NONLINEAR ELECTRONICS (NOEL) PACKAGE 4:
NONLINEAR TRANSIENT ANALYSIS

by

An-Chang Deng and Leon O. Chua

Memorandum No. UCB/ERL M86/27

28 March 1986

NONLINEAR ELECTRONICS (NOEL) PACKAGE 4:

NONLINEAR TRANSIENT ANALYSIS

by

An-Chang Deng and Leon O. Chua

Memorandum No. UCB/ERL M86/27

28 March 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

NOEL PACKAGE 4 : NONLINEAR 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 performs the transient analysis of a nonlinear circuit. A preprocessor of circuit formulation routine is called to translate the circuit into a C source code, which describes the circuit equation $\mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}, t) = 0$ and the corresponding Jacobian matrix $\mathbf{J}_f = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}$. The BDF (Backward Differentiation Formula) algorithm is then applied to solve the circuit equation.

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 4 : NONLINEAR TRANSIENT ANALYSIS

1. Introduction

A general dynamic network can be represented by a set of algebraic-differential equation (or an implicit differential equation)

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

by the algorithms and routines in [1], which gives the circuit equation and the corresponding Jacobian matrix in the form of a C source code. Compiling this equation routine and linking it with the integration routines, we can find the solution of Eq.(1) in terms of the transient waveform or the phase portrait of any two variables in the circuit.

There are various integration routines for solving the ordinary differential equations (Adams-Moulton, Adams-Bashforth, and Runge-Kutta)[2] which, however, require that the differential equation be expressed in explicit form; namely,

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}\,,\,t\,) \tag{2}$$

The explicit state equation (2) is a special case for the implicit equation (1) and it takes some extra effort to reduce it to the explicit form. Moreover, Eq.(2) may not exist for general dynamic circuits[2]. Henceforth, we only consider the implicit state equation (1) and apply the BDF (Backward Differentiation Formula) algorithm, which is an implicit integration routine, to solve it.

However, the transient analysis using the BDF routine requires a starting point $\mathbf{x}_0$ such that $\mathbf{f}(\mathbf{x}_0\,,\,\dot{\mathbf{x}}_0\,,\,t_0) = 0$, where t_0 is the starting time. This starting point $\mathbf{x}_0$ includes the initial voltages or currents of the nonlinear resistors in addition to the initial condition of dynamic elements. Therefore it is required to perform a dc analysis to find the dc operating point at the starting time (including the controlling variables listed in Section 4 of [3], and the initial condition of dynamic elements which is given by the user).

The equation for the modified circuit at starting time (with each capacitor (resp.; inductor) replaced by an independent voltage source (resp.; independent current source), and the source value is equal to the initial condition given by the user) is described by the initial equation routine

ini_eq(n, iz, f, x, z, x_dot, t)

and the corresponding Jacobian matrix routine

ini_jacob(x, jf, t)

Applying the Newton-Raphson algorithm of [3] to the initial equation routine, we can perform a dc analysis to find the dc operating point at starting time, and use it as the starting point for BDF transient analysis.

2. Algorithm

Step 1.

Find the circuit equation and the corresponding Jacobian matrix for the dynamic circuit to be analyzed.

Step 2.

Find the circuit equation and the corresponding Jacobian matrix for the modified circuit at starting time.

Step 3.

Read the parameters for BDF transient analysis :

(1) starting time t_0

(2) final time t_f

(3) order of BDF formula k

(4) initial stepsize h

(5) type of output : waveform or phase portrait

(6) graphical parameters : xmin, xmax, ymin, ymax

Step 4.

Perform the dc analysis to find the dc operating point $\mathbf{x}_0$ for the modified circuit at starting time t_0.

Step 5.

Use the m-th order formula (see the k-th order formula in Step 6) to find the solution $\mathbf{x}_m$ at $t_m = t_0 + m*h$ for m=1,2,...,k.

Step 6.

Given the solutions $\mathbf{x}_0, \mathbf{x}_1, \ldots \mathbf{x}_k$ at k+1 timing points $t_0, t_1, \ldots, t_k$ respectively, apply the k-th order formula to find the solution $\mathbf{x}_{k+1}$ at the next timing point t_{k+1} until $t_{k+1} > t_f$:

(a) Find the coefficients $\alpha_0, \alpha_1, \ldots, \alpha_k$ such that the approximation

$$\dot{\mathbf{x}}_{k+1} = \frac{-1}{h} \sum_{i=0}^{k} \alpha_i \mathbf{x}_{k+1-i} \tag{3}$$

of $\dot{\mathbf{x}}$ at $t = t_{k+1} = t_k + h$ is exact if the solution trajectory is a k-th order polynomial.

(b) Find the coefficients $\gamma_1, \gamma_2, \ldots, \gamma_k, \gamma_{k+1}$ such that the approximation

$$\mathbf{x}_{k+1}{}^P = \sum_{i=1}^{k+1} \gamma_i \mathbf{x}_{k+1-i} \tag{4}$$

of the solution $\mathbf{x}_{k+1}$ at t_{k+1} is exact if the solution trajectory is a k-th order polynomial.

(c) Apply Newton-Raphson iteration to find the solution $\mathbf{x}_{k+1}$ of the nonlinear algebraic equation

$$\mathbf{f}(\mathbf{x}_{k+1}, \dot{\mathbf{x}}_{k+1}, t_{k+1}) = \mathbf{f}(\mathbf{x}_{k+1}, \frac{-1}{h} \sum_{i=0}^{k} \alpha_i \mathbf{x}_{k+1-i}, t_{k+1}) = 0 \tag{5}$$

with $\mathbf{x}_{k+1}{}^P$ as the initial guess for the iteration.

(d) If the iteration does not converge, go to (h).

(e) Compute the ratio for local truncation error

$$ratio = \frac{E_k}{10^{-2}}$$

where

$$E_k = \max\{E_{ki} \mid i=1,2,\ldots n\}$$

$$E_{ki} = \frac{h}{t_{k+1} - t_0} \times \frac{|x_{k+1,i} - x_{k+1,i}{}^P|}{|x_{k+1,i}| + 10^{-3}}$$

and $x_{k+1,i}$ (resp.; $x_{k+1,i}{}^P$) is the i-th component of $\mathbf{x}_{k+1}$ (resp.; $\mathbf{x}_{k+1}{}^P$).

(f) If *ratio* > 2 then reject the computed solution and go to (h) to reduce the stepsize; else accept the computed solution $\mathbf{x}_{k+1}$ as the solution at $t = t_{k+1}$ and adjust the stepsize for next timing point to

$$h = 2h \quad \text{if } ratio < 0.5$$

$$h = \frac{h}{ratio} \quad \text{if } 0.5 < ratio < 2.$$

(g) Renew the set of solutions and timing points such that it contains k+1 solutions at the most recent k+1 timing points; namely

$$t_0 = t_1,\ t_1 = t_2,\ \ldots\ldots\ t_k = t_{k+1}$$

$$\mathbf{x}_0 = \mathbf{x}_1,\ \mathbf{x}_1 = \mathbf{x}_2,\ \ldots\ldots\ \mathbf{x}_k = \mathbf{x}_{k+1}$$

and choose $t_{k+1} = t_k + h$. Go to (a).

(h) Reduce the stepsize by a half, h=h/2, and choose $t_{k+1} = t_k + h$; go to (a).

3. User's Instruction

Step 1.

Create a file "xx...x.spc" which describes the dynamic circuit to be analyzed and follows the rules of the input format language defined in [4] for each class of circuit elements, where "xx...x" is the filename for the input file with extension ".spc". All the circuit elements defined in [4] can be included in "xx...x.spc" except those described by pwl characteristics with numerical expression; namely

'R' : 2-terminal resistor (linear or general nonlinear char.)

'C' : 2-terminal capacitor (linear or general nonlinear char.)

'L' : 2-terminal inductor (linear or general nonlinear char.)

'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

'K' : nonlinear controlled source (at most 2 controlling variables)

'N' : 2-port or 3-terminal resistor (linear or general nonlinear char.)

For elements with pwl characteristics to appear in "xx...x.spc", they should be described by the absolute function fabs().

Steps 2-5 are combined as a batch process and are executed by typing the command

bdfsim xx...x

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

Step 2.

Type the command

form xx...x

to produce the C source code "xx...x.c" for the circuit equation and the corresponding Jacobian matrix.

Step 3.

Type the command

formi xx...x

to append the C source code for the circuit equation and the corresponding Jacobian matrix of the initial circuit to the file "xx...x.c".

Step 4.

Compile the file "xx...x.c" to get the object code and link it with the BDF simulation routine.

Step 5.

Type the command

bdf xx...x

to perform the transient analysis which proceeds interactively with user in the following way :

(a) **enter the initial time**

Enter the starting time t_0 for BDF transient analysis.

(b) **enter the initial condition for C??..? (or L??..?)**

Enter the initial condition (capacitor voltage or inductor current) for each dynamic element. If the circuit contains no nonlinear resistor and time-varying source, then the initial condition can be used as the starting point without running the dc analysis and Steps (c)-(g) are skipped.

(c) **start dc analysis for starting point .....**

Perform a dc analysis to find the starting point for BDF transient analysis.

(d) **default initial guess? y/n**

Type 'y' for default initial guess (equal to zero) in Newton-Raphson iteration.

(e) If 'n' in (d), the user has to provide the initial guess for each controlling variable listed in Section 4 of [3], the capacitor current and the inductor voltage; e.g.,

v(R1) =
i(R2) =
i(Cx) =
v(Ly) =
.....

(f) If the iteration converges, the computed starting point will be printed in terms of the variables in (e) except the capacitor current (resp.; inductor voltage) is replaced by the initial capacitor voltage (resp.; inductor current) given in (c) :

v(R1) =
i(R2) =
v(Cx) =
i(Ly) =
.....

would you like to try another starting point? y/n

If 'y' then go to (d) to repeat the dc analysis;

else stop the dc analysis and go to (h) by using the computed solution as the starting point.

Type 'y' if the circuit possesses multiple dc operating points at starting time and the user likes to start from the other point.

(g) If the iteration does not converge, the solution at last iteration is printed in terms of the variables in (e); it then asks the user whether to continue the dc analysis for finding the starting point :

would you like to continue? y/n

Type 'y' and go to (d) to repeat the dc analysis with a new initial guess.

Type 'n' if the user decides to abort the transient analysis since no starting point is found.

(h) **enter the BDF order**

Enter any integer k between 1 and 6 for k-th order BDF formula. (recommend to use the 3rd or 4th order)

(i) **enter the final time**

Enter the final time t_f to terminate the transient analysis, which should be greater than the starting time t_0.

(j) **enter the initial stepsize**

Enter the initial stepsize used in starting stage, which should be small enough not to cause large truncation error, since only the 1st, 2nd, ..., and (k-1)-th order of BDF formula are available to find the solutions at the 1st, 2nd, ..., and (k-1)-th timing point after the starting time t_0 respectively.

(k) **phase portrait? y/n**

If 'y', then go to (l) for phase portrait;

else go to (m) for plotting the transient waveforms.

(l) Choose the indices for x and y variables from the table of the variable definitions :

x[0] = v(R1)

x[1] = i(R2)

x[2] = v(Cx)

x[3] = i(Ly)

.....

x_variable = x[?]

Enter the index for x variable; e.g., 2 for v(Cx)

scaling factor for v(Cx) = 1E?

Enter the scaling factor for v(Cx); e.g., 3 (resp.; -2) if 1E3×v(Cx) (resp.; 1E-2×v(Cx)) is to be plotted as the x variable in the phase portrait.

y_variable = x[?]

scaling factor for ??...? = 1E?

Similar to the x variable.

(m) Determine the variables whose transient waveforms are to be plotted in the color monitor; e.g.,

draw v(R1)? y/n

If 'y', then enter the scaling factor for this variable :

scaling factor for v(R1) = 1E?

Enter 3 (resp.; -2) if the transient waveform of 1E3×v(R1) (resp.; 1E-2×v(R1)) is to be plotted.

(n) **enter xmin and xmax**

Enter the lower bound xmin and the upper bound xmax for the x coordinate in the color screen.

(o) **enter ymin and ymax**

Enter the lower bound ymin and the upper bound ymax for the y coordinate in the color screen.

4. Output Format

The BDF simulation results are shown in the color monitor in two different modes of operations :

(1) transient waveforms 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) each controlling variable listed in Section 4 of [3]

(ii) voltage of linear capacitor

(iii) voltage of voltage-controlled nonlinear capacitor

(iv) voltage and charge of charge-controlled or implicit-relation nonlinear capacitor

(v) current of linear inductor

(vi) current of current-controlled nonlinear inductor

(vii) current and flux of flux-controlled or implicit-relation nonlinear inductor

(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 graphic box.

In addition to the graphical output, the numerical results of the transient response are written to the output file "xx...x.out", provided the -o option is specified in Step 5; namely,

bdf -o xx...x

or

bdfsim -o xx...x

for the batch command. In this case, the user has to enter the printing stepsize $ h sub p $ such that each data point at $ t ^=^ t sub 0 ^+^ l*h sub p $ is printed in the output file "xx...x.out" for $ l $=1,2,...

5. Examples

Example 1 : in file "ex1.spc"

A dynamic circuit with the piecewise-linear resistor (Fig.1).

Example 2 : in file "ex2.spc"

A single transistor amplifier (Fig.2).

Example 3 : in file "ex3.spc"

The double scroll chaotic circuit (Fig.3).

6. Diagnosis

1. BDF TABLE_FILE

   Bad command line, the correct one should be

   bdf xx...x

   where "xx...x.tbl" is the table file for the input file "xx...x.spc".

2. CAN'T OPEN THE TABLE FILE xx...x.tbl

   The table file "xx...x.tbl" does not exist in the current directory.

3. SINGULAR JACOBIAN MATRIX FOR DC ANALYSIS

   The Jacobian matrix is singular or some of the matrix entries overflow numerically at some iteration point of Newton-Raphson iteration for finding the starting point of BDF transient analysis; should try another initial guess for Newton-Raphson iteration.

4. WARNING MESSAGE : JACOBIAN MATRIX IS SINGULAR

   The Jacobian matrix is singular at some timing point of transient analysis; the program automatically chooses a new initial guess for Newton-Raphson iteration to avoid the singularity.

5. WARNING MESSAGE : NONCONVERGENT ITERATION

   The Newton-Raphson iteration does not converge (iteration number > 5) due to sharp turning point; the stepsize will be reduced until a convergent solution is achieved.

References

[1] A.C.Deng and L.O.Chua, "NOnlinear ELectronics utility programs"

[2] L.O.Chua and P.M.Lin, *Computer Aided Analysis of Electronic Circuits : Algorithms and Computational Techniques* , Englewood Cliffs, NJ : Prentice-Hall, 1975.

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

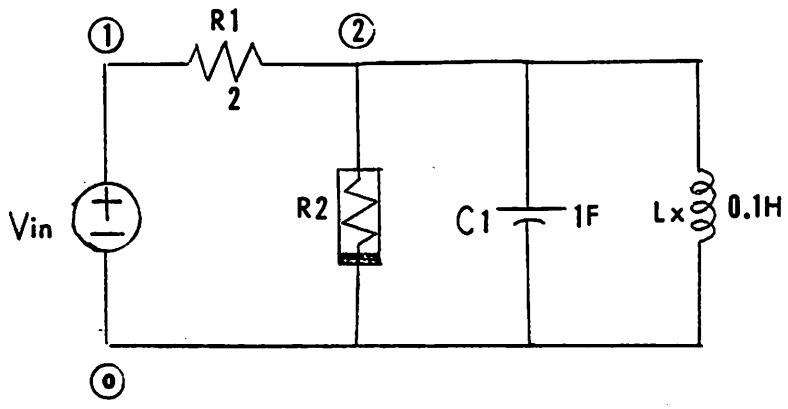
[4] A.C.Deng and L.O.Chua, "NOnlinear ELectronics package 0 : general description," ERL Memo., M86, University of California, Berkeley, 1986.

Figure Captions

Fig.1 A dynamic circuit with the piecewise-linear resistor.

Fig.2 A single transistor amplifier.

Fig.3 The double scroll chaotic circuit.



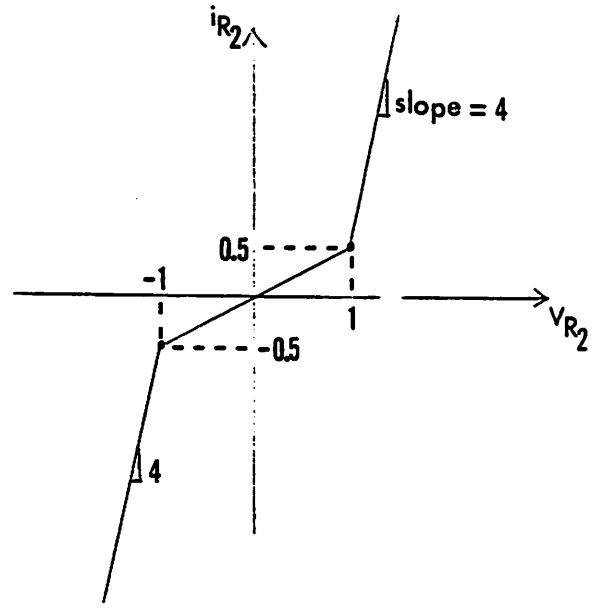


Fig.1

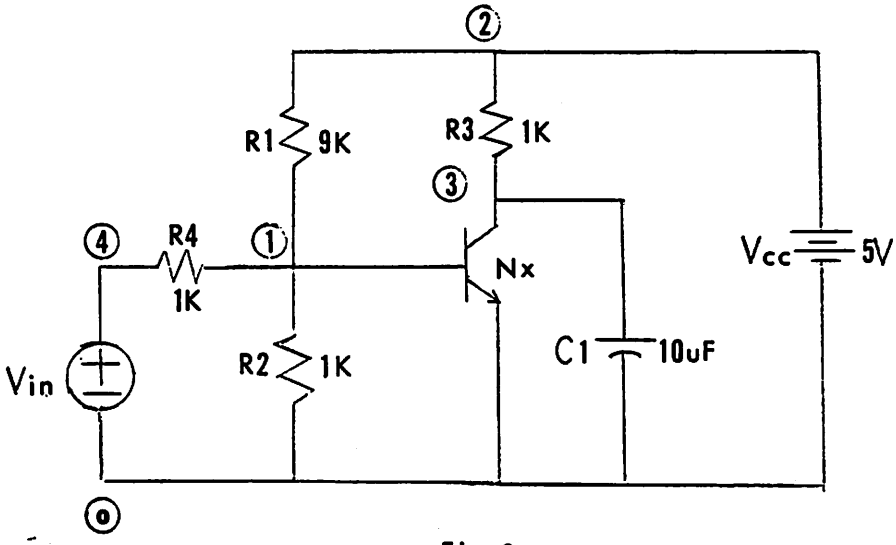


Fig.2

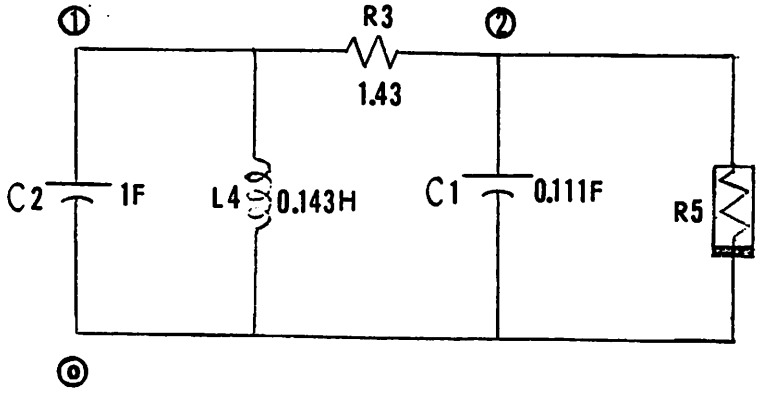
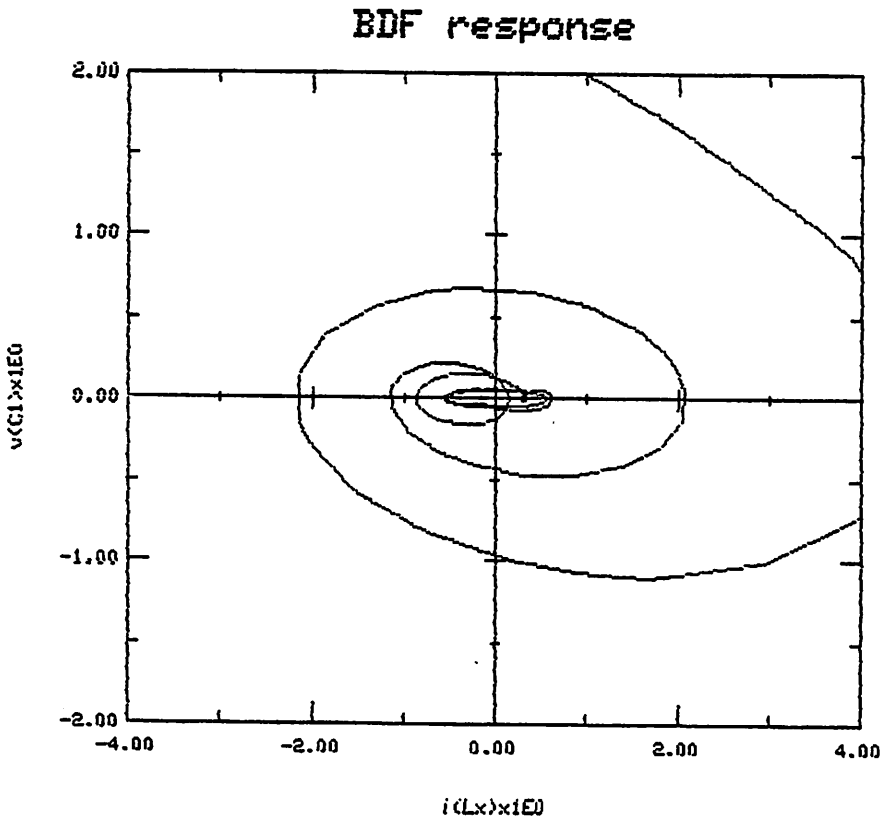


Fig.3

```
* Example 1
*
* Dynamic circuit with piecewise-linear resistor
*
* a nonlinear resistor characterized by pwl model
R2 2 0 {i=4*v-1.75*fabs(v+1)+1.75*fabs(v-1)}
*
C1 2 0 1
Lx 2 0 0.1
R1 1 2 2
*
* sinusoidal input source
Vin 1 0 {sin(t)}
*
* include the file with mathematical declarations
.include "math.h"
.end
```

```
bdf ex1
enter the initial time
0
enter the initial condition for Lx
1
enter the initial condition for C1
2
start dc analysis for starting point .....

default initial guess? y/n
y

starting point .....
v(R2)=2.000E+000
i(Lx)=1.000E+000
v(C1)=2.000E+000
i(Vin)=1.000E+000

would you like to try another starting point? y/n
n

end of dc analysis

enter the BDF order
3
enter the final time
20
enter the initial stepsize
1e-4
enter the starting time to draw the graphics
0
phase portrait? y/n
y
x[0]=v(R2)
x[1]=i(Lx)
x[2]=v(C1)
x[3]=i(Vin)
x_variable=x[?]
1
scaling factor for i(Lx) = 1E?
0
y_variable=x[?]
2
scaling factor for v(C1) = 1E?
0
enter xmin and xmax
-4 4
enter ymin and ymax
-2
2
```

C>
φ

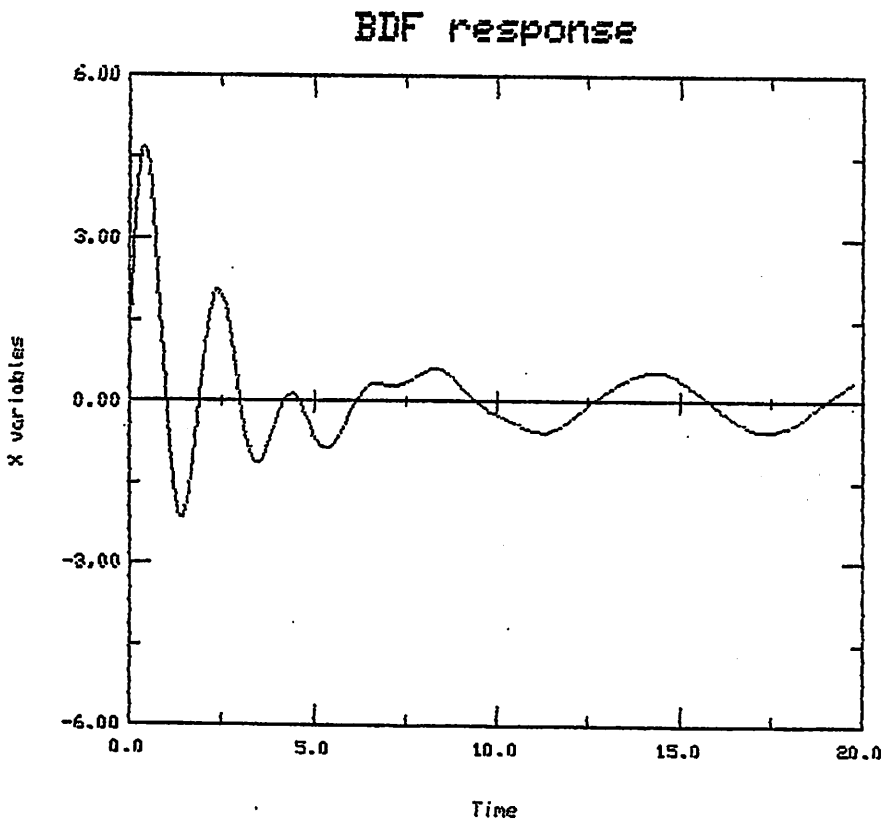

BDF response
2.00
1.00
0.00
-1.00
-2.00
v(C1)x1E0
-4.00
-2.00
0.00
2.00
4.00
i(Lx)x1E0

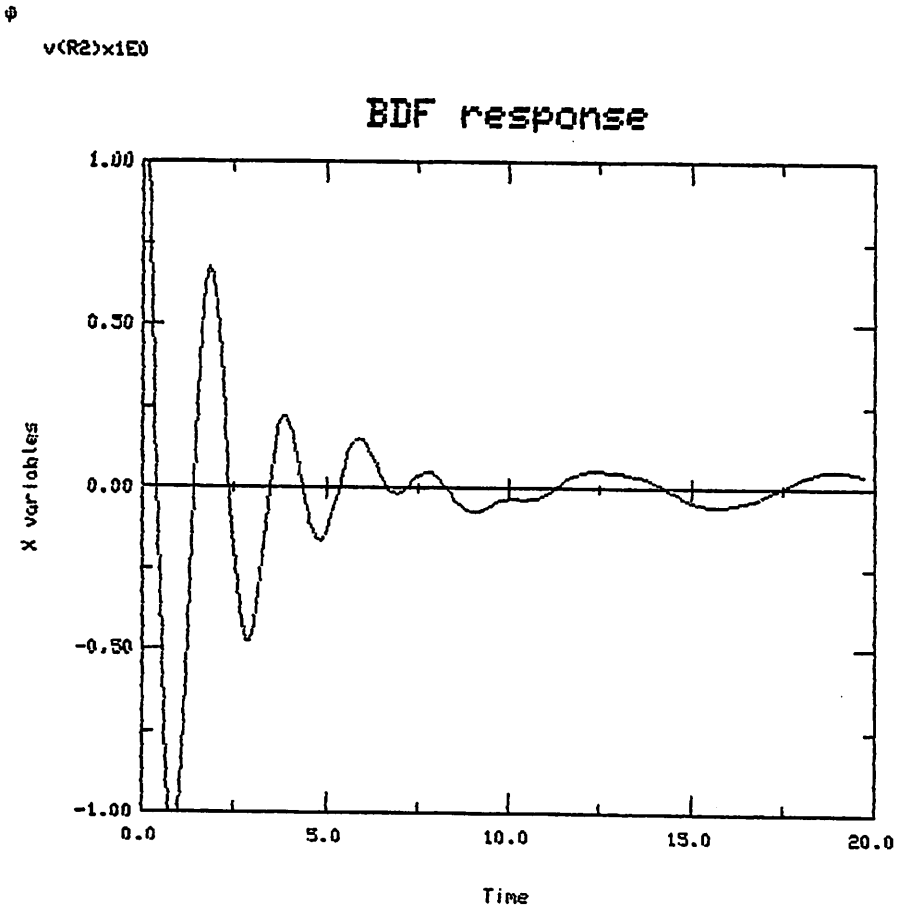
```
bdf ex1
enter the initial time
0
enter the initial condition for Lx
1
enter the initial condition for C1
2
start dc analysis for starting point .....

default initial guess? y/n
y

starting point .....
v(R2)=2.000E+000
i(Lx)=1.000E+000
v(C1)=2.000E+000
i(Vin)=1.000E+000

would you like to try another starting point? y/n
n

end of dc analysis

enter the BDF order
3
enter the final time
20
enter the initial stepsize
1e-4
enter the starting time to draw the graphics
0
phase portrait? y/n
n
draw v(R2) ? y/n
n
scaling factor for v(R2) = 1E?
0
draw i(Lx) ? y/n
y
scaling factor for i(Lx) = 1E?
0
draw v(C1) ? y/n
n
draw i(Vin) ? y/n
y
scaling factor for i(Vin) = 1E?
0
enter xmin and xmax
0 20
enter ymin and ymax
-6 6
```

C>

φ

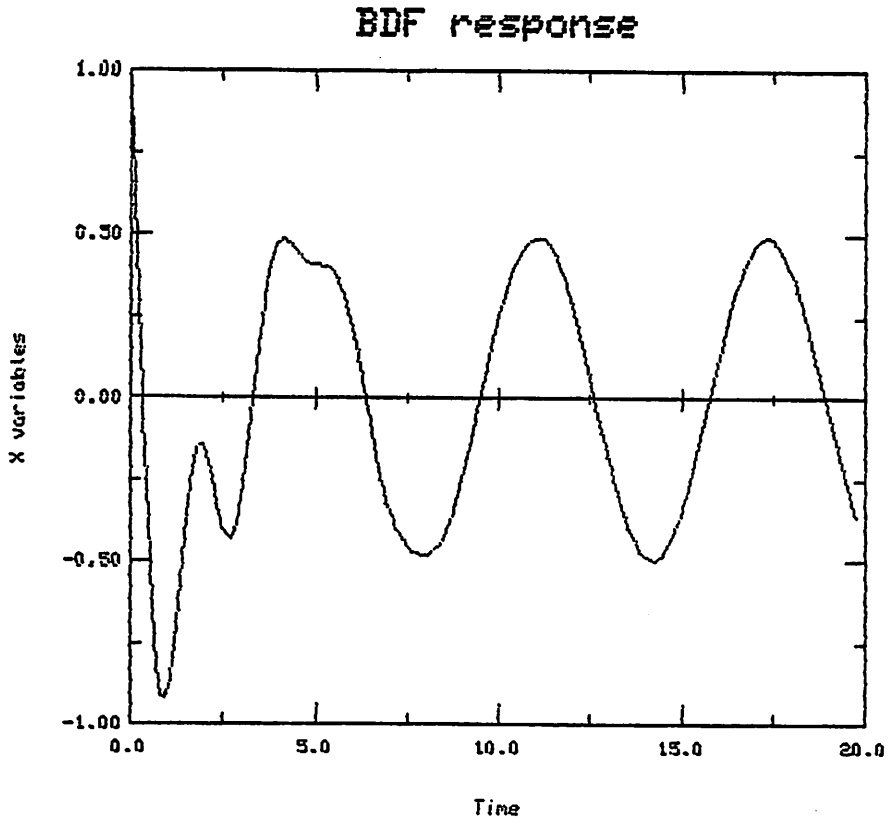
I(Lx)x1E0


BDF response
6.00
3.00
0.00
-3.00
-6.00
X variables
0.0
5.0
10.0
15.0
20.0
Time


v(R2)x1E0
BDF response
1.00
0.50
0.00
-0.50
-1.00
X variables
0.0
5.0
10.0
15.0
20.0
Time

φ

I(Vin)x1E0


BDF response
1.00
0.50
0.00
-0.50
-1.00
X variables
0.0
5.0
10.0
15.0
20.0
Time

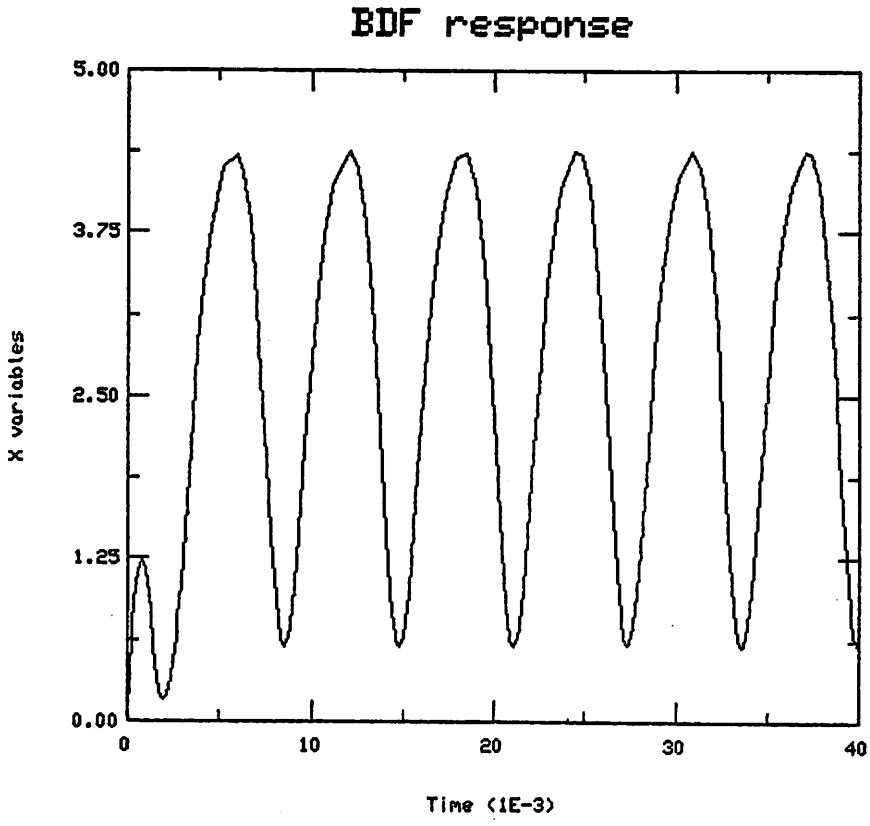
```
* Example 2
*
* single transistor amplifier
Nx 1 0 1 3 bpmod
C1 3 0 1u
R4 1 4 1K
R1 1 2 8K
R2 1 0 1K
R3 2 3 1K
Vcc 2 0 5
*
* sinusoidal input
Vin 4 0 {0.8+0.1*sin(1000*t)}
*
* bipolar transistor model
.model bpmod {i1=1.005e-14*exp(38.46*v1)-1.0e-14*exp(38.46*v2);
$i2=2.0e-14*exp(38.46*v2)-1.0e-14*exp(38.46*v1)}
* include "math.h"
.include "math.h"
.end
```

```
bdf ex2
enter the initial time
0
enter the initial condition for C1
0
start dc analysis for starting point .....

default initial guess? y/n
y

starting point .....
v1(Nx)=6.066E-001
v2(Nx)=6.066E-001
v(C1)=0.000E+000
i(Vin)=-1.934E-004

would you like to try another starting point? y/n
n

end of dc analysis

enter the BDF order
3
enter the final time
0.04
enter the initial stepsize
1e-5
enter the starting time to draw the graphics
0
phase portrait? y/n
n
draw v1(Nx) ? y/n
n
draw v2(Nx) ? y/n
n
draw v(C1) ? y/n
y
scaling factor for v(C1) = 1E?
0
draw i(Vin) ? y/n
n
enter xmin and xmax
0 0.04
enter ymin and ymax
0 5
```

C>

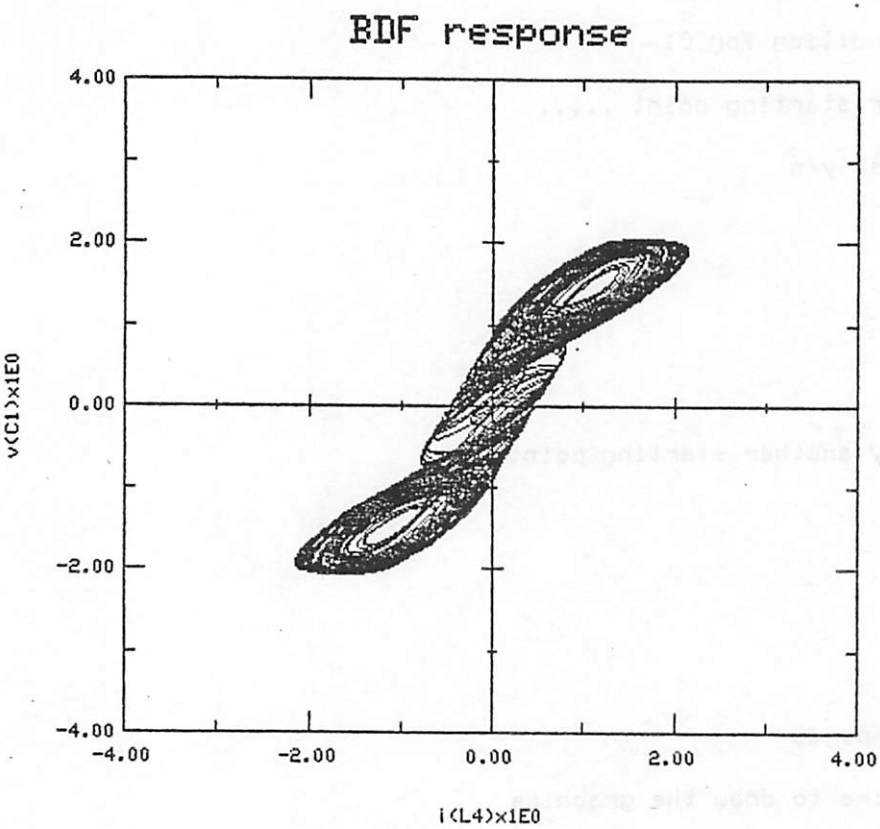
Φ

v(C1)x1E0


BDF response
5.00
3.75
2.50
1.25
0.00
X variables
0
10
20
30
40
Time (1E-3)

```
* 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=1.3*v-0.9*fabs(v+2)-0.15*fabs(v+1)
$+0.15*fabs(v-1)+0.9*fabs(v-2)}
.include "math.h"
.end
```

```
  bdf ex3
enter the initial time
0
enter the initial condition for C2
0.2
enter the initial condition for L4
0.1
enter the initial condition for C1
0.3
start dc analysis for starting point .....

default initial guess? y/n
y

starting point .....
v(C2)=2.000E-001
i(L4)=1.000E-001
v(C1)=3.000E-001
v(R5)=3.000E-001

would you like to try another starting point? y/n
n

end of dc analysis

enter the BDF order
3
enter the final time
400
enter the initial stepsize
1e-4
enter the starting time to draw the graphics
0
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
```


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

```c
#include <stdio.h>

main(argc,argv)
int argc;
char *argv[];
{
    bdf(argc,argv);
}

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

double ipow(x,i)
int i;
double x;
{
    int j;
    double z=1.0;

    if (i >= 0)
        for (j=1;j<=i;j++)
            z=z*x;
    return(z);
}
```

```
#include <stdio.h>

int n,k;
double h,*x,*t,*c,*z,*y,ft;
double *r,*s,*rx,*sx,*d,*e;
extern double u,v,ux,vx;

/*************************************************************************/
/* Apply the BDF algorithm to perform the transient analysis of a dynamic  */
/* circuit.                                                                */
/*************************************************************************/

bdf(argc,argv)
int argc;
char *argv[];
{
    char *calloc(),line[20];
    FILE *fopen();

    /* find the starting point */
    ini_pt(argc,argv);

    /* read the bdf parameters */
    read_bdf();

    /* allocate spaces for arrays of variables */
    pre_alloc();

    /* find solutions of the first k+1 points */
    start();

    /* follow bdf formula to find the transient response */
    bdf_comp();
}

/*************************************************************************/
/* Read the BDF parameters : order, final time, and the initial stepsize.  */
/*************************************************************************/

read_bdf()
{
    printf("enter the BDF order\n");            /* read k */
    scanf("%d",&k);
    rallocd(&x,n*(k+2),n);
    printf("enter the final time\n");           /* read ft */
    scanf("%lf",&ft);
    printf("enter the initial stepsize\n");     /* read h */
    scanf("%lf",&h);
}

/*************************************************************************/
/* Allocate spaces for array of variables after knowing the dimension n    */
/* and the order k.                                                        */
/*************************************************************************/

pre_alloc()
```

```
{
    callocd(n,&c,"c");
    callocd(n,&z,"z");
    callocd(n,&y,"y");
    callocd(k+2,&r,"r");
    callocd(k+2,&rx,"rx");
    callocd(k+2,&s,"s");
    callocd(k+2,&sx,"sx");
    callocd((k+2)*(k+2),&d,"d");
    callocd(k+2,&e,"e");
}

/*************************************************************************/
/* Evaluate the initial guess z and parameters c and ha such that the    */
/* derivative x_dot is approximated by    x_dot = ha*z + c               */
/*************************************************************************/

appx_d_ig(ha)
double *ha;
{
    int i,m;

    for (i=0;i<n;i++)
    {
        c[i]=0;
        for (m=1;m<=k;m++)
            c[i]=c[i]+s[m]*x[i+(k+1-m)*n];
        c[i] = -c[i]/h;
        z[i]=0;
        for (m=1;m<=k+1;m++)
            z[i]=z[i]+r[m]*x[i+(k+1-m)*n];
        y[i]=z[i];
    }
    *ha = -s[0]/h;
}

/*************************************************************************/
/* Evaluate the local truncation error.                                  */
/*************************************************************************/

trunca_err(ratio)
double *ratio;
{
    int i;
    double trunca=0,fabs(),*xx;

    callocd(n,&xx,"xx");
    for (i=0;i<n;i++)
    {
        xx[i]=fabs(z[i]-y[i])/(fabs(z[i])+1E-3);
        if (trunca<xx[i])
            trunca=xx[i];
    }
    trunca=trunca*h/(t[k+1]-t[0]);
    *ratio=trunca/1e-2;
    cfree(xx);
```

```
}

/*************************************************************************/
/* Reduce the stepsize h in case of excessive truncation error.          */
/*************************************************************************/

/* the ij-th entry of the matrix d is expressed as d[i*(k+2)+j] */

reduce_h(cg,ratio)
int cg;
double ratio;
{
    int i,j;

    /* restore to the original set of parameters r,s,u and v in */
    /* order to evaluate the new x_dot and initial guess when a */
    /* smaller stepsize is chosen                               */
    for (i=1;i<k+2;i++)
    {
        r[i]=rx[i];
        s[i-1]=sx[i-1];
    }
    u=ux;
    v=vx;

    if (cg==-1)          /* reduce to a quarter h when nonconvergent */
        h=h/4;           /* Newton-Raphson iteration                 */

    else
        if (ratio>2)     /* reduce to a half h if ratio>2 and h/ratio */
            h=h/2;       /* if 1<ratio<h                              */
        else
            h=h/ratio;

    /* renew d for a new stepsize h */
    d[1]=h;
    for (j=2;j<=k+1;j++)
        d[j]=d[1]+e[j-1];
    for (j=1;j<=k+1;j++)
        d[j*(k+2)] = -d[j];
}

/*************************************************************************/
/* Renew the BDF formula for computing the solution of the next point.   */
/*************************************************************************/

/* the ij-th entry of the matrix d is expressed as d[i*(k+2)+j] */

next_pt()
{
    int i,j,m;

    /* save the previous r,s,u,v,and e:first of d, in case x_dot and */
    /* the initial guess of x have to be re-evaluated for a new      */
    /* reduced stepsize                                              */
    for (m=1;m<=k+1;m++)
```

```
    {
        rx[m]=r[m];
        sx[m-1]=s[m-1];
    }
    ux=u;
    vx=v;

    /* save e and renew d */
    for (j=k+1;j>0;j--)
    {
        e[j]=d[j];
        for (i=k+1;i>0;i--)
            d[i*(k+2)+j]=d[(i-1)*(k+2)+j-1];
    }
    d[1]=h;
    for (j=2;j<=k+1;j++)
        d[j]=d[1]+e[j-1];
    for (j=1;j<=k+1;j++)
        d[j*(k+2)] = -d[j];
}
```

```
#include <stdio.h>

extern int n,k;
extern double *t,*x;
extern double *c,*z,u,v,ux,vx,*r,*s,*rx,*sx,*d;

/*************************************************************************/
/* Newton-Raphson iteration for solving the nonlinear algebraic equation  */
/*                        f(x) = 0                                        */
/*************************************************************************/

nr(ini,ha,tx)
int ini;
double ha,tx;
{
    int i,l=0,lx,*ipvt;
    double max=1,rcond,*f,*x_dot,*dx_dot,*zq,*jf;
    double fabs();

    /* allocate spaces for the variables used in Newton iterations */
    nr_alloc(&ipvt,&f,&zq,&x_dot,&dx_dot,&jf);

    /* iterate 20 times for the starting (k+1) points */
    /* and 5 times for other points                   */
    if (ini==1)
        lx=20;
    else
        lx=5;

    /* follow the iteration formula                                  */
    /*          x_k+1 = x_k - inv(jf)*f(x_k)                         */
    /* to find the solution of f(x)=0; iteration converges if        */
    /*   |x_k+1 - x_k| < 1.0e-5  and k < lx                          */
    while (l++<lx && max>1.0e-5)
    {
        /* approximate x_dot and dx_dot by the solutions at */
        /* the previous k timing points and the projected   */
        /* solution at present time                         */
        approx_d(x_dot,dx_dot,ha);

        /* evaluate f(x_k) */
        equation(f,z,x_dot,tx);

        /* evaluate jf(x_k) */
        jacob(z,x_dot,dx_dot,jf,tx);

        /* LU decomposition of jf(x_k) */
        sgeco(jf,n,ipvt,&rcond,zq);

        /* singular Jacobian matrix : change the initial guess    */
        /* for starting period; otherwise return non_convergenvce */
        /* to reduce the stepsize                                 */
        if (fabs(rcond)<1.0e-20)
        {
            if (ini==0)
                return(-1);
```

```
            singular(ini,ha,tx);
            break;
        }

        else
        {
            /* find inv(jf)*f */
            sgesl(jf,n,ipvt,f,0);

            /* find x_k+1 = x_k - inv(jf)*f and evaluate    */
            /*          max = max | x_k+1[i] - x_k[i] |     */
            max=0.0;
            for (i=0;i<n;i++)
            {
                if (max<fabs(f[i])) max=fabs(f[i]);
                z[i]-=f[i];
            }
        }
    }

    /* free the spaces occupied by the variables in this routine */
    nr_free(ipvt,f,zq,x_dot,dx_dot,jf);

    if (l>=lx && max>1.0e-5)
        return(-1);
    else
        return(0);
}

/*************************************************************************/
/* Allocate spaces for the variables used in Newton-Raphson iteration      */
/* routine.                                                                 */
/*************************************************************************/

nr_alloc(ipvt,f,zq,x_dot,dx_dot,jf)
int **ipvt;
double **f,**zq,**x_dot,**dx_dot,**jf;
{
    calloci(n,ipvt,"ipvt");
    callocd(n,zq,"zq");
    callocd(n,f,"f");
    callocd(n,x_dot,"x_dot");
    callocd(n,dx_dot,"dx_dot");
    callocd(n*n,jf,"jf");
}

/*************************************************************************/
/* Free the spaces occupied by the variables in Newton-Raphson iteration   */
/* routine, which no longer required after exit from that routine.         */
/*************************************************************************/

nr_free(ipvt,f,zq,x_dot,dx_dot,jf)
int *ipvt;
double *f,*zq,*x_dot,*dx_dot,*jf;
{
    cfree(ipvt);
```

```
    cfree(f);
    cfree(zq);
    cfree(x_dot);
    cfree(dx_dot);
    cfree(jf);
}

/*************************************************************************/
/* Eavluate the approximation of x_dot and dx_dot.                       */
/*************************************************************************/

approx_d(x_dot,dx_dot,ha)
double x_dot[],dx_dot[],ha;
{
    int i;

    for (i=0;i<n;i++)
    {
        x_dot[i]=ha*z[i]+c[i];
        dx_dot[i]=ha;
    }
}

/*************************************************************************/
/* Choose a new initial guess in case of singular Jacobian matrix.       */
/*************************************************************************/

singular(ini,ha,tx)
int ini;
double ha,tx;
{
    int i;

    printf("WARNING MESSAGE : JACOBIAN MATRIX IS SINGULAR\n");
    printf("....TRYING ANOTHER INITIAL GUESS....\n");
    for (i=0;i<n;i++)
        z[i]=z[i]+1/ha;
    nr(ini,ha,tx);
}

/*************************************************************************/
/* Initially evaluate the coefficients s_i (resp.; r[]) for the          */
/* approximation of                                                      */
/*              x_dot = sum (s_i*x_(n+1-i))  i=0 to k                    */
/* (resp.;        x^p = sum (r_i*x_(n+1-i))  i=1 to k+1   )              */
/* where x^p is the initial guess for x.                                 */
/*************************************************************************/

ini_form()

/* the ij-th entry of the matrix d is expressed as   d[i*(k+2)+j]   */
{
    int i,j,m;

/* all of the following calculations follow the formulas in p.107 */
/* of the reference paper, where                                  */
```

```
/*              d[i*(k+2)+j] <==> A_ij                          */
/*                       r_j <==> r_j(n,k)                      */
/*                       s_j <==> alpha_j(n,k)                  */
/*                         u <==> F(n,k)                        */
/*                         v <==> delta(n,k)                    */

    for (i=k+1;i>=0;i--)
        for (j=k+1;j>=0;j--)
            d[i*(k+2)+j]=t[k+1-i]-t[k+1-j];
    for (j=1;j<=k+1;j++)
    {
        r[j]=1;
        for (m=1;m<=k+1;m++)
            if (m!=j) r[j]=r[j]*d[m]/d[j*(k+2)+m];
    }
    for (j=1;j<=k;j++)
    {
        s[j]=d[1]/d[j];
        for (m=1;m<=k;m++)
            if (m!=j) s[j]=s[j]*d[m]/d[j*(k+2)+m];
    }
    s[0]=0;
    for (m=1;m<=k;m++)
        s[0]=s[0]-s[m];
    u=1;
    for (m=0;m<=k;m++)
       u=u*d[m+1];
    v=d[1]/d[k+1];
}

/*********************************************************************/
/* After obtaining the solution x_n for the present timing point t_n,     */
/* renew the coefficients s_i (resp.; r_i) in the approximation of        */
/*              x_dot = sum (s_i*x_(n+1-i))  i=0 to k                     */
/* (resp.;        x^p = sum (r_i*x_(n+1-i)) , i=1 to k+1  )               */
/* for the next timing point t_n+1.                                       */
/*********************************************************************/

re_form()

{
    int m;
    double gx,uv;

/* all the following calculations follow the formulas of BDF-1, */
/* BDF-2, and BDF-3 in p.107 of the reference paper; where      */
/*                       u <==> F(n+1,k)                        */
/*                       v <==> delta(n+1,k)                    */
/*                      gx <==> G1                              */
/*                     r_j <==> r_j(n+1,k)                      */
/*                     s_j <==> alpha_j(n+1,k)                  */

    uv=u*v;
    u=1;
    for (m=0;m<=k;m++)
        u=u*d[m+1];
```

```
    gx = -u/uv;
    v=d[1]/d[k+1];
    for (m=1;m<=k;m++)
        r[m+1]=gx*s[m]*d[k+2+m+1]/d[m+1];
    r[1]=0;
    for (m=1;m<=k;m++)
        r[1]=r[1]+r[m+1];
    r[1]=1-r[1];
    for (m=1;m<=k;m++)
        s[m]=v*r[m]*d[m*(k+2)+k+1]/d[m];
    s[0]=0;
    for (m=1;m<=k;m++)
        s[0]=s[0]+s[m];
    s[0] = -s[0];
}

/*************************************************************************/
/* Renew the time and solution vector when the solution at next point is  */
/* obtained such that t[0],t[1],...,t[k] and x[i],x[i+n],...,x[i+k*n] for */
/* i=0,1,...,(n-1), always represent the time and solution of the newly   */
/* computed k+1 points.                                                    */
/*************************************************************************/

renew()

{
    int i,j;

    for (i=0;i<n;i++)
        x[i+(k+1)*n]=z[i];
    for (j=1;j<=k+1;j++)
    {
        t[j-1]=t[j];
        for (i=0;i<n;i++)
            x[i+(j-1)*n]=x[i+j*n];
    }
}
```

```
#include <stdio.h>

extern int n,k;
extern double h,*c,*z,*t,*x;

/*************************************************************************/
/* Initiate the BDF routine by computing solutions at the starting k+1     */
/* points for the k-th order BDF integration routine.                      */
/*************************************************************************/

start()
{
    int i;

    /* uniform stepsize in starting period */
    for (i=1;i<=k+1;i++)
        t[i]=t[0]+i*h;

    switch (k)
    {
        case 1 : first_order(); break;
        case 2 : second_order(); break;
        case 3 : third_order(); break;
        case 4 : fourth_order(); break;
        case 5 : fifth_order(); break;
        case 6 : sixth_order(); break;
    }
}

/*************************************************************************/
/* use the 1st order BDF formula to compute the solution of next point .  */
/*************************************************************************/

first_order()

{
    int i;
    double ha;

    for (i=0;i<n;i++)
    {
        c[i] = -x[i]/h;
        z[i]=x[i]+h;
    }
    ha=1/h;
    nr(1,ha,t[1]);
    for (i=0;i<n;i++)
        x[i+n]=z[i];
}

/*************************************************************************/
/* Use the 2nd order BDF formula to compute the solutions of next two     */
/* points.                                                                 */
/*************************************************************************/

second_order()
```

```
{
    int i;
    double ha;

    first_order();
    for (i=0;i<n;i++)
    {
        c[i] = -(2*x[i+n]-x[i]/2)/h;
        z[i]=2*x[i+n]-x[i];
    }
    ha=1.5/h;
    nr(1,ha,t[2]);
    for (i=0;i<n;i++)
        x[i+2*n]=z[i];
}

/*************************************************************************/
/* Use the 3rd order of BDF formula to compute the solutions at the next  */
/* three points.                                                          */
/*************************************************************************/

third_order()

{
    int i;
    double ha;

    second_order();
    for (i=0;i<n;i++)
    {
        c[i] = -(x[i]/3-1.5*x[i+n]+3*x[i+2*n])/h;
        z[i]=x[i]-3*x[i+n]+3*x[i+2*n];
    }
    ha=11/(6*h);
    nr(1,ha,t[3]);
    for (i=0;i<n;i++)
        x[i+3*n]=z[i];
}

/*************************************************************************/
/* Use the 4-th order BDF formula to compute the solutions at the next    */
/* four points.                                                           */
/*************************************************************************/

fourth_order()

{
    int i;
    double ha;

    third_order();
    for (i=0;i<n;i++)
    {
        c[i] = -(-x[i]/4+4*x[i+n]/3-3*x[i+2*n]+4*x[i+3*n])/h;
        z[i] = -x[i]+4*x[i+n]-6*x[i+2*n]+4*x[i+3*n];
    }
```

```
    ha=25/(12*h);
    nr(1,ha,t[4]);
    for (i=0;i<n;i++)
        x[i+4*n]=z[i];
}

/*************************************************************************/
/* Use the 5-th order BDF formula to compute the solutions at the next    */
/* five points.                                                           */
/*************************************************************************/

fifth_order()

{
    int i;
    double ha;

    fourth_order();
    for (i=0;i<n;i++)
    {
        c[i] = -(0.2*x[i]-1.25*x[i+n]+3.333*x[i+2*n]-5*x[i+3*n]+5*x[i+4*n])/h;
        z[i]=x[i]-5*x[i+n]+10*x[i+2*n]-10*x[i+3*n]+5*x[i+4*n];
    }
    ha=137/(60*h);
    nr(1,ha,t[5]);
    for (i=0;i<n;i++)
        x[i+5*n]=z[i];
}

/*************************************************************************/
/* Use the 6-th order BDF formula to compute the solutions at the next    */
/* six points.                                                            */
/*************************************************************************/

sixth_order()

{
    int i;
    double ha;

    fifth_order();
    for (i=0;i<n;i++)
    {
        c[i] = -x[i]/6+1.2*x[i+n]-15*x[i+2*n]/4+20*x[i+3*n]/3;
        c[i] = -(c[i]-7.5*x[i+4*n]+6*x[i+5*n])/h;
        z[i] = -x[i]+6*x[i+n]-15*x[i+2*n]+20*x[i+3*n]-15*x[i+4*n]+6*x[i+5*n];
    }
    ha=147/(60*h);
    nr(1,ha,t[6]);
    for (i=0;i<n;i++)
    {
        x[i+6*n]=z[i];
        z[i]=x[i]-7*x[i+n]+21*x[i+2*n]-35*x[i+3*n]+35*x[i+4*n]
             -21*x[i+5*n]+7*x[i+6*n];
    }
}
```

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

extern int n,k;
extern double h,*t,*x,*c,*z,*r,*s,*rx,*sx,*d;
extern double ft,*y;
extern char *var_name[];

double u,v,ux,vx;
double xmin,xmax,ymin,ymax;
int *di,*scale,pw,xv,yv;
double *sca_x;
extern int lx,outfile;
extern double t0,hy;
extern FILE *op;

/*************************************************************************/
/* Follow the BDF formula in the reference paper to compute the solution   */
/* of the implicit differential equation                                   */
/*                        f(x,x_dot,t) = 0                                 */
/*************************************************************************/

bdf_comp()
{
    int cg;
    double ratio,ha,pow();
    GRAF *gp;
    FILE *fopen();

    calloci(n,&di,"di");
    calloci(n,&scale,"scale");
    callocd(n,&sca_x,"sca_x");

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

    t[k+1]=t[k]+h;

    /* establish the coefficients for the approximation of */
    /* x_dot and the initial guess of x                    */
    ini_form();

    /* find the transient response until the final time is reached */
    while (t[k+1]<ft)
    {
        /* approximate x_dot and the initial guess of x */
        appx_d_ig(&ha);

        /* reduce the differential equation f(x,x_dot,t)   */
        /* to a nonlinear algebraic equation and solve it  */
        /* by Newton-Raphson iteration; cg=-1 if iteration */
        /* does not converge                               */
        cg=nr(0,ha,t[k+1]);

        /* evaluate the truncation error */
        trunca_err(&ratio);
```

```
        /* reduce stepsize if nonconvergent iteration or */
        /* excessive truncation error                    */
        if (ratio >= 1 || cg==-1)
            reduce_h(cg,ratio);

        else
        {
            /* accept the computed solution and renew the */
            /* timing vector and the solution vector      */
            renew();

            if (pw==1)                  /* 2-dim phase plot */
                draw_phase(gp);
            else
                draw_waveform(gp);      /* plot the transient waveform */

            /* increase h if very small truncation error */
            if (ratio < 0.5)
                h=2*h;

            /* the stepsize should not exceed the printing stepsize */
            if (outfile == 1 && h > hy)
                h=hy;

            /* renew the parameters in BDF formula for  */
            /* next point iteration                     */
            next_pt();
        }
        t[k+1]=t[k]+h;          /* new timing point */

        /* renew the coefficients in BDF formula */
        re_form();
    }
    graf_close(gp);
    mgideagp();
    if (outfile == 1)
        fclose(op);
}

/*************************************************************************/
/* 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();

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

    mgiclearpln(0,-1,0);
    mgihue(3);
    /* 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,"%s1E%d",var_name[xv],scale[xv]);
    strcpy(x_name,line);
    sprintf(line,"%s1E%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,"%s1E%d",var_name[i],scale[i]);
            mgihue(i+2);
            mgigfs(80+(j++)*80,540,0,xi);
        }
    }
}

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

draw_phase(gp)
GRAF *gp;
{
    double ratx;

    mgihue(3);
    graf_move(sca_x[xv]*x[xv+n*(k-1)],sca_x[yv]*x[yv+n*(k-1)],gp);
    graf_draw(sca_x[xv]*z[xv],sca_x[yv]*z[yv],gp);
    if (outfile == 1)
        /* the l-th output point is reached */
        if ((t[k-1]-t0-lx*hy)*(t[k]-t0-lx*hy) < 0)
        {
            ratx=(t0+lx*hy-t[k-1])/h;
            fprintf(op,"%.3e\t\t%.3e\t\t\%.3e\n",t0+lx*hy,
                    (1-ratx)*x[xv+n*(k-1)]+ratx*z[xv],
                    (1-ratx)*x[yv+n*(k-1)]+ratx*z[yv]);
            lx++;
        }
}

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

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

    if (outfile == 1)
        /* the l-th output point is reached */
        if ((t[k-1]-t0-lx*hy)*(t[k]-t0-lx*hy) < 0)
```

```
        {
            ratx=(t0+lx*hy-t[k-1])/h;
            j=1;
            if (t>=0)
                fprintf(op,"%.3e  ",t0+lx*hy);
            else
                fprintf(op,"%.3e ",t0+lx*hy);
            lx++;
        }
    for (i=0;i<n;i++)
    {
        if (di[i]==1)
        {
            mgihue(i+2);
            graf_move(t[k-1],sca_x[i]*x[i+n*(k-1)],gp);
            graf_draw(t[k],sca_x[i]*z[i],gp);
            if (j == 1)
            {
                zout=(1-ratx)*x[i+n*(k-1)]+ratx*z[i];
                if (zout >= 0)
                    fprintf(op,"%.3e  ",zout);
                else
                    fprintf(op,"%.3e ",zout);
            }
        }
    }
    if (j == 1)
        fprintf(op,"\n");
}
```

```
#include "stdio.h"

extern int n;
extern double *x,*t;
char *var_name[50]; /* string representation for each variable */
int *iz,lx=1,outfile=0;
double *f,*jf,*dummy1,*dummy2;
double *wi,t0,hy;
FILE *op; /* pointer to the output file "xx...x.out" */

/*************************************************************************/
/* Given the initial condition (capacitor voltage and inductor current) of */
/* each dynamic element, perform a dc analysis to find the dc operating    */
/* point ( capacitor current, inductor voltage, voltage (resp.; current)   */
/* of v-controlled (resp.; i-controlled) nonlinear resistor, current       */
/* (resp.; voltage) of time-varying voltage source (resp.; current         */
/* source)) corresponding to the initial condition.                        */
/*************************************************************************/

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

    /* open the variable-table file */
    open_tbl_op(argc,argv,&fp);

    /* read the starting time and allocate space for timing variable */
    callocd(8,&t,"t");
    printf("enter the initial time\n");
    scanf("%lf",t);

    /* if -o option is specified in the command line */
    if (outfile == 1)
    {
        t0=t[0];
        printf("enter the printing stepsize for the output file\n");
        scanf("%lf",&hy);
    }

    /* perform dc analysis */
    dc_simu(fp);
    printf("\nend of dc analysis\n\n");
}

/*************************************************************************/
/* Open the file containing the mapping table between the circuit variables*/
/* (voltages and currents) and the equation variables (independent         */
/*  variables  x and dependent variables y)                                */
/*************************************************************************/

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

```
{
    FILE *fopen();
    char *sc,line[12];

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

    if (argc != 1)
        exit_message("BDF TABLE_FILE");
    sprintf(line,"%s.tbl",*argv);
    if ((*fp=fopen(line,"r"))==NULL)
    {
        printf("CAN'T OPEN THE TABLE FILE %s\n",line);
        exit();
    }
    if (outfile == 1)
    {
        sprintf(line,"%s.out",*argv);
        if ((op=fopen(line,"w")) == NULL)
        {
            printf("CAN'T OPEN THE OUTPUT DATA FILE %s\n",line);
            exit();
        }
    }
}

/*************************************************************************/
/* DC simulation for DC operating point(s)                               */
/*************************************************************************/

dc_simu(fp)
FILE *fp;
{
    int iter=1,j;

    /* allocate spaces for the variables used in */
    /* equation and Jacobian matrix routines     */
    var_alloc(&n,&f,&x,&dummy1,&dummy2,&jf);

    /* read the table and enter the initial condition for    */
    /* each dynamic element, return j=-1 if each variable    */
    /* is either a capacitor voltage or an inductor current, */
    /* hence the starting point is immediately available     */
    /* and no need to perform the dc analysis                */
    if ((j=get_tbl(fp))==1)
        printf("start dc analysis for starting point .....\n\n");
    fclose(fp);

    while(iter==1 && j==1)      /* while iteration has't converged */
```

```
    {
        /* get the initial guess */
        get_ini_gs();

        if (newton()==-1)       /* not convergent */
            no_cg(&iter);
        else                    /* convergent */
            dc_pt(&iter);
    }
    cfree(f);
    cfree(jf);
    cfree(dummy1);
    cfree(dummy2);
}

/*************************************************************************/
/* Get the initial guess for Newton-Raphson iteration; zero for default    */
/* initial guess.                                                          */
/*************************************************************************/

get_ini_gs()
{
    int i;
    char ch[2],st[30];

    printf("default initial guess? y/n\n");
    scanf("%1s",ch);
    if (ch[0]=='n')
    {
        printf("enter the initial guess\n");
        for (i=0;i<n;i++)
        {
            strcpy(st,var_name[i]);
            if (find_index("v(C",st)==0)
                replace(st,"v(C","i(C");
            if (find_index("i(L",st)==0)
                replace(st,"i(L","v(L");
            printf("%s=",st);
            scanf("%lf",x+i);
        }
    } else
        for (i=0;i<n;i++) x[i]=0.0;
}

/*************************************************************************/
/* Print the computed dc operating point in the output file filename.$op.  */
/*************************************************************************/

dc_pt(iter)
int *iter;
{
    char ch[2],line[30];
    int i;

    printf("\nstarting point .....\n");
    for (i=0;i<n;i++)
```

```
    {
        /* capacitor current and inductor voltage are chosen as variables */
        /* in dc analysis for finding the starting point; restore the     */
        /* given initial capacitor voltage or inductor current to the     */
        /* starting point of dynamic elements                             */
        if (iz[i]==1)
            x[i]=wi[i];

        sprintf(line,"%s=%.3e\n",var_name[i],x[i]);
        printf("%s",line);
    }
    printf("\nwould you like to try another starting point? y/n\n");
    scanf("%1s",ch);
    if (ch[0]=='n') *iter=0;
}

/*********************************************************************/
/* Print the last iteration point when Newton-Raphson iteration does not   */
/* converge.                                                               */
/*********************************************************************/

no_cg(iter)
int *iter;
{
    int i;
    char ch[2],st[30];

    printf("solution at last iteration is...\n");
    for (i=0;i<n;i++)
    {
        strcpy(st,var_name[i]);
        if (find_index("v(C",st)==0)
            replace(st,"v(C","i(C");
        if (find_index("i(L",st)==0)
            replace(st,"i(L","v(L");
        printf("%s=%.3e\n",st,x[i]);
    }
    printf("would you like to continue? y/n\n");
    scanf("%1s",ch);
    if (ch[0]=='n') *iter=0;
}

/*********************************************************************/
/* Newton-Raphson iteration for solving the nonlinear equation             */
/*              f(x) = 0                                                   */
/*********************************************************************/

newton()
{
    int k=0,*ipvt;
    double *zq;
    double rcond,max,*z,fabs(),*ff;

    /* allocate spaces for the variables used in calling */
    /* Linpack routines sgeco and sgesl                  */
    alloc_1(&ipvt,&zq,&z,&ff);
```

```
    /* Newton-Raphson iteration */
    while (++k<20)              /* limited to 20 iterations */
    {
        /* evaluate f(x) */
        ini_eq(n,iz,f,x,wi,dummy1,t[0]);

        /* prevent overflow */
        if (pre_ovfl(k,&max,z,ff)==-1)
            return(-1);

        if (max<1.0e-7)         /* convergent */
            break;

        /* find the Jacobian matrix jf */
        ini_jacob(x,jf,t[0]);

        /* iteration formula : x_n+1 = x_n - inv(jf)*f(x_n) */
        sgeco(jf,n,ipvt,&rcond,zq);     /* LU decomposion for jf */
        if (fabs(rcond)<1.0e-16)        /* singular Jacobian matrix */
        {
            printf("SINGULAR JACOBIAN MATRIX FOR DC ANALYSIS\n");
            return(-1);
        } else
            next_iter(ipvt,z,ff);       /* find next iteration pt x_n+1 */
    }
    free_1(ipvt,zq,z,ff);
    if (k>=20) return(-1);
    else return(1);
}

/*************************************************************************/
/* If f(x) numerically overflows, reduce the distance between sequential   */
/* iteration points to avoid overflow (especially due to exp function).    */
/*************************************************************************/

pre_ovfl(k,max,z,ff)
int k;
double *max,*ff,*z;
{
    int i,j=0;
    double norm();

    /* reduce iteration distance if f(x)>100 for k-th iteration with */
    /* k>1 (iteration distance |x_k - x_k-1| is undefined for k=1    */
    while ((*max=norm(n,f))>100 && k>1)
    {
        if (++j>10)
            return(-1);
        for (i=0;i<n;i++)
        {
            ff[i]=0.5*ff[i];
            x[i]=z[i]-ff[i];
        }
        ini_eq(n,iz,f,x,wi,dummy1,t[0]);        /* re-evaluate f(x) */
    }
```

```
    return(1);
}

/*************************************************************************/
/* Find the next iteration point x_k+1 = x_k - inv(jf)*f(x_k) and save     */
/* the previous iteration point x_k and the iteration distance in case     */
/* the iteration distance has to be reduced due to the overflow at the new */
/* iteration point x_k+1                                                   */
/*************************************************************************/

next_iter(ipvt,z,ff)
int *ipvt;
double *z,*ff;
{
    int i;

    sgesl(jf,n,ipvt,f,0);       /* find inv(jf)*f */
    for (i=0;i<n;i++)
    {
            z[i]=x[i];          /* save previous point  */
            ff[i]=f[i];         /* iteration distance   */
            x[i]=z[i]-f[i];     /* next iteration point */
    }
}

/*************************************************************************/
/* Find the 1-1 norm of a vector.                                          */
/*************************************************************************/

double norm(m,y)
int m;          /* vector dimension */
double y[];     /* vector */
{
   int i;
   double max=0.0,fabs();

   for (i=0;i<m;i++)
      if (max<fabs(y[i])) max=fabs(y[i]);
   return(max);
}

/*************************************************************************/
/* Read the file with the mapping table (between the circuit variables v,i */
/* and the equation variables x,y) and input the initial condition for     */
/* each dynamic element (L or C) which will remain constant in the circuit */
/* equation routine for evaluating f(x).                                   */
/*************************************************************************/

get_tbl(fp)
FILE *fp;
{
    int i= -1,j,k;
    char ex[3],line[30],d[30],*calloc();

    calloci(n,&iz,"iz");
    callocd(n,&wi,"wi");
```

```
    while (fgets(line,30,fp)!=NULL)
    {
        /* get the variable index j from "x[j]=xxx...xxx" */
        strdel(line,0,2);
        k=find_index("]=",line);
        strncpy(ex,line,k);
        j=atoi(ex);

        /* if dynamic element, input the initial condition  */
        /* else i=1 denotes there exists memoryless element */
        if (find_index("(C",line)>0 || find_index("(L",line)>0)
            ini_CL(j,line,d);
        else
        {
            iz[j] = -1;
            i=1;
        }

        /* extract the name of circuit element; e.g., R1, Vin */
        k=find_index("=",line);
        strdel(line,0,k+1);
        line[strlen(line)-1]='\0';
        var_name[j]=calloc(10,sizeof(char));
        strcpy(var_name[j],line);
    }
    return(i);
}

/*************************************************************************/
/* Get the initial condition for the dynamic element C and L.            */
/*************************************************************************/

ini_CL(j,line,d)
int j;
char line[],d[];
{
    strcpy(d,line);
    strdel(d,strlen(d)-2,2);
    strdel(d,0,find_index("(",d)+1);
    printf("enter the initial condition for %s\n",d);
    scanf("%lf",wi+j);
    iz[j]=1;
}

/*************************************************************************/
/* Allocate spaces for the variables used in Linpack routines sgeco and  */
/* sgesl.                                                                */
/*************************************************************************/

alloc_1(ipvt,zq,z,ff)
int **ipvt;
double **zq,**z,**ff;
{
    char *calloc();

    *ipvt=(int *)calloc(n,sizeof(int));
```

```
    *zq=(double *)calloc(n,sizeof(double));
    *z=(double *)calloc(n,sizeof(double));
    *ff=(double *)calloc(n,sizeof(double));
}

/*************************************************************************/
/* Free the spaces for the variables used in Linpack routines sgeco and    */
/* sgesl when finishing Newton-Raphson iteration.                          */
/*************************************************************************/

free_1(ipvt,zq,z,ff)
int *ipvt;
double *zq,*z,*ff;
{
    cfree(ipvt);
    cfree(zq);
    cfree(z);
    cfree(ff);
}
```

```
#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);
}
```

```
#include <stdio.h>

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

replace(d,e1,e2)
char d[],e1[],e2[];
{
    char *dx,*calloc();
    int k;

    k=find_index(e1,d);
    dx=calloc(strlen(d)-k,sizeof(char));
    strcpy(dx,d+k+strlen(e1));
    strcpy(d+k,e2);
    strcpy(d+k+strlen(e2),dx);
}

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

rep_all(d,e1,e2)
char d[],e1[],e2[];
{
    int k;

    while ((k=find_index(e1,d))>=0)
    {
        if (k>0 && d[k-1]>='a' && d[k-1]<='z')
            d[k]='$';
        else replace(d,e1,e2);
    }
    while ((k=find_index("$",d))>=0)
        d[k]=e1[0];
}

/*************************************************************************/
/*************************************************************************/
/*
   Convert a real number expression terminated by a unit character to a
   real number with double precision.                                    */

double stof(s)

char *s;        /* input string expression */
{
    char *d;    /* number field expression */
    char ch;    /* unit character */
    char *calloc();
    double x,atof();

    d=calloc(strlen(s)+1,sizeof(char));
    ch = *(s+strlen(s)-1);      /* extract the last character */
    if (ch<'0' || ch>'9')       /* if it is a unit character */
    {
        strcpy(d,s);
```

```
        strdel(d,strlen(s)-1,1);        /* extract the number field */
        x=atof(d);
        switch(ch)
        {
            case 'K' : x=x*1e3;   break;        /* Kilo */
            case 'M' : x=x*1e6;   break;        /* Mega */
            case 'G' : x=x*1e9;   break;        /* Giga */
            case 'T' : x=x*1e12;  break;        /* Tera */
            case 'm' : x=x*1e-3;  break;        /* milli */
            case 'u' : x=x*1e-6;  break;        /* micro */
            case 'n' : x=x*1e-9;  break;        /* naro */
            case 'p' : x=x*1e-12; break;        /* pico */
            case 'f' : x=x*1e-15; break;        /* femptl */
            default  : {
                            printf("UNDEFINED UNIT CHARACTER %c\n",ch);
                            exit();
                        }
                        break;
        }
    }
    else x=atof(s);
    return(x);
}

/*************************************************************************/
/*************************************************************************/
/*
   Give the sign of a real number x; 1 if x>=0 and -1 if x=0.            */

sgn(x)

double x;
{
    if (x>=0) return(1);
    else return(-1);
}

/*************************************************************************/
/*************************************************************************/
/*
   Switch two real number.                                               */

switch_d(a,b)

double *a,*b;
{
    double c;

    c = *a;
    *a = *b;
    *b = c;
}

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

```
   Switch two integer number.                                          */

switch_i(a,b)

int *a,*b;
{
    int c;

    c = *a;
    *a = *b;
    *b=c;
}

/*************************************************************************/
/*************************************************************************/
/*
   Find the i-th word w from the string s.                             */

find_word(s,w,i)

char *s,*w;
int i;
{
    int k=1,m=0,n=0,j;
    char *ps;

    j=strlen(s)+1;
    ps=s;       /* starting position in the input string s */
    while (*s==' ' && m++<j)    /* delete leading blanks in s */
        s++;
    if (m<j)
    {
        while (k<i && n++<j)
        {
            /* search the starting position for the i-th word */
            if (*s==' ' && *(s+1)!=' ')
                k++;
            s++;
        }
        while (*s!=' ' && *s!='\t' && *s!='\0' && n<j)
            *w++ = *s++;        /* copy the i-th word to w */
        *w='\0';
    }
    if (m>=j || n>=j)
    {
        printf("FAIL TO FIND A WORD IN %s\n",ps);
        exit();
    }
}

/*************************************************************************/
/*************************************************************************/
/*
   Find the position of the string t within the string s; -1 is returned if
   t is not found within s.                                            */
```

```
find_index(t,s)
char *s,*t;
{
    int i,j,k;

    for (i=0;s[i]!='\0';i++)
    {
        for (j=i,k=0;t[k]!='\0' && s[j]==t[k];j++,k++);
        if (t[k]=='\0')
            return(i);
    }
    return(-1);
}

/*************************************************************************/
/*************************************************************************/
/*
   Delete n characters from the n1-th position of string s.              */

strdel(s,n1,n)
char *s;
int n1,n;
{
    int i,j,k=0;

    j=strlen(s)+1;
    for (i=0;k<=j,s[i+n1+n]!='\0';k++,i++)
        s[i+n1]=s[i+n1+n];
    if (k>j)
    {
        printf("ERROR IN STRDEL IN STRING\n");
        printf("%s\n",s);
        exit();
    }
    else s[i+n1]='\0';
}

/*************************************************************************/
/*************************************************************************/
/*
   Delete all the blanks within the string s.                            */

sqeez(s)
char *s;
{
    char *sx,*tx,*t,*calloc();
    int i=0,k;

    sx=s;
    k=strlen(s);
    t=calloc(k+1,sizeof(char));
    tx=t;
    while (*sx != '\0' && i++<k)
    {
        if (*sx != ' ' && *sx != '\t' && *sx != '\n')
            *t++ = *sx++;
```

```
        else sx++;
    }
    *t='\0';
    if (i>k){
        printf("ERROR IN SQEEZ WITH I=%d\n",i);
        exit();
    }
    else strcpy(s,tx);
}

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

exit_message(message)
char *message;
{
    printf("%s\n",message);
    exit();
}

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

calloci(n,pt,s)
int n,**pt;
char *s;
{
    char *calloc();

    if ((*pt=(int *)calloc(n,sizeof(int)))==NULL)
    {
        printf("CAN'T ALLOCATE SPACE FOR %s\n",s);
        exit();
    }
}

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

callocd(n,pt,s)
int n;
double **pt;
char *s;
{
    char *calloc();

    if ((*pt=(double *)calloc(n,sizeof(double)))==NULL)
    {
        printf("CAN'T ALLOCATE SPACE FOR %s\n",s);
        exit();
    }
}

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

```
ralloci(ip,newsize,oldsize)
int **ip,newsize,oldsize;
{
    char *calloc();
    int i,size,*pt;

    pt = *ip;
    if ((*ip=(int *)calloc(newsize,sizeof(int)))==NULL)
        exit_message("CAN'T RE_ALLOCATE");
    if (newsize<oldsize) size=newsize;
    else size=oldsize;
    for (i=0;i<size;i++)
        (*ip)[i]=pt[i];
    cfree(pt);
}

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

rallocd(dp,newsize,oldsize)
int newsize,oldsize;
double **dp;
{
    char *calloc();
    int i,size;
    double *pt;

    pt = *dp;
    if ((*dp=(double *)calloc(newsize,sizeof(double)))==NULL)
        exit_message("CAN'T RE_ALLOCATE\n");
    if (newsize<oldsize) size=newsize;
    else size=oldsize;
    for (i=0;i<size;i++)
        (*dp)[i]=pt[i];
    cfree(pt);
}

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