SIMULATING CIRCUITS

Circuit modeling programs have replaced "breadboards" as the building blocks of circuit design and development. With these programs, engineers can simulate a circuit, substitute components, vary parameters such as voltage and current in discrete time increments, and observe the changes at every point in the circuit. This capability represents an enormous savings in development time and effort.

The articles that follow present two approaches to the modeling of electrical circuits. The first, "Computer Circuit Simulation" by Wolfram Blume, is an introduction to the problems inherent in circuit modeling, such as linearizing nonlinear elements like diodes and transistors and solving the large simultaneous equations involved in circuit computations. The second article, "Analog Circuit Analysis" by David McNeill, describes the author's efforts in developing a set of circuit simulation programs for the Commodore 64, based on algorithms used in the SPICE simulation program of Dr. Laurence Nagel of the University of California.

—Charles Weston
Technical Editor

COMPUTER CIRCUIT SIMULATION

Circuit modeling programs make breadboards obsolete

BY WOLFRAM BLUME

COMPUTER SIMULATION is an important tool for circuit design engineers. The means for testing out ideas and checking a design has traditionally been the "breadboard" (a test circuit built on the workbench).

Computer simulation does the same job. It lets designers check out ideas while designing the circuit. Simulation programs also check the final design before it is released for production.

Wolfram Blume is president of MicroSim Corporation, which manufactures and markets the PSPICE *analog simulation program. He can be contacted at MicroSim Corp., 23175 La Cadena Dr., Laguna Hills, CA 92653.*

ANALOG CIRCUITS

The goal of a simulation program is to calculate the voltage at each node and the current through each device of the circuit. We will first want to calculate these values at the circuit's steady state (bias point). Using the steady-state data, we are then able to calculate the circuit voltages and currents as functions of time and frequency.

Most analog circuit simulators in use today are based on the program SPICE (Simulation Program for Integrated Circuit Engineering), which was developed at the University of California at Berkeley in the early 1970s. SPICE included several advances over earlier simulators, such as a "sparse matrix" data structure.

DATA STRUCTURE

Suppose we want to analyze the circuit shown in figure 1. Including ground, it has three nodes: 0, 1, and 2. The goal is to find the voltage at each node and the current through each device. For now, we will try to find the steady-state, or bias point, solution.

To analyze this circuit using a program we need a somewhat different approach than we would take in

(continued)

As a rule, any non-linear circuit has a linear equivalent, but only for one set of node voltages.

analyzing it by hand. Our approach is known as the nodal analysis method. We will set up a vector that lists the total current being pumped into each node by current sources. For all nodes that are not attached to a current source (such as node 2) this sum will be 0 (since electrical charge cannot be created or destroyed). For node 1 this will be −0.1, since the current source I_1 is driving 0.1 amp into node 1.

For node 0 this will be +0.1, since I_1 is pulling 0.1 amp out of node 0. This vector, I in figure 2, describes the input to the circuit. The current sources are the stimuli that drive the circuit. Note that the vector I has the same number of elements as the number of nodes in the circuit.

Next we need a vector that describes the response of the circuit to I. This vector, V in figure 2, is a list of the voltages at each node. V is the vector we want to find.

Finally, we need a description of how V relates to I. This is provided by the conductance matrix, G (figure 2). G describes how current flows in elements that are not current sources. In our example, the currents through resistors R1 and R2 are proportional to the voltages across them. The term of G at row i and column j specifies how much current will flow away from node i if the voltage at node j were increased by 1 volt and the voltages at all the other nodes were held constant. This may seem like a peculiar way of thinking about the circuit, but it has the great advantage that

$$I = G \times V$$

Note that each term in G is the sum of the contributions of the relevant circuit elements. So, knowing which devices in the circuit are connected to which nodes, our program can build G in a straightforward way. G has the same number of rows and the same number of columns as there are nodes in the circuit. Now that I is known and G is known, we can solve for V and get the set of node voltages we are after.

Even in this small example, it is obvious that most of the terms of G will be 0. This becomes even more true as the circuit gets larger. In typical circuits of 30 or 40 transistors, over 90 percent of the terms in G will be 0. G is therefore referred to as a "sparse matrix."

Circuits of this size also typically have 100 to 150 nodes. Since the number of elements in G is n^2 (n = number of nodes) it becomes clear that we must take advantage of all those 0 terms in G or its size will get out of hand. We do this by storing only the nonzero terms of G and a list of pointers to those terms. This process complicates the algorithms, but it is necessary in order to simulate realistic circuits.

Solving the Circuit Equations

Now that we have established a data structure, we can proceed to analyze the circuit. There are three levels of analysis:

1. Solving a linear circuit by Gaussian elimination or LU factorization.
2. Solving a nonlinear circuit by repeatedly linearizing the circuit and using (1).
3. Solving a time-varying circuit by repeatedly converting the circuit to a nonlinear, non-time-varying circuit and using (2).

Calculating the circuit's response with respect to frequency is a special case of (1).

Solving a Linear Circuit

Our example circuit (figure 1) is linear and does not vary with time. A linear circuit is composed of linear elements (those elements whose currents are proportional to the voltages within the circuit). For example, resistors are linear because the current through them is proportional to the voltage across their terminals. Ideal amplifiers are also linear because their output is proportional to their input.

Since our system is described by the matrix equation I = G × V, we can use the standard methods for solving systems of linear equations. The two most common methods are Gaussian elimination and LU factorization. Both methods take about the same amount of computer time given one G and one I, but LU factorization has the advantage of being much faster given one G and several different I's. SPICE uses LU factorization.

Here is a brief overview of LU factorization: The idea is to split G into two matrices, L and U, which multiplied will give G.

$$G = L \times U$$

Therefore,

$$I = L \times U \times V$$

(continued)

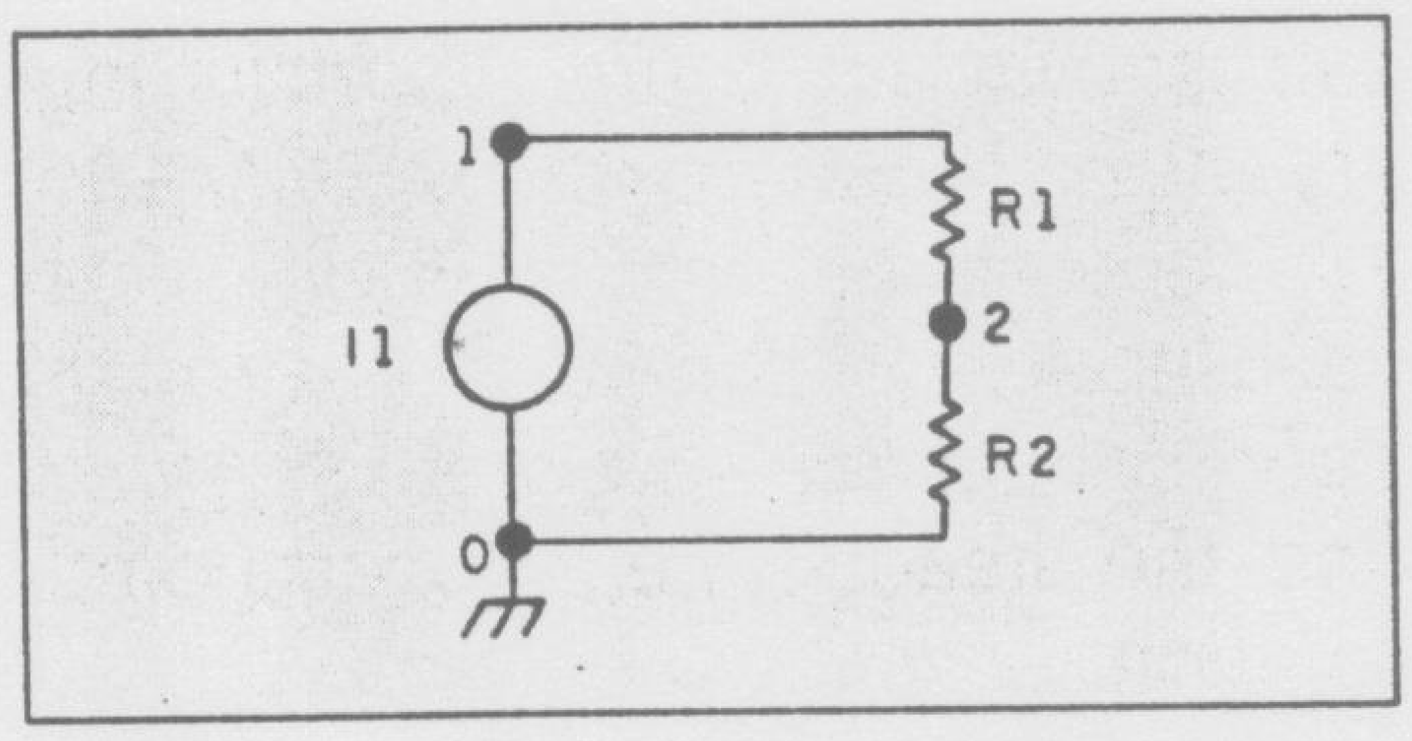


Figure 1: *A simple three-node circuit consisting of two resistors and a current source.*

$$\begin{pmatrix} -I_1 \\ +I_1 \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{1}{R2} & -\frac{1}{R2} & 0 \\ \frac{1}{R2} & \frac{1}{R2}+\frac{1}{R1} & -\frac{1}{R1} \\ 0 & -\frac{1}{R1} & \frac{1}{R1} \end{pmatrix} \begin{pmatrix} V_0 \\ V_1 \\ V_2 \end{pmatrix}$$

Figure 2: *The conductance matrix for the circuit shown in figure 1.*

L is lower triangular and U is upper triangular, as shown in figure 3.

An important feature of LU factorization is that for any G there is only one pair of L and U that have this structure. It is also true that the process of splitting G into L and U uses the major part of the computation time in this method.

Once L and U are found, their triangular structure (all 0s above or below the diagonal) allows the program to quickly calculate V from I. Since most of the time is spent finding L and U, we can find the solutions for several I's almost as quickly as for one.

Solving a Nonlinear Circuit

Now that we have a method to solve linear circuits, we can use it to find the solution for nonlinear circuits. The idea here is to use our last guess at V to linearize the circuit. Then, using our method for solving linear circuits, we can calculate a new guess for V. Hopefully, after a few iterations, successive guesses of V will converge to the correct answer.

How can we convert a nonlinear circuit into a linear one? By converting each nonlinear element into its linear equivalent. For instance, we can replace the diode in the circuit shown in figure 4 with a current source and a conductor.

The current through a diode is given by

$$I_d = I_s \times (e^{V/VT} - 1)$$

At a given voltage V across the diode we can replace the diode with a current source whose current is

$$I_{eq} = I_d - V \times G_{eq}$$

in parallel with a conductor whose conductance is

$$d(I_d)/dV = I_s/V_t$$

Note that the equivalent current and conductance change with the voltage V across the diode. This is true of diodes and of nonlinear elements in general. As a rule, any nonlinear circuit has a linear equivalent, but only for one set of node voltages. As soon as the node voltages change, we must "linearize" the circuit again.

Our algorithm for solving nonlinear circuits now looks like this:

```
1. Pick an initial guess for the node voltages V. Set $V_{old}$ = V.
2. Using the node voltages V, calculate the linear equivalent circuit and fill in the terms for the circuit matrix G and the current vector I.
3. Using the LU method, solve the circuit for a new set of node voltages V.
4. If V is close enough to $V_{old}$:
4a. Stop. We are done.
    Else:
4b. Set $V_{old}$ = V.
    Go gack to step 2.
```

Solving Time-Varying Circuits

So far the circuits we have considered have not varied with time. That is, neither the sources nor the other elements change with time. Capacitors have been treated as open circuits, and inductors have been treated as short circuits. This means that the solution, V, does not vary with time either. So far, we have been solving for the steady-state, or bias point, solution.

To accommodate time, we need to add time-varying sources and we need to handle capacitors and inductors. The first objective is easily done. We just allow the value of a source to be a function of time, such as a sine wave, pulse, square wave, etc.

The second objective is done in a way similar to the technique for solving a nonlinear circuit using the method for solving a linear one. We will replace each capacitor and inductor by equivalent circuit elements. Our algorithm for simulating the circuit now looks like this:

```
1. Use the nonlinear circuit method above to calculate the bias point of the circuit.
2. Set Time = 0. Set TimeStep to a small positive number.
3. Set all sources to their values at time = Time + TimeStep.
4. Using TimeStep and the past values of V, replace each capacitor and inductor by its equivalent circuit.
5. Use the nonlinear circuit method above to calculate V for time = Time + TimeStep. Use V from time = Time as the initial guess for time = Time + TimeStep.
6. Set Time = Time + TimeStep. Update TimeStep based on error-estimation formulas.
7. If Time => final time requested for simulation:
```

$$L = \begin{array}{cccccc} 1 & 0 & 0 & 0 & 0 & \ldots \\ x & 1 & 0 & 0 & 0 & \ldots \\ x & x & 1 & 0 & 0 & \ldots \\ x & x & x & 1 & 0 & \ldots \\ \cdot \\ \cdot \\ \cdot \end{array}$$

and

$$U = \begin{array}{cccccc} x & x & x & x & x & \ldots \\ 0 & x & x & x & x & \ldots \\ 0 & 0 & x & x & x & \ldots \\ 0 & 0 & 0 & x & x & \ldots \\ \cdot \\ \cdot \\ \cdot \end{array}$$

Figure 3: A *typical LU-factorization matrix. Note that L has 1s on the diagonal and 0s above the diagonal, and U has 0s below the diagonal.*

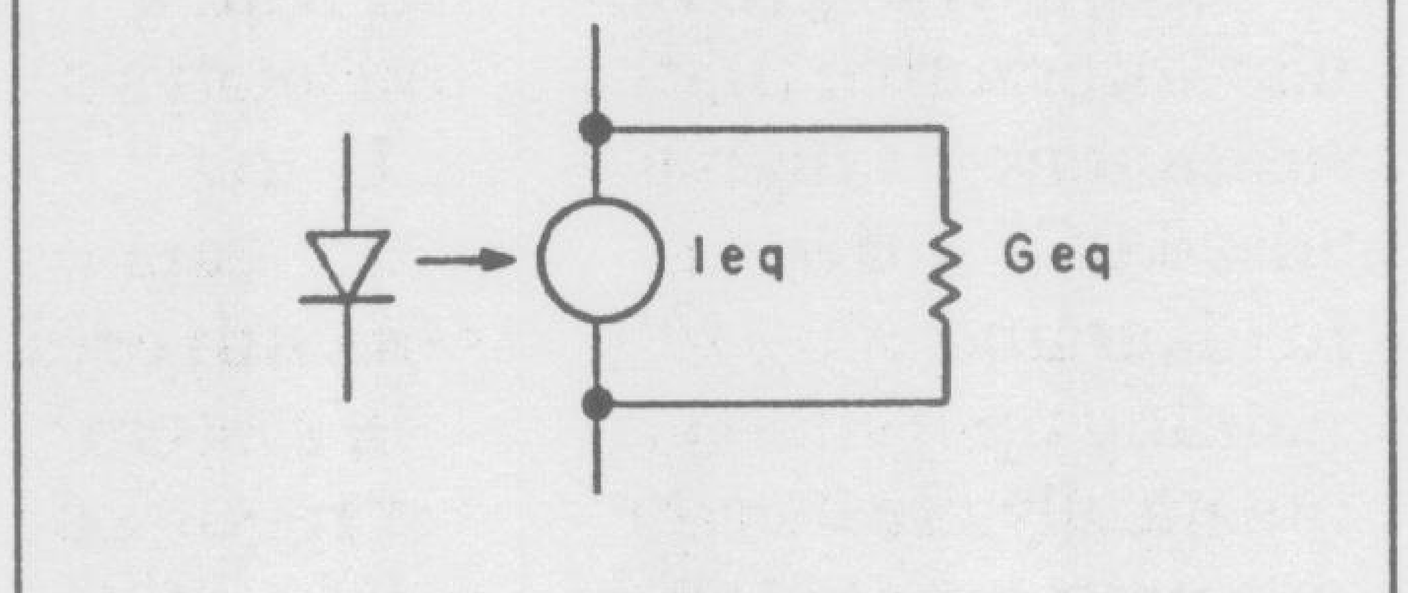


Figure 4: A *diode and its linear equivalent circuit, a current source and a conductor.*

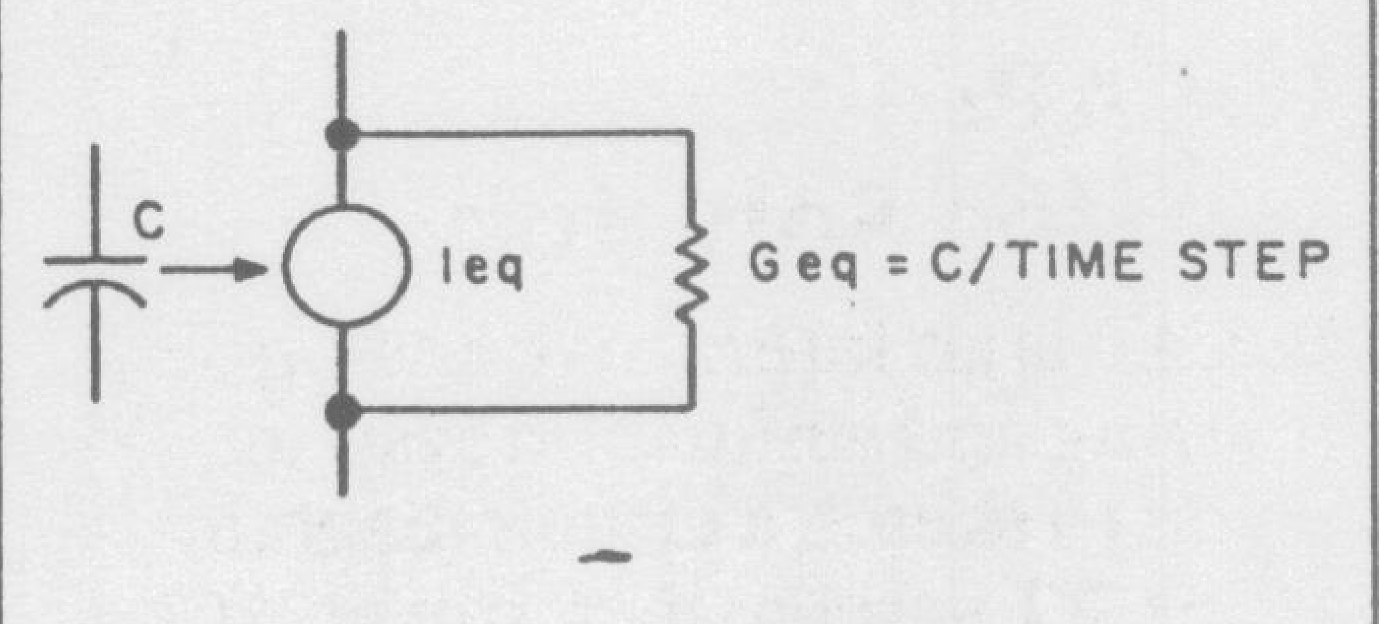


Figure 5: A *capacitor and its linear equivalent circuit, a conductance and a current source in parallel. The size of the time step is varied as simulation proceeds.*

```
7a. Stop. We are done.
    Else:
7b. Go to step 3.
```

Replacing the capacitors and inductors by equivalent circuit elements is possible because we know the size of the time step and the past values of each element's charge or current. For example, a capacitor is converted into a current source with a conductance in parallel as shown in figure 5.

The size of the time step is changed as the simulation proceeds. We could use a fixed time step, but this would waste computer time. We need to use a step small enough to accurately calculate those areas where things are changing rapidly (such small steps would be wasted during times when nothing was changing). By changing the step size based on the activity of the circuit, we can use a fine step size where it is needed and a coarser step size otherwise.

Calculating Frequency Response

The response of the circuit to different frequencies is calculated in a way very similar to solving a linear circuit. The difference is that the voltages, currents, and conductances now have real and imaginary parts instead of just real parts. The circuit algorithm now looks like this:

```
1. Use the nonlinear circuit method previously mentioned to calculate the bias point.
2. Set Frequency = beginning frequency.
3. Linearize the circuit using V calculated in step 1. Fill in terms, both real
   and imaginary parts, for I and G. Fill in the terms for capacitors and inductors
   as well. These are implemented as conductances, with imaginary values dependent
   upon Frequency.
4. Using the LU method, solve for V. V will have real and imaginary parts.
5. Set Frequency to the next frequency. If Frequency > ending frequency:
5a. Stop. We are done.
    Else:
5b. Go to step 3.
```

Displaying Results

Personal computers are well suited for displaying results. Because they are interactive and have graphics widely available, they are a natural for conveniently displaying the results of a simulation. Although SPICE does not provide for an interactive display of the results, this can easily be added.

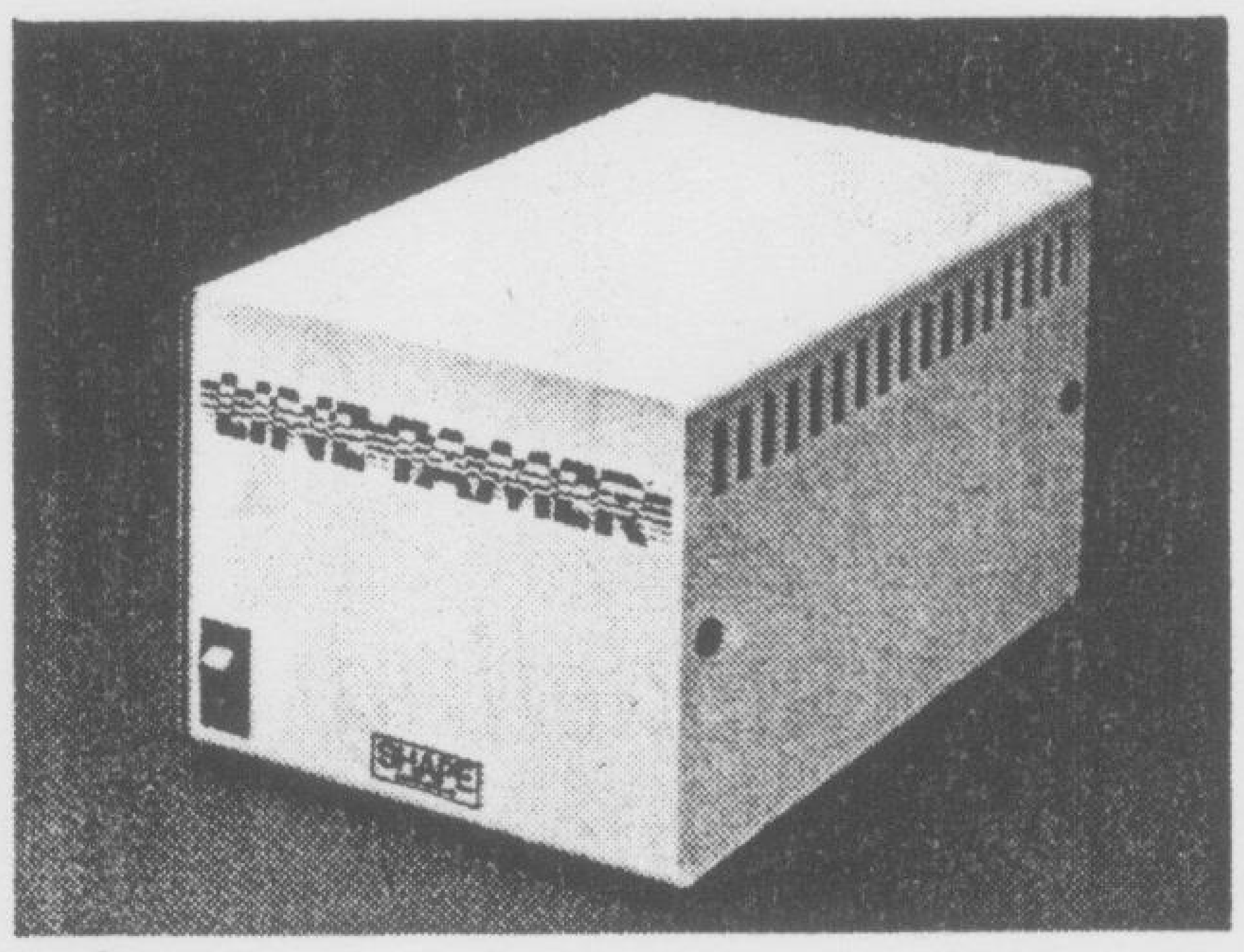
The approach I followed is to store all the node voltages and all the device currents at each step of the simulation (e.g., at each time or frequency). These are stored in a binary data file. The display software is a separate program that implements a virtual-memory scheme to allow the data file to be larger than the computer's RAM.

Besides drawing voltages and currents it is very convenient also to be able to draw expressions. For example, by multiplying a voltage and a

(continued)

Don't get burned by surge protection alone.

Now get total power protection . . . for as little as $139 . . . with the Personal Computer Line Tamer™ Power Conditioner.



Why risk any power trouble?

You will if you buy just a surge protector. Transients and spikes cause less than half of all power problems. Overvoltages, brownouts and just plain power noise can mess up your system just as badly–and they're much more frequent. Only Line Tamer's ferroresonant technology protects you from any conceivable power problem, short of a blackout.

Line Tamers have protected the big boys for years, on all the big mainframes and minis. Those guys would laugh at surge suppression alone.

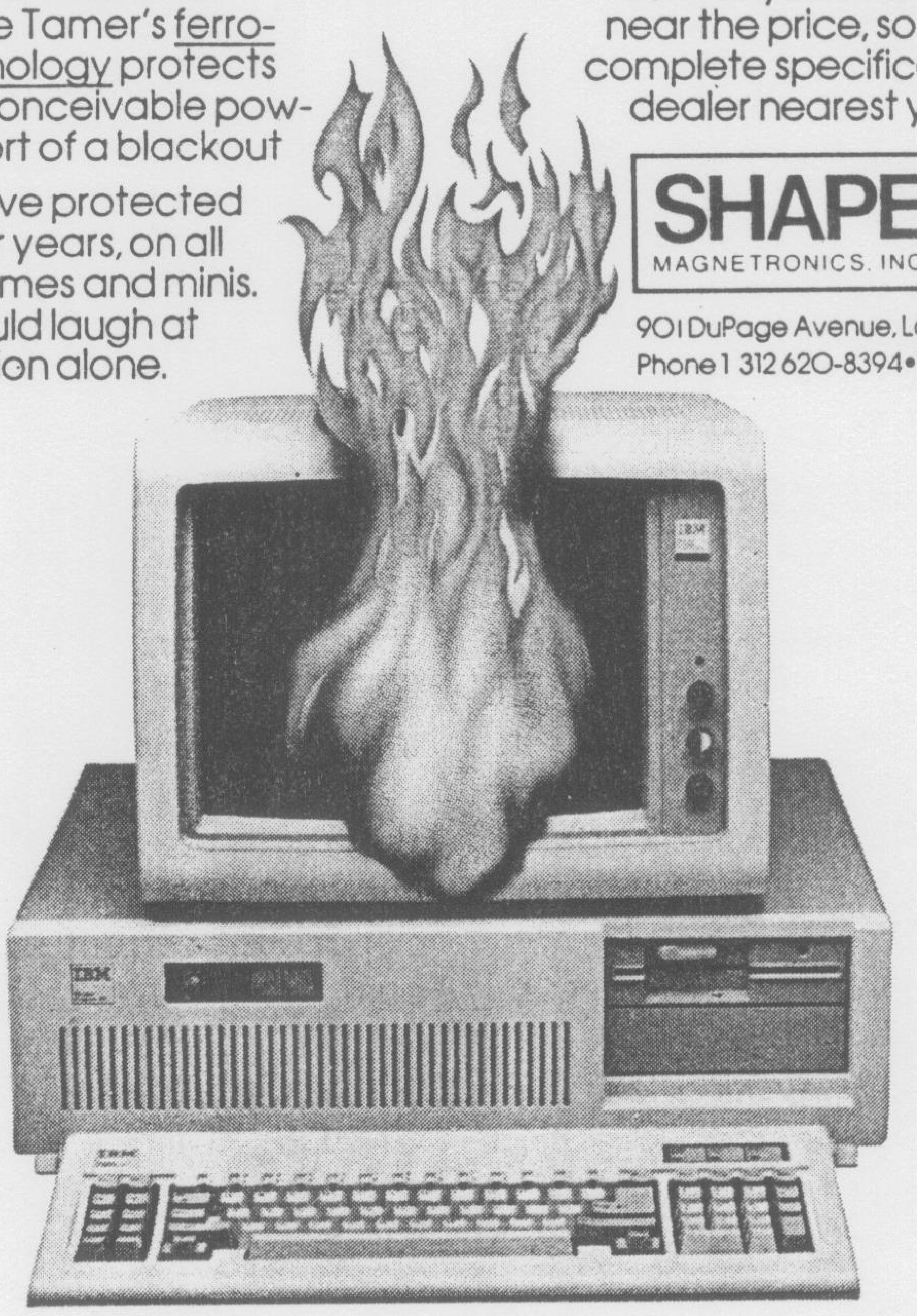
Choose from 150, 300, 450, or 600 VA models for the Line Tamer that's right for you.

Does it make sense to you to be half safe . . . especially when so few dollars are involved?

You won't find this total power protection from anyone else at anywhere near the price, so contact us for complete specifications and the dealer nearest you.

SHAPE MAGNETRONICS, INC — THE CLEAN POWER SOURCE

901 DuPage Avenue, Lombard, IL 60148
Phone 1 312 620-8394 • TWX 910-991-2352



current we can display instantaneous power.

COMMENTS

Analog circuit simulation makes heavy use of floating-point arithmetic. For accurate results, double-precision (64-bit) numbers must be used for the V, I, and G terms. As a result, it is necessary to have a floating-point coprocessor, such as the Intel 8087, in the computer. It improves run time by a factor of 15 in a typical circuit. In other words, without a floating-point coprocessor the simulation will be too slow to be useful.

Another requirement is RAM. Typical analog simulations are of circuits with 30 to 40 transistors. This requires about 100K bytes of RAM. Paging to and from disk would make the simulation intolerably slow. The program code itself takes up several hundred kilobytes of space, depending on how sophisticated the transistor models are and how many kinds of analyses are implemented. Therefore, to run a reasonable simulation, a machine with 512K bytes or more of RAM is needed.

Both these requirements are readily met by an IBM PC XT with 640K bytes and the 8087 coprocessor, which can simulate circuits of up to about 120 transistors. Calculating the time-varying response of a 30-transistor circuit takes about 12 minutes. Once the simulation is done, waveforms can be displayed almost instantly. ■

Editor's note: *If you are interested in the concepts and algorithms of circuit simulation, the author strongly recommends that you read* "SPICE: A Computer Program to Simulate Semiconductor Circuits" *by Laurence W. Nagel.*

You can obtain a copy of this document (Memorandum No. ERL-M520) by sending a check for $20 made out to: Regents of the University of California. Send check to Ms. Deborah Dunster, EECS Industrial Liaison Program, 457 Cory Hall, University of California, Berkeley, CA 94720.

The document is Dr. Nagel's Ph.D. thesis and contains an excellent discussion of the various algorithms in SPICE. He covers the material in this article in more detail and also provides discussions of alternative algorithms and the reasons for selecting those that finally went into SPICE.

ANALOG CIRCUIT ANALYSIS

An analog circuit modeling and simulation program for the Commodore 64

BY DAVID MCNEILL

OF THE THREE GOALS that I wanted to accomplish with this project, the main one was to produce a program like the circuit analysis program SPICE for my Commodore 64. My second goal was to better understand how transistor models worked and how to incorporate these models into the nonlinear algorithms used by SPICE to solve electronic circuits. My third goal was to fit the program into the memory space of the Commodore 64 and make it run at a reasonable speed.

My circuit analysis package is composed of three programs: a preprocessor and editor program, the AC analysis program, and the DC analysis program. To perform a circuit analysis, you first load the preprocessor and editor program used for entering the circuit description. Then, using a chaining technique, you can jump between all three programs to run the different analyses.

You can model 12 devices with the programs: resistors, inductors, capacitors, independent voltage sources, independent current sources, voltage-controlled current sources, diodes, bipolar transistors (*npn* and *pnp*), field-effect transistors (*n*-channel and *p*-channel), and operational amplifiers.

The programs allow you to sweep various parameters and print voltages, currents, impedances, or gains as the output variables. For example, in the model for the bipolar transistor, you can specify forward and reverse beta, Early voltage, reverse saturation current, junction capacitances, and transit times as parameters.

David McNeill is an electronics engineer who works for Tektronix in Beaverton, Oregon. He can be contacted at 11739 SW Beaverton-Hillsdale Hwy. #164, Beaverton, OR 97005.

PROGRAM DESIGN PHILOSOPHY

When I developed these programs, I decided to have as many model parameters as possible to better describe the device and produce more accurate data. Although this increased the complexity of the programs and tended to reduce the size of circuits that could be analyzed, I preferred to have programs that could better model actual transistors rather than analyze larger circuits with less accuracy. I felt that the limitation on running large circuits would be the speed of analysis rather than the amount of memory available.

The heart of the analysis programs is a routine that solves a set of linear equations. These equations arise from performing Kirchhoff's current equation at every node in the circuit. (Kirchhoff's current law states that the sum of all the instantaneous currents flowing toward a given node is equal to the sum of all the instantaneous currents flowing away from that node.)

For simple circuit elements such as resistors, voltage sources, and current sources, the formulation of the set of

(continued)

equations is straightforward, and since the equations in the set are all linear, they can be solved using standard numerical techniques such as Gaussian elimination or LU decomposition. The solution to this set of equations gives the voltage at each node in the circuit.

An element such as a diode, however, is not as easy to model because the relationship between the current through and the voltage across the diode is not linear but exponential. Because of this, we cannot directly assemble and solve a set of equations and arrive at the correct answer.

Modeling Nonlinear Elements

To solve this problem, two things must be done. One is to somehow modify the diode to make it appear linear. If it were linear, then we could formulate a set of equations that could be solved directly.

The second thing that must be done is to create an iterative process where we progressively change the linear diode that we created so that it will come closer and closer to the actual diode that exists in the circuit. As our model of the diode becomes better, the solutions to the set of equations will also get closer and closer to the actual answer. This second process is the program algorithm that controls the whole sequence of solving the set of equations, updating the linear diode, checking for convergence to a correct answer, and then repeating the procedure.

There are several crucial parts to this algorithm that can mean either success or total disaster. Due to the exponential nature of the diode equation, it is very easy to cause an overflow on the computer. Because of this, a good algorithm for clamping and limiting the exponential function was needed.

Figure 1 shows a simple resistor/diode circuit with a graph showing the load line for each element. The solution to this circuit is the point where the two lines intersect, which is at this time unknown. To start the analysis, we first make a guess at the voltage across the diode and pick this point on the diode curve. We can then linearize the diode by drawing a straight line through this point that is tangent to the diode curve, as shown in figure 2.

Notice that we now have two straight lines that intersect close to the actual intersection of the resistor and diode line. The point where the two straight lines intersect is found by the solution of two simultaneous equations. This solution then becomes an approximation of the actual answer. Although I've shown this graphically, the mathematical approach would be to perform a Taylor's series expansion on the diode equation (see figure 3).

Notice that in the resulting equation, shown in figure 3, we can identify two elements: a constant current source and a conductance. This equation is simply the equation of the straight line that was drawn through the point that we first guessed at on the diode curve.

The important point here is that we have been able to model the diode with two simple linear elements: a resistor and a current source. We now have a circuit that is composed entirely of linear elements. We can use the Gaussian elimination technique to calculate the point of intersection of the two straight lines. At this point we have an answer that is an approximation of the actual answer.

In the next phase we repeat the process and gradually improve the model of the diode so that the answers that are computed each time become closer and closer to the actual answer.

The Linearization Algorithm

Figure 4a is a flow diagram of the algorithm used to perform this iterative process for the diode. Figure 4b is a graphic description of the iterative process of the linearization algorithm. The first step is to check to see if the program has changed the temperature (such as in a temperature sweep analysis). If the temperature has been changed, the value of the reverse saturation current I_s must be recomputed, since it is temperature-dependent. Next, the program calculates and stores the value of the voltage across the diode that resulted from the last iteration.

The program then checks to make sure that if this value of voltage is used, the exponential function will not overflow. If overflow is possible, the program then clamps the voltage appropriately. Next, the program checks to see if this is the first iteration, and if so, guesses at a starting point for

(continued)

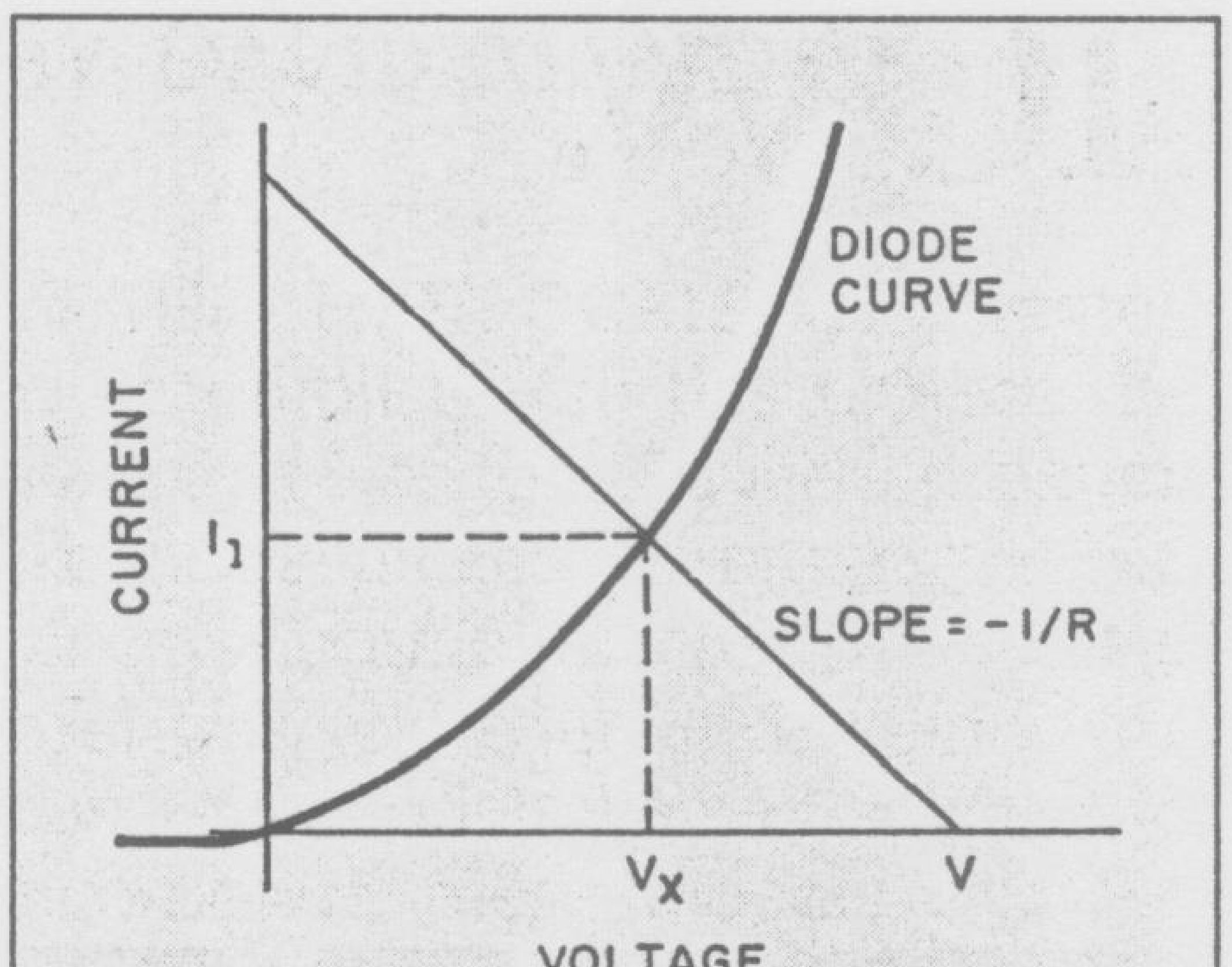


Figure 1: *This graph shows the load lines for the resistor and diode in the inset circuit.*

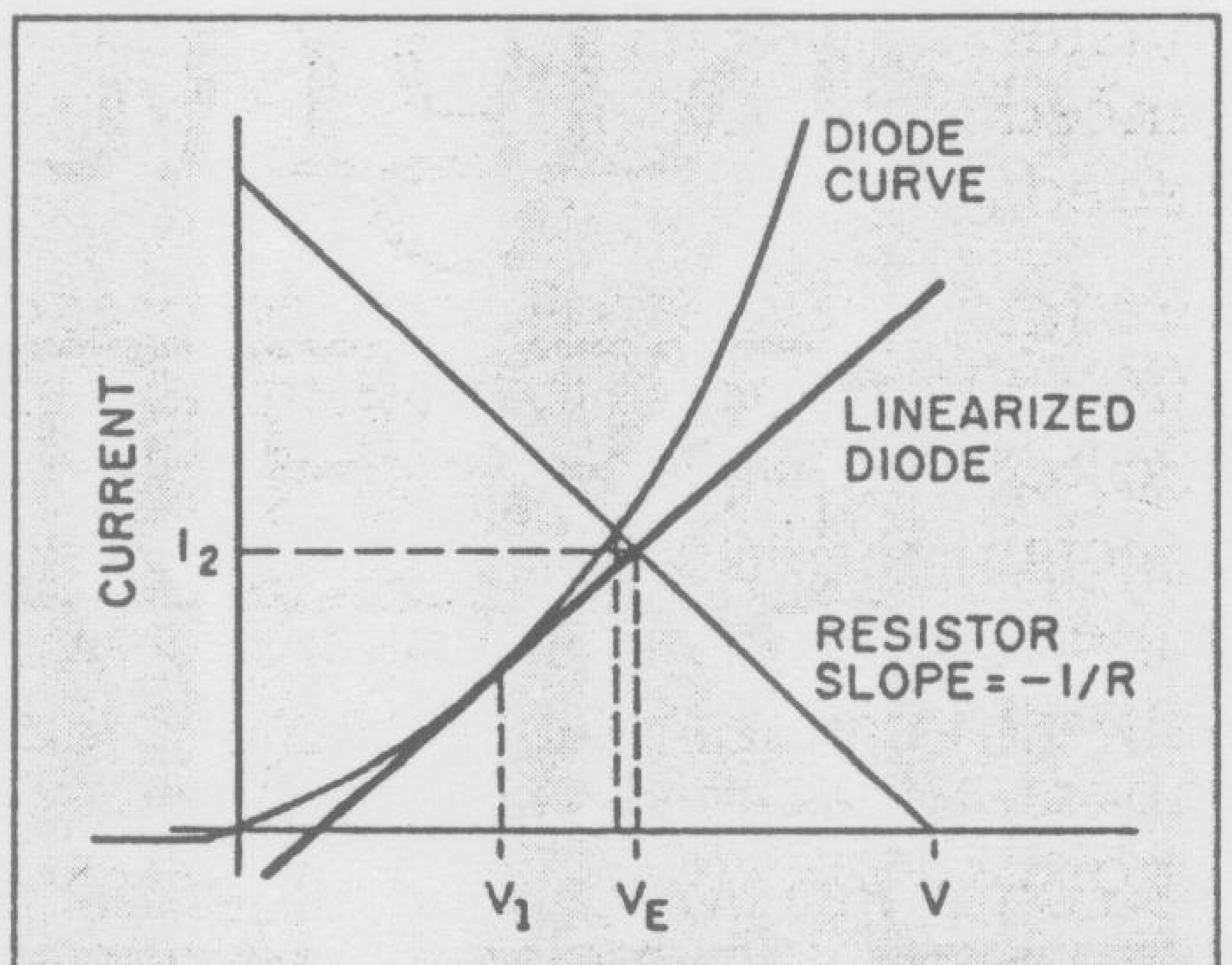


Figure 2: *This graph shows the resistor load line, the diode curve, and the linearized diode load line.*

By performing a Taylor's series expansion on the diode equation about an operating point, V_O, you are able to estimate the current in the diode for any voltage, V.

Diode equation: $I_{D_O} = I_S\,(e^{V_O/V_T} - 1)$

Taylor's series expansion equation: $f(x) = f(a) + f'(a)\,(x-a)$ (first two terms)

First compute the derivative:

$$f'(a) \rightarrow f'(V_O) = \frac{\partial I_{D_O}}{\partial V_O} = \frac{\partial}{\partial V_O}\,(I_S\,(e^{V_O/V_T} - 1)) = \frac{I_S}{V_T}\,e^{V_O/V_T}$$

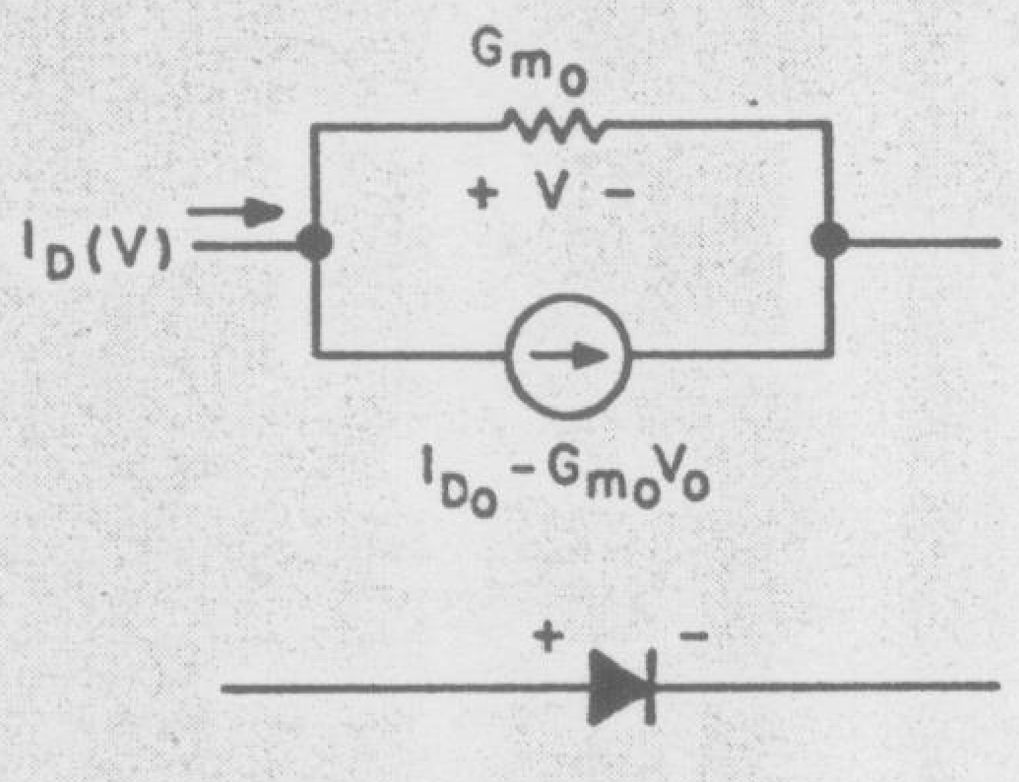
Since this is a conductance, let $\frac{I_S}{V_T}\,e^{V_O/V_T} = G_{M_O}$

Plugging into the expansion equation:

$$I_D(V) = I_S\,(e^{V_O/V_T} - 1) + G_{M_O}(V - V_O)$$

$$I_D(V) = I_{D_O} + G_{M_O}V - G_{M_O}V_O$$

This equation represents a current source and a conductance:



This model then becomes the approximation for the diode:

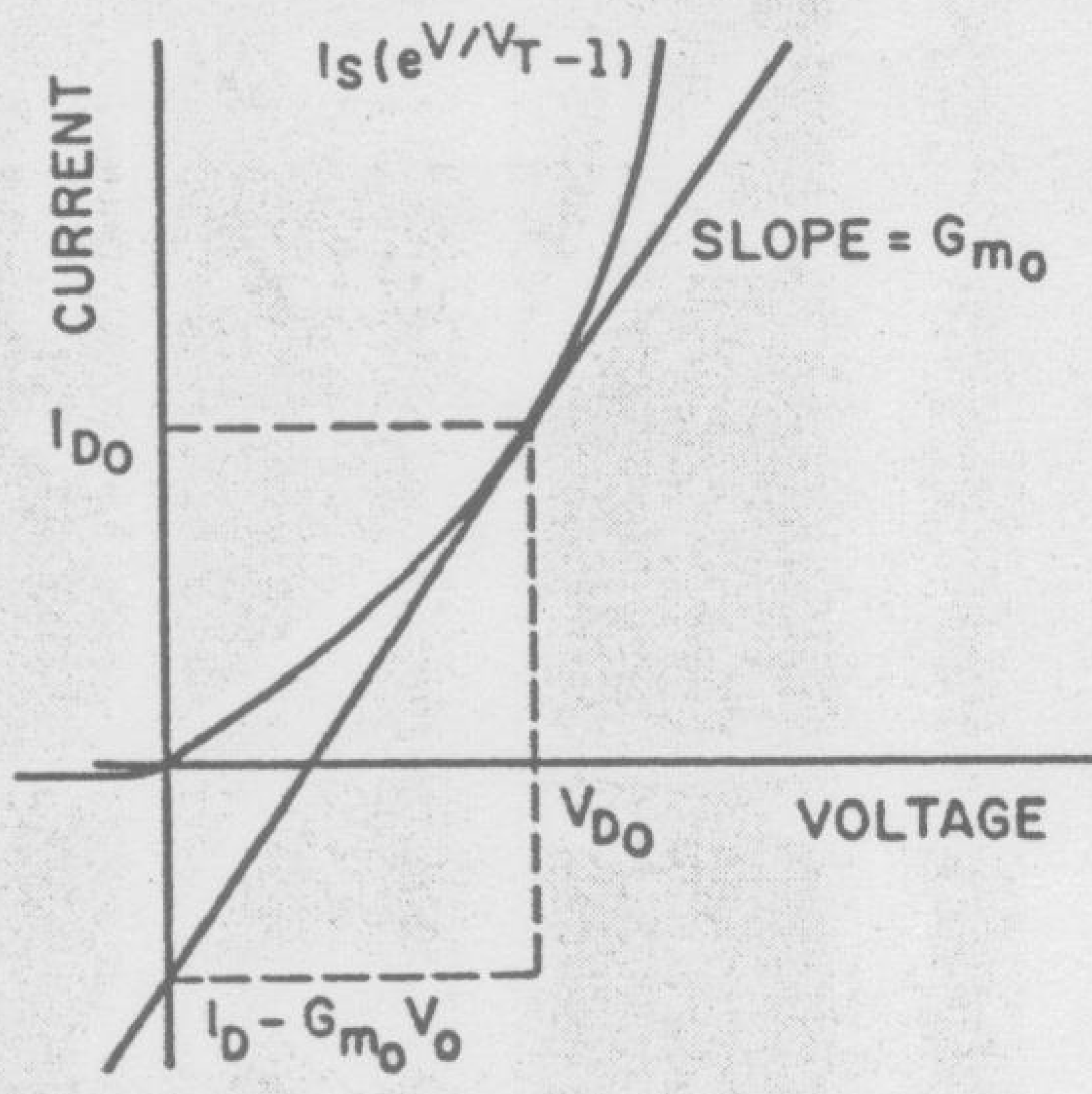


Figure 3: A *Taylor's series expansion of the diode equation. The graph now represents the diode as a constant current source and a conductance.*

the voltage across the diode. (The value used is the point on the exponential curve that has the minimum radius of curvature.) This point is called the critical voltage.

The next section is very important. Here the program tries to estimate the next voltage to use to linearize the diode. The goal here is to progressively move closer to the actual voltage that is across the diode. The process of determining the next voltage must be done carefully because it is easy to pick a voltage that will make this process diverge so that it can't converge to an answer.

In this algorithm there are two ways of selecting the next voltage to use. When we solve a set of equations with the Gaussian elimination routine, we have calculated a voltage value that is an estimate of the actual voltage across the diode. We can now use this calculated voltage as the starting point for the next guess. This first method of estimating the voltage is called "iterating on the voltage."

Another way of determining the starting point for the next iteration is to determine the value of current flowing in the diode at the point that the Gaussian routine indicated as the solution. Then, with this value of current, the corresponding voltage across the diode is calculated using the diode equation, and this voltage is then used as the starting point for the next guess. This method is called "iterating on the current."

This method of alternating between voltage and current is illustrated in figure 4b. The algorithm says that if the last voltage computed across the diode is greater than the critical voltage, iterate on the current; if the voltage across the diode is less than the critical voltage, then iterate on the voltage.

I found that using this algorithm helped tremendously in reducing the number of iterations it took to converge to the correct answer. It also helped prevent the iteration process from diverging, especially when I had a circuit where a transistor was in the off region of operation or was saturated.

At this point, I have determined a voltage across the diode either through assigning a value to it or by calculating a new value by iterating on the voltage or current. I now take this voltage and insert it into the equations for linearizing the diode and calculate the values for the conductance and current source for the diode model.

Now I use Kirchhoff's current law to create a set of linear, simultaneous equations based on these new values. The Gaussian elimination program is again used to solve the equations, and, based on the results, the entire process is repeated to converge on the correct answer.

MODELING TRANSISTORS

The algorithm described above is the procedure I used to convert a diode into linear components so that I could solve a circuit as if it contained only linear elements. Since diodes are integral parts of the bipolar and junction FET (field-effect transistor) models, I used this same algorithm for modeling both of these devices.

For the bipolar transistor, I started with the nonlinear hybrid-pi model as shown in figure 5. I chose this model because it simulates all four regions of transistor operation. Using this model, I performed the same steps that I did to linearize the diode. However, because the defining equations have dependent values and are functions of two variables, the linearization process is a little more difficult.

Figure 6 shows the linearized hybrid-pi model. This model results from performing a Taylor's series expansion on the equations defining each element of the nonlinear hybrid-pi model. The resulting expansion equation was then arranged so that its parts represented a controlled current source, a constant current source, and/or a conductance.

Next, the nonlinear components in the hybrid-pi model were replaced with these linear components. The iteration process is the same as with the diode. Once the terminal voltages are determined (from the last iteration), those voltages are inserted into the equations in figure 6, and the resulting values are used to compute (using Kirchhoff's current law) the

(continued)

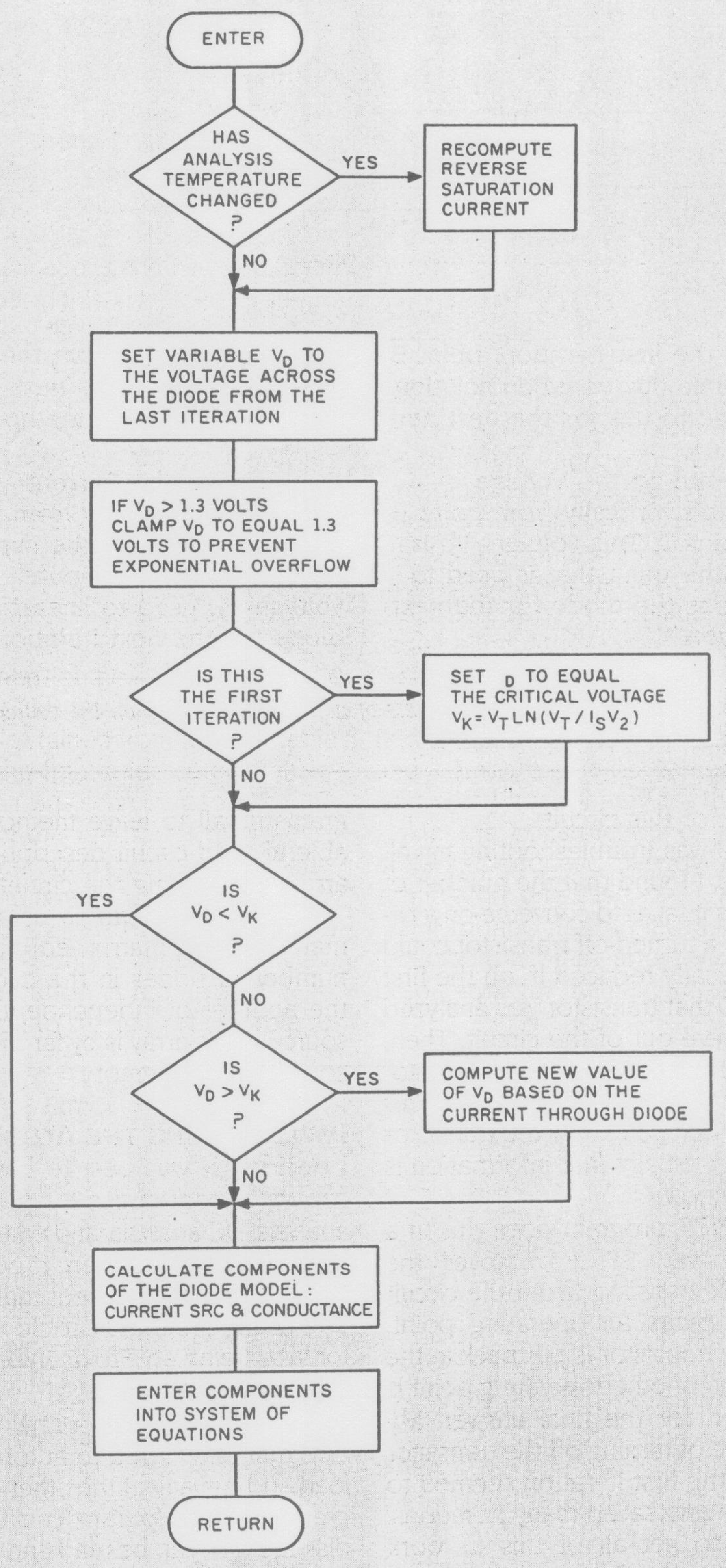
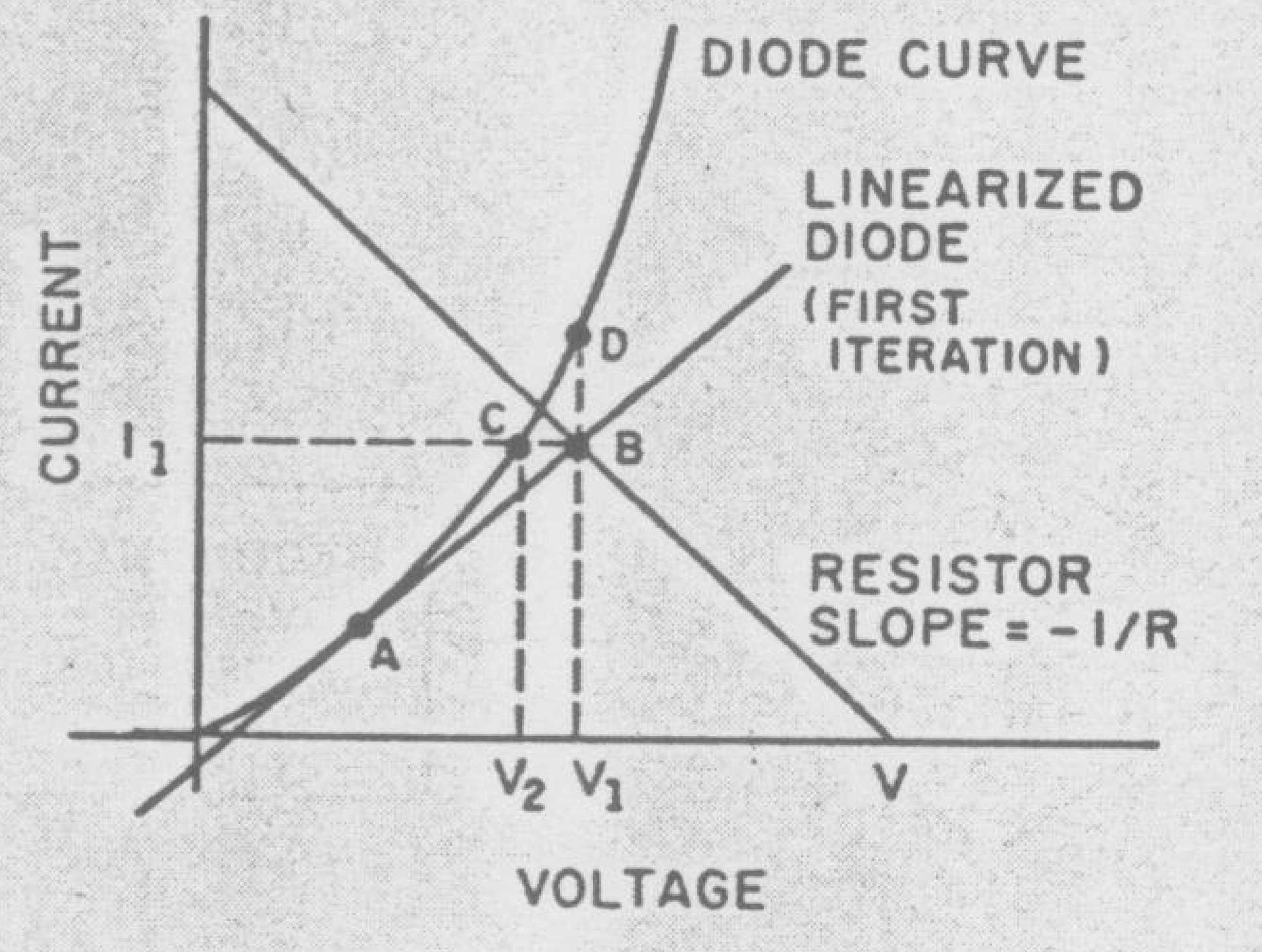

ENTER
HAS ANALYSIS TEMPERATURE CHANGED ?
YES
RECOMPUTE REVERSE SATURATION CURRENT
NO
SET VARIABLE V_D TO THE VOLTAGE ACROSS THE DIODE FROM THE LAST ITERATION
IF $V_D > 1.3$ VOLTS CLAMP V_D TO EQUAL 1.3 VOLTS TO PREVENT EXPONENTIAL OVERFLOW
IS THIS THE FIRST ITERATION ?
YES
SET $_D$ TO EQUAL THE CRITICAL VOLTAGE $V_K = V_T LN(V_T / I_S V_2)$
NO
IS $V_D < V_K$?
YES
NO
IS $V_D > V_K$?
YES
COMPUTE NEW VALUE OF V_D BASED ON THE CURRENT THROUGH DIODE
NO
CALCULATE COMPONENTS OF THE DIODE MODEL: CURRENT SRC & CONDUCTANCE
ENTER COMPONENTS INTO SYSTEM OF EQUATIONS
RETURN

Figure 4a: *The flow chart of the algorithm used to find the diode operating point.*



After the first iteration, point B is the solution given from the simultaneous equation solution routine. To compute the next voltage to use for the next iteration, use either of two methods:

I. Iterate on the Voltage
Proceed vertically from point B to point D. This voltage, V_1, is then the point that is used to linearize the diode for the next iteration.

II. Iterate on the Current
Proceed horizontally from point B to point C. From the current at this point, I_1, compute voltage V_2, used to linearize the diode for the next iteration.

Figure 4b: *This graph shows the process of iterating alternately on the voltage and current values from previous iterations.*

solution for this circuit.

When I was troubleshooting my algorithms, I found that the number of iterations it takes to converge on a circuit with a turned-off transistor could be drastically reduced if, on the first iteration, that transistor was analyzed as if it were out of the circuit. Then, on the second iteration, the transistor is analyzed as usual. Of course, this requires that you know the transistor is off, but usually that information is already known.

The SPICE program does this in a different way. SPICE removes the turned-off transistor from the circuit and computes an operating point. Then the transistor is put back in the circuit and another operating point is computed for the final answer. My technique of turning off the transistor for only the first iteration seemed to work fine and saved many iterations.

Trying to get all of this to work together on the Commodore 64 was the real challenge. When the Commodore 64 is powered up, only 38,911 bytes of memory are available to hold the program and variables. It was important to keep the size of the programs small to leave memory available to hold circuit descriptions and arrays for solving the circuit.

Each circuit requires an approximate square matrix equal to the number of nodes in the circuit plus the number of independent voltage sources. This array is by far the largest consumer of memory.

IMPLEMENTING THE ALGORITHMS

I decided it was best to have three separate programs to handle the AC analysis, DC analysis, and editing functions. I felt that even though this would be less convenient than having one larger program, I could make up for it by being able to analyze reasonable-size circuits.

Each program has a chaining feature that allows you to automatically load and run any of the other two programs. Each program can create a disk file that can be read and used by the other programs. For example, the preprocessor program will create a disk file that contains all the information about a circuit in a form that the AC and DC analysis routines can use.

One method I used to allow analyzing larger circuits was to dynamically allocate memory according to the size of the circuit. After loading one of the analysis programs, a fixed amount of memory remains to hold a circuit description and the array used to solve the circuit. If this memory were divided into two areas, one to hold the circuit description and the other to hold the main array, it is possible to have a circuit that would need more memory than allocated in one area but would use up only half of the other memory area.

In this case, there would be physically enough memory to work, but it is arranged such that it won't work. To prevent this, I dynamically allocate the memory for the circuit description and arrays as it is being loaded by the analysis program. A disadvantage to this is that I don't know the exact size of a circuit that can be analyzed, since it depends on the size of the circuit description. However, an advantage is that a wider variety of circuits can be analyzed, and available memory is used more efficiently.

Another technique that I used to help conserve memory was to use a utility program that went through the BASIC program and removed all remarks and would crunch up to 255 BASIC tokens on a single line. This made the program incapable of being edited, but it saved an average of 3.3K bytes of memory for each program and made a slight improvement in the speed of operation.

Despite all these attempts, the program still ran too slow. Analyzing the 741 op-amp circuit (26 nodes and 23 transistors) with the DC analysis program was taking about an hour and 20 minutes to compute the operating point. After doing some investigation, I found that the two bottlenecks were the Gaussian elimination routine and the routine used to clear the contents of the main array used by the Gaussian elimination routine. So my next major project was to write these routines in machine language.

Since I had the locations of the floating-point routines used by the operating system available, I fortunately did not have to create those routines myself. Within the Gaussian elimination routine, the subroutine that performed partial pivoting was the big bottleneck. The partial-pivoting routine is used during the elimination procedure to place, on the diagonal of the matrix, the largest number for a particular column. This ensures maximum accuracy during the elimination process. Placing the largest number on the diagonal involved checking the values on the column below the diagonal and then interchanging the elements of two rows when a number was found.

This interchanging procedure can take a long time. I tried using a permutation vector to act as an index for each row. Then, instead of interchanging every element, only the indexes to the rows are interchanged. This is a good idea, but when implemented in BASIC there was not a very significant improvement in speed for the size of arrays that I was using. However, when I wrote the whole routine in machine language, I did use a permutation vector since it was simple to implement.

I think one of the major improvements I was able to make in the performance of the Gaussian elimination routine was in the speed of accessing the elements of the array. The Commodore 64 operating system stores arrays in column-major form. So to access, for instance, the element in column 6 and row 3, the formula used to calculate the address of that element would be

ADDRESS = STARTING ADDRESS + (COLUMN# * (TOTAL # ROWS) + ROW#) * 5

The number 5 comes from the fact that each floating-point number uses 5 bytes. This formula indicates that every time you want to access any element of an array, you have to perform two multiplications and two additions. To get around this, I decided to set up two tables to assist in accessing elements. The first table consists of the addresses of the zeroth element of each column. The second table is a list of constants that are multiples of 5, starting with 0.

Then, to access the element at column 6 and row 3, I would index into the sixth entry of the first table to get the address of the zeroth element of column 6. Next, I would index into the third element of the second table and obtain 15. This would be added to the number I obtained from the first table to yield the final address of the element that I was searching for. The advantage of this method is that the indexing operation can be done by using the indirect-indexing commands of the 6502 instruction set, and beyond that only a simple 16-bit addition operation is required.

Of course to set up the first table requires a loop of multiplications to be performed. To get around this, I made the following improvement. The first time an array is solved using the Gaussian elimination routine, I find the starting address for the array and save its value. Then, the first table described above is created and the actual elimination process begins.

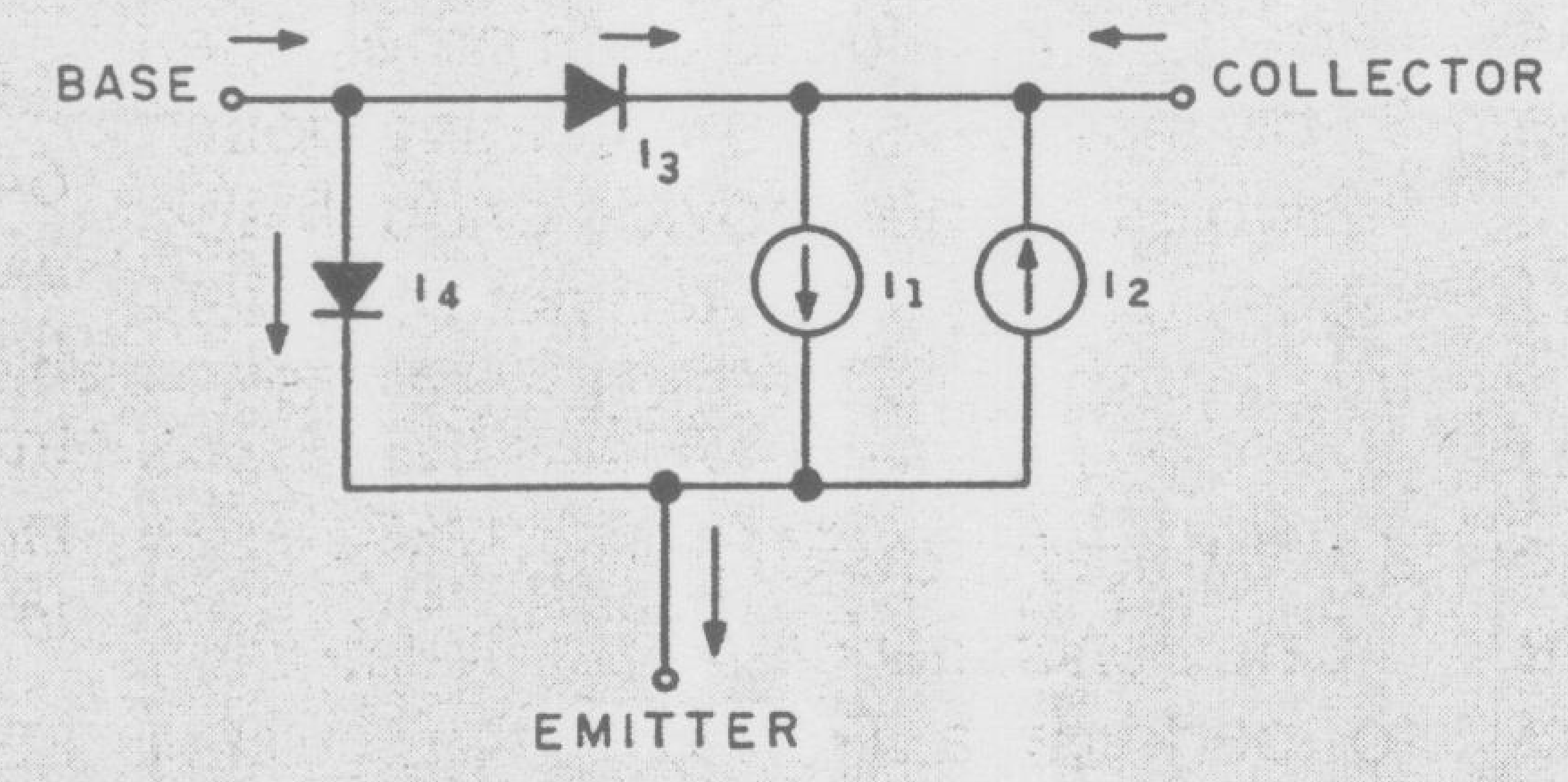
The next time the array is to be solved (as on the second iteration), I find the starting address of the array and compare it with the previous starting address. If they are the same, I skip the construction of the first table and proceed directly with the elimination process.

The routine that replaces the contents of the main array with 0s in preparation for the next iteration can

(continued)

By rewriting the whole BASIC *indexing routine in machine language* I *was able to increase its speed by a factor of* 50.

Nonlinear Hybrid-Pi Model with Early Voltage Effect



$$I_1 = I_S\,(e^{qV_{BE}/KT} - 1)\,(1 - \frac{V_{BC}}{V_A})$$

$$I_2 = I_S\,(e^{qV_{BC}/KT} - 1)\,(1 - \frac{V_{BC}}{V_A})$$

$$I_3 = I_S\,(e^{qV_{BC}/KT} - 1)\,(1/B_R)$$

$$I_4 = I_S\,(e^{qV_{BE}/KT} - 1)\,(1/B_F)$$

where:

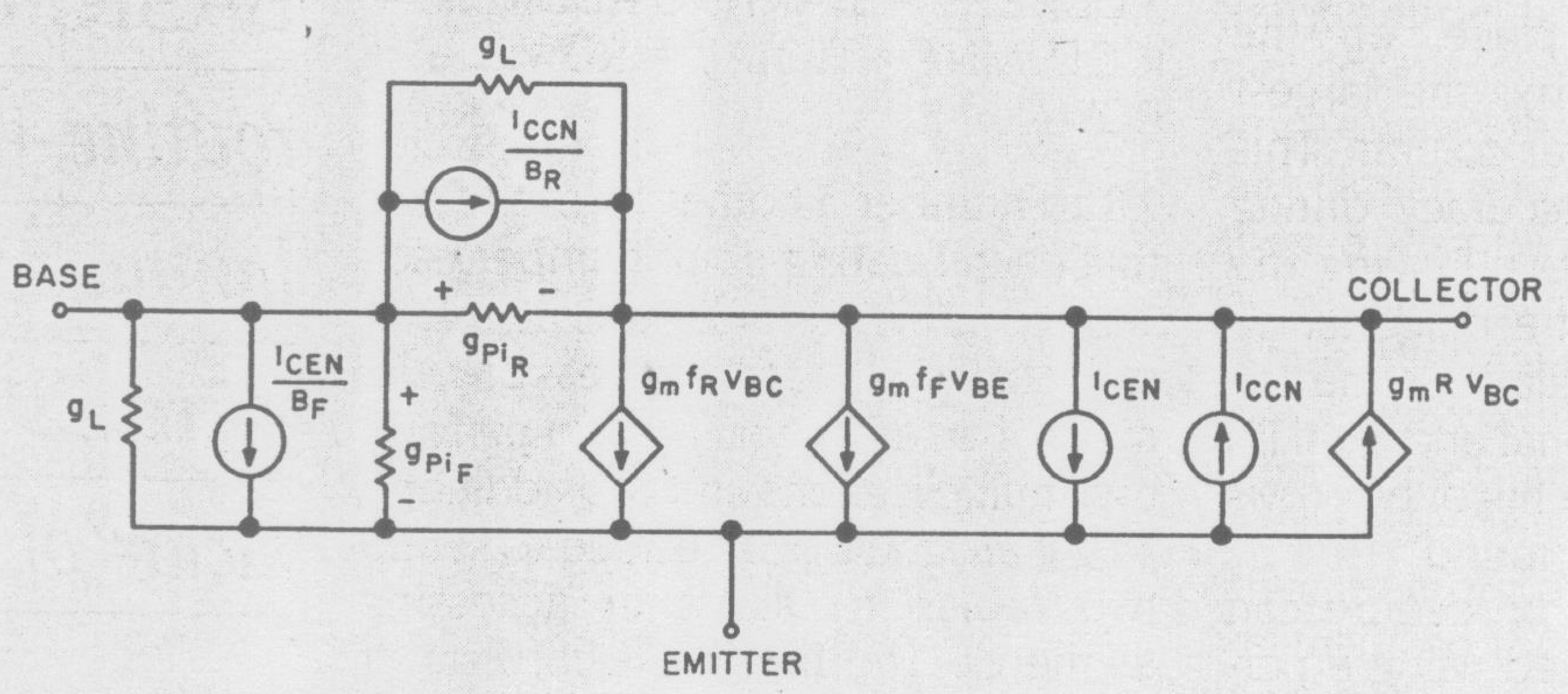
B_R = reverse beta

B_F = forward beta

V_A = Early voltage

Figure 5: *The nonlinear hybrid-pi model of the transistor.*

Linearized Hybrid-Pi Model



$$I_{CEN}/B_F = \frac{I_S\ (e^{V_{BE}\lambda} - 1) - \lambda I_S\ e^{V_{BE}\lambda} V_{BE}}{B_F}$$

$$g_{pi_F} = \frac{\lambda I_S\ e^{V_{BE}\lambda}}{B_F} \qquad g_{pi_R} = \frac{\lambda I_S\ e^{V_{BC}\lambda}}{B_R}$$

$$I_{CCN}/B_R = \frac{I_S\ (e^{V_{BC}\lambda} - 1) - \lambda I_S\ e^{V_{BC}\lambda} V_{BC}}{B_R}$$

$$\text{Let } \gamma = \frac{1}{V_A},\ \lambda = \frac{1}{V_T},\ g_L = IE-12$$

$$g_m f_R = I_S\ (e^{V_{BE}\lambda} - 1)\ \gamma$$

$$g_m f_F = \lambda I_S\ e^{V_{BE}\lambda}\ (1 - \gamma V_{BC})$$

$$g_m R = I_S\ [\lambda e^{V_{BC}\lambda}\ (1 - \gamma V_{BC}) - \gamma\ (e^{V_{BC}\lambda} - 1)]$$

Figure 6: *The linearized hybrid-pi model of the transistor resulting from performing a Taylor's series expansion on the nonlinear hybrid-pi model shown in figure 5.*

be very time-consuming since each element of the array is being addressed in BASIC. The equivalent routine written in machine language proved to be well worth the effort, as it was about 50 times faster than the BASIC routine.

Early in the project I became concerned with accuracy and whether or not the single-precision BASIC that I was using would do the job. In the Gaussian elimination routine, the roundoff error could reduce the number of significant digits by $1 + 2 \times \log N$, where N is the number of equations. However, in practice, I found on the many circuits I tested that the answers were well within the accuracy I needed.

On one circuit, however, the transconductance value calculated for a particular transistor was way off. This was because the computation for the transconductance of the transistor involved subtracting two numbers, and since that transistor was saturated, those two numbers happened to be very close in value to each other. The 31-bit mantissas of the floating-point variables did not provide enough accuracy for this case. I did not consider this a problem since the transconductance of a saturated transistor is not a very useful quantity anyway.

One benchmark test that I used was to perform a DC operating point analysis of the internal circuitry of a 741 operational amplifier with 26 nodes and 23 transistors. My program was able to perform this analysis in about 12 minutes and give results to within 1 percent of the results given by SPICE.

Although major simulation needs obviously cannot be met with a program like this, I find that, for small circuits, it has been very useful in providing analysis information without unduly compromising accuracy or computing-time requirements. ■

Editor's note: David McNeill's Commodore 64 program library "Circuits" is available in a variety of formats. See pages 459–461 for full details. You can obtain a copy of "Electronic Circuit Analysis," the 80-page user's manual for these programs, by sending $18 to the author.

BIBLIOGRAPHY

Biehl, Brian L. "Nonlinear JFET Model for Computer Aided Circuit Analysis," *IEEE Journal of Solid State Circuits*, Correspondence, February 1971.

Getreu, Ian. *Modeling the Bipolar Transistor*. Beaverton, OR: Tektronix Inc., 1976.

Nagel, Laurence W. "SPICE: A Computer Program to Simulate Semiconductor Circuits," Memorandum No. ERL-M520. Berkeley, CA: University of California, Electronics Research Laboratory.