IMPLEMENTATION OF ALGORITHMS FOR THE PERIODIC-STEADY-STATE ANALYSIS OF NONLINEAR CIRCUITS

by

Pranav N. Ashar

Memorandum No. UCB/ERL M89/31

15 March 1989

COVER PAGE

67

IMPLEMENTATION OF ALGORITHMS FOR THE PERIODIC-STEADY-STATE ANALYSIS OF NONLINEAR CIRCUITS

by

Pranav N. Ashar

Memorandum No. UCB/ERL M89/31

15 March 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

The periodic steady-state of a circuit is important because the true characteristics of many circuits can only be obtained when the circuit is in the periodic steady-state. One way of determining the steady-state response is to carry out a transient analysis until the start-up transients in the circuit die out. Such an approach can prove to be computationally expensive. A number of algorithms for the direct computation of the periodic steady-state are available. In the present work, a class of algorithms that operate in the *time domain* using *shooting techniques* has been studied. One such technique, based on Newton-Raphson iterations, was proposed in the early seventies [AT72b] and implemented in a program called **SINC-S** [Fan75]. The *FORTRAN* version of **SINC-S** was converted to *C* as a part of the present work [GJ88]. Another algorithm belonging to this class was proposed and implemented in a circuit simulator in the early eighties [Ske80]. **SINC-S** does not have this algorithm implemented in it. Both these algorithms have been reexamined in the present work and implemented into **SPICE3** [QNPS87]. The resulting program, **SSPICE**, has been used to evaluate the performance of the algorithms and to ascertain their relative merits and demerits. The performance of these algorithms has also been compared with that of an algorithm based on harmonic-balance and implemented in **SPECTRE** [KS86].

Acknowledgements

I am grateful to Prof. Donald Pederson for suggesting the project and for providing me with unqualified support during its execution. He has been a constant source of ecouragement. I am also thankful to him for reading the manuscript very carefully. I am responsible for any errors that may still remain in the thesis.

I have been very liberal in taking help from Tom Quarles. His patience while explaining the internals of SPICE3 to me is greatly appreciated. I have had occasion to interact with Ken Kundert. I enjoyed the exchange of ideas with him. Beorn Johnson and Wayne Christopher have also been very helpful.

This work was supported by the state of California MICRO grant, a grant from Microelectronics and Computer Technology Corporation, and a grant from Polaroid Corporation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Periodic-Steady-State Problem

In many circuit simulations the steady-state behavior of the circuit is of primary interest. In particular, the time-dependent circuit variables should be observed only when the start-up transient behavior is no longer present and an established periodic response is present. A typical plot of the transient behaviour of a *high-Q* circuit is shown in Figure 1.1. The figure shows clearly the decay of the start-up transients and the subsequent build up of the steady-state. Measurement of distortion, for example, is inaccurate if transients are present because the start-up transients would be confused with the nonlinear behavior of the circuit. This can happen when the circuit itself is nonlinear or the input signal is non-sinusoidal and the circuit is linear. Power dissipation, distortion generation, noise, gain and transfer characteristics are some of the commonly measured circuit parameters which require that the circuit have reached the steady-state.

Oscillators are a class of circuits for which characterizing the steady-state behavior is especially important, because in the case of oscillators the steady-state is usually the primary region of interest. Generally, only an approximate value of the period is available initially. The actual period can only be determined when the oscillator has reached the steady-state.

Given the above factors, a simulator with the ability to establish the steady-state rapidly and accurately is an important tool for circuit designers.

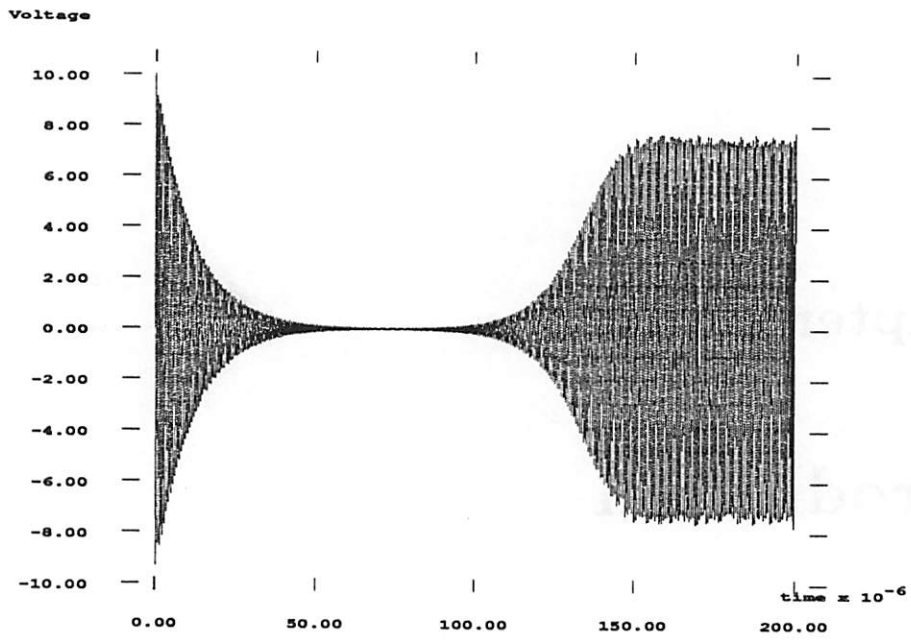


Figure 1.1: Typical Transient Behaviour of a High-Q Colpitts Oscillator

1.2 Possible Solutions

Transient Analysis

One approach for a simulator to determine the steady-state is to carry out a transient analysis until a criterion is satisfied indicating that the steady-state is reached. Such an approach can require a large amount of cpu time. Consider the example of a crystal oscillator with a Q of 100,000. If one uses conventional transient-analysis to reach the periodic steady-state, one would typically require that the circuit be simulated for at least 10 time-constants, which is the equivalent of one million cycles for a circuit with a Q of 100,000. Even after simulating for a long time, the steady-state may not have been reached to the desired accuracy. It is possible to use customized integration methods to reduce the computing time required. For example, in case lightly damped oscillations are present in the transient, the integration method can be made to follow the envelope rather than solution itself [Pet81].

Direct Computation of the Periodic Steady-State

A possibly more efficient approach is to compute the steady-state behavior directly. Figure 1.2 shows a typical steady-state waveform for a Colpitts oscillator computed using

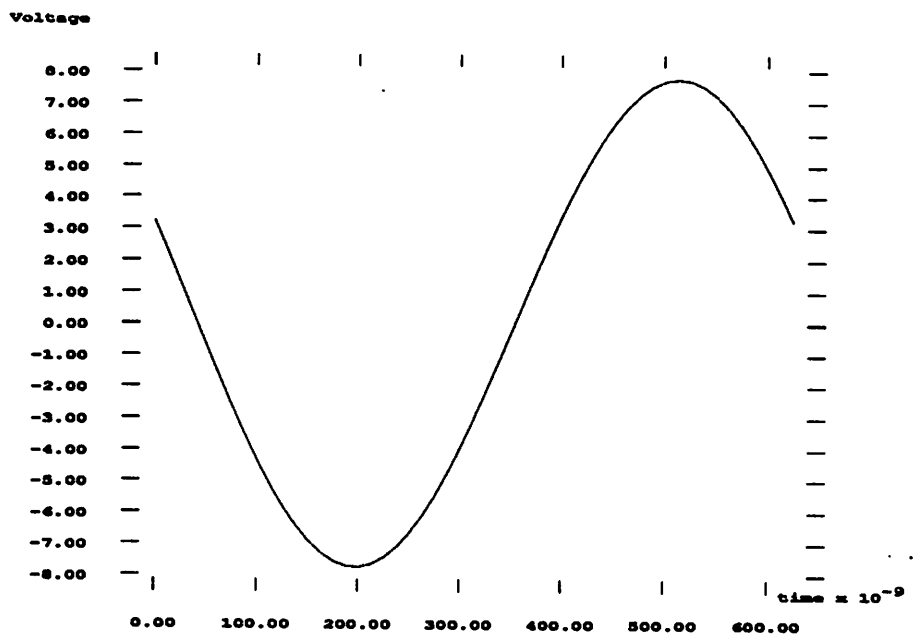


Figure 1.2: Steady-State of a Colpitts Oscillator Computed Using One of the Algorithms

---

such a method. Only 12 cycles of transient analysis were required to establish the steady-state using the steady-state algorithm. The cpu-time required was 6.05 seconds. This cpu-time requirement is less than one fifth of the cpu-time required when conventional transient analysis is used. In addition to accelerating the computation of the steady-state, direct computation gives a more accurate indication of whether the steady-state has been reached. When conventional transient-analysis is used for determining the periodic steady-state, the convergence to the steady-state may be very slow. Thus, the distance of the circuit from the periodic steady-state does not change very much from one cycle to the subsequent cycle. It becomes difficult to decide what the exact point in time should be when one can assume that the steady-state has been reached. When the algorithms for the direct computation of the periodic steady-state are being used, the difference in the distance of the circuit from the periodic steady-state is usually quite large from one cycle to the next. This makes it possible to do just enough computation so that the steady-state is reached to the desired accuracy.

The problem of computing the periodic steady-state can be posed as a 2-point boundary-value problem. The boundary constraints define a known relationship between the initial state and the final state, but neither of the two states is known beforehand. It is necessary to devise a numerical algorithm that finds a solution such that the initial and

final conditions satisfy the given requirements. This problem is much more difficult than an initial-value problem which is solved when a conventional transient-analysis is done. In the initial-value problem the integration starts from a given initial state and proceeds forward in time with no constraints on the final state.

Many approaches to solving the 2-point boundary-value problem have been proposed [CT73,Ske80,KS86,DC76]. In general, it is possible to classify these approaches into three categories:

- Shooting methods [Appendices C and D]
- Finite-difference methods
- Expansion methods

Shooting methods are time-domain methods that attempt to solve the 2-point boundary-value problem by treating it as a sequence of initial-value problems. Each initial-value problem is evaluted for one period. The aim is to obtain an initial condition that eliminates any start-up transient behaviour and results immediately in periodicity. The solution of each initial-value problem is used iteratively to obtain the required initial condition $x(0)$ that satisfies $x(T) - x(0) = 0$. Different methods can be used for iteration. For example, the Newton-Raphson[1] method is used in [AT72b,AT72a,CT73,TCF75,GT82,Fan75] to iterate while the extrapolation method is used in [Ske80]. The present research has focussed on shooting methods. Shooting methods are appropriate for highly nonlinear circuits that none-the-less have a linear state-transition function. Both the Newton-Raphson-based and the extrapolation-based methods have been implemented into the circuit simulator **SPICE3.**

Finite-difference methods [KSS88] solve the 2-point boundary-value problem by replacing the differential equations with finite-difference equations on a mesh of points in time that cover one period. Trial solutions of the resulting system of equations consist of one discrete value for each point on the mesh so that, even though the difference equations may not be satisfied, the boundary conditions are satisfied. The solution at each point is then iterated until the difference equations are also satisfied. The solution that satisfies

[1]The use of the Newton-Raphson method in this context is distinct from the conventional use of the Newton-Raphson method by circuit simulators

both, the boundary constraints and the difference equations, is the desired solution. Finite-Difference methods are difficult to implement. The effort involved in implementing these methods is equivalent to the effort required to implement a full fledged circuit-simulator.

Expansion methods try to express the solution in terms of a finite number of basis functions. The problem then reduces to finding the multiplying coefficient for each of these basis functions so that some error is minimized. Harmonic-balance is a particular expansion method that uses sinusoids as basis functions[KS86]. Expansion methods are well suited to simulating circuits with distributed devices as long as the circuits are only mildly nonlinear.

1.3 The Focus of The Present Work

The application of shooting and expansion methods to nonautonomous circuits has been demonstrated and reported [AT72b,Fan75,Ske80,KS86]. Work toward using shooting methods for autonomous systems has been done in the past[AT72a,Fan75]. The application to autonomous systems is more difficult because the period of autonomous circuits is unknown. Modifications to the methods for nonautonomous circuits are required to deal with autonomous circuits.

The present work has concentrated on shooting methods because the emphasis has been on implementing methods that could be used to analyze autonomous systems. A program known as **SINC-S 87** that implements shooting methods for both autonomous and nonautonomous systems exists. This program was first written in the early seventies in FORTRAN [Fan75]. It was then converted to FORTRAN-77 and subsequently to *C* [GJ88] in 1987. In the conversion, imperfections were found in the implementation of the algorithms in the program. It was also found difficult to add new features to the steady-state portion of the program. In addition, **SINC-S 87** does not have a MOS model implemented in it. Consequently, the shooting methods have been implemented into **SPICE3** [QNPS87, Qua89] because of its use of good device models and the fact that it is not very difficult to add a new analysis-type into it.

The two shooting methods, as mentioned above, have been implemented into **SPICE3** and their performance has been evaluated. The algorithms implemented in **SPICE3** have improved heuristics so that faster and more accurate convergence to the steady-state is obtained. The program is available in the public domain.

A comparision of these methods has also been made with **SPECTRE** [KS86].

```
Rectifier Circuit for Illustrating the NR Algorithm
vin 1 0 sin(5 20 50)
d1 1 2 mod1
.model mod1 d (is=1e-16 cjo=2pf)
c1 2 0 1mF
r1 2 0 1k
*Command for steady-state analysis:
*steady act non_auto 20ms 50 .0001 duic 10.0
*plot v(2)
.end
```

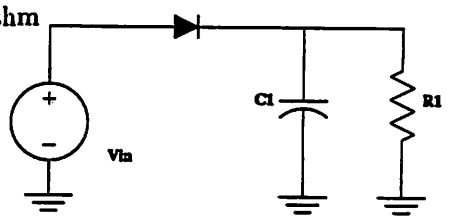


Figure 1.3: SPICE File for a Rectifier Circuit

SPECTRE implements the harmonic-balance algorithm mentioned above.

1.4 The NR Method: Explanation By Example

The Newton-Raphson[2] iterative process can be described by going through a typical simple example step by step. An example is the rectifier circuit shown in Figure 1.3. Since the circuit only has one reactive element there is only one state variable, making all the relevant equations one-dimensional. The SPICE file for the circuit is also shown in Figure 1.3. The description of the command for steady-state analysis and the associated parameters is given in the user's manual in Appendix A. The details of the shooting methods in general and the Newton-Raphson and extrapolation algorithms in particular are presented in Appendices C-D. To understand the following description, it is useful to know that a matrix of partial derivatives, known as the Jacobian, is required for every Newton-Raphson iteration. The steady-state algorithm based on Newton-Raphson iterations builds up the Jacobian by first computing a sensitivity matrix, a column of which is the set of partial derivatives of each of the state variables after one complete cycle with respect to a particular state variable at the beginning of the cycle. Because the sensitivity matrix is expensive to compute, heuristics are used to avoid its computation unless necessary.

The initial-state chosen for the circuit at the beginning of the steady-state analysis

[2]The use of the Newton-Raphson method in this context is distinct from the conventional use of the Newton-Raphson method by circuit simulators

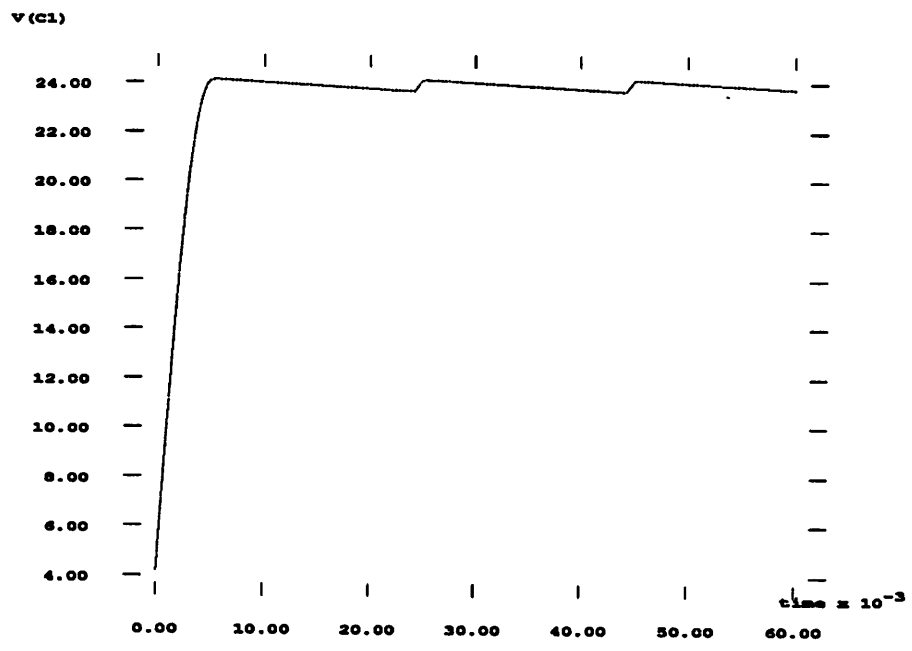


Figure 1.4: The First Three Cycles

does not have any bearing on the ability of the algorithm to determine the steady-state. Assuming that nothing is known about the steady-state of the circuit, it is reasonable to start with the capacitor voltage set to zero. The first n cycles of circuit simulation always use transient analysis without sensitivity computation and Newton-Raphson iteration is not used after a cycle to compute the initial conditions for the following cycle. This usually ensures that the extremely fast transients which may exist in the start-up phase have died out. The number of cycles so used should be such that the penalty paid when the circuit does not have very fast start-up transients is not excessive and at the same time when the very fast start-up transients are present, they are eliminated during these n cycles. $n = 3$ is found to be suitable for most of the examples. The resulting waveform of the voltage across the capacitor is shown in Figure 1.4. As can be seen, integration for three cycles is probably not necessary because the fast start-up transients are completed during the first cycle. All the same, the end result is that the circuit is taken into a state where Newton-Raphson iteration can be used if certain conditions are satisfied.One of the conditions that must be satisfied for a sensitivity computation to be carried out is that the difference between the states of the circuit at the start and end of the previous cycle should be within certain bounds. Since the state at the beginning of the first cycle is found to be far removed from the state at the end of the third cycle, this condition is not satisfied, and, the algorithm decides

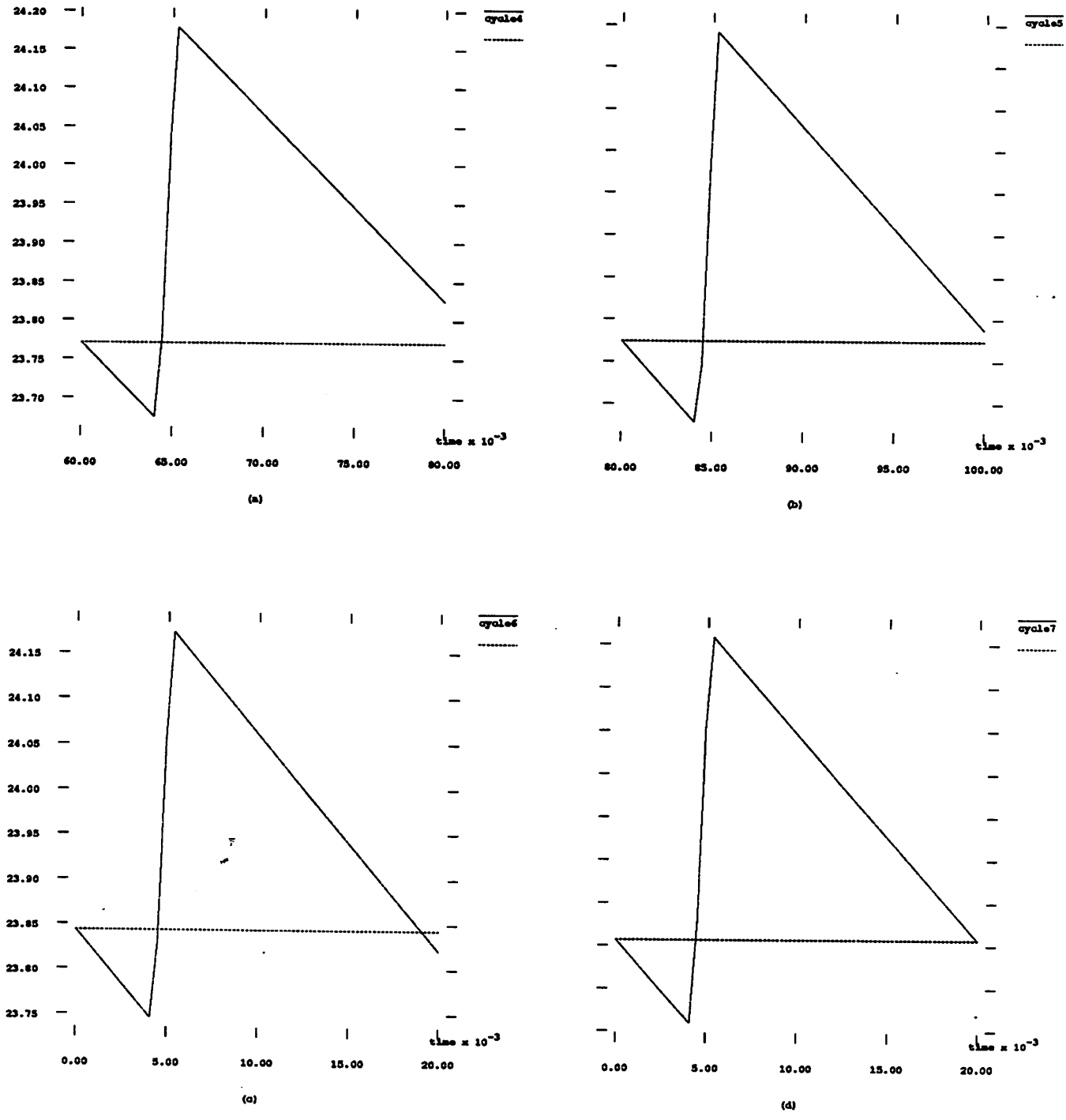


Figure 1.5: The Fourth, Fifth, Sixth and Seventh Cycles

not to do sensitivity computation in the fourth cycle and a Newton-Raphson iteration following it.The fourth cycle is then merely used to carry out one more cycle of transient analysis as in the first three cycles. The waveform for this cycle is shown in Figure 1.5(a). The state at the end of the fourth cycle is found to be close enough to the initial state used for this cycle that the algorithm decides to do a sensitivity computation during the fifth cycle. The waveform for the fifth cycle is shown in Figure 1.5(b). Instead of using the value of the state-variable at the end of the fifth cycle of transient analysis as the initial condition for the sixth cycle as was done for the earlier cycles, this time a Newton-Raphson iteration is done using the sensitivity of the final state to the initial state that was computed during the fifth cycle. The value so computed is used as the initial condition for the sixth cycle. The waveform for the sixth cycle is shown in Figure 1.5(c). The difference between the initial and the final states is found to be larger than the minimum error specified but small enough to be able to do a Newton-Raphson iteration to predict the initial condition for the seventh cycle and to do a sensitivity computation during the seventh cycle. The waveform for the seventh cycle is shown in Figure 1.5(d). The difference between the initial and the final states for the seventh cycle is found to be smaller than the specified minimum error. Hence, the circuit is said to have reached the steady-state. The state of the circuit at the end of the seventh cycle can then be used as the initial condition for the circuit when the circuit is required to go directly into the steady-state. Once such an initial condition is computed, the steady-state problem is said to have been solved.

Chapter 2

Results

2.1 Introduction

The Newton-Raphson-based and the extrapolation-based shooting methods have been implemented in **SPICE3** [QNPS87] to obtain the program **Sspice**. Sspice is identical to **SPICE3** apart from the addition of these two algorithms, and all the analyses and device models in **SPICE3** are available in Sspice. Shooting methods compute a set of capacitor voltages and inductor currents for the circuit so that if these values are used as the initial conditions for the circuit during a transient analysis, the circuit goes directly into the steady-state. Sspice should be used to first compute an initial condition for the circuit that corresponds to the circuit being in the steady-state. This initial condition can then be used during the subsequent analyses to obtain the steady-state circuit-parameters.

2.2 Comparision of the Algorithms

To evaluate the shooting methods, the ratio of the number of cycles of transient analysis required by these methods to the number of cycles required using conventional transient-analysis, such that some measure of the proximity to the steady-state is satisfied, is computed. A low ratio for a circuit indicates that the steady-state algorithm is efficient. In addition, the run-times required by the steady-state algorithms should be compared with the run-times required by conventional transient-analysis. Run-times are affected by the number of Newton-Raphson or extrapolation iterations required to reach the steady-state because each iteration has an computational overhead associated with it as explained in

Appendices C and D. Even though the run-times are important indications of the usefulness of a steady-state algorithm, run-times are implementation dependent. Since the aim of the present work has been to compare algorithms and not implementations, the run-times should be used with caution. The ratio of cycles and the run-times have both been tabulated for the two shooting methods and for conventional transient-analysis. Only the run-times have been tabulated for **SPECTRE**.

Another important indication of the efficacy of a steady-state algorithm is the number of circuits for which the algorithm is successful in finding the steady-state solution. Since the same circuits were used to test all the four methods, it is possible to use this criterion to compare the methods.

The results obtained when using conventional transient-analysis are shown in Table 2.1. The results from using the two implementations on typical circuits are presented in Table 2.2 and Table 2.3. Table 2.4 gives the results obtained for the same circuits using **SPECTRE** [KS86], a harmonic-balance-based simulator for steady-state analysis. Finally, Table 2.5 gives the run-times for the examples using all the four methods. The input file and the circuit diagram for each of the benchmark circuits are in Appendix B.

The results allow us to evaluate each of the methods independently as well as compare one method with another. It must be pointed out here that since the transient analysis in **SPICE3** and the extrapolation and Newton-Raphson methods have been implemented in the same framework and are both shooting methods, it is possible to compare not only their run-times but also their efficiencies in terms of the number of iterations that they require. On the other hand, **SPECTRE** is a separate entity altogether and it is possible to compare only its run-time with the run-times of the shooting methods.

It should be noted that the same error measure was used as the stopping criterion for all of the time-domain methods[1] including conventional transient-analysis. The error measure gives an indication of the distance of the circuit from the steady-state solution. The relative difference between the state of the circuit after a cycle of transient analysis to the initial state used for that cycle was found individually for each of the states. The maximum of these was then used as the error measure. This was then compared against a user specified *desired error* to check whether the steady-state had been reached. For some examples(Circuit B.6, Circuit B.7, etc), using the **tran** field in the **steady** command

[1] SPECTRE uses a different criterion to decide the proximity to the steady-state

Circuit	Number of States	Total # of Cycles	Run Time
DC power supply (Figure B.1)	4	61	6.13
C-B class C - lo Q (Figure B.2)	5	15	3.77
C-B class C - hi Q (Figure B.3)	5	42	11.82
C-B class C (Figure B.4)	11	22	16.02
X3 AMP (Figure B.5)	5	*	*
Colpitts Osc - lo Q (Figure B.6)[1]	3	241	18.9
Colpitts Osc - hi Q (Figure B.7)[1]	3	470	34.2
Hi Frq Colpitts Osc (Figure B.8)[1]	3	41	8.9
Wien Bridge Osc (Figure B.9)	2	6	4.87
OP AMP Wien Bridge Osc (Figure B.10)[1]	2	87	31.7
EC Colpitts Osc (Figure B.11)	3	86	20.5
EC XFRMR Coupled Osc (Figure B.12)[1]	3	>100	>66
Phase-Shift Osc (Figure B.13)	3	5	0.90
LC EC Oscillator (Figure B.14)[1]	2	22	5.43
BJT Rlxn Osc (Figure B.15)[1]	4	>100	>334

"*" - Very Large

[1]Cycles for conventional transient-analysis obtained from hi-res plot.

All times are in secs. on a VAX 8800

Table 2.1: Run-Time Statistics Using Transient Analysis

gave the wrong result. A high-resolution plot of the the state-variables obtained using conventional transient-analysis was used to establish the steady-state for these circuits. This method is less precise than the stopping criterion used for the Newton-Raphson and extrapolation methods.

2.2.1 Evaluation of the Newton-Raphson Method

The Newton-Raphson method can be evaluated by comparing the numbers in Table 2.2 with those in Table 2.1. Most of the additional computation per cycle of transient analysis in the Newton-Raphson method relative to the computational requirements per cycle of conventional transient analysis involves the computation of the sensitivity matrix which contains as elements the partial derivatives of the state of the circuit after a cycle of transient analysis to the initial state used for that cycle. This matrix is required for

computing the Jacobian needed for the Newton-Raphson iteration. It is observed that the Newton-Raphson algorithm usually leads to substantial savings relative to conventional transient analysis in the number of cycles required to reach the periodic steady-state. In addition, the savings in the number of cycles carry over to savings in the run-times indicating that the sensitivity-computation overhead is not large enough for the savings in the number of cycles to be nullified. There is a clear advantage in using the Newton-Raphson algorithm compared to conventional transient-analysis. The savings in the run-time and number of cycles required become more pronounced when conventional transient-analysis is found to require a large number of cycles to reach the steady-state (Circuit B.7). The savings are less pronounced when either the number of states is large (Circuit B.4) or when the steady-state can be reached using conventional transient-analysis in a small number of cycles (Circuit B.2).

Evaluation of the NR Method for Oscillators

The Newton-Raphson algorithm performs equally well for autonomous as well as nonautonomous circuits. This proves the validity of the modifications that had to be made to the algorithm so that autonomous circuits could be handled. Solving for the steady-state of a circuit is more difficult for the autonomous case than for the nonautonomous case because in the case of autonomous circuits the period is an unknown. It turns out that the number of independent equations available is only equal to the number of state variables in the circuit. This implies that the unknown period cannot just be added to the list of unknowns but its addition to the set of unknowns has to be accompanied by the removal of a state variable from the set. A reasonable value is assumed for the state variable removed from the set of unknowns as explained in Appendix C. Assuming a value for a state variable is equivalent to fixing the phase of the oscillator. Because the phase of the oscillator is not of concern, this modification should not affect the iterative process, theoretically. The results in Table 2.2 are an experimental verification of this theory. All the circuits that were used as test cases are highly nonlinear and a majority of them are oscillators. The improvement in the run-time because of the use of the Newton-Raphson algorithm has been found to be by as much as a factor of 5.5 for some circuits (Circuit B.7).

Circuit	Number of States	Number of Iterations	Total # of Cycles	Ratio of Cycles	Run Time
DC power supply (Figure B.1)	4	5	12	0.20	3.97
C-B class C - lo Q (Figure B.2)	5	3	8	0.53	4.78
C-B class C - hi Q (Figure B.3)	5	4	9	0.21	5.67
C-B class C (Figure B.4)	11	6	15	0.68	17.42
X3 AMP (Figure B.5)	5	21	34	*[1]	22.78
Colpitts Osc - lo Q (Figure B.6)	3	7	14	0.06[1]	6.81
Colpitts Osc - hi Q (Figure B.7)	3	6	12	0.03[1]	6.05
Hi Freq Colpitts Osc (Figure B.8)	3	10	14	0.34[1]	11.68
Wien Bridge Osc (Figure B.9)	2	1	6	1.0	4.98
OP AMP Wien Bridge Osc(Figure B.10)	2	6	12	0.14[1]	3.70
EC Colpitts Osc (Figure B.11)	3	14	26	0.30	19.82
EC XFRMR Coupled Osc (Figure B.12)	3	14	24	<0.24[1]	21.70
Phase-Shift Osc (Figure B.13)	3	4	10	2.0	3.73
LC EC Oscillator (Figure B.14)	2	6	12	0.55[1]	11.60
BJT Rlxn Osc (Figure B.15)	4	2	8	<0.08[1]	19.33

Ratio of cycles = Total cycles required by algorithm / Total cycles required by transient-analysis

"*" - Very Small

[1]Cycles for conventional transient-analysis obtained from hi-res plot.

All times are in secs. on a VAX 8800

Table 2.2: Run-Time Statistics for Newton-Raphson Based Algorithm

Drawbacks of the NR Method

The main drawback of the Newton-Raphson method is the inability to handle waveforms that do not have well defined periodicity. Such situations can, for example, arise when the circuit is a multiplier and the inputs have frequencies that are not integral multiples or when the circuit is a squegging oscillator. This inability to handle multiple frequencies is probably the reason why the algorithm seems to work slightly better for the *high-Q* version than for the *low-Q* version of two of the circuits tested. The nonlinearities in the circuits result in the generation of harmonics of the frequency of interest. In addition, the resonant start-up transients have frequencies slightly off the center frequency of oscillation. A correctly designed circuit with higher Q implies lower relative amplitudes for the uninteresting harmonics than in the lower Q case. In other words, the periodicity of the composite waveform at the frequency of interest is more pronounced for a *high-Q* circuit than for the corresponding *low-Q* circuit.

One of the circuits tested was an *X3* amplifier. The waveform at a node in this circuit would have harmonic components with substantial amplitude. But the algorithm was able to handle this circuit because all the frequencies in the steady-state are integral multiples of the input frequency. Hence, the periodicity of the waveform at any node is defined by the frequency of the input signal.

2.2.2 Evaluation of the Extrapolation Method

The simulation results obtained using the extrapolation method are given in Table 2.3. The extrapolation method operates successfully only for nonautonomous circuits. When compared with conventional transient-analysis, the gain in the number of cycles required was by as much as a ratio of 2.5*to*4.0:1 for some circuits (Circuits B.1 and B.3) while the gain in the run-time was by as much as a ratio of 2:1 (Circuits B.1). But extrapolation can be expected to give good speedups over transient analysis only when the number of reactive elements in the circuit is small. The overhead of each iteration is expensive enough that the computation time using extrapolation becomes larger than the computation time using normal transient-analysis fairly rapidly with the number of reactive elements. The common-base *class-C* circuit with 11 states the extrapolation algorithm has a larger run-time than normal transient-analysis even though the number of cycles that are required is a little more than half of that required by normal transient-analysis.

Circuit	Number of States	Number of Iterations	Total # of Cycles	Ratio of Cycles	Run Time
DC power supply (Figure B.1)	4	1	14	0.23	3.32
C-B class C - lo Q (Figure B.2)	5	2	18	1.2	35.13
C-B class C - hi Q (Figure B.3)	5	2	20	0.48	38.18
C-B class C (Figure B.4)	11	1	20	0.91	51.72
X3 AMP (Figure B.5)[1]	5	*	*	-	*

Ratio of cycles = Total cycles required by algorithm / Total cycles required by transient-analysis

"*" - Very Large

[1]Cycles for conventional transient-analysis obtained from hi-res plot.

All times are in secs. on a VAX 8800

Table 2.3: Run-Time Statistics for the Extrapolation Based Algorithm

A large number of heuristics were tried without success for computing the steady-state of autonomous circuits using the extrapolation method. It was not possible to find the solution for any of the oscillator circuits tested using the extrapolation algorithm.

Comparision of Extrapolation with the NR Method

For driven circuits, the simulation results for the Newton-Raphson algorithm and for the extrapolation algorithm can be compared. It can be seen from the data of Tables 2.2 and 2.3 that even though the extrapolation method requires fewer iterations to reach the steady-state in general, it usually requires more cycles of transient analysis and has either a run-time that is comparable or larger. This fact is again brought out by the common-base *class-C* circuit with 11 states (Circuit B.4). Even though the number of iterations required by the Newton-Raphson method is three times the number of iterations required by extrapolation, the run-time for the extrapolation method is more than twice as large. Each iteration in the extrapolation method consists of up to $n + 2$ cycles of integration, where n is the number of states. On the other hand, the Newton-Raphson requires just one cycle of transient analysis per iteration, where each cycle has an overhead added to it due to the sensitivity computation requirement. In general, the Newton-Raphson method is less cpu-expensive than the extrapolation method.

Circuit	Number of States	Run Time
DC power supply (Figure B.1)	4	3.47
C-B class C - lo Q (Figure B.2)	5	15.32
C-B class C - hi Q (Figure B.3)	5	1.17
C-B class C (Figure B.4)	11	46.18
X3 AMP (Figure B.5)	5	6.03

All times are in secs. on a VAX 8800

Table 2.4: Run-Time Statistics for SPECTRE - a Harmonic-Balance Based Program

2.2.3 Shooting Methods and Harmonic-Balance

Results for the test circuits using the harmonic-balance algorithm are given in Table 2.4. The run-times for the driven circuits using harmonic-balance are found to be comparable to those using the Newton-Raphson method on the average. But the Newton-Raphson method is found to be about 2.5 times better for the highly nonlinear common-base class C amplifier with 11 states (Circuit B.4). Shooting methods are more suitable, in general, for highly nonlinear circuits or highly nonsinusoidal waveforms than harmonic-balance. A physical explanation for this is that shooting methods require the values of a signal in the circuit only at the start and end of a cycle while harmonic-balance requires the values of the signal at a large number of points in between the start and end points. On the other hand, harmonic-balance is known to be well suited for microwave circuits since distributed elements can be included in the circuit. Even though it is possible to model microwave circuits by means of lumped approximations, the large number of states that are required for this purpose make the use of shooting methods impractical.

The run-times for all the methods are shown together in Table 2.5 for an overall comparision. It can be seen that the Newton-Raphson method performs better than conventional transient-analysis and the other steady-state algorithms on the average. The savings in run-time can be expected to increase rapidly when convergence to the steady-state is desired to a greater accuracy The reason for this is that once the steady-state algorithms are close to the steady-state, convergence to the steady-state is quadratic. Convergence using conventional transient-analysis, on the other hand, is linear.

Circuit	Newton-Raphson	Extrapolation	Harmonic-Balance	Transient Analysis
DC power supply (Figure B.1)	3.97	3.32	3.47	6.13
C-B class C - lo Q (Figure B.2)	4.78	35.13	15.32	3.77
C-B class C - hi Q (Figure B.3)	5.67	38.18	1.17	11.82
C-B class C (Figure B.4)	17.42	51.72	46.18	16.02
X3 AMP (Figure B.5)	22.78	*	6.03	*[1]
Colpitts Osc - lo Q (Figure B.6)	6.81	-	-	18.9[1]
Colpitts Osc - hi Q (Figure B.7)	6.05	-	-	34.2[1]
Hi Frq Colpitts Osc (Figure B.8)	11.68	-	-	8.9[1]
Wien Bridge Osc (Figure B.9)	4.98	-	-	4.87
OP AMP Wien Bridge Osc (Figure B.10)	3.70	-	-	31.7[1]
EC Colpitts Osc (Figure B.11)	19.82	-	-	20.5
EC XFRMR Coupled Osc (Figure B.12)	21.70	-	-	>66[1]
Phase-Shift Osc (Figure B.13)	3.73	-	-	0.90
LC EC Oscillator (Figure B.14)	11.60	-	-	5.43[1]
BJT Rlxn Osc (Figure B.15)	19.33	-	-	>334[1]

"*" - Very Large

[1]Cycles for conventional transient-analysis obtained from hi-res plot.

All times are in secs. on a VAX 8800

Table 2.5: Comparision of the Run-Times

Chapter 3

Adding a Steady-State Algorithm to SPICE3

The modular structure of **SPICE3** [QNPS87] offers a significant advantage when adding a new analysis capability. New analyses can be added to **SPICE3** without having to understand all the intricacies of **SPICE3**. The steady-state analysis portion of the program **Sspice** has been organized with the same philosophy, so that including additional steady-state algorithms to the basic skeleton should be an easy task. In this chapter the steps that were taken to add the Newton-Raphson [AT72b,CT73] and extrapolation [Ske80] steady-state algorithms to **SPICE3** are outlined. Similar steps have to be taken to add any other new steady-state algorithms. In addition, the steps that were taken to add the basic skeleton for steady-state analysis are similar to the steps that would be required to add any new analysis type to **SPICE3**. Some files specific to **nutmeg** [CNPS87], the input-output processor used by **SPICE3**, also have to be modified.

The preliminary steps in adding the steady-state analysis skeleton involved modifying the following **SPICE3**-specific and *nutmeg*-specific files:

- spiceif.c
- cmdtab.c
- runcoms.c
- types.c
- INP2dot.c

- SPIinit.c
- CKTdoJob.c

spiceif.c is a *nutmeg* file that contains routines which interface **SPICE3** to *nutmeg*. The existence of the steady-state analysis skeleton has to be made known to this file in two locations for input parsing. *cmdtab.c* is a *nutmeg* file that contains a list of all available commands. The command for steady-state analysis has to be added to this file in one location. *runcoms.c* is also a *NUTMEG* file that contains one function each for all the commands available. A function called *com_steady(wl)* has to be added for steady-state analysis. This function (see Figure 3.1) calls *dosim()* with the correct arguements. *types.c* is a *nutmeg* file that defines types for plots. A type for steady-state analysis has to be added to this file. These four files are the only *nutmeg*-specific files that have to be modified, in general, to add a new analysis.

```
void
com_steady(wl)
        wordlist *wl;
{
        dosim("steady", wl);
        return;
}
```

Figure 3.1: Function Added to *runcoms.c* for Steady-State Analysis

The function in **SPICE3** that calls most analyses is in *CKTdoJob.c*. The call to the top-level steady-state function has to be added to *CKTdoJob.c*. The top-level function for steady-state analysis is *STEADYan()* and the call to this function is found in *CKTdoJob.c*. The fields that need to be present on the command line for the steady-state analysis command are coded in the **SPICE3** file called *INP2dot.c*. The data-type of each input parameter is specified in this file. The function *INP2dot()* is organized as *if-then-else* clauses. One such clause has to be added for steady-state analysis. The *STEADYinfo* structure, which is defined in *STEADYsetParm.c*, is declared in *SPIinit.c*.

The new files that had to be added to complete the basic skeleton for steady-state analysis are the following:

- STEADYsetParm.c
- STEADYaskQuest.c
- STEADYan.c
- STEADY.h

The *STEADYinfo* structure is defined in *STEADYsetParm.c*. The basic data structure used by steady-state analysis (see Figure 3.2) is *STEADYAN*. The function *STEADYsetParm* transfers the input parameters to this data structure. The file *STEADYaskQuest.c* contains a function that answers queries about steady-state analysis. The top-level function *STEADYan()* for steady-state analysis is in *STEADYan.c*. Once this function is called, control is transferred to steady-state analysis. *STEADY.h* is a header file specific to steady-state analysis. The *STEADYAN* structure is defined here. Once the above modifications have been made to the basic **SPICE3** structure, the basic skeleton for steady-state analysis is complete.

After the basic skeleton is added, the actual steady-state algorithms can be incorporated into it. The *STEADYan()* function calls the function corresponding to the desired algorithm. The algorithm desired is specified on the command line for steady-state analysis. For each new algorithm added, a new field is added to the *STEADYAN* structure corresponding to the pointer to the algorithm-specific structure. The algorithms implemented need to call *DCtran()*, which is a **SPICE3**-specific function that does transient analysis. The *STEADYAN* structure has to be padded with dummy fields corresponding to the fields present in the transient-analysis-specific structure so that *DCtran()* can be called from the steady-state-analysis-specific functions. The first three fields in *STEADYAN* are fields that are mandatory for all analyses. Apart from the fields in *STEADYAN* mentioned above, the rest of the fields correspond to the other input parameters.

To add a new steady-state algorithm to the basic skeleton, only the minor additions have to made to the steady-state-specific files. An extra field has to be added to the *STEADYAN* structure in *STEADY.h* corresponding to the pointer to algorithm-specific structure. A call to the top-level algorithm-specific function has to be inserted in

```
typedef struct {
    int JOBtype;
    JOB *JOBnextJob; /* pointer to next thing to do */
    char *JOBname; /* name of this job */

    double TRANfinalTime; /* have to duplicate the fields in the transient */
    double TRANstep; /* analysis structure so that DCtran can be called */
    double TRANmaxStep; /* from the steady state package */
    double TRANinitTime;
    long TRANmode;

    enum algorithm SteadyAlgorithm; /* name of the algorithm */
    enum mode SteadyMode; /* indicates whether autonomous or nonautonomous */
    double SteadyPeriod; /* period of oscillation */
    int SteadyNumpts; /* number of points per cycle */
    double SteadyTol; /* tolerance within which steady-state is to be obtained */
    int SteadyUic; /* indicates whether uic flag is set */
    double SteadyDampFact; /* damping factor for the Newton-Raphson method */

    ACTAN *act; /* pointer to the data structure for the Newton-Raphson algorithm specific functions */
    EXTAN *ext; /* pointer to the data structure for the extrapolation algorithm specific functions */
} STEADYAN;
```

Figure 3.2: Data Structure Used by the Steady-State functions in Sspice

STEADYan() in *STEADYan.c.* Minor modifications have to be made to *STEADYsetParm.c* and *STEADYaskQuest.c* to recognize that the algorithm field on the *steady* command line can have one more possible choice of algorithms. Once this has been done, new algorithm-specific files can be added to carry out the required functions. Each algorithm has its own header file and an independent file that contains its top-level function. In addition, there has to an algorithm-specific file that does the memory allocation for the structure used by the algorithm. In general, each algorithm uses the devices in a different way. There has to be at least one algorithm-specific file for each device that the algorithm needs to access. **SPICE3** is constructed so that the device-dependent functions are independent of the analysis-specific functions to a very large degree. The addition of any new device-dependent has to be logged in three places. First, if the function is the first of its kind for any device, it has to be declared in *DEVdefs.h.* Any function added for a device has to be logged in the device-specific header file. In addition, once a function has been declared in *DEVdefs.h* it also has to be logged in *dev.c* for all devices (*dev* is a generic name for a device. The file name could be *CAP.c*, *IND.c* or a similar name for the other devices). The order in which it is logged in *dev.c* should be the same as that in *DEVdefs.h.* If the function is not present for a particular device, *dev*, a *NULL* should be logged in place of the function name in *dev.c.*

Once these requirements have been satisfied, the remaining algorithm-specific functions can be added. The following new files have been added specifically for the Newton-Raphson algorithm:

- ACT.h
- ACTan.c
- ACTerror.c
- ACTinit.c
- ACTsenSolve.c
- ACTstateUpdt.c
- *dev*actLoad.c (*dev*ice-specific)
- NIactIntegrat.c

- NIactSolve.c

The substring *act* is common to all the files added. In addition, a **SPICE3**-specific file, *NIiter.c*, has been modified so that the LU-factored circuit Jacobian is available for use in the sensitivity computations. The data structure pertaining to the Newton-Raphson algorithm is shown in Figure 3.3. The main loop of the Newton-Raphson algorithm is in the function *ACTan()* in the file *ACTan.c*. The call to *DCtran()* are made from this function. *ACTinit()* in *ACTinit.c* allocates memory for the data structures required by the Newton-Raphson algorithm and initializes them. The loop for sensitivity computation is present in *ACTsenSolve()* which is in *ACTsenSolve.c*. This function is called from *DCtran()* after every time-point at which a solution is found for the circuit. *ACTsenSolve()* loops through all the devices and loads the right hand side required for sensitivity computation (see Appendix C). The function *SMPsolve()* is then called to solve for the sensitivities. This process is repeated for every column in the sensitivity matrix. *ACTstateUpdt.c* contains the function *ACTstateUpdate()* that loops through the devices to load the capacitor voltages and inductor currents into the data structure for the Newton-Raphson method. *NIactIntegrat.c* contains the function *NIactIntegrate()* which is a modified version of the function *NIintegrate()* used by **SPICE3**. This function does integration by the trapezoidal rule for the reactive elements in the sensitivity circuits. *NIactSolve()* is a function in *NIactSolve.c* for factoring a non-sparse matrix and for doing the subsequent forward and backward substitutions. This function is required to be called every time a Newton-Raphson iteration is done because the Jacobian is not sparse.

The following new files have been added specifically for the extrapolation-based steady-state algorithm:

- EXT.h
- EXTan.c
- EXTinit.c
- EXTgetPeriod.c
- EXTstateUpdt.c
- *dev*extLoad.c (*dev*ice-specific)

The substring *ext* is common to all the files added. The data structure used by the extrapolation algorithm is shown in Figure 3.4. *DCtran.c* is another **SPICE3**-specific file that had to be modified for the extrapolation and Newton-Raphson algorithms. This was done so that calls to algorithm-specific functions could be made from inside the function *DCtran()*. *EXTan()* is a function in *EXTan.c* which contains the main loop for the extrapolation-based algorithm. *EXTinit()* in *EXTinit.c* does the memory allocation and the initialization for the data structures used by the extrapolation-based method. *EXTgetPeriod()* in *EXTgetPeriod.c* computes the current period of the signal. This function is called from *DCtran()* after every time-point at which a solution is obtained for the circuit (see Appendix D for details). *EXTstateUpdate()* in *EXTstateUpdt.c* loads the capacitor voltages and inductor currents into data structures used by the extrapolation method.

```
typedef struct {
    int ActSenUpdate; /* update sen matrix if ActSenUpdate = 1 */

    double **ActSenMat; /* sensitivity matrix */
    double **ActSenMatOld; /* old sensitivity matrix required for integration */
    double **ActCurState; /* the current calculated at the current time point */
    double **ActCurStateOld; /* the current calculated at the previous time point */
    double *ActDqDv; /* The nonlinear caps(inds) at the previous time point */
    double *ActSenT; /* sensitivity wrt T for the autonomous case */
    double *ActCktState0k; /* state x(0) */
    double *ActCktState0Kp1;
    double *ActCktStateT; /* state x(T) */
    double *ActCktStateArch; /* state x*(0) */

    int ActSubCol; /* the entry chosen to be replaced by the period */

    double **ActJacMat; /* the jacobian matrix in some form */
    double *ActRhs; /* the right hand side used during the newton iteration */
    int ActNumStates;

    int ActNumIterOfState;
    int ActNumIterOfSenMat;
    int ActNumTimePoint;

    int ActDqDvFlag;

    double ActError1, ActError2;
    double ActError1Old, ActError2Old;

    double ActDenominator; /* yet another restricted global variable! */

    double ActMaxSenT; /* max sensitivity wrt T */

} ACTAN;
```

Figure 3.3: Data Structure Used by the NR Algorithm

```
typedef struct {
    double **ExtState;
    double *ExtStateT;
    double *ExtStateTprime;
    double *ExtStateExtrap;

    double ExtPeriod;

    double ExtMinNorm;
    double ExtTimeOfMinNorm;

    int ExtGetPeriod;
    int ExtNumStates;
    int ExtNumIterOfState;
} EXTAN;
```

Figure 3.4: Data Structure Used by the Extrapolation Algorithm

Appendix A

Manual for Sspice

Sspice is an augmented version of *SPICE3* in which algorithms for the computation of the periodic steady-state are implemented. The algorithms currently implemented are the Newton-Raphson-based and the extrapolation-based methods. Both these methods belong to the general class of methods known as shooting methods. The algorithms can be used by means of a new analysis-type called **steady** that has been incorporated into *SPICE3*.

The algorithm to be used for steady-state analysis is specified on the command line for steady-state analysis. The usual *SPICE3* analyses can still be used from within *Sspice*. To use the steady-state analysis, the **steady** command, with the fields on the command line appropriately specified, is used. A review of the user's manual for *SPICE3* [QNPS87] may be necessary before using *Sspice*. The fields that need to be specified on the **steady** command line are described below in the order in which they should occur. The general steady-state command is the following:

steady *'algorithm' 'circuit type' 'time-period' 'no. of points' 'accuracy'*
'initial condition flag' 'damping'

A.1 Description of the Options Available for Steady-State Analysis

Specifying the Algorithm

This field, **act** in the example (see Figures A.1 and A.2), specifies the algorithm that one desires to use for steady-state analysis. Currently, two algorithms have been implemented in *Sspice*. Thus, the algorithm name can be either **act** when the Newton-Raphson-based method is to be used, **ext** when the extrapolation-based method is to be used or **tran** when conventional transient-analysis is to be used.

Specifying the Type of the Circuit

In the example, the field, **auto**, specifies that the circuit being simulated is an oscillator. A driven circuit can have a periodic input which may be either sinusoidal or

nonsinusoidal. This parameter can either be **auto** for an autonomous (oscillator) circuit or **non_auto** for a nonautonomous (driven) circuit.

Specifying the Time-Period

The time-period is a real number giving the value of the time period for a nonautonomous circuit or a close estimate for an autonomous circuit. The time-period in the example is 0.62μseconds.

Specifying the Number of Points

The 'number of points' is the minimum number of points per cycle at which the simulator should solve the circuit. This is an integer that sets a bound on the maximum time-step taken during an integration. 50 points per cycle has been found to be sufficient for all the example circuits.

Specifying the Desired Accuracy

The desired accuracy is a real number specifying the accuracy with which convergence to the steady-state is tested. This number is compared with the difference between the state of the circuit after a cycle of transient analysis and the initial circuit-state used for that cycle. An accuracy of 0.01 was used for all of the example circuits.

Specifying Whether the Device Initial-Conditions Should be Used

The flag to indicate whether the device initial-conditions should be used can be either **uic**, which is identical to the **uic** flag in *SPICE3*, or **duic** which means that the **uic** flag has been turned off for the first cycle. The **uic** flag is always on for all subsequent cycles.

Specifying the Damping

When the algorithm specified is **act**, the 'damping' is used for computing the damping factor for the Newton-Raphson iterations. The higher the value of the damping factor, the closer the damping factor is to unity. Different values of the damping factor may be suitable for different problems. A value of 10.0 was used for all of the example circuits. When the algorithm used is **ext**, the 'damping' specifies the number of cycles of transient analysis carried out before the extrapolation algorithm is applied. Three cycles of integration before the first extrapolation iteration were found to be sufficient for all the circuits for which the extrapolation algorithm was successful.

A.2 An Example of the use of Steady-State Analysis

The circuit, with the corresponding **SPICE3** file, used for the steady-state analysis example shown in Figure A.1 is shown in Figure A.2. The algorithm field on the **steady** command line in Figure A.1 is specified to be **act**, which implies that the Newton-Raphson

```
shell_prompt> Sspice
/* Example of the use of the Newton-Raphson algorithm */
Sspice_prompt> source ckt
Sspice_prompt> steady act auto 0.62uss 50 .01 uic 10.0
Sspice_prompt> plot v(1) - v(3)
```

Figure A.1: Example of the use of Steady-State Analysis

```
Colpitts Oscillator: q=50 [Fan75]
rl 3 1 10k
re 2 4 20k
q1 1 0 2 mod1
.model mod1 npn (rb=100 rc=20 tf=.1ns)
l1 3 1 20uh ic=0
c1 1 2 0.5nf ic=0
c2 2 0 40nf ic=-0
vcc 3 0 10
vee 4 0 -10
.end
```

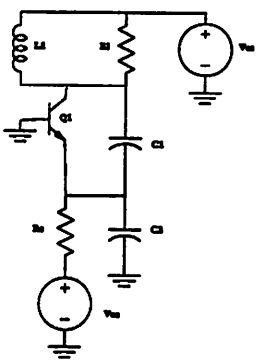


Figure A.2: Colpitts Oscillator with a Q of 50

method will be used for steady-state analysis. The circuit is a colpitts oscillator. Because the circuit is autonomous, the second field on the command line is specified as **auto**. The period of the input source is 0.62μseconds, and this is specified in the third field. The fourth field specifies the minimum number of points per cycle that the circuit must be solved for. 50 points per cycle is found to be sufficient for the example circuit. The fifth field is the accuracy with which convergence to the steady-state is required. An accuracy of 0.01 is used for the example circuit. The sixth field is specified to be **uic** implying that the device initial conditions will be used for the first cycle. A 'damping' of 10.0 is used for the circuit.

The circuit is first sourced into Sspice as shown in Figure A.1. Then the steady-state analysis is carried out. Sspice saves the data from each cycle of transient analysis leading to the steady-state. It is possible to plot the waveform corresponding to any variable for any of the cycles leading to the steady-state. The choice of the cycle to be plotted can be chosen using the **setplot** command in *nutmeg*. The output produced by Sspice is shown in Figure A.3. The progression of the circuit towards the steady-state is shown in the series of plots in Figure A.4. This particular set of of plots is interesting because it shows a typical example of how the Newton-Raphson algorithm is capable of recovering from an error during one of the intermediate iterations.

A.3 Distribution of the Program

The source-code for the program and additional copies of this report are available in the public domain through the EECS/ERL Industrial Support Office, Cory Hall, the University of California, Berkeley.

```
The circuit has 3 state(s).
The Steady-State has been reached!
Values of the state variables in the circuit are the following :
c2 : -0.7700 Volts
c1 : 4.8747 Volts
l1 : 0.0259 Amps
The period of waveform: 6.245321e-07
The number of NR iterations required : 7
Number of cycles required : 14
CPU time for the steady-state analysis = 7.63 sec
```

Figure A.3: Output Produced by Sspice for the Example

Comments

- *Sspice* is still in the developmental stage.
- The extrapolation *algorithm* is not available for oscillators.
- All the fields in the **steady** command are mandatory.

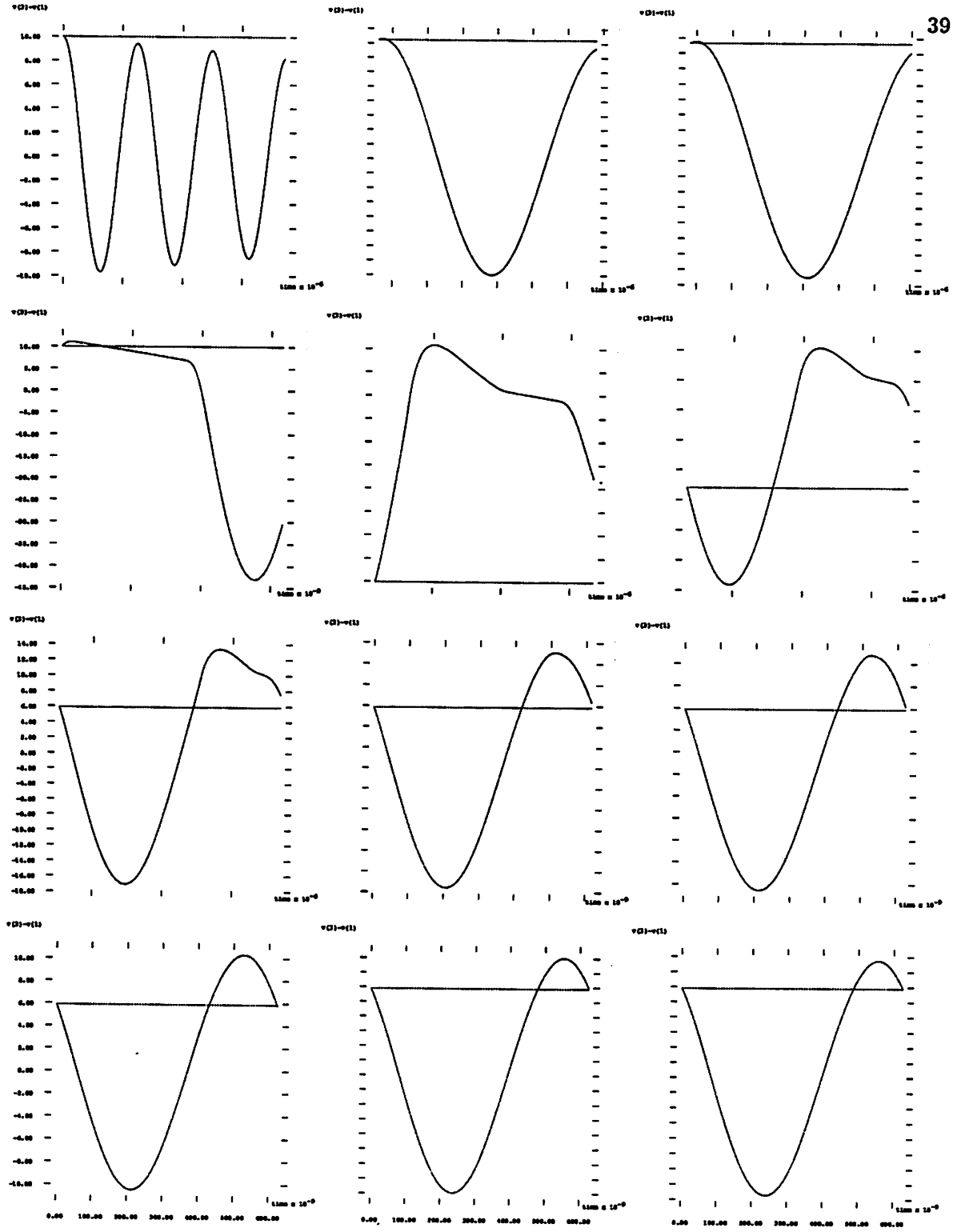


Figure A.4: Sequence of Plots Showing Progression to the Steady-State

Appendix B

Input Files for the Test Circuits

```
DC Power-Supply [Ske80]
vin 1 0 sin(0 20 50)
d1 1 2 mod1
.model mod1 d (is=1e-16 cjo=2pf)
c1 1 2 1uF
c2 3 0 1mF
c3 4 0 1mF
l1 3 4 0.1H
r1 2 3 5
r2 4 0 1k
*plot v(4)
*commands for steady-state analysis:
*steady act non_auto 20ms 50 .01 uic 10.0
*steady ext non_auto 20ms 50 .01 uic 3.0
*steady tran non_auto 20ms 50 .01 uic 10.0
.end
```

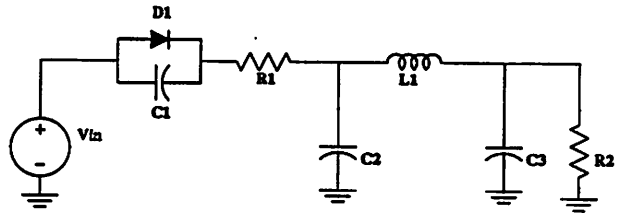


Figure B.1: DC Power-Supply

```
C-B CLASS C AMPLIFIER (LO-Q) [Fan75]*
RB1 7 6 3.9K
RB2 6 0 1.2K
CB 6 0 1UF
RE 4 0 510
RC 7 5 1.6K
LT 7 5 50UH
CT 7 5 2NF
Q1 5 6 4 mod1
.model mod1 NPN(IS=1.e-14 BF=80 RB=100
+RC=10 CJE=2PF CJC=2PF VA=50)
VCC 7 0 15
RS 1 2 50
LS 2 3 10UH
CS 3 4 10NF
RLK 3 4 100MEG
VSIN 1 0 sin(0 500MV 500KHZ)
*PLOT -5 25 VOUT 5 0
*commands for steady-state analysis :
*steady act non_auto 2us 50 .01 duic 10.0
*steady ext non_auto 2us 300 .01 duic 3.0
*steady tran non_auto 2us 50 .01 duic 10.0
.end
```

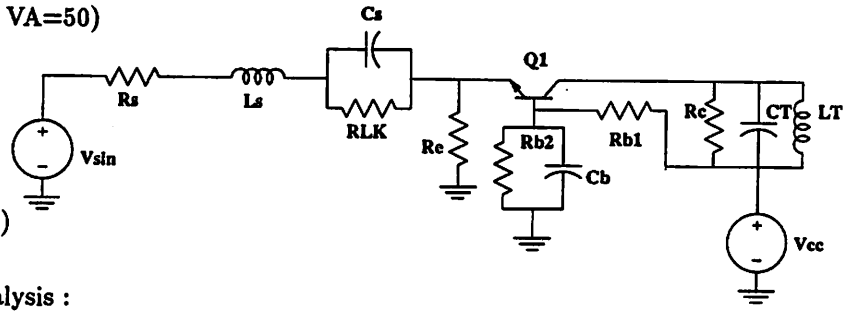


Figure B.2: Common-Base Class-C Amplifier (Low-Q)

```
C-B Classs-C Amplifier (HI-Q) [Fan75]
RB1 7 6 3.9K
RB2 6 0 1.2K
CB 6 0 1UF
RE 4 0 510
RC 7 5 1.6K
LT 7 5 10UH
CT 7 5 10NF
Q1 5 6 4 mod1
.model mod1 NPN(BF=80 RB=100
+RC=10 CJE=2PF CJC=2PF VA=50)
VCC 7 0 15
RS 1 2 50
LS 2 3 50UH
CS 3 4 2NF
RLK 3 4 100MEG
VSIN 1 0 sin(0 500MV 500KHZ)
*PLOT -5 25 VOUT 5 0
*commands for steady-state analysis:
*steady act non_auto 2us 50 .01 duic 10.0
*steady tran non_auto 2us 300 .01 duic 3.0
.end
```

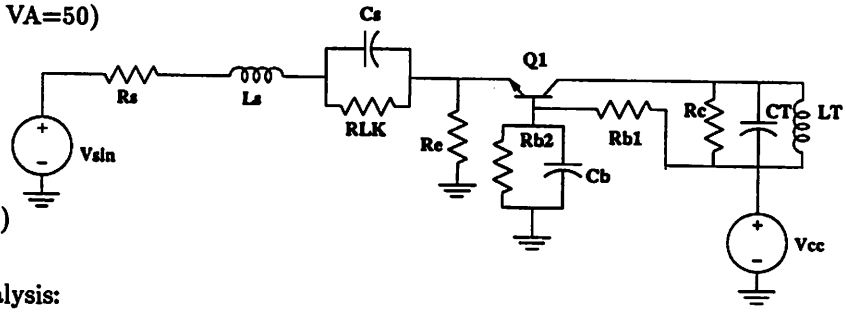


Figure B.3: Common-Base Class-C Amplifier (High-Q)

```
Class-C Amplifier [CT73]
r1 1 0 50
r2 6 0 50
c1 1 2 100pf
c2 2 0 10pf
c3 3 0 10pf
c4 4 0 10pf
c5 5 0 10pf
c6 5 6 100pf
c7 7 0 100pf
l8 1 3 0.025uh
l9 3 0 1.2uh
l10 4 5 .025uh
l11 5 7 1.2uh
q1 4 0 3 mod1
.model mod1 NPN(IS=1e-16 BF=80
+RB=100 VA=50 tf=.1ns)
vdc 7 0 30v
Iin 1 0 sin(0 0.1 100MEGHz)
*plot v(6)
*commands for steady-state analysis:
*steady act non_auto .01us 50 .01 uic 10.0
*steady ext non_auto .01us 300 .01 uic 3.0
*steady tran non_auto .01us 50 .01 duic 10.0
.end
```

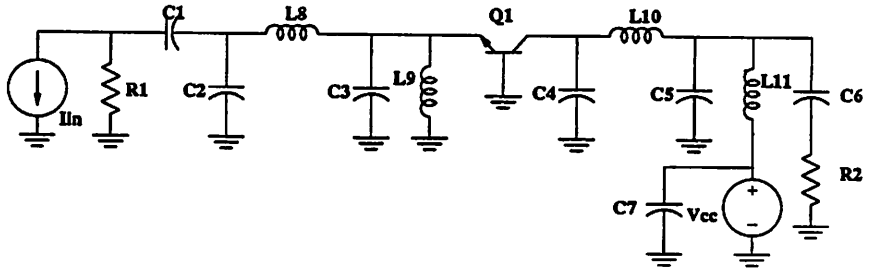


Figure B.4: Common-Base Class-C Amplifier With a Large Number of States

```
X3 FREQUENCY MULTIPLIER [Fan75]
RE 3 5 5K
CE 3 0 1UF
RC 2 6 10
Q1 6 1 3 mod1
.model mod1 NPN(IS=1e-14 BF=80 RB=100
+CJE=5PF CJC=2PF VA=50)
RT 4 2 2K
LT 4 2 1UH
CT 4 2 112PF
RS 1 0 1K
CS 1 0 500PF
LS 1 0 2UH
ISIN 1 0 sin(0 250UA 5MEGHZ)
VCC 4 0 10
VEE 5 0 -10
*frequency : 5MEGHZ
*command for steady-state analysis :
*steady act non_auto .2us 50 .01 uic 10.0
*steady tran non_auto .2us 50 .01 uic 10.0
.end
```

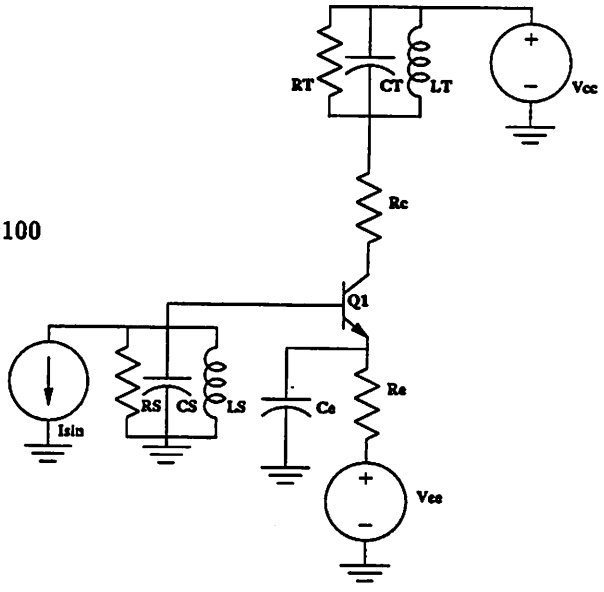


Figure B.5: X3 Frequency Multiplier

```
Colpitts Oscillator: q=50 [Fan75]
rl 3 1 10k
re 2 4 20k
q1 1 0 2 mod1
.model mod1 npn (rb=100 rc=20 tf=.1ns)
l1 3 1 20uh ic=0
c1 1 2 0.5nf ic=0
c2 2 0 40nf ic=-0
vcc 3 0 10
vee 4 0 -10
* period = 0.62us
*vout = v(3) - v(1)
*command for steady-state analysis :
*steady act auto .62us 50 .01 uic 10.0
*steady tran auto .62us 50 .01 uic 10.0
.end
```

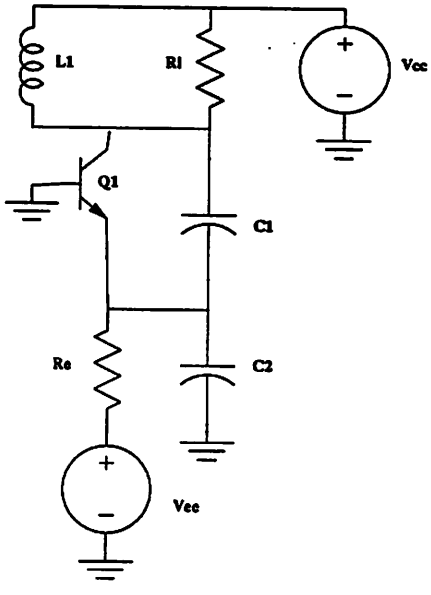


Figure B.6: Colpitts Oscillator with a Q of 50

```
Colpitts oscillator: q=100 [Fan75]
rl 3 1 10k
re 2 4 20k
q1 1 0 2 mod1
.model mod1 npn (rb=100 rc=20 tf=.1ns)
l1 3 1 10uh ic=0ma
c1 1 2 1.013nf ic=0v
c2 2 0 79nf ic=-0v
vcc 3 0 10
vee 4 0 -10
* period = 0.62us
*vout = v(3) - v(1)
*command for steady-state analysis :
*steady act auto .62us 50 .01 uic 10.0
*steady tran auto .62us 50 .01 uic 10.0
.end
```

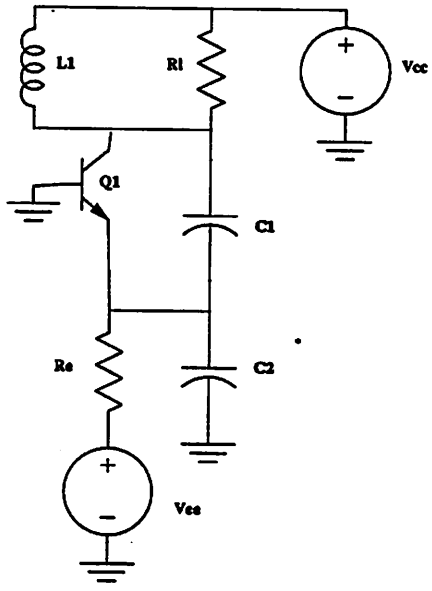
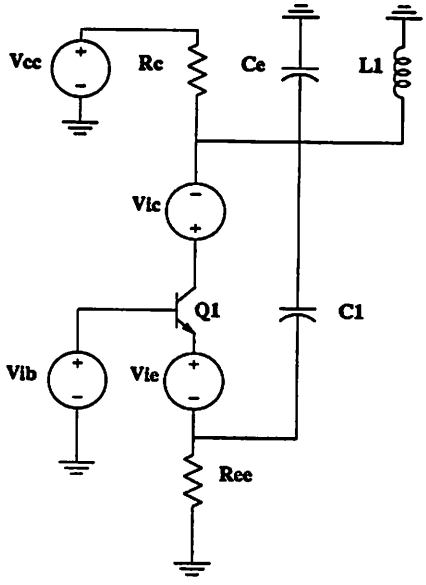


Figure B.7: Colpitts Oscillator with a Q of 100




```
Colpitts oscilator - High Frequency [Ngy88]
* osc.freq = 822meg output from v(5)=700mV
* power supply
vcc 3 0 5
vee 4 0 -5
ree 2 4 840
* active device
q1 5 6 7 4 nd230ew
* tank circuit
rc 3 1 1k
l1 1 0 9n
ce 1 0 3p
c1 1 2 3p
* ammeter
vic 1 5 0
vib 0 6 0
vie 2 7 0
* transistor model
.model nd230ew npn( xtb=1.5 xti=2.148 is=94.7e-18 bf=205 nf=0.978
+ vaf=22 ikf=142m ise=0.00 ne=1.50 br=62.0 nr=1.000 var=2.2 isc=0.00
+ nc=1.5 rb=41 irb=134u rbm=10.1 re=0.36 rc=8.9 eg=1.232 cje=300.0f
+ vje=0.995 mje=0.46 tf=10.0p xtf=1.00 itf=42.00m ptf=0.0 cjc=503f
+ vjc=0.42 mjc=0.22 xcjc=0.13 tr=400p cjs=240.0f vjs=0.65 mjs=0.31
+ fc=0.875 )
* Vout = v(5)
*Command for steady-state analysis:
*steady act auto 1.2195ns 50 .01 uic 10.0
*steady tran auto 1.2195ns 50 .01 uic 10.0
.end
```

Figure B.8: High Frequency Colpitts Oscillator

Wien Oscillator [Fan75]

```
rc 6 1 7k
re 4 0 470
rf 4 2 1k
q1 1 2 0 mod1
q2 3 1 4 mod1
.model mod1 npn(bf=80 rb=100 rc=10 cje=2pf
+cjc=2pf tf=1ns va=50)
r1 6 3 2.2k
r2 5 2 2.2k
c1 3 0 470pf ic=0.0v
c2 3 5 470pf ic=0.0v
vcc 6 0 7.5v
* period = 6.67us
*vout v(3, 0)
*command for steady-state analyis :
*steady act auto 6.67us 50 .01 uic 10.0
*steady tran auto 6.67us 50 .01 uic 10.0
.end
```

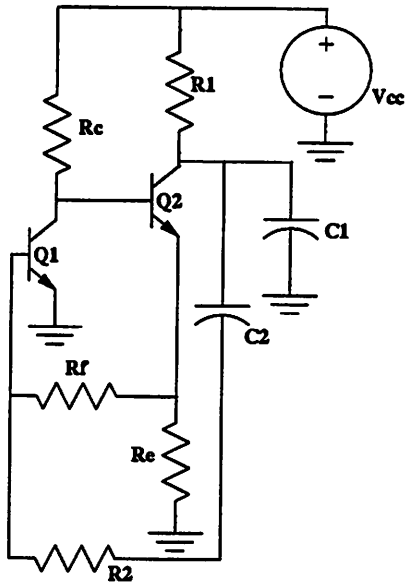


Figure B.9: Wien Oscillator

OP AMP Based Wien Bridge Oscillator [Ped88]

```
v1 1 0 0
r2 1 2 1k
c2 2 0 0.0159u ic=1v
r1 2 6 1k
c1 6 7 0.0159u ic=1v
eo 5 0 2 0 3.05
ro 5 7 1
dcl1 7 8 mod1
vbcl1 8 0 14
dcl2 0 9 mod1
vcl2 9 7 14
.model mod1 d is=1e-16
*period 100us
*plot v(7)
*command for steady-state analysis :
*steady act auto 100.0us 50 .01 uic 10.0
*steady tran auto 100.0us 50 .01 uic 10.0
.end
```

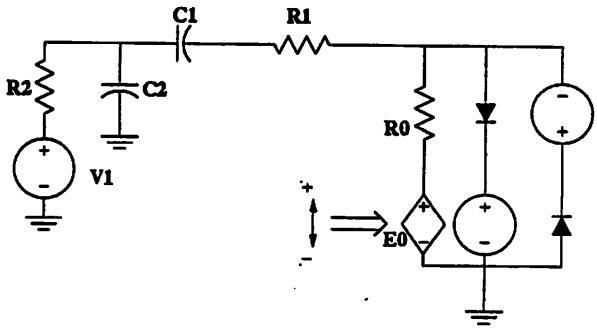


Figure B.10: OP AMP Based Wien Oscillator

```
Emitter Coupled Colpitts Oscillator [Fan75]
RB1 1 0 5K
RB2 6 0 5K
RE 3 4 4.7K
RC 5 2 5K
LT 5 2 10UH ic=1.5ma
C1 2 1 100pF ic=10V
C2 1 0 1000PF ic=0V
Q1 5 1 3 MOD
Q2 2 6 3 MOD
.model MOD NPN(RB=150 RC=20 CJE=2P
+CJC=2P VA=50)
VCC 5 0 10
VEE 4 0 -10
* Vout = v(5, 2)
*steady 1 5meghz 0 75
*commands for steady-state analysis:
*steady act auto .2us 50 .01 uic 10.0
*steady tran auto .2us 50 .01 uic 10.0
.end
```

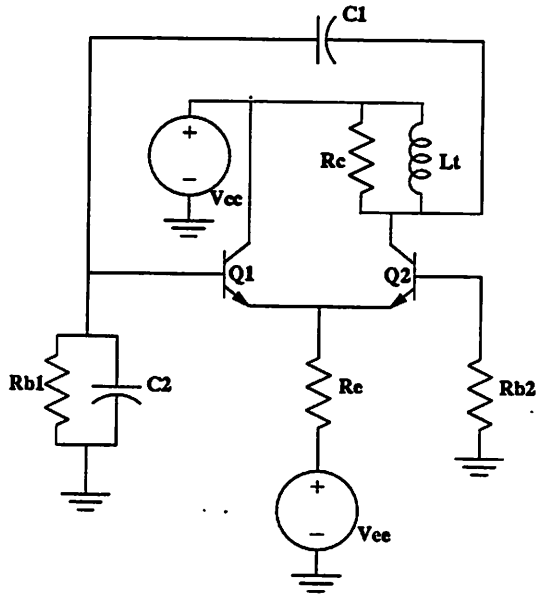


Figure B.11: Emitter-Coupled Colpitts Oscillator

```
E-C transformer feedback oscillator [Fan75]
rl 1 3 1k
cl 1 3 10nf ic=2.0v
l1 1 3 10uH ic=0.2ma
l2 4 0 1uH ic=0.1ma
kt l1 l2 0.98
re 2 5 3k
q1 3 4 2 mod1
q2 1 0 2 mod1
.model mod1 npn(bf=80 rb=100)
vcc 3 0 10
vee 5 0 -10
*plot 5 15 vout 1 0
*steady 1 500kHz 0 75
*command for steady-state analysis :
*steady act auto 2us 50 .01 uic 2.0
*steady tran auto 2us 50 .01 uic 2.0
.end
```

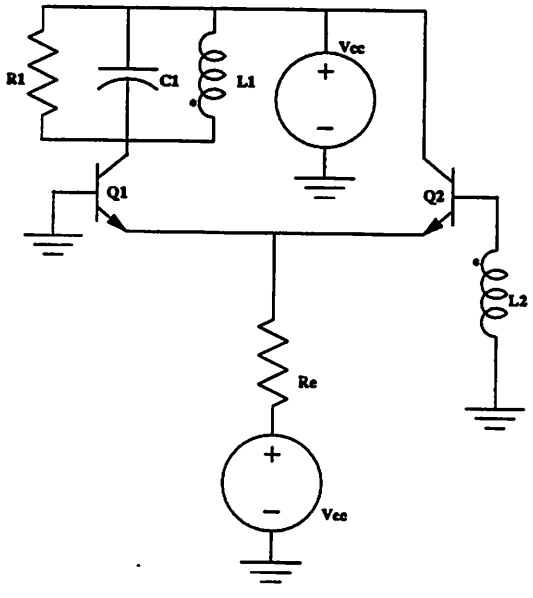


Figure B.12: Emitter-Coupled Transformer Feedback Oscillator

```
phase-shift oscillator [Fan75]
rc 5 1 3.3k
rb 1 2 470k
q1 1 2 0 mod1
.model mod1 npn(rb=100 rc=10 va=50)
c1 1 3 .047uF
r1 3 0 7k
c2 3 4 .047uF
r2 4 0 5k
c3 4 2 .047uF
vcc 5 0 10
*plot 0 10 vout 1 0 vb 2 0
*command for steady-state analysis :
*steady act auto 4.0ms 50 .01 uic 10.0
*steady tran auto 4.0ms 50 .01 uic 10.0
.end
```

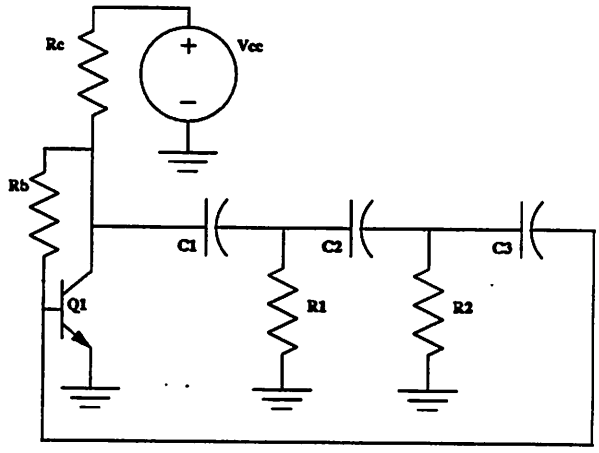


Figure B.13: Phase-Shift Oscillator

```
sony osc1 [Sony Corp.]
q1 6 2 4 mod1
q2 2 6 4 mod1
iee 4 0 0.2m
li 5 2 2.39u
ci 5 2 106p
ri 5 2 100k
.model mod1 npn bf=100 is=1e-16
+rb=50
vcc 5 0 5
vcc1 6 0 5
*period = 0.1us
*vout v(2)
*command for steady-state analysis :
*steady act auto .1us 50 .01 uic 10.0
*steady tran auto .1us 50 .01 uic 10.0
.end
```

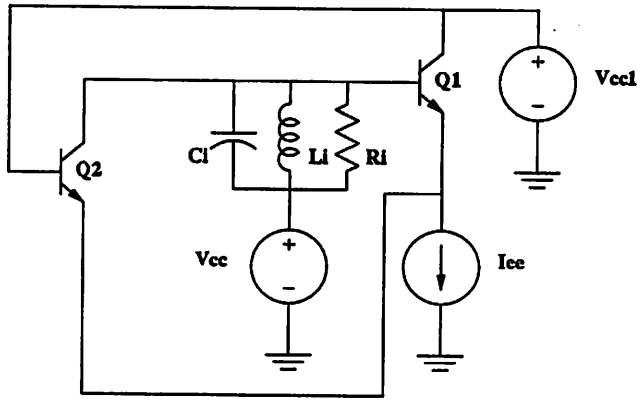


Figure B.14: LC-Tank Based Emitter-Coupled Oscillator

```
Bipolar relaxation-oscillator [Ped88]
.option reltol=1e-6
i1 0 1 5u
cb1 1 0 1pF
q1 3 1 0 mod1
rc1 5 3 10k
c1 3 4 100pF
i2 0 4 5u
cb2 4 0 1pF
q2 6 4 0 mod1
rc2 5 6 10k
c2 6 1 100pF
vcc 5 0 5
.model mod1 npn is=1e-16
*period 180us
*plot v(1) v(6)
*command for steady-state analysis :
*steady act auto 180us 50 .01 uic 10.0
*steady tran auto 180us 50 .01 uic 10.0
.end
```

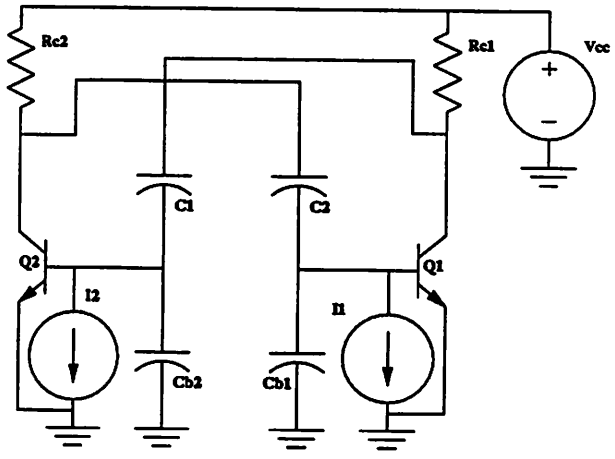


Figure B.15: Bipolar Relaxation-Oscillator

Appendix C

Shooting Methods : The Newton-Raphson Approach

C.1 Introduction

The Newton-Raphson iteration[1] algorithm can be used for solving the boundary-value problem. The boundary-value problem with the periodicity constraint can be expressed as follows:

$$\begin{aligned} y'(t) = \mathbf{f}(t, y(t)),\ \mathbf{g}(y(t+\mathcal{T}),\ y(t)) &= y(t+\mathcal{T}) - y(t) = 0, \\ \text{such that } t \in [t_1, t_2],\ y \in \mathcal{R}^N,\ &\text{and } \mathbf{f} : \mathcal{R}^N \rightarrow \mathcal{R}^N \\ \text{where } t_1 \text{ and } t_2 \text{ are distinct and } &\ \mathbf{g} : \mathcal{R}^N \rightarrow \mathcal{R}^N \end{aligned} \tag{C.1}$$

To solve the fixed-point problem $y(t_0+\mathcal{T}, y(t_0) = y\left(t_0^+\right)$ using Newton-Raphson, the following iterative equation is used:

$$y^{(i+1)}\left(t_0^+\right) = y^{(i)}\left(t_0^+\right) - \left[\mathbf{I} - \frac{\delta y^{(i)}\left(t_0^+ + \mathcal{T}\right)}{\delta y^{(i)}\left(t_0^+\right)}\right]^{-1} \left[y^{(i)}\left(t_0^+\right) - y^{(i)}\left(t_0^+ + \mathcal{T}\right)\right] \tag{C.2}$$

$\mathbf{f}$ is assumed to be differentiable with a Lipschitz continuous derivative. In Eqn. C.2, given $y^{(i)}\left(t_0^+\right)$, one can compute $y^{(i)}\left(t_0^+ + \mathcal{T}\right)$ using a program that can do conventional transient analysis. In the next section it is shown how $\left[\mathbf{I} - \frac{\delta y^{(i)}(t_0^+ + \mathcal{T})}{\delta y^{(i)}(t_0^+)}\right]$, the Jacobian, can be computed efficiently by means of sensitivity circuits under the assumption that the capacitor voltages and the inductor currents are independent. This implies that state equations are not necessary. It is shown in a later section that the sensitivity circuit approach is valid even when some dependencies exist among the capacitor voltages and inductor currents. The modification of this method for solving autonomous systems is presented in the section on autonomous systems. Finally, the implementation of the algorithm and the heuristics used are presented.

[1] The application of the NR method in this context is distinct from the conventional use of NR iterations by simulators.

C.2 Sensitivity Circuits

Sensitivity circuits[TCF75] are an efficient way of computing the Jacobian (see Eqn. C.2) required for the Newton-Raphson iteration. The circuit equations using the sparse-tableau formulation[HBG71] are:

$$\mathcal{A}\underline{i} = \underline{0} \quad \text{(KCL)} \tag{C.3}$$

$$\mathcal{A}\underline{e} = \underline{v} \quad \text{(KVL)} \tag{C.4}$$

and the branch equations

$$\underline{i}_r = \underline{f}_r(\underline{v}_r) \quad \text{(resistive)} \tag{C.5}$$

$$\underline{q}_c = \underline{f}_c(\underline{v}_c) \quad \text{(capacitive)} \tag{C.6}$$

$$\underline{\lambda}_l = \underline{f}_l(\underline{i}_l) \quad \text{(inductive)} \tag{C.7}$$

where $\dfrac{d\underline{q}_c}{dt} = \underline{i}_c;\ \underline{v}_c(0) = \underline{v}_{c0}$

and $\dfrac{d\underline{\lambda}_l}{dt} = \underline{v}_l;\ \underline{i}_l(0) = \underline{i}_{l0}$

the independent source equations:

$$C\underline{i}_s + D\underline{v}_s = \underline{G}(t) \tag{C.8}$$

If the Eqns. C.4 - C.8 are differentiated with respect to the initial value of one of the state variables, the following set of equations are obtained:

$$\mathcal{A}\frac{\delta \underline{i}}{\delta y_k(t_0)} = \underline{0} \quad \text{(KCL)} \tag{C.9}$$

$$\mathcal{A}^T\frac{\delta \underline{e}}{\delta y_k(t_0)} = \frac{\delta \underline{v}}{\delta y_k(t_0)} \quad \text{(KVL)} \tag{C.10}$$

$$\frac{\delta \underline{i}_r}{\delta y_k(t_0)} = \frac{\delta \underline{f}_r}{\delta \underline{v}_r}\Big|_{\underline{v}_r(t)=\underline{v}_r^{(i)}(t)} \frac{\delta \underline{v}_r}{\delta y_k(t_0)} \text{ (resistive)} \tag{C.11}$$

$$\frac{\delta \underline{q}_c}{\delta y_k(t_0)} = \frac{\delta \underline{f}_c}{\delta \underline{v}_c}\Big|_{\underline{v}_c(t)=\underline{v}_c^{(i)}(t)} \frac{\delta \underline{v}_c}{\delta y_k(t_0)} \text{ (capacitive)} \tag{C.12}$$

$$\frac{\delta \underline{\lambda}_l}{\delta y_k(t_0)} = \frac{\delta \underline{f}_l}{\delta \underline{i}_l}\Big|_{\underline{i}_l(t)=\underline{i}_l^{(i)}(t)} \frac{\delta \underline{i}_l}{\delta y_k(t_0)} \text{ (inductive)} \tag{C.13}$$

where

$$\frac{d}{dt}\left[\frac{\delta \underline{q}_c}{\delta y_k(t_0)}\right] = \frac{\delta \underline{i}_c}{\delta y_k(t_0)},$$

$$\frac{d}{dt}\left[\frac{\delta \underline{\lambda}_c}{\delta y_k(t_0)}\right] = \frac{\delta \underline{v}_l}{\delta y_k(t_0)},$$

and the initial conditions are given by,

$$\frac{\delta \underline{v}_c(t_0)}{\delta y_k(t_0)} = \begin{cases} \underline{e}_j, & \text{if } y_k \text{ is the voltage on the } j^{th} \text{ capacitor} \\ \underline{0}, & \text{if } y_k \text{ is an inductor current} \end{cases} \tag{C.14}$$

$$\frac{\delta \underline{i}_l(t_0)}{\delta y_k(t_0)} = \begin{cases} \underline{e}_j, & \text{if } y_k \text{ is the current in the } j^{th} \text{ inductor} \\ \underline{0}, & \text{if } y_k \text{ is a capacitor voltage} \end{cases}$$

where e_j is the unit vector,

$$C\frac{\delta \underline{i}_s}{\delta y_k(t_0)} + D\frac{\delta \underline{v}_s}{\delta y_k(t_0)} = 0 \qquad \text{(C.15)}$$

Eqns. C.10 - C.15, expressed in terms of the set of parameters $\left(\frac{\delta \underline{i}}{\delta y_k(t_0)}, \frac{\delta \underline{e}}{\delta y_k(t_0)}\right)$, are identical to the original sparse-tableau equations (Eqns. C.4 - C.8) expressed in terms of $(\underline{i}, \underline{e})$ if the original circuit is linear. In general the sensitivity circuit is a *linearized version* of the original circuit at the current operating point. There is a one-to-one correspondance between each variable and its partial derivative with respect to the chosen state variable. The only major difference between the two circuits is in the initial conditions for the differential equations as shown in Eqn. C.10 to Eqn. C.15. The initial voltage (current) of all capacitive (inductive) elements is zero except for the capacitor voltage (inductor current) with respect to which the partial derivatives are being computed, the initial condition for this capacitor (inductor) being $1volt$ ($1amp$). One sensitivity circuit can be generated for every state-variable in the original circuit. The solutions of each of these sensitivity circuits, with the unity initial condition mentioned above, at $t = t_0 + \mathcal{T}$ is the set of partial derivatives required to form the Jacobian, $\mathcal{J}(t_0 + \mathcal{T})$, used in the Newton-Raphson iteration.

Since each of the sensitivity circuits is just a linearized version of the original circuit at the present operating point in the original circuit, its circuit matrix has already been obtained in a LU-factored form while solving the original circuit. Thus, if each sensitivity circuit is solved in-step with the original circuit, the only computationally intensive steps in the solution of the sensitivity circuits at each time step are the forward and back substitutions. Therefore, an overhead of the order of $\mathcal{O}(N^3)$, where N is the number of states in the circuit, is added to the cost of carrying out a transient analysis on the original circuit because of the need to compute the sensitivities. There are other ways of computing the required Jacobian, notably using adjoint techniques[Dir71], but all of these are much more expensive than the sensitivity-circuit approach in terms of the overhead. The sensitivity matrix starts off as a diagonal matrix at $t = t_0$. With time, the off-diagonal entries get filled up. The ultimate structure of the matrix is not easily described. An algorithm for predicting the ultimate structure would be extremely helpful in reducing the overhead even further.

The end result is that the $y(t_0 + \mathcal{T})$ and $\mathcal{J}(t_0 + \mathcal{T})$, which are required for the Newton-Raphson iteration, are computed simultaneously and are both available when a transient analysis has been carried out for one period. Since a sufficiently general sparse-tableau formulation is used in the above development of the sensitivity circuits, it does not matter what formulation is actually used by the simulator to solve the circuit. The same conclusons are still valid. Because it is assumed that the circuit states are independent, it has to be shown that the presence of dependent states does not change the situation.

C.3 Circuits with Dependent States

In situations with dependencies, it is not possible to choose the initial conditions independently. The analysis for circuits with dependent states are considered in [TCF75]. It has been proven that the sensitivity approach described above leads to the same results as the state formulation approach and one need not worry about finding the dependencies among the state variables. This follows from the rigorous use of t_0^+ in the Newton-Raphson equation. Essentially, the circuit is allowed to relax for an infinitesimal amount of time so that the charge and flux redistribution can take place between the dependent storage devices.

C.4 Newton-Raphson for Autonomous Systems

Shooting methods can be modified to handle autonomous systems. The phase of the autonomous system is not of importance, and it is possible to assign some value to one of the state-variables and remove it from the set of unknowns and make the period of oscillation an unknown, giving us a solvable system with $\mathcal{N}$ unknowns and the same number of equations. In this section it is described how the Newton-Raphson method can be modified to do so.

When the substitution mentioned above is carried out, the following modified Newton-Raphson equation is obtained [AT72a]:

$$v^{(i+1)}\left(t_0^+\right) = v^{(i)}\left(t_0^+\right) - \left[\mathrm{I}' - \frac{\delta y^{(i)}\left(t_0^+ + \mathrm{p}^{(i)}\right)}{\delta v^{(i)}\left(t_0^+\right)}\right]^{-1} \left[y^{(i)}\left(t_0^+\right) - y^{(i)}\left(t_0^+ + \mathrm{p}^{(i)}\right)\right]$$

where, v is given by, $v \equiv \{y_1, y_2 \ldots y_{k-1}, \mathrm{p}, y_{k+1} \ldots y_{\mathcal{N}}\}$

p being the unknown that represents the period (C.16)

I' is the identity matrix with the k^{th} column replaced by a column consisting of all zeroes. The Jacobian in Eqn. C.16 requires that the k^{th} column consisting of the $\frac{\delta y^{(i)}(t_0^+ + \mathrm{p})}{\delta \mathrm{p}}$ be computed. This computation is not done as part of the solution of the sensitivity circuits. But again it can be shown that this computation is not expensive. For example, for a capacitor:

$$v_c^{(i)}\left(\mathrm{p} + t_0^+\right) - v_c^{(i)}\left(t_0^+\right) = \frac{1}{C}\int_{t_0^+}^{\mathrm{p}+t_0^+} i_c^{(i)} dt$$

$$\Rightarrow \frac{\delta v^{(i)}\left(\mathrm{p} + t_0^+\right)}{\delta \mathrm{p}} = \frac{1}{C} i_c^{(i)}\left(\mathrm{p} + t_0^+\right) \qquad \text{(C.17)}$$

Similarly for an inductor,

$$i_l^{(i)}\left(\mathrm{p} + t_0^+\right) - i_l^{(i)}\left(t_0^+\right) = \frac{1}{L}\int_{t_0^+}^{\mathrm{p}+t_0^+} v_l^{(i)} dt$$

$$\Rightarrow \frac{\delta i^{(i)}\left(\mathrm{p} + t_0^+\right)}{\delta \mathrm{p}} = \frac{1}{L} v_l^{(i)}\left(\mathrm{p} + t_0^+\right) \qquad \text{(C.18)}$$

Usually, the capacitor current and inductor voltage is available from the simulator and can be read from the data-structure. The k^{th} column is zero because the initial condition chosen obviously does not depend on the current iterate of the period. The $(i+1)^{th}$ iterate of the period is obtained by using Eqn. C.16. The following observation [AT72a] shows that the Jacobian as constructed in Eqn. C.16 is nonsingular and its inverse is available. The $n \times (n+1)$ matrix $[\Psi : \Delta]$ where $\Psi = \mathbf{I} - \Phi$, Φ is the state transition matrix and $\Delta = \frac{\delta y^{(i)}(t_0^+ + \mathbf{p}^{(i)})}{\delta w^{(i)}(t_0^+)}$, has n independent columns if and only if the column k is such that $\frac{\delta y_k^{(i)}(t_0^+ + \mathbf{p}^{(i)})}{\delta \mathbf{p}^{(i)}} \neq 0$.

According to this observation, the element of the vector of partial derivatives with respect to the period that will become a diagonal element in the new sensitivity matrix should not be zero if the new sensitivity matrix is to be nonsingular. It should be noted that the desirable properties of the Newton-Raphson iteration are not altered. In particular, the quadratic convergence when one is close to the solution is maintained. Also, it is important to choose the column to be substituted (the k^{th} column) such that the corresponding variable is in the range of the orbit. We choose $\frac{\delta y_k^{(i)}(t_0^+ + \mathbf{p}^{(i)})}{\delta \mathbf{p}^{(i)}} \neq 0$ to satisfy the observation by choosing a k such that $\left\| \frac{\delta y_k^{(i)}(t_0^+ + \mathbf{p}^{(i)})}{\delta \mathbf{p}^{(i)}} \right\|$ is maximum. The desirable side effect of this is that $y_k^{(i)}$ is assured of being in the range of the steady-state oscillation because when $\left\| \frac{\delta y_k^{(i)}(t)}{\delta t} \right\| = 0$, $y_k^{(i)}(t)$ is at its minimum or maximum values. This implies that if $\left\| \frac{\delta y_k^{(i)}(t)}{\delta t} \right\| \neq 0$, $y_k^{(i)}(t)$ lies between the extremes and is valid.

C.5 Implementation, Heuristics and Practical Considerations

The Newton-Raphson method has been implemented in SPICE3 [QNPS87]. The resulting program is called Sspice. The program is made sufficiently general that new steady-state algorithms can be added as required. The details are in Chapter 3. The pseudo code for the implemented algorithm is given in Figure C.1.

The main task is to solve the Newton-Raphson iterative equation (Eqn. C.2) to convergence with a minimal number of iterations. Each iteration in the Newton-Raphson iterative process is extremely expensive because it involves a transient analysis for one complete period of the circuit with the additional cost of computing the sensitivities. Also, the Newton-Raphson iteration may overshoot if the current guess happened to be far away from the solution. Finally, if more than one solution exists, the Newton-Raphson method may lead to the wrong solution. To reduce the probability of the iterative process going astray or the leading to the wrong solution, the following heuristics are used:

(1) Do a transient analysis in the beginning for as many cycles as required to reach a threshold distance from the solution. Computation of a measure of the distance from

```
int
NRiterate(circuit) {
    NRinit(); /* initialize the data-structures */
    exitLoop = FALSE;
START:
    while(TRUE) {
        if (Too Many Iterations) {
            return(ERROR);
        } else {
            if (doSensitivityUpdate) {
                Do a transient analysis for one cycle with sensitivity update;
            } else {
                Do a transient analysis for one cycle without sensitivity update;
            }
            Compute error;
            if (exitLoop) {
                if (converged) {
                    return(DONE);
                } else if (error ≥ THRESHOLD) {
                    exitLoop = FALSE;
                    doSensitivityUpdate = FALSE;
                } else {
                    break;
                }
            } else if (error < THRESHOLD) {
                exitLoop = TRUE;
                doSensitivityUpdate = TRUE;
            }
        }
    } /* end while loop */
    case Autonomous: Do NR for oscillators;
    case Nonautonomous : Do NR for driven circuits;
    goto START;
}
```

Figure C.1: Psuedo Code Showing the NR Algorithm

the solution is explained later.

(2) Dampen the Newton-Raphson iteration so that overshoot is prevented. Damping reduces the effect of the Jacobian on the iteration. The damping coefficient is computed dynamically as follows:

$$\text{Damping Coefficient } \eta = \max\ (0.1,\ (1-\alpha^m)) \tag{C.19}$$

It is necessary to restrict η to be a nonzero quantity to avoid dividing by zero while computing the damped Jacobian for an autonomous system. A restriction of η to a minimum value of 0.1 [Fan75], as in Eqn. C.19, has been empirically found to be small enough to leave the iterative process unaffected.

α is an error measure that gives an indication of the distance from the solution. For an autonomous system,

$$\alpha = \max\ \left(\left\| y^{(i)}\left(t_0 + \mathrm{p}^{(i)}\right) - y^{(i)}(t_0)\right\|\right). \tag{C.20}$$

For a nonautonomous system the period is a known quantity,

$$\alpha = \max\ \left(\left\| y^{(i)}(t_0 + \mathcal{T}) - y^{(i)}(t_0)\right\|\right). \tag{C.21}$$

Other error measures [CT73,GT82] have been proposed but were found during the course of the present work to be less effective than the simple measure above. The positive integer m is either a user defined quantity or some default value that was arrived at empirically. Since an overdamped system is also not desirable, the value of m in Eqn. C.19 has to be carefully chosen. It has been found experimentally that a value of $m = 10$ gives good results for most circuits. This method of damping implies that when the error is large, the Newton-Raphson iteration reduces to a fixed-point iteration, and when close to the solution it becomes an undamped Newton-Raphson iteration.

The damped Jacobian is computed in the following way:

$$\begin{gathered}\text{For a nonautonomous system, } \mathcal{J} = \mathbf{I} - \eta\,\Phi, \\ \text{where } \Phi \text{ is the state transition matrix defined earlier}\end{gathered} \tag{C.22}$$

$$\begin{gathered}\text{For an autonomous system, } \mathcal{J} = \mathbf{I}' - \eta\,\Phi_k, \\ \text{where } \mathbf{I}' \text{ has been defined earlier and } \Phi_k = \frac{\delta y^{(i)}(t_0^+ + \mathrm{p}^{(i)})}{\delta v^{(i)}(t_0^+)} \\ \text{with the } k^{th} \text{ diagonal element divided by } \eta^2\end{gathered} \tag{C.23}$$

(3) The error indicating the distance of the latest value of the state vector from the solution is checked before each transient analysis. If the error is greater than an acceptable threshold, no sensitivity computation is done during that transient analysis. This saves the cost of computing the Jacobian which would not have been

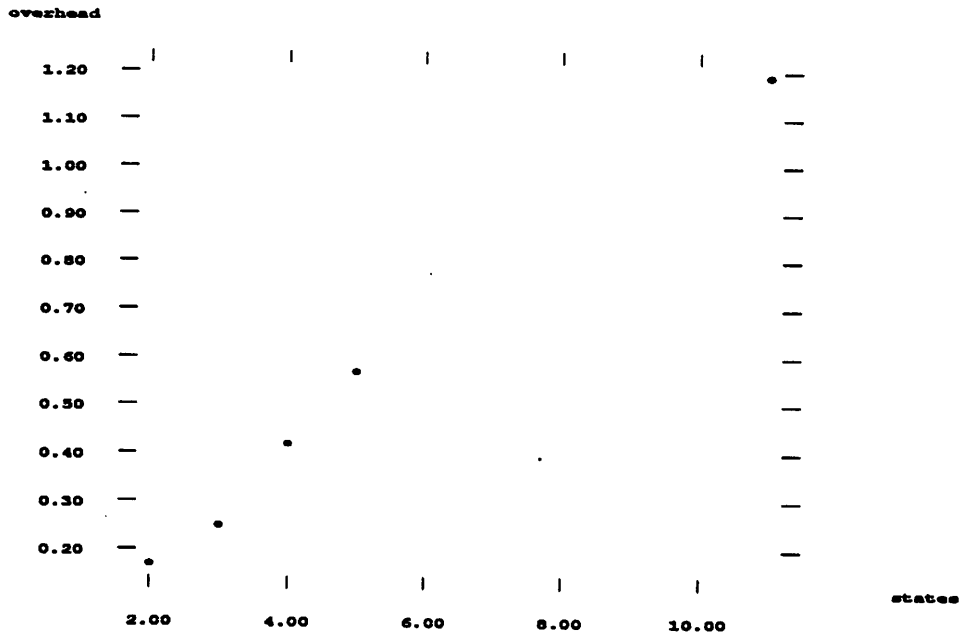


Figure C.2: Variation of Sensitivity Computation Overhead with Number of States

used anyway since after such a transient analysis a fixed-point iteration rather than a Newton-Raphson iteration is carried out.

(4) The Newton-Raphson algorithm for autonomous systems requires special heuristics to ensure that there is no overshoot while computing a new iterate for the time-period. During any iteration of the period, the change in the period is not allowed to exceed a certain fraction of its current value. It has been found that allowing a maximum change in the period of 0.1 [Fan75] times the current period prevents overshoot during the iteration and at the same time allows the iterations to proceed normally when there is no overshoot. It is necessary to protect against such overshoot because a wrong guess of the period could potentially lead the algorithm astray.

C.6 Cost of Sensitivity Computation and NR Iteration

It is possible to reduce the overall cost of the sensitivity computation required for the Newton-Raphson method. Programs like **NITSWIT** [KS88] require the computation of the sensitivity matrix an extremely large number of times. Even small reductions in the overhead can lead to significant improvements in the times in such situations. Figure C.2 shows the variation of the ratio of the sensitivity computation time to the total time taken for transient analysis with an increasing number of states.

One technique for reducing the cost is the well-known Samanskii's method [OR70] that uses the same Jacobian over many iterations. Thus, only one LU-factoring of the Jacobian needs to be done over many iterations. Each iteration just requires a loading of the right-hand side and a forward-back substitution. Though each iteration becomes cheaper, the number of iterations required increases. This method was suggested

and utilized in [CT73]. It was implemented in Sspice for experimentation. It was found that for many circuits it is difficult to predict when the Jacobian would have elements significantly different from the previous Jacobian. Frequently, either the wrong solutions are obtained or the number of iterations required was larger. In cases where a saving was obtained, it was not significant. Hence, Samanskii's method was not used.

Another approach to reducing the cost of sensitivity computation is the elimination of states that do not affect the ultimate solution but make the convergence to it slower and increase sensitivity computation overhead. Experiments have shown [GT82] that states due to transistor parasitics need not be considered. This result has been used in Sspice.

Future Possibilities

An approach to reducing the the cost of each Newton-Raphson iteration is to make the Jacobian more sparse. Each Newton-Raphson iteration is $\mathcal{O}$(cube of the number of states) in complexity because the Jacobian can, in general, be expected to be dense. A sparse Jacobian would reduce it to $\mathcal{O}$(super-linear in the number of states). The problem is to find elements of the Jacobian that can be predicted to have an insignificant effect on the solution of the system of equations that is computed during each iteration so that they can be zeroed out. As long as the algorithm used to find such elements is quadratic or less in complexity, it will lead to an improvement from $\mathcal{O}$(cube of the number of states) in complexity to at worst $\mathcal{O}$(square of the number of states) in complexity. This is a new approach and hasn't been tried yet. It is important to note that for a linear circuit, the Jacobian is time invariant. After the Jacobian has been computed for the first iteration, it need not be computed again. Since the Jacobian is time invariant, its structure does not change with time. It may be possible to reach a conclusion on the effects of each of the elements of the Jacobian on the solution of each Newton-Raphson iteration after the first iteration has been carried out. One could essentially zero those elements out and use the resulting more sparse matrix for all subsequent iterations. It is not clear how such an approach or another approach could be used for the case of nonlinear circuits. One possible technique for locating elements in the Jacobian that would not affect the final solution would be to first normalize the system of linear equations set up for the Newton-Raphson iteration with respect to the right hand side. The elements of the resulting matrix could then be compared with some norm of the matrix and if much smaller than the norm, could be zeroed out, making the matrix more sparse.

Appendix D

Shooting Methods : The Extrapolation Approach

D.1 Introduction

Extrapolation is another technique that can be used for the fixed-point iterations required for steady-state computation using shooting methods. The following three extrapolation algorithms were proposed in [Ske80]:

- scalar
- vector ϵ
- minimum polynomial

Only the minimum-polynomial method has been considered in the present work because it is usually the most efficient, requiring fewer periods to perform an extrapolation. In essence, extrapolation is the acceleration of the solution of the nonlinear equation $y = \mathcal{F}(y)$ based on a sequence $\{y_0, y_1, \ \cdots \ , y_{r-1}, \ , y_r\}$ which is generated by r iterations of the type $y_{n+1} = \mathcal{F}(y_n)$. In the following sections, it is shown that extrapolation is quadratic-convergent if certain conditions are satisfied, the algorithm is extended to autonomous systems and the implementation details are presented.

D.2 Foundation

The equation that extrapolation tries to solve is the same as Eqn. C.1 with the periodicity constraints. Extrapolation is based on the assumption that the sequence $\{y_j\}$ is generated by the finite-dimensional linear system of the form

$$y_{j+1} = \mathcal{A}\, y_j \ + \ b$$
$$\text{where} \quad \mathcal{A} \in \mathcal{R}^{N \times N} \quad \text{and} \quad b \in \mathcal{R}^N \tag{D.1}$$

$\mathcal{A}$ and b in Eqn. D.1 are useful for the formulation of the equation but are not actually required to be computed. If $\mathcal{F}(y)$ is linear, Eqn. D.1 models $\mathcal{F}(y)$ exactly and the fixed

point is found in just one iteration. More iterations are required if $\mathcal{F}(y)$ is nonlinear. Expanding $\mathcal{F}(y)$ about its fixed point z_0,

$$\begin{aligned}\mathcal{F}(z_0 + \delta y) &= \mathcal{F}(z_0) + \mathcal{J}_{\mathcal{F}}(z_0)\,\delta y + \mathcal{O}(\delta y^2) \\ &= \mathcal{F}(z_0) - \mathcal{J}_{\mathcal{F}}(z_0)z_0 + \mathcal{J}_{\mathcal{F}}(z_0)(z_0+\delta y) + \mathcal{O}(\delta y^2) \\ &= b + \mathcal{A}(z_0+\delta y) + \mathcal{O}(\delta y^2) \end{aligned} \tag{D.2}$$

Consider the sequence $\{\delta y_j\}$ generated by Eqn. D.1, where $\delta y_j = y_j - z_0$. Since $\delta y_j \in \mathcal{R}^N$, there are at most N linearly independent vectors in the sequence. If $\mathcal{A}$ is not of full rank because of constraints on the basic circuit equation (Eqn. C.1) or an ill-formed matrix, $\mathcal{A}$ has a rank of $r \leq N$ and a null space of $N - r$. At the most r of the vectors in the sequence $\{\delta y_j\}$ are linearly independent. If some of the eigenvalues of $\mathcal{A}$ have multiplicity greater than one, the number of linearly independent vectors may be less than r. As stated in the following theorem, the number of independent vectors is actually less than or equal to the degree of the minimum polynomial of $\mathcal{A}$. Equivalently, there is a number $q \leq r \leq N$ such that there are at most q linearly independent vectors in the sequence $\{\delta y_j\}$. This implies that there exists a set of $q+1$ nonzero coefficients $\{c_j\}$ such that

$$\sum_{j=0}^{q}\{c_j\}\delta y_j = \underline{0} \tag{D.3}$$

Since $\delta y_j = y_j - z_0$,

$$z_0 = \frac{\sum_{j=0}^{q} c_j y_j}{\sum_{j=0}^{q} c_j} \tag{D.4}$$

The coefficients $\{c_j\}$ are found by the following equations:

$$\text{Since, } y_{j+1} = \mathcal{A}y_j + b \text{ and, } z_0 = \mathcal{A}z_0 + b$$

$$\text{If } \Delta y_j = y_{j+1} - y_j \text{ we have, } \sum_{j=0}^{q} c_j \Delta y_j =$$

$$\sum_{j=0}^{q} c_j(\mathcal{A}-\mathrm{I})(y_j - z_0) = \sum_{j=0}^{q} c_j(\mathcal{A}-\mathrm{I})\delta y_j = \underline{0}$$

$$\text{Thus, if } \mathcal{Y} = \{\Delta y_0, \ldots \Delta y_{q-1}\},\ c = \{\Delta c_0, \ldots \Delta c_{q-1}\}^T$$

$$\text{and } c_q = -1, \text{ we have that } \mathcal{Y}c = \Delta y_q \tag{D.5}$$

If $q < N$, the problem is overdetermined and cannot be solved using LU factorization. Even if $q = N$, $\mathcal{Y}$ may be ill-conditioned because of small eigenvalues of $\mathcal{A}$. Thus, Eqn. D.5 is treated as a least-squares problem and is solved using QR factorization. QR factorization chooses a coefficient vector c such that $\epsilon = \|\mathcal{Y}c - \Delta y_q\|^2$ is minimized.

The value of q cannot be precomputed because the eigenvalues of $\mathcal{A}$ are not known. To compute q, the value of ϵ is monitored. The smaller of the minimum number of

iterations that give a value of ϵ less than a threshold and N is chosen as q. After every iteration, an error, giving a measure of the distance from the solution, is computed to check for convergence to the steady-state. This iterative process has been shown to be quadratically convergent under the conditions specified in [Ske80]. In the same paper, it has also been proven that efficiency of extrapolation is independent of the formulation of the network equations. It is shown that there is no need for choosing state-equations and no set of state-variables has to be identified.

D.3 Extension of Extrapolation to Autonomous Systems

The extension of the extrapolation algorithm to autonomous systems was suggested in [Ske80]. When the circuit is autonomous, the iteration is done on a new function obtained by replacing one of the variables by the unknown period. That this new function satisfies the properties required for convergence has been shown in [Ske80]. The underlying principle that allows the replacement of a state-variable by the period is the same as explained for the Newton-Raphson case.

Even though the application of the extrapolation method to autonomous systems is theoretically sound, practical difficulties were encountered during the implementation. These are explained in the following section.

D.4 Implementation, Heuristics and Practical Considerations

The extrapolation algorithm has been implemented into Sspice. The pseudo code for the implementation is shown in Figure D.1. The implementation is found to work well for nonautonomous circuits, but not for autonomous circuits.

It has been noted in [Ske80] and also observed in the present work that the lack of precision in numerical integration can prevent convergence if some of the transients are very slowly decaying. It is obvious that the change in the transient from $n\mathcal{T}$ to $(n+1)\mathcal{T}$ must be well above the level of integration errors for extrapolation to work properly. Effectively, the agreement between the discretized system and the continuous system must be close enough to represent the slowest changing transient in the continuous solution. In this connection, it has been observed that for some problems, forcing the error to be low by decreasing the maximum step size is found necessary to be able to reach convergence.

The motivation for the heuristics used for the extrapolation method is similar to that for the heuristics used in the Newton-Raphson method. As in the Newton-Raphson method, the purpose of the heuristics here is to reduce the effects of the start-up transients, to reduce the cost of the extrapolation method and to make sure that the solution computed is the desired solution. Some of the important heuristics that were tried, successfully and unsuccessfully, are described below.

```
int
EXTiterate(circuit) {
    EXTinit(); /* initialize the data-structures */
    Integrate for a user specified number of initial cycles;
    while(not(CONVERGED)) {
        Integrate for one cycle;
        Check convergence;
        if(CONVERGED) {
            return(DONE);
        } else {
            CONVERGED = FALSE;
START:
            DO as many cycles of transient analysis required for next EXTRAPOLATION;
            if(error > THRESHOLD for any cycle) {
                goto START;
            }
            doExtrapolation;
        }
    }
}
```

Figure D.1: Psuedo Code Showing the Extrapolation Algorithm

(1) The circuit is integrated for one cycle after every new initial state has been computed by extrapolation, and the state obtained after this cycle of integration is used as the initial state for the subsequent extrapolation. This results in a reduction of the effects of any error that may have occurred during the previous extrapolation on the next one.

(2) There is a constant monitoring of the difference between the state obtained after a cycle of integration and the initial state used for that cycle. Unless the error during all the cycles leading to the extrapolation is found to be less than a threshold, no extrapolation is done.

(3) The coefficients required for the extrapolation are computed using a variation of the QR algorithms based on Householder transformations [GL83]. As mentioned earlier in this appendix, the coefficients are computed so that the least-square error, $\|Ax - b\|^2$ for the linear system of equations $Ax = b$, is minimized. To prevent the domination of the error by a single equation, the equations are normalized with respect to the right hand side. This allows the error to be more uniformly distributed over the equations in the linear system.

(4) Each extrapolation is, in general, more expensive than a Newton-Raphson based iteration because $q + 2$ cycles of transient analysis are required to obtain the q δy_j vectors necessary for each extrapolation. In the Newton-Raphson method, each iteration involves only a single integration with some overhead. Thus extrapolation is usually comparable to the Newton-Raphson method only when the number of states is small. In the present work an attempt was made to reduce the overall cost of doing an extrapolation by sharing vectors computed for one extrapolation iteration with the next such iteration. It was found that the ability of the extrapolation method to

reach the solution was reduced when this heuristic was being used. An attempt was made to reduce the number of vectors required to be computed for each extrapolation iteration by using the method explained in an earlier section in this appendix. This was not found to be of much use because the number of vectors usually required using this method was equal to the number that would have been required anyway. It is not clear that such techniques to reduce the cost of each extrapolation are worthwhile because even if they are successful, it is unlikely for the extrapolation method to be comparable in the computational requirements to the Newton-Raphson method when the number of states is large.

(5) In case of autonomous circuits, a large number of heuristics were tried out but without success. As in the Newton-Raphson method for autonomous circuits, the change in the period computed during any iteration is not allowed to exceed a certain fraction of its current value. It has been found that allowing a maximum change in the period of 0.1 times the current period prevents overshoot during the iteration and at the same time allows the iterations to proceed normally when there is no overshoot. It is necessary to protect against such overshoot because a wrong guess of the period could potentially lead the algorithm astray.

(6) One way of incorporating the period into the extrapolation iteration for autonomous circuits is to measure the period at the end of every cycle of transient analysis. It was proposed in [Ske80] that the period be measured by first obtaining the maximum and minimum of some signal in the circuit during one of the initial cycles of transient analysis. A value between the maximum and minimum is chosen. The period of the circuit is then measured as the time between two time-points when the signal takes this value. This technique assumes that a relatively good estimate of the period is available in the beginning.

A better way to obtain the period is to compute a norm so that the period corresponds to the time-point when this norm is minimized. The norm is computed as the sum of the squares of the difference between the current values of the state variables and the values of the state-variables at the first time point in the current cycle of transient analysis. During the transient analysis this norm is checked after each time-point at which the circuit is solved once some fraction (e.g. 0.8) of the current estimate of the period has elapsed. The time-point at which the norm is the minimum is noted as are the values of the state-variables at that point. The difference between this time and the starting time for the cycle is then used as the period. There are two ways of using the period measured in every cycle. One is to use these values to do an unaccelerated fixed-point iteration in the period. Every time a new period is computed, it becomes the current estimate of the period. Another way of using the period computed during every cycle is to add it as an element in the vectors of state-variables obtained at the end of every cycle leading to an extrapolation iteration. This way a new iterate of the period can be obtained using extrapolation instead of just a fixed-point iteration. Both the methods were tried but were found to be unsuccessful in computing the steady-state of the circuit.

References

[AT72a] T. J. Aprille and T. N. Trick. A computer algorithm to determine the steady-state responsse of nonlinear oscillators. *IEEE Transactions on Circuit Theory*, CT-19(4):354–360, July 1972.

[AT72b] T. J. Aprille and T. N. Trick. Steady-state analysis of nonlinear circuits with periodic inputs. *Proc. IEEE*, 60:108–114, 1972.

[CL55] E. A. Coddington and N. Levinson. In *Theory of ordinary differential equations, New York: McGraw-Hill*, 1955.

[CNPS87] Wayne Christopher, A. R. Newton, D. O. Pederson, and A. Sangiovanni-Vincentelli. Nutmeg users guide. *The University of California at Berkeley*, 1987.

[CT73] F. R. Colon and T. N. Trick. Fast periodic steady-state analysis for large-signal electronic circuits. *IEEE J. Solid-State Circuits*, SC-8(4):260–269, August 1973.

[DC76] S. W. Director and K. W. Current. Optimization of forced nonlinear periodic circuits. *IEEE Transactions on Circuits and Systems*, CAS-23(6):329–334, June 1976.

[Dir71] S. W. Director. A method for quick determination of periodic steady-state in nonlinear circuits. In *Proc. 9th annu. Allerton Conf. Circuit and system theory, Univ. Illinois, Urbana-Champaign*, pages 131–139, October 1971.

[DM77] J. E. Dennis and J. J. More. Quasi-Newton methods, motivation and theory. *SIAM Rev.*, 19:46–89, Jan 1977.

[Fan75] S. P. Fan. SINC-S : A computer program for the steady-state analysis of transistor oscillators. *Ph.D. Dissertation, The University of California at Berkeley*, 1975.

[GJ88] Chris Guthrie and Beorn Johnson. *Personal communication.* 1988.

[GL83] G. H. Golub and C. F. Loan. *Matrix Computations, Johns Hopkins , Johns Hopkins University press.* 1983.

[GT82] F. B. Grosz and T. N. Trick. Some modifications to Newton's method for the determination of the steady-state response of nonlinear oscillatory circuits. *IEEE Transactions on Computer-Aided Design*, CAD-1(3):116–120, July 1982.

[Hal80] J. K. Hale. In *Ordinary Differential Equations, Krieger*, 1980.

[HBG71] G. D. Hachtel, R. K. Brayton, and F. G. Gustavson. The sparse tableau approach to network analysis and design. *IEEE Transactions on Circuit Theory*, CT-18(1):101–113, January 1971.

[Hil69] E. Hille. In *Lectures on ordinary differential equations, New York:Addision-Wesley*, 1969.

[KS86] Kenneth S. Kundert and Alberto Sangiovanni-Vincentelli. Simulation of nonlinear circuits in the frequency domain. *IEEE Transactions on Computer-Aided Design*, CAD-5(4):521–535, October 1986.

[KS88] Kenneth S. Kundert and Alberto Sangiovanni-Vincentelli. Finding the steady-state response of analog and microwave circuits. In *Proceedings of the Custom Integrated Circuits Conference*, 1988.

[KSS88] Kenneth S. Kundert, Alberto Sangiovanni-Vincentelli, and Tsutomu Sugawara. Techniques for finding the periodic steady-state response of circuits. In *Analog Methods for Circuit Analysis and Diagnosis, Marcel Dekker, Takeo Ozawa (editor)*, 1988.

[KWS88a] Kenneth S. Kundert, Jacob White, and Alberto Sangiovanni-Vincentelli. An envelope-following method for the efficient transient simulation of switching power and filter circuits. In *Proceedings of the International Conference on Computer-Aided Design*, 1988.

[KWS88b] Kenneth S. Kundert, Jacob White, and Alberto Sangiovanni-Vincentelli. A mixed frequency-time approach for finding the steady-state of clocked analog circuits. In *Proceedings of the Custom Integrated Circuits Conference*, 1988.

[Ngy88] Nhat Ngyuen. *Personal communication.* 1988.

[OR70] J. M. Ortega and W. C. Rheinboldt. In *Iterative solution of non-linear equations in several variables, New York : Academic*, 1970.

[Ped88] D. O. Pederson. *Class notes for EECS 142, The University of California at Berkeley.* 1988.

[Pet81] L. Petzold. An efficient numerical method for highly oscillatory differential equations. *SIAM J. Numer. Anal.*, 18(3), June 1981.

[QNPS87] T. Quarles, A. R. Newton, D. O. Pederson, and A. Sangiovanni-Vincentelli. SPICE3B1 users guide. *The University of California at Berkeley*, 1987.

[Qua89] Thomas Quarles. SPICE3. *Ph.D. Dissertation, The University of California at Berkeley*, 1989.

[Ske80] Stig Skelboe. Computation of the periodic steady-state response of nonlinear networks by extrapolation methods. *IEEE Transactions on Circuits and Systems*, CAS-27(3):161–175, March 1980.

[Ske82] Stig Skelboe. Conditions for quadratic convergence of quick periodic steady-s tate methods. *IEEE Transactions on Circuits and Systems*, CAS-29(4):234–239, April 1982.

[TCF75] T. N. Trick, F. R. Colon, and S. P. Fan. Computation of capacitor voltage and inductor current sensitivities with respect to initial conditions for the steady-state response of nonlinear periodic circuits. *IEEE Transactions on Circuits and Systems*, CAS-22(5):391–396, May 1975.