359*225302.9545/v(nws)
Rntw ntw 0 1e6
bsin1 nsin1 0 v=sin(v(ntw)-v(nthe))
```

```
Rsin1 nsin1 0 1e6
bcos1 ncos1 0 v=cos(v(ntw)-v(nthe))
Rcos1 ncos1 0 1e6
bsin2 nsin2 0 v=sin(2*v(ntw)-v(nthe))
Rsin2 nsin2 0 1e6
bcos2 ncos2 0 v=cos(2*v(ntw)-v(nthe))
Rcos2 ncos2 0 1e6
*
bao nao 0 v=1e9*(sqrt(v(nas)^2+v(nac)^2)+2*v(no)*(1+v(nthe)/v(ntw))-
V(nao))
RAo nao 0 1e9
bido nido 0 v=v(nws)*100e-9*v(nao)*(v(ncos)-1)/3.14159265359
RIDo nido 0 1e6
biqo niqo 0 v=(v(nws)*100e-9/3.14159265359)*(v(nao)-2*v(no))*(1-
v(ncos1))
RIQo niqo 0 1e6
*
vfs nfs 0 38k ac 1
*PWL(0 37.93k 1.6m 37.93k 1.602m 35.93k 4.1m 35.93k 4.102m 37.93k
6.6m 37.93k)
Rfs nfs 0 1e6
bws nws 0 v=2*3.14159265359*v(nfs)
Rws nws 0 1e6
*
.nodeset v(no)=0
.ic v(no)=0
.option itl1=1e6 itl2=1e6 reltol=0.001 itl4=500 trtol=0.1
*.tran 0.600u 6m 0 0.6u uic
*.dc Vfs 35k 60k 0.2k
.ac dec 101 10 35k
*.print ac vdb(no)
.End
```

Appendix IV

Classes of object-SPICE

I. OVERVIEW OF CLASSES IN OBJECT-SPICE

Object-SPICE contains approximately 800 files in C++ language[2], it was written for use on the personal computer. The computer operates under the Windows NT operating system. The program is developed with a clear internal structure that a detailed hierarchy chart of the program is established in three major categories shown in figure A5-1. The SPICE simulator is the main portion of the program. The simulator calls the specific devices for making simulation and performing analyses.

[2] C++ language is a programming language supporting the OOP paradigm.

Hierarchy Chart
The new SPICE Class

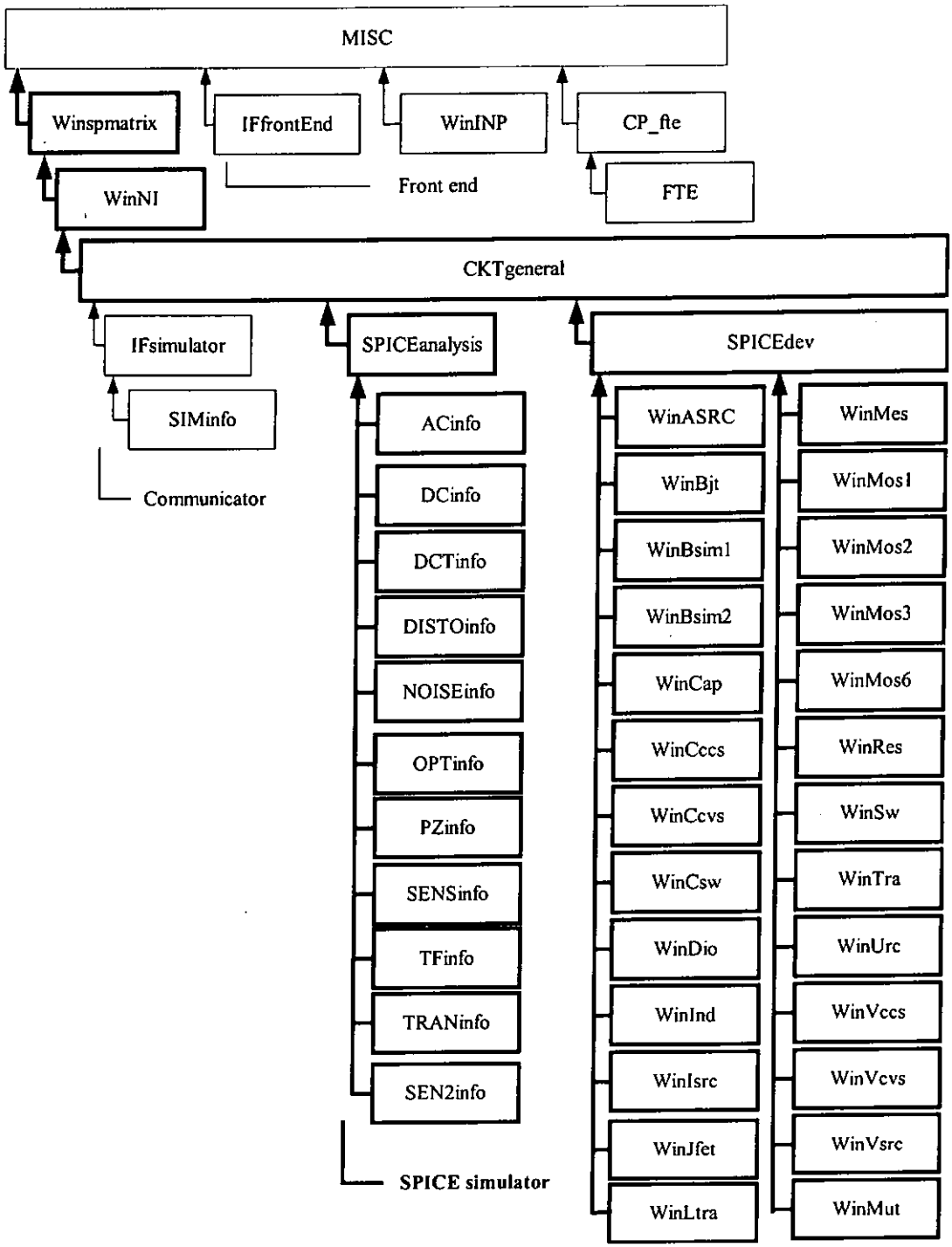


Figure A5-1, The internal structure of object SPICE simulation program.

SPICE simulator

In this category, all internal algorithms, device equations and analysis methods are encapsulated in forty-two classes shown in figure A5-2. The SPICE simulator performs the specific analysis through the SPICEanalysis class and its derivatives. The simulator calls on the SPICEdev class for setting up a set of circuit equations. By using the basic algorithm with the WinNI class for the numerical iteration and integration, the simulator then performs factorization to complete the solution of the circuit equations on the Winspmatrix class. The overview of each class in this category is given below.

1. A Winspmatrix class handles sparse matrices in SPICE. The class provides matrices allocation, elements creation in matrices and calculation of the solution of circuit equations.

2. A WinNI class is responsible for the basic numerical iteration and integration methods used by SPICE as well as places device equations in the circuit simulation into parse matrices.

3. A CKTgeneral class provides the basic functionality of the circuit simulation. It establishes the initial condition, determines the operating point of the simulated circuit and drives the various devices to setup their parameters for the circuit simulation.

4. A SPICEanalysis class encapsulates all type of analysis provides by SPICE. For each type analysis, by calling on device functions, numerical methods, matrix operations and output operations as needed to perform the required analysis.

5. A SPICEdev class handles all type of devices provided by SPICE. For each type of device, there is a group of necessary functions for performing all necessary jobs for the simulation on instances and models of that type.

The detailed descriptions of these classes are given in next section.

The relationship of these classes is shown in figure A5-2. The Winspmatrix class that is at the highest level in the hierarchy tree within this category, is a single base class for the SPICE simulator. The Winspmatrix class provides a useful capability to all classes derived from it. The second level of the tree is the WinNI class that is derived form Winspmatrix class. The CKTgeneral class is the child class of the WinNI class providing basic functionality for the SPICEanalysis and SPICEdev classes. All device classes and analyses classes are at the lowest level, handling a particular device equation and analysis method used by the simulator respectively.

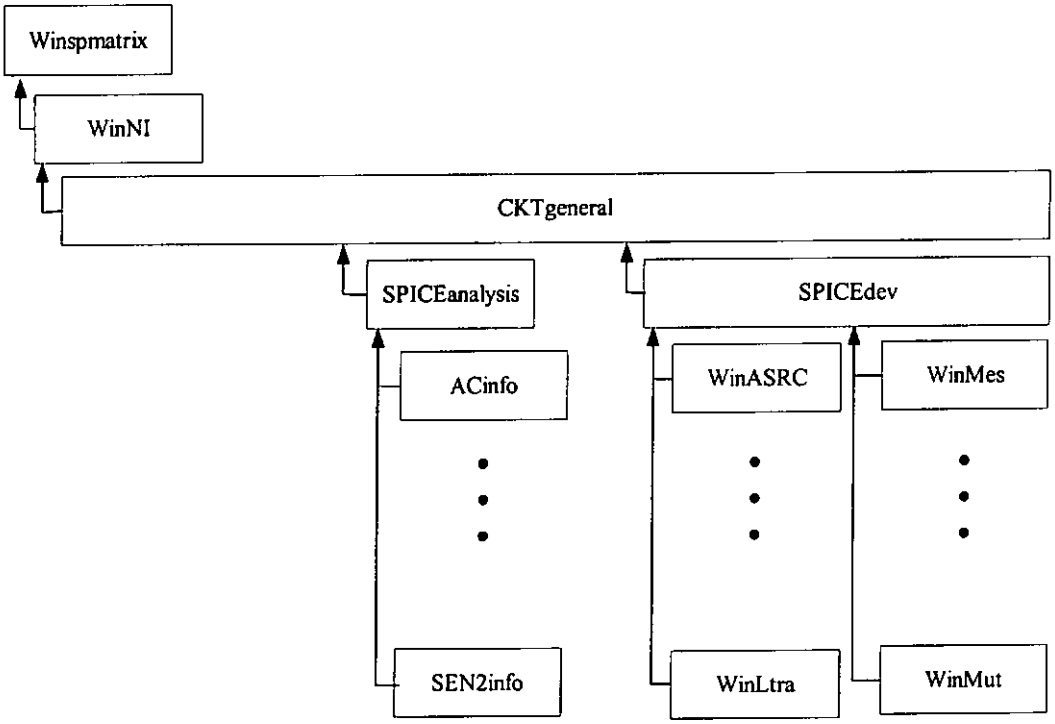


Figure A5-2, The hierarchy chart of SPICE simulator.

Front end

The second category of classes in the hierarchy chart shown in figure A5-1 is the front end of the program that contains four classes. These classes are: CP_fte, FTE, WinINP and IFfrontEnd classes. They work for the development of an input language, graphics output and data management. The overview of each class in this category is given below.

1. A CP_fte class handles the front end command loop of SPICE. It provides an interactive command mode for SPICE. It is responsible for reading the input commands from the user, parsing the command line to perform the necessary commands provided by SPICE and recording the history of the command line.

2. A FTE class reads the input circuit via netlist file and runs the circuit. It then is responsible for plotting results generated by the SPICE simulator, writing an output raw file and printing out the resource usage information of SPICE.

3. A WinINP class handles the tools necessary to parse netlist format input. It is responsible for examining all of the SPICE input line, allocating and initializing the symbol tables which is used for storing the node names of the simulated circuit and performing the evaluation at each node.

4. An IFfrontEnd class provides output functionality for SPICE. It is responsible for recording simulation data that result from the analyses preformed by the SPICE simulator.

The relationship of these classes is shown in figure A5-3. All classes in this category are derived from the MISC class that is root class of the SPICE program providing utilities functions for its derivative. Both IFfrontEnd and WinINP classes that are independent each other are located at the second level of the hierarchy chart. The CP_fte class also is another child class of the MISC class and gives basic functionality for the FTE class. The FTE class derived from the CP_fte class providing a specific operation of reading a simulation circuit and writing the results data.

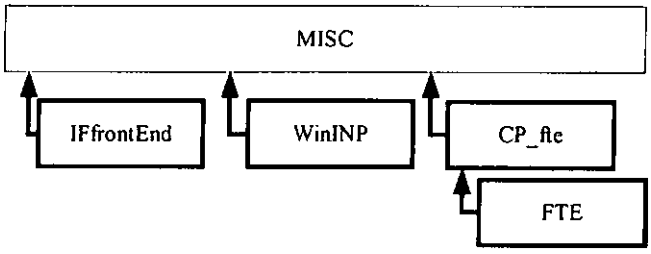


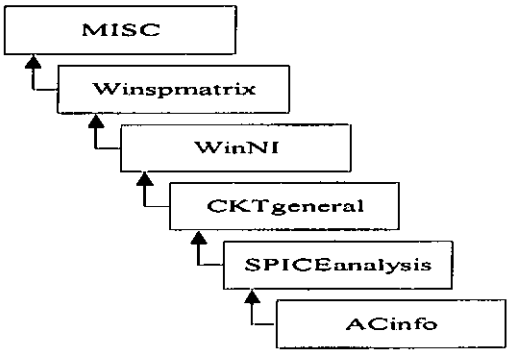
Figure A5-3, The hierarchy chart of front end.

Communicator

In this category, the IFsimulator and its derivative classes shown in figure A5-1 are derived from the CKTgeneral class that gives a set of basic functions for circuit simulation. They support methods for communication between the front end (CP_fte, FTE, WinINP and IFfrontEnd classes) and the SPICE simulator (Winspmatrix, WinNI, SPICEdev and SPICEanalysis classes). Classes that are in this category are responsible for creating a new circuit structure and a new set of analysis parameters for the SPICE simulator. The program then performs simulations requested by the front end of the program through the communicator. The detailed descriptions of classes in both front end and communicator categories are given in the following section.

II. THE CLASSES OF OBJECT SPICE

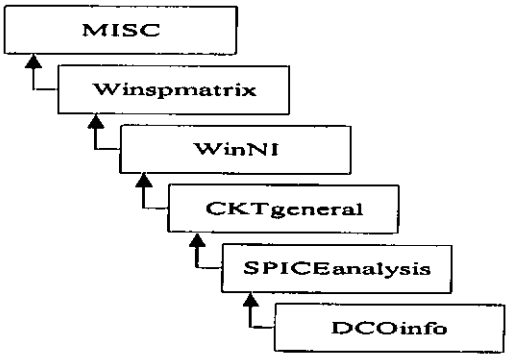
ACinfo



The ACinfo class encapsulates an AC analysis. It performs the AC analysis by computing a DC operating point and all the necessary small signal parameters, then sweeps all AC sources through a set of frequencies, computing the AC small-signal response. The ACinfo class stores the necessary small signal parameters values with The ACinfo::setParms function. There are three mutually exclusive keywords: *dec*, *oct* and *lin* and the last one specified overrides the others. The AskQuest member

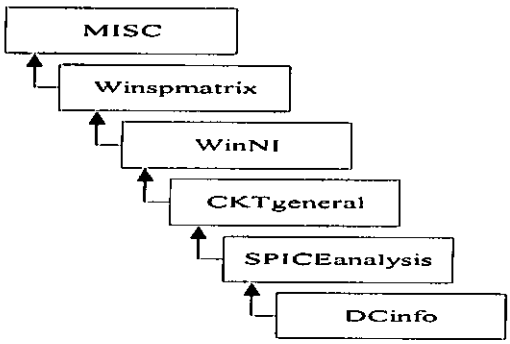
function performs a query of those parameters. The ACinfo class actually performs the analysis with the ACinfo::an_func member function.

DCOinfo



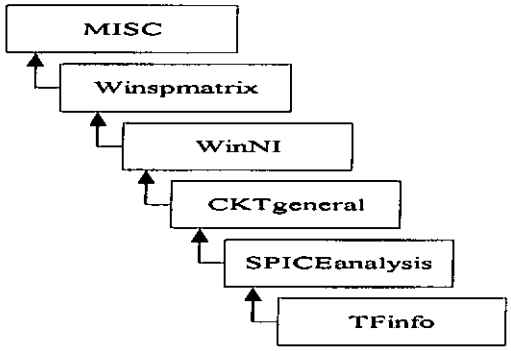
The DCOinfo class provides a DC analysis that all capacitances are open circuited and all inductances are shorted. The result is both outputs as an analysis and left in the CKTrhsOld[3] vector for future use by other analyses. DCOinfo responsible for the DC analysis calls the DCOinfo::an_func member function, which uses the Newton-Raphson iteration to solve the circuit. Besides the simple the Newton-Raphson iteration, the DCOinfo::an_func has Gmin Stepping and source stepping techniques. Both askQuest and setParm member functions always return E_BADPARM, since the DC operating point analysis does not support any options.

DCTinfo



The DCTinfo class is responsible for a DC transfer curve analysis, which sweeps a voltage or current source through a set of values and stores the output variables for sequential source. The class assumes only two levels of nesting in the loop for the analysis. It responsible for setting the necessary parameters calls The DCTinfo::setParm function. The class provides the DC transfer curve analysis with The DCTinfo::an_func member function. The DCTinfo::askQuest function always return E_BADPARM owing to the historical reasons.

TFinfo

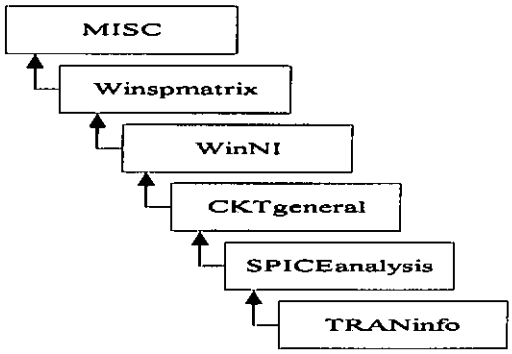


The TFinfo class allows a small signal DC transfer function to be computed. It supports calculations of the DC small-signal value of the transfer function (output/input), input resistance, and output resistance. The setParm member function stores specified parameters into the transfer function analysis structure. The TFinfo class uses the TFinfo::askQuest function to query parameters of a transfer function analysis. This function always returns E_BADPARM, the actual query code still to be written as the need for it arises. The TFinfo class responsible for the actual analysis calls the TFinfo::ana_func function.

---

[3] CKTrhsOld is the right hand side vector for the matrix package from the previous iteration which now contain the solution for that iteration.

TRANinfo

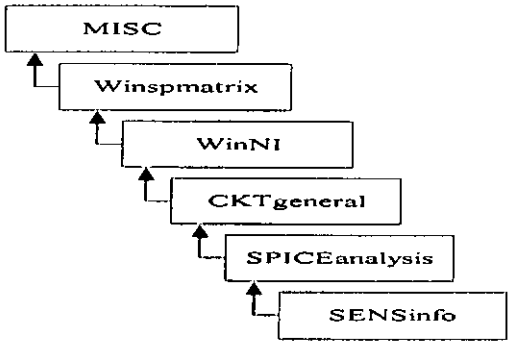


The TRANinfo class encapsulates a transient analysis. In order to query parameters of the analysis, set the needed paramaters and perform the transient analysis call the TRANinfo member functions: askQuest, setParm and an_func. Besides these basic members, the TRANinfo class provides three other member functions:

- CKTaccept, a function that allows each device to perform any once per time point cleanup or preparation for the next time point.
- CKTclrBreak, a function that is used to clear a breakpoint.
- CKTtrunc, a function that is used to calculate the truncation error.

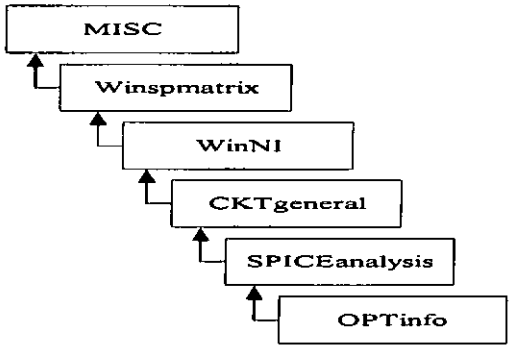
The TRANinfo class also has an initialized function name an_init that is used to set up the analysis.

SENSinfo



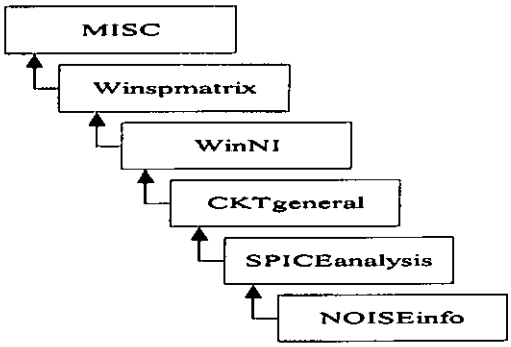
The SENSinfo class supprots a sensitivity analysis. To set up and query parameters in sensitivity structures that the simulator uses in the sensitivity analysis, use the SENSinfo member functions setParm and askQuest. In addition, the analysis is performed by calling the SENSinfo member function an_func. The sensitivity analysis through private member functions in the SENSinfo class provides for setting parameters, incrementing frequency, loading devices and setting the devices temperature.

OPTinfo



The OPTinfo class provides an overall control of various parameters effecting the analysis operations, such as numerical tolerances, operating temperature, DC iteration limit, number of Gmin steps etc. The OPTinfo class supports member function for setting values of parameters, which SPICE placed on a ".options" card, working with the OPTinfo::setParm function. OPTinfo handles requests of overall information about the circuit with the OPTinfo::askQuest member function. The class does not provides the an_func member function, since it does not perform an actual analysis but use a same interface of another analysis.

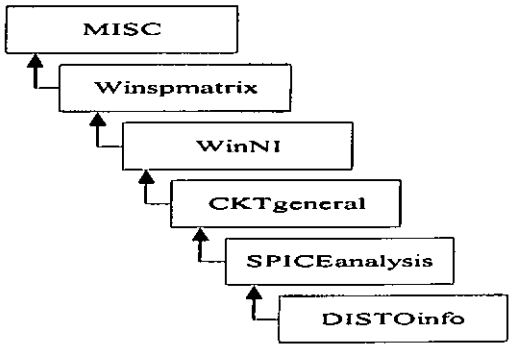
NOISEinfo



The NOISEinfo class is responsible for a noise anlaysis of the circuit. It uses NOISEinfo::setParm, NOISEinfo::askQuest to store and request parameters of the noise analysis. NOISEinfo class performs the analysis by calling member function an_func. Besides these basic member functions in the class, the NOISEinfo class has another member function:

- CKTnoise, a function that works for naming and evaluating all of the noise sources in the circuit.

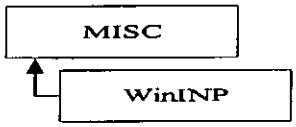
DISTOinfo



The DISTOinfo class handles a small-signal distortion analysis of the circuit. The class responsible for storing parameters of the analysis calls the DISOinfo::setParm member function, which is similar to the ACinfo::setParm function which has three mutually exclusive keywords: *dec*, *oct*, and *lin*, but the DISTOinfo::setParm provides an additional parameter F2OVERF1. It also supports query of parameters with the

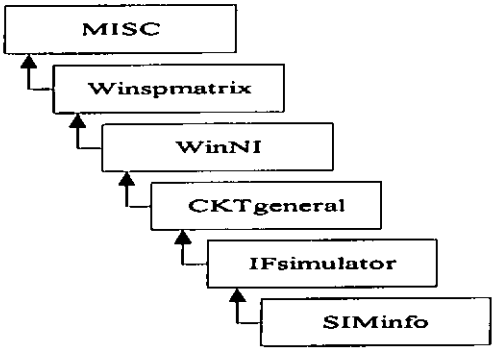
DISTOinfo member function askQuest. The class works for setting up and performing the analysis by calling DISTOinfo::CKTdisto and DISTOinfo::an_func.

WinINP



The WinINP class provides tools necessary to parse netlist format input. It responsible for examining all of the SPICE input line except for ".MODEL" cards calls the WinINP member function INPpas2. The INPpas2 member function handles the individual device lines by calling a separate member function. These functions name INP2X where X is the keyletter used by SPICE for the first letter of the device name. The class handles ".MODEL" card by using WinINP::INPpas1 member function. To allocate and initialize symbol table which is used for node names of the circuit calls the WinINP::INPtabInit member function. The WinINP class frees the space allocated for the symbol table with the WinINP::INPtabEnd member function. It calls WinINP::INPgetTak to find a next token on the input line given. To performs evaluating numbers, inserting string into the string table operation uses the WinINP::INPgetValue member function. Both WinINP::INPerrCat and WinINP::INPerr member functions handle error message during the parsing operation.

SIMinfo



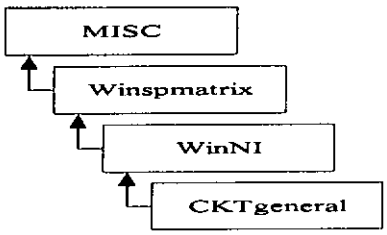
The SIMinfo class supports communication between a front end (IfrontEnd, WinINP, CP_fte and FTE classes) and SPICE simulator (SPICEanalysis, SPICEdev, WinNI and Winspmatrix classes). The class provides six areas of the functionality:

- Circuit area, which allocates a structure necessary to describe a circuit and sets a pointer to the newly allocated structure by calling the SIMinfo::newCircuit member function. Besides newCircuit, the area handles freeing the structure with the SIMinfo::deleteCircuit member function.
- Node area, which creates a node in the circuit and sets a pointer to the newly node by using the SIMinfo::newNode member function. Specifying a node to be the ground node of the circuit and connecting a terminal of an instance to a node uses the SIMinfo::bindNode member function. The area sets parameters of a node by calling SIMinof::setNodParm.
- Instance area, which creates a device (instance) of a specified type in the circuit using the SIMinfo::newInstance member function. It provides SIMinfo::setInstanceParm and SIMinfo::askInstanceQuest to set and request parameters of an instance.
- Model area, which creates a model of a device in the circuit using SIMinfo::newModel. It provides SIMinfo::setModelParm and SIMinfo::askModelParm to set and query parameters of the device model.

Besides these member functions, it allows to find and remove a specified model from the circuit by calling SIMinfo::find and SIMinfo::deleteModel.

- Analysis area, which creates a new analysis using the SIMinfo class member function, newAnalysis. To set and ask the analysis parameters calls SIMinfo::setAnalysisParm and SIMinfo::askAnalysisQuest member funcitons. It responsible for locating the analysis uses the SIMinfo::findAnalysis member function.
- Task area, which creates a new task (a group of analyses) associated with circuit using SIMinfo::newTask. Finding and deleting the tasks uses the SIMinfo class member functions, findTask and deleteTask.

CKTgeneral

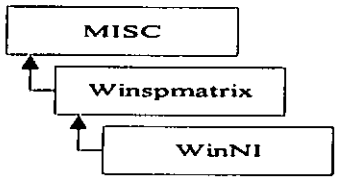


The CKTgeneral class is the base class for SPICEanalysis, SPICEdev and IFsimulator classes. The class encapsulates the original routines in CKT package used by SPICE3. It supports establishing the initial conditions before performing an analysis, determining operating point of the circuit for the SPICEanalysis class and outputting analysis results by calling CKTgeneral::CKTic, CKTgeneral::CKTop and CKTgeneral::CKTdump member functions. CKTgeneral::CKTacDump is responsible for the outputs of AC analysis. The CKTgeneral class also provides member functions for driving the SPICEdev class:

- CKTpName, a driving function that takes and sets parameters by name on a device.
- CKTtemp, a driving function that calls temperature dependency member functions in the SPICEdev class.
- CKTsetup, a driving function that calls all devices setup member functions in the circuit.
- CKTunsetup, a driving function that frees CKTstates[4] vectors used by devices in the circuit.
- CKTtypelook, a member function that finds a device in the circuit.

Besides these member functions, the class has CKTgeneral::CKTdltNNum, CKTgeneral::CKTnames and CKTgneral::CKTnum2nod to generate a namelist of nodes and equations, convert a node number to a node pointer and delete a node from the circuit.

WinNI



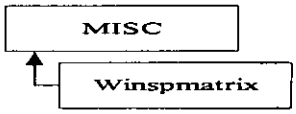
The WinNI class provides basic functionality for the CKTgeneral class and encapsulates the numerical iteration and integration used by SPICE. It has the following services:

- The WinNI class has the WinNI::CKTload member function to load all devices in the circuit at each iteration of transient and operating point analysis. To use the WinNI::CKTacLoad member function drives device loading member functions in

the SPICEdev class for a AC analysis. It provides WinNI::CKTnodName for output a name of a node in the circuit. Also, to initialize, reinitialize and free data structure used by the WinNI class call WinNI::NIinit, WinNI::NIreinit and WinNI::NIdestroy member functions.

- The WinNI class responsible for the Newton-Raphson iteration, numerical integration and finding roots of a complex polynomial calls WinNI::NIiter, WinNI::NIintegrated and WinNI::NIpzMuller member functions. The class supports computing an integration coefficient by using the WinNI::NIconCof member function.

Winspmatrix



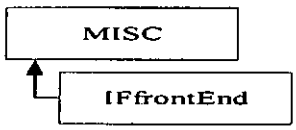
The Winspmatrix class is the base class for SPICE simulator classes. It handles all routines in the spare matrix package used by SPICE3. It provides Winspmatrix::SMPnewMatrix for initializing the spare matrix. To clear to zero all entries in the spare matrix uses Winspmatrix::SMPclear or Winspmatrix::SMPcClear member functions. The SMPcClear is responsible for complex entries. First, the class handles removing zero from the diagonal of the spare matrix with the Winspmatrix::SMPpreorder member function, then calls Winspmatrix::SMPreorder, Winspmatrix::SMPluFac and Winspmatrix::SMPsolve to reorder the spare matrix, perform L-U decompositions and solve matrix equations. The class handles a complex matrix using SMPcReorder, SMPcluFac and SMPcSolve member functions instead. It also provides Winspmatrix::SMPfindElt and Winspmatrix::SMPmakeElt

---

[4] CKTstates are vectors that are used to store the per time point data.

member functions for finding an element in the spare matrix, creating the element in the spare matrix. Besides these functions, it has Winspmatrix::SMPmatsize and Winspmatrix::SMPprint member functions to measure size of the matrix and print the matrix.

IFfrontEnd



The IFfrontEnd class provides output functionality for SPICE simulator classes. The class responsible for recording time for statistics gathering uses the IFfrontEnd::IFseconds member function. It provides the IFfrontEnd::IFerror member function for SPICE simulator classes to output messages. To initialize and free data type uses by IFfrontEnd calls IFfrontEnd::newUid and IFfrontEnd::IFdelUid member functions. Besides these functions, the class handles output for the SPICEanalysis class with five member functions:

- OUTpBeginPlot, a function that describes the plot to be produced.
- OUTpData, an output function that is called repeatedly and outputs data for the SPICEanalysis class per point.
- OUTendPlot, a function that tells that all data has been outputted.
- IFpauseTest, a function that indicates that the front end classes wants control.
- OUTattributes, a function that allows the SPICEanalysis class to output additional information other than linear scale plot.

Appendix V

The cross-reference table of functions for SPICE3 and object-SPICE

SPICE3f3 function	Object SPICE member functions	accessibility
Circuit glue		
CKTacDump	CKTgeneral::CKTacDump(CKTcircuit *, double , GENERIC *)	protected
CKTacLoad	WinNI::CKTacLoad(CKTcircuit*)	protected
CKTaccept	TRANinfo::CKTaccept(CKTcircuit *)	private
CKTconvTest	WinNI::CKTconvTest(CKTcircuit*)	protected
CKTfndBr	CKTgeneral::CKTfnBranch(CKTcircuit *,IFuid)	protected
CKTic	CKTgeneral::CKTic(CKTcircuit *)	protected
CKTload	WinNI::CKTload(GENERIC *)	protected
CKTpzLoad	PZinfo::CKTpzLoad(CKTcircuit *,Spcomplex *,int)	private
CKTsetup	CKTgeneral::CKTsetup(CKTcircuit *)	protected
CKTpzSetup	PZinfo::CKTpzSetup(CKTcircuit *)	private
CKTtemp	CKTgeneral::CKTtemp(CKTcircuit *)	protected
CKTtrunc	TRANinfo::CKTtrunc(CKTcircuit *,double *)	private
Analysis packages		
ACsetParm	ACinfo::setParm(CKTcircuit *,GENERIC *,int ,IFvalue *)	public
ACaskQuest	ACinfo::askQues(CKTcircuit *,GENERIC *,int ,IFvalue *)	public
ACan	ACinfo::an_func(CKTcircuit *,int)	public
DCOaskQuest	DCOinfo::askQues(CKTcircuit *,GENERIC *,int ,IFvalue *)	public
DCOsetParm	DCOinfo::setParm(CKTcircuit *,GENERIC *,int ,IFvalue *)	public
DCop	DCOinfo::an_func(CKTcircuit *,int)	public
DCTaskQuest	DCTinfo::askQues(CKTcircuit *,GENERIC *,int ,IFvalue *)	public
DCTsetParm	DCTinfo::setParm(CKTcircuit *,GENERIC *,int ,IFvalue *)	public
DCtrCurv	DCTinfo::an_func(CKTcircuit *,int)	public
TFanal	TFinfo::an_func(CKTcircuit *,int)	public
TFaskQuest	TFinfo::askQues(CKTcircuit *,GENERIC *,int ,IFvalue *)	public
TFsetParm	TFinfo::setParm(CKTcircuit *,GENERIC *,int ,IFvalue *)	public
TRANaskQuest	TRANinfo::askQues(CKTcircuit *,GENERIC *,int ,IFvalue *)	public
TRANsetParm	TRANinfo::setParm(CKTcircuit *,GENERIC *,int ,IFvalue *)	public
DCtran	TRANinfo::an_func(CKTcircuit *,int)	public
PZan	PZinfo::an_func(CKTcircuit *,int)	public
PZaskQuest	PZinfo::askQues(CKTcircuit *,GENERIC *,int ,IFvalue *)	public
PZsetParm	PZinfo::setParm(CKTcircuit *,GENERIC *,int ,IFvalue *)	public
SENSask	SENSinfo::askQues(CKTcircuit *,GENERIC *,int	public

	,IFvalue *)	
SENSsetParam	SENSinfo::setParm(CKTcircuit *,GENERIC *,int ,IFvalue *)	public
sens_sens	SENSinfo::an_func(CKTcircuit *,int)	public
Analysis control		
CKTsetOpt	OPTinfo::setParm(CKTcircuit *,GENERIC *, int ,Ifvalue *)	public
CKTacct	OPTinfo::askQuest(GENERIC *, GENERIC *,int ,IFvalue *)	public
Utility		
CKTclrBreak	TRANinfo::CKTclrBreak(CKTcircuit *)	private
CKTdump	CKTgeneral::CKTdump(CKTcircuit *, double ,GENERIC *)	protected
CKTlinkEq	SPICEdev::CKTlinkEq(CKTcircuit *,CKTnode *)	protected
CKTmkCur	SPICEdev::CKTmkCur(CKTcircuit *,CKTnode **,IFuid ,char *)	protected
CKTmkNode	SPICEdev::CKTmkNode(CKTcircuit *,CKTnode **)	protected
CKTmkVolt	SPICEdev::CKTmkVolt(CKTcircuit *,CKTnode **,IFuid ,char *)	protected
CKTnames	CKTgeneral::CKTnames(CKTcircuit *, int ,IFuid **)	protected
CKTnum2nod	CKTgeneral::CKTnum2nod(CKTcircuit *, int)	protected
CKTpModName	SPICEdev::CKTpModName(char *,Ifvalue *,CKTcircuit *,int ,IFuid ,GENmodel **)	protected
CKTpName	CKTgeneral::CKTpName(char *, IFvalue *,CKTcircuit *,int ,char *,GENinstance **)	protected
CKTsetBreak	SPICEdev::CKTsetBreak(CKTcircuit *, double)	protected
CKTterr	SPICEdev::CKTterr(int ,CKTcircuit *,double *)	protected
CKTtypelook	CKTgeneral::CKTtypelook(char *)	protected
CKTbreakDump	SPICEdev::CKTbreakDump(CKTcircuit *)	protected
Numerical package		
NiacIter	WinNI::NIacIter(CKTcircuit *)	protected
NIcomCof	WinNI::NIcomCof(CKTcircuit *)	protected
NIconvTest	WinNI::NIconvTest(CKTcircuit *)	protected
Nidestroy	WinNI::NIdestroy(CKTcircuit *)	protected
NIinit	WinNI::NIinit(CKTcircuit *)	protected
NIintegrate	WinNI::NIintegrate(CKTcircuit *, double *, double *, double , int)	protected
NIiter	WinNI::NIiter(CKTcircuit *, int)	protected
NIpzMuller	WinNI::NIpzMuller(PZtrial **, PZtrial *)	protected
NIpzComplex	WinNI::NIpzComplex(PZtrial **, PZtrial *)	protected
NIpzSym2	WinNI::NIpzSym2(PZtrial **, Pztrial *)	protected
NIpzSym	WinNI::NIpzSym(PZtrial **, Pztrial *)	protected
NIreinit	WinNI::NIreinit(CKTcircuit *)	protected
NIsenReinit	WinNI::NIsenReinit(CKTcircuit *)	protected
Sparse Matrix Package		
SMPnewMatrix	Winspmatrix::SMPnewMatrix(SMPmatrix **)	protected
SMPdestroy	Winspmatrix::SMPdestroy(SMPmatrix *)	protected

SMPaddElt	Winspmatrix::SMPaddElt(SMPmatrix *, int ,int ,double)	protected
SMPclear	Winspmatrix::SMPclear(SMPmatrix *)	protected
SMPfindElt	Winspmatrix::SMPfindElt(SMPmatrix *, int ,int ,int)	protected
SMPmakeElt	Winspmatrix::SMPmakeElt(SMPmatrix *,int ,int)	protected
SMPnewNode	Winspmatrix::SMPnewNode(int ,SMPmatrix *)	protected
SMPgetError	Winspmatrix::SMPgetError(SMPmatrix *,int ,int)	protected
SMPmatSize	Winspmatrix::SMPmatSize(SMPmatrix *)	protected
SMPpreOrder	Winspmatrix::SMPpreOrder(SMPmatrix *)	protected
SMPreorder	Winspmatrix::SMPreorder(SMPmatrix *,double ,double ,double)	protected
SMPluFac	Winspmatrix::SMPluFac(SMPmatrix *,double ,double)	protected
SMPsolve	Winspmatrix::SMPsolve(SMPmatrix *,double ,double)	protected
SMProwSwap	Winspmatrix::SMProwSwap(SMPmatrix *,int ,int)	protected
SMPcolSwap	Winspmatrix::SMPcolSwap(SMPmatrix *,col1 ,col2)	protected
SMPprint	Winspmatrix::SMPprint(SMPmatrix *,FILE)	protected
SMPfillup	Winspmatrix::SMPfillup(SMPmatrix *)	protected
SMPcClear	Winspmatrix::SMPcClear(SMPmatrix *)	protected
SMPcLUfac	Winspmatrix::SMPcLUfac(SMPmatrix *,double)	protected
SMPcProdDiag	Winspmatrix::SMPcProdDiag(SMPmatrix *, complex *, int *)	protected
SMPcReorder	Winspmatrix::SMPcReorder(SMPmatrix *,double ,double ,int *)	protected
SMPcSolve	Winspmatrix::SMPcSolve(SMPmatrix *,double *,double *,double *,double *)	protected
Input package		
INPpas1	WinINP::INPpas1(GENERIC *,card *,INPtables *,GENERIC *)	public
INPpas2	WinINP::INPpas2(GENERIC *,card *,INPtables *,GENERIC *)	public
INP2B	WinINP::INP2B(GENERIC*,INPtables*,card*)	private
INP2C	WinINP::INP2C(GENERIC*,INPtables*,card*)	private
INP2D	WinINP::INP2D(GENERIC*,INPtables*,card*)	private
INP2E	WinINP::INP2E(GENERIC*,INPtables*,card*)	private
INP2F	WinINP::INP2F(GENERIC*,INPtables*,card*)	private
INP2G	WinINP::INP2G(GENERIC*,INPtables*,card*)	private
INP2H	WinINP::INP2H(GENERIC*,INPtables*,card*)	private
INP2I	WinINP::INP2I(GENERIC*,INPtables*,card*)	private
INP2J	WinINP::INP2J(GENERIC*,INPtables*,card*)	private
INP2K	WinINP::INP2K(GENERIC*,INPtables*,card*)	private
INP2L	WinINP::INP2L(GENERIC*,INPtables*,card*)	private
INP2M	WinINP::INP2M(GENERIC*,INPtables*,card*)	private
INP2O	WinINP::INP2O(GENERIC*,INPtables*,card*)	private
INP2Q	WinINP::INP2Q(GENERIC*,INPtables*,card*,GENERIC*)	private
INP2R	WinINP::INP2R(GENERIC*,INPtables*,card*)	private
INP2S	WinINP::INP2S(GENERIC*,INPtables*,card*)	private
INP2T	WinINP::INP2T(GENERIC*,INPtables*,card*)	private
INP2U	WinINP::INP2U(GENERIC*,INPtables*,card*)	private
INP2V	WinINP::INP2V(GENERIC*,INPtables*,card*)	private
INP2W	WinINP::INP2W(GENERIC*,INPtables*,card*)	private

INP2Z	WinINP::INP2Z(GENERIC*,INPtables*,card*)	private
INP2dot	WinINP::INP2dot(GENERIC*,INPtables*,card*,GENERIC*,GENERIC*)	private
Interface functions		
IFeval	IFparseTree::IFeval(IFparseTree *,double *,double *,double *)	public
IFnewUid	IFfrontEnd::IFnewUid(GENERIC *,Ifuid *,Ifuid ,char *, int GENERIC **)	public
Convenience functions		
INPaName	WinINP::INPaName(char *,Ifvalue *,GENERIC *,int *)	public
INPapName	WinINP::INPapName(GENERIC *,int type,GENERIC *,char *,IFvalue *)	public
INPcaseFix	WinINP::INPcaseFix(char *)	public
INPdevParse	WinINP::INPdevParse(char **,GENERIC *,int ,GENERIC *,double *,int *,INPtables *)	private
INPdoOpts	WinINP::INPdoOpts(GENERIC *,GENERIC *,card *,INPtables *)	private
INPdomodel	WinINP::INPdomodel(GENERIC *,card *,INPtables *)	private
INPerrCat	WinINP::INPerrCat(char *,char *)	public
INPerror	WinINP::INPerror(int)	public
INPevaluate	WinINP::INPevaluate(char **, int *,int)	private
INPfindLev	WinINP::INPfindLev(char *, int *)	private
INPgetMod	WinINP::INPgetMod(GENERIC *,char *,INPmodel **,INPtables *)	private
INPgetTok	WinINP::INPgetTok(char **,char **)	public
INPgetValue	WinINP::INPgetValue(GENERIC *,char **,int ,INPtables *)	public
INPkillMods	WinINP::INPkillMods()	public
INPlookMod	WinINP::INPlookMod(char *)	private
INPmakeMod	WinINP::INPmakeMod(char *, int , card *)	private
INPmkTemp	WinINP::INPmkTemp(char *)	public
INPpName	WinINP::INPpName(char *,Ifvalue *,GENERIC *,int ,GENERIC *)	private
INPparseTree	WinINP::INPparseTree(char **,INPparseTree **,GENERIC *,INPtables *)	private
mkcon	WinINP::mkcon(double)	private
mkb	WinINP::mkb(int ,INPparseNode *,INPparseNode *)	private
mkf	WinINP::mkf(int ,INPparseNode *)	private
PTcheck	WinINP::PTcheck(INPparseNode *)	private
PTparse	WinINP::PTparse(char **)	private
makepnode	WinINP::makepnode(PTelement *)	private
mkbnode	WinINP::mkbnode(int, INPparseNode *,INPparseNode *)	private
mkfnode	WinINP::mkfnode(char *,INPparseNode *)	private
mknnode	WinINP::mknnode(double)	private
mksnode	WinINP::mksnode(char *)	private
PTlexer	WinINP::PTlexer(char **)	private
INPptPrint	WinINP::INPptPrint(char *,IfparseTree *)	private
INPtabInit	WinINP::INPtabInit(int)	public
INPtermInsert	WinINP::INPtermInsert(GENERIC *,char **, INPtables *,GENERIC **)	private

INPmkTerm	WinINP::INPmkTerm(GENERIC *,char **,INPtables *, GENERIC **)	public
INPgndInsert	WinINP::INPgndInsert(GENERIC *,char **, INPtables *,GENERIC **)	private
INPinsert	WinINP::INPinsert(char **, INPtables *)	public
INPtabEnd	WinINP::INPtabEnd(INPtables *)	public
hash	WinINP::hash(char *, int)	private
INPtypelook	WinINP::INPtypelook(char *)	private

Appendix VI

The programming list of the modified *ASRCload()* routine

```
//asrcload.cpp
/*
 * singh@ic.Berkeley.edu
 */

#include "spice.h"
#include <stdio.h>
#include "cktdefs.h"
#include "asrcdefs.h"
#include "sperror.h"
#include "util.h"
#include "suffix.h"

#include "Asrcitf.h"

double *asrc_vals, *asrc_derivs;
int     asrc_nvals;

/*ARGSUSED*/
int
ASRCload(GENmodel *, CKTcircuit *)
GENmodel *inModel;
CKTcircuit *ckt;
{

    /* actually load the current voltage value into the
     * sparse matrix previously provided
     */
#define NONODE 9999
        int it,fun_maxit;
        double x,xb,ep,il,ep_per;
        ep_per=0.1;
        it=0;
        fun_maxit=ckt->CKTdcMaxIter;
    static char *msg = "Too many iterations without convergence";
        static char *msg1 = "No node for fzero function to iterate";

    register ASRCmodel *model = (ASRCmodel*)inModel;
    register ASRCinstance *here;
    int i, v_first, j, branch,fun_ind;
    int node_num,fun_node_num;
    int size,error;
    double rhs,fun_rhs,prev_rhs;
    double prev;
    double diff;
    double tol;
        fun_ind=NONODE;
        double xf, *asrc_derivs_prev;

    /*  loop through all the Arbitrary source models */
    for( ; model != NULL; model = model->ASRCnextModel ) {

        /* loop through all the instances of the model */
        for (here = model->ASRCinstances; here != NULL ;
                here=here->ASRCnextInstance)
        {

            /*
             * Get the function and its derivatives evaluated
             */
            v_first = 1;
            i = here->ASRCtree->numVars;
            if (asrc_nvals < i) {
                    if (asrc_nvals) {
                            FREE(asrc_vals);
                            FREE(asrc_derivs);
                    }
                    asrc_nvals = i;
                    asrc_vals = NEWN(double, i);
```

```
                asrc_derivs = NEWN(double, i);
            }

            j=0;

            /*
             * Fill the vector of values from the previous solution
             */
            for( i=0; i < here->ASRCtree->numVars; i++){
                if( here->ASRCtree->varTypes[i] == IF_INSTANCE){
                 branch = CKTfndBranch(ckt,
                    here->ASRCtree->vars[i].uValue);
                 asrc_vals[i] = *(ckt->CKTrhsOld+branch);
                } else {
                if (here->ASRCtype == ASRC_FZERO) {//add code for fzero as follows//
                        fun_node_num = ((CKTnode *)(here->ASRCtree->vars[i].nValue))-
>number;
                        if (((CKTnode *)(here->ASRCtree->vars[i].nValue))->number ==
here->ASRCposNode) {
                                fun_ind = i; //find the node number for fzero
                        }
                                asrc_vals[i] = *(ckt->CKTrhsOld+fun_node_num);
                }else {//end of the code for fzero//
                        node_num = ((CKTnode *)(here->ASRCtree->vars[i].nValue))-
>number;
                        asrc_vals[i] = *(ckt->CKTrhsOld+node_num);}
                }
            }

/*          if ((*(here->ASRCtree->IFeval))(here->ASRCtree,ckt->CKTgmin, &rhs,
                asrc_vals,asrc_derivs) == OK)
            {*/// comment out the original statement
                if( here->ASRCtype == ASRC_VOLTAGE || here->ASRCtype ==
ASRC_CURRENT){//add code for fzero as follows//
            error=here->ASRCtree->IFeval(here->ASRCtree,ckt->CKTgmin, &rhs,
                        asrc_vals,asrc_derivs);
                }else if (here->ASRCtype == ASRC_FZERO){
            if(fun_ind == NONODE) {
                /*printf("No node for fzero function to iterate\n",fun_ind);*/
                errMsg = MALLOC(strlen(msg1)+1);
                strcpy(errMsg,msg1);
                return(E_NOTERM);
            }
               xb = asrc_vals[fun_ind];
               for(;;){
                        it=++it;
            if(it > fun_maxit) {
                /*printf("too many iterations without convergence: %d iter's\n",
                         iterno);*/
                //ckt->CKTstat->STATnumIter += it;
                errMsg = MALLOC(strlen(msg)+1);
                strcpy(errMsg,msg);
                return(E_ITERLIM);
            }

               error=here->ASRCtree->IFeval(here->ASRCtree,ckt->CKTgmin, &fun_rhs,
                                asrc_vals,asrc_derivs);

               if(xb=asrc_vals[fun_ind]) {asrc_vals[fun_ind] = asrc_vals[fun_ind] -
fun_rhs/asrc_derivs[fun_ind];} //Newton's method
               xf = asrc_vals[fun_ind] - asrc_derivs[fun_ind]*pow((xb-
asrc_vals[fun_ind]),2)/(2*(prev_rhs-fun_rhs-asrc_derivs[fun_ind]*(xb-
asrc_vals[fun_ind])));//Tensor's method
                        if(fabs(xf-asrc_vals[fun_ind]) >= ep && asrc_vals[fun_ind] ==
0) {
                                xb = asrc_vals[fun_ind];
                                error=here->ASRCtree->IFeval(here->ASRCtree,ckt-
>CKTgmin, &prev_rhs,
                                        asrc_vals,asrc_derivs);
                                asrc_vals[fun_ind] = xf;
                                continue;
                //}else if(fabs(asrc_vals[fun_ind]-xb) >= ep) {
                                }else if(fabs((xf-asrc_vals[fun_ind])/asrc_vals[fun_ind]*100)
>= ep_per) {
                                xb = asrc_vals[fun_ind];
```

```
                                error=here->ASRCtree->IFeval(here->ASRCtree,ckt-
>CKTgmin, &prev_rhs,
                                        asrc_vals,asrc_derivs);
                                asrc_vals[fun_ind] = xf;
                                continue;}
                    else{
                    rhs = asrc_vals[fun_ind];
                    break;
                    }
                    }
                    }

                    if (error ==OK){//end of the code for fzero//
                    /* The convergence test */

                    if ( (ckt->CKTmode & MODEINITFIX) ||
                         (ckt->CKTmode & MODEINITFLOAT) )
                    {
#ifndef NEWCONV
                        prev = here->ASRCprev_value;
                        diff = FABS( prev - rhs);
                        if ( here->ASRCtype == ASRC_VOLTAGE||here->ASRCtype ==
ASRC_FZERO){//add ASRC_FZERO for fzero//
                            tol = ckt->CKTreltol *
                                MAX(FABS(rhs),FABS(prev)) + ckt->CKTvoltTol;
                        } else {
                            tol = ckt->CKTreltol *
                                MAX(FABS(rhs),FABS(prev)) + ckt->CKTabstol;
                        }

                        if ( diff > tol) {
                            ckt->CKTnoncon++;
                            ckt->CKTtroubleElt = (GENinstance *) here;
                        }
#endif /* NEWCONV */
                    }
                    here->ASRCprev_value = rhs;

                    /* The ac load precomputation and storage */

                    if (ckt->CKTmode & MODEINITSMSIG) {
                        size = (here->ASRCtree->numVars)+1 ;
                        here->ASRCacValues = NEWN(double, size);
                        for ( i = 0; i < here->ASRCtree->numVars; i++){
                               if ( here->ASRCtype == ASRC_FZERO && i==fun_ind)
                                   here->ASRCacValues[i] = asrc_derivs[i]; //add code for
fzero
                                   else
                                   here->ASRCacValues[i] = asrc_derivs[i];
                        }
                    }

                    for(i=0; i < here->ASRCtree->numVars; i++) {
                        rhs -= (asrc_vals[i] * asrc_derivs[i]);
                        switch(here->ASRCtree->varTypes[i]){
                        case IF_INSTANCE:
                           if( here->ASRCtype == ASRC_VOLTAGE||here->ASRCtype ==
ASRC_FZERO){//add ASRC_FZERO for fzero//
                               /* CCVS */
                               if(v_first){
                                   *(here->ASRCposptr[j++]) += 1.0;
                                   *(here->ASRCposptr[j++]) -= 1.0;
                                   *(here->ASRCposptr[j++]) -= 1.0;
                                   *(here->ASRCposptr[j++]) += 1.0;
                                   v_first = 0;
                               }
                               *(here->ASRCposptr[j++]) -= asrc_derivs[i];
                           } else{
                               /* CCCS */
                               *(here->ASRCposptr[j++]) += asrc_derivs[i];
                               *(here->ASRCposptr[j++]) -= asrc_derivs[i];
                           }
                           break;

                       case IF_NODE:
```

```
                    if(here->ASRCtype == ASRC_VOLTAGE||here->ASRCtype ==
ASRC_FZERO){//add ASRC_FZERO for fzero//
                        /* VCVS */
                        if( v_first){
                            *(here->ASRCposptr[j++]) += 1.0;
                            *(here->ASRCposptr[j++]) -= 1.0;
                            *(here->ASRCposptr[j++]) -= 1.0;
                            *(here->ASRCposptr[j++]) += 1.0;
                            v_first = 0;
                        }
                        *(here->ASRCposptr[j++]) -= asrc_derivs[i];
                    } else {
                        /*VCCS*/
                        *(here->ASRCposptr[j++]) += asrc_derivs[i];
                        *(here->ASRCposptr[j++]) -= asrc_derivs[i];
                    }
                    break;

                default:
                    return(E_BADPARM);
                }
            }

            /* Insert the RHS */
            if( here->ASRCtype == ASRC_VOLTAGE ||here->ASRCtype ==
ASRC_FZERO){//add ASRC_FZERO for fzero//
                *(ckt->CKTrhs+(here->ASRCbranch)) += rhs;
            } else {
                *(ckt->CKTrhs+(here->ASRCposNode)) -= rhs;
                *(ckt->CKTrhs+(here->ASRCnegNode)) += rhs;
            }

            /* Store the rhs for small signal analysis */
            if (ckt->CKTmode & MODEINITSMSIG){

                if ( here->ASRCtype == ASRC_FZERO)
                here->ASRCacValues[here->ASRCtree->numVars] = rhs;//add code
for fzero//
                else
                here->ASRCacValues[here->ASRCtree->numVars] = rhs;
            }
        } else{
            return(E_BADPARM);
        }
    }
  }
  return(OK);
}
```

Appendix VII

Additional example for using the new nonlinear dependent source

The generalized state-space model [17,28] for another type of DC-DC converter – PWM boost converters [27,29] with feedback controls as shown in Figure A7-1 is used to illustrate the usefulness of new nonlinear source. The averaged state-space equations generated by [17,28] is given by

$$\begin{aligned}\frac{d\langle i\rangle_k}{dt} &= -jk\omega_S\langle i\rangle_k + \frac{1}{L}(-\langle v\rangle_k + \langle qv\rangle_k)\\ \frac{d\langle v\rangle_k}{dt} &= -jk\omega_S\langle v\rangle_k + \frac{1}{C}(\langle i\rangle_k - \langle qi\rangle_k - \frac{\langle v\rangle_k}{R})\end{aligned} \qquad \text{(A7-1)}$$

where $\langle\bullet\rangle_k$ denotes the kth coefficient (index-k) of the Fourier series. ω_S is a switching frequency. q=1 when the switch is on (close) and q=0 when it is off (open). The control scheme is given by

$$m_0 + m_1\cos(\omega_S t^* + \theta_1) - \frac{S}{T}t^* = 0 \qquad \text{(A7-2)}$$

where $m_\bullet$ is a modulating signal. $m_\bullet$ and θ_1 are functions of averaged state variables ($\langle\langle\bullet\rangle_\bullet\rangle$). S is the peak magnitude of the sawtooth signal.

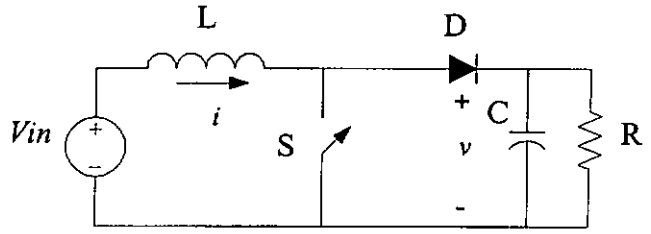


Figure A7-1, Boost converter.

In [17,28], t^* is to be solved at which the sawtooth signal crosses the modulating signal in each switching cycle. Equation (A7-2) is input directly via a circuit netlist file to SPICE3 while [14,17,28] requires indirect and additional input using the special technique [14,17,24]. Both methods can simulate the result with the index-0+index-1 averaged model without difficulty. However, the special technique fails to converge when using the index-0+index-1+index-3 averaged model. The new nonlinear dependent source can solve the average model indifferent to the number of indexes used. The result of capacitor voltage for index-0+index-1+index-3 is shown in Fig. A7-2.

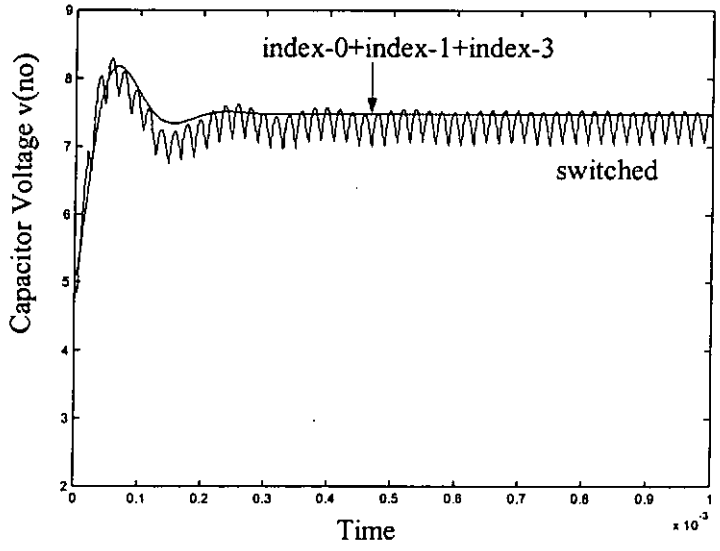


Figure A7-2, Closed-loop simulation for a boost converter with Vin=5V, L=50 μH, C=4.4 μF, R=28 Ω, and fs=50kHz and from transient simulation (switched) of a full model.

The netlist of the averaged model for the boost converter is given below.

```
modelling the boost converter in 3 harmonics

*
vfs nfs 0 50000
rfs nfs 0 1meg
bws nws 0 v=2*pi*v(nfs)
rws nws 0 1meg
*
bd nd 0 fzero={0.13-0.174*(i(vi0)+2*i(vr1)*cos(2*pi*v(nd))-
2*i(vi1)*sin(2*pi*v(nd))+2*i(vr3)*cos(2*3*pi*v(nd))-
```

```
2*i(vi3)*sin(2*3*pi*v(nd)))+0.0435*(v(no)+2*v(nrc1)*cos(2*pi*v(nd))-
2*v(nic1)*sin(2*pi*v(nd))+2*v(nrc3)*cos(2*3*pi*v(nd))-2*v(nic3)*sin(2*3*pi*v(nd)))-v(nd))
*
rd nd 0 1meg
bq0 nq0 0 v=v(nd)
rq0 nq0 0 1meg
bqr nqr1 0 v=sin(2*pi*v(nd))/(2*pi)
rqr nqr1 0 1meg
bqi nqi1 0 v=(cos(2*pi*v(nd))-1)/(2*pi)
rqi nqi1 0 1meg
bqr3 nqr3 0 v=sin(2*3*pi*v(nd))/(2*pi*3)
rqr3 nqr3 0 1meg
bqi3 nqi3 0 v=(cos(2*3*pi*v(nd))-1)/(2*pi*3)
rqi3 nqi3 0 1meg
*
.ic v(no)=5
vin nin 0 5
*rmin ni ni0 1u
l1 nin ni0 50u
*ic=0.5
vi0 ni0 ni01 0
bi01 ni01 ni02 v=(1-v(nq0))*v(no)
bi02 ni03 ni02 v=2*(v(nrc1)*v(nqr1)+v(nic1)*v(nqi1))
bi03 0 ni03 v=2*(v(nrc3)*v(nqr3)+v(nic3)*v(nqi3))
*
bc0 0 no i=(1-v(nq0))*i(vi0)
bc1 no 0 i=2*(i(vr1)*v(nqr1)+i(vi1)*v(nqi1))
bc3 no 0 i=2*(i(vr3)*v(nqr3)+i(vi3)*v(nqi3))
c1 no 0 4.4u
boout5 no 0 i=v(no)/v(ro)
*r1 no 0 28
*rm5 no 0 1meg
*
bnli nli 0 v=v(nws)*50e-6*i(vi1)
l2 nli nr1 50u
*rm3 nr1a nr1 1u
vr1 nr1 nr11 0
br11 nr11 nr12 v=(1-v(nq0))*v(nrc1)
br12 0 nr12 v=v(no)*v(nqr1)
*
bnlr 0 nlr v=v(nws)*50e-6*i(vr1)
l3 nlr ni1 50u
*rm4 ni1a ni1 1u
vi1 ni1 ni11 0
bi11 ni11 ni12 v=(1-v(nq0))*v(nic1)
bi12 0 ni12 v=v(no)*v(nqi1)
*
brc0 0 nrc1 i=(1-v(nq0))*i(vr1)
brc1 nrc1 0 i=i(vi0)*v(nqr1)
brc2 0 nrc1 i=v(nws)*4.4e-6*v(nic1)
c2 nrc1 0 4.4u
boout4 nrc1 0 i=v(nrc1)/v(ro)
*r2 nrc1 0 28
*rm2 nrc1 0 1meg
*
bic0 0 nic1 i=(1-v(nq0))*i(vi1)
bic1 nic1 0 i=i(vi0)*v(nqi1)
bic2 nic1 0 i=v(nws)*4.4e-6*v(nrc1)
c3 nic1 0 4.4u
boout3 nic1 0 i=v(nic1)/v(ro)
*r3 nic1 0 28
*rm1 nic1 0 1meg
*
bnl3i nl3i 0 v=3*v(nws)*50e-6*i(vi3)
l2h3 nl3i nr3 50u
*rm3 nr3a nr3 1u
vr3 nr3 nr33 0
br33 nr33 nr32 v=(1-v(nq0))*v(nrc3)
br32 0 nr32 v=v(no)*v(nqr3)
*
bnl3r 0 nl3r v=3*v(nws)*50e-6*i(vr3)
l3h3 nl3r ni3 50u
*rm4 ni3a ni3 1u
vi3 ni3 ni33 0
bi33 ni33 ni32 v=(1-v(nq0))*v(nic3)
bi32 0 ni32 v=v(no)*v(nqi3)
*
brc30 0 nrc3 i=(1-v(nq0))*i(vr3)
brc31 nrc3 0 i=i(vi0)*v(nqr3)
brc32 0 nrc3 i=3*v(nws)*4.4e-6*v(nic3)
c2h3 nrc3 0 4.4u
boout2 nrc3 0 i=v(nrc3)/v(ro)
```

```
*r2h3 nrc3 0 28
*
bic30 0 nic3 i=(1-v(nq0))*i(vi3)
bic31 nic3 0 i=i(vi0)*v(nqi3)
bic32 nic3 0 i=3*v(nws)*4.4e-6*v(nrc3)
c3h3 nic3 0 4.4u
boout nic3 0 i=v(nic3)/v(ro)
*r3h3 nic3 0 28
*
vro ro 0 28
*pwl(0 28 0.8m 28 0.801m 18 2m 18 2.001m 28 2.5m 28)
rro ro 0 1meg
.tran 0.6u 1m 0 0.6u uic
*.ac dec 101 10 100k
*.dc vfs 5k 1000k 100
.end
```

Appendix VIII

Published paper

1. Wong S. C., Lei W. H. and Lee Y. S. "Averaged Piecewise Transform for Resonant Power Converters". *Proc. IEEE 1999 International Conference on Power Electronics and Drive Systems,* Hong Kong, 972-977 August, 1999, pp. 972-977 (1999)

AVERAGED PIECEWISE TRANSFORM FOR RESONANT POWER CONVERTERS

Siu-Chung Wong, Weng-Hong Lei and Yim-Shu Lee

Department of Electronic and Information Engineering
Hong Kong Polytechnic University
Hung Hom, Hong Kong

Abstract - **A new macromodel using the proposed averaged piecewise transform is developed. When compare to the earlier published macromodels based on the conformal mapping (averaged Fourier transform), the new model is applicable to nearly all types of resonant power converter operating above resonance with a much more accurate results.**

I. INTRODUCTION

RESONANT converters have many advantages over PWM type converters, for example, high power density, high efficiency, low EMI, and low device stresses. However, due to the lack of universal design tools, resonant type converters are difficult to design and implement. The state-space averaging technique [1], which has proved to be so fruitful in the analysis and modeling of PWM converters is not applicable to resonant converters [2, 12] as the principal time constants are not long when compared to the switching period. Other design tools, CAP [3], SWEAP [4], and SPICE models [5-7], based on state-space averaging method, are therefore not well suited to resonant type converters. Design and analysis techniques specific to resonant converters have been developed elsewhere: Bhat [8] uses the state-space approach to find the steady-state solution numerically or in closed form. Verghese *et al.* [9] use sampled-data methods to obtain the transient and dynamic responses. Steigerwald [10] uses complex analysis to make a comparison of resonant topologies. Kazimierzuk *et al.* [11] use Fourier series to analyze the resonant converter in the frequency domain. Batarseh *et al.* [12] use state-plane diagrams to obtain the steady-state and small-signal behavior of resonant type converters. Witulski *et al.* [13] use lumped parameter equivalent circuits to obtain the small-signal equivalent circuits. Each of the above techniques have specific and significant restrictions. None of the methods can be easily incorporated into a *single* SPICE model that will perform dc, ac, and transient analyses.

Macromodels based on the conformal mapping [2] has found to be successful for series [14], parallel [15], and series-parallel [16, 17] resonant power converters (the approach used by [17] is a special case of [14-16]). It offers a speed advantage of two to three orders of magnitude over the full circuit simulation without any convergence problem in circuit simulators. The method assumes that the resonant tank state variables can be represented as a series of sinusoidal function with the harmonic frequencies being the multiples of switching frequency. However, it is well known that abrupt functions cannot be represented by Fourier series with a finite number of harmonics. This limits the accuracy of the macromodel at its optimum to retain only the fundamental component as a trade-off for speed.

In this paper, a new averaged piece-wise transform is proposed to implement a macromodel. Instead of using Fourier series to represent the tank state variables, the piece-wise differentiable solution of the tank state variables at each stages of a switching cycle are used in this new transform. Such change induces no assumption on the tank state variables but the actual solution of the variables at their natural frequency. The job of the piece-wise transform is to extract the magnitudes and phases of each tank state variables within a repeatable cycle to be implemented in the macromodel.

II. THEORY

Suppose that a state variable $x(t)$ can be represented by a piece-wise differentiable sinusoidal function with natural frequency ω_D within a switching period of $2\pi/\omega_S$.

$$x(t) = X_i \sin\left[\omega_D (t - \tau_i)\right] \qquad t \in \left(t_i, t_{i+1}\right] \qquad (1)$$

where $i = 0 \cdots 2n$ is the i-stage of the converter, $t_0 = 0$, $t_n = \dfrac{\pi}{\omega_S}$ and $t_{2n} = \dfrac{2\pi}{\omega_S}$.

It is noted that $x(t)$ has period $2\pi/\omega_S$, but X_i and τ_i have period $2\pi/\omega_S$. Equation (1) can be transformed to a (almost) time-invariant state space equation by using the Piece-wise transform

$$x(t) = \begin{cases} \displaystyle\sum_{k=-\infty}^{\infty} \langle x \rangle_k (t) \exp(jk\omega_S t) & t \in \left(t_0, t_n\right] \\ \displaystyle\sum_{k=-\infty}^{\infty} \langle x \rangle_k \left(t - \frac{\pi}{\omega_S}\right) \exp\left[jk\omega_S \left(t - \frac{\pi}{\omega_S} + \frac{\pi}{\omega_D}\right)\right] & t \in \left(t_n, t_{2n}\right] \end{cases} \qquad (2)$$

$\langle x \rangle_k (t)$ has period π/ω_S and is determined by

$$\langle x \rangle_k (t) = \frac{\omega_S}{\pi} \int_{t - \frac{\pi}{\omega_S}}^{t} x(\tau) \exp(-jk\omega_D \tau) d\tau \qquad (3)$$

Similarly, we can write

$$\left\langle \frac{dx}{dt} \right\rangle_k (t) = \frac{\omega_S}{\pi} \int_{t}^{t+\frac{\pi}{\omega_S}} \frac{dx(\tau)}{d\tau} \exp(-jk\omega_O \tau) d\tau$$

$$= \frac{d}{dt}\langle x \rangle_k + jk\omega_O \langle x \rangle_k + \frac{\omega_S}{\pi}\left(x(t_n+)\exp(-j\pi) - x(t_n-)\exp(-jk\omega_O t_n)\right) + \sum_{i=1}^{n-1}\left[x(t_i+) - x(t_i-)\right]\exp(-jk\omega_O t_i)\Big) \tag{4}$$

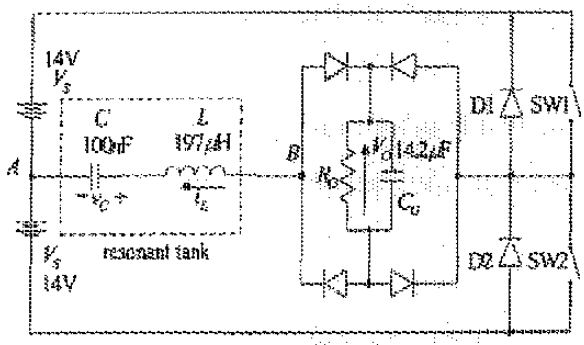


Fig. 1. The power stage of a series resonant converter.

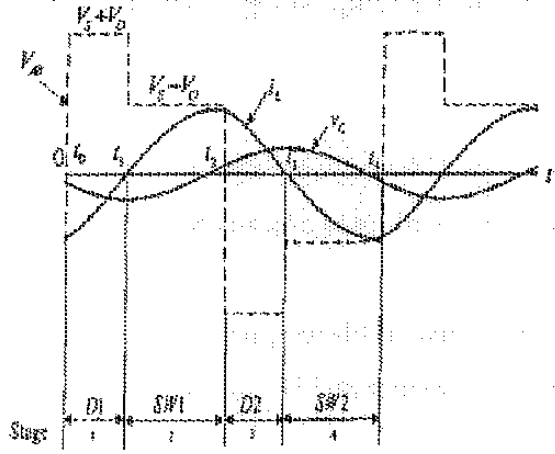


Fig. 2. The above resonance waveforms of the tank inductor current (i_L) and the tank capacitor voltage (v_C).

III. Implementation

As an example, the theory is applied to a half-bridge series resonant converter as shown in Fig. 1. The circuit parameters are given in the figure and diodes are assumed to be ideal. Throughout the analysis, the switching frequency is considered to be higher than the resonant frequency. A state-space model for this circuit takes the form

$$\begin{pmatrix} \frac{di_L}{dt} \\ \frac{dv_C}{dt} \end{pmatrix} = \begin{pmatrix} -\frac{1}{L} & 0 \\ 0 & \frac{1}{C} \end{pmatrix} \begin{pmatrix} v_C \\ i_L \end{pmatrix} + \begin{pmatrix} \frac{-V_O \operatorname{sgn}(i_L) + V_S \operatorname{sgn}(\sin(\omega_S t))}{L} \\ 0 \end{pmatrix} \tag{5}$$

where sgn(•) is the sign functions

A. Solution of the state-space equation

The typical waveforms of the tank inductor current (i_L), the tank capacitor voltage (v_C) and total voltage V_{AB} over the resonant tank are shown in Fig. 2. Suppose *SW1* and *SW2* open and close alternately in complementary fashion at a frequency equal to ω_S. Using the assumption in equation (1), the solutions of i_L and v_C at the first two stages in a period are written in a piecewise sinusoidal function with angular resonance frequency ω_O. Thus,

At stage 1 ($0 < t < t_1$)

$$i_{L1}(t) = X_1 \sin(\omega_O(t - \tau_1)) \tag{6a}$$

$$v_{C1}(t) = -X_1 Z_O \cos(\omega_O(t - \tau_1)) \tag{6b}$$

At stage 2 ($t_1 < t < t_2$)

$$i_{L2}(t) = \frac{(2V_O - X_1 Z_O)}{Z_O} \sin(\omega_O t) \tag{7a}$$

$$v_{C2}(t) = (2V_O - X_1 Z_O)\cos(\omega_O t) \tag{7b}$$

where

$$\text{The Characteristic impedance} = Z_O = \sqrt{\frac{L}{C}}\ \Omega$$

$$\text{Angular resonance frequency} = \omega_O = \frac{1}{\sqrt{LC}}$$

The amplitude and phase of $i_{L1}(t)$ in equation (6a) is represented by the variable X_1 and τ_1. Base on equation (6a), the amplitudes of equation (6b) and (7) can be written in terms of X_1, V_O, Z_O by substituting the initial values at each stage.

B. Derivation of averaged time-invariant equations

We shall use the new averaged piece-wise transform to implement a macromodel of the series-resonant converter in this and the following sections. The fist step in the modelling process is to derive the averaged time-invariant equations of tank states variables i_L and v_C. We can apply the transform equation (4) to equation (6) and (7). Note that equation (4) is a complex equation, there are two corresponding equations in real variables (one for real part and the other for imaginary part). Consistent with Fig. 2, we have $t_0 = 0, t_1 = \tau_1$ and $t_2 = \pi/\omega_S$. Calculating the end points of each piece sinusoidal function for equation (6) and equation (7), and then, putting

$$i_L(t_1-) = i_{L1}(\tau_1), i_L(t_1+) = i_{L2}(0),$$

$$i_L(t_2-) = i_{L2}(\frac{\pi}{\omega_S} - \tau_1), i_L(t_2+) = i_{L2}(\frac{\pi}{\omega_S} - \tau_1),$$

$$v_C(t_1-) = v_{C1}(\tau_1), v_C(t_1+) = v_{C2}(0),$$

$$v_C(t_2-) = v_{C2}(\frac{\pi}{\omega_S} - \tau_1)$$

and

$$v_C(t_2+) = v_{C2}(\frac{\pi}{\omega_S} - \tau_1) + 2V_S$$

in equation (4), we can obtain four simplified time-invariant equations of the tank state variables

$$\left\langle \frac{di_L}{dt} \right\rangle_k^r (t) = \frac{d}{dt}\langle i_L \rangle_k^r - \omega_0 \langle i_L \rangle_k^i - \frac{2\omega_S(-2V_O + X_1 Z_O)}{\pi Z_O} \cos^2\left(\frac{\pi\omega_O}{2\omega_S}\right) \sin\left(\omega_O\left(\tau_1 - \frac{\pi}{\omega_S}\right)\right)$$

$$\left\langle \frac{di_L}{dt} \right\rangle_k^i (t) = \frac{d}{dt}\langle i_L \rangle_k^i + \omega_O \langle i_L \rangle_k^r - \frac{\omega_S(2V_O - X_1 Z_O)}{\pi Z_O} \sin\left(\frac{\pi\omega_O}{2\omega_S}\right) \sin\left(\omega_O\left(\tau_1 - \frac{\pi}{\omega_S}\right)\right)$$

$$\left\langle \frac{dv_C}{dt} \right\rangle_k^r (t) = \frac{d}{dt}\langle v_C \rangle_k^r - \omega_O \langle v_C \rangle_k^i + \frac{2\omega_S}{\pi} \begin{Bmatrix} V_S - V_O \cos(\omega_O \tau_1) \\ + (2V_O - X_1 Z_O)\cos^2\left(\frac{\pi\omega_O}{2\omega_S}\right) \\ + (2V_O - X_1 Z_O)\cos\left(\omega_O\left(\tau_1 - \frac{\pi}{\omega_S}\right)\right) \end{Bmatrix}$$

$$\left\langle \frac{dv_C}{dt} \right\rangle_k^i (t) = \frac{d}{dt}\langle v_C \rangle_k^i - \omega_O \langle v_C \rangle_k^r + \frac{\omega_S}{2\pi} \begin{Bmatrix} (2V_O + X_1 Z_O)\sin(\omega_O \tau_1) \\ + (2V_O - X_1 Z_O)\sin\left(\omega_O\left(\tau_1 - \frac{2\pi}{\omega_S}\right)\right) \end{Bmatrix} \quad (8)$$

where $\langle x \rangle_k^r$ denotes the real part of $\langle x \rangle_k$ and $\langle x \rangle_k^i$ denotes the imaginary part of $\langle x \rangle_k$. Both of them are real. And equations (8) is given by putting k=1.

Then, let us derive the average input current ($\overline{i_S}$) and output current ($\overline{|i_O|}$) of the series resonant converter. It can be found by straight forward integration. As the inductor current (i_L) is a piecewise sinusoidal function, the integration should be partitioned. It is shown in equations (9) and (10).

$$\overline{i_S} = \frac{\omega_S}{\pi}\int_0^{\frac{\pi}{\omega_S}} i_L(t)dt = \frac{\omega_S}{\pi}\left(\int_0^{\tau_1} i_{L1}(t)dt - \int_0^{\frac{\pi}{\omega_S}-\tau_1} i_{L2}(t)dt\right) = \frac{\omega_S}{\pi\omega_O Z_O}\begin{Bmatrix} -2V_O + 2X_1 Z_O - X_1 Z_O \cos(\omega_O \tau_1) \\ + (2V_O - X_1 Z_O)\cos\left(\omega_O\left(\tau_1 - \frac{\pi}{\omega_S}\right)\right) \end{Bmatrix} \quad (9)$$

$$\overline{|i_O|} = \frac{\omega_S}{\pi}\int_0^{\frac{\pi}{\omega_S}} |i_L(t)|dt = \frac{\omega_S}{\pi}\left(\int_0^{\tau_1} i_{L1}(t)dt + \int_0^{\frac{\pi}{\omega_S}-\tau_1} i_{L2}(t)dt\right) = \frac{\omega_S}{\pi\omega_O Z_O}\begin{Bmatrix} -2V_O + X_1 \cos(\omega_O \tau_1) \\ + (2V_O - X_1 Z_O)\cos\left(\omega_O\left(\tau_1 - \frac{\pi}{\omega_S}\right)\right) \end{Bmatrix} \quad (10)$$

Before we proceed to usedel the converter, we need two additional equations to complete our transform. Equations (8), (9) and (10) all contain two unknown variables X_1 and τ_1. They can be solved by applying equation (3) to equations (6) and (7). we also have four equations

$$\begin{aligned} \langle i_L \rangle_k^r &= \frac{1}{4\pi\omega_O Z_O}(C + D) \\ \langle i_L \rangle_k^i &= \frac{1}{4\pi\omega_O Z_O}(A + B) \\ \langle v_C \rangle_k^r &= \frac{1}{4\pi\omega_O}(A - B) \\ \langle v_C \rangle_k^i &= \frac{1}{4\pi\omega_O}(C - D) \end{aligned} \quad (11)$$

where

$$C = (-2V_O\omega_S + X_1\omega_S Z_O)\cos(\omega_O\tau_1) + \omega_S(2V_O - X_1Z_O)\cos\left(\omega_O\tau_1 - \frac{2\pi\omega_O}{\omega_S}\right)$$

$$D = 2\omega_O(2\pi V_O - 2\tau_1 V_O\omega_S - \pi X_1 Z_O)\sin(\omega_O\tau_1)$$

$$A = 2\omega_O(2\pi V_O - 2\tau_1 V_O\omega_S - \pi X_1 Z_O)\cos(\omega_O\tau_1)$$

$$B = (2V_O\omega_S + X_1\omega_S Z_O)\sin(\omega_O\tau_1) + (2V_O - X_1Z_O)\sin\left(\omega_O\tau_1 - \frac{2\pi\omega_O}{\omega_S}\right)$$

Solving equation (11), the magnitude X_1 and the phase τ_1 are given by

$$X_1 = \frac{1}{Z_O}\left(\sqrt{Q^2 + P^2} + 2V_O\left(1 + \frac{I_S\omega_S}{\pi\omega_O}\right)\right) \quad (12)$$

$$\tau_1 = \tan^{-1}\left(\frac{Q}{P}\right) \quad (13)$$

where $Q = Z_O\langle i_L\rangle' - \langle v_C\rangle'$, $P = Z_O\langle i_L\rangle' + \langle v_C\rangle'$

C. Macromodel

Now the collection of equations (8), (12), (13), the average input equation (9) and the average output equation (10) gives a complete model for the half-bridge series resonant converter. The macromodel for the converter is shown in Fig. 3. Extracting the resonant tank (shown in Fig. 1) from the converter, and connecting total dependent voltage source ($V^{Re}_{(total)}$) to the tank inductor in series and total dependent current source ($I^{Re}_{(total)}$) to the tank capacitor in parallel models the real part circuit. Similarly, connecting $V^{Im}_{(total)}$ and $I^{Im}_{(total)}$ to the resonant tank models the imaginary part circuit, where dependent voltage sources and dependent current sources are obtained by rewriting equation (8) with L and C. The input and output part of the converter are modeled by equation (9) and (10). The voltage sources V_{i1} and V_{i2} are used as the current sensors in SPICE simulations. The only independent source in Fig. 3 is V_S. The others are dependent sources whose control equations are given by

$$V^{Re}_{(total)} = \omega_O L\langle i_L\rangle' - \frac{2\omega_S L(2V_O - X_1Z_O)}{\pi Z_O}\cos^2\left(\frac{\pi\omega_O}{2\omega_S}\right)\sin\left(\omega_O\left(\tau_1 - \frac{\pi}{\omega_S}\right)\right)$$

$$I^{Re}_{(total)} = \omega_O C\langle v_C\rangle' - \frac{2\omega_S C}{\pi}\left\{\begin{array}{l} V_S - V_O\cos(\omega_O\tau_1) \\ +(2V_O - X_1Z_O)\cos^2\left(\frac{\pi\omega_O}{2\omega_S}\right) \\ +(2V_O - X_1Z_O)\cos\left(\omega_O\left(\tau_1 - \frac{\pi}{\omega_S}\right)\right)\end{array}\right\} \quad (14)$$

$$V^{Im}_{(total)} = -\omega_O L\langle i_L\rangle' + \frac{\omega_S L(2V_O - X_1Z_O)}{\pi Z_O}\sin\left(\frac{\pi\omega_O}{2\omega_S}\right)\sin\left(\omega_O\left(\tau_1 - \frac{\pi}{\omega_S}\right)\right)$$

$$I^{Im}_{(total)} = -\omega_O C\langle v_C\rangle' - \frac{\omega_S C}{2\pi}\left\{\begin{array}{l}(2V_O + X_1Z_O)\sin(\omega_O\tau_1) \\ +(2V_O - X_1Z_O)\sin\left(\omega_O\left(\tau_1 - \frac{2\pi}{\omega_S}\right)\right)\end{array}\right\} \quad (15)$$

$$\overline{i_S} = \frac{\omega_S}{\pi\omega_O Z_O}\left\{\begin{array}{l}-2V_O + 2X_1Z_O - X_1Z_O\cos(\omega_O\tau_1) \\ +(2V_O - X_1Z_O)\cos\left(\omega_O\left(\tau_1 - \frac{\pi}{\omega_S}\right)\right)\end{array}\right\} \quad (16)$$

$$\overline{|i_o|} = \frac{\omega_S}{\pi\omega_O Z_O}\left\{\begin{array}{l}-2V_O + X_1\cos(\omega_O\tau_1) \\ +(2V_O - X_1Z_O)\cos\left(\omega_O\left(\tau_1 - \frac{\pi}{\omega_S}\right)\right)\end{array}\right\} \quad (17)$$

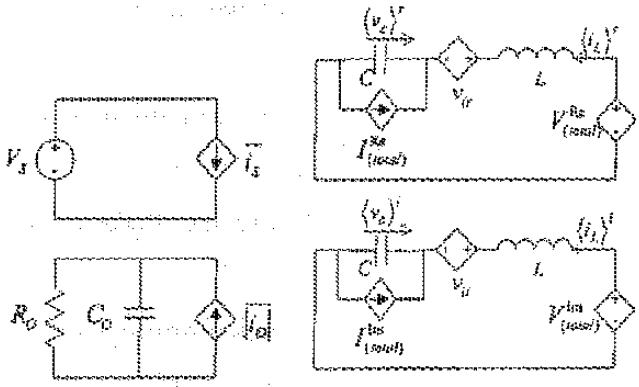


Fig. 3. The complete averaged piecewise macromodel of the series resonant converter.

IV. Results

We have implemented the complete macromodel of the series resonant converter in SPICE3 (the technique applies equally well to other resonant converters like parallel, series-parallel and phase controlled resonant converters). The component values are taken from established designs [2, 14] as shown in Fig. 1. All the diodes used are idealized by setting n=1u in the SPICE diode model. The results of the steady-state output voltage as a function of switching frequency above resonance for different R_O / Z_O ratios (0.1, 0.5, 1.0 and 2.0) are shown in Fig. 4. It is shown that the new macromodel gives accurate results even for $R_O = 2Z_O$ where the old macromodel deviated from the full transient simulations. To demonstrate the accuracy of the new macromodel, transient and ac small signal analyses are also performed for $R_O = 2Z_O$. Fig. 5 shows the responses when a drive frequency abruptly changes between 50kHz and 40kHz. Fig. 6 shows the open loop Bode response of the series resonant converter at a switching frequency of 38kHz.

V. Conclusion

An averaged piece-wise transform has been developed to model the resonant converter. The new transformation has a better accuracy over the averaged Fourier transform (conformal mapping) when implemented as a macromodel in SPICE.

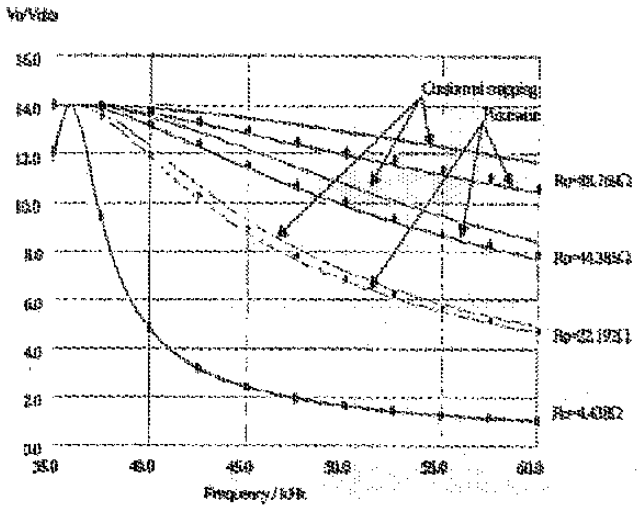


Fig. 4. Comparison of simulation result for the series resonant converter with R_O/Z_O = 0.1, 0.5, 1.0 and 2.0 (Z_O =44.3847Ω). The results from DC analysis (full lines) of the conformal mapping macromodel and the piecewise macromodel and from steady state transient simulations (polygons) of a full model

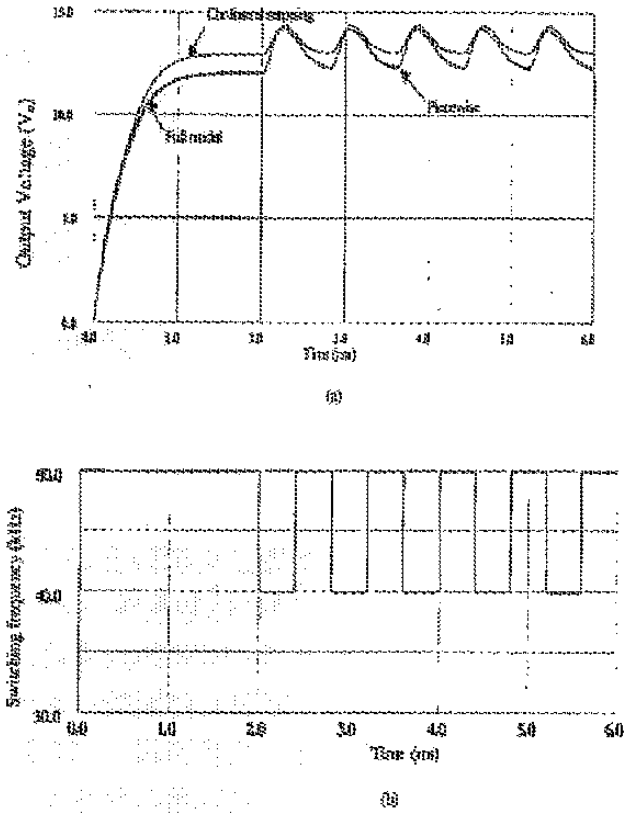


Fig. 5. Comparison of results from transient analysis of the conformal mapping macromodel and the piecewise macromodel and from transient simulation of a full model. A step change in switching frequency is applied.

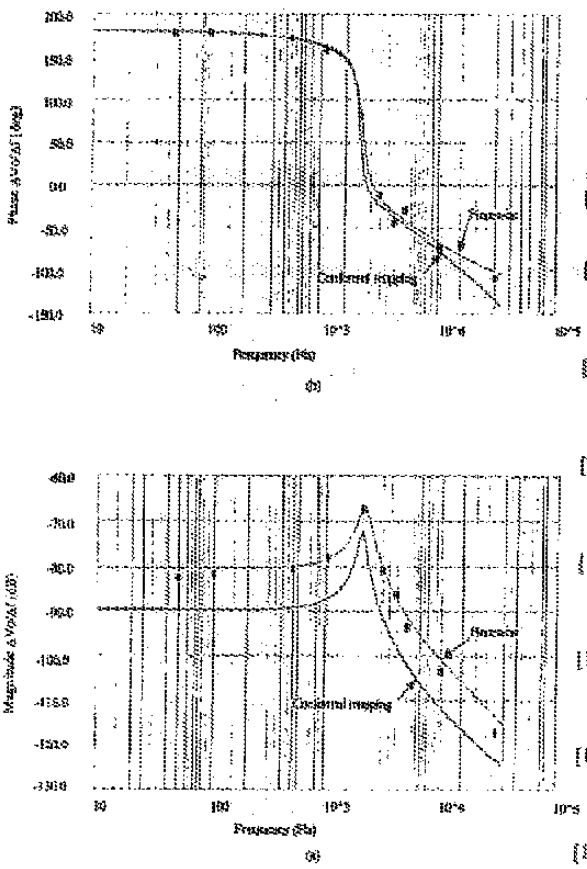


Fig. 6. Comparison of results form AC small signal analysis (full lines) of the conformal mapping and the piecewise macromodel and from transient simulations (squares) of a full model.

VI. REFERENCES

[1] Middlebrook R. D. and Cuk S., *Advances in Switching-Mode Power Conversion*, vol. 1-3. TESLAco, 1983.

[2] Sender S. R., Noworolski J. M., Liu X. Z. and Verghese G. C., "Generalized averaging method for power conversion circuits," *IEEE Transactions on Power Electronics*, vol. 6, No. 2, April 1991, pp. 251-159

[3] Liu C. C., Hsieh J., Chang C. H. K., Bocek J. M., and Hsiao U. T., "A fast-decoupled method for time-domain simulation of power converter," *IEEE Transactions on Power Electronics*, vol. 8, No. 1, January 1993, pp. 37-45

[4] Dirkman R. J., "The simulation of general circuits containing ideal switches," in ," *IEEE Power Electron. Spec. Conf. Rec.*, 1987, pp. 185-194.

[5] Lee Y. S., Wong S. C. and Lai Y. M., "SPICE modeling and simulation of power converters," in *Proc. China 9th Conf. Circuit. Syst.*, China, 1990, pp. 139-145.

[6] Lee Y. S., Cheng D. and Wong S. C., "A new approach to the modeling of converters for SPICE simulation," ," *IEEE Transactions on Power Electronics*, vol. 7, No. 4, October 1992, pp. 741-753

[7] Wong S. C. and Lee Y. S., "Modeling, analysis and SPICE simulation of hysteretic current-controlled converters," *IEEE Transactions on Power Electronics*, vol. 8, No. 4, October 1993, pp. 580-587

[8] Bhat, A. K. S., "Analysis and design of a series-parallel resonant converter," *IEEE Transactions on Power Electronics*, vol. 8, No. 1, January 1993, pp. 1-11

[9] Verghese G. C., Elbuluk M. E., and Kassakian J. G., "A general approach to sampled-data modeling for power electronic circuits," *IEEE Transactions on Power Electronics*, vol. 1, No. 2, April 1986, pp. 76-89

[10] Steigerwald R. L., "A comparison of half-bridge resonant converter topologies," *IEEE Transactions on Power Electronics*, vol. 3, No. 2, April 1988, pp. 174-182

[11] Kazimierzuk M. K., Thirusarayan N. and Wang S., "Analysis of series-parallel resonant converter," *IEEE Transactions on Aerosp. Electron. Syst.*, vol. 29, No. 1, January 1993, pp. 88-99

[12] Batarseh I. and Siri K., "Generalized approach to the small signal modeling of dc-to-dc resonant converters," *IEEE Transactions on Aerosp. Electron. Syst.*, vol. 29, No. 3, July 1993, pp. 894-909

[13] Witulski A. F., Hernandez A. F. and Erickson R. W., "Small signal equivalent circuit modeling resonant converters," *IEEE Transactions on Power Electronics*, vol. 6, No. 1, January 1991, pp. 11-27

[14] Wong S. C. and Brown A. D., "Macromodelling the series resonant converter circuits," *IEE Proceedings: Circuits, Devices and Systems*, Vol. 142, No. 1, February 1995, pp. 83-89

[15] Wong S. C. and Brown A. D., "Parallel resonant converter as a circuit simulation primitive," *IEE Proceedings: Circuits, Devices and Systems*, Vol. 142, No. 6, December 1995, pp. 379-386

[16] Wong S. C. and Brown A. D., "Analysis, modeling and simulation of series-parallel resonant converter circuits," *IEEE Transactions on Power Electronics*, vol. 10, No. 5, September 1995, pp. 605-614

[17] S. Ben-Yaakov and G. Rahav, "Average modelling and simulation of series-parallel resonant converters by PSPICE compatible behavioural dependent sources," *Electronics Letters*, vol. 32 No. 4, 15th February 1996, pp. 288-290

References

[1] Quarles, T. L. *The SPICE3 implementation guide.* UC Berkeley, USA, 114pp (1989)

[2] Verghese G. C., Elbuluk M. E. and Kassakian J. G. "A general approach to sampled-data modeling for the power conversion circuits". *IEEE Transactions on Power Electronics,* Vol. PE-1, No. 2, pp.76-89 (1989)

[3] Bhat A. K. S. "Analysis and design of series-parallel resonant power supply". *IEEE Transactions on Aerospace and Electronic Systems,* Vol. 28, No. 1, pp.249-258 (1992)

[4] Bhat A. K. S. "A generalized approach for the steady-state analysis of resonant inverters". *IEEE Transactions on Industry Applications,* Vol. 25, No. 2, pp.326-338 (1989)

[5] Wong S.C. "A Macromodel for resonant converter circuits", *Ph.D. dissertation,* Dep. Elec. Comput. Sci. University of Southamption, Southamption (1997)

[6] Wong S.C. and Brown A. D. "Macromodelling the series resonant converter circuits". *IEE Proceedings: Circuits, Devices and Systems,* Vol.142, No. 1, pp.83-89 (1995)

[7] Wong S.C. and Brown A. D. "Parallel resonant converter as circuit simulation primitive". *IEE Proceedings: Circuits, Devices and Systems,* Vol.142, No. 6, pp. 379-386 (1995)

[8] Wong S.C. and Brown A. D. "Analysis, modeling and simulation of series-parallel resonant converter circuits". *IEEE Transactions on Power Electronics,* Vol. 10, No. 5, pp.605-614 (1995)

[9] Bhat A. K. S. "Analysis and design of a series-parallel resonant converter with capacitive output filter". *IEEE Transactions on Industry Applications,* Vol. 27, No. 3, pp. 523-530 (1991)

[10] Nelms R. M. "Harmonic analysis of a parallel-loaded resonant converter". *IEEE Transactions on Aerospace and Electronic Systems,* Vol. 27, No. 4, pp.683-688 (1991)

[11] Steigerwald Robert L. "A comparison of half-bridge resonant converter topologies". *IEEE Transactions on Power Electronics,* Vol. 3, No. 2, pp.174-182 (1988)

[12] Batarseh I., Liu R., Lee C. Q. and Upadhyay A. K., "Theoretical and experimental studies of the LCC-type parallel resonant converter". *IEEE Transactions on Power Electronics,* Vol. 5, No. 2, pp.140-150 (1990)

[13] Wong S. C., Lei W. H. and Lee Y. S. "Averaged Piecewise Transform for Resonant Power Converter". *Proc. IEEE 1999 International Conference on Power Electronics and Drive Systems,* Hong Kong, 972-977 August, 1999, pp. 972-977 (1999)

[14] Lee Y. S., Chow M. H. L. and Wong J. S. L. "SPICE simulation of nonlinear equations and circuits". *IEE Proceedings-G,* Vol. 138, No. 2, pp.273-281 (1991)

[15] Schnabel, R. B. and Frank, P. D. "Tensor methods for nonlinear equations". *SIAM J. Numer. Anal.* Vol. 21, No. 5, pp. 815-843 (1984)

[16] Bouaricha, A. "Algorithm 765: STENMIN: A software package for large, sparse unconstrained optimization using tensor methods". *ACM Transaction on Mathematical Software,* Vol. 23, No. 1, pp. 81-90 (1997)

[17] Vahe A. Caliskan, George C. Verghese and Aleksandar M. Stankovic "Multifrequency averaging of DC/DC converters". *IEEE Transactions on Power Electronics*, Vol. 14, No. 1, pp. 124-133 (1999)

[18] Laurence W. Nagel *SPICE2: A computer program to simulate semiconductor cirucit,* UC Berkeley, USA, 233pp (1975)

[19] Kazimierzuk M. K. and Wang S. "Frequency-domain Analysis of series resonant converter for continuous conduction mode". *IEEE Transactions on Power Electronics,* Vol. 7, No. 2, pp.270-279 (1992)

[20] Sender S. R., Noworolski J. M., Liu X. Z. and Vergheses G. C. "Generalized averaging method for power conversion circuits". *IEEE Transactions on Power Electronics,* Vol. 6, No. 2, pp.251-259 (1991)

[21] Bhat A. K. S. "Analysis and design of a series-parallel resonant converter". *IEEE Transactions on Power Electronics,* Vol. 8, No. 1, pp.1-11, (1993)

[22] Raimund K. Ege *Object-oriented programming with C++.* AP professional, USA, 358pp (1994)

[23] Yang E. X., Lee F. C. and Jovanovic M. M. "Extended describing function technique applied to the modelling of resonant converters". *Proc. VPEC 1991 PES,* 179-191, 1991, pp. 179-191 (1991)

[24] Chow M. H. L., Lee Y. S. and Wong J. S. L. "Use of SPICE in analogue computation and simulation of systems consisting of circuit components and transfer functions". *IEE Proceedings-G,* Vol. 138, No. 2, pp.282-288 (1991)

[25] Batarseh I. "Small signal analysis of the LCC-type parallel resonant converter". *IEEE Transactions on Aerospace and Electronic Systems,* Vol. 32, No. 2, pp.702-713 (1996)

[26] Lee Y. S., Tse C. K., So W. C. and Ng S. W. "Modelling the half-bridge series resonant converter using the MISSCO approach". *Proceedings of IEEE Power Electronics Specialists Conference,* 629-635, pp. 629-635 (1995)

[27] Xu J. and Lee C. Q. "Generalized state-space averaging approach for a class of periodically switched networks". *IEEE Transactions on circuits and systems-I: Fundamental Theory and Applications,* Vol. 44, No. 11, pp.1078-1081 (1997)

[28] Vahe A. Caliskan, George C. Verghese and Aleksandar M. Stankovic "Multifrequency averaging of DC/DC converters". *IEEE 5th Workshop on Computers in Power Electronics,* Portland, OR, 113-119 August 1996, pp. 113-119 (1996)

[29] Mahdavi J., Emaadi A., Bellar M. D. and Ehsani M. "Analysis of power electronic converters using the generalized state-space averaging approach". *IEEE Transactions on circuits and systems-I: Fundamental Theory and Applications,* Vol. 44, No. 8, pp.767-770 (1997)

[30] Frank P. D. "Tensor methods for solving systems of nonlinear equations", *Ph.D. dissertation,* Dep. Comput. Sci. University of Colorado, Colorado (1984)

[31] Hernandez A. F. and Erickson R. W. "A large signal computer model for the series resonant converter". *Proceedings of IEEE Power Electronics Specialists Conference,* 737-744, pp. 737-744 (1991)

[32] Batarseh I. and Siri K., "Generalized approach to the small signal modelling of DC-to-DC resonant converters". *IEEE Transactions on Aerospace and Electronic Systems,* Vol. 29, No. 3, pp.894-909 (1993)

[33] ftp sit: ic.berkeley.edu

[34] Richard L. Burden, J. Douglas Faires, *Numerical Analysis.* Brooks/Cole Pub. Co., USA, 326pp (1997)

[35] Malek-Madani, Rezo., *Advanced engineering mathematics with mathematica and matlab.* Addison Wesley Longman, Inc., USA, 521pp (1997)