NONLINEAR ELECTRONICS (NOEL) PACKAGE 6:

CANONICAL PIECEWISE-LINEAR DC ANALYSIS

by

An-Chang Deng and Leon O. Chua

Memorandum No. UCB/ERL M86/33

21 April 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

NONLINEAR ELECTRONICS (NOEL) PACKAGE 6:

CANONICAL PIECEWISE-LINEAR DC ANALYSIS

by

An-Chang Deng and Leon O. Chua

Memorandum No. UCB/ERL M86/33

21 April 1986

TITLE PAGE

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

NOEL PACKAGE 6 : CANONICAL PIECEWISE-LINEAR DC ANALYSIS†

An-Chang Deng and Leon O. Chua

Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California, Berkeley, CA 94720

ABSTRACT

The program in this package implements the Generalized Breakpoint Hopping algorithm published in [4]. It solves the dc problem for finding the dc operating point(s), and for tracing the driving-point and transfer characteristics of an extremely broad class of nonlinear resistive circuits. In particular, bipolar and MOS transistor circuits are included.

March 26, 1986

† Research supported by the Office of Naval Research under Contract N00014-76-C-0572, and by the National Science Foundation under Grant ECS-8313278.

NOEL PACKAGE 6 : CANONICAL PIECEWISE-LINEAR DC ANALYSIS

1. Introduction

The Newton-Raphson algorithm used in [1] for dc analysis is the most popular algorithm for solving nonlinear algebraic equation. It is however relatively inefficient due to the evaluation of the Jacobian matrix at each iteration, and may take a lot of iterations before converging to a solution or may not even converge. Moreover, the dc analysis in [1] requires an intermediate step to formulate the circuit equation to a C source code, and needs to compile and link it with the simulation routines. This compiling and linking process takes a major part of the cpu time in doing dc analysis. We are therefore motivated to develop a completely different approach, called Canonical Piecewise-Linear Analysis[2,3,4], for reasons of efficiency and convergence.

The new approach requires a preliminary step to model each nonlinear resistor to a canonical piecewise-linear representation[5]. More specifically, each 2-terminal nonlinear resistor is represented by a 1-dimensional canonical piecewise-linear function

$$y = a + bx + \sum_{i=1}^{\sigma} c_i \mid x - \beta_i \mid \tag{1}$$

where x denotes the terminal voltage (resp.; terminal current) of the resistor whose v-i characteristic is voltage-controlled (resp.; current-controlled), and y is the complementary variable. The 2-port or 3-terminal nonlinear resistor is modeled by a 2-dimensional canonical piecewise-linear function

$$y_1 = a_1 + b_{11}x_1 + b_{12}x_2 + \sum_{i=1}^{\sigma} c_{1i} \mid \alpha_{i1}x_1 + \alpha_{i2}x_2 - \gamma_i \mid \tag{2a}$$

$$y_2 = a_2 + b_{21}x_1 + b_{22}x_2 + \sum_{i=1}^{\sigma} c_{2i} \mid \alpha_{i1}x_1 + \alpha_{i2}x_2 - \gamma_i \mid \tag{2b}$$

where x_1 and x_2 are the controlling variables for port-1 and port-2 respectively.

The modeling process, called Canonical Piecewise-Linear Modeling[6], requires an optimization process and may take considerable amount of cpu time. It is however a one-shot process : the canonical piecewise-linear models, Eqs. (1) and (2), can be repeatedly applied in various circuits once they have been found.

With each nonlinear resistor modeled by an explicit global representation Eq.(1) or (2), the resistive circuit can be formulated as the canonical piecewise-linear equation[4]

$$\mathbf{f(x)} = \mathbf{a} + \mathbf{Bx} + \sum_{m=1}^{n_a}\sum_{i=1}^{\sigma_m} \mathbf{c}_{mi} \mid x_m - \beta_{mi} \mid + \sum_{j=l_a+1}^{l_a+l_b}\sum_{i=1}^{\sigma_j} \mathbf{h}_{ji} \mid <\boldsymbol{\alpha}_{ji} , \mathbf{z}_j > - \gamma_{ji} \mid$$

$$= \mathbf{0} \tag{3}$$

which can be solved with the Generalized Breakpoint Hopping Algorithm[4] by tracing the corresponding solution curve as shown later.

Besides the dc operating point problem, another important part of dc analysis is to find the DP (Driving-Point) and TC (Transfer) characteristics with respect to a driving source. The resistive circuit with a driving source can be represented by the canonical piecewise-linear equation[4]

$$\mathbf{g(x}, \rho) = -\mathbf{a} - \rho\mathbf{r} + \mathbf{Bx} + \sum_{m=1}^{n_a}\sum_{i=1}^{\sigma_m} \mathbf{c}_{mi} \mid x_m - \beta_{mi} \mid$$

$$+ \sum_{j=l_a+1}^{l_a+l_b} \sum_{i=1}^{\sigma_j} \mathbf{h}_{ji} \mid <\alpha_{ji} \ , \mathbf{z}_j > - \gamma_{ji} \mid$$

$$= \mathbf{0} \tag{4}$$

Equation (4) is characterized by a linear equation in each region where each absolute function is uniquely determined to be either positive or negative of the expression within $\mid \cdot \mid$. If we vary the value of ρ, then the corresponding solution of Eq.(4) will follow a straight line in that region until it hits a boundary. Hence the solution of Eq.(4) will in general follow a 1-dimensional polygonal curve, called the solution curve, in the domain space as we vary ρ continuously. The algorithm for tracing the solution curve is called the *Generalized Breakpoint Hopping Algorithm* [4], which calculates each breakpoint of the solution curve sequentially. The DP (Driving-Point) and TC (Transfer) characteristics can be obtained easily from the solution curve since the output variable of DP or TC is one of the components in the vector $\mathbf{x}$ or their linear combination.

The same algorithm, Generalized Breakpoint Hopping Algorithm which traces DP or TC characteristics, can be applied to find the dc operating point(s) for solving Eq.(3). We introduce a parameter ρ and transform Eq.(3) into a parametric equation

$$\begin{aligned} \mathbf{g}(\mathbf{x} \ , \rho) &= \mathbf{f}(\mathbf{x}) - (1-\rho)\mathbf{f}(\hat{\mathbf{x}}) \\ &= \mathbf{a} + \mathbf{B}\mathbf{x} + \sum_{m=1}^{n_a} \sum_{i=1}^{\sigma_m} \mathbf{c}_{mi} \mid x_m - \beta_{mi} \mid \\ &\quad + \sum_{j=l_a+1}^{l_a+l_b} \sum_{i=1}^{\sigma_j} \mathbf{h}_{ji} \mid <\alpha_{ji} \ , \mathbf{z}_j > - \gamma_{ji} \mid - (1-\rho)\hat{\mathbf{y}} \\ &= \mathbf{0} \end{aligned} \tag{5}$$

which has the same form as Eq.(4), where $\hat{\mathbf{y}} = \mathbf{f}(\hat{\mathbf{x}})$, and $\hat{\mathbf{x}}$ is an arbitrary point similar to the initial guess in Newton-Raphson algorithm. Using the Generalized Breakpoint Hopping Algorithm, the solution curve of Eq.(5) can be traced from $\mathbf{x} = \hat{\mathbf{x}}$ at $\rho = 0$ and the solution of $\mathbf{f}(\mathbf{x}) = 0$ is found at $\rho = 1$ where

$$\mathbf{g}(\mathbf{x} \ , 1) = \mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{6}$$

Hence, the Generalized Breakpoint Hopping Algorithm encountered in canonical piecewise-linear analysis provides a unified method for analyzing piecewise-linear resistive circuits; namely, determining dc operating point(s) and tracing DP and TC characteristics.

2. Algorithm

Step 1.

Find the generalized implicit equation

$$\mathbf{Pv} + \mathbf{Qi} + \mathbf{s} = 0 \tag{7}$$

for the linear n-port obtained by extracting each 1-port or 2-port piecewise-linear resistor from the circuit such that all the linear elements and the independent voltage or current sources are contained within the n-port.

Step 2.

Decode the element characteristic of each piecewise-linear resistor into the canonical piecewise-linear expression :

(i) 2-terminal voltage-controlled piecewise-linear resistor characterized by

$$\{i = (x_0 , y_0)(x_1 , y_1).....(x_\sigma , y_\sigma)(x_{\sigma+1} , y_{\sigma+1})\}$$

is decoded into the 1-dimensional canonical piecewise-linear expression

$$i = a + bv + \sum_{j=1}^{\sigma} c_j \mid v - \beta_j \mid \tag{8}$$

where

$$\beta_j = x_j \qquad j = 1,2,...,\sigma \tag{9a}$$

$$b = \frac{m_0 + m_\sigma}{2} \tag{9b}$$

$$c_j = \frac{m_j - m_{j-1}}{2} \qquad j = 1,2,...,\sigma \tag{9c}$$

$$a = y_0 - bx_0 - \sum_{j=1}^{\sigma} c_j \mid x_0 - \beta_j \mid \tag{9d}$$

$$m_j = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} \qquad j = 0,1,...,\sigma \tag{9e}$$

(ii) 2-terminal current-controlled piecewise-linear resistor characterized by

$$\{v = (x_0 , y_0)(x_1 , y_1).....(x_\sigma , y_\sigma)(x_{\sigma+1} , y_{\sigma+1})\}$$

is decoded into the 1-dimensional canonical piecewise-linear expression

$$v = a + bi + \sum_{j=1}^{\sigma} c_j \mid i - \beta_j \mid \tag{10}$$

where a , b , c_j , and β_j are defined in Eq.(9).

(iii) piecewise-linear 2-port resistor characterized by

$\{(z_1 , z_2):P\,1=(a_1 , b_{11} , b_{12} , c_{11} , c_{12} , ... c_{1\sigma});P\,2=(a_2 ,$

$\$ b_{21} , b_{22} , c_{21} , c_{22} , ... c_{2\sigma});Bd=(\alpha_{11} , \alpha_{12} , \gamma_1 , \alpha_{21} , \alpha_{22} ,$

$\$ \gamma_2 , ... , \alpha_{\sigma 1} , \alpha_{\sigma 2} , \gamma_\sigma\}$

is decoded as 2-dimensional canonical piecewise-linear function

$$y_1 = a_1 + b_{11}x_1 + b_{12}x_2 + \sum_{i=1}^{\sigma} c_{1i} \mid \alpha_{i1}x_1 + \alpha_{i2}x_2 - \gamma_i \mid \tag{11a}$$

$$y_2 = a_2 + b_{21}x_1 + b_{22}x_2 + \sum_{i=1}^{\sigma} c_{2i} \mid \alpha_{i1}x_1 + \alpha_{i2}x_2 - \gamma_i \mid \tag{11b}$$

where

(1) $(x_1 , x_2) = (v_1 , v_2)$ and $(y_1 , y_2) = (i_1 , i_2)$
if $z_1 = i$, $z_2 = i$ for a voltage-voltage controlled 2-port resistor.

(2) $(x_1 , x_2) = (i_1 , i_2)$ and $(y_1 , y_2) = (v_1 , v_2)$
if $z_1 = v$, $z_2 = v$ for a current-current controlled 2-port resistor.

(3) $(x_1 , x_2) = (v_1 , i_2)$ and $(y_1 , y_2) = (i_1 , v_2)$
if $z_1 = i$, $z_2 = v$ for a voltage-current controlled 2-port resistor.

(4) $(x_1 , x_2) = (i_1 , v_2)$ and $(y_1 , y_2) = (v_1 , i_2)$
if $z_1 = v$, $z_2 = i$ for a current-voltage controlled 2-port resistor.

Steps 3-7 are to find the dc operating point(s) :

Step 3.

Combine the linear n-port equation (7) and the canonical piecewise-linear models Eqs.(8) and (10) to the canonical piecewise-linear equation (3) for finding the dc operating point(s) (see [4] for details).

Step 4.

Choose an initial point $\hat{\mathbf{x}}$ and transform Eq.(3) to the parametric equation (5).

Step 5.

Apply the Generalized Breakpoint Hopping Algorithm (see [4] for details) to trace the solution curve of Eq.(5) with starting point $\hat{\mathbf{x}}$ at $\rho = 0$. A solution $\mathbf{x}^*$ is obtained at $\rho = 1$ and is written to the output file "xx...x.op" if it differs from any of the previously found solution in "xx...x.op". More specifically, if

$$| \, | \mathbf{x}^* - \mathbf{x}^{[i]} | \, |_1 > \epsilon = 10^{-6}$$

for i=1,2,...,s, then store $\mathbf{x}^*$ to "xx...x.op" by assigning $\mathbf{x}^{[s+1]} = \mathbf{x}^*$, where "xx...x.op" contains s distinct dc operating points $\mathbf{x}^{[1]}$, $\mathbf{x}^{[2]}$,...., $\mathbf{x}^{[s]}$.

Step 6.

Stop the analysis if an aborting message is entered by the user.

Step 7.

If the solution curve enters an unbounded region, go to Step 4 for tracing another branch of solution curve if and only if a continue message is entered by the user.

Steps 8-12 are to trace the DP and TC characteristics.

Step 8.

Decode the driving source

$\{(d_{\min} , d_{\max}) : dir \}$

such that the solution curve starts from $d_{\min}$ until $d_{\max}$ is reached, and it is in the increasing direction (resp.; decreasing direction) of source value if dir ='+' (resp.; dir ='-').

Step 9.

Combine the generalized implicit equation (7), the canonical piecewise-linear models Eqs.(8) and (10), and the driving source characteristic to formulate the canonical piecewise-linear equation (4) for tracing DP and TC characteristics (see [4] for details).

Step 10.

Follow Steps 3-7 to find the dc operating point(s) at starting source value $\rho^{(0)} = d_{\min}$.

Step 11.

Apply the Generalized Breakpoint Hopping Algorithm to trace the solution curve of Eq.(4). Each branch of solution curve starts from a point $\mathbf{x}^{[i]}$ at $\rho = \rho^{(0)}$ for i=1,2,...,s, where the file "xx...x.op" contains s distinct operating points $\mathbf{x}^{[1]}$, $\mathbf{x}^{[2]}$,..., $\mathbf{x}^{[s]}$ at $\rho = \rho^{(0)}$.

Step 12.

Plot the DP or TC characteristic by selecting an output variable from the solution curve. The output variable for DP characteristic is the driving-point current (resp.; driving-point voltage) if it is driven by a voltage source (resp.; current source). The output variable for TC characteristic may be any one of the following variables :

(1) voltage of a 2-terminal voltage-controlled pwl resistor

(2) current of a 2-terminal current-controlled pwl resistor

(3) port-1 voltage or port-2 voltage of a voltage-voltage controlled 2-port resistor

(4) port-1 current or port-2 current of a current-current controlled 2-port resistor

(5) port-1 voltage or port-2 current of a voltage-current controlled 2-port resistor

(6) port-1 current or port-2 voltage of a current-voltage controlled 2-port resistor

3. User's Instruction

Steps 1-2 are to find the dc operating point(s) :

Step 1.

Create a file "xx...x.spc" which describes the piecewise-linear resistive circuit and follows the rules of the input format language defined in [7] for each class of circuit elements, where "xx...x" is the filename for the input file with extension ".spc". Only the resistive circuit element can be included in "xx...x.spc"; namely

'R' : 2-terminal resistor (linear or pwl)

'V' : independent voltage source or driving voltage source

'I' : independent current source or driving current source

'E' : linear voltage-controlled voltage source

'F' : linear current-controlled current source

'G' : linear voltage-controlled current source

'H' : linear current-controlled voltage source

'N' : 2-port or 3-terminal resistor (linear or pwl)

Each nonlinear resistor must be modeled by a numerical pwl expression as defined in [7] for 1-port or 2-port resistor.

Step 2.

Type the command

pwldc xx...x

to perform canonical piecewise-linear dc analysis for finding the operating point(s). It proceeds interactively with user in the following procedures :

(a) **enter the initial point**

v(R1) =

i(R2) =

.....

An arbitrary initial point $\hat{x}$ is entered in terms of each controlling variable in Step 12 of Section 2.

(b) **the solution is**

v(R1) =

i(R2) =

.....

continue tracing present branch of solution curve ? y/n

If a solution is found, then it is printed in terms of each controlling variable and prompts a message to ask the user whether to continue tracing the present branch of solution curve in order to find more operating points.

(c) If the user decides to continue tracing the present branch of solution curve, then it repeats the computing process in (a) until another operating point is found;

else abort the present branch of solution curve and go to (d).

(d) **try another branch of solution curve? y/n**

If 'y' then go to (a) to repeat the same procedures for finding other operating point(s);

else stop the analysis.

(e) If the solution curve goes to an unbounded region without hitting any boundary, then it prompts a message :

solution curve goes to infinity in the unbounded region

try another branch of solution curve? y/n

If 'y' then go to (a);

else stop the analysis.

Steps 3-6 are to trace the DP and TC characteristics and are combined as a batch process which can be executed by typing the command

pwl xx...x

where "xx...x.spc" is the input file.

Step 3.

Create a file "xx...x.spc" which describes the piecewise-linear resistive circuit including the driving source. The graphic control lines are required to specify the graphic titles and ranges (see [7]).

Step 4.

Type the command

start xx...x

to transform the file "xx...x.spc" into a temporary file "temp.spc" such that the driving source is replaced by an independent source with value equal to the starting source value d_{min}.

Step 5.

Type the command

pwldc temp

to perform the dc analysis for finding the dc operating point(s) at the starting source value d_{min}. Each operating point is written into the file "xx...x.op".

Step 6.

Type the command

pwltf xx...x

to perform canonical piecewise-linear dc analysis for tracing the DP and TC characteristics. It proceeds interactively with user in the following way :

(a) It first prints a mapping table for the definition of each lattice structure variable x and semi-lattice structure variable z :

****

x[0]=v(R1)

x[1]=i(R2)

z[0]=v1(Nbp)

z[1]=i2(Nbp)

.....

****

(b) **enter the output variable : x/z/i**

Choose the output variable for DP or TC characteristic; type

'i' for DP characteristic

'x' for TC characteristic and the output variable is chosen from the lattice structure variable

'z' for TC characteristic and the output variable is chosen from the semi-lattice structure variable

(c) **TC output = x[?]**

If 'x', enter the index number for the output variable (look up the mapping table for the definitions of lattice structure variables).

(d) **TC output = z[?]**

If 'z', enter the index number for the output variable (look up the mapping table for the definitions of semi-lattice structure variables).

Step (e) appears only when -r option is specified in Step 6; namely

pwltf -r xx...x

or

pwl -r xx...x

for the batch command line.

(e) **enter input variable range for region information**

Enter the input range r_1 and r_2 such that the region information is printed for any input value x with $r_1 < x < r_2$.

4. Output Format

(i) DC operating point(s)

Each computed operating point is written into the output file "xx...x.op" if it differs from any operating point already existing in "xx...x.op". Each operating point, immediately following a line with stars "****.....**", is given in terms of the controlling variables of the piecewise-linear 1-port and 2-port resistors, which are listed in Step 12 of Section 2. The file is terminated by a line "&&&&&.....&&" to denote end of the file.

(ii) DP and TC characteristics

Each branch of DP or TC curve, starting from each operating point of the file "xx...x.op", is shown on the color monitor with a designated color.

In addition to this graphic output, the breakpoints of each curve are also listed in the output file "xx...x.out" if the -o option is specified in Step 6 of Section 3; namely,

pwltf -o xx...x

or

pwl -o xx...x

for the batch command. When the region information is requested by the user, the boundary equations of the particular region are shown as follow :

AT INPUT = %%...%
********** REGION INFORMATION **********

a <= v(R1) <= b
c <= i(R2) <= d
.....
p*v1(Nbp)+q*i2(Nbp)+r >= 0
s*v1(Nbp)+t*i2(Nbp)+u <= 0
.....

5. Examples

Example 1 : in file "ex1.spc"

A mixed 1-port and 2-port piecewise-linear resistive circuit (Fig.1).

Example 2 : in file "ex2.spc"

A 2-transistor type-N negative resistance circuit (Fig.2).

Example 3 : in file "ex3.spc"

A two-transistor odd-symmetric voltage-controlled negative resistance circuit (Fig.3).

Example 4 : in file "ex4.spc"

Schmitt-Trigger circuit (Fig.4).

Example 5 : in file "ex5.spc"

An OP circuit (Fig.5).

Example 6 : in file "ex6.spc"

An NMOS depletion load inverter (Fig.6).

Example 7 : in file "ex7.spc"

A CMOS inverter (Fig.7).

6. Diagnosis

DC operating point(s)

1. see Section 6 of [8].
2. PWLDC SPICE_FILE

   Bad command line, the correct one should be

   pwldc xx...x

   where "xx...x.spc" is the input file.
3. CAN'T OPEN OUTPUT_FILE xx...x.op

   Can't open the output file "xx...x.op" which is to store the computed operating point(s).
4. INCORRECT DIMENSION J=??..?

   The total number of branches in pwl elements is not equal to n, the port number of the linear n-port.
5. HIT A CORNER POINT ON LATTICE STRUCTURE BOUNDARIES

   The solution curve hits a corner point located in the intersection of more than one lattice structure boundaries.
6. HIT A CORNER POINT ON SEMI-LATTICE STRUCTURE BOUNDARIES

   The solution curve hits a corner point located in the intersection of more than one semi-lattice structure boundaries.
7. HIT A CORNER POINT ON LATTICE AND SEMI-LATTICE BOUNDARIES

   The solution curve hits a corner point located in the intersection of lattice and semi-lattice structure boundaries.
8. GET STUCKED IN A BOUNDARY

   The given initial point is accidentally located in a lattice structure boundary, and the direction of solution curve is to leave the region where the initial equation is defined.

DP and TC characteristics

9. PWLTF SPICE_FILE

   Bad command line, the correct one should be

   pwltf xx...x

   where "xx...x.spc" is the input file.
10. CAN'T OPEN OP_FILE xx...x.op

    The file "xx...x.op" does not exist in the current directory.
11. INCORRECT DIMENSION J=??...?

    The total number of the pwl branches and the driving source is not equal to n, the port number of the linear n-port.
12. INCORRECT DRIVING SOURCE MODEL FOR TRACING DIRECTION

    Bad format in the driving source model for tracing direction; should be either {.....:+} or {.....:-}.
13. TOO MANY DIGITS FOR STARTING POINT

    The numerical data ???....? for the starting point in the driving source model {(???....?,.....} has too many digits; it should not exceed 20 digits.
14. TOO MANY DIGITS FOR ENDING POINT

    The numerical data ???....? for the ending point in the driving source model {.....,???....?)..} has too many digits; it should not exceed 20 digits.

15. CAN'T READ THE STARTING POINT

    Can't read the starting point from the file "xx...x.op".

16. MISMATCHED LATTICE VARIABLE NAME

    The element name of lattice structure variable in the input file "xx...x.spc" does not match that in the file "xx...x.op" obtained by dc analysis for operating point(s).

17. MISMATCHED SEMI-LATTICE VARIABLE NAME

    The element name of semi-lattice structure variable in the input file "xx...x.spc" does not match that in the file "xx...x.op" obtained by dc analysis for operating point(s).

18. BREAK A LOOP

    Abort the present branch of solution curve when the solution curve is detected to follow a closed loop; it will start from the next starting point in the file "xx...x.op" for a new branch of solution curve.

19. WARNING MESSAGE : CORNER POINT IN LATTICE BOUNDARIES

    The solution curve hits a corner point located in the intersection of more than one lattice structure boundaries.

20. WARNING MESSAGE : CORNER POINT IN SEMI-LATTICE BOUNDARIES

    The solution curve hits a corner point located in the intersection of more than one semi-lattice structure boundaries.

21. WARNING MESSAGE : CORNER POINT IN LATTICE AND SEMI-LATTICE BOUNDARIES

    The solution curve hits a corner point located in the intersection of lattice and semi-lattice structure boundaries.

Both Operating Point and DP, TC Characteristics

22. CAN'T OPEN SPICE_FILE xx...x.spc

    The input file "xx...x.spc" does not exist in the current directory.

23. NOT ENOUGH SPACE; LAT should be no less than ??...?

    The total number of 2-terminal pwl resistors in the circuit (=??...?) exceeds the upper bound 60; the constant LAT in the program should be changed to ??...? in order to analyze this circuit.

24. NOT ENOUGH SPACE; SEMI should be no less than ??...?

    The total number of pwl 2-port resistors in the circuit (=??...?) exceeds the upper bound 30; the constant SEMI in the program should be changed to ??...? in order to analyze this circuit.

25. NOT ENOUGH SPACE; DIM should be no less than ??...?

    The total number of breakpoints (=??...?) of the 2-terminal pwl resistors in the circuit is beyond the upper bound 1000; the constant DIM in the program should be changed to ??...? in order to analyze this circuit.

26. NOT ENOUGH SPACE; DIP should be no less than ??...?

    The total number of branches for 2-port pwl resistors is beyond the limit 60; the constant DIP in the program should be changed to ??...? in order to analyze this circuit.

27. NOT ENOUGH SPACE; DIQ should be no less than ??...?

    The total number of partition boundaries in the models of pwl 2-port resistors exceeds the upper bound 2000; the constant DIQ should be changed to ??...? in order to analyze this circuit.

28. NOT ENOUGH SPACE; DIN should be no less than ??...?

The total number of partition boundary parameters in the pwl models for pwl 2-port resistors exceeds the upper bound 2000; the constant DIN in the program should be changed to ??...? in order to analyze this circuit.

Remark :

The numbers LAT, SEMI, DIM, DIN, DIP, and DIQ in 23-28 are defined constants in the program. They are used as rough estimations for some arrays before the exactly dimensions are available. When each pwl element in the circuit has been processed for the formulation of canonical pwl equation, the required dimension for each array is then available and used for a second allocation with exact required spaces. The messages 23-28 occur when the required spaces of some arrays exceed the pre-allocated spaces.

29. TOO MANY DIGITS IN PWL 1-PORT MODEL

The numerical parameter in the pwl 1-port (or 2-terminal) resistor has too many digits and is beyond the upper bound 20.

30. TOO MANY BREAKPOINTS IN PWL 1-PORT MODEL

The pwl model for 1-port (2-terminal) resistor has too many breakpoints and is beyond the upper bound 10.

31. MISSING '}' IN THE MODEL

The model description of a pwl resistor is not included within a pair of brackets "{.....}".

32. TOO MANY PARAMETERS IN THE MODEL OF PWL 2-PORT

The model of a pwl 2-port resistor has too many numerical parameters and is beyond the upper bound 30.

33. TOO MANY DIGITS IN PWL 2-PORT MODEL

The numerical parameter in the pwl 2-port resistor has too many digits and is beyond the upper bound 20.

34. INCORRECT PWL 2-PORT MODEL

The model description of a pwl 2-port resistor is not one of the following 4 types :

(1) {(i,i):.....}

(2) {(v,v):.....}

(3) {(v,i):.....}

(4) {(i,v):.....}

35. INITIAL POINT IS IN UNDEFINED REGION

Can't find the initial region where the initial point is located; should check the breakpoints in pwl 1-port model to see whether they are in sequential order.

References

[1] A.C.Deng and L.O.Chua, "NOnlinear ELectronics package 3 : nonlinear DC analysis," ERL Memo., M86, University of California, Berkeley, 1986.

[2] L.O.Chua and R.Ying, "Canonical piecewise-linear analysis," *IEEE Trans. Circuits and Systems* , CAS-30, pp.125-140, Mar. 1983.

[3] L.O.Chua and A.C.Deng, "Canonical piecewise-linear analysis, Part II : tracing driving-point and transfer characteristics," *IEEE Trans. Circuits and Systems* , CAS-32, pp.417-444, May 1985.

[4] L.O.Chua and A.C.Deng, "Canonical piecewise-linear analysis : generalized breakpoint-hopping algorithm," *Int. J. of Circuit Theory and Applications* , Vol. 14, Jan. 1986.

[5] S.M.Kang and L.O.Chua, "A global representation of multi-dimensional piecewise-linear functions with linear partitions," *IEEE Trans. Circuits and Systems* , CAS-25, pp.938-940, Nov. 1978.

[6] L.O.Chua and A.C.Deng, "Canonical piecewise-linear modeling," *IEEE Trans. Circuits and Systems* , Vol. CAS-33, Apr., 1986.

[7] A.C.Deng and L.O.Chua, "NOnlinear ELectronics package 0 : general description," ERL Memo., M86, University of California, Berkeley, 1986.

[8] A.C.Deng and L.O.Chua, "NOnlinear ELectronics package 1 : linear circuit formulations, n-port representations and state equations," ERL Memo., M86, University of California, Berkeley, 1986.

Figure Captions

Fig.1 A mixed 1-port and 2-port piecewise-linear resistive circuit.

Fig.2 A 2-transistor type-N negative resistance circuit.

Fig.3 A 2-transistor odd-symmetric voltage-controlled negative resistance circuit.

Fig.4 Schmitt-Trigger circuit.

Fig.5 An OP circuit.

Fig.6 An NMOS depletion load inverter.

Fig.7 A CMOS inverter.

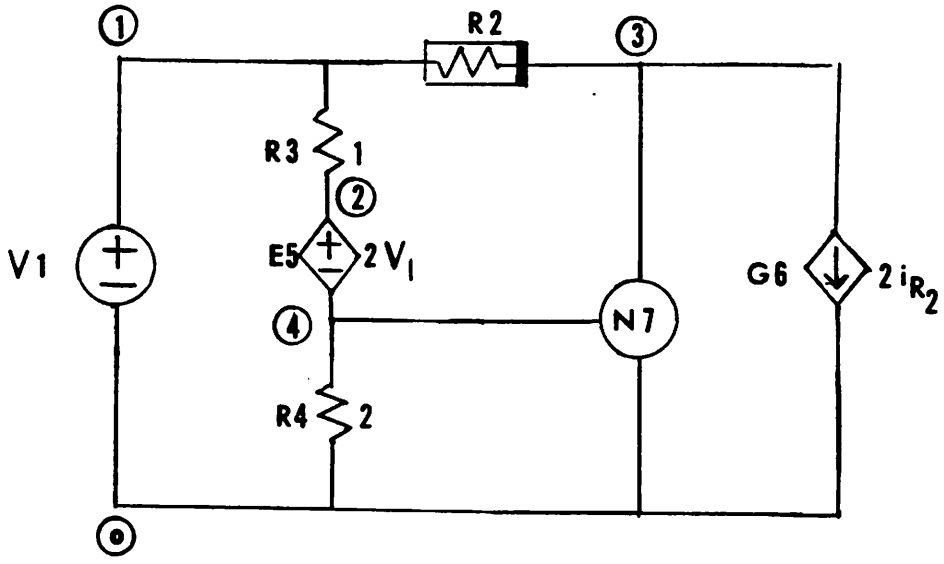

①
R2
③
R3
1
②
V1
E5
2 V1
G6
2 iR2
④
N7
R4
2
⓪

Fig.1

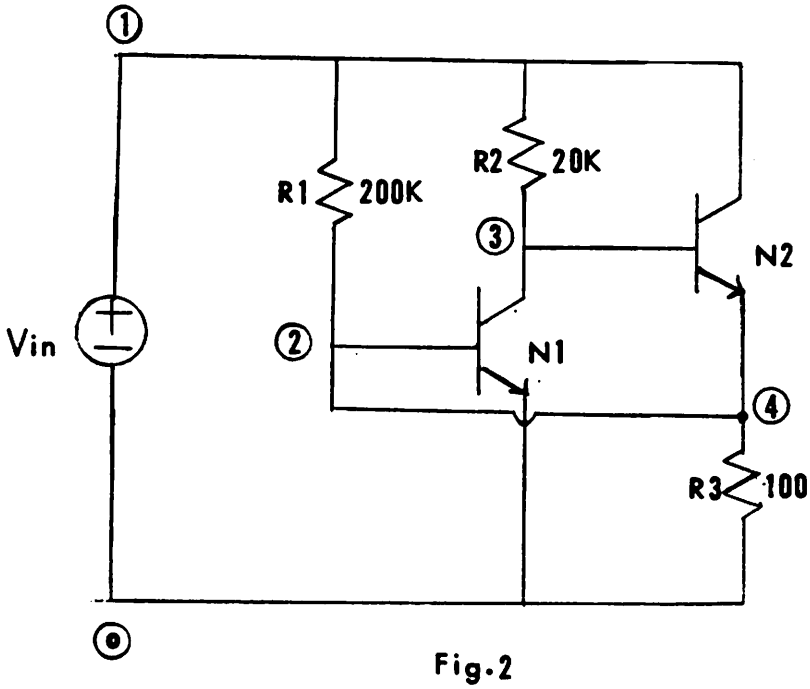

①
R1
200K
R2
20K
③
N2
Vin
②
N1
④
R3
100
⓪

Fig.2

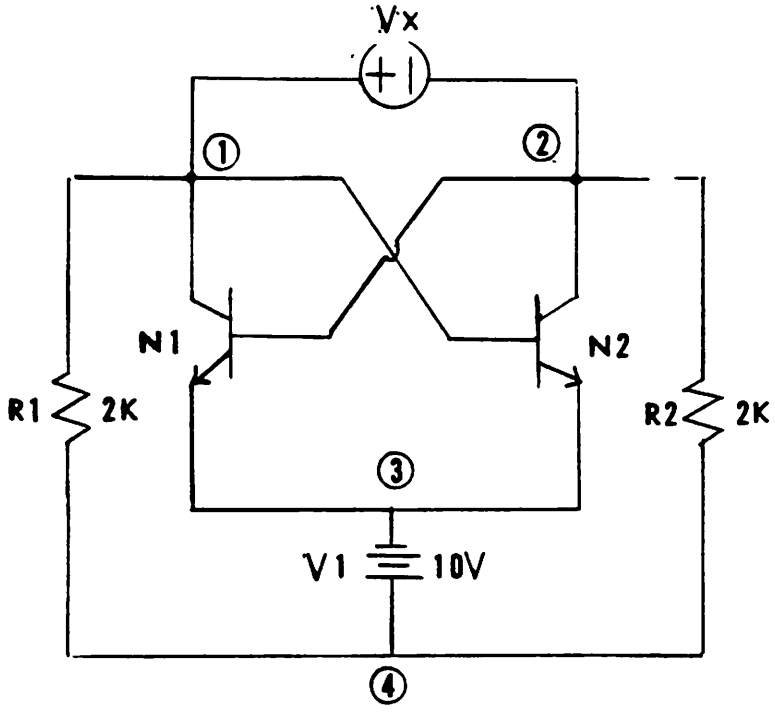

Vx
+
①
②
N1
N2
R1 2K
R2 2K
③
V1 10V
④

Fig.3

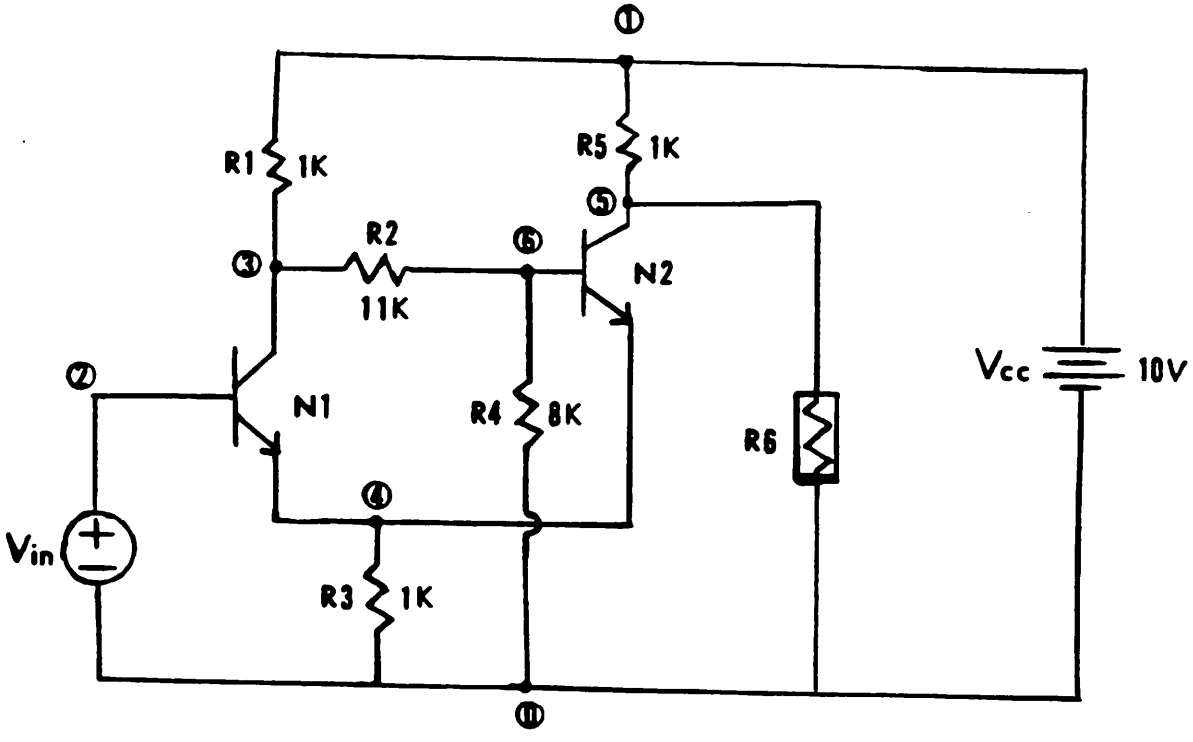

①
R1 1K
R5 1K
⑤
R2
11K
③
⑥
N2
②
N1
R4 8K
R6
Vcc 10V
④
Vin
R3 1K
⑪

Fig.4

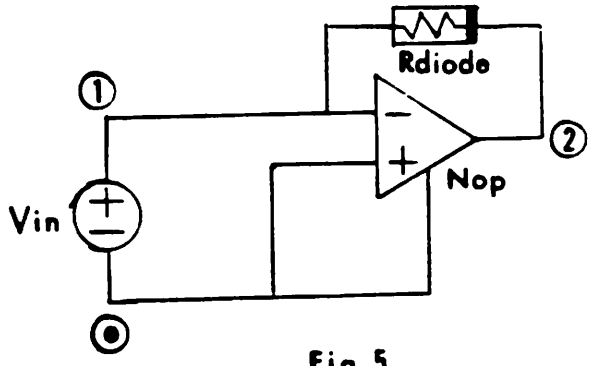


Fig.5

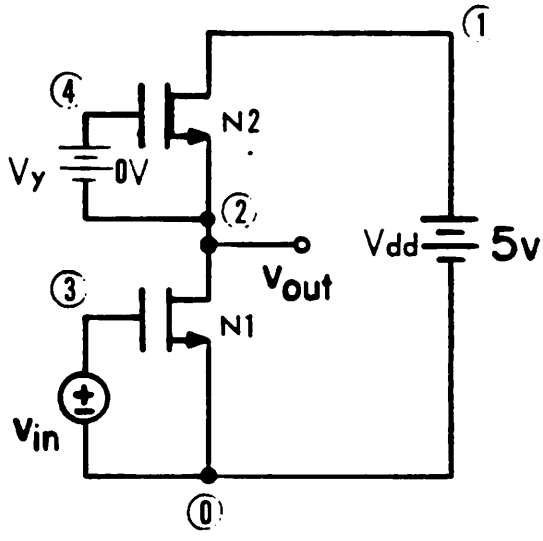


Fig.6

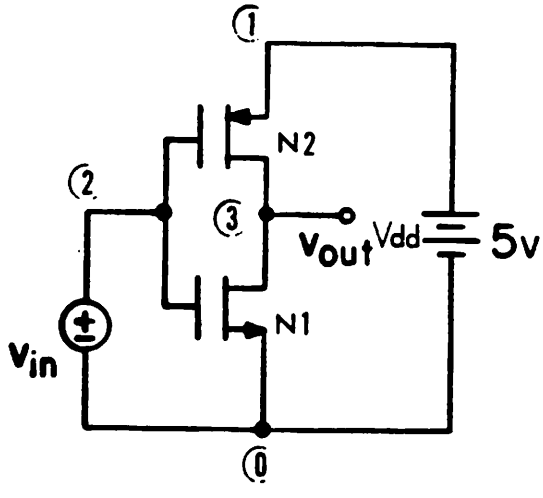
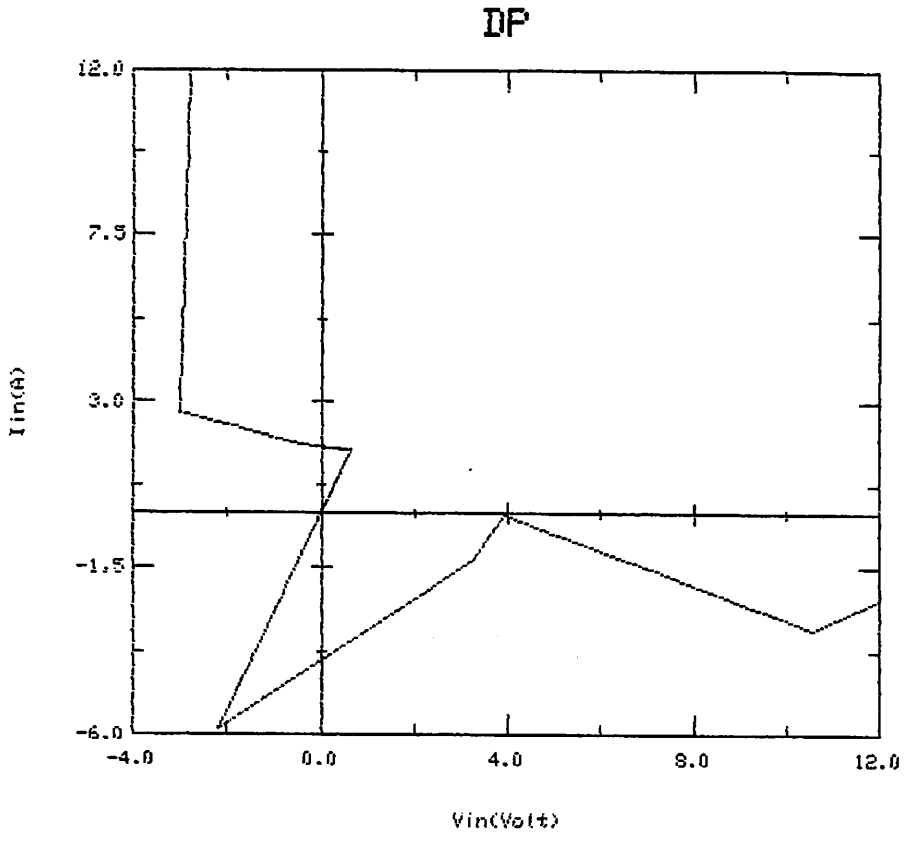


Fig.7

```
* Example 1
*
* mixed 1-port and 2-port pwl resistive circuit
V1 1 0 {(0,12):+}
R2 1 3 {i=(0,0)(1,2)(3,-2)(4,-1)}
R3 1 2 1
R4 4 0 2
E5 2 4 1 0 2
G6 3 0 1 3 2
N7 4 3 4 0 mod
.x_name Vin(Volt)
.y_name Iin(A)
.title DP
.x_axis -4 12
.y_axis -6 12
.model mod {(i,v):P1=(-3,1,4,2,1,0);P2=(-1,1,5,1,1,-1);
$Bd=(1,-1,-1,1,1,1,0,1,-1)}
.end
```

```
CANONICAL PWL DC ANALYSIS : FINDING DC OPERATING POINT

************************************************************
x[0]=v(R2)
z[0]=v1(N7)
z[1]=i2(N7)
************************************************************
enter the initial point
v(R2)= 1
v1(N7)= 2
i2(N7)= 3
the solution is
v(R2)=-1.887e-15
v1(N7)=-6.661e-16
i2(N7)=-9.437e-16
continue tracing present branch of solution curve? y/n
y

the solution is
v(R2)=-2.000e+00
v1(N7)=-2.000e+00
i2(N7)=-1.221e-15
continue tracing present branch of solution curve? y/n
y
solution curve goes to infinity in the unbounded region
try another branch of solution curve? y/n
y
enter the initial point
v(R2)= 2
v1(N7)= 3
i2(N7)= 1
the solution is
v(R2)=1.302e+00
v1(N7)=9.302e-01
i2(N7)=-6.512e-01
continue tracing present branch of solution curve? y/n
n
try another branch of solution curve? y/n
n

END OF CANONICAL PWL DC ANALYSIS FOR DC OPERATING POINT

************************************************************
x[0]=v(R2)
z[0]=v1(N7)
z[1]=i2(N7)
************************************************************
enter the output variable : x/z/i
i
```

C>
φ



```
* Example 2
*
* 2-transistor type-N negative resistance circuit
R1 1 2 200K
R2 1 3 20K
R3 4 0 100
R4 2 4 10K
N1 2 0 2 3 bpmod
N2 3 4 3 1 bpmod
Vin 1 0 {(0,18):+}
*
* graphic control lines
.x_name Vin(Volt)
.y_name Iin(A)
.title DP
.x_axis 0 18
.y_axis 0 0.01
*
* the bipolar trsnsistor is modeled by a 2-dim canonical pwl function
.model bpmod {(i,i):P1=(-2.4604e-4,4.083e-2,-4.04465e-2,2.461e-4,
$9.824e-3,3.076e-2,-2.405e-4,-9.726e-3,-3.048e-2);
$P2=(-2.634e-2,-4.04167e-2,8.0891e-2,-2.407e-4,-9.726e-3,-3.045e-2,
$4.81e-4,1.945e-2,6.096e-2);
$Bd=(1,0,0.4413,1,0,0.6165,1,0,0.6632,0,1,0.4392,0,1,0.6165,
$0,1,0.6633)}
.end
```

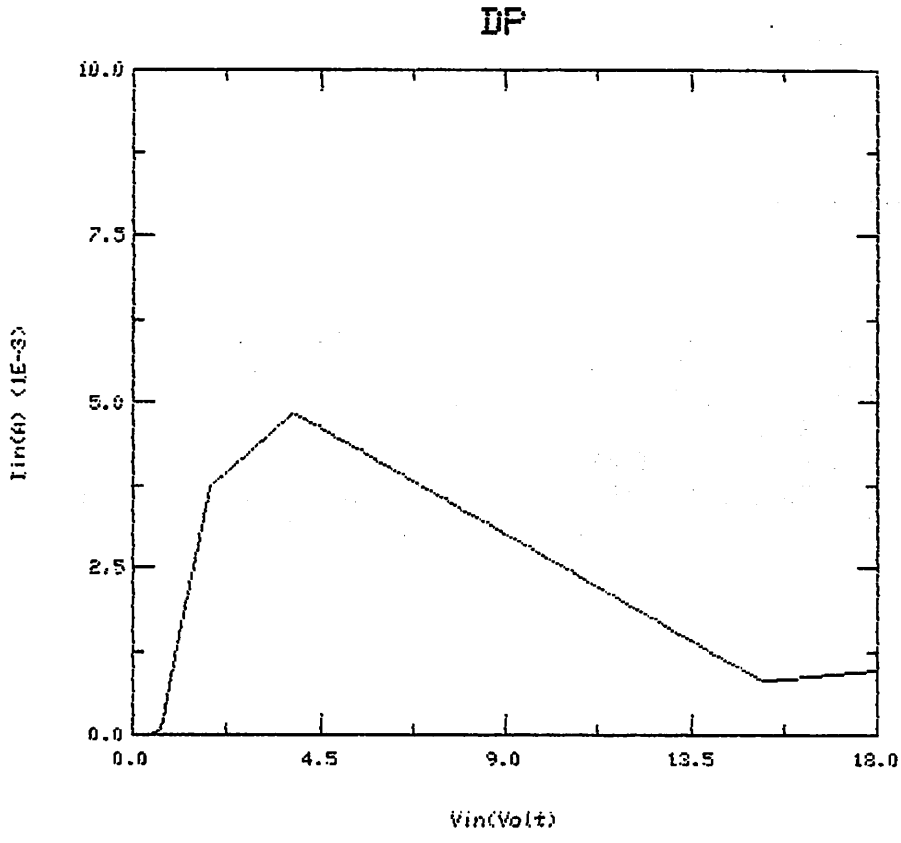
CANONICAL PWL DC ANALYSIS : FINDING DC OPERATING POINT

```
************************************************************
x[0]=v1(N1)
x[1]=v2(N1)
x[2]=v1(N2)
x[3]=v2(N2)
************************************************************
enter the initial point
v1(N1)= 0
v2(N1)= 0
v1(N2)= 0
v2(N2)= 0
the solution is
v1(N1)=-2.017e-03
v2(N1)=-1.991e-03
v1(N2)=-6.773e-06
v2(N2)=-2.661e-05
continue tracing present branch of solution curve? y/n
n
try another branch of solution curve? y/n
n
```

END OF CANONICAL PWL DC ANALYSIS FOR DC OPERATING POINT

```
************************************************************
x[0]=v1(N1)
x[1]=v2(N1)
x[2]=v1(N2)
x[3]=v2(N2)
************************************************************
enter the output variable : x/z/i
i
```

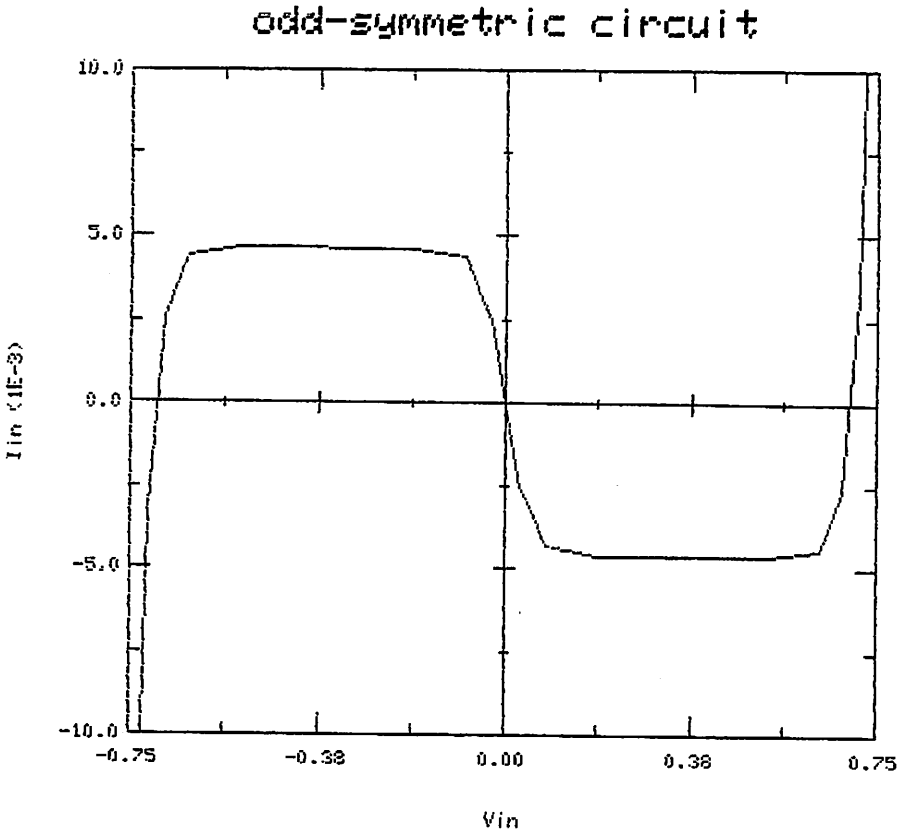
D>
Φ



```
* Example 3
*
* two transistor odd-symmetric v-controlled negative resistance circuit
*
N1 2 3 2 1 mod
N2 1 3 1 2 mod
R1 1 4 2K
R2 2 4 2K
V1 4 3 10
Vx 1 2 {(0.75,-0.75):-}
*
* graphic control lines
.x_name Vin
.y_name Iin
.title odd-symmetric circuit
.x_axis -0.75 0.75
.y_axis -0.01 0.01
*
* the bipolar transistor is modelled by 2-dim canonical pwl function
.model mod {(i,i):P1=(-1.84e-3,0.2654,-0.2628,1.115e-3,1.8786e-2,
$6.885e-2,0.17668,-1.104e-3,-1.86e-2,-6.817e-2,-0.1749);
$P2=(-0.1839,-0.2628,0.5256,
$-1.104e-3,-1.86e-2,-6.817e-2,-0.17493,
$2.208e-3,3.721e-2,0.13634,0.3499);Bd=(1,0,0.5297,1,0,0.6362,1,0,0.6817,
$1,0,0.7144,0,1,0.5297,0,1,0.6362,0,1,0.6817,0,1,0.7144)}
.end
```

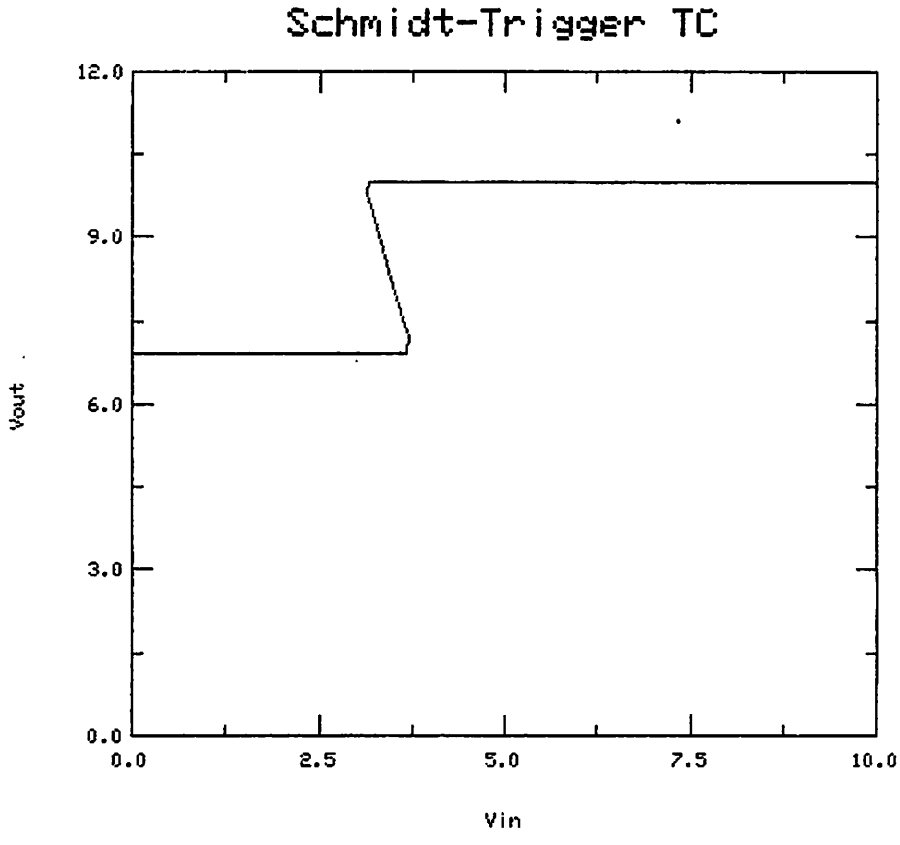
```
CANONICAL PWL DC ANALYSIS : FINDING DC OPERATING POINT

************************************************************
x[0]=v1(N1)
x[1]=v2(N1)
x[2]=v1(N2)
x[3]=v2(N2)
************************************************************
enter the initial point
v1(N1)= 0.2
v2(N1)= 0.3
v1(N2)= 0.4
v2(N2)= 0.5
the solution is
v1(N1)=7.676e-01
v2(N1)=7.500e-01
v1(N2)=1.757e-02
v2(N2)=-7.500e-01
continue tracing present branch of solution curve? y/n
n
try another branch of solution curve? y/n
n

END OF CANONICAL PWL DC ANALYSIS FOR DC OPERATING POINT

************************************************************
x[0]=v1(N1)
x[1]=v2(N1)
x[2]=v1(N2)
x[3]=v2(N2)
************************************************************
enter the output variable : x/z/i
i
```

D>
Φ


odd-symmetric circuit
10.0
5.0
0.0
-5.0
-10.0
Iin (1E-3)
-0.75
-0.38
0.00
0.38
0.75
Vin

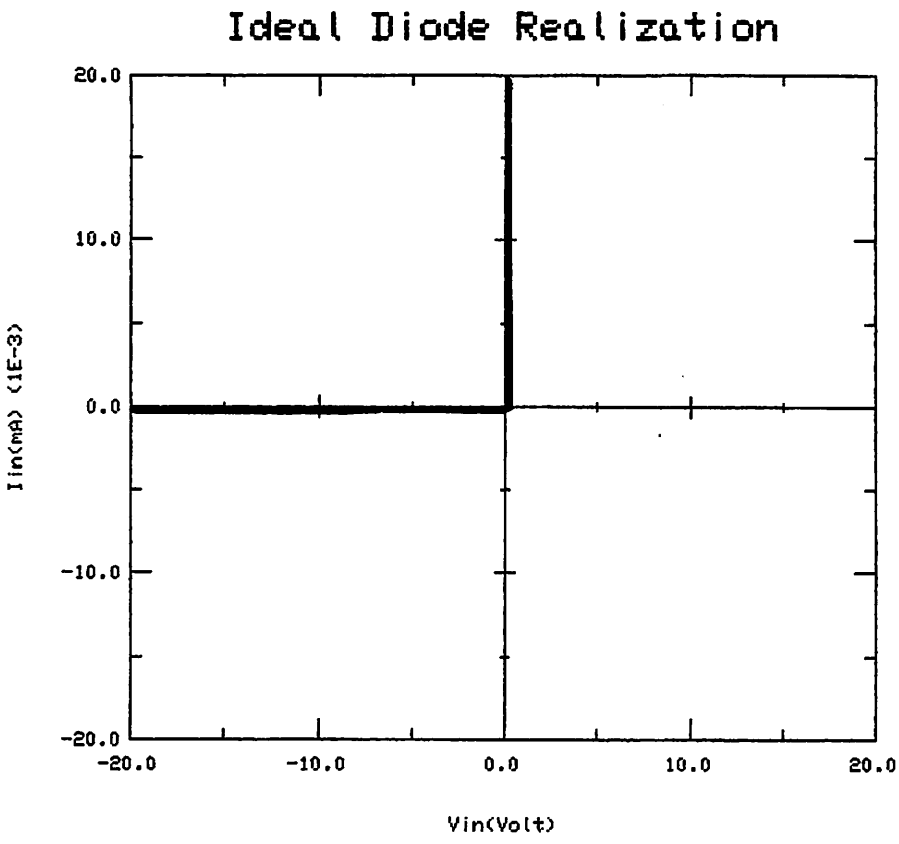
```
* Example 4
*
* Schmitt-Trigger Circuit
*
* the transistors are treated as 2-port resistors
N1 2 4 2 3 bpmod
N2 6 4 6 5 bpmod
*
Vcc 1 0 10
R1 1 3 1K
R2 3 6 11K
R3 4 0 1K
R4 6 0 8K
R5 1 5 1K
*
* the output nodes , nodes 5 and 0, are connected by an open-circuit resistor
R6 5 0 {i=(0,0)(0.5,0)}
*
* graphic control lines
.x_name Vin
.y_name Vout
.title Schmitt Trigger TC
.x_axis 0 10
.y_axis 0 12
*
* driving voltage source
Vin 2 0 {(0,10):+}
*
* the bipolar transistor is modelled by 2-dimensional canomical pwl function
.model bpmod {(i,i):P1=(-1.85935e-3,0.265431,-0.262774,1.115e-3,1.8786e-2,
$6.885e-2,0.17668,-1.104e-3,-1.86e-2,-6.817e-2,-0.1749);
$P2=(-0.18389,-0.262804,0.525658,
$-1.104e-3,-1.86e-2,-6.817e-2,-0.17493,
$2.208e-3,3.721e-2,0.13634,0.3499);Bd=(1,0,0.5297,1,0,0.6362,1,0,0.6817,
$1,0,0.7144,0,1,0.5297,0,1,0.6362,0,1,0.6817,0,1,0.7144)}
*
.end
```

```
CANONICAL PWL DC ANALYSIS : FINDING DC OPERATING POINT

************************************************************
x[0]=v(R6)
x[1]=v1(N1)
x[2]=v2(N1)
x[3]=v1(N2)
x[4]=v2(N2)
************************************************************
enter the initial point
v(R6)= 0.2
v1(N1)= 0
v2(N1)= 0.3
v1(N2)= -1
v2(N2)= 0.4

the solution is
v(R6)=6.893e+00
v1(N1)=-3.143e+00
v2(N1)=-9.490e+00
v1(N2)=6.879e-01
v2(N2)=-3.062e+00
continue tracing present branch of solution curve? y/n
n
try another branch of solution curve? y/n
n

END OF CANONICAL PWL DC ANALYSIS FOR DC OPERATING POINT

************************************************************
x[0]=v(R6)
x[1]=v1(N1)
x[2]=v2(N1)
x[3]=v1(N2)
x[4]=v2(N2)
************************************************************
enter the output variable : x/z/i
x
output=x[?]
0
```


Schmidt-Trigger TC
12.0
9.0
6.0
3.0
0.0
Vout
0.0
2.5
5.0
7.5
10.0
Vin

```
* Example 5
*
* OP circuit realization of an ideal diode
*
N1 0 1 2 0 opmod
Rdiode 1 2 (i=(0,0)(0.4,0)(0.6,1.05e-4)(0.65,7.2e-4)(0.7,4.9e-3))
Vin 1 0 ((-20,1):+)
*
* graphic control lines
.x_name Vin(volt)
.y_name Iin(mA)
.title Ideal Diode Realization
.x_axis -20 20
.y_axis -0.02 0.02
*
* canonical pwl model of operation amplifier
.model opmod ((i,v):P1=(0,0,0,0,0);P2=(0,0,0,500,-500);
$Bd=(1,0,-0.015,1,0,0.015))
.end
```

```
CANONICAL PWL DC ANALYSIS : FINDING DC OPERATING POINT

************************************************************
x[0]=v(Rdiode)
x[1]=v1(N1)
x[2]=i2(N1)
************************************************************
enter the initial point
v(Rdiode)= 0
v1(N1)= 0
i2(N1)= 0
the solution is
v(Rdiode)=-3.500e+01
v1(N1)=2.000e+01
i2(N1)=0.000e+00
continue tracing present branch of solution curve? y/n
n
try another branch of solution curve? y/n
n

END OF CANONICAL PWL DC ANALYSIS FOR DC OPERATING POINT

************************************************************
x[0]=v(Rdiode)
x[1]=v1(N1)
x[2]=i2(N1)
************************************************************
enter the output variable : x/z/i
i
```


Ideal Diode Realization
20.0
10.0
0.0
-10.0
-20.0
-20.0
-10.0
0.0
10.0
20.0
Iin(mA) (1E-3)
Vin(Volt)

```
* Example 6
*
* NMOS depletion-load inverter
*
* driver trsnsistor
N1 3 0 2 0 nmod
* depletion load transistor
N2 4 2 1 2 dep
Vdd 1 0 5
Vin 3 0 ((0,5):+)
Vy 4 2 0
*
* graphic control lines
.x_name Vin(Volt)
.y_name Vout(Volt)
.title NMOS Inverter TC
.x_axis 0 5
.y_axis 0 5
*
* canonical pwl model for MOS transistor
.model nmod ((i,i):P1=(0,0,0,0,0,0,0);P2=(-3.305e-5,7.92e-6,
$7.279e-5,-4.972e-5,-2.103e-5,2.03e-6,1.26e-8);Bd=(0.8175,-1,2.105,
$1.02,-1,1.465,23.41,1,69,1772,1,2232))
.model dep ((i,i):P1=(0,0,0,0,0,0,0);P2=(-9.29e-6,7.92e-6,
$7.279e-5,-4.972e-5,-2.103e-5,2.03e-6,1.26e-8);Bd=(0.8175,-1,-0.3473,
$1.02,-1,-1.59,23.41,1,-1.22,1772,1,-3083))
.end
```

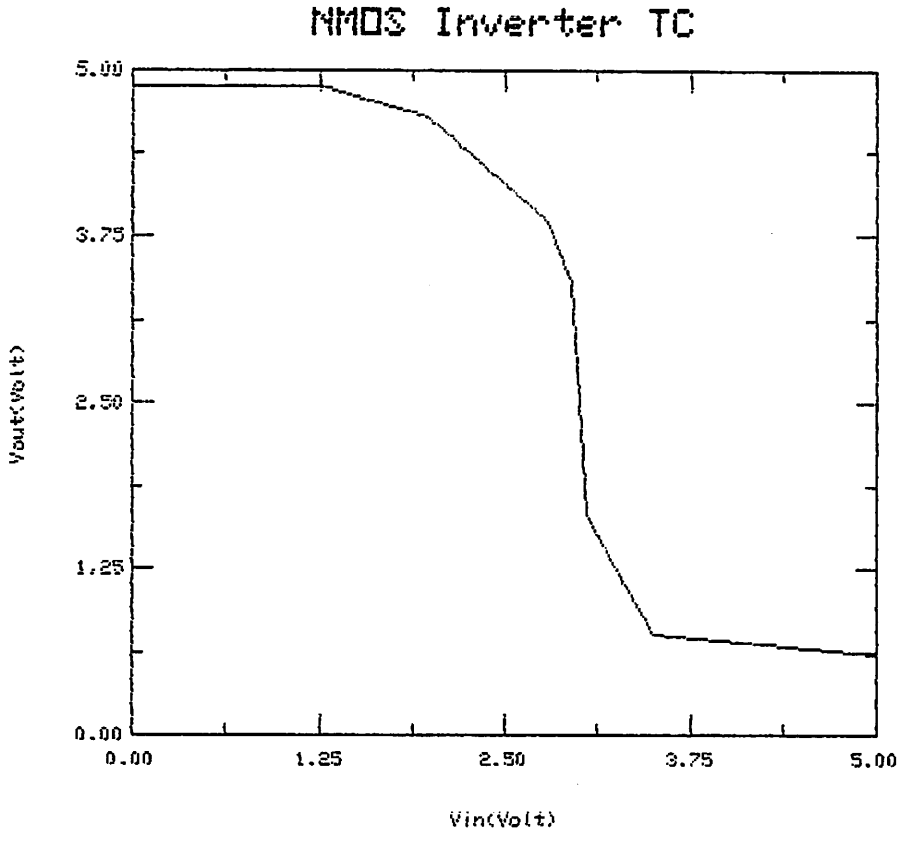
CANONICAL PWL DC ANALYSIS : FINDING DC OPERATING POINT

```
**************************************************************
z[0]=v1(N1)
z[1]=v2(N1)
z[2]=v1(N2)
z[3]=v2(N2)
**************************************************************
enter the initial point
v1(N1)= 0
v2(N1)= 0
v1(N2)= 0
v2(N2)= 0
the solution is
v1(N1)=0.000e+00
v2(N1)=4.874e+00
v1(N2)=0.000e+00
v2(N2)=1.259e-01
continue tracing present branch of solution curve? y/n
n
try another branch of solution curve? y/n
n
```

END OF CANONICAL PWL DC ANALYSIS FOR DC OPERATING POINT

```
**************************************************************
z[0]=v1(N1)
z[1]=v2(N1)
z[2]=v1(N2)
z[3]=v2(N2)
**************************************************************
enter the output variable : x/z/i
z
output=z[?]
1
```

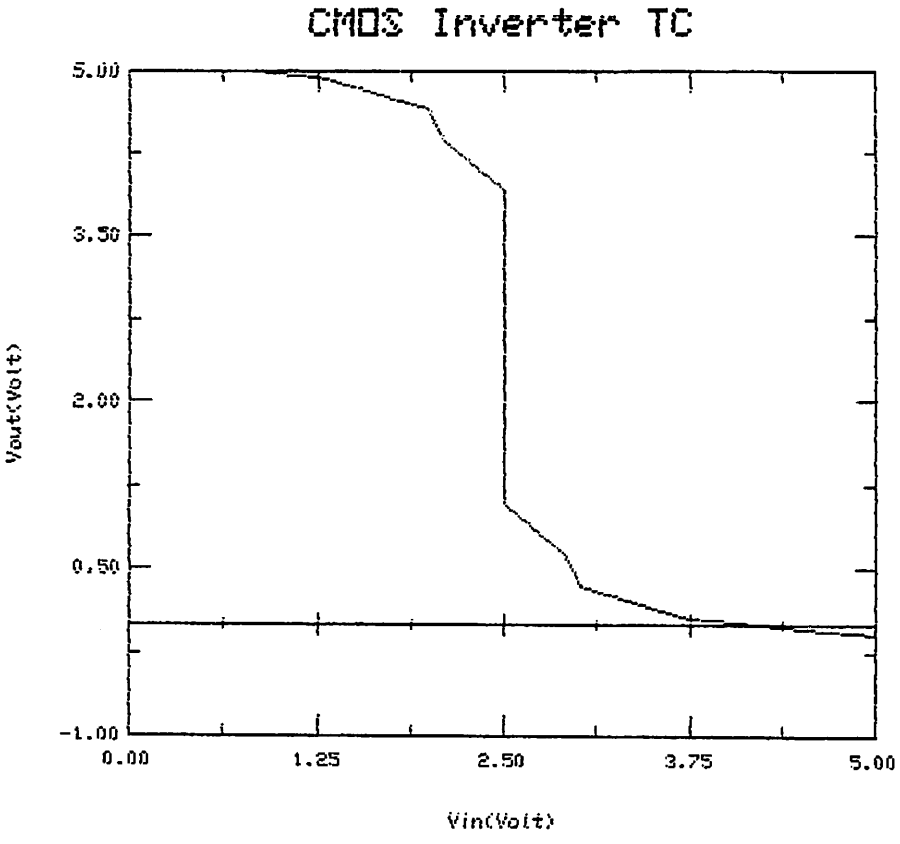
C>
φ


NMOS Inverter TC
5.00
3.75
2.50
1.25
0.00
Vout(Volt)
0.00
1.25
2.50
3.75
5.00
Vin(Volt)

```
* Example 7
*
* CMOS inverter circuit
*
* n-channel driving transistor
N1 2 0 3 0 nmod
* p-channel load transistor
N2 1 2 1 3 nmod
Vdd 1 0 5
Vin 2 0 ((0,5):+)
*
* graphic control lines
.x_name Vin(Volt)
.y_name Vout(Volt)
.title CMOS Inverter TC
.x_axis 0 5
.y_axis -1 5
*
* canonical pwl model of MOS transistor
.model nmod ((i,i):P1=(0,0,0,0,0,0,0);P2=(-3.305e-5,7.92e-6,
$7.279e-5,-4.972e-5,-2.103e-5,2.03e-6,1.26e-8);Bd=(0.8175,-1,2.105,
$1.02,-1,1.465,23.41,1,69,1772,1,2232))
.end
```

```
CANONICAL PWL DC ANALYSIS : FINDING DC OPERATING POINT

************************************************************
z[0]=v1(N1)
z[1]=v2(N1)
z[2]=v1(N2)
z[3]=v2(N2)
************************************************************
enter the initial point
v1(N1)= 1
v2(N1)= 2
v1(N2)= 3
v2(N2)= 4
the solution is
v1(N1)=0.000e+00
v2(N1)=5.089e+00
v1(N2)=5.000e+00
v2(N2)=-8.882e-02
continue tracing present branch of solution curve? y/n
n
try another branch of solution curve? y/n
n

END OF CANONICAL PWL DC ANALYSIS FOR DC OPERATING POINT

************************************************************
z[0]=v1(N1)
z[1]=v2(N1)
z[2]=v1(N2)
z[3]=v2(N2)
************************************************************
enter the output variable : x/z/i
z
output=z[?]
1
```

C>
φ


CMOS Inverter TC
5.00
3.50
2.00
0.50
-1.00
Vout(Volt)
0.00
1.25
2.50
3.75
5.00
Vin(Volt)

APPENDIX : SOURCE CODE LISTINGS

1. **DC Operating Point**

   pwldc.c, pwldc2.c, pwldc3.c, pwldc4.c, pwldc5.c, pwldc6.c

2. **Driving-Point and Transfer Characteristics**

   pwl1.c, pwl2.c, pwl3.c, pwl4.c, pwl5.c, pwl6.c

```
#define ELEM 80
#define MODEL 30
#define LAT 60
#define SEMI 30
#define DIM 1000
#define DIN 2000
#define DIP 60
#define DIQ 2000
```

```
#include "stdio.h"
#include "nport.h"
#include "pwl.h"

/*****************************************************************************/
/*****************************************************************************/

char *xx[LAT];   /* mapping table between the lattice structure          */
                 /* variables and the branch voltages and currents       */
char *zz[100];   /* mapping table between the semi-lattice variables     */
                 /* and the branch voltages and currents                 */
int n;           /* circuit dimension                                    */
int ix=0;        /* # of elements with lattice structure                 */
int jx=0;        /* # of elements with semi-lattice structure            */
int *im;         /* # of breakpoints in the lattice structure variables */
int *jm;         /* # of boundaries in the semi-lattice variables        */
int *in;         /* # of ports in each multiport element                 */
int *iz;         /* branch indices of the lattice structure elements     */
int *jz;         /* branch indices of the semi-lattice elements          */
int *dim;        /* dim[i] = total number of breakpoints up to the       */
                 /* (i-1)-th lattice structure elements                  */
int *din;        /* din[i] = total number of partition boundary          */
                 /* parmeters up to the (i-1)-th semi-lattice structure */
                 /* elements                                             */
int *dip;        /* dip[i] = total number of branches (or ports) up to   */
                 /* the (i-1)-th semi-lattice structure elements         */
int *diq;        /* diq[i] = total number of partition boundaries up to */
                 /* the (i-1)-th semi-lattice structure elements         */
int ns;          /* number of dc operating points                        */
struct INLINE branch[ELEM];
                 /* array of element record                              */
struct B_VECTOR branch_vector[ELEM];
                 /* array for branch indices                             */
double *P,*Q,*s; /* grneralized hybrid equation parameters               */
double *x;       /* lattice structure variables                          */
double *z;       /* semi-lattice structure variables                     */
double tp,tn;    /* positions for neighboring breakpoints                */

/* parameters for canonical piecewise-linear equation */
double *B,*a,*r;
double *beta,*c;
double *h,*alpha;

/*****************************************************************************/
/*****************************************************************************/

main(argc,argv)
int argc;
char *argv[];
{
    char ch[2];
    int i;
    double *ss,*BB;
    FILE *gp;

    printf("\nCANONICAL PWL DC ANALYSIS : FINDING DC OPERATING POINT\n\n");
```

```
    /* formulate the circuit to the canonical pwl equation */
    pwl_formulation(argc,argv,&gp);

    /* ss (resp.; BB) stores original s (resp.; B) before tracing */
    /* the solution curve since s and BB will change values after */
    /* computing each breakpoint in the solution curve; they have */
    /* to be restored to the original values if it is necessary   */
    /* to trace another branch of solution curve                  */
    callocd(n,&ss,"ss");
    callocd(n*n,&BB,"BB");
    for (i=0;i<n;i++)
        ss[i]=s[i];
    for (i=0;i<n*n;i++)
        BB[i]=B[i];

    ch[0]='y';
    ns=0;

    /* trace branch(es) of solution curve; a dc operating point is   */
    /* obtained if t=1 is reached in the parametrized solution curve */
    while (ch[0]=='y')
    {
        /* reset s and B */
        for (i=0;i<n;i++)
            s[i]=ss[i];
        for (i=0;i<n*n;i++)
            B[i]=BB[i];

        /* start from t=0 */
        tp=0.0;

        /* get the initial point of the solution curve at t=0 */
        initial_point(s);

        /* trace the solution curve by calculating each breakpoint */
        /* using the Generalized Breakpoint Hopping algorithm      */
        pwl_computation(gp);

        printf("try another branch of solution curve? y/n\n");
        scanf("%1s",ch);
    }

    /* print EOF in the output file */
    fputs("&&&\n",gp);

    fclose(gp);
    printf("\nEND OF CANONICAL PWL DC ANALYSIS FOR DC OPERATING POINT\n\n");
}

/*****************************************************************************/
/* Formulate the pwl resistive circuit to the canonical pwl equation       */
/*        f(x) = -(a+t0*r) + B*x + sum (c_ji*|x_j - beta_i|)                 */
/*                 + sum (h_i*|(alpha_i,z) - gama_i|)                        */
/*             = 0                                                           */
/* (see Eq.(3.1) in reference [1])                                           */
/*****************************************************************************/
```

```
pwl_formulation(argc,argv,gp)
int argc;
char *argv[];
FILE **gp;
{
    FILE *fp,*fopen();
    int mn;
    char *model[MODEL],*calloc();

    /* open and read the input spice file */
    open_spice_file(argc,argv,&fp,gp);

    /* get the generalized hybrid equation             */
    /*          P*v + Q*i + s = 0                      */
    /* for the linear n-port formed by extracting each */
    /* nonlinear element in the circuit                */
    n=ELEM;
    mn=MODEL;
    n_port(fp,model,branch,branch_vector,&P,&Q,&s,&n,&mn);

    fclose(fp);

    /* pre-allocate space for each parameter in the */
    /* canonical pwl equation                       */
    alloc_1();

    /* decode the model description of each nonlinear resistor */
    /* to the canonical pwl representation                     */
    decode_pwl();

    /* check whether a multiport pwl model description possesses the */
    /* the lattice structure partition boundaries; if so, append it  */
    /* to the lattice structure part in the canonical pwl equation   */
    lattice();

    /* re-allocate spaces for each parameter ( pre-allocated in */
    /* alloc_1() ) for a more exact allocation once the exact   */
    /* dimension has been found                                 */
    alloc_2();

    /* print the mapping table */
    print_table();

    /* check point for circuit dimension */
    check_dim();

    /* partition the B matrix into parts of lattice */
    /* and semi-lattice structure                   */
    partition();

}

/*****************************************************************************/
/* Print the mapping table between the equation variable x in the canonical*/
/* pwl equation and the circuit variables (voltage and currents in the     */
```

```
/* circuit).                                                                  */
/****************************************************************************/

print_table()
{
    int i,j;

    printf("*************************************************************\n");
    for (i=0;i<ix;i++)
        printf("x[%d]=%s\n",i,xx[i]);
    for (i=0;i<jx;i++)
        for (j=0;j<in[i];j++)
            printf("z[%d]=%s\n",dip[i]+j,zz[dip[i]+j]);
    printf("*************************************************************\n");
}

/****************************************************************************/
/* Check whether the circuit dimension                                      */
/*               n =   ix          (lattice structure branches)             */
/*                   + dip[jx]     (semi-lattice structure branches)        */
/****************************************************************************/

check_dim()
{
    int j;

    j=ix+dip[jx];
    if (j!=n)
    {
        printf("INCORRECT DIMENSION J=%d\n",j);
        exit();
    }
}

/****************************************************************************/
/* Read the input spice file xx..x.spc and open the output file xx..x.op    */
/* into which the computed dc operating point(s) will be written.           */
/****************************************************************************/

open_spice_file(argc,argv,fp,gp)
int argc;
char *argv[];
FILE **fp,**gp;
{
    char line[12];
    FILE *fopen();

    if (argc!=2)        /* incorrect command line */
        exit_message("PWLDC SPICE_FILE");

    else
    {
        /* open the spice file */
        sprintf(line,"%s.spc",*++argv);
        if ((*fp=fopen(line,"r"))==NULL)
        {
```

```
            printf("CAN'T OPEN SPICE_FILE %s\n",line);
            exit();
        }

        /* open the output file */
        sprintf(line,"%s.op",*argv);
        if ((*gp=fopen(line,"w"))==NULL)
        {
            printf("CAN'T OPEN OUTPUT_FILE %s\n",line);
            exit();
        }
    }
}

/*****************************************************************************/
/* Pre-allocate space for each parameter in the canonical pwl equation.      */
/*****************************************************************************/

alloc_1()
{
    calloci(LAT,&im,"im");
    calloci(SEMI,&jm,"jm");
    calloci(SEMI,&in,"in");
    calloci(LAT,&iz,"iz");
    calloci(DIP+SEMI,&jz,"jz");
    calloci(LAT,&dim,"dim");
    calloci(SEMI,&din,"din");
    calloci(SEMI,&dip,"dip");
    calloci(SEMI,&diq,"diq");
    callocd(n,&x,"x");
    callocd(n,&z,"z");
    callocd(n,&a,"a");
    callocd(n,&r,"r");
    callocd(n*n,&B,"B");
    callocd(DIM,&beta,"beta");
    callocd(n*DIM,&c,"c");
    callocd(n*DIQ,&h,"h");
    callocd(DIN,&alpha,"alpha");
}

/*****************************************************************************/
/* Decode each nonlinear element ( connected across the external ports of    */
/* the linear n-port ) to the canonical pwl representation.                  */
/*****************************************************************************/

decode_pwl()
{
    int i,j;

    /* decode each element across each external port of the n-port */
    for (i=0;i<n;i++)
    {
        j=(branch_vector+i)->a;          /* in the j-th element line */
        switch ((branch+j)->port)
        {
            case 1 :
```

```
                port_1(branch,j,i);
                break;  /* 1-port element */
            case 2 :
                port_2(branch,branch_vector,j);
                break;  /* 2-port element */
        }
    }
}

/*****************************************************************************/
/* Re-allocate spaces for the parameters in the canonical pwl equation       */
/* with the exact required dimension once the exact size for each parameter  */
/* has been found.                                                           */
/*****************************************************************************/

alloc_2()
{
    /* check whether enough spaces in the preliminary allocation */
    /* alloc_1()                                                 */
    check_space();

    ralloci(&im,ix,LAT);
    ralloci(&jm,jx,SEMI);
    ralloci(&in,jx,SEMI);
    ralloci(&iz,ix,LAT);
    ralloci(&jz,dip[jx]+jx+1,DIP+SEMI);
    ralloci(&dim,ix+1,LAT);
    ralloci(&din,jx+1,SEMI);
    ralloci(&dip,jx+1,SEMI);
    ralloci(&diq,jx+1,SEMI);
    rallocd(&x,ix,n);
    rallocd(&z,dip[jx],n);
    rallocd(&beta,dim[ix],DIM);
    rallocd(&c,n*dim[ix],n*DIM);
    rallocd(&h,n*diq[jx],n*DIQ);
    rallocd(&alpha,din[jx],DIN);
}

/*****************************************************************************/
/* Check whether enough spaces are allocated in the preliminary allocation */
/* alloc_1() for each parameter in the canonical pwl equation.               */
/*****************************************************************************/

check_space()
{
    if (ix>LAT)
    {
        printf("NOT ENOUGH SPACE; LAT should be no less than %d\n",ix);
        exit();
    }
    if (jx>SEMI)
    {
        printf("NOT ENOUGH SPACE; SEMI should be no less than %d\n",jx);
        exit();
    }
    if (dim[ix]>DIM)
```

```
    {
        printf("NOT ENOUGH SPACE; DIM should be no less than %d\n",dim[ix]);
        exit();
    }
    if (din[jx]>DIN)
    {
        printf("NOT ENOUGH SPACE; DIN should be no less than %d\n",din[jx]);
        exit();
    }
    if (dip[jx]>DIP)
    {
        printf("NOT ENOUGH SPACE; DIP should be no less than %d\n",dip[jx]);
        exit();
    }
    if (diq[jx]>DIQ)
    {
        printf("NOT ENOUGH SPACE; DIQ should be no less than %d\n",diq[jx]);
        exit();
    }
}
```

```
#include "stdio.h"
#include "nport.h"
#include "pwl.h"

extern int n,ix,jx,*iz,*jz,*dip,*in,*diq,*din;
extern int *im,*jm,*dim;
extern char *xx[],*zz[];
extern double *B,*P,*Q,*s,*r;
extern double *c,*beta;
extern double *h,*alpha;
extern double *x,*z;

/*************************************************************************/
/* Partition the B matrix into two parts with lattice and semi-lattice   */
/* structure respectively.                                               */
/*************************************************************************/

/* the ij-th entry of the matrices (i) B (ii) Bx is expressed as */
/*      (i) B[i*n+j]            (ii) Bx[i*n+j]      respectively */
/*                                                               */
/* the index  i = jz[dip[j]+j+k+1]  denotes that the k-th branch */
/* of the j-th multiport element is connected across the i-th    */
/* port of the linear n-port                                     */

partition()
{
    int i,j,k;
    double *Bx;

    callocd(n*n,&Bx,"Bx");

    /* B matrix before partition */
    for (j=0;j<n;j++)
        for (i=0;i<n;i++)
            Bx[i*n+j]=B[i*n+j];

    /* lattice structure part : the iz[j]-th port is */
    /* re-numbered as the j-th port                  */
    for (j=0;j<ix;j++)
        for (i=0;i<n;i++)
            B[i*n+j]=Bx[i*n+iz[j]];

    /* semi-lattice structure part : the jz[dip[j]+j+k+1]-th */
    /* port is re-numbered as the (ix+dip[j]+k)-th port      */
    for (j=0;j<jx;j++)
    {
        for (i=0;i<n;i++)
            for (k=0;k<in[j];k++)
                B[i*n+ix+dip[j]+k]=Bx[i*n+jz[dip[j]+j+k+1]];
    }
    cfree(Bx);
}

/*************************************************************************/
/* Decode the 1-port element ( which is connected across the i-th port of  */
/* the linear n-port and is in the j-th element line ) for canonical pwl   */
```

```
/* equation.                                                                  */
/*****************************************************************************/

port_1(branch,j,i)
int j;          /* element index in the input file */
int i;          /* branch index for this 1-port element */
struct INLINE branch[]; /* array of element record */
{
    int m,k;
    char *calloc();
    double a[12],*zx,*zy;

    iz[ix]=i;    /* store the port index for this element */
    xx[ix]=calloc(20,sizeof(char));

    /* decode the 1-port element for element characteristic */
    decode_1((branch+j)->relation,im+ix,a);

    callocd(n,&zx,"zx");
    callocd(n,&zy,"zy");

    /* voltage-controlled case */
    if (find_index("i=",(branch+j)->relation))0)
    {
        for (k=0;k<n;k++)
        {
            zx[k]=P[k*n+i];
            zy[k]=Q[k*n+i];
        }
        sprintf(xx[ix],"v(%s)",(branch+j)->name);
    }

    /* current-controlled case */
    if (find_index("v=",(branch+j)->relation))0)
    {
        for (k=0;k<n;k++)
        {
            zx[k]=Q[k*n+i];
            zy[k]=P[k*n+i];
        }
        sprintf(xx[ix],"i(%s)",(branch+j)->name);
    }

    /* dim[i] = sum of # of breakpoints for the j-th 1-port */
    /* element with j=0,1,2,...,i-1                         */
    dim[ix+1]=dim[ix]+im[ix];

    /* substitute the pwl model into the hybrid equation */
    for (k=0;k<n;k++)
    {
        s[k]=s[k]+zy[k]*a[0];
        B[k*n+i]=zx[k]+zy[k]*a[1];
    }
    for (m=0;m<im[ix];m++)
    {
      for (k=0;k<n;k++)
```

```
            c[(dim[ix]+m)*n+k]=a[2+2*m]*zy[k];
        beta[dim[ix]+m]=a[3+2*m];
    }

    ix++;                /* increment # of 1-port elements */

    cfree(zx);
    cfree(zy);
}

/*****************************************************************************/
/* Read the starting point for the solution curve and determine the tracing*/
/* vector in the range space.                                                */
/*****************************************************************************/

initial_point(a)
double a[];
{
    int i,j,k;
    double fabs();

    /* read the starting point */
    printf("enter the initial point\n");
    for (i=0;i<ix;i++)           /* lattice structure part */
    {
        printf("%s= ",xx[i]);
        scanf("%lf",&x[i]);
    }
    for (j=0;j<jx;j++)           /* semi-lattice structure part */
        for (k=0;k<in[j];k++)
        {
            printf("%s= ",zz[dip[j]+k]);
            scanf("%lf",&z[dip[j]+k]);
        }

    /* find the tracing vector */
    r_vector(a);
}

/*****************************************************************************/
/* Find the tracing vector r = -f(x0), where x0 is the starting point and  */
/*      f(x) = a + B*x + sum c_ji*| x_j - beta_i |                           */
/*                     + sum  h_i*| <alpha_i,z> - gama_i |                   */
/* is the canonical pwl function.                                            */
/*****************************************************************************/

r_vector(a)
double a[];
{
    int i,j,k,l;
    double cx,dx;

    for (i=0;i<n;i++)
    {
        /* calculate a+B*x */
        cx=a[i];
```

```
        for (j=0;j<ix;j++)              /* lattice structure part */
            a[i]=a[i]+B[i*n+j]*x[j];
        for (j=0;j<jx;j++)              /* semi-lattice structure part */
            for (k=0;k<in[j];k++)
                a[i]+=B[i*n+ix+dip[j]+k]*z[dip[j]+k];

        /* add the lattice structure part sum c_ji*|x_j - beta_i| */
        for (j=0;j<ix;j++)
            for (k=0;k<im[j];k++)
                a[i]=a[i]+c[(dim[j]+k)*n+i]*fabs(x[j]-beta[dim[j]+k]);

        /* add the semi-lattice structure part          */
        /*              sum h_i |(alpha_i,z) - gama_i|  */
        for (j=0;j<jx;j++)
            for (k=0;k<jm[j];k++)
            {
                dx=0;
                for (l=0;l<in[j];l++)
                    dx=dx+alpha[din[j]+k*(in[j]+1)+l]*z[dip[j]+l];
                dx=fabs(dx-alpha[din[j]+k*(in[j]+1)+in[j]]);
                a[i]=a[i]+h[(diq[j]+k)*n+i]*dx;
            }

        r[i] = -a[i];
        a[i]=a[i]-cx;
    }
}

/*****************************************************************************/
/* Decode the 1-port element model.                                          */
/*****************************************************************************/

decode_1(model,m,a)

char *model;    /* model string */
int *m;         /* # of breakpoints */
double a[];     /* numerical information in the model */

{
    char xc[21],yc[21];
    int nn=0,i,k,l;
    double xd[10],yd[10];
    double atof();

    l=strlen(model)+1;
    k=find_index("=",model);
    model=model+k+1;
    while (*model!=')' && k++<l)
    {
        if (*model==' ') model++;
        if (*model=='(' && nn<10)
        {

            /* get the x-component of the breakpoint */
            i=0;
            while (*model!=',' && i<21)
```

```
            xc[i++] = *(++model);
        if (i>=21) exit_message("TOO MANY DIGITS IN PWL 1-PORT MODEL");
        xc[i-1]='\0';
        model++;
        xd[nn]=atof(xc);

        /* get the y-component of the breakpoint */
        i=0;
        while (*model!=')' && i<20)
            yc[i++] = *(model++);
        if (i>=20) exit_message("TOO MANY DIGITS IN PWL 1-PORT MODEL");
        yc[i]='\0';
        model++;
        yd[nn++]=atof(yc);
    }
    if (nn>=10)
        exit_message("TOO MANY BREAKPOINTS IN PWL 1-PORT MODEL");
}

/* incorrect model description */
if (k>=1)
    exit_message("MISSING ')' IN THE MODEL");

*m=nn-2;    /* # of breakpoints */

/* construct the canonical pwl representation */
canonical_pwl(*m,xd,yd,a);
}

/*****************************************************************************/
/* Construct the canonical piecewise-linear representation for a             */
/* 1-dimensional pwl function.                                               */
/*****************************************************************************/

canonical_pwl(m,xd,yd,a)

int m;              /* # of breakpoints */
double xd[],yd[];   /* breakpoins */
double a[];         /* parameters for canonical pwl representation */

{
    int i;
    double *slope;
    double fabs();

    callocd(m+1,&slope,"slope");

    /* calculate the slope of each segment */
    for (i=0;i<=m;i++)
        slope[i]=(yd[i+1]-yd[i])/(xd[i+1]-xd[i]);

    /* construct the parameters in the canonical pwl representation */
    a[1]=(slope[m]+slope[0])/2;
    for (i=0;i<m;i++)
    {
        a[2+2*i]=(slope[i+1]-slope[i])/2;
```

```
        a[3+2*i]=xd[i+1];
    }
    a[0]=yd[0]-a[1]*xd[0];
    for (i=0;i<m;i++)
        a[0]=a[0]-a[2+2*i]*fabs(a[3+2*i]-xd[0]);
    cfree(slope);
}
```

```
#include "stdio.h"
#include "nport.h"
#include "pwl.h"

extern int n,jx,*jm,*in,*jz,*dip,*diq,*din;
extern double *P,*Q,*s,*B;
extern double *h,*alpha;
extern char *zz[];

/*****************************************************************************/
/* Decode the 2-port model into a 2-dimensional canonical piecewise-linear */
/* equation.                                                                 */
/*****************************************************************************/

decode_2(model,nn,a1,a2,bd)
char model[];   /* 2-port model */
int *nn;        /* # of partition boundaries */
double a1[];    /* coefficients of 1st canonical pwl equation */
double a2[];    /* coefficients of 2nd canonical pwl equation */
double bd[];    /* coefficients of partition boundary equations */

{
    char *dx,*calloc();
    double atof();
    int j,k;

    dx=calloc(strlen(model)+1,sizeof(char));
    /* temporary storage for the model */

    /* decode the 1st canonical pwl equation */
    k=find_index("P1=(",model);
    strcpy(dx,model);
    strdel(dx,0,k+4);
    extract(dx,nn,a1);

    /* decode the 2nd canonical pwl equation */
    k=find_index("P2=(",dx);
    dx=dx+k+4;
    extract(dx,nn,a2);

    /* decode the partition boundaries */
    k=find_index("Bd=(",dx);
    dx=dx+k+4;
    extract(dx,&j,bd);
}

/*****************************************************************************/
/*  Extract the numerical data from ASCII model equation.                    */
/*****************************************************************************/

extract(code,nn,a)
char *code;     /* ASCII code of numerical information in the model */
int *nn;        /* # of partition boundaries */
double a[];     /* decoded numerical data */
{
    char st[21];
```

```
    double atof();
    int i,j=0,l=0;

    while (find_index(",",code)<find_index(")",code) &&
        find_index(",",code))=0)
    {
        l++;

        /* too many data in the model */
        if (l>30)
            exit_message("TOO MANY PARAMETERS IN THE MODEL OF PWL 2-PORT");

        i=0;
        if (*code==',')
            code++;
        while (*code!=',' && *code!=')' && i<20)
            st[i++] = *(code++);
        if (i>=20)
            exit_message("TOO MANY DIGITS IN PWL 2-PORT MODEL");
        st[i]='\0';
        a[j++]=atof(st);
    }
    *nn=j-3;
}

/*****************************************************************************/
/* Decode the 2-port element ( which is in the j-th element line ) for its */
/* canonical piecewise-linear equation.                                      */
/*****************************************************************************/

port_2(branch,branch_vector,j)
struct INLINE branch[];         /* array of element record */
struct B_VECTOR branch_vector[];/* array for branch indices */
int j;                          /* branch index of the 2-port element */

{
    int i,k;
    double a1[13],a2[13],bd[30],*zx1,*zx2,*zy1,*zy2;
    char *calloc();

    if (chosen(jx,j)==-1 || jx==0)      /* check whether processed */
    {
        in[jx]=2;       /* port # = 2 */

        /* dip[i] = sum of ( # of ports in the j-th multiport element ) */
        /* for j = 0,1,2,...,i-1                                        */
        dip[jx+1]=dip[jx]+in[jx];

        /* get the branch indices for each port branch */
        /* of this multiport element                   */
        semi_index(branch_vector,j,jx);

        /* decode the 2-port pwl model */
        decode_2((branch+j)->relation,jm+jx,a1,a2,bd);

        /* diq[i] = sum of ( # of partition boundaries in the  */
```

```
        /* canonical pwl model of the j-th multiport element ) */
        /* for j=0,1,2,...,i-1                                 */
        diq[jx+1]=diq[jx]+jm[jx];

        /* din[i] = sum of ( # of parameters in the partition  */
        /* boundary equations of the j-th multiport element )  */
        /* for j=0,1,2,...,i-1                                 */
        din[jx+1]=din[jx]+jm[jx]*(in[jx]+1);

        callocd(n,&zx1,"zx1");
        callocd(n,&zy1,"zy1");
        callocd(n,&zx2,"zx2");
        callocd(n,&zy2,"zy2");

        /* get the semi-lattice structure variable definition      */
        /* and the corresponding canonical pwl equation parameters */
        semi_var(branch,branch_vector,j,zx1,zy1,zx2,zy2);

        /* find the canonical pwl equation parameters */
        for (i=0;i<n;i++)
        {
            s[i]=s[i]+zy1[i]*a1[0]+zy2[i]*a2[0];
            B[i*n+jz[dip[jx]+jx+1]]=zx1[i]+zy1[i]*a1[1]+zy2[i]*a2[1];
            B[i*n+jz[dip[jx]+jx+2]]=zx2[i]+zy1[i]*a1[2]+zy2[i]*a2[2];
        }
        for (i=0;i<jm[jx];i++)
        {
            for (k=0;k<=in[jx];k++)
                alpha[din[jx]+i*(in[jx]+1)+k]=bd[3*i+k];
            for (k=0;k<n;k++)
                h[(diq[jx]+i)*n+k]=zy1[k]*a1[3+i]+zy2[k]*a2[3+i];
        }
        jx++;           /* increment # of multiport elements */
        cfree(zx1);
        cfree(zy1);
        cfree(zx2);
        cfree(zy2);
    }
}

/*****************************************************************************/
/* Determine whether any branch in the 2-port element has been processed;  */
/* return -1 if it is the 1st branch in this 2-port element ever processed */
/*****************************************************************************/

int
chosen(j,ja)
int j,ja;
{
    int k=0;

    for (k=0;k<=j;k++)
        if (jz[dip[k]+k]==ja)
            return(k);
    if (k>j)
        return(-1);
```

```
}

/*****************************************************************************/
/* Determine the branch indices for 2-port elements.                         */
/*****************************************************************************/

semi_index(bv,ja,j)
struct B_VECTOR bv[];
int ja,j;
{
    int k;

    jz[dip[j]+j]=ja;
    for (k=0;k<n;k++)
        if ((bv+k)->a==ja)
        {
            /* 1st port of the j-th 2-port element is at port k */
            /* of the linear n-port                             */
            if ((bv+k)->b==1) jz[dip[j]+j+1]=k;

            /* 2nd port of the j-th 2-port element is at port k */
            /* of the linear n-port                             */
            if ((bv+k)->b==2) jz[dip[j]+j+2]=k;
        }
}

/*****************************************************************************/
/* Identify the element type for the 2-port element and construct the        */
/* corresponding canonical pwl equation parameters and the semi-lattice      */
/* structure variable table.                                                 */
/*****************************************************************************/

semi_var(branch,branch_vector,ja,zx1,zy1,zx2,zy2)
struct INLINE branch[];
struct B_VECTOR branch_vector[];
int ja;
double *zx1,*zy1,*zx2,*zy2;
{
    int i,l= -1;
    char *calloc();

    if (find_index("(i,i):",(branch+ja)->relation))>=0)
        l=0;    /* (1) voltage (2) voltage -controlled */
    if (find_index("(i,v):",(branch+ja)->relation))>=0)
        l=1;    /* (1) voltage (2) current -controlled */
    if (find_index("(v,i):",(branch+ja)->relation))>=0)
        l=2;    /* (1) current (2) voltage -controlled */
    if (find_index("(v,v):",(branch+ja)->relation))>=0)
        l=3;    /* (1) current (2) current -controlled */
    if (l<0 || l>3)
        exit_message("INCORRECT 2-PORT MODEL");

    /* find the variable table for 2-port element with    */
    /* semi-lattice structure canonical pwl model and the */
    /* corresponding canonical pwl equation parameters    */
    for (i=0;i<2;i++)
```

```
        zz[dip[jx]+i]=calloc(20,sizeof(char));
    if (l<2)
    {
        sprintf(zz[dip[jx]],"v1(%s)",(branch+ja)->name);
        if (l==0)
            sprintf(zz[dip[jx]+1],"v2(%s)",(branch+ja)->name);
        else sprintf(zz[dip[jx]+1],"i2(%s)",(branch+ja)->name);
        for (i=0;i<n;i++){
            zx1[i]=P[i*n+jz[dip[jx]+jx+1]];
            zy1[i]=Q[i*n+jz[dip[jx]+jx+1]];
            if (l==1)
            {
                 zx2[i]=Q[i*n+jz[dip[jx]+jx+2]];
                 zy2[i]=P[i*n+jz[dip[jx]+jx+2]];
            } else
            {
                 zx2[i]=P[i*n+jz[dip[jx]+jx+2]];
                 zy2[i]=Q[i*n+jz[dip[jx]+jx+2]];
            }
        }
    }
    if (l>=2)
    {
        sprintf(zz[dip[jx]],"i1(%s)",(branch+ja)->name);
        if (l==2)
            sprintf(zz[dip[jx]+1],"v2(%s)",(branch+ja)->name);
        else sprintf(zz[dip[jx]+1],"i2(%s)",(branch+ja)->name);
        for (i=0;i<n;i++)
        {
            zx1[i]=Q[i*n+jz[dip[jx]+jx+1]];
            zy1[i]=P[i*n+jz[dip[jx]+jx+1]];
            if (l==3)
            {
                zx2[i]=Q[i*n+jz[dip[jx]+jx+2]];
                zy2[i]=P[i*n+jz[dip[jx]+jx+2]];
            } else
            {
                zx2[i]=P[i*n+jz[dip[jx]+jx+2]];
                zy2[i]=Q[i*n+jz[dip[jx]+jx+2]];
            }
        }
    }
}
```

```
#include "stdio.h"
#include "pwl.h"

extern char *xx[];
extern int n,ix,jx,*in,*im,*jm,*dim,*dip,*diq,*din,ns;
extern double *alpha,*beta;
extern double *B,*r,*c;
extern double *x,*z,*h;
extern double tp,tn;
int i0,j0,k0,*iy,*jy;
double *xd,*zd,*xp[10],*zp[10];

/*****************************************************************************/
/* Implement the Generalized Breakpoint Hopping Algorithm to calculate each*/
/* breakpoint sequentially for tracing the solution curve of a canonical   */
/* pwl equation and finding the dc operating point. .                        */
/*****************************************************************************/

pwl_computation(gp)
FILE *gp;
{
    int q,s;
    double *d,*d1,fabs();

    calloci(diq[jx],&jy,"jy");
    callocd(n,&d,"d");
    callocd(n,&d1,"d1");
    callocd(ix,&xd,"xd");
    callocd(dip[jx],&zd,"zd");

    /* determine the initial region for the starting point */
    location_index();

    /* construct the initial Jacobian matrix and calculate */
    /* the initial direction vector d                      */
    ini_B_d(d);

    /* find the first breakpoint */
    if (next_point(gp,d,&q)==-1)
        return;

    /* tracing the solution curve */
    while (fabs(tn)<1.0e3)
    {
        /* find the Jacobian matrix and direction */
        /* vector in the next region              */
        B_and_d(d,d1,q);

        /* determine the correct direction vector in order */
        /* to continue tracing the solution curve          */
        d_vector(&s,q,d,d1);

        /* calculate each distance of the present breakpoint */
        /* to the neighboring region                         */
        if(increment(gp,d,s,&q)!=1)  /* unbounded region */
            break;
```

```
        /* calculate the next breakpoint; return -1 if quit */
        /* tracing the present branch of the solution curve */
        if (break_point(gp,s,d)==-1)
            break;
    }
    cfree(iy);
    cfree(jy);
    cfree(d);
    cfree(d1);
    cfree(xd);
    cfree(zd);
}

/*****************************************************************************/
/* Determine the region where the initial point located.                     */
/*****************************************************************************/

location_index()
{
    int i,j,k;
    double cx;

    calloci(ix,&iy,"iy");

    /* determine the region indices for lattice structure boundaries */
    for (i=0;i<ix;i++)
    {
        if ((x[i]-beta[dim[i]])<=0)
            iy[i]=0;
        else
        {
            if ((x[i]-beta[dim[i]+im[i]-1])>0) iy[i]=im[i];
            else
            {
                for (k=1;k<im[i];k++)
                    if ((x[i]-beta[dim[i]+k-1])*(x[i]-beta[dim[i]+k])<=0)
                    {
                        iy[i]=k;
                        break;
                    }
                if (k==im[i])
                    exit_message("INITIAL POINT IS IN UNDEFINED REGION");
            }
        }
    }

    /* determine the region indices for semi-lattice structure boundaries */
    for (j=0;j<jx;j++)
        for (k=0;k<jm[j];k++)
        {
            cx=0;
            for (i=0;i<in[j];i++)
                cx=cx+alpha[din[j]+k*(in[j]+1)+i]*z[dip[j]+i];
            if ((cx-alpha[din[j]+k*(in[j]+1)+in[j]])<0) jy[diq[j]+k]=0;
            else jy[diq[j]+k]=1;
```

```
        }
}

/*****************************************************************************/
/* Find the Jacobian matrix and the direction vector of the solution curve */
/* in the initial region.                                                    */
/*****************************************************************************/

ini_B_d(d)
double *d;      /* direction vector */
{
    int i,j,k,l,lx,*ipvt;
    double *Bx,*ux,rcond;

    /* construct the Jacobian matrix in the initial region */
    for (i=0;i<ix;i++)          /* lattice structure part */
        for (j=0;j<im[i];j++)
            for (k=0;k<n;k++)
            {
                if (j>=iy[i]) B[k*n+i]-=c[(dim[i]+j)*n+k];
                else B[k*n+i]+=c[(dim[i]+j)*n+k];
            }
    for (i=0;i<jx;i++)          /* semi-lattice structure part */
    {
        lx=0;
        if (i>0) for (l=0;l<i;l++) lx+=in[l];
        for (j=0;j<jm[i];j++)
            for (k=0;k<n;k++)
                for (l=0;l<in[i];l++)
                {
                    if (jy[diq[i]+j]==1)
                        B[k*n+ix+lx+l]+=alpha[din[i]+j*(in[i]+1)+l]
                                        *h[(diq[i]+j)*n+k];
                    else
                        B[k*n+ix+lx+l]-=alpha[din[i]+j*(in[i]+1)+l]
                                        *h[(diq[i]+j)*n+k];
                }
    }

    calloci(n,&ipvt,"ipvt");
    callocd(n,&ux,"ux");
    callocd(n*n,&Bx,"Bx");

    /* compute the direction vector by solving Bd=r */
    for (i=0;i<n;i++)
    {
        d[i]=r[i];
        for (j=0;j<n;j++)
            Bx[i*n+j]=B[i*n+j];
    }
    sgeco(Bx,n,ipvt,&rcond,ux); /* LU decomposition */
    sgesl(Bx,n,ipvt,d,0);
    cfree(ipvt);
    cfree(ux);
    cfree(Bx);
}
```

```
/*************************************************************************/
/* Find the first breakpoint after the starting point.                   */
/*************************************************************************/

next_point(gp,d,q)
FILE *gp;       /* output file pointer */
int *q;         /* boundary index; 0 for lattice and 1 for semi-lattice */
double *d;      /* direction vector */
{
    double fabs(),dti_t=1e5,dtj_t=1e5;
    double delta_t;

    /* calculate the distance to each lattice structure boundary */
    if (d_lat(1,&dti_t,d)==-1)
        return(-1);

    /* calculate the distance to each semi-lattice structure boundary */
    if (d_semi(1,0,0,0,&dtj_t,d)==-1)
        return(-1);

    /* classified as a cornor point if within the distance = 1e-8 */
    /* of a cornor point                                          */
    if (dti_t<1.0e-8 && dtj_t<1.0e-8)
    {
        printf("HIT A CORNER POINT ON LATTICE AND SEMI-LATTICE BOUNDARIES\n");
        printf("please try another initial point\n");
        return(-1);
    }

    /* compare the minimal distances from the lattice structure boundary */
    /* and the semi-lattice structure boundary and choose the minimum    */
    if (dti_t>dtj_t)
    {
        *q=1;
        delta_t=dtj_t;
    } else
    {
        *q=0;
        delta_t=dti_t;
    }

    tn=tp+delta_t;

    /* calculate the breakpoint */
    break_point(gp,1,d);
}

/*************************************************************************/
/* Calculate the distance of the present breakpoint to each lattice      */
/* structure boundary and the corresponding required amount of input for */
/* reaching that boundary. Choose the boundary with minimal amount of input*/
/* which will be the 1st lattice structure boundary hit by the solution  */
/* curve; return -1 if it goes to unbounded region without hitting any   */
/* boundary.                                                             */
/*************************************************************************/
```

```
d_lat(s,dti_t,d)
int s;
double *dti_t;
double *d;
{
    int i,m=0;
    double *dta,fabs();

    callocd(ix,&dta,"dta");

    for (i=0;i<ix;i++)
        if (fabs(d[i])>1.0e-9 && im[i]>0)
        {
            if ((s*d[i])>0)
            {
                if (iy[i]==im[i])    /* i-th component goes to + infinity */
                    dta[i]=1.0e5*fabs(d[i]);
                else
                    dta[i]=beta[dim[i]+iy[i]]-x[i];
                if (dta[i]<1.0e-15)
                {
                    printf("GET STUCKED IN A BOUNDARY\n");
                    printf("please try another initial point\n");
                    return(-1);
                }
            }
            else
            {
                if (iy[i]==0)        /* i-th component goes to - infinity */
                    dta[i]=1.0e5*fabs(d[i]);
                else
                    dta[i]=x[i]-beta[dim[i]+iy[i]-1];
            }
            dta[i]=dta[i]/fabs(d[i]); /* required amount of input for the */
                                     /* i-th component to hit a boundary */

            /* choose the minimum and determine the lattice */
            /* structure boundary index i0                  */
            if (*dti_t>dta[i])
            {
                *dti_t=dta[i];
                i0=i;
            }
        }

    for (i=0;i<ix;i++)
        if (fabs(dta[i]-*dti_t)<1.0e-17 && *dti_t<1.0e4 && d[i]!=0)
            m++;        /* present breakpoint is in the intersection */
                        /* of m lattice structure boundaries         */

    cfree(dta);
    if (m>1)
    {
        printf("HIT A CORNER POINT IN LATTICE STRUCTURE BOUNDARIES\n");
        printf("please try another initial point\n");
```

```
        return(-1);
    }
    else return(1);
}

/*************************************************************************/
/* Calculate the distance of the present breakpoint to each semi-lattice  */
/* structure boundary and the corresponding required amount of input for  */
/* reaching that boundary. Choose the boundary with minimal amount of input*/
/* which will be the first semi-lattice structure boundary hit by the     */
/* solution curve; return -1 when entering an unbounded region without    */
/* hitting any boundary.                                                  */
/*************************************************************************/

d_semi(s,q,j1,k1,dtj_t,d)
int s,q,j1,k1;
double *dtj_t,*d;
{
    int i,j,k,m=0;
    double *dtb,dx,dy,fabs();

    calloc(diq[jx],&dtb,"dtb");

    for (i=0;i<jx;i++)
    {
        for (j=0;j<jm[i];j++)
        {
            dx=0;
            dy=0;
            for (k=0;k<in[i];k++)
            {
                dx+=alpha[din[i]+j*(in[i]+1)+k]*d[ix+dip[i]+k];
                dy+=alpha[din[i]+j*(in[i]+1)+k]*z[dip[i]+k];
            }
            dy=alpha[din[i]+j*(in[i]+1)+in[i]]-dy;
            if (fabs(dy)<1.0e-8)
                m++;
            if (dx!=0)
            {
                dtb[diq[i]+j]=s*dy/dx;
                if (dtb[diq[i]+j]>0 && *dtj_t>dtb[diq[i]+j])
                {
                    if (q!=1 || j1!=i || k1!=j)
                    {
                        j0=i;
                        k0=j;
                        *dtj_t=dtb[diq[i]+j];
                    }
                }
            }
        }
    }
    cfree(dtb);
    if (m>1)
    {
        printf("HIT A CORNER POINT IN SEMI-LATTICE STRUCTURE BOUNDARIES\n");
```

```
        printf("please try another initial point\n");
        return(-1);
    } else return(1);
}

/*****************************************************************************/
/*****************************************************************************/
```

```
#include "stdio.h"
#include "pwl.h"

extern char *xx[],*zz[];
extern int n,ix,jx,*in,*im,*jm,*dim,*din,*dip,*diq;
extern double *alpha,*beta;
extern double *B,*r,*c;
extern double *h,*x,*z;
extern double tp,tn;
extern int i0,j0,k0,*iy,*jy;
extern int ns;
extern double *xd,*zd,*xp[],*zp[];

/*****************************************************************************/
/* Construct the Jacobian matrix and calculate the direction vector in the */
/* next region.                                                              */
/*****************************************************************************/

B_and_d(d,d1,q)
int q;          /* boundary index; 0 (resp,; i) for lattice */
                /* (resp.; semi-lattice) structure boundary */
double *d;      /* previous direction vector */
double *d1;     /* present direction vector */
{
    int i,j,*ipvt;
    double cx,*Bx,rcond,*ux;

    /* if crossing the lattice structure boundary */
    if (q==0)
    {   /* renew the region index for lattice structure component */
        if (d[i0]*(tn-tp))0)    /* crossing the right boundary */
            iy[i0]++;
        else                    /* crossing the left boundary */
            iy[i0]--;

        /* renew the Jacobian matrix */
        for (i=0;i(n;i++)
        {
            if (d[i0]*(tn-tp))0)
                B[i*n+i0]+=2*c[(dim[i0]+iy[i0]-1)*n+i];
            else
                B[i*n+i0]-=2*c[(dim[i0]+iy[i0])*n+i];
        }
    }

    /* if crossing the semi-lattice structure boundary */
    if (q==i)
    {
        /* renew the region index for semi-lattice structure component */
        if (jy[diq[j0]+k0]==1)
        {
            cx = -1;
            jy[diq[j0]+k0]=0;
        }
        else
        {
```

```
            cx=1;
            jy[diq[j0]+k0]=1;
        }
        for (i=0;i<n;i++)
            for (j=0;j<in[j0];j++)
                B[i*n+ix+dip[j0]+j]+=2*cx*h[(diq[j0]+k0)*n+i]
                                *alpha[din[j0]+k0*(in[j0]+1)+j];
    }

    calloci(n,&ipvt,"ipvt");
    callocd(n,&ux,"ux");
    callocd(n*n,&Bx,"Bx");

    /* find the new direction vector by solving B*d1=r */
    for (i=0;i<n;i++)
    {
        d1[i]=r[i];
        for (j=0;j<n;j++)
            Bx[i*n+j]=B[i*n+j];
    }
    sgeco(Bx,n,ipvt,&rcond,ux);
    sgesl(Bx,n,ipvt,d1,0);

    cfree(ipvt);
    cfree(ux);
    cfree(Bx);
}

/*****************************************************************************/
/* Determine the correct direction vector in order to continue tracing     */
/* the solution curve.                                                       */
/*****************************************************************************/

d_vector(s,q,d,d1)
int q;          /* boundary index; 0 for lattice and 1 for semi-lattice */
int *s;         /* direction sign; 1 for entering and -1 for returning */
double *d;      /* direction vector in the previous region */
double *d1;     /* direction vector in the present region */
{
    int i,k;
    double dx,dy;

    /* crossing the lattice structure boundary */
    if (q==0)
    {
        if ((d1[i0]*d[i0]*(tn-tp)))0)   /* d1 is entering */
            *s=1;
        else            /* d1 is returning to previous region */
            *s = -1;
    }

    /* crossing the semi-lattice structure boundary */
    if (q==1)
    {
        dx=0;
        dy=0;
```

```
        for (k=0;k<in[j0];k++)
        {
            dx+=d[ix+dip[j0]+k]*alpha[din[j0]+k0*(in[j0]+1)+k];
            dy+=d1[ix+dip[j0]+k]*alpha[din[j0]+k0*(in[j0]+1)+k];
        }
        if (((tn-tp)*dx*dy))0)          /* entering */
            *s=1;
        else                            /* returning */
            *s = -1;
    }

    /* renew the direction vector */
    for (i=0;i<n;i++)
        d[i]=d1[i];
}

/*****************************************************************************/
/* Calculate the distance between the present breakpoint and each boundary */
/* and determine the next region by finding the boundary with minimal      */
/* amount of required input for the solution curve to hit that boundary    */
/* from the present breakpoint. Return -1 when entering unbounded region   */
/* or passing a corner point.                                              */
/*****************************************************************************/

increment(gp,d,s,q)
FILE *gp;
int s;          /* direction vector sign */
int *q;         /* boundary index */
double *d;      /* direction vector */
{
    int j1,k1;
    double dti_t=1e5,dtj_t=1e5;
    double fabs(),delta_t;

    /* present breakpoint is located in the k0-th boundary */
    /* of the j0-th multiport element with semi-lattice    */
    /* structure partition boundaries in its canonical pwl */
    /* model                                               */
    j1=j0;
    k1=k0;

    /* calculate the distance to each lattice structure */
    /* boundary and choose the minimum; return -1 when  */
    /* entering unbounded region                        */
    if (d_lat(s,&dti_t,d)==-1)
        return(-1);

    /* calculate the distance to each semi-lattice      */
    /* structure boundary and choose the minimum; return*/
    /* -1 when entering unbounded region                */
    if (d_semi(s,*q,j1,k1,&dtj_t,d)==-1)
        return(-1);

    /* classified as a corner point if the present breakpoint is */
    /* within the distance =1e-8 of a corner point               */
    if (dti_t<1.0e-8 && dtj_t<1.0e-8)
```

```
    {
        printf("HIT A CORNER POINT ON LATTICE AND SEMI-LATTICE BOUNDARIES\n");
        printf("please try another initial point\n");
        return(-1);
    }

    /* compare the distances and choose the minimum */
    if (dti_t>dtj_t)    /* hit a semi-lattice structure boundary first */
    {
        *q=1;
        delta_t=dtj_t;
    }                   /* hit a lattice structure boundary first */
    else
    {
        *q=0;
        delta_t=dti_t;
    }

    /* entering unbouned region; 1.0e4 is an arbitrary large */
    /* number to represent an infinite distance              */
    if (delta_t>1.0e4)
    {
        /* a solution is located in this unbounded region */
        if ((tn-1)*(tn+s*delta_t-1)<0)
        {
            tp=tn;

            /* choose a nearby point as an ending point of the  */
            /* solution curve in the unbounded region such that */
            /* it passes a solution located at t=1 in this      */
            /* unbounded region                                 */
            if (tp>1)
                tn=0.0;
            else
                tn=2.0;

            /* print the solution; return -1 when no more solution */
            /* is needed in this branch of the solution curve      */
            if (print_sol(gp,s,d)==-1)
                return(-1);
        }

        printf("solution curve goes to infinity in the unbounded region\n");
        return(-1);
    }
    tp=tn;
    tn+=s*delta_t;
    return(1);
}

/*************************************************************************/
/* Print the dc operating point which is located at t=1 of the parametrized*/
/* solution curve. Return -1 when no more solution is needed is to quit    */
/* tracing the present branch of the solution curve.                       */
/*************************************************************************/
```

```
print_sol(gp,s,d)
FILE *gp;       /* output file pointer */
int s;          /* direction vector sign */
double *d;      /* direction vector */
{
    char line[30],ch[2];
    int i;
    double fabs();

    /* if t=1 is passed, a solution is found */
    if (((tn-1)*(tp-1)) <= 0)
    {
        printf("\nthe solution is\n");

        /* a fresh solution is found */
        if (get_sol(s,d)!=-1)
        {
            sprintf(line,"***************\n");
            fputs(line,gp);
            for (i=0;i<ix;i++)
            {
                sprintf(line,"%s=%.3e\n",xx[i],xd[i]);
                fputs(line,gp);
            }
            for (i=0;i<dip[jx];i++)
            {
                sprintf(line,"%s=%.3e\n",zz[i],zd[i]);
                fputs(line,gp);
            }
            ns++;       /* increment the # of solutions */
        }
        printf("continue tracing present branch of solution curve? y/n\n");
        scanf("%1s",ch);
        if (ch[0]=='y')
            return(1);
        else
            return(-1);
    }
    else return(1);
}

/*****************************************************************************/
/* Compute the dc operating point corresponding to t=1 in the solution     */
/* curve and check whether this solution has been found before. Return -1  */
/* if the solution is not new else return the total number of solutions    */
/* obtained so far.                                                        */
/*****************************************************************************/

get_sol(s,d)
int s;
double *d;
{
    int i,j,k;
    double sum,fabs();

    /* compute the lattice structure components of the solution */
```

```
    for (i=0;i<ix;i++)
    {
        xd[i]=x[i]+s*d[i]*fabs(1-tp);
        printf("%s=%.3e\n",xx[i],xd[i]);
    }

    /* compute the semi-lattice structure components of the solution */
    for (i=0;i<jx;i++)
        for (j=0;j<in[i];j++)
        {
            zd[dip[i]+j]=z[dip[i]+j]+s*d[ix+dip[i]+j]*fabs(1-tp);
            printf("%s=%.3e\n",zz[dip[i]+j],zd[dip[i]+j]);
        }

    /* check whether the solution is new; a fresh solution is found if */
    /* it differs by at least 1.0e-6 to any of the previous solutions  */
    /* xp[k][i] (resp.; zp[k][i]) denotes the i-th lattice structure   */
    /* (resp.; semi-lattice structure) component of the k-th solution  */
    for (k=0;k<ns;k++)
    {
        sum=0.0;
        for (i=0;i<ix;i++)
            sum+=fabs(xd[i]-xp[k][i]);
        for (i=0;i<dip[jx];i++)
            sum+=fabs(zd[i]-zp[k][i]);
        if (sum<1.0e-6)
            return(-1);
    }
    if (k==ns)
    {
        callocd(ix,&xp[ns],"xp");
        callocd(dip[jx],&zp[ns],"zp");
        for (i=0;i<ix;i++)
            xp[ns][i]=xd[i];
        for (i=0;i<dip[jx];i++)
            zp[ns][i]=zd[i];
    }
    return(ns);
}

/*****************************************************************************/
/* Find the next breakpoint.                                                 */
/*****************************************************************************/

break_point(gp,s,d)
FILE *gp;
int s;
double *d;
{
    int i,j;
    double fabs();

    /* check whether a solution at t=1 is reached */
    if (print_sol(gp,s,d)==-1)
        return(-1);
```

```
    /* calculate the lattice structure components */
    /* of the next breakpoint                     */
    for (i=0;i<ix;i++)
        x[i]+=d[i]*(tn-tp);

    /* calculate the semi-lattice structure components */
    /* of the next breakpoint                          */
    for (i=0;i<jx;i++)
        for (j=0;j<in[i];j++)
            z[dip[i]+j]+=d[ix+dip[i]+j]*(tn-tp);

    return(1);
}
```

```
#include "stdio.h"
#include "pwl.h"

extern int n,ix,jx,*iz,*jz,*im,*jm,*in,*dim,*din,*dip,*diq;
extern double *c,*beta;
extern double *h,*alpha;
extern char *xx[],*zz[];

/*****************************************************************************/
/* Determine whether the canonical pwl model of a multiport element        */
/* possess partition boundaries with lattice structure; if so, convert it  */
/* to a lattice structure element for computational efficiency. This new   */
/* converted lattice structure element is then appended to the lattice     */
/* structure part of the canonical pwl equation.                           */
/*****************************************************************************/

lattice()
{
    int i,*la,kx[20];
    double cx[20];

    calloci(jx,&la,"la");
    for (i=0;i<jx;i++)
    {
        la[i]=0;

        if (detect_lattice(i,kx)==-1)
            break;              /* semi-lattice structure element */
        else
            la[i]=1;            /* lattice structure element /

        /* convert semi-lattice structure parameters to lattice */
        /* structure parameters : alpha to beta and h to c      */
        move_alpha_h(i,kx,cx);

        /* convert the semi-lattice structure variables to */
        /* lattice structure variables                     */
        move_table(i,kx,cx);
    }

    /* reorder the semi-lattice parameters */
    re_order(la);

}

/*****************************************************************************/
/* Reorder the parameters in the semi-lattice structure part of the        */
/* canonical piecewise-linear equation.                                    */
/*****************************************************************************/

re_order(la)

int *la;        /* multiport element indices; 1 if lattice structure */
{
    int i=0,j,k;
```

```
    while (i<jx)
    {
        if (la[i]==0)   /* the i-th element possesses no lattice structure */
            i++;
        else            /* possesses lattice structure */
        {

            /* move the parameters of the last semi-lattice element to */
            /* the i-th element which has been detected to possess the */
            /* lattice structure and all the corresponding parameters  */
            /* have been moved to the lattice structure part           */
            la[i]=la[jx-1];
            in[i]=in[jx-1];
            jm[i]=jm[jx-1];
            for (j=0;j<in[i];j++)
                zz[dip[i]+j]=zz[dip[jx-1]+j];
            for (j=0;j<=in[i];j++)
                jz[dip[i]+i+j]=jz[dip[jx-1]+jx-1+j];
            for (j=0;j<jm[i];j++)
            {
                for (k=0;k<=in[i];k++)
                    alpha[din[i]+j*(in[i]+1)+k]
                        =alpha[din[jx-1]+j*(in[jx-1]+1)+k];
                for (k=0;k<n;k++)
                    h[(diq[i]+j)*n+k]=h[(diq[jx-1]+j)*n+k];
            }
            jx--;       /* decrement the # of semi-lattice elements */
        }
    }
}

/*****************************************************************************/
/* Order a set of m numbers cx[id[0]], cx[id[1]],...., cx[id[m-1]] to the  */
/* increasing order such that                                                */
/*              cx[ie[0]] < cx[ie[1]] < ... < cx[ie[m-1]]                    */
/*****************************************************************************/

order_index(m,id,cx,ie)

int m;          /* # of objects to be ordered */
int id[];       /* old index */
int ie[];       /* new index with increasing order */
double cx[];    /* array for the objects to be ordered */
{
    double y[10];
    int i,j,is[10];

    for (i=0;i<m;i++)
        y[i]=cx[id[i]];
    for (i=0;i<m;i++)
    {
        is[i]=0;
        for (j=0;j<m;j++)
            if (y[j]<=y[is[i]]) is[i]=j;
        y[is[i]]=1e6;
    }
```

```
    for (i=0;i<m;i++)
        ie[i]=id[is[i]];
}
/*****************************************************************************/
/* Check whether a multiport element possesses a lattice structure in its  */
/* canonical pwl model; if so, return 1.                                     */
/*****************************************************************************/

detect_lattice(i,kx)
int i,*kx;
{
    int j,k,num;

    /* detect whether possessing lattice structure which exists */
    /* only when it has a single nonzero coefficient in each of */
    /* its partition boundary equations                         */
    for (j=0;j<jm[i];j++)
    {
        num=0;
        for (k=0;k<in[i];k++)
            if (alpha[din[i]+j*(in[i]+1)+k]!=0)
            {
                num++;
                kx[j]=k;
            }
        if (num!=1)     /* possess no lattice structure */
            return(-1);
    }
    return(1);
}

/*****************************************************************************/
/* Modify the single nonzero coefficient term in canonical pwl equation    */
/*                      h_i |<alpha_i,z> - gama_i|                           */
/* to                                                                        */
/*                    p*h_i |x_j - gama_i/p|                                 */
/* where p is the nonzero coefficient and is in the j-th component of      */
/* alpha_i.                                                                  */
/*****************************************************************************/

move_alpha_h(i,kx,cx)
int i,*kx;
double *cx;
{
    int j,l;
    double fabs();

    for (j=0;j<jm[i];j++)
    {
        cx[j]=alpha[din[i]+j*(in[i]+1)+in[i]]/alpha[din[i]+j*(in[i]+1)+kx[j]];
        for (l=0;l<n;l++)
            h[(diq[i]+j)*n+l]=fabs(alpha[din[i]+j*(in[i]+1)+kx[j]])
                              *h[(diq[i]+j)*n+l];
    }
}
```

```
/*****************************************************************************/
/* Move the i-th multiport element parameters (which has been detected to  */
/* possess structure partition boundaries) to the lattice structure        */
/* part of the canonical pwl equation.                                     */
/*****************************************************************************/

move_table(i,kx,cx)
int i,*kx;
double *cx;
{
    char *calloc();
    int j,k,m,l,id[10],ie[10];

    for (l=0;l<in[i];l++)
    {
        iz[ix]=jz[dip[i]+i+l+1];        /* jz to iz */
        xx[ix]=calloc(20,sizeof(char));
        xx[ix]=zz[dip[i]+l];            /* zz to xx */
        m=0;
        for (j=0;j<jm[i];j++)           /* group the same variable */
            if (kx[j]==l) id[m++]=j;
        im[ix]=m;                       /* # of breakpoints */
        dim[ix+1]=dim[ix]+im[ix];
        order_index(m,id,cx,ie);        /* order the beta parameters */
        for (k=0;k<m;k++)
        {
            beta[dim[ix]+k]=cx[ie[k]];
            for (j=0;j<n;j++)           /* h to c */
                c[(dim[ix]+k)*n+j]=h[(diq[i]+ie[k])*n+j];
        }
        ix++;
    }
}
```

```
#define ELEM 80
#define MODEL 30
#define LAT 60
#define SEMI 30
#define DIM 1000
#define DIN 2000
#define DIP 60
#define DIQ 2000
```

```
#include <stdio.h>
#include "nport.h"
#include "pwl.h"
#include "/usr/include/local/graf.h"

/*************************************************************************/
/*************************************************************************/

char *xx[LAT];  /* mapping table between the lattice structure variables  */
                /* and the branch voltages and currents                   */
char *zz[100];  /* mapping table between the semi-lattice structure       */
                /* variables and the branch voltages and currents         */
char cho[2];    /* output variable type                                   */
int n;          /* circuit dimension                                      */
int ix=0;       /* index for lattice structure elements                   */
int jx=0;       /* index for semi-lattice structure elements              */
int *im;        /* # of breakpoints in the lattice structure variable     */
int *jm;        /* # of boundaries in the semi-lattice structure variable */
int *in;        /* array for # of ports of each multiport element         */
int *iz;        /* branch indices of the lattice structure elements       */
int *jz;        /* branch indices of the semi-lattice elements            */
int *dim;       /* dim[i] = total number of breakpoints up to the (i-1)-th*/
                /* lattice structure elements                             */
int *din;       /* din[i] = total number of partition boundary parameters */
                /* up to the (i-1)-th semi-lattice structure elements     */
int *dip;       /* dip[i] = total number of branches (or ports) up to the */
                /* (i-1)-th semi-lattice structure elements               */
int *diq;       /* diq[i] = total number of partition boundaries up to the*/
                /* (i-1)-th semi-lattice structure elements               */
struct INLINE branch[ELEM];
                /* array of element record                                */
struct B_VECTOR branch_vector[ELEM];
                /* array for branch indices                               */
double *P,*Q,*s;/* generalized hybrid equation parameters                 */
double *x;      /* lattice structure variables                            */
double *z;      /* semi-lattice structure variables                       */

/* parameters for canonical piecewise-linear equation */
double *B,*a,*r;
double *beta,*c;
double *h,*alpha;

double start;   /* starting position */
double ending;  /* ending position   */
int    dir;     /* tracing direction */
double tp,tn;   /* temporary storage for neighboring breakpoint positions */

/* graphic label variables in the screen */
char x_name[30],y_name[30],title[30];
double xmin,xmax,ymin,ymax;

int g1,g2,mn;   /* output variable indices */

/* input range for exhibiting operating region information */
double range1,range2;
```

```
char *model[MODEL];     /* string array for models or feature spec. */
int outfile=0;          /* outfile changes to 1 if -o option is present */
int region=0;           /* region changes to 1 if -r option is present */
FILE *op;               /* pointer to the output file "xx...x.out" */
FILE *rp;               /* pointer to the region inf. file "xx...x.reg */

/*************************************************************************/
/* Canonical piecewise-linear DC analysis for tracing the driving-point  */
/* and transfer characteristics.                                         */
/*************************************************************************/

main(argc,argv)
int argc;
char *argv[];
{
    FILE *gp;   /* pointer for input file */

    pwl_formulation(argc,argv,&gp);
    pwl_analysis(gp);
    if (outfile ==1)
        fclose(op);
    if (region ==1)
        fclose(rp);
}

/*************************************************************************/
/* Formulate the pwl resistive circuit to the canonical pwl equation     */
/*      B*x + sum (c_ji*|x_j - beta_i|) + sum (h_i*|(alpha_i,z) - gama_i|) */
/*    = a + t*r                                                          */
/* (see Eq.(2.17) in reference [1])                                      */
/*************************************************************************/

pwl_formulation(argc,argv,gp)
int argc;
char *argv[];
FILE **gp;
{
    FILE *fp;
    int is;

    /* open the input files */
    open_spice_file(argc,argv,&fp,gp);

    /* get the generalized hybrid equation                          */
    /*          Pv + Qi + s = 0                                     */
    /* for the linear n-port formed by extracting all nonlinear     */
    /* elements and the driving source                              */
    n=ELEM;
    mn=MODEL;
    n_port(fp,model,branch,branch_vector,&P,&Q,&s,&n,&mn);

    fclose(fp);

    /* pre-allocate spaces for each parameter in the */
    /* canonical pwl equation                        */
    alloc_1();
```

```
    /* decode the driving source and the model description of      */
    /* each nonlinear resistor to the canonical pwl representation */
    decode_pwl(&is);

    /* check whether a multiport pwl model description possesses the */
    /* lattice structure partition boundaries; if so, append it to   */
    /* the lattice structure part in the canonical pwl equation      */
    lattice();

    /* re-allocate spaces for each parameter ( pre-allocated in */
    /* alloc_1()) for a more exact allocation once the exact    */
    /* dimension has been found                                 */
    alloc_2();

    /* check point for circuit dimension */
    check_dim();

    /* partition the B matrix into parts of lattice */
    /* and semi-lattice structure                   */
    partition(is);

    /* print the mapping table */
    print_table();
}

/*****************************************************************************/
/* Perform the canonical pwl analysis for tracing the DP (Driving-Point)   */
/* and TC (Transfer) characteristics.                                        */
/*****************************************************************************/

pwl_analysis(gp)
FILE *gp;
{
    GRAF *graf_open(),*hp;
    int i,j,k;
    double *ss,*BB;

    mgiasngp(0,0);
    hp=graf_open();

    /* read the graphic information */
    read_gf();

    /* get the graphic specification */
    get_spec(mn);

    /* draw the graphic axises */
    setup_graf(xmin,xmax,ymin,ymax,x_name,y_name,title,hp);

    callocd(n,&ss,"ss");
    callocd(n*n,&BB,"BB");
    for (i=0;i<n*n;i++)
        BB[i]=B[i];
    for (i=0;i<n;i++)
        ss[i]=s[i];
```

```
    i=3;        /* starting from the 3rd color */

    /* trace branch(es) of solution curve */
    while (initial_point(gp,B,s,start)==1)
    {
        /* change color for next branch of solution curve */
        mgihue(3);

        tp=start;       /* starting position for input variable */

        /* calculate each breakpoint sequentially for tracing */
        /* the solution curve                                  */
        pwl_computation(hp);

        /* reset s and B */
        for (j=0;j<n;j++)
        {
            s[j]=ss[j];
            for (k=0;k<n;k++)
                B[j*n+k]=BB[j*n+k];
        }
    }
    fclose(gp);

    graf_close(hp);
    mgideagp();

}

/*****************************************************************************/
/* Decode the driving source and each nonlinear element (connected across  */
/* the external ports of the linear n-port) to the canonical pwl           */
/* representation.                                                         */
/*****************************************************************************/

decode_pwl(is)
int *is;
{
    char ch;
    int i,j;

    for (i=0;i<n;i++)
    {
        j=(branch_vector+i)->a;         /* in the j-th element line */
        ch=(branch+j)->name[0];
        if (ch=='V' || ch=='I')         /* driving source */
            driving_source(ch,i,is,(branch+j)->relation);
        else
        {
            switch((branch+j)->port)
            {
                case 1 : port_1(branch,j,i);
                         break;         /* 1-port element */
                case 2 : port_2(branch,branch_vector,j);
                         break;         /* 2-port element */
```

```
            }
        }
    }
}

/*****************************************************************************/
/* Read the graphic information                                              */
/*****************************************************************************/

read_gf()
{
    char chn[2];

    /* read the output variable type and indices;   */
    /*          i : driving output                  */
    /*          x : lattice structure variable      */
    /*          z : semi-lattice structure variable */
    printf("enter the output variable : x/z/i\n");
    scanf("%1s",cho);
    if (cho[0]=='x'){   /* output chosen from lattice structure variable */
        printf("output=x[?]\n");
        scanf("%d",&g1);
    }
    if (cho[0]=='z'){   /* output chosen from semi-lattice variable */
        printf("output=z[?]\n");
        scanf("%d",&g1);
    }

    if (region ==1)
    {
        printf("enter input variable range for region information\n");
        scanf("%lf%lf",&range1,&range2);
    }
}

/*****************************************************************************/
/* Extract the feature specification from the input file.                     */
/*****************************************************************************/

get_spec(na)
int na;
{
    char temp[30];
    int i;
    double atof();

    for (i=0;i<na;i++)
    {
        if (find_index(".x_name",model[i])==0)
            strcpy(x_name,model[i]+8);
        if (find_index(".y_name",model[i])==0)
            strcpy(y_name,model[i]+8);
        if (find_index(".title",model[i])==0)
            strcpy(title,model[i]+7);
        if (find_index(".x_axis",model[i])==0)
        {
```

```
            find_word(model[i],temp,2);
            xmin=atof(temp);
            find_word(model[i],temp,3);
            xmax=atof(temp);
        }
        if (find_index(".y_axis",model[i])==0)
        {
            find_word(model[i],temp,2);
            ymin=atof(temp);
            find_word(model[i],temp,3);
            ymax=atof(temp);
        }
    }
}

/*************************************************************************/
/* Print the mapping table between the equation variable x in the canonical*/
/* pwl equation and the circuit variables (voltages and currents in the    */
/* circuit)                                                                */
/*************************************************************************/

print_table()
{
    int i,j;

    printf("*********************************************************\n");
    for (i=0;i<ix;i++)
        printf("x[%d]=%s\n",i,xx[i]);
    for (i=0;i<jx;i++)
        for (j=0;j<in[i];j++)
            printf("z[%d]=%s\n",dip[i]+j,zz[dip[i]+j]);
    printf("*********************************************************\n");
}

/*************************************************************************/
/* Check whether the circuit dimension                                     */
/*               n =  ix          (lattice structure branches)             */
/*                  + dip[jx]     (semi-lattice stucture branches)         */
/*                  + 1           (driving source)                         */
/*************************************************************************/

check_dim()
{
    int j;

    j=ix+dip[jx]+1;
    if (j!=n)
    {
        printf("INCORRECT DIMENSION J=%d\n",j);
        exit();
    }
}

/*************************************************************************/
/* Open the spice file xxx..x.spc and the file xxx..x.op with the starting */
/* point(s). If -o (resp.; -r) option is specified, it also opens          */
```

```
/* xx...x.out (resp.; xx...x.reg).                                          */
/***************************************************************************/

open_spice_file(argc,argv,fp,gp)
int argc;
char *argv[];
FILE **fp,**gp;
{
    char *sc,line[12];
    FILE *fopen();

    /* extract the options */
    while (--argc > 0 && (*++argv)[0] == '-')
        for (sc=argv[0]+1; *sc != '\0'; sc++)
            switch(*sc)
            {
                case 'o' : outfile=1; break;
                case 'r' : region=1; break;
                default  : printf("illegal option %c\n",*sc);
                           argc=0;
                           break;
            }

    if (argc != 1 )
        exit_message("usage: pwltf -o -r filename");
    else
    {
        sprintf(line,"%s.spc",*argv);
        if ((*fp=fopen(line,"r"))==NULL)
        {
            printf("CAN'T OPEN SPICE_FILE %s\n",line);
            exit();
        }
        sprintf(line,"%s.op",*argv);
        if ((*gp=fopen(line,"r"))==NULL)
        {
            printf("CAN'T OPEN OP_FILE temp.op\n");
            exit();
        }

        /* open the output file "xx...x.out" */
        if (outfile == 1)
        {
            sprintf(line,"%s.out",*argv);
            if ((op=fopen(line,"w"))==NULL)
                exit_message("CAN'T OPEN OUT_FILE");
        }

        /* open the region information file "xx...x.reg" */
        if (region == 1)
        {
            sprintf(line,"%s.reg",*argv);
            if ((rp=fopen(line,"w"))==NULL)
                exit_message("CAN'T OPEN REG_FILE");
        }
    }
```

```
}

/*****************************************************************************/
/* Pre-allocate spaces for the parameters in the canonical pwl equation.   */
/*****************************************************************************/

alloc_1()
{
    calloci(LAT,&im,"im");
    calloci(SEMI,&jm,"jm");
    calloci(SEMI,&in,"in");
    calloci(LAT,&iz,"iz");
    calloci(DIP+SEMI,&jz,"jz");
    calloci(LAT,&dim,"dim");
    calloci(SEMI,&din,"din");
    calloci(SEMI,&dip,"dip");
    calloci(SEMI,&diq,"diq");
    callocd(n,&x,"x");
    callocd(n,&z,"z");
    callocd(n,&a,"a");
    callocd(n,&r,"r");
    callocd(n*n,&B,"B");
    callocd(DIM,&beta,"beta");
    callocd(n*DIM,&c,"c");
    callocd(n*DIQ,&h,"h");
    callocd(DIN,&alpha,"alpha");
}

/*****************************************************************************/
/* Re-allocate spaces for the parameters in the canonical pwl equation     */
/* with the exact required dimension since the exact size of dimension has */
/* been found.                                                               */
/*****************************************************************************/

alloc_2()
{
    check_space();

    ralloci(&im,ix,LAT);
    ralloci(&jm,jx,SEMI);
    ralloci(&in,jx,SEMI);
    ralloci(&iz,ix,LAT);
    ralloci(&jz,dip[jx]+jx+1,DIP+SEMI);
    ralloci(&dim,ix+1,LAT);
    ralloci(&din,jx+1,SEMI);
    ralloci(&dip,jx+1,SEMI);
    ralloci(&diq,jx+1,SEMI);
    rallocd(&x,ix+1,n);
    rallocd(&z,dip[jx],n);
    rallocd(&beta,dim[ix],DIM);
    rallocd(&c,n*dim[ix],n*DIM);
    rallocd(&h,n*diq[jx],n*DIQ);
    rallocd(&alpha,din[jx],DIN);
}

/*****************************************************************************/
```

```
/* Check whether enough spaces are allocated in the preliminary allocation */
/* alloc_1() for each parameter in the canonical pwl equation.             */
/*************************************************************************/

check_space()
{
    if (ix>LAT)
    {
        printf("NOT ENOUGH SPACE; LAT should be no less than %d\n",ix);
        exit();
    }
    if (jx>SEMI)
    {
        printf("NOT ENOUGH SPACE; SEMI should be no less than %d\n",jx);
        exit();
    }
    if (dim[ix]>DIM)
    {
        printf("NOT ENOUGH SPACE; DIM should be no less than %d\n",dim[ix]);
        exit();
    }
    if (din[jx]>DIN)
    {
        printf("NOT ENOUGH SPACE; DIN should be no less than %d\n",din[jx]);
        exit();
    }
    if (dip[jx]>DIP)
    {
        printf("NOT ENOUGH SPACE; DIP should be no less than %d\n",dip[jx]);
        exit();
    }
    if (diq[jx]>DIQ)
    {
        printf("NOT ENOUGH SPACE; DIQ should be no less than %d\n",diq[jx]);
        exit();
    }
}
```

```
#include <stdio.h>
#include "nport.h"
#include "pwl.h"

extern int n,ix,jx,*iz,*jz,*in;
extern int *im,*jm,*dim,*dip,*diq,*din;
extern char *xx[],*zz[];
extern double *B,*P,*Q,*s,*r;
extern double *c,*beta;
extern double *h,*alpha;
extern double *x,*z;
extern double start,ending;
extern int dir;

/*****************************************************************************/
/* Decode the driving source for tracing range and direction of DP and TC */
/*****************************************************************************/

driving_source(ch,i,is,model)

char ch;        /* 'V' (resp.; 'I') for voltage (resp.; current) source */
char *model;    /* driving source description */
int i,*is;      /* branch index for the driving source */
{
    int j,k;
    char xz[21],yz[21];
    double atof();

    /* delete unnecessary spacings */
    sqeez(model);

    /* decode the tracing direction; dir=1 (resp.; dir=-1) for */
    /* the direction of increasing (resp.; decreasing) input   */
    if (find_index(":+",model))>=0)
        dir=1;
    else
    {
        if (find_index(":-",model))>=0)
            dir = -1;
        else
            exit_message("INCORRECT DRIVING SOURCE MODEL FOR TRACING DIRECTION");
    }

    /* decode the starting position */
    k=find_index("(",model);
    model+=k+1;
    j=0;
    while (*model!=',' && j<20)
        xz[j++] = *(model++);
    if (j>=20)
        exit_message("TOO MANY DIGITS FOR STARTING POINT");
    xz[j]='\0';
    start=atof(xz);

    /* decode the ending position */
    model++;
```

```
    j=0;
    while (*model!=')' && j<20)
        yz[j++] = *(model++);
    if (j>=20)
        exit_message("TOO MANY DIGITS FOR ENDING POINT");
    yz[j]='\0';
    ending=atof(yz);

    /* construct the B matrix and r vector */
    for (k=0;k<n;k++)
    {
        if (ch=='V')
        {
            r[k] = -P[k*n+i];
            B[k*n+i]=Q[k*n+i];
        }
        if (ch=='I')
        {
            r[k] = -Q[k*n+i];
            B[k*n+i]=P[k*n+i];
        }
    }

    *is=i;      /* store the branch index for the driving source */
}

/*****************************************************************************/
/* Re-arrange and partition the B matrix into two parts with lattice and  */
/* semi-lattice structure respectively.                                      */
/*****************************************************************************/

partition(is)
int is;         /* branch index for driving source */
{
    int i,j,k,l;
    double *Bx;

    calloc(n*n,&Bx,"Bx");

    /* B matrix before partition */
    for (j=0;j<n;j++)
        for (i=0;i<n;i++)
            Bx[i*n+j]=B[i*n+j];

    /* lattice structure part */
    for (j=0;j<ix;j++)
        for (i=0;i<n;i++)
            B[i*n+j]=Bx[i*n+iz[j]];

    /* semi-lattice structure part */
    for (j=0;j<jx;j++)
    {
        l=0;
        if (j>0)
            for (k=0;k<j;k++) l+=in[j];
        for (i=0;i<n;i++)
```

```
            for (k=0;k<in[j];k++)
                B[i*n+ix+1+k]=Bx[i*n+jz[dip[j]+j+k+1]];
    }

    /* driving source part */
    for (i=0;i<n;i++)
        B[i*n+n-1]=Bx[i*n+is];

    cfree(Bx);
}

/*****************************************************************************/
/* Decode the 1-port element for canonical pwl equation.                     */
/*****************************************************************************/

port_1(branch,ja,i)
int ja;         /* element index in the input file */
int i;          /* branch index for this 1-port element */
struct INLINE branch[]; /* array of element record */
{
    int j,k;
    char *calloc();
    double a[12],*zx,*zy;

    iz[ix]=i;   /* the element is connected across the i-th port */
                /* of the linear n-port                          */

    xx[ix]=calloc(20,sizeof(char));

    /* decode the 1-port element for element characteristic */
    decode_1((branch+ja)->relation,im+ix,a);

    callocd(n,&zx,"zx");
    callocd(n,&zy,"zy");

    /* voltage-controlled case */
    if (find_index("i=",(branch+ja)->relation)>0)
    {
        for (k=0;k<n;k++)
        {
            zx[k]=P[k*n+i];
            zy[k]=Q[k*n+i];
        }
        sprintf(xx[ix],"v(%s)",(branch+ja)->name);
    }

    /* current-controlled case */
    if (find_index("v=",(branch+ja)->relation)>0)
    {
        for (k=0;k<n;k++)
        {
            zx[k]=Q[k*n+i];
            zy[k]=P[k*n+i];
        }
        sprintf(xx[ix],"i(%s)",(branch+ja)->name);
    }
```

```
    /* dim[i] = sum of (# of breakpoints in the j-th 1-port element) */
    /*          for j=0,1,2,...,i-1                                  */
    dim[ix+1]=dim[ix]+im[ix];

    /* substitute the pwl model into the hybrid equation */
    for (k=0;k<n;k++)
    {
        s[k]=s[k]+zy[k]*a[0];
        B[k*n+i]=zx[k]+zy[k]*a[1];
    }
    for (j=0;j<im[ix];j++)
    {
        for (k=0;k<n;k++)
            c[(dim[ix]+j)*n+k]=a[2+2*j]*zy[k];
        beta[dim[ix]+j]=a[3+2*j];
    }
    ix++;
    cfree(zx);
    cfree(zy);
}

/*****************************************************************************/
/* Read the starting point for the solution curve and calculate the          */
/* initial driving input current(resp.; voltage) if the driving source       */
/* is a voltage (resp.; current) source.                                     */
/*****************************************************************************/

initial_point(gp,BB,a,sta)
FILE *gp;       /* pointer for initial condition file */
double sta;     /* initial value for input variable */
double a[];     /* source vector */
double BB[];
{
    char line[50],*calloc();
    int i,j,k,l;
    double dx,fabs(),atof();

    if ((fgets(line,50,gp))==NULL)
        exit_message("CAN'T READ THE STARTING POINT");
    if (line[0]=='&')
        exit();

    /* read the starting operating point */
    for (i=0;i<ix;i++)
    {
        /* lattice structure variable */
        if ((fgets(line,50,gp))==NULL)
            exit_message("CAN'T READ THE STARTING POINT");
        if ((l=match_lat(line))==-1)
            exit_message("MISMATCHED LATTICE_VARIABLE NAME");
        x[l]=atof(line);
    }
    for (j=0;j<jx;j++)
        for (k=0;k<in[j];k++)
        {
```

```
            /* semi-lattice structure variable */
            if ((fgets(line,50,gp))==NULL)
                exit_message("CAN'T READ THE STARTING POINT");
            if ((l=match_semi(line))==-1)
                exit_message("MISMATCHED SEMI_VARIABLE NAME");
            z[l]=atof(line);
        }

    /* calculate the driving input current or voltage */
    i = -1;
    while (BB[(++i)*n+n-1]==0 && i<n);   /* search a nonzero coefficient */
    if (i<n)
    {
        for (j=0;j<ix;j++)
            a[i]=a[i]+BB[i*n+j]*x[j];
        for (j=0;j<jx;j++)
        {
            l=0;
            if (j>0)
                for (k=0;k<j;k++) l+=in[k];
            for (k=0;k<in[j];k++)
                a[i]+=BB[i*n+ix+k+l]*z[dip[j]+k];
        }
        for (j=0;j<ix;j++)
            for (k=0;k<im[j];k++)
                a[i]=a[i]+c[(dim[j]+k)*n+i]*fabs(x[j]-beta[dim[j]+k]);
        for (j=0;j<jx;j++)
            for (k=0;k<jm[j];k++)
            {
                dx=0;
                for (l=0;l<in[j];l++)
                    dx=dx+alpha[din[j]+k*(in[j]+1)+l]*z[dip[j]+l];
                dx=fabs(dx-alpha[din[j]+k*(in[j]+1)+in[j]]);
                a[i]=a[i]+h[(diq[j]+k)*n+i]*dx;
            }
        x[ix]=(sta*r[i]-a[i])/BB[i*n+n-1];
    }
    return(1);
}

/*****************************************************************************/
/* Decode the 1-port element model                                           */
/*              (i (or v)=(xx.x,yy.y)(xx.x,yy.y)...(xx.x,yy.y))              */
/* to the canonical pwl representation                                       */
/*              y = a + b*x + sum c_i*|x - beta_i|                           */
/*****************************************************************************/

decode_1(model,m,a)
char *model;    /* model string */
int *m;         /* # of breakpoints */
double a[];     /* numerical information in the model */
{
    char xz[21],yz[21];
    int na=0,i,k,l;
    double xw[10],yw[10];
    double atof();
```

```
    l=strlen(model)+1;
    k=find_index("=",model);
    model=model+k+1;
    while (*model!=')' && k++<l)
    {
        if (*model==' ') model++;
        if (*model=='(')
        {

            /* get the x-component of the breakpoint */
            i=0;
            while (*model!=',' && i<21)
                xz[i++] = *(++model);
            if (i>=21)
                exit_message("TOO MANY DIGITS IN PWL 1-PORT MODEL");
            xz[i-1]='\0';
            model++;
            if (na>=10)
                exit_message("TOO MANY BREAKPOINTS IN PWL 1-PORT MODEL");
            xw[na]=atof(xz);

            /* get the y-component of the breakpoint */
            i=0;
            while (*model!=')' && i<20)
                yz[i++] = *(model++);
            if (i>=20)
                exit_message("TOO MANY DIGITS IN PWL 1-PORT MODEL");
            yz[i]='\0';
            model++;
            yw[na++]=atof(yz);
        }
    }

    /* incorrect model description */
    if (k>=l)
        exit_message("MISSING ')' IN THE MODEL");

    *m=na-2;    /* # of breakpoints */

    /* construct the canonical pwl representation */
    canonical_pwl(*m,xw,yw,a);
}

/*************************************************************************/
/* Given the x and y coordinates of each breakpoint and two arbitrary    */
/* points in the ending segments, construct the 1-dimensional canonical  */
/* pwl representation                                                    */
/*              y = a + b*x + sum c_i*|x - beta_i|                        */
/*************************************************************************/

canonical_pwl(m,xw,yw,a)

int m;                  /* # of breakpoints */
double xw[],yw[];       /* breakpoins */
double a[];             /* parameters for canonical pwl representation */
```

```
{
    int i;
    double *slope;
    double fabs();

    callocd(m+1,&slope,"slope");

    /* calculate the slope of each segment */
    for (i=0;i<=m;i++)
        slope[i]=(yw[i+1]-yw[i])/(xw[i+1]-xw[i]);

    /* construct the parameters in the canonical pwl representation */
    a[1]=(slope[m]+slope[0])/2;
    for (i=0;i<m;i++)
    {
        a[2+2*i]=(slope[i+1]-slope[i])/2;
        a[3+2*i]=xw[i+1];
    }
    a[0]=yw[0]-a[1]*xw[0];
    for (i=0;i<m;i++)
        a[0]=a[0]-a[2+2*i]*fabs(a[3+2*i]-xw[0]);
    cfree(slope);
}

/*************************************************************************/
/* Match the input line in the file with starting points to the mapping  */
/* table of lattice structure variables to identify the index i such that */
/* the input line is the atarting point for x[i]; return -1 if there is no */
/* matched variable.                                                     */
/*************************************************************************/

match_lat(line)
char *line;
{
    int i,k;

    k=find_index("=",line);

    /* search the mapping table for lattice structure variables */
    for (i=0;i<ix;i++)
        if (find_index(xx[i],line)==0)
        {
            /* extract the ASCII code for the starting point */
            strdel(line,0,k+1);
            strdel(line,strlen(line)-1,1);

            return(i);
        }

    if(i==ix)
        return(-1);
}

/*************************************************************************/
/* Match the input line in the file with starting points to the mapping  */
```

```
/* table of semi-lattice structure variables for identifying the indices   */
/* i and j such that the numerical value in the input line is the starting */
/* point for the semi-lattice structure variable z[dip[i]+j], the j-th     */
/* port variable of the i-th semi-lattice structure multiport element;     */
/* return -1 if there is no matched variable.                              */
/***************************************************************************/

match_semi(line)
char *line;
{
    int i,j,k;

    k=find_index("=",line);

    /* search the mapping table for semi-lattice structure variables */
    for (i=0;i<jx;i++)
        for (j=0;j<in[i];j++)
            if (find_index(zz[dip[i]+j],line)==0)
            {
                /* extract the ASCII code for the starting point */
                strdel(line,0,k+1);
                strdel(line,strlen(line)-1,1);

                return(dip[i]+j);
            }
    if (i==jx)
        return(-1);
}
```

```
#include <stdio.h>
#include "nport.h"
#include "pwl.h"

extern int n,jx,*jm,*in,*jz,*dip,*diq,*din;
extern double *P,*Q,*s,*B;
extern double *h,*alpha;
extern char *zz[];

/*****************************************************************************/
/* Decode the 2-port model into a 2-dimensional canonical piecewise-linear */
/* equation.                                                                 */
/*****************************************************************************/

decode_2(model,nn,a1,a2,bd)
char model[];   /* 2-port model */
int *nn;        /* # of partition boundaries */
double a1[];    /* coefficients of 1st canonical pwl equation */
double a2[];    /* coefficients of 2nd canonical pwl equation */
double bd[];    /* coefficients of partition boundary equations */

{
    char *dx,*calloc();
    double atof();
    int j,k;

    dx=calloc(strlen(model)+1,sizeof(char));/* temporary storage for the model */

    /* decode the 1st canonical pwl equation */
    k=find_index("P1=(",model);
    strcpy(dx,model);
    strdel(dx,0,k+4);
    extract(dx,nn,a1);

    /* decode the 2nd canonical pwl equation */
    k=find_index("P2=(",dx);
    dx=dx+k+4;
    extract(dx,nn,a2);

    /* decode the partition boundaries */
    k=find_index("Bd=(",dx);
    dx=dx+k+4;
    extract(dx,&j,bd);
}

/*****************************************************************************/
/*  Extract the numerical data from ASCII model equation.                    */
/*****************************************************************************/

extract(code,nn,a)
char *code;     /* ASCII code of numerical information in the model */
int *nn;        /* # of partition boundaries */
double a[];     /* decoded numerical data */
{
    char st[21];
    double atof();
```

```
    int i,j=0,l=0;

    while (find_index(",",code)<find_index(")",code) &&
           find_index(",",code)>=0)
    {
        l++;
        if (l>30)
            exit_message("TOO MANY PARAMETERS IN THE MODEL OF PWL 2-PORT");
        i=0;
        if (*code==',') code++;
        while (*code!=',' && *code!=')' && i<20)
            st[i++] = *(code++);
        if (i>=20) exit_message("TOO MANY DIGITS IN PWL 2-PORT MODEL");
        st[i]='\0';
        a[j++]=atof(st);
    }
    *nn=j-3;
}

/*****************************************************************************/
/* Decode the 2-port element ( which is in the j-th element line ) for its */
/* canonical piecewise-linear equation.                                      */
/*****************************************************************************/

port_2(branch,branch_vector,ja)

struct INLINE branch[];          /* array of element record */
struct B_VECTOR branch_vector[];/* array for branch indices */
int ja;                          /* branch index of the 2-port element */
{
    int i,k;
    double a1[13],a2[13],bd[30],*zx1,*zx2,*zy1,*zy2;
    char *calloc();

    if (chosen(jx,ja)==-1 || jx==0)     /* check whether processed */
    {
        in[jx]=2;       /* port # = 2 */

        /* dip[i] = sum of (# of ports in the j-th multiport element) */
        /* for j=0,1,2,...,i-1                                        */
        dip[jx+1]=dip[jx]+in[jx];

        /* get the branch indices for each port branch */
        /* of this multiport element                   */
        semi_index(branch_vector,ja,jx);

        /* decode the 2-port pwl model */
        decode_2((branch+ja)->relation,&jm[jx],a1,a2,bd);

        /* diq[i] = sum of (# of partition boundaries in the  */
        /* canonical pwl model of the j-th multiport element) */
        /* for j=0,1,2,...,i-1                                */
        din[jx+1]=din[jx]+jm[jx]*(in[jx]+1);

        /* din[i] = sum of (# of parameters in the partition */
        /* boundary equations of the j-th multiport element) */
```

```
        /* for j=0,1,2,...,i-1                                  */
        diq[jx+1]=diq[jx]+jm[jx];

        callocd(n,&zx1,"zx1");
        callocd(n,&zx2,"zx2");
        callocd(n,&zy1,"zy1");
        callocd(n,&zy2,"zy2");

        /* get the semi-lattice structure variable definition and */
        /* the corresponding canonical pwl equation parameters    */
        semi_var(branch,branch_vector,ja,zx1,zy1,zx2,zy2);

        /* find the canonical pwl equation parameters */
        for (i=0;i<n;i++)
        {
            s[i]=s[i]+zy1[i]*a1[0]+zy2[i]*a2[0];
            B[i*n+jz[dip[jx]+jx+1]]=zx1[i]+zy1[i]*a1[1]+zy2[i]*a2[1];
            B[i*n+jz[dip[jx]+jx+2]]=zx2[i]+zy1[i]*a1[2]+zy2[i]*a2[2];
        }
        for (i=0;i<jm[jx];i++)
        {
            for (k=0;k<=in[jx];k++)
                alpha[din[jx]+i*(in[jx]+1)+k]=bd[3*i+k];
            for (k=0;k<n;k++)
                h[(diq[jx]+i)*n+k]=zy1[k]*a1[3+i]+zy2[k]*a2[3+i];
        }
        jx++;   /* increment # of multiport elements */
        cfree(zx1);
        cfree(zy1);
        cfree(zx2);
        cfree(zy2);
    }
}

/*************************************************************************/
/* Determine whether any branch in the 2-port element has been processed;  */
/* return -1 if it is the 1st branch of this 2-port element ever processed */
/*************************************************************************/

chosen(j,ja)
int j,ja;
{
    int k=0;

    for (k=0;k<=j;k++)
        if (jz[dip[k]+k]==ja)
            return(k);
    if (k>j)
        return(-1);
}

/*************************************************************************/
/* Determine the branch indices for 2-port elements.                     */
/*************************************************************************/

semi_index(bv,ja,j)
```

```
struct B_VECTOR bv[];
int ja,j;
{
    int k;

    jz[dip[j]+j]=ja;
    for (k=0;k<n;k++)
        if ((bv+k)->a==ja)
        {
            /* 1st port of the j-th 2-port element is at port k */
            /* of the linear n-port                             */
            if ((bv+k)->b==1)
                jz[dip[j]+j+1]=k;
            /* 2nd port of the j-th 2-port element is at port k */
            /* of the linear n-port                             */
            if ((bv+k)->b==2)
                jz[dip[j]+j+2]=k;
        }
}

/*************************************************************************/
/* Identify the element type for the 2-port element and construct the    */
/* corresponding canonical pwl equation parameters and the semi-lattice  */
/* structure variable table.                                             */
/*************************************************************************/

semi_var(branch,branch_vector,ja,zx1,zy1,zx2,zy2)
struct INLINE branch[];
struct B_VECTOR branch_vector[];
int ja;
double *zx1,*zy1,*zx2,*zy2;
{
    int i,l= -1;
    char *calloc();

    if (find_index("(i,i):",(branch+ja)->relation)>=0)
        l=0;    /* (1) voltage (2) voltage -controlled */
    if (find_index("(i,v):",(branch+ja)->relation)>=0)
        l=1;    /* (1) voltage (2) current -controlled */
    if (find_index("(v,i):",(branch+ja)->relation)>=0)
        l=2;    /* (1) current (2) voltage -controlled */
    if (find_index("(v,v):",(branch+ja)->relation)>=0)
        l=3;    /* (1) current (2) current -controlled */
    if (l<0 || l>3)
        exit_message("INCORRECT 2-PORT MODEL");

   /* find the variable table for 2-port element with    */
   /* semi-lattice structure canonical pwl model and the */
   /* corresponding canonical pwl equation parameters    */
   for (i=0;i<2;i++)
       zz[dip[jx]+i]=calloc(20,sizeof(char));
   if (l<2)
   {
       sprintf(zz[dip[jx]],"v1(%s)",(branch+ja)->name);
       if (l==0)
           sprintf(zz[dip[jx]+1],"v2(%s)",(branch+ja)->name);
```

```
        else sprintf(zz[dip[jx]+1],"i2(%s)",(branch+ja)->name);
        for (i=0;i<n;i++){
            zx1[i]=P[i*n+jz[dip[jx]+jx+1]];
            zy1[i]=Q[i*n+jz[dip[jx]+jx+1]];
            if (l==1)
            {
                 zx2[i]=Q[i*n+jz[dip[jx]+jx+2]];
                 zy2[i]=P[i*n+jz[dip[jx]+jx+2]];
            } else
            {
                 zx2[i]=P[i*n+jz[dip[jx]+jx+2]];
                 zy2[i]=Q[i*n+jz[dip[jx]+jx+2]];
            }
        }
    }
    if (l>=2)
    {
        sprintf(zz[dip[jx]],"i1(%s)",(branch+ja)->name);
        if (l==2)
            sprintf(zz[dip[jx]+1],"v2(%s)",(branch+ja)->name);
        else sprintf(zz[dip[jx]+1],"i2(%s)",(branch+ja)->name);
        for (i=0;i<n;i++)
        {
            zx1[i]=Q[i*n+jz[dip[jx]+jx+1]];
            zy1[i]=P[i*n+jz[dip[jx]+jx+1]];
            if (l==3)
            {
                zx2[i]=Q[i*n+jz[dip[jx]+jx+2]];
                zy2[i]=P[i*n+jz[dip[jx]+jx+2]];
            } else
            {
                zx2[i]=P[i*n+jz[dip[jx]+jx+2]];
                zy2[i]=Q[i*n+jz[dip[jx]+jx+2]];
            }
        }
    }
}
```

```
#include <stdio.h>
#include "pwl.h"
#include "/usr/include/local/graf.h"

/*****************************************************************************/
/*****************************************************************************/

extern int n,ix,jx,*in,*im,*jm,dir,*dim,*dip,*diq,*din;
extern char *xx[],*zz[],cho[];
extern double *alpha,*beta;
extern double *B,*r,*c;
extern double *x,*z,*h;
extern double start,ending,tp,tn;
extern int g1,g2;
extern double range1,range2;
extern FILE *op,*rp;
extern int outfile,region;
int i0,j0,k0,*iy,*jy;
int nc=0;

/*****************************************************************************/
/* Implement the Generalized Breakpoint Hopping Algorithm to calculate     */
/* each breakpoint sequentially for tracing the solution curve of a        */
/* canonical pwl equation and find the DP (Driving-Point) and TC (Transfer)*/
/* characteristic of a nonlinear resistive circuit.                        */
/*****************************************************************************/

pwl_computation(hp)
GRAF *hp;
{
    FILE *fopen();
    int q,s,ub=0;
    double *d,*d1,fabs();
    double out,ti;

    callocd(diq[jx],&jy,"jy");
    callocd(n,&d,"d");
    callocd(n,&d1,"d1");

    /* select the output variable */
    switch(cho[0])
    {
        case 'x' :  out=x[g1]; break;
        case 'z' :  out=z[g1]; break;
        case 'i' :  out = -x[ix]; break;
    }
    graf_move(start,out,hp);
    if (outfile == 1)
    {
        fprintf(op,"\n******** PWL CURVE %d ********\n",++nc);
        fprintf(op,"input = %.3e\t\toutput = %.3e\n",start,out);
    }

    /* determine the initial region for the starting point */
    location_index();
```

```
    if (region == 1 && tp<range2 && tp>=range1)
        reg_inf(tp);

    /* construct the initial Jacobian matrix and calculate */
    /* the initial direction vector                        */
    ini_B_d(d);

    /* find the 1st breakpoint and draw the line segment */
    /* between the breakpoint and the starting point     */
    if (next_point(d,&q)==-1)
        ub=1;;
    switch(cho[0])
    {
        case 'x' :  out=x[g1]; break;
        case 'z' :  out=z[g1]; break;
        case 'i' :  out = -x[ix]; break;
    }
    graf_draw(tn,out,hp);
    if (outfile == 1)
        fprintf(op,"input = %.3e\t\toutput = %.3e\n",tn,out);
    ti=tn;

    while(((dir>0 && tn<ending) || (dir<0 && tn>ending)) && ub==0)
    {
        /* find the Jacobian matrix and direction */
        /* vector in the next region              */
        B_and_d(d,d1,q);

        if (tn<range2 && tn>=range1)
            reg_inf(tn);

        /* determine the correct direction vector in order */
        /* to continue tracing the solution curve          */
        d_vector(&s,q,d,d1);

        /* calculate each distance of the present breakpoint */
        /* to the neighboring region                         */
        increment(d,s,&q);

        /* calculate the next breakpoint and draw the line segment */
        /* between the present and the next breakpoint             */
        if (break_point(s,d)==-1)
            ub=1;
        switch(cho[0])
        {
            case 'x' :  out=x[g1]; break;
            case 'z' :  out=z[g1]; break;
            case 'i' :  out = -x[ix]; break;
        }
        graf_draw(tn,out,hp);
        if (outfile == 1)
            fprintf(op,"input = %.3e\t\toutput = %.3e\n",tn,out);
        if (fabs(tn-ti)<1.0e-8)
        {
            printf("BREAK A LOOP\n");
            break;
```

```
        }
    }
    cfree(iy);
    cfree(jy);
    cfree(d);
    cfree(d1);

}

/*****************************************************************************/
/* Find the first breakpoint after the starting point.                       */
/*****************************************************************************/

next_point(d,q)
int *q;         /* boundary index; 0 for lattice and 1 for semi-lattice */
double *d;      /* direction vector */
{
    double fabs(),dti_t=1e5,dtj_t=1e5;
    double delta_t;

    /* calculate the distance to each lattice structure boundary */
    d_lat(dir,&dti_t,d);

    /* calculate the distance to each semi-lattice structure boundary */
    d_semi(dir,0,0,0,&dtj_t,d);

    /* classified as a corner point if within the distance =1e-8 */
    /* of a corner point                                         */
    if (dti_t<1.0e-8 && dtj_t<1.0e-8)
    {
        printf("WARNING MESSAGE : CORNER POINT IN LATTICE AND ");
        printf("SEMI-LATTICE STRUCTURE BOUNDARIES\n");
    }

    /* compare the minimal distances from the lattice structure boundary */
    /* and the semi-lattice structure boundary and choose the minimum    */
    if (dti_t>dtj_t)
    {
        *q=1;
        delta_t=dir*dtj_t;
    } else
    {
        *q=0;
        delta_t=dir*dti_t;
    }
    tn=tp+delta_t;       /* position for next breakpoint */

    /* calculate the breakpoint */
    if (break_point(1,d)==-1)
        return(-1);
    else
        return(0);
}

/*****************************************************************************/
/* Print the boundary equations for a region.                                */
```

```
/*****************************************************************************/

reg_inf(tx)
double tx;
{
    int i,k;
    char d1[20],d2[20],line[80];

    fprintf(rp,"\nPWL CURVE %d AT INPUT=%.3e\n",nc,tx);
    fprintf(rp,"********* REGION INFORMATION *********\n");

    /* print the lattice structure boundary */
    for (i=0;i<ix;i++)
    {
        if (iy[i]==0)
        {
            f_to_ax(beta[dim[i]],d1);
            fprintf(rp,"%s >= %s \n",d1,xx[i]);
        }
        if (iy[i]==im[i])
        {
            f_to_ax(beta[dim[i]+im[i]-1],d1);
            fprintf(rp,"%s >= %s \n",xx[i],d1);
        }
        if (iy[i]>0 && iy[i]<im[i])
        {
            f_to_ax(beta[dim[i]+iy[i]],d1);
            f_to_ax(beta[dim[i]+iy[i]-1],d2);
            fprintf(rp,"%s >= %s >= %s\n",d1,xx[i],d2);
        }
    }

    /* print the semi-lattice structure boundary */
    for (i=0;i<jx;i++)
        for (k=0;k<jm[i];k++)
        {
            /* construct the boundary expression */
            semi_reg(line,zz[dip[i]],zz[dip[i]+1],alpha[din[i]+k*(in[i]+1)],
                     alpha[din[i]+k*(in[i]+1)+1],alpha[din[i]+k*(in[i]+1)+2]);

            if (jy[diq[i]+k]==1)
                fprintf(rp,"%s >= 0\n",line);
            if (jy[diq[i]+k]==0)
                fprintf(rp,"%s <= 0\n",line);
        }
    fprintf(rp,"\n");
}

/*****************************************************************************/
/* Construct the string expression of a boundary equation from the           */
/* numerical values of the equation coefficients.                            */
/*****************************************************************************/

semi_reg(line,c1,c2,a1,a2,a3)
char *line,*c1,*c2;
double a1,a2,a3;
```

```
{
    int l;
    char d1[20],d2[20],d3[20];

    /* determine the sign of the coefficients */
    if (a2)=0 && a3)=0)
        l=1;
    if (a2)=0 && a3<0)
        l=2;
    if (a2<0 && a3)=0)
        l=3;
    if (a2<0 && a3<0)
        l=4;

    /* convert the numerical value of each coefficient to */
    /* the ASCII expression                               */
    f_to_ax(a1,d1);
    f_to_ax(a2,d2);
    f_to_ax(a3,d3);

    /* combine each coefficient expression with the corresponding   */
    /* variable into the ASCII expression of the boundary equation  */
    /* each following case corresponds to a particular sign pattern */
    /* of the coefficients in order to avoid the expression like    */
    /*                  3*v(R1)--2*i(R2)--1                         */
    /* (should be       3*v(R1)+2*i(R2)+1                        ) */
    switch(l)
    {
        case 1 : sprintf(line,"%s*%s+%s*%s-%s",d1,c1,d2,c2,d3); break;
        case 2 : {
                     strdel(d3,0,1);
                     sprintf(line,"%s*%s+%s*%s+%s",d1,c1,d2,c2,d3);
                     break;
                 }
        case 3 : sprintf(line,"%s*%s%s*%s-%s",d1,c1,d2,c2,d3); break;
        case 4 : {
                     strdel(d3,0,1);
                     sprintf(line,"%s*%s%s*%s+%s",d1,c1,d2,c2,d3);
                     break;
                 }
    }
}

/*****************************************************************************/
/* Determine the region where the initial point is located.                  */
/*****************************************************************************/

location_index()
{
    int i,j,k;
    double cx;

    calloci(ix,&iy,"iy");

    /* determine the region indices for lattice structure boundaries */
    for (i=0;i<ix;i++)
```

```
    {
        if ((x[i]-beta[dim[i]])<=0)
            iy[i]=0;
        else
        {
            if ((x[i]-beta[dim[i]+im[i]-1])>0) iy[i]=im[i];
            else
            {
                for (k=1;k<im[i];k++)
                    if ((x[i]-beta[dim[i]+k-1])*(x[i]-beta[dim[i]+k])<=0)
                    {
                        iy[i]=k;
                        break;
                    }
                if (k==im[i])
                    exit_message("INITIAL POINT IS IN UNDEFINED REGION");
            }
        }
    }

    /* determine the region indices for semi-lattice structure boundaries */
    for (j=0;j<jx;j++)
        for (k=0;k<jm[j];k++)
        {
            cx=0;
            for (i=0;i<in[j];i++)
                cx=cx+alpha[din[j]+k*(in[j]+1)+i]*z[dip[j]+i];
            if ((cx-alpha[din[j]+k*(in[j]+1)+in[j]])<0) jy[diq[j]+k]=0;
            else jy[diq[j]+k]=1;
        }
}

/*****************************************************************************/
/* Find the Jacobian matrix and the direction vector of the solution curve */
/* in the initial region.                                                    */
/*****************************************************************************/

ini_B_d(d)
double *d;      /* direction vector */
{
    int i,j,k,l,lx,*ipvt;
    double *Bx,*ux,rcond;

    /* construct the Jacobian matrix in the initial region */
    for (i=0;i<ix;i++)          /* lattice structure part */
        for (j=0;j<im[i];j++)
            for (k=0;k<n;k++)
            {
                if (j>=iy[i]) B[k*n+i]-=c[(dim[i]+j)*n+k];
                else B[k*n+i]+=c[(dim[i]+j)*n+k];
            }
    for (i=0;i<jx;i++)          /* semi-lattice structure part */
    {
        lx=0;
        if (i>0) for (l=0;l<i;l++) lx+=in[l];
        for (j=0;j<jm[i];j++)
```

```
            for (k=0;k<n;k++)
                for (l=0;l<in[i];l++)
                {
                    if (jy[diq[i]+j]==1)
                        B[k*n+ix+lx+l]+=alpha[din[i]+j*(in[i]+1)+l]
                                        *h[(diq[i]+j)*n+k];
                    else
                        B[k*n+ix+lx+l]-=alpha[din[i]+j*(in[i]+1)+l]
                                        *h[(diq[i]+j)*n+k];
                }
    }

    calloci(n,&ipvt,"ipvt");
    callocd(n,&ux,"ux");
    callocd(n*n,&Bx,"Bx");

    /* compute the direction vector by solving Bd=r */
    for (i=0;i<n;i++)
    {
        d[i]=r[i];
        for (j=0;j<n;j++)
            Bx[i*n+j]=B[i*n+j];
    }
    sgeco(Bx,n,ipvt,&rcond,ux); /* LU decomposition */
    sgesl(Bx,n,ipvt,d,0);

    cfree(ipvt);
    cfree(ux);
    cfree(Bx);
}


/*************************************************************************/
/* Calculate the distance of the present breakpoint to each lattice      */
/* structure boundary and the corresponding required amount of input for */
/* reaching that boundary. Choose the boundary with minimal amount of input*/
/* which will be the 1st lattice structure boundary hit by the solution  */
/* curve; return -1 if if goes to unbounded region without hitting any   */
/* boundary.                                                             */
/*************************************************************************/

d_lat(s,dti_t,d)
int s;
double *dti_t;
double *d;
{
    int i,m=0;
    double *dta,fabs();

    callocd(ix,&dta,"dta");

    for (i=0;i<ix;i++)
        if (fabs(d[i])>1.0e-9 && im[i]>0)
        {
            if ((s*d[i])>0)
            {
```

```
                if (iy[i]==im[i])    /* i-th component goes to + infinity */
                    dta[i]=1.0e5*fabs(d[i]);
                else
                    dta[i]=beta[dim[i]+iy[i]]-x[i];
            }
            else
            {
                if (iy[i]==0)        /* i-th component goes to - infinity */
                    dta[i]=1.0e5*fabs(d[i]);
                else
                    dta[i]=x[i]-beta[dim[i]+iy[i]-1];
            }
            dta[i]=dta[i]/fabs(d[i]);  /* required amount of input for the */
                                       /* i-th component to hit a boundary */

            /* choose the minimum and determine the lattice */
            /* structure boundary index i0                  */
            if (*dti_t>dta[i])
            {
                *dti_t=dta[i];
                i0=i;
            }
        }

        for (i=0;i<ix;i++)
            if (fabs(dta[i]-*dti_t)<1.0e-8 && *dti_t<1.0e-4 && d[i]!=0)
                m++;    /* present breakpoint is in the intersection */
                        /* of m lattice structure boundaries         */

    cfree(dta);
    if (m>1)
    {
        printf("WARNING MESSAGE : CORNER POINT IN LATTICE STRUCTURE ");
        printf("BOUNDARIES\n");
    }

}

/*************************************************************************/
/* Calculate the distance of the present breakpoint to each semi-lattice   */
/* structure boundary and the corresponding required amount of input for   */
/* reaching that boundary. Choose the boundary with minimal amount of input*/
/* which will be the 1st semi-lattice structure boundary hit by the        */
/* solution curve; return -1 when entering an unbounded region without     */
/* hitting any boundary.                                                   */
/*************************************************************************/

d_semi(s,q,j1,k1,dtj_t,d)
int s,q,j1,k1;
double *dtj_t,*d;
{
    int i,j,k,l,m=0;
    double *dtb,dx,dy;

    callocd(diq[jx],&dtb,"dtb");
```

```
    for (i=0;i<jx;i++)
    {
        l=0;
        if (i>0)
            for (k=0;k<i;k++) l+=in[k];
        for (j=0;j<jm[i];j++)
        {
            dx=0;
            dy=0;
            for (k=0;k<in[i];k++)
            {
                dx+=alpha[din[i]+j*(in[i]+1)+k]*d[ix+l+k];
                dy+=alpha[din[i]+j*(in[i]+1)+k]*z[dip[i]+k];
            }
            dy=alpha[din[i]+j*(in[i]+1)+in[i]]-dy;
            if (fabs(dy)<1.0e-8)
                m++;
            if (dx!=0)
                dtb[diq[i]+j]=s*dy/dx;
            if (dtb[diq[i]+j]>0 && *dtj_t>dtb[diq[i]+j])
            {
                if (q!=1 || j1!=i || k1!=j)
                {
                    j0=i;
                    k0=j;
                    *dtj_t=dtb[diq[i]+j];
                }
            }
        }
    }
    if (m>1)
    {
        printf("WARNING MESSAGE : CORNER POINT IN SEMI-LATTICE ");
        printf("STRUCTURE BOUNDARIES\n");
    }
    cfree(dtb);
}

/*****************************************************************************/
/*****************************************************************************/
```

```
#include <stdio.h>
#include "pwl.h"

extern int n,ix,jx,*in,*im,*jm,dir,*dim,*din,*dip,*diq;
extern char *xx[],*zz[];
extern double *alpha,*beta;
extern double *B,*r,*c;
extern double *h,*x,*z;
extern double tp,tn,ending;
extern int i0,j0,k0,*iy,*jy;

/*****************************************************************************/
/* Calculate the distance between the present breakpoint and each boundary */
/* and determine the next region by finding the boundary with minimal      */
/* amount of required input for the solution curve to hit that boundary    */
/* from the present breakpoint. Return -1 when entering unbounded region.  */
/*****************************************************************************/

increment(d,s,q)

int s;          /* direction vector sign */
int *q;         /* boundary index */
double *d;      /* direction vector */
{
    int j1,k1;
    double dti_t=1e5,dtj_t=1e5;
    double fabs(),delta_t;

    /* present breakpoint is located in the k0-th boundary of the    */
    /* j0-th multiport element with semi-lattice structure partition */
    /* boundaries in its canonical pwl model                         */
    j1=j0;
    k1=k0;

    /* calculate the distance to each lattice structure boundary and */
    /* choose the minimum to determine the next region               */
    d_lat(s,&dti_t,d);

    /* calculate the distance to each semi-lattice structure boundary */
    /* and choose the minimum to determine the next region            */
    d_semi(s,*q,j1,k1,&dtj_t,d);

    /* classified as a corner point if within the distance =1e-8 */
    /* of a corner point                                         */
    if (dti_t<1.0e-8 && dtj_t<1.0e-8)
    {
        printf("WARNING MESSAGE : CORNER POINT IN LATTICE ");
        printf("AND SEMI-LATTICE STRUCTURE BOUNDARIES\n");
    }

    /* compare the distances and choose the minimum */
    if (dti_t>dtj_t)    /* hit a semi-lattice structure boundary first */
    {
        *q=1;
        delta_t=dtj_t;
    }
```

```
    else                /* hit a lattice structure boundary first */
    {
        *q=0;
        delta_t=dti_t;
    }

    /* renew the position to next breakpoint */
    tp=tn;
    tn += s*delta_t;
}

/*****************************************************************************/
/* Calculate the next breakpoint; return -1 if entering an unbounded       */
/* region and an arbitrary point on the ending segment is calculated.      */
/*****************************************************************************/

break_point(s,d)
int s;          /* direction vector sign */
double *d;      /* direction vector */
{
    int i,j,ub=0;
    double fabs();

    /* entering the unbounded region */
    if (fabs(tn)>5.0e4)
    {
        ub=1;
        tn=tp+s*100;
    }

    /* check whether the ending position is reached */
    if ((dir>0 && tn>ending) || (dir<0 && tn<ending))
        tn=ending;

    /* calculate the lattice structure coordinates */
    /* of the next breakpoint                      */
    for (i=0;i<ix;i++)
        x[i]+=d[i]*(tn-tp);

    /* calculate the semi-lattice structure coordinates */
    /* of the next breakpoint                           */
    for (i=0;i<jx;i++)
        for (j=0;j<in[i];j++)
            z[dip[i]+j]+=d[ix+dip[i]+j]*(tn-tp);

    /* calculate the driving current or voltage of the next breakpoint */
    x[ix]+=d[ix+dip[jx]]*(tn-tp);

    if (ub==1)
        return(-1);
    else
        return(0);
}

/*****************************************************************************/
/* Renew the Jacobian matrix and calculate the direction vector in the     */
```

```
/* next region.                                                              */
/***************************************************************************/

B_and_d(d,d1,q)
int q;          /* boundary index */
double *d;      /* previous direction vector */
double *d1;     /* present direction vector */
{
    int i,j,*ipvt;
    double cx,*Bx,rcond,*ux;

    /* if crossing the lattice structure boundary */
    if (q==0)
    {
        /* renew the region index for lattice structure component */
        if (d[i0]*(tn-tp))0)    /* crossing the right boundary */
            iy[i0]++;
        else                    /* crossing the left boundary  */
            iy[i0]--;

        /* renew the Jacobian matrix in lattice structure part */
        for (i=0;i(n;i++)
        {
            if (d[i0]*(tn-tp))0)
                B[i*n+i0]+=2*c[(dim[i0]+iy[i0]-1)*n+i];
            else
                B[i*n+i0]-=2*c[(dim[i0]+iy[i0])*n+i];
        }
    }

    /* if crossing the semi-lattice structure boundary */
    if (q==1)
    {
        /* renew the region index for semi-lattice component */
        if (jy[diq[j0]+k0]==1)
        {
            cx = -1;
            jy[diq[j0]+k0]=0;
        } else
        {
            cx=1;
            jy[diq[j0]+k0]=1;
        }
        for (i=0;i(n;i++)
            for (j=0;j(in[j0];j++)
                B[i*n+ix+dip[j0]+j]+=2*cx*h[(diq[j0]+k0)*n+i]
                                    *alpha[din[j0]+k0*(in[j0]+1)+j];
    }

    calloci(n,&ipvt,"ipvt");
    callocd(n,&ux,"ux");
    callocd(n*n,&Bx,"Bx");

    /* find the new direction vector by solving B*d1=r */
    for (i=0;i(n;i++)
    {
```

```
        d1[i]=r[i];
        for (j=0;j<n;j++)
            Bx[i*n+j]=B[i*n+j];
    }
    sgeco(Bx,n,ipvt,&rcond,ux);
    sgesl(Bx,n,ipvt,d1,0);

    cfree(ipvt);
    cfree(ux);
    cfree(Bx);
}

/*****************************************************************************/
/* Determine the correct direction vector in order to continue tracing     */
/* the solution curve.                                                       */
/*****************************************************************************/

d_vector(s,q,d,d1)
int q;          /* boundary index; 0 for lattice and 1 for semi-lattice */
int *s;         /* direction sign; 1 for entering and -1 for returning */
double *d;      /* direction vector in the previous region */
double *d1;     /* direction vector in the present region */
{
    int i,k;
    double dx,dy;

    /* crossing the lattice structure boundary */
    if (q==0)
    {
        if ((d1[i0]*d[i0]*(tn-tp))>0)           /* d1 is entering */
            *s=1;
        else                /* d1 is returning to previous region */
            *s = -1;
    }

    /* crossing the semi-lattice structure boundary */
    if (q==1)
    {
        dx=0;
        dy=0;
        for (k=0;k<in[j0];k++)
        {
            dx+=d[ix+dip[j0]+k]*alpha[din[j0]+k0*(in[j0]+1)+k];
            dy+=d1[ix+dip[j0]+k]*alpha[din[j0]+k0*(in[j0]+1)+k];
        }
        if (((tn-tp)*dx*dy)>0)  /* entering  */
            *s=1;
        else                    /* returning */
            *s = -1;
    }

    /* renew the direction vector */
    for (i=0;i<n;i++)
        d[i]=d1[i];
}
```

```
/*****************************************u*****************************************/
/*****************************************************************************/
/*****************************************************************************/
/*****************************************************************************/

#include "stdio.h"
#include "pwl.h"

extern int n,ix,jx,*iz,*jz,*im,*jm,*in,*dim,*din,*dip,*diq;
extern double *c,*beta;
extern double *h,*alpha;
extern char *xx[],*zz[];

/*****************************************************************************/
/* Determine whether the canonical pwl model of a multiport element        */
/* possess partition boundaries with lattice structure; if so, convert it  */
/* to a lattice structure element for computational efficiency. This new   */
/* converted lattice structure element is then appended to the lattice     */
/* structure part of the canonical pwl equation.                           */
/*****************************************************************************/

lattice()
{
    int i,*la,kx[20];
    double cx[20];

    calloci(jx,&la,"la");
    for (i=0;i<jx;i++)
    {
        la[i]=0;

        if (detect_lattice(i,kx)==-1)
            break;              /* semi-lattice structure element */
        else
            la[i]=i;            /* lattice structure element /

        /* convert semi-lattice structure parameters to lattice */
        /* structure parameters : alpha to beta and h to c      */
        move_alpha_h(i,kx,cx);

        /* convert the semi-lattice structure variables to */
        /* lattice structure variables                     */
        move_table(i,kx,cx);
    }

    /* reorder the semi-lattice parameters */
    re_order(la);

}

/*****************************************************************************/
/* Reorder the parameters in the semi-lattice structure part of the        */
/* canonical piecewise-linear equation.                                    */
/*****************************************************************************/

re_order(la)
```

```
int *la;        /* multiport element indices; 1 if lattice structure */
{
    int i=0,j,k;

    while (i<jx)
    {
        if (la[i]==0)   /* the i-th element possesses no lattice structure */
            i++;
        else            /* possesses lattice structure */
        {

            /* move the parameters of the last semi-lattice element to */
            /* the i-th element which has been detected to possess the */
            /* lattice structure and all the corresponding parameters  */
            /* have been moved to the lattice structure part           */
            la[i]=la[jx-1];
            in[i]=in[jx-1];
            jm[i]=jm[jx-1];
            for (j=0;j<in[i];j++)
                zz[dip[i]+j]=zz[dip[jx-1]+j];
            for (j=0;j<=in[i];j++)
                jz[dip[i]+i+j]=jz[dip[jx-1]+jx-1+j];
            for (j=0;j<jm[i];j++)
            {
                for (k=0;k<=in[i];k++)
                    alpha[din[i]+j*(in[i]+1)+k]
                        =alpha[din[jx-1]+j*(in[jx-1]+1)+k];
                for (k=0;k<n;k++)
                    h[(diq[i]+j)*n+k]=h[(diq[jx-1]+j)*n+k];
            }
            jx--;       /* decrement the # of semi-lattice elements */
        }
    }
}

/*****************************************************************************/
/* Order a set of m numbers cx[id[0]], cx[id[1]],...., cx[id[m-1]] to the  */
/* increasing order such that                                                */
/*              cx[ie[0]] < cx[ie[1]] < ... < cx[ie[m-1]]                    */
/*****************************************************************************/

order_index(m,id,cx,ie)

int m;          /* # of objects to be ordered */
int id[];       /* old index */
int ie[];       /* new index with increasing order */
double cx[];    /* array for the objects to be ordered */
{
    double y[10];
    int i,j,is[10];

    for (i=0;i<m;i++)
        y[i]=cx[id[i]];
    for (i=0;i<m;i++)
    {
```

```
        is[i]=0;
        for (j=0;j<m;j++)
            if (y[j]<=y[is[i]]) is[i]=j;
        y[is[i]]=1e6;
    }
    for (i=0;i<m;i++)
        ie[i]=id[is[i]];
}
/*****************************************************************************/
/* Check whether a multiport element possesses a lattice structure in its  */
/* canonical pwl model; if so, return 1.                                     */
/*****************************************************************************/

detect_lattice(i,kx)
int i,*kx;
{
    int j,k,num;

    /* detect whether possessing lattice structure which exists */
    /* only when it has a single nonzero coefficient in each of */
    /* its partition boundary equations                         */
    for (j=0;j<jm[i];j++)
    {
        num=0;
        for (k=0;k<in[i];k++)
            if (alpha[din[i]+j*(in[i]+1)+k]!=0)
            {
                num++;
                kx[j]=k;
            }
        if (num!=1)     /* possess no lattice structure */
            return(-1);
    }
    return(1);
}

/*****************************************************************************/
/* Modify the single nonzero coefficient term in canonical pwl equation    */
/*                      h_i |<alpha_i,z> - gama_i|                           */
/* to                                                                        */
/*                    p*h_i |x_j - gama_i/p|                                 */
/* where p is the nonzero coefficient and is in the j-th component of      */
/* alpha_i.                                                                  */
/*****************************************************************************/

move_alpha_h(i,kx,cx)
int i,*kx;
double *cx;
{
    int j,l;
    double fabs();

    for (j=0;j<jm[i];j++)
    {
        cx[j]=alpha[din[i]+j*(in[i]+1)+in[i]]/alpha[din[i]+j*(in[i]+1)+kx[j]];
        for (l=0;l<n;l++)
```

```
            h[(diq[i]+j)*n+l]=fabs(alpha[din[i]+j*(in[i]+1)+kx[j]])
                              *h[(diq[i]+j)*n+l];
    }
}

/*************************************************************************/
/* Move the i-th multiport element parameters (which has been detected to  */
/* possess structure partition boundaries) to the lattice structure        */
/* part of the canonical pwl equation.                                     */
/*************************************************************************/

move_table(i,kx,cx)
int i,*kx;
double *cx;
{
    char *calloc();
    int j,k,m,l,id[10],ie[10];

    for (l=0;l<in[i];l++)
    {
        iz[ix]=jz[dip[i]+i+l+1];        /* jz to iz */
        xx[ix]=calloc(20,sizeof(char));
        xx[ix]=zz[dip[i]+l];            /* zz to xx */
        m=0;
        for (j=0;j<jm[i];j++)           /* group the same variable */
            if (kx[j]==l) id[m++]=j;
        im[ix]=m;                       /* # of breakpoints */
        dim[ix+1]=dim[ix]+im[ix];
        order_index(m,id,cx,ie);        /* order the beta parameters */
        for (k=0;k<m;k++)
        {
            beta[dim[ix]+k]=cx[ie[k]];
            for (j=0;j<n;j++)           /* h to c */
                c[(dim[ix]+k)*n+j]=h[(diq[i]+ie[k])*n+j];
        }
        ix++;
    }
}
```

```
#include "stdio.h"

/*****************************************************************************/
/* Replace the driving source of the spice file "xx...x.spc" by an           */
/* independent source with value equal to the starting value of the          */
/* driving source.                                                           */
/*****************************************************************************/

main(argc,argv)
int argc;
char *argv[];
{
    int i,k,m;
    char line[81];
    FILE *fp,*gp,*fopen();

    if (argc!=2)
        exit_message("START SPICE_FILE");
    sprintf(line,"%s.spc",*++argv);
    if ((fp=fopen(line,"r"))==NULL)
    {
        printf("CAN'T OPEN THE INPUT FILE %s\n",line);
        exit();
    }
    if ((gp=fopen("temp.spc","w"))==NULL)
    {
        printf("CAN'T OPEN THE FILE temp.spc\n");
        exit();
    }
    while (fgets(line,80,fp)!=NULL)
    {
        if (line[0]=='V' || line[0]=='I')
        {
            k=find_index("(",line);
            if (k>=0)   /* a driving source */
            {
                /* replace the driving source by the starting value */
                m=find_index(",",line);
                for (i=k;i<m-2;i++)
                    line[i]=line[i+2];
                line[m-2]='\n';
                line[m-1]='\0';
            }
        }
        fprintf(gp,"%s",line);
    }
    fclose(fp);
    fclose(gp);
}
```