NONLINEAR ELECTRONICS (NOEL) PACKAGE 7: CANONICAL PIECEWISE-LINEAR TRANSIENT ANALYSIS

by

An-Chang Deng and Leon O. Chua

Memorandum No. UCB/ERL M86/34

21 April 1986

NONLINEAR ELECTRONICS (NOEL) PACKAGE 7:

CANONICAL PIECEWISE-LINEAR TRANSIENT ANALYSIS

by

An-Chang Deng and Leon O. Chua

Memorandum No. UCB/ERL M86/34

21 April 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

NOEL PACKAGE 7 : CANONICAL PIECEWISE-LINEAR TRANSIENT 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 applies the BDF algorithm[1] to solve the piecewise-linear dynamic circuits. It is shown to possess the computational efficiency of canonical piecewise-linear analysis, and the reliability of the BDF algorithm.

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 7 : CANONICAL PIECEWISE-LINEAR TRANSIENT ANALYSIS

1. Introduction

The BDF algorithm introduced in [1] for solving the general dynamic circuit is the most reliable integration routine for handling the stiff algebraic-differential equation. It, however, is computationally inefficient since several Newton-Raphson iterations are required for computing the solution at each timing point. Moreover, like the DC analysis in [2], it also requires an intermediate step to formulate the circuit equation

$$\mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}, t) = 0 \tag{1}$$

to a C source code, which then is compiled and linked to the BDF simulation routine. The compiling and linking processes always take extra cpu time in addition to the computation time. In this package, we present a different approach, called PWL-BDF algorithm, which applies the BDF algorithm to perform the canonical piecewise-linear analysis on piecewise-linear dynamic circuits. It is shown to possess the computational efficiency of canonical piecewise-linear analysis and the reliability of BDF algorithm.

Similar to the canonical piecewise-linear DC analysis in [3], we assume each nonlinear element in the dynamic circuit has been modeled by a canonical piecewise-linear representation through the canonical piecewise-linear modeling process[4]; namely,

$$y = a + bx + \sum_{j=1}^{\sigma} | x - \beta_j | \tag{2}$$

where

1. $(x, y) = (v, i)$ (resp.; $(x, y) = (i, v)$) for a 2-terminal voltage-controlled resistor (resp.; current-controlled resistor)
2. $(x, y) = (v, q)$ for a 2-terminal capacitor
3. $(x, y) = (i, \phi)$ for a 2-terminal inductor

With each nonlinear element modeled by the representation of Eq.(2), the dynamic circuit can be represented as the canonical piecewise-linear differential equation

$$\mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}, t) = \mathbf{s}(t) + \mathbf{A}\mathbf{x}_d + \mathbf{B}\dot{\mathbf{x}}_d + \mathbf{C}\mathbf{x}_r + \mathbf{F}\mathbf{x}_s + \sum_{j=1}^{n_r}\sum_{i=1}^{\sigma_j} \mathbf{d}_{ji} \, | x_{rj} - \beta_{ji} | + \sum_{j=1}^{n_d}\sum_{i=1}^{r_j} \mathbf{e}_{ji} \, sgn(x_{dj} - \gamma_{ji})\dot{\mathbf{x}}_{dj} = 0 \tag{3}$$

where

$$\mathbf{s} \in \mathbf{R}^n, \quad \mathbf{A} \in \mathbf{R}^{n \times n_d}, \quad \mathbf{B} \in \mathbf{R}^{n \times n_d}, \quad \mathbf{C} \in \mathbf{R}^{n \times n_r}, \quad \mathbf{F} \in \mathbf{R}^{n \times n_s}$$

$$\mathbf{d}_{ji} \in \mathbf{R}^n, \quad \mathbf{e}_{ji} \in \mathbf{R}^n, \quad \beta_{ji} \in \mathbf{R}^1, \quad \gamma_{ji} \in \mathbf{R}^1$$

$$\mathbf{x}_r \in \mathbf{R}^{n_r}, \quad \mathbf{x}_d \in \mathbf{R}^{n_d}, \quad \mathbf{x}_s \in \mathbf{R}^{n_s}, \quad \mathbf{x} = [\mathbf{x}_d{}^T \ \mathbf{x}_r{}^T \ \mathbf{x}_s{}^T]^T \in \mathbf{R}^n$$

and $\mathbf{x}_{dj}$ (resp.; $\mathbf{x}_{rj}$) is the j-th component of $\mathbf{x}_d$ (resp.; $\mathbf{x}_r$); n_d, n_r, and n_s are the number of dynamic elements (linear or nonlinear), piecewise-linear resistors, and time-varying sources, respectively.

The canonical equation (3) can be reduced to a linear differential equation

$$\mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}, t) = \mathbf{w}^{(k)} + \mathbf{U}^{(k)}\mathbf{x} + \mathbf{V}^{(k)}\dot{\mathbf{x}} = 0 \tag{4}$$

in each region $R^{(k)}$ where each absolute function $| \, . \, |$ and each sign function sgn (.) are uniquely defined.

The solution trajectory in $R^{(k)}$ can then be easily found by using 2nd order BDF algorithm which replaces the derivative at $t = t_{l+1}$ by

$$\dot{\mathbf{x}}_{l+1} = \alpha_0 \mathbf{x}_{l+1} + \alpha_1 \mathbf{x}_l + \alpha_2 \mathbf{x}_{l-1} \tag{5}$$

and Eq.(4) is reduced to a linear differential equation

$$\mathbf{G}\mathbf{x}_{l+1} = \mathbf{g} \tag{6}$$

where

$$\mathbf{G} = \alpha_0 \mathbf{V}^{(k)} + \mathbf{U}^{(k)} \tag{7a}$$

$$\mathbf{g} = -(\mathbf{w}^{(k)} + \alpha_1 \mathbf{V}^{(k)} \mathbf{x}_l + \alpha_2 \mathbf{V}^{(k)} \mathbf{x}_{l-1}) \tag{7b}$$

and $\mathbf{x}_l$, $\mathbf{x}_{l-1}$ are the solutions at t_l and t_{l-1} respectively.

In view of Eq.(6), the solution $\mathbf{x}_{l+1}$ is easily obtained by solving a linear equation only. Compared with several Newton-Raphson iterations for solving a nonlinear algebraic equation in the conventional BDF algorithm, the PWL-BDF approach is obviously superior in computational efficiency. Moreover, the piecewise-linear dynamic circuit is formulated as Eq.(3) which is represented in terms of the numerical coefficients instead of a C source code, and hence can skip the compiling and linking processes on the equation source code.

2. Algorithm

Step 1.

Find the generalized implicit equation

$$\mathbf{Pv} + \mathbf{Qi} + \mathbf{s} = 0 \tag{8}$$

for the linear n-port obtained by extracting each nonlinear and/or dynamic element and/or time-varying source from the circuit such that all the linear resistors, linear controlled sources, and independent sources are contained within the n-port.

Step 2.

Decode the element characteristic of each external port element :

(i) 2-terminal piecewise-linear resistor :

see Step 2 of Section 2 in [3]

(ii) 2-terminal linear capacitor :

$$i = C^* \dot{v} \tag{9}$$

where C is the capacitance.

(iii) 2-terminal voltage-controlled piecewise-linear capacitor characterized by

$$\{q = (x_0, y_0)(x_1, y_1) \ldots\ldots (x_\sigma, y_\sigma)(x_{\sigma+1}, y_{\sigma+1})\} \tag{10}$$

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

$$i = [b + \sum_{j=1}^{\sigma} c_j \, sgn\,(v - \beta_j)]\dot{v} \tag{11}$$

(iv) 2-terminal linear inductor :

$$v = L^* \dot{i} \tag{12}$$

where L is the inductance.

(v) 2-terminal current-controlled piecewise-linear inductor characterized by

$$\{phi = (x_0, y_0)(x_1, y_1) \ldots\ldots (x_\sigma, y_\sigma)(x_{\sigma+1}, y_{\sigma+1})\} \tag{13}$$

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

$$v = [b + \sum_{j=1}^{\sigma} c_j \, sgn\,(i - \beta_j)]\dot{i} \tag{14}$$

(vi) time-varying voltage source (resp.; current source) characterized by {f(t)} is decoded as $v = f(t)$ (resp.; $i = f(t)$).

Step 3.

Combine the linear n-port equation and the external port-element characteristics to form the canonical piecewise-linear differential equation (3).

Step 4.

Given the initial condition (capacitor voltage and inductor current), find the dc solution at the starting time ($t = t_0$) :

(i) Replace each capacitor (resp.; inductor) by an independent voltage source (resp.; current source) with source value equal to the given initial condition.

(ii) Replace each time-varying voltage source (resp.; current source) by a piecewise-linear resistor characterized by $\{v = (0,E)(1,E)\}$ (resp.; $\{i = (0,I)(1,I)\}$) where $E = s(t_0)$ (resp.; $I = s(t_0)$).

(iii) Perform the canonical piecewise-linear dc analysis[3] to find the dc solution at $t = t_0$, which is represented in terms of

(a) voltage of the voltage-controlled pwl resistor

(b) current of the current-controlled pwl resistor

(c) current of the time-varying voltage source

(d) voltage of the time-varying current source

Step 5.

Find the initial differential equation

$$\mathbf{w}^{(0)} + \mathbf{U}^{(0)}\mathbf{x} + \mathbf{V}^{(0)}\dot{\mathbf{x}} = 0 \tag{15}$$

corresponding to the region $R^{(0)}$ where the initial point (found in Step 4) is located; set k=0.

Step 6.

Use the Backward Euler Formula

$$\dot{\mathbf{x}}_1 = \frac{\mathbf{x}_1 - \mathbf{x}_0}{h} \tag{16}$$

to find the solution $\mathbf{x}_1$ of the linear differential equation

$$\mathbf{w}^{(k)} + \mathbf{U}^{(k)}\mathbf{x} + \mathbf{V}^{(k)}\dot{\mathbf{x}} = 0 \tag{17}$$

at $t = t_0 + h$.

Step 7.

If $\mathbf{x}_1$ is not in $R^{(k)}$, then reduce h to h/10 and go to Step 6; else go to Step 8.

Step 8.

Use the second order BDF formula to approximate

$$\dot{\mathbf{x}}_2 = \alpha_0\mathbf{x}_2 + \alpha_1\mathbf{x}_1 + \alpha_2\mathbf{x}_0 \tag{18}$$

and

$$\mathbf{x}_2{}^p = \gamma_0\mathbf{x}_1 + \gamma_1\mathbf{x}_0 + \gamma_2\mathbf{x}_{-1} \tag{19}$$

such that the approximation $\dot{\mathbf{x}}_2$ and $\mathbf{x}_2{}^p$ are exact if the solution trajectory is a 2nd order polynomial curve, where $\mathbf{x}_2$ is the solution at $t = t_2 = t_1 + h$, and $\mathbf{x}_{-1}$ is the solution at the timing point prior to t_0 (use linear approximation for $\mathbf{x}_2{}^p$ if t_0 is the starting time or a timing point corresponding to a new region).

Step 9.

Substitute Eq.(18) into Eq.(17) to solve $\mathbf{x}_2$. If $\mathbf{x}_2$ is not in $R^{(k)}$, then go to Step 14; else go to Step 10.

Step 10.

Estimate the truncation error and the error ratio

$$Err = \frac{\| \mathbf{x}_2 - \mathbf{x}_2{}^p \| * h}{t_2 - t_0}, \quad ratio = \frac{Err}{10^{-3}} \tag{20}$$

Step 11.

If ratio>2 then reduce h to h/2 and go to Step 8 to repeat solving Eq.(15); else go to Step 12.

Step 12.

If ratio<0.5 then increase h to 2h; else adjust h to h/ratio.

Step 13.

Renew the solution vectors such that $\mathbf{x}_1, \mathbf{x}_0$ are the most recent solutions at the present timing point t_1 and the previous timing point t_0; namely

$$\mathbf{x}_0 = \mathbf{x}_1, \quad \mathbf{x}_1 = \mathbf{x}_2, \quad t_0 = t_1, \quad t_1 = t_2, \quad t_2 = t_1 + h$$

If t_2 exceeds the final time, then stop the transient analysis;

else go to Step 8.

Step 14.

Find the distance ratio r such that $\mathbf{x}_b$ is located in the first boundary hit by the solution trajectory segment between $\mathbf{x}_1$ and $\mathbf{x}_2$, where

$$\mathbf{x}_b = \mathbf{x}_1 + r^*(\mathbf{x}_2 - \mathbf{x}_1) \tag{20}$$

and $0 < r \leq 1$.

Step 15.

Choose $t_0 = t_1 + r^* h$ and $\mathbf{x}_0 = \mathbf{x}_b$.

Step 16.

Renew the linear differential equation

$$\mathbf{w}^{(k+1)} + \mathbf{U}^{(k+1)}\mathbf{x} + \mathbf{V}^{(k+1)}\dot{\mathbf{x}} = 0 \tag{21}$$

corresponding to the new region $R^{(k+1)}$; increment k by 1 and go to Step 6.

3. User's Instruction

Step 1.

Create a file "xx...x.spc" which describes the piecewise-linear dynamic circuit to be analyzed and follows the rules of the input format language defined in [5] for each class of circuit elements, where "xx...x" is the filename of the input file with extension ".spc". All the linear elements or 2-terminal elements with piecewise-linear characterization can be included in "xx...x.spc"; namely

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

'C': 2-terminal capacitor (linear or pwl)

'L': 2-terminal inductor (linear or pwl)

'V': independent voltage source (time-invariant or time-varying)

'I': independent current source (time-invariant or time-varying)

'E': linear voltage-controlled voltage source

'F': linear current-controlled current source

'G': linear voltage-controlled current source

'H': linear current-controlled voltage source

Steps 2-4 are combined as a single batch process and are executed by typing the command

pwldsim xx...x

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

Step 2.

Type the command

dctrf xx...x

to transform the input file "xx...x.spc" to the equivalent resistive circuit file "dcspc.spc" (see Step 4 in Section 2). It proceeds interactively with the user as follows :

CONVERT TO DC PWL RESISTIVE CIRCUIT.....

enter the starting time

enter the starting time for transient analysis

enter the initial voltage of Cx..x

enter the initial current of Lx..x

enter the initial .....

enter the initial condition for each dynamic element (capacitor voltage and inductor current)

END OF CONVERTING PROCESS

Step 3.

Type the command

pwldc dcspc

to find the dc operating point at starting time (see Step 2 of Section 3 in [3]).

Step 4.

Type the command

pwldn xx...x

to perform the canonical piecewise-linear transient analysis with the following interactive

procedures :

CANONICAL PWL TRANSIENT ANALYSIS.....

See procedures of Step 5 of Section 3 in [1].

4. Output Format

The computed result is shown in the color monitor in two different modes of operation:

(1) transient waveform vs time

Each of the following variables can be chosen by the user as the output variable to show its transient waveform on the screen, with a particular designated color and scaling factor for fitting into the graphic box.

(i) voltage of voltage-controlled piecewise-linear resistor

(ii) current of current-controlled piecewise-linear resistor

(iii) capacitor voltage (linear or piecewise-linear)

(iv) inductor current (linear or piecewise-linear)

(v) current of time-varying voltage source

(vi) voltage of time-varying current source

(2) phase portrait

The user can choose arbitrary two variables from the list of variables in (1) to plot the phase portrait. One variable is designated as the x variable and the other is y variable. Either the x or y variable has its own scaling factor entered by the user for showing the phase portrait properly within the range of graphic box.

(3) output file

In addition to the graphical output, the numerical results of the computed solution are written into the output file "xx...x.out" provided the –o option is specified in the command line; namely,

pwldn -o xx...x

in Step 4, or

pwldsim -o xx...x

in the batch command.

5. Examples

Example 1 : in file "ex1.spc"

An impasse circuit with jump behavior (Fig.1).

Example 2 : in file "ex2.spc"

A dynamic circuit with piecewise-linear resistor and capacitor (Fig.2).

Example 3 : in file "ex3.spc"

The double scroll chaotic circuit (in Fig.3).

6. Diagnosis

1. See Section 6 of [6].
2. PWLDN SPICE_FILE

   Bad command line, the correct one should be

   pwldn xx...x

   where "xx...x.spc" is the input file.
3. CAN'T OPEN THE SPICE FILE

   Can't open the input spice file "xx...x.spc".
4. TOO MANY TIME-VARYING SOURCES; INCREASE NS TO mm..m

   The number of time-varying sources is is beyond the limit NS which should be increased to mm..m.
5. TOO MANY PWL RESISTORS; INCREASE DIM TO mm..m

   The number of pwl resistors exceeds the upper bound DIM which should be increased to mm..m.
6. TOO MANY PWL DYNAMIC ELEMENTS; INCREASE DIM TO mm..m

   The number of pwl dynamic elements exceeds the upper bound DIM which should be increased to mm..m.
7. INSUFFICIENT SPACES ALLOCATED; INCREASE nn..n to mm..m.

   mm..m spaces are required which exceed the upper bound nn..n.
8. UNDEFINED ELEMENT TYPE

   The element type is not in the class of allowed elements listed in Step 1 of Section 3.
9. TOO MANY DIGITS IN PWL MODEL

   The numerical data mm..m in the pwl model has too many digits; it should not exceed 20 digits.
10. TOO MANY BREAKPOINTS IN PWL MODEL

    The pwl model has too many breakpoints which exceeds the upper bound 10.
11. MISSING '}' IN THE PWL MODEL

    The model description of a pwl element is not included within a pair of brackets "{.....}"
12. BOUNDARY CROSSING PROBLEM AT INITIAL POINT

    The solution trajectory at the starting point crosses a boundary; reduce the initial stepsize or perturb the starting point.
13. CAN'T READ THE FILE dc.op

    The file of the starting operating point is not available.
14. CAN'T READ THE STARTING TIME

    The starting time stored in the file "dc.op" is not available.
15. INITIAL POINT IS IN UNDEFINED REGION

    Can't find the initial region where the initial point is located; should check the breakpoints in each pwl model to see whether they are in the sequential order.
16. UNABLE TO EVALUATE THE TIME-VARYING SOURCE

    Improper description of the time-varying source characteristic.
17. WARNING MESSAGE : CLOSE TO A NUMERICAL IMPASSE POINT

    The solution trajectory gets stuck in a boundary; the PWL-BDF routine will restart the analysis by choosing the boundary point as the initial point.

18. WARNING MESSAGE : CLOSE TO A SHARP TURNING POINT
The solution trajectory reaches a region where the transient time changes abruptly.

References

[1] A.C.Deng and L.O.Chua, "NOnlinear ELectronics package 4 : nonlinear transient analysis," ERL Memo., M86, University of California, Berkeley, 1986.

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

[3] A.C.Deng and L.O.Chua, "NOnlinear ELectronics package 6 : canonical piecewise-linear DC analysis," ERL Memo., M86, University of California, Berkeley, 1986.

[4] A.C.Deng and L.O.Chua, "NOnlinear ELectronics package 5 : canonical piecewise-linear modeling," ERL Memo., M86, University of California, Berkeley, 1986.

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

[6] 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 An impasse circuit with jump behavior.

Fig.2 A dynamic circuit with piecewise-linear resistor and capacitor.

Fig.3 The double scroll chaotic circuit.

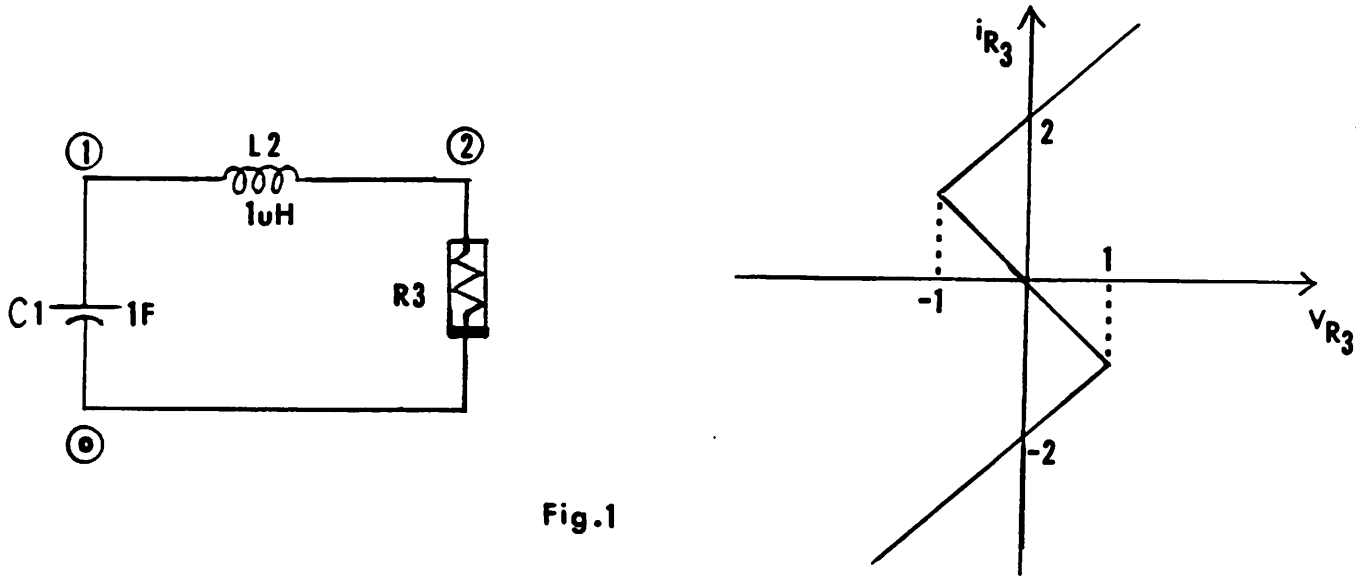


Fig.1

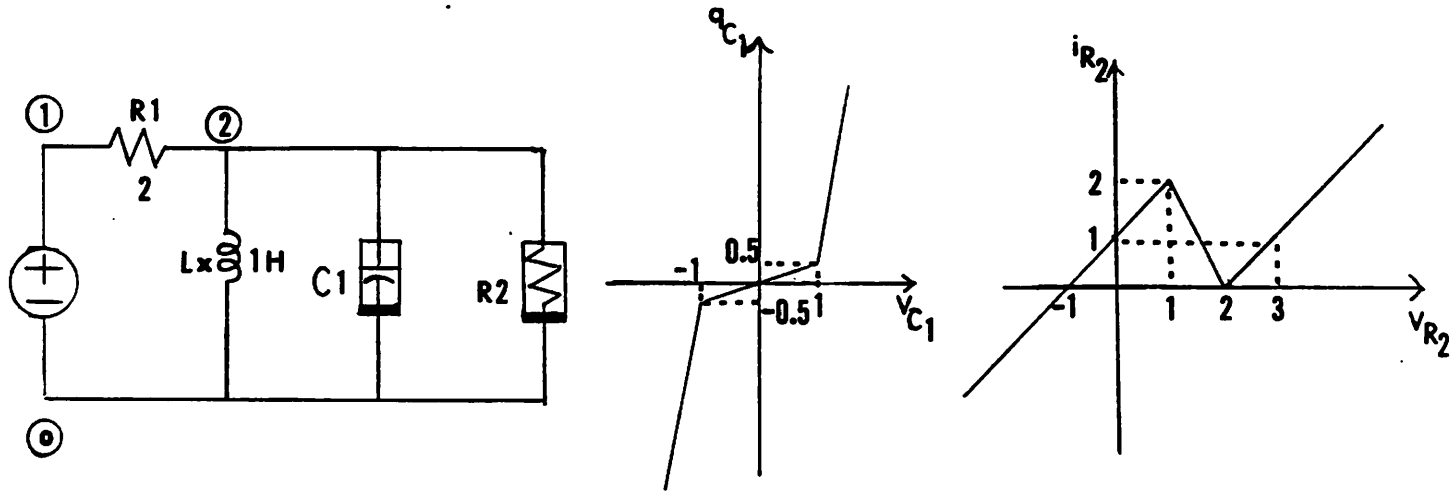


Fig.2

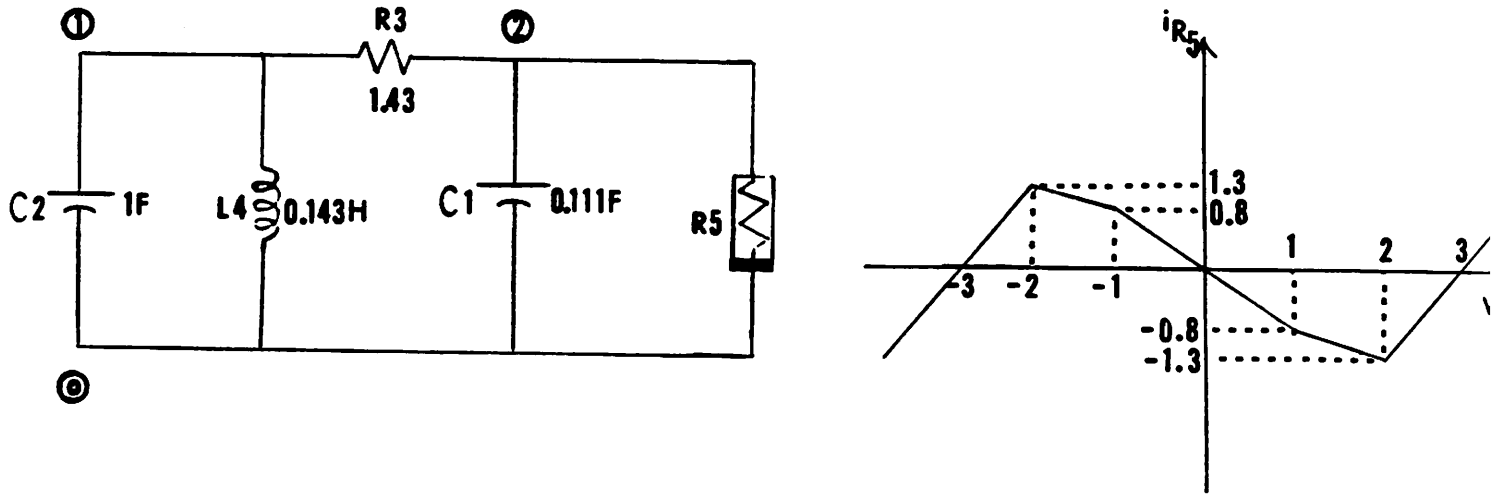
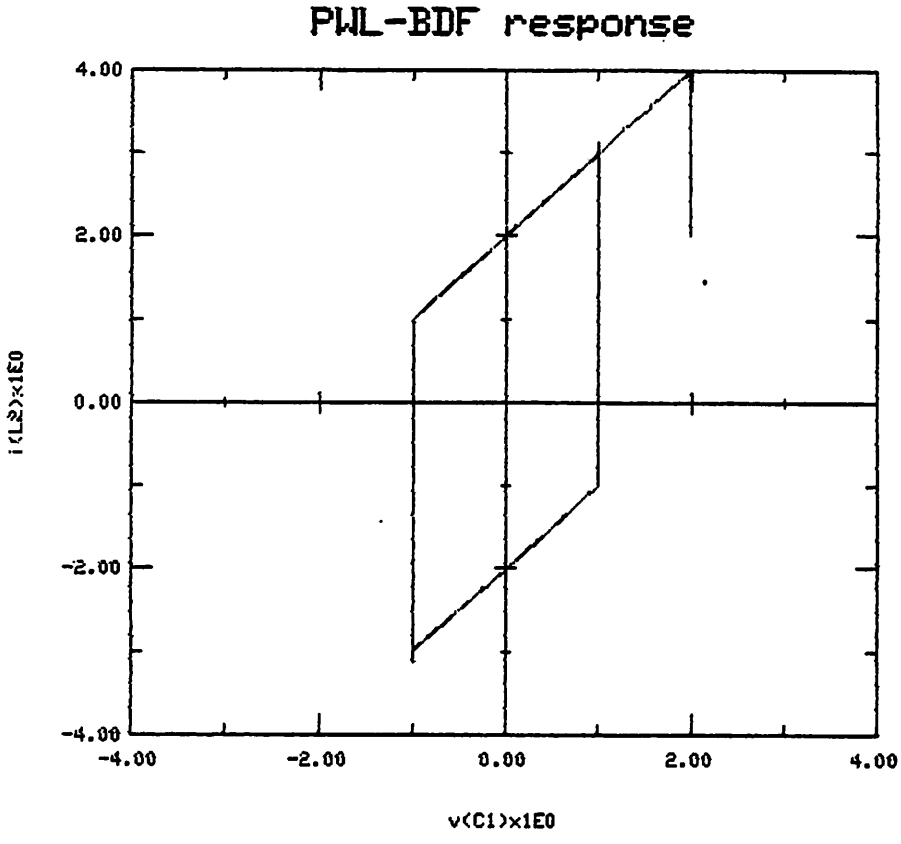
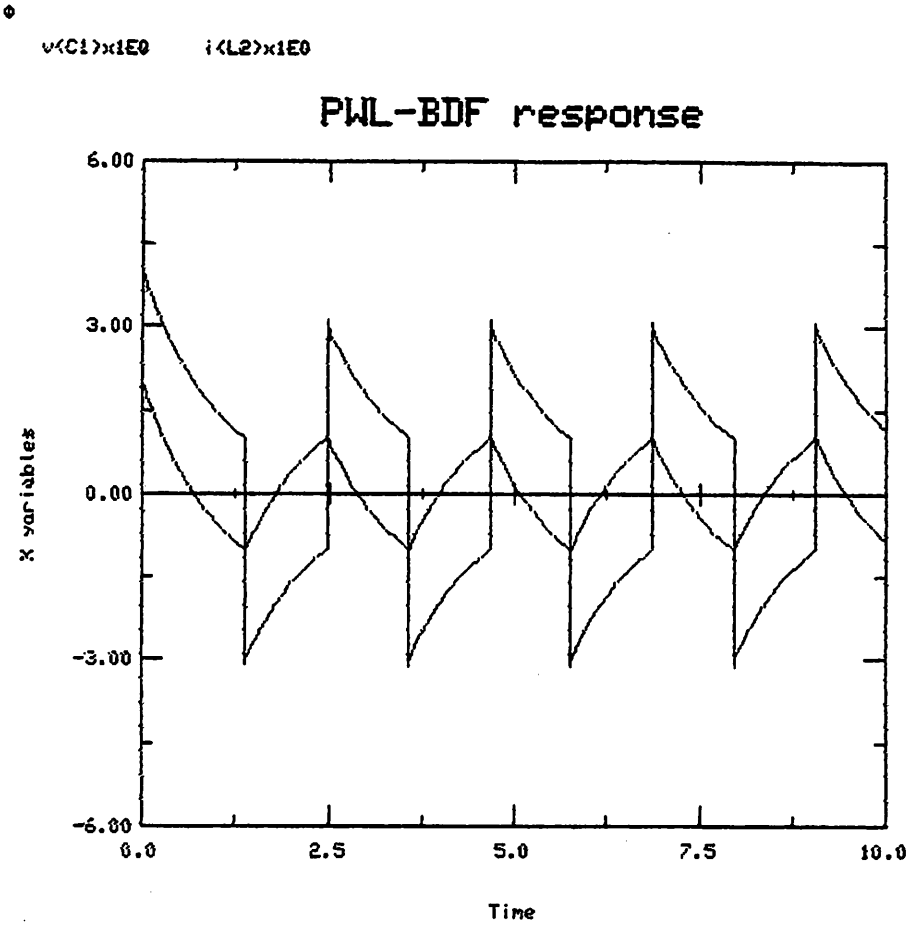


Fig.3

```
* Example 1
*
* impasse circuit with jump behavior
C1 1 0 1
L2 1 2 1u
R3 2 0 {v=(-2,0)(-1,1)(1,-1)(2,0)}
.end
```

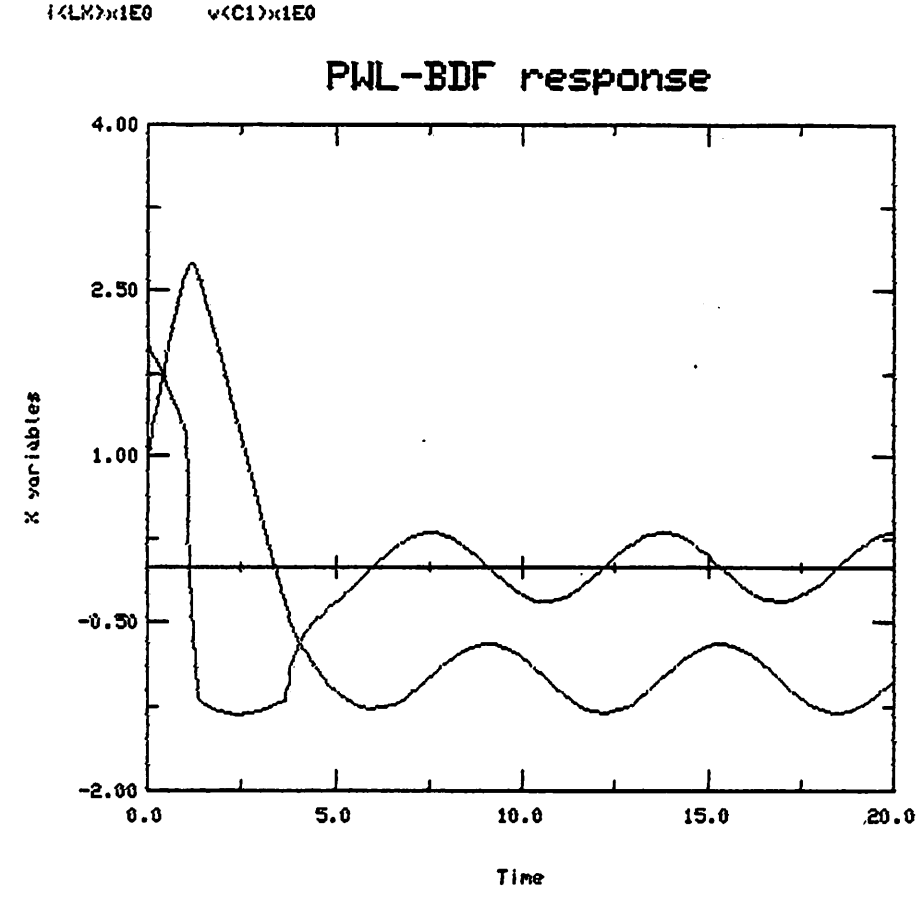
```
CONVERT TO DC PWL RESISTIVE CIRCUIT.....

enter the starting time
0
enter the initial voltage of C1
0.1
enter the initial current of L2
0.2

END OF CONVERTING PROCESS

CANONICAL PWL DC ANALYSIS : FINDING DC OPERATING POINT

*************************************************************
x[0]=i(R3)
*************************************************************
enter the initial point
i(R3)= 3
the solution is
i(R3)=2.000e-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

CANONICAL PWL TRANSIENT ANALYSIS.....
enter the initial stepsize
1e-9
enter the final time
10
phase portrait? y/n
y
x[0]=v(C1)
x[1]=i(L2)
x[2]=i(R3)
x_variable=x[?]
0
scaling factor for v(C1) = 1E?
0
y_variable=x[?]
1
scaling factor for i(L2) = 1E?
0
enter xmin and xmax
-4 4
enter ymin and ymax
-4 4

END OF CANONICAL PWL TRANSIENT ANALYSIS
```


PWL-BDF response
4.00
2.00
0.00
-2.00
-4.00
i(L2)x1E0
-4.00
-2.00
0.00
2.00
4.00
v(C1)x1E0


v(C1)x1E0
i(L2)x1E0
PWL-BDF response
6.00
3.00
0.00
-3.00
-6.00
X variables
0.0
2.5
5.0
7.5
10.0
Time

```
* Example 2
*
* dynamic circuit with piecewise-linear resistor and capacitor
R2 2 0 {i=(0,1)(1,2)(3,0)(4,1)}
C1 2 0 {q=(-2,-4.5)(-1.2,-0.5)(1.2,0.5)(2,4.5)}
LX 2 0 1
R1 1 2 2
*
* sinusoidal driving source
Vin 1 0 {sin(t)}
.end
```

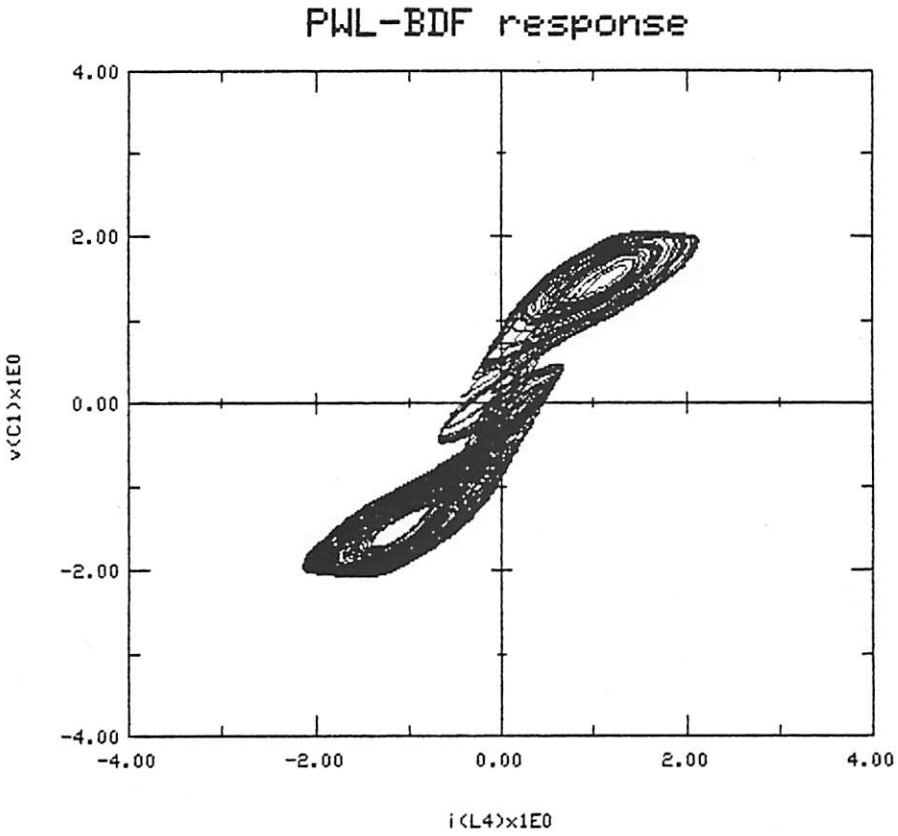
```
CONVERT TO DC PWL RESISTIVE CIRCUIT.....

enter the starting time
0
enter the initial voltage of C1
1
enter the initial current of LX
2

END OF CONVERTING PROCESS


CANONICAL PWL DC ANALYSIS : FINDING DC OPERATING POINT

************************************************************
x[0]=v(R2)
x[1]=i(RVin)
************************************************************
enter the initial point
v(R2)= 3
i(RVin)= 2
the solution is
v(R2)=1.000e+00
i(RVin)=5.000e-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


CANONICAL PWL TRANSIENT ANALYSIS.....
enter the initial stepsize
1e-4
enter the final time
20
phase portrait? y/n
n
draw i(LX) ? y/n
y
scaling factor for i(LX) = 1E?
0
draw v(C1) ? y/n
y
scaling factor for v(C1) = 1E?
0
draw v(R2) ? y/n
n
draw i(Vin) ? y/n
n
enter xmin and xmax
0 20
enter ymin and ymax
-2 4
```


i(LX)x1E0
v(C1)x1E0
PWL-BDF response
4.00
2.50
1.00
-0.50
-2.00
X variables
0.0
5.0
10.0
15.0
20.0
Time

```
* Example 3
*
* Double scroll chaotic circuit
C2 1 0 1
L4 1 0 0.143
R3 1 2 1.43
C1 2 0 0.111
R5 2 0 {i=(-3,0)(-2,1.3)(-1,0.8)(1,-0.8)(2,-1.3)(3,0)}
.end
```

```
CONVERT TO DC PWL RESISTIVE CIRCUIT.....

enter the starting time
0
enter the initial voltage of C2
0.2
enter the initial current of L4
0.1
enter the initial voltage of C1
0.2

END OF CONVERTING PROCESS


CANONICAL PWL DC ANALYSIS : FINDING DC OPERATING POINT

*************************************************************
x[0]=v(R5)
*************************************************************
enter the initial point
v(R5)= 3
the solution is
v(R5)=2.000e-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


CANONICAL PWL TRANSIENT ANALYSIS.....
enter the initial stepsize
1e-4
enter the final time
400
phase portrait? y/n
y
x[0]=v(C2)
x[1]=i(L4)
x[2]=v(C1)
x[3]=v(R5)
x_variable=x[?]
1
scaling factor for i(L4) = 1E?
0
y_variable=x[?]
2
scaling factor for v(C1) = 1E?
0
enter xmin and xmax
-4 4
enter ymin and ymax
-4 4

END OF CANONICAL PWL TRANSIENT ANALYSIS
```

Φ


PWL-BDF response
4.00
2.00
0.00
-2.00
-4.00
v(C1)x1E0
-4.00
-2.00
0.00
2.00
4.00
i(L4)x1E0

APPENDIX

SOURCE CODE LISTINGS

```
#define NODE 50
#define DIM 10
#define NS 5
#define DIMA 100
#define DIMB 200
#define DIMC 40
struct INLINE
{
   char name[9];
   int port;
   int node1,node2,node3,node4,node5,node6,node7,node8;
   char *relation;
};
struct B_VECTOR
{
   int a,b;
};
typedef char *STRING;
```

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

int n;          /* dimension of canonical pwl diff. eq. */
int mn;         /* # of model lines in the spice input file */
int ir;         /* # of pwl resistor */
int ix;         /* # of dynamic element */
int is;         /* # of time-varying source */
int *r_brk;     /* r_brk[i] = # of breakpoints in the i-th pwl resistor */
int *x_brk;     /* x_brk[i] = # of breakpoints in the i-th dynamic element */
int *nr;        /* nr[j] = r_brk[0] + r_brk[1] + ... + r_brk[j-1] */
int *nx;        /* nx[j] = x_brk[0] + x_brk[1] + ... + x_brk[j-1] */
int *jx;        /* jx[i] = k if and only if the present solution is */
                /* located in the k-th segment of the               */
                /*      i-th pwl dynamic element   if i<ix          */
                /*      (i-ix)-th pwl resistor     if i>=ix         */

/* parameters of the canonical pwl differential equation           */
/*      s + src*s(t) + a*x + b*x' + c*z + sc*zs                    */
/*      + sum d_ij*|z_i - beta_ij| + sum e_ij*|x_i - gamma_ij| = 0 */
double *s,*a,*b,*c,*d,*e,*beta,*gamma,*sc,*src;

double *p,*q;   /* parameters of generalized hybrid equation */

double *f,*g;   /* parameters of the linear differential equation */
                /*      s + a*x + f*x' + g*z = 0                  */
                /* in a certain region                            */

double *u;      /* parameter of linear matrix equation u*x=v */
double *x2;     /* solution of next timing point */
double *x1;     /* solution of present timing point */
double *x;      /* solution of previous timing point */
double *x0;     /* solution of two timing points prior to the */
                /* present timing point                       */
double *xp;     /* predicted solution for next timing point */
double t;       /* present time */
double tf;      /* final time for transient analysis to terminate */
double h;       /* next time stepsize */
double h1;      /* last time stepsize */
double *ft;     /* ft[i] = source value of the i-th time-varying source */
                /* at the next timing point                             */
char *sx[NS];   /* sx[i] = symbolic expression of the i-th time-varying */
                /* source                                               */

/* string representations for model description or circuit variables */
char *model[30],xr[DIM][12],xd[DIM][12],xs[NS][12];

struct INLINE branch[70];
struct B_VECTOR branch_vector[70];

int outfile=0;          /* outfile changes to 1 if -o option is present */
FILE *op;               /* pointer to the output file "xx...x.out" */

/*************************************************************************/
/* Canonical piecewise-linear transient analysis for dynamic circuits.   */
/*************************************************************************/
```

```
main(argc,argv)
int argc;
char *argv[];
{
    FILE *fp;

    printf("\nCANONICAL PWL TRANSIENT ANALYSIS.....\n");

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

    /* formulate the circuit to a canonical piecewise-linear */
    /* differential equation                                 */
    pwl_form(fp);
    fclose(fp);

    /* solve the canonical pwl differential equation */
    pwl_comp();

    if (outfile == 1)
        fclose(op);
    printf("\nEND OF CANONICAL PWL TRANSIENT ANALYSIS\n");
}

/*************************************************************************/
/* Open the spice input file "xx...x.spc".                               */
/*************************************************************************/

open_spice_file(argc,argv,fp)
int argc;
char *argv[];
FILE **fp;
{
    FILE *fopen();
    char *s,line[20];

    /* check the option */
    while (--argc > 0 && (*++argv)[0] == '-')
        for (s = argv[0]+1; *s != '\0'; s++)
            switch(*s)
            {
                case 'o' :
                    outfile=1;
                    break;
                default :
                    printf("ILLEGAL OPTION %c\n",*s);
                    argc=0;
                    break;
            }

    /* incorrect command line */
    if (argc!=1)
        exit_message("PWLDN SPICE_FILE");

    sprintf(line,"%s.spc",*argv);
```

```
    callocd(DIMA,&c,"c");
    callocd(DIMB,&d,"d");
    callocd(DIMA,&e,"e");
    callocd(DIMC,&beta,"beta");
    callocd(DIMC,&gamma,"gamma");
    callocd(DIMC,&sc,"sc");
    callocd(DIMC,&src,"src");
}

/*************************************************************************/
/* Allocate spaces for the parameters of the canonical pwl differential    */
/* equation with exact sizes.                                              */
/*************************************************************************/

post_allx()
{
    rallocd(&a,n*ix,DIMA);
    rallocd(&b,n*ix,DIMA);
    rallocd(&c,n*ir,DIMA);
    rallocd(&d,n*nr[ir],DIMB);
    rallocd(&e,n*nx[ix],DIMA);
    rallocd(&beta,nr[ir],DIMC);
    rallocd(&gamma,nx[ix],DIMC);
    rallocd(&sc,n*is,DIMC);
    rallocd(&src,n*is,DIMC);
    callocd(n*ix,&g,"g");
    callocd(n*ir,&f,"f");
    callocd(n*n,&u,"u");
    callocd(n,&x,"x");
    callocd(n,&x0,"x0");
    callocd(n,&x1,"x1");
    callocd(n,&x2,"x2");
    callocd(n,&xp,"xp");
    callocd(is,&ft,"ft");
    calloci(ir+ix,&jx,"jx");
}

/*************************************************************************/
/* Check whether enough spaces have been allocated in pre_allx().          */
/*************************************************************************/

check_dim()
{
    if (is >= NS)
    {
        printf("TOO MANY TIME-VARYING SOURCES; INCREASE NS TO %d\n",is);
        exit();
    }
    if (ir >= DIM)
    {
        printf("TOO MANY PWL RESISTORS; INCREASE DIM TO %d\n",ir);
        exit();
    }
    if (ix >= DIM)
    {
        printf("TOO MANY DYNAMIC ELEMENTS; INCREASE DIM TO %d\n",ix);
```

```
    if ((*fp=fopen(line,"r"))==NULL)
    {
        printf("CAN'T OPEN THE SPICE FILE %s\n",line);
        exit();
    }
    if (outfile == 1)
    {
            sprintf(line,"%s.out",*argv);
        op=fopen(line,"w");
    }
}

/*************************************************************************/
/* Formulate the dynamic circuit into a canonical pwl differential equation*/
/*              s + src*s(t) + a*x + b*x' + c*z + sc*zs                  */
/*              + sum d_ij*|z_i - beta_ij + sum e_ij*|x_i - gamma_ij| = 0  */
/*************************************************************************/

pwl_form(fp)
FILE *fp;
{
    /* find the generalized hybrid equation for the linear n-port */
    n=70;
    mn=30;
    n_port(fp,model,branch,branch_vector,&p,&q,&s,&n,&mn);

    /* allocate spaces for equation parameters with rough size */
    pre_alx();

    /* decode the element characteristics for the nonlinear */
    /* and/or dynamic elements                              */
    ir=0;
    ix=0;
    is=0;
    decode_port();

    /* check whether enough spaces have been allocated */
    check_dim();

    /* allocate equation parameters with exact dimensions */
    post_allx();
}

/*************************************************************************/
/* Allocate spaces for the parameters of the canonical pwl differential    */
/* equation before the exact sizes are available.                          */
/*************************************************************************/

pre_alx()
{
    calloci(n,&r_brk,"r_brk");
    calloci(n,&x_brk,"x_brk");
    calloci(n+1,&nr,"nr");
    calloci(n+1,&nx,"nx");
    callocd(DIMA,&a,"a");
    callocd(DIMA,&b,"b");
```

```
        exit();
    }
    if (n*ix >= DIMA)
    {
        printf("INSUFFICIENT SPACE ALLOCATED; INCREASE DIMA TO %d\n",n*ix);
        exit();
    }
    if (n*ir >= DIMA)
    {
        printf("INSUFFICIENT SPACE ALLOCATED; INCREASE DIMA TO %d\n",n*ir);
        exit();
    }
    if (n*nx[ix] >= DIMA)
    {
        printf("INSUFFICIENT SPACE ALLOCATED; INCREASE DIMA TO %d\n",
                n*nx[ix]);
        exit();
    }
    if (n*nr[ir] >= DIMA)
    {
        printf("INSUFFICIENT SPACE ALLOCATED; INCREASE DIMB TO %d\n",
                n*nr[ir]);
        exit();
    }
    if (nr[ir] >= DIMC)
    {
        printf("INSUFFICIENT SPACE ALLOCATED; INCREASE DIMC TO %d\n",nr[ir]);
        exit();
    }
    if (nx[ix] >= DIMC)
    {
        printf("INSUFFICIENT SPACE ALLOCATED; INCREASE DIMC TO %d\n",nx[ix]);
        exit();
    }
    if (n*is >= DIMC)
    {
        printf("INSUFFICIENT SPACE ALLOCATED; INCREASE DIMC TO %d\n",
                n*is);
        exit();
    }
}
```

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

extern char xr[][12],xd[][12],xs[][12],*sx[];
extern int n,ir,ix,is,*nr,*nx,*r_brk,*x_brk;
extern double *a,*b,*c,*d,*e,*beta,*gamma,*sc,*src;
extern double *p,*q,*s;
extern struct INLINE branch[];
extern struct B_VECTOR branch_vector[];

/*************************************************************************/
/* Decode the model of each nonlinear and/or dynamic element.            */
/*************************************************************************/

decode_port()
{
    int i,j;

    for (i=0;i<n;i++)
    {
        j=(branch_vector+i)->a;
        switch((branch+j)->name[0])
        {
            case 'R' : pwl_r(i,j); break;       /* pwl resistor */
            case 'C' :
            case 'L' : d_element(i,j); break;   /* dynamic element */
            case 'V' :
            case 'I' : source(i,j); break;      /* time-varying source */
            default  : {
                            printf("UNDEFINED ELEMENT TYPE %s\n",
                                    (branch+j)->name);
                            exit();
                        }
        }
    }
}

/*************************************************************************/
/* Decode the model of a pwl resistor into the canonical pwl function.   */
/*************************************************************************/

pwl_r(i,j)
int i,j;
{
    int m,k;
    double *zx,*zy,y[10];

    /* decode the pwl model */
    decode_1((branch+j)->relation,r_brk+ir,y);
    callocd(n,&zx,"zx");
    callocd(n,&zy,"zy");

    if (find_index("(i=",(branch+j)->relation)==0)
    {
        /* voltage-controlled pwl resistor */
        for (k=0;k<n;k++)
```

```
        {
            zx[k]=p[k*n+i];
            zy[k]=q[k*n+i];
        }
        sprintf(xr[ir],"v(%s)",(branch+j)->name);
    }
    else
    {
        if (find_index("(v=",(branch+j)->relation)==0)
        {
            /* current-controlledpwl resistor */
            for (k=0;k<n;k++)
            {
                zx[k]=q[k*n+i];
                zy[k]=p[k*n+i];
            }
            sprintf(xr[ir],"i(%s)",(branch+j)->name);
        }
        else
            exit_message("INCORRECT FORMAT IN PWL RESISTOR MODEL");
    }

    /* nr[j] = r_brk[0] + r_brk[1] + ... + r_brk[j-1] */
    nr[ir+1]=nr[ir]+r_brk[ir];

    for (k=0;k<n;k++)
    {
        s[k]+=zy[k]*y[0];
        c[ir*n+k]=zx[k]+zy[k]*y[1];
    }

    for (m=0;m<r_brk[ir];m++)
    {
        for (k=0;k<n;k++)
            d[(nr[ir]+m)*n+k]=y[2+2*m]*zy[k];
        beta[nr[ir]+m]=y[3+2*m];
    }
    ir++;
    cfree(zx);
    cfree(zy);
}

/*************************************************************************/
/* Decode the model of the dynamic element.                              */
/*************************************************************************/

d_element(i,j)
int i,j;
{
    int k;
    double *zx,*zy,value,stof();

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

    /* capacitor */
```

```
    if ((branch+j)->name[0]=='C')
    {
        for (k=0;k<n;k++)
        {
            zx[k]=p[k*n+i];
            zy[k]=q[k*n+i];
        }
        sprintf(xd[ix],"v(%s)",(branch+j)->name);
    }

    /* inductor */
    else
    {
        for (k=0;k<n;k++)
        {
            zx[k]=q[k*n+i];
            zy[k]=p[k*n+i];
        }
        sprintf(xd[ix],"i(%s)",(branch+j)->name);
    }

    /* pwl dynamic element */
    if (find_index("(",(branch+j)->relation)==0)
        pwl_x(i,j,zx,zy);

    /* linear dynamic element */
    else
    {
        /* get the capacitance or inductance */
        value=stof((branch+j)->relation);
        for (k=0;k<n;k++)
        {
            a[ix*n+k]=zx[k];
            b[ix*n+k]=zy[k]*value;
        }
        x_brk[ix]=0;    /* no breakpoint */
        nx[ix+1]=nx[ix];
    }
    ix++;
    cfree(zx);
    cfree(zy);
}

/*************************************************************************/
/* Decode the pwl dynamic element.                                       */
/*************************************************************************/

pwl_x(i,j,zx,zy)
int i,j;
double *zx,*zy;
{
    int m,k;
    double y[10];

    /* decode the pwl model */
    decode_1((branch+j)->relation,x_brk+ix,y);
```

```
    /* nx[j] = x_brk[0] + x_brk[1] + ... + x_brk[j-1] */
    nx[ix+1]=nx[ix]+x_brk[ix];

    for (k=0;k<n;k++)
    {
        a[ix*n+k]=zx[k];
        b[ix*n+k]=zy[k]*y[1];
    }
    for (m=0;m<x_brk[ix];m++)
    {
        for (k=0;k<n;k++)
            e[(nx[ix]+m)*n+k]=y[2+2*m]*zy[k];
        gamma[nx[ix]+m]=y[3+2*m];
    }
}

/*************************************************************************/
/* Decode the time-varying source characteristic.                        */
/*************************************************************************/

source(i,j)
int i,j;
{
    char *calloc();
    int k;

    /* extract the characteristic of the time-varying source */
    sx[is]=calloc(strlen((branch+j)->relation)+1,sizeof(char));
    strcpy(sx[is],(branch+j)->relation+1);
    sx[is][strlen(sx[is])-1]='\0';

    if ((branch+j)->name[0]=='V')
    {
        /* time-varying voltage source */
        for (k=0;k<n;k++)
        {
            sc[is*n+k]=q[k*n+i];
            src[is*n+k]=p[k*n+i];
        }
        sprintf(xs[is],"i(%s)",(branch+j)->name);
    }
    else
    {
        /* time-varying current source */
        for (k=0;k<n;k++)
        {
            sc[is*n+k]=p[k*n+i];
            src[is*n+k]=q[k*n+i];
        }
        sprintf(xs[is],"v(%s)",(branch+j)->name);
    }
    is++;
}

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

```
/* 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,aa)
char *model;    /* model string */
int *m;         /* # of breakpoints */
double aa[];    /* numerical information in the model */
{
    char xz[21],yz[21];
    int nn=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 MODEL");
            xz[i-1]='\0';
            model++;
            if (nn>=10)
                exit_message("TOO MANY BREAKPOINTS IN PWL MODEL");
            xw[nn]=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 MODEL");
            yz[i]='\0';
            model++;
            yw[nn++]=atof(yz);
        }
    }

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

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

    /* construct the canonical pwl representation */
```

```
    canonical_pwl(*m,xw,yw,aa);
}

/*************************************************************************/
/* 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,aa)

int m;                  /* # of breakpoints */
double xw[],yw[];       /* breakpoins */
double aa[];            /* 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 */
    aa[1]=(slope[m]+slope[0])/2;
    for (i=0;i<m;i++)
    {
        aa[2+2*i]=(slope[i+1]-slope[i])/2;
        aa[3+2*i]=xw[i+1];
    }
    aa[0]=yw[0]-aa[1]*xw[0];
    for (i=0;i<m;i++)
        aa[0]=aa[0]-aa[2+2*i]*fabs(aa[3+2*i]-xw[0]);
    cfree(slope);
}
```

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

extern int n,ir,ix,is,*nr,*nx,*r_brk,*x_brk,*jx;
extern double *x,*s,*a,*b,*c,*d,*e,*g,*f,*u,*beta,*gamma;
extern double *sc,*src,h,h1,t,tf,*ft,*x1,*x2,*xp;
extern char xr[][12],xd[][12],xs[][12],*sx[];
extern int outfile;
int lx=1;
double tstart,hy,*sa;
GRAF *gp;
FILE *hp;

/*************************************************************************/
/* Canonical pwl transient analysis.                                     */
/*************************************************************************/

pwl_comp()
{
    int i,mx,kx;
    double r;

    /* save the s vector */
    callocd(n,&sa,"sa");
    for (i=0;i<n;i++)
        sa[i]=s[i];

    /* read the initial condition */
    ini_pt();
    fclose(hp);

    /* determine the initial region */
    location_index();

    /* find the initial differential equation */
    ini_eq();

    t=tstart;

    /* draw the graphic axises, labels, and the title name */
    draw_graf(&gp);

    /* compute the solution of the next timing point */
    start(h);
    if (boundary(&mx,&kx,&r,x,x1) == -1)
        exit_message("BOUNDARY CROSSING PROBLEM AT INITIAL POINT");
    t+=h;

    /* use bdf formula to perform the transient analysis */
    while (t<tf)
    {
        if (tr_analysis() == -1)
            re_start();
    }
}
```

```
/*************************************************************************/
/* Restart the analysis with the previous exit point as the starting point */
/* when a numerical impasse point or very sharp turning point is reached.  */
/*************************************************************************/

re_start()
{
    location_index();
    tstart=t;
    ini_eq();
    h=1e-8;
    start(h);
    t+=h;
}


/*************************************************************************/
/* Read the starting time, final time, initial stepsize, and the initial   */
/* condition.                                                              */
/*************************************************************************/

ini_pt()
{
    int i;

    get_ini_t();
    printf("enter the initial stepsize\n");
    scanf("%lf",&h);
    if (outfile == 1)
    {
        printf("enter the printing stepsize for the output file\n");
        scanf("%lf",&hy);
    }

    /* get the starting dc operating point */
    get_ini_dp();

    printf("enter the final time\n");
    scanf("%lf",&tf);
}

/*************************************************************************/
/* Get the starting time from the file "dc.p" which is created in the      */
/* pre_process for finding the starting point.                             */
/*************************************************************************/

get_ini_t()
{
    FILE *fopen();
    char line[80],w1[80];
    double stof();

    hp=fopen("dc.op","r");
    if (fgets(line,80,hp) == NULL)
        exit_message("CAN'T READ THE FILE dc.p");
```

```
    if (find_index("t_0=",line) != 0)
        exit_message("CAN'T READ THE STARTING TIME");
    strdel(line,0,4);
    strdel(line,strlen(line)-1,1);
    tstart=stof(line);
}

/*********************************************************************/
/* Get the initial dc operating point from the file "dc.p".          */
/*********************************************************************/

get_ini_dp()
{
    char line[100];
    int i;
    double extract();

    while (fgets(line,80,hp) != NULL)
    {
        if (line[0]=='v' || line[0]=='i')
        {
            for (i=0;i<ix;i++)
                if (find_index(xd[i],line) == 0)
                    x[i]=extract(line);
            for (i=0;i<ir;i++)
                if (find_index(xr[i],line) == 0)
                    x[i+ix]=extract(line);
            if (strlen(line)>6)
                strdel(line,2,1);
            for (i=0;i<is;i++)
                if (find_index(xs[i],line) == 0)
                    x[i+ir+ix]=extract(line);
        }
    }
}

/*********************************************************************/
/* Extract the numerical data of starting dc operating point from the  */
/* ASCII code.                                                         */
/*********************************************************************/

double extract(line)
char line[];
{
    int k;
    double zz,stof();

    k=find_index("=",line);
    strdel(line,0,k+1);
    strdel(line,strlen(line)-1,1);
    zz=stof(line);
    return(zz);
}

/*********************************************************************/
/* Determine the initial region                                      */
```

```
/*      jx[i] = k if and only if the initial point is located in the k-th  */
/* segment of the i-th dynamic element if i<ix; or the (i-ix)-th pwl       */
/* resistor if i>=ix.                                                      */
/*************************************************************************/

location_index()
{
    int i,k;

    /* dynamic element */
    for (i=0;i<ix;i++)
    {
        if (x[i]<=gamma[nx[i]])         /* leftmost segment */
            jx[i]=0;
        else
        {
            if (x[i]>gamma[nx[i+1]-1])  /* rightmost segment */
                jx[i]=x_brk[i];
            else
            {
                for (k=1;k<x_brk[i];k++)
                    if ((x[i]-gamma[nx[i]+k-1])*(x[i]-gamma[nx[i]+k])<=0)
                    {
                        jx[i]=k;
                        break;
                    }
                if (k==x_brk[i])
                    exit_message("INITIAL POINT IS IN UNDEFINED REGION");
            }
        }
    }

    /* resistive element */
    for (i=0;i<ir;i++)
    {
        if (x[i+ix]<=beta[nr[i]])       /* leftmost element */
            jx[i+ix]=0;
        else
        {
            if (x[i+ix]>beta[nr[i+1]-1])        /* rightmost segment */
                jx[i+ix]=r_brk[i];
            else
            {
                for (k=1;k<r_brk[i];k++)
                    if ((x[i+ix]-beta[nr[i]+k-1])*(x[i+ix]-beta[nr[i]+k])<=0)
                    {
                        jx[i+ix]=k;
                        break;
                    }
                if (k==r_brk[i])
                    exit_message("INITIAL POINT IS IN UNDEFINED REGION");
            }
        }
    }
}
```

```
/*************************************************************************/
/* Reduce the canonical pwl differential equation to a linear differential */
/* equation                                                               */
/*              s + a*x + f*x' + g*z = 0                                  */
/* corresponding to the initial region.                                   */
/*************************************************************************/

ini_eq()
{
    int i,j,k;

    for (i=0;i<n;i++)
        s[i]=sa[i];

    for (j=0;j<ix;j++)
        for (i=0;i<n;i++)
        {
            g[i*ix+j]=b[j*n+i];
            for (k=0;k<x_brk[j];k++)
            {
                if (jx[j]<=k)
                    g[i*ix+j]-=e[(nx[j]+k)*n+i];
                else
                    g[i*ix+j]+=e[(nx[j]+k)*n+i];
            }
        }
    for (j=0;j<ir;j++)
        for (i=0;i<n;i++)
        {
            f[i*ir+j]=c[j*n+i];
            for (k=0;k<r_brk[j];k++)
            {
                if (jx[j+ix]<=k)
                {
                    s[i]+=beta[nr[j]+k]*d[(nr[j]+k)*n+i];
                    f[i*ir+j]-=d[(nr[j]+k)*n+i];
                }
                else
                {
                    s[i]-=beta[nr[j]+k]*d[(nr[j]+k)*n+i];
                    f[i*ir+j]+=d[(nr[j]+k)*n+i];
                }
            }
        }
}

/*************************************************************************/
/* Use the Backward Euler Formula                                         */
/*              x1' = (x1 - x0)/hx                                        */
/* to compute the solution of the differential equation in the initial    */
/* region.                                                                */
/*************************************************************************/

start(hx)
double hx;
{
```

```
    char *calloc();
    int i,j,*ipvt;
    double fst(),*zq,rcond;

    ipvt=(int *)calloc(n,sizeof(int));
    zq=(double *)calloc(n,sizeof(double));

    /* estimate the value of each time-varying source */
    for (j=0;j<is;j++)
        ft[j]=fst(sx[j],t+hx);

    /* replace x1' by (x1-x0)/hx and reduce the diff. eq. to */
    /* a linear equation                                      */
    for (i=0;i<n;i++)
    {
        x1[i] = -1*s[i];
        for (j=0;j<ix;j++)
        {
            u[i*n+j]=a[j*n+i]+g[i*ix+j]/hx;
            x1[i]+=g[i*ix+j]*x[j]/hx;
        }
        for (j=0;j<ir;j++)
            u[i*n+j+ix]=f[i*ir+j];
        for (j=0;j<is;j++)
        {
            u[i*n+j+ix+ir]=sc[j*n+i];
            x1[i]-=src[j*n+i]*ft[j];
        }
    }
    sgeco(u,n,ipvt,&rcond,zq);
    sgesl(u,n,ipvt,x1,0);
    h1=hx;

    /* approximate the predicted value of next timing point */
    /* by the present solution                              */
    for (i=0;i<n;i++)
        xp[i]=x1[i];

    cfree(ipvt);
    cfree(zq);
}

/*************************************************************************/
/* Evaluate the value of f(x) at x=t where f(x) is represented by s_exp.   */
/*************************************************************************/

double fst(s_exp,tt)
char *s_exp;
double tt;
{
    double sym_eval(),z;
    SYM_TREE *encode(),*tree;

    tree=encode(s_exp);
    sym_set("t",tt);
    z=sym_eval(tree);
```

```
    if (sym_stat != 0)
        exit_message("UNABLE TO EVALUATE THE TIME-VARYING SOURCE");
    free_eqn(tree);
    return(z);
}

/*************************************************************************/
/* Compute z to the power of i.                                          */
/*************************************************************************/

double ipow(z,i)
int i;
double z;
{
    int j=0;
    double w=1;

    while (j<=i)
    {
        w=w*z;
        j++;
    }
    return(w);
}
```

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

extern char *sx[];
extern int n,ir,ix,is,*nr,*nx,*r_brk,*x_brk,*jx;
extern double *x0,*x,*x1,*x2,*xp,*s,*a,*b,*c,*d,*e,*f,*g;
extern double *u,*beta,*gamma,*sc,*src,h,h1,t,*ft,tstart;
extern GRAF *gp;
extern int pw,outfile;
extern double hy;
double new_h,h0,xe,bd;

/*************************************************************************/
/* Perform the transient analysis for a pwl dynamic circuit; return -1 if  */
/* a numerical impasse point or very sharp turning point is reached.       */
/*************************************************************************/

tr_analysis()
{
    int i,m,k,kx,mx,j=0,bx;
    double r;

    /* compute the next solution */
    new_mx();

    /* repeat with a smaller stepsize in case of sharp turning point */
    while (angle()==-1 && j++<2)
    {
        h=h/2;
        if (h0<1e-10)
            h0=h1;
        pred_value(x0,x,x1,h0,h1,h);
        new_mx();
    }

    bx=boundary(&m,&k,&r,x1,x2);

    /* if cross a boundary */
    if (bx == 1)
    {
        j=0;

        /* renew the linear diff. eq. for the new region */
        cross_bdry(m,k,r);

        /* start with a very small stepsize in the new region */
        h=h/50;
        start(h);

        /* reduce the stepsize if the solution returns to the */
        /* previous region                                     */
        while ((bd-xe)*(bd-x1[k])>0 && ++j<7)
        {
            h=h/10;
            start(h);
        }
```

```
        /* if the stepsize is reduced to a tiny size (1e-7 of */
        /* the original size), then it is a numerical impasse */
        /* point                                              */
        if (j==7)
        {
            printf("WARNING MESSAGE: CLOSE TO A NUMERICAL IMPASSE POINT\n");
            if (x[k]>bd)
                x[k]=bd - 1e-8;
            else
                x[k]=bd + 1e-8;
            return(-1);
        }

        j=0;
        while (boundary(&mx,&kx,&r,x,x1)==1)
        {
            if (mx==m && kx==k)
                break;
            if (j++ > 3)
            {
                printf("WARNING MESSAGE: CLOSE TO A SHARP TURNING POINT\n");
                return(-1);
            }
            h=h/10;
            start(h);
        }

        /* graphic plotting */
        if (pw==1)
            draw_phase(gp,x,x1);
        else
            draw_waveform(gp,x,x1);

        t+=h;
    }

    else
    {
        if (bx == -1)
            exit_message("BOUNDARY CROSSING PROBLEM");

        /* accept the new computed solution */
        if (det_h()==0)
        {
            if (pw==1)
                draw_phase(gp,x1,x2);
            else
                draw_waveform(gp,x1,x2);

            /* predict the next solution */
            pred_value(x,x1,x2,h1,h,new_h);

            /* renew the solution vector */
            new_sol();
        }
```

```
        else
            /* reject and repeat with a smaller stepsize */
            pred_value(x0,x,x1,h0,h1,h);
    }
    return(0);
}

/*************************************************************************/
/* Renew the solution vectors x0, x, x1 such that they are solutions at    */
/* three most recent timing points; x and x1 are required to approximate   */
/* the derivative and predict the next solution, x0 is required in case    */
/* the computed solution is rejected and should repeat with a smaller      */
/* stepsize.                                                               */
/*************************************************************************/

new_sol()
{
    int i;

    h0=h1;
    h1=h;
    h=new_h;    /* new stepsize */
    for (i=0;i<n;i++)
    {
        x0[i]=x[i];
        x[i]=x1[i];
        x1[i]=x2[i];
    }
    t+=h1;
}

/*************************************************************************/
/* Reduce the linear differential equation                                 */
/*      s + a*x_n+1 + g*(x_n+1)' + f*z_n+1 = 0                             */
/* to a linear equation                                                    */
/*              u*x_n+1 = s                                                */
/* by replacing (x_n+1)' with the approximation                            */
/*              a0*x_n+1 + a1*x_n + a2*x_n-1                               */
/*************************************************************************/

new_mx()
{
    char *calloc();
    int i,j,*ipvt;
    double r,a0,a1,a2,*zq,rcond,fst();

    ipvt=(int *)calloc(n,sizeof(int));
    zq=(double *)calloc(n,sizeof(double));

    /* coefficients a0, a1, a2 are chosen such that the approximation */
    /*          (x_n+1)' = a0*x_n+1 + a1*x_n + a2*x_n-1                */
    /* is exact if the solution trajectory is a 2nd order polynomial  */
    r=h1/h;
    a1 = -1*(1.0+r)/(h*r);
    a2=1.0/(r*(1+r)*h);
    a0 = -(a1+a2);
```

```
    /* estimate the value of the time-varying source */
    for (j=0;j<is;j++)
        ft[j]=fst(sx[j],t+h);

    for (i=0;i<n;i++)
    {
        x2[i] = -i*s[i];
        for (j=0;j<ix;j++)
        {
            u[i*n+j]=a[j*n+i]+a0*g[i*ix+j];
            x2[i]-=g[i*ix+j]*(a1*xi[j]+a2*x[j]);
        }
        for (j=0;j<ir;j++)
            u[i*n+j+ix]=f[i*ir+j];
        for (j=0;j<is;j++)
        {
            u[i*n+j+ix+ir]=sc[j*n+i];
            x2[i]-=src[j*n+i]*ft[j];
        }
    }
    sgeco(u,n,ipvt,&rcond,zq);
    sgesl(u,n,ipvt,x2,0);
    cfree(ipvt);
    cfree(zq);
}

/*********************************************************************/
/* Determine whether the solution trajectory crosses a boundary. Return 1  */
/* if it crosses the boundary and enters the m-th segment of the kx-th     */
/* element with a ratio rx, where rx is the ratio of the distance from     */
/* the present solution to the boundary over the distance from the present */
/* solution to the next solution which is out of the present region.       */
/*********************************************************************/

boundary(m,kx,rx,zp,zn)
int *m,*kx;
double *rx,*zp,*zn;
{
    char *calloc();
    int i,*index,k= -1;
    double rin=1.0e8,r,fabs();

    index=(int *)calloc(n,sizeof(int));
    for (i=0;i<ix;i++)
    {
        /* pwl dynamic element */
        if (x_brk[i]>0)
        {
            r=10.0;

            /* cross the right boundary from the leftmost region */
            if (jx[i]==0 && zn[i]>gamma[nx[i]])
            {
                r=(gamma[nx[i]]-zp[i])/(zn[i]-zp[i]);
                if (fabs(r)<1.0e-12)
```

```
                return(-1);
            if (rin>r && r>0)
            {
                k=i;
                bd=gamma[nx[i]];
                rin=r;
                index[i]=1;
            }
        }

        /* cross the left boundary from the rightmost region */
        if (jx[i]==x_brk[i] && zn[i]<gamma[nx[i]+x_brk[i]-1])
        {
            r=(gamma[nx[i]+x_brk[i]-1]-zp[i])/(zn[i]-zp[i]);
            if (fabs(r)<1.0e-12)
                return(-1);
            if (rin>r && r<1 && r>0)
            {
                k=i;
                bd=gamma[nx[i]+x_brk[i]-1];
                index[i]=jx[i]-1;
                rin=r;
            }
        }

        /* in the bounded region */
        if ((jx[i] != 0) && (jx[i] != x_brk[i]))
        {
            r=(gamma[nx[i]+jx[i]-1]-zp[i])/(zn[i]-zp[i]);
            if (fabs(r)<1.0e-12)
                return(-1);
            /* cross the left boundary */
            if (r<1 && r>0 && rin>r)
            {
                k=i;
                bd=gamma[nx[i]+jx[i]-1];
                rin=r;
                index[i]=jx[i]-1;
            }
            else
            {
                r=(gamma[nx[i]+jx[i]]-zp[i])/(zn[i]-zp[i]);
                if (fabs(r)<1.0e-12)
                    return(-1);
                /* cross the right boundary */
                if (r<1 && r>0 && rin>r)
                {
                    k=i;
                    bd=gamma[nx[i]+jx[i]];
                    rin=r;
                    index[i]=jx[i]+1;
                }
            }
        }
    }
}
```

```
    /* check whether crosses a boundary of pwl resistors */
    if (r_bdry(&k,&rin,index,zp,zn) == -1)
        return(-1);
    if (k>=0)
        *m=index[k];
    cfree(index);
    if (rin<1)
    {
        *rx=rin;
        *kx=k;
        return(1);
    }
    else
        return(0);
}

/*************************************************************************/
/* Determine whether the solution trajectory crosses a boundary of the     */
/* resistive element.                                                      */
/*************************************************************************/

r_bdry(k,rin,index,zp,zn)
int *k,*index;
double *rin,*zp,*zn;
{
    int i;
    double r,fabs();

    for (i=0;i<ir;i++)
    {
        /* pwl resistor */
        if (r_brk[i]>0)
        {
            r=10.0;

            /* cross the right boundary from the leftmost region */
            if (jx[i+ix]==0 && zn[i+ix]>beta[nr[i]])
            {
                r=(beta[nr[i]]-zp[i+ix])/(zn[i+ix]-zp[i+ix]);
                if (fabs(r)<1.0e-12)
                    return(-1);
                if (*rin>r && r>0)
                {
                    *k=i+ix;
                    bd=beta[nr[i]];
                    *rin=r;
                    index[i+ix]=1;
                }
            }

            /* cross the left boundary from the rightmost boundary */
            if (jx[i+ix]==r_brk[i] && zn[i+ix]<beta[nr[i]+r_brk[i]-1])
            {
                r=(beta[nr[i]+r_brk[i]-1]-zp[i+ix])/(zn[i+ix]-zp[i+ix]);
                if (fabs(r)<1.0e-12)
```

```
                    return(-1);
                if (*rin>r && r<1 && r>0)
                {
                    *k=i+ix;
                    bd=beta[nr[i]+r_brk[i]-1];
                    index[i+ix]=jx[i+ix]-1;
                    *rin=r;
                }
            }

            /* in the bounded region */
            if ((jx[i+ix] != 0) && (jx[i+ix] != r_brk[i]))
            {
                r=(beta[nr[i]+jx[i+ix]-1]-zp[i+ix])/(zn[i+ix]-zp[i+ix]);
                if (fabs(r)<1.0e-12)
                    return(-1);

                /* cross the left boundary */
                if (r<1 && r>0 && *rin>r)
                {
                    *k=i+ix;
                    bd=beta[nr[i]+jx[i+ix]-1];
                    *rin=r;
                    index[i+ix]=jx[i+ix]-1;
                }
                else
                {
                    r=(beta[nr[i]+jx[i+ix]]-zp[i+ix])/(zn[i+ix]-zp[i+ix]);
                    if (fabs(r)<1.0e-12)
                        return(-1);

                    /* cross the right boundary */
                    if (r<1 && r>0 && *rin>r)
                    {
                        *rin=r;
                        bd=beta[nr[i]+jx[i+ix]];
                        *k=i+ix;
                        index[i+ix]=jx[i+ix]+1;
                    }
                }
            }
        }
    }
}

/*************************************************************************/
/* Renew the linear differential equation                                */
/*              s + a*x + f*x' + g*z = 0                                 */
/* in the new region.                                                    */
/*************************************************************************/

cross_bdry(m,k,r)
int m,k;
double r;
{
    int i;
```

```
    h=r*h;
    for (i=0;i<n;i++)
        x[i]=x1[i]+r*(x2[i]-x1[i]);
    if (m>jx[k])
        x[k]+=1e-8;
    else
        x[k]-=1e-9;

    if (pw==1)
        draw_phase(gp,x1,x);
    else
        draw_waveform(gp,x1,x);
    t+=h;

    /* cross a boundary of dynamic element */
    if (k<ix)
    {
        if (m>jx[k])
            for (i=0;i<n;i++)
                g[i*ix+k]+=2*e[(nx[k]+jx[k])*n+i];
        else
            for (i=0;i<n;i++)
                g[i*ix+k]-=2*e[(nx[k]+jx[k]-1)*n+i];
    }

    /* cross a boundary of pwl resistor */
    else
    {
        if (m>jx[k])
            for (i=0;i<n;i++)
            {
                s[i]-=2*beta[nr[k-ix]+jx[k]]*d[(nr[k-ix]+jx[k])*n+i];
                f[i*ir+k-ix]+=2*d[(nr[k-ix]+jx[k])*n+i];
            }
        else
            for (i=0;i<n;i++)
            {
                f[i*ir+k-ix]-=2*d[(nr[k-ix]+jx[k]-1)*n+i];
                s[i]+=2*beta[nr[k-ix]+jx[k]-1]*d[(nr[k-ix]+jx[k]-1)*n+i];
            }
    }
    jx[k]=m;
    xe=x1[k];   /* boundary value */
}

/*************************************************************************/
/* Calculate the truncation error and determine the new stepsize.        */
/*************************************************************************/

det_h()
{
    int i;
    double err,ratio,trunc=1.0e5,fabs();

    /* choose uniform stepsize for the initial 3 points */
```

```
    if (t<(tstart+4*h))
    {
        new_h=h;
        return(0);
    }

    /* calculate the truncation error */
    for (i=0;i<n;i++)
    {
        err=fabs((x2[i]-xp[i]))/(fabs(x2[i])+1.0e-4);
        if (err<trunc)
            trunc=err;
    }
    ratio=trunc*h/((h1+h)*1E-4);

    /* reduce stepsize by a half if excessive truncation error */
    if (ratio>2)
    {
        h=h/2;
        return(-1);
    }

    /* double the stepsize if tiny truncation error */
    if (ratio<0.5)
        new_h=2*h;
    else
        new_h=h/ratio;

    /* the stepsize should not exceed the printing stepsize */
    if (outfile == 1 && new_h > hy)
        new_h=hy;
    return(0);
}

/*************************************************************************/
/* Predict the solution at next timing point by the previous solutions.    */
/*************************************************************************/

pred_value(xa,xb,xc,ha,hb,hc)
double *xa,*xb,*xc,ha,hb,hc;
{
    int i;
    double at,bt,c1,c2,c3;

    /* find coefficients c1, c2, c3 such that the predicted value is */
    /* exact if the solution trajectory is a 2nd order polynomial    */
    at=1+hb/hc;
    bt=at+ha/hc;
    c3=at/((bt-1)*(bt-at));
    c2=((1-bt)*c3-1)/(at-1);
    c1=1-c2-c3;

    for (i=0;i<n;i++)
        xp[i]=c1*xc[i]+c2*xb[i]+c3*xa[i];
}
```

```
/*************************************************************************/
/* Find cos(theda) to determine the changing rate of the solution        */
/* trajectory, where theda is the angle between two consecutive solution  */
/* vectors  v1=(x1-x), v2=(x2-x1).                                       */
/*************************************************************************/

angle()
{
    int i;
    double r1=0,r2=0,r3=0,r4,sqrt();

    for (i=0;i<n;i++)
    {
        r1+=(x1[i]-x[i])*(x2[i]-x1[i]);
        r2+=(x1[i]-x[i])*(x1[i]-x[i]);
        r3+=(x2[i]-x1[i])*(x2[i]-x1[i]);
    }
    r4=r1/sqrt(r2*r3);
    if (r4<0.8)
        return(-1);
    else return(0);
}
```

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

extern int n,ir,ix,is;
extern char xr[][12],xd[][12],xs[][12];
extern double t,tstart,h,hy,*x1,*x2;
extern FILE *op;
extern int outfile,lx;
char var_name[DIM][12];
double v,ux,vx;
double xmin,xmax,ymin,ymax;
int *di,*scale,pw,xv,yv;
double *sca_x;

/*************************************************************************/
/* Draw graphic axises, labels, coordinate titles, and scaling factor of   */
/* each variable for either phase portrait or waveform plotting.           */
/*************************************************************************/

draw_graf(gp)
GRAF **gp;
{
    int i;
    char ch[2],x_name[30],y_name[30],title[30];
    double pow();
    GRAF *graf_open();

    calloci(n,&di,"di");
    calloci(n,&scale,"scale");
    callocd(n,&sca_x,"sca_x");
    for (i=0;i<n;i++)
    {
        if (i<ix)
            strcpy(var_name[i],xd[i]);
        if (i>=ix && i<ir+ix)
            strcpy(var_name[i],xr[i-ix]);
        if (i>=ir+ix)
            strcpy(var_name[i],xs[i-ir-ix]);
    }

    mgiasngp(0,0);
    mgipln(31);
    *gp=graf_open();

    /* phase portrait or waveform plotting */
    printf("phase portrait? y/n\n");
    scanf("%1s",ch);
    if (ch[0]=='y')
        phase_plot(x_name,y_name);
    else
        waveform(x_name,y_name);

    /* input the graphic parameters */
    read_graf();
```

```
    strcpy(title,"PWL-BDF response");   /* graphic title */
    for (i=0;i<n;i++)                   /* scaling factor */
        sca_x[i]=pow(10.0,(double)scale[i]);

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

    /* if plotting the waveform, draw the scaling factor */
    /* for each variable                                 */
    if (pw==-1)
        draw_x();
}

/*************************************************************************/
/* Determine the variables for phase portrait and draw the graphic axises  */
/* and titles.                                                             */
/*************************************************************************/

phase_plot(x_name,y_name)
char x_name[],y_name[];
{
    char line[20];
    int i;

    pw=1;
    for (i=0;i<n;i++)
        printf("x[%d]=%s\n",i,var_name[i]);
    printf("x_variable=x[?]\n");
    scanf("%d",&xv);                    /* xv : variable index for x axis */
    printf("scaling factor for %s = 1E?\n",var_name[xv]);
    scanf("%d",scale+xv);               /* plot x_value = x*10^scale[xv] */
    printf("y_variable=x[?]\n");
    scanf("%d",&yv);                    /* yv : variable index for y axis */
    printf("scaling factor for %s = 1E?\n",var_name[yv]);
    scanf("%d",scale+yv);               /* plot y_value = y*10^scale[yv] */
    sprintf(line,"%sx1E%d",var_name[xv],scale[xv]);
    strcpy(x_name,line);
    sprintf(line,"%sx1E%d",var_name[yv],scale[yv]);
    strcpy(y_name,line);

    /* print the headings in the output file */
    if (outfile == 1)
        fprintf(op,"**Col(1)=time** **Col(2)=%s** **Col(3)=%s**\n",
                var_name[xv],var_name[yv]);
}

/*************************************************************************/
/* Determine the variables whose transient responses are to be plotted.    */
/*************************************************************************/

waveform(x_name,y_name)
char x_name[],y_name[];
{
    char ch[2];
    int i,j=2;
```

```
    pw = -1;
    strcpy(x_name,"Time");
    strcpy(y_name,"X variables");

    /* di[i] = 1 if the x[i] waveform is to be plotted */
    /* with scaling factor 10^scale[i]                 */
    for (i=0;i<n;i++)
    {
        printf("draw %s ? y/n\n",var_name[i]);
        scanf("%1s",ch);
        if (ch[0]=='y')
        {
            di[i]=1;
            printf("scaling factor for %s = 1E?\n",var_name[i]);
            scanf("%d",scale+i);
        }
    }

    /* print the headings in the output file */
    if (outfile == 1)
    {
        fprintf(op,"**Col(1)=time** ");
        for (i=0;i<n;i++)
            if (di[i] == 1)
                fprintf(op,"**Col(%d)=%s** ",j++,var_name[i]);
        fprintf(op,"\n");
    }
}

/*********************************************************************/
/* Read the graphic ranges for x-axis and y-axis variables.          */
/*********************************************************************/

read_graf()
{
    printf("enter xmin and xmax\n");
    scanf("%lf%lf",&xmin,&xmax);
    printf("enter ymin and ymax\n");
    scanf("%lf%lf",&ymin,&ymax);
}

/*********************************************************************/
/* Draw the graphic color definition for each variable whose transient  */
/* response is to appear in waveform plotting.                       */
/*********************************************************************/

draw_x()
{
    char xi[20];
    int i,j=0;

    for (i=0;i<n;i++)
    {
        if (di[i]==1)
        {
            sprintf(xi,"%sx1E%d",var_name[i],scale[i]);
```

```
            mgihue(i+2);
            mgigfs(80+(j++)*80,500,0,xi);
        }
    }
}

/*************************************************************************/
/* Draw the phase portrait in x-y plane.                                 */
/*************************************************************************/

draw_phase(gp,xt,yt)
GRAF *gp;
double *xt,*yt;
{
    double ratx;

    mgihue(3);
    graf_move(sca_x[xv]*xt[xv],sca_x[yv]*xt[yv],gp);
    graf_draw(sca_x[xv]*yt[xv],sca_x[yv]*yt[yv],gp);
    if (outfile == 1)
        /* the lx-th printing point is reached */
        if ((t+h-tstart-lx*hy)*(t-tstart-lx*hy)<0)
        {
            ratx=(tstart+lx*hy-t)/h;
            fprintf(op,"%.3e\t\t%.3e\t\t%.3e\n",tstart+lx*hy,
                    (1-ratx)*xt[xv]+ratx*yt[xv],
                    (1-ratx)*xt[yv]+ratx*yt[yv]);
            lx++;
        }
}

/*************************************************************************/
/* Draw the transient waveform.                                          */
/*************************************************************************/

draw_waveform(gp,xt,yt)
GRAF *gp;
double *xt,*yt;
{
    int i,j=0;
    double ratx,yout;

    if (outfile == 1)
        /* the lx-th printing point is reached */
        if ((t-tstart-lx*hy)*(t+h-tstart-lx*hy)<0)
        {
            ratx=(tstart+lx*hy-t)/h;
            if (tstart+lx*hy >= 0)
                fprintf(op,"%.3e  ",tstart+lx*hy);
            else
                fprintf(op,"%.3e ",tstart+lx*hy);
            j=1;
            lx++;
        }
    for (i=0;i<n;i++)
    {
```

```
        if (di[i]==1)
        {
            mgihue(i+2);
            graf_move(t,sca_x[i]*xt[i],gp);
            graf_draw(t+h,sca_x[i]*yt[i],gp);
            if (j == 1)
            {
                yout=(1-ratx)*xt[i]+ratx*yt[i];
                if (yout >= 0)
                    fprintf(op,"%.3e  ",yout);
                else
                    fprintf(op,"%.3e ",yout);
            }
        }
    }
    if (j == 1)
        fprintf(op,"\n");
}
```

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

/*************************************************************************/
/* Draw the graphic box, axises, titles, and assign the colors.          */
/*************************************************************************/

setup_graf(xmin,xmax,ymin,ymax,x_name,y_name,title,gp)
double xmin,xmax,ymin,ymax;
char x_name[],y_name[],title[];
GRAF *gp;
{
    if (gp==NULL)
    {
        printf("gp=NULL\n");
        exit();
    }
    define_colors();
    mgihue(1);
    set_screen(80,600,120,520,gp);
    set_real(xmin,xmax,ymin,ymax,gp);
    set_x_axis(N_LBLS,N_TICKS,TICK_LENGTH,SIG_FIGS,LABEL_SIDE,LABEL_SHIFT,
    x_name,NAME_SHIFT,gp);
    set_y_axis(N_LBLS,N_TICKS,TICK_LENGTH,SIG_FIGS,LABEL_SIDE,LABEL_SHIFT,
    y_name,NAME_SHIFT,gp);
    set_title(title,SIZE,OFFSET,gp);
    TI=1;
    draw_bounds(BOX,LABELS,TICKS,AXES,TI,gp);
}

/*************************************************************************/
/* define various types of colors.                                       */
/*************************************************************************/

define_colors()
{
    mgipln(31);
    mgiclearpln(0,-1,0);
    mgicm(1,0xf0f0f0L);
    mgicm(2,0x00f0a0L);
    mgicm(3,0xf0f008L);
    mgicm(4,0xf008f0L);
    mgicm(5,0xf00000L);
    mgicm(6,0xa0b005L);
    mgicm(7,0x00f000L);
    mgicm(8,0x0000f0L);
}
```