NONLINEAR ELECTRONICS (NOEL) PACKAGE 8: PHASE PORTRAIT OF SECOND ORDER NONLINEAR CIRCUIT

by

An-Chang Deng and Leon O. Chua

Memorandum No. UCB/ERL M86/46

3 June 1986

NONLINEAR ELECTRONICS (NOEL) PACKAGE 8:

PHASE PORTRAIT OF SECOND ORDER NONLINEAR CIRCUIT

by

An-Chang Deng and Leon O. Chua

Memorandum No. UCB/ERL M86/46

3 June 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

NONLINEAR ELECTRONICS (NOEL) PACKAGE 8:

PHASE PORTRAIT OF SECOND ORDER NONLINEAR CIRCUIT

by

An-Chang Deng and Leon O. Chua

Memorandum No. UCB/ERL M86/46

3 June 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

NOEL PACKAGE 8 : PHASE PORTRAIT OF SECOND ORDER NONLINEAR CIRCUITS

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 finds the phase portrait of the second order dynamic circuits. It applies the BDF integration routine to trace the *forward* and *backward* phase trajectories corresponding to initial conditions chosen from the grid points of the phase plane.

June 3, 1986

This research is supported in part by the Office of Naval Research Contract N00014-76-C-0572 and the National Science Foundation Grant ECS-8313278.

NOEL PACKAGE 8 : PHASE PORTRAIT OF SECOND ORDER NONLINEAR CIRCUITS

1. Introduction

The phase portrait of a second order circuit is an important problem in nonlinear circuit analysis. To find the phase portrait, the user has to provide the symbolic representation of the state equation of the circuit to be analyzed, and this circuit formulation step usually causes some inconveniencies. Moreover, if each voltage-controlled (resp.; current-controlled) nonlinear resistor is not connected in parallel (resp.; in series) with a capacitor (resp.; an inductor), the symbolic representation of the explicit state equation $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$ may not exist, or can be obtained only after solving a nonlinear equation.

To make the phase portrait program more user-friendly, we apply the circuit formulation utility program[1] to formulate the second order dynamic circuit to an implicit state equation

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

where the dimension of $\mathbf{x}$ is no less than two. Similar to the nonlinear transient analysis program in package 4 [2], this equation is written in the form of a C source code, which also includes the symbolic expression of the Jacobian matrix

$$\mathbf{J}_f = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}$$

This equation is then solved by the BDF (Backward Differentiation Formula) integration routine which finds the *forward* and *backward* transient trajectories for each initial condition; namely, the transient responses for $t \geq t_0$ and $t \leq t_0$ respectively, where t_0 is the starting time. The initial conditions are chosen from the grid points of the phase plane; namely,

$$x = x_{\min} + (x_{\max} - x_{\min}) * i / (m+1)$$
$$y = y_{\min} + (y_{\max} - y_{\min}) * j / (m+1)$$

for $i = 1,2,...,m$, $j = 1,2,...,m$, where $x_{\min}$ (resp.; $x_{\max}$) is the lower bound (resp.; upper bound) of the x-coordinate, and $y_{\min}$ (resp.; $y_{\max}$) is the lower bound (resp.; upper bound) of the y-coordinate in the phase plane, and m^2 initial conditions are chosen in the phase portrait. The computed forward and backward transient trajectories are shown in the phase plane and emanate from each initial condition with distinct colors.

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) initial stepsize h

(4) phase plane boundaries : x_{min}, x_{max}, y_{min}, y_{max}

(5) number of grid points in each axis : m

Step 4.

Perform the dc analysis to find the dc operating point $\mathbf{x}_0$ at starting time t_0; namely, $\mathbf{f}(\mathbf{x}_0 , \dot{\mathbf{x}}_0 , t_0) = 0$.

Step 5.

For each initial condition located in the grid position of the phase plane, follow the BDF procedures (Step 6 of Section 2 in package 4 [2]) to perform the forward transient analysis to find the phase trajectory for $t \geq t_0$ until $t \geq t_f$, or until the trajectory goes out of the phase plane defined by $x_{min} \leq x \leq x_{max}$ and $y_{min} \leq y \leq y_{max}$.

Step 6.

Find the backward phase trajectory for $t \leq t_0$ until $t < -t_f$. This backward transient analysis follows the same procedures in Step 5 except using *negative* stepsize $h < 0$.

3. User's Instruction

Step 1.

Create a file "xx...x.spc" which describes the second order dynamic circuit to be analyzed and follows the rules of the input format language defined in [3] 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 [3] 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.)

The circuit should contain ***exactly two*** dynamic elements (capacitor or inductor) for second order phase portrait. 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

phase2 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

phap 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 phase portrait.

(b) **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.

(c) **enter the initial stepsize**

Enter the initial stepsize used in starting stage, which should be small enough not to cause large truncation error.

(d) **enter the number of grid points in x and y axis**

enter the integer m such that m^2 grid points of the phase plane are chosen as the initial conditions.

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

x[0] = i(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)

y_variable = x[?]

Similar to the x variable.

(e) **enter xmin and xmax**

Enter the lower bound xmin and the upper bound xmax for the x coordinate in the phase plane.

(f) **enter ymin and ymax**

Enter the lower bound ymin and the upper bound ymax for the y coordinate in the phase plane.

4. Examples

Example 1 : in file "ex1.spc"

Second order non-autonomous circuit with a nonlinear inductor (Fig.1).

Example 2 : in file "ex2.spc"

Second order circuit with two nonlinear resistors (Fig.2).

Example 3 : in file "ex3.spc"

Second order Van der Pol circuit (Fig.3).

5. Diagnosis

See Section 6 of [2].

References

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

[2] A.C.Deng and L.O.Chua, "NOnlinear ELectronics package 4 : nonlinear transient analysis," ERL Memo., M86/27, University of California, Berkeley, Mar. 28, 1986.

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

Figure Captions

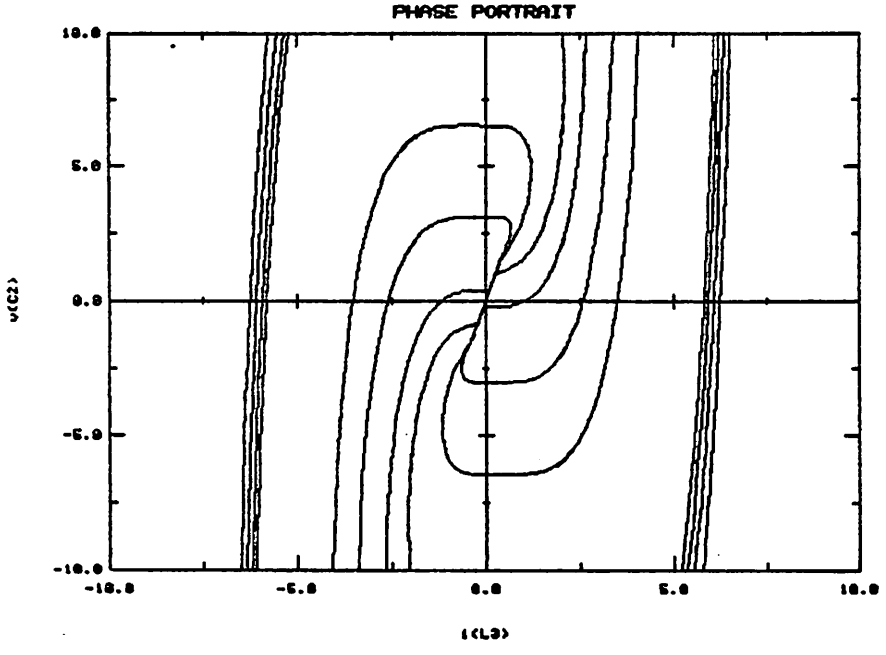
Fig.1 A second order non-autonomous circuit with a nonlinear inductor.

Fig.2 A second order circuit with two nonlinear resistors.

Fig.3 A second order circuit with Van der Pol equation.

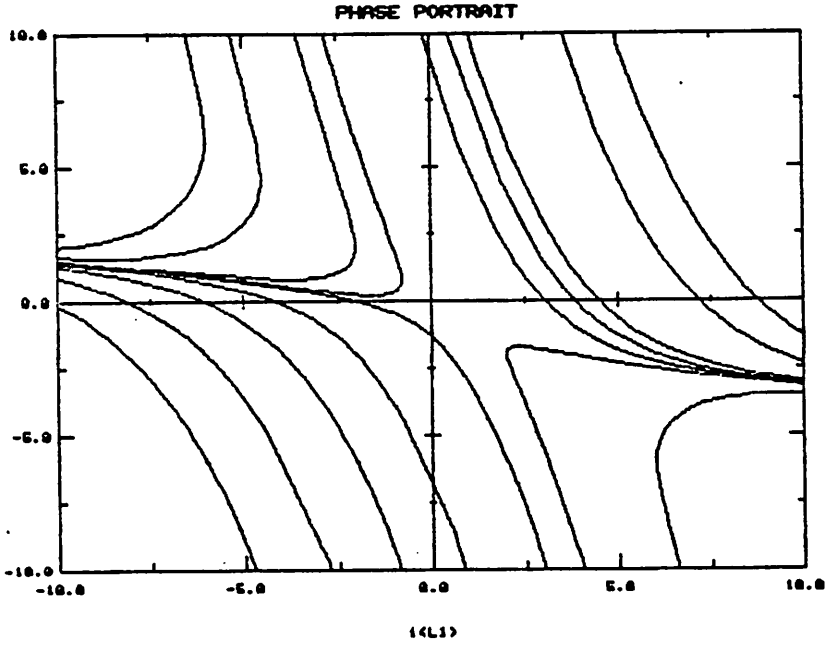
```
* Example 1
*
* 2nd order non-autonomous circuit with nonlinear inductor
Vin 1 0 {sin(t)}
R1 1 2 10
C2 2 0 1
L3 2 3 {phi=ipow(i,3)}
R4 3 0 4
.include "math.h"
.end
```

```
enter the initial time
0
enter the final time
10
enter the initial stepsize
1e-4
enter the number of grid points in x and y axis
4
x[0]=i(Vin)
x[1]=i(L3)
x[2]=v(C2)
x_variable=x[?]
1
y_variable=x[?]
2
enter xmin and xmax
-10 10
enter ymin and ymax
-10 10
```


PHASE PORTRAIT
10.0
5.0
0.0
-5.0
-10.0
-10.0
-5.0
0.0
5.0
10.0
v(C2)
i(L3)

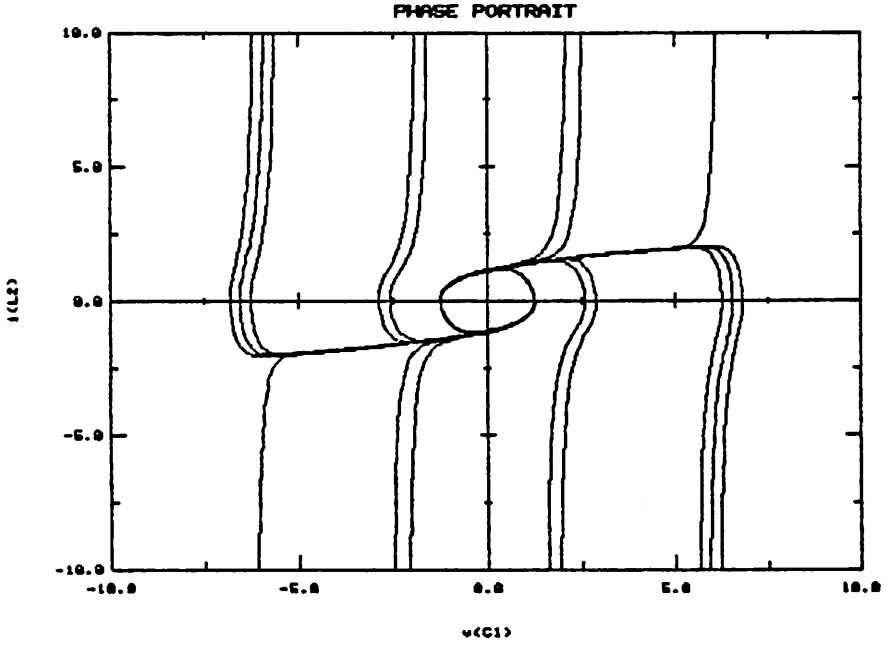
```
* Example 2
*
* 2nd order circuit with 2 nonlinear resistors
L1 1 0 1
C1 2 0 1
Ra 2 0 {v=3*i+2}
R3 1 2 -1
Rnon 1 0 {i=ipow(v,3)+2*v}
.include "math.h"
.end
```

```
enter the initial time
0
enter the final time
10
enter the initial stepsize
1e-4
enter the number of grid points in x and y axis
4
x[0]=i(L1)
x[1]=v(C1)
x[2]=i(Ra)
x[3]=v(Rnon)
x_variable=x[?]
0
y_variable=x[?]
1
enter xmin and xmax
-10 10
enter ymin and ymax
-10 10
```


PHASE PORTRAIT
10.0
5.0
0.0
-5.0
-10.0
-10.0
-5.0
0.0
5.0
10.0
ω(L1)
ι(L1)

```
* Example 3
*
* 2nd order Van der Pol circuit
C1 1 0 1
L2 1 2 1
R3 2 0 {v=-i+ipow(i,3)}
.include "math.h"
.end
```

```
enter the initial time
0
enter the final time
10
enter the initial stepsize
1e-4
enter the number of grid points in x and y axis
4
x[0]=v(C1)
x[1]=i(L2)
x[2]=i(R3)
x_variable=x[?]
0
y_variable=x[?]
1
enter xmin and xmax
-10 10
enter ymin and ymax
-10 10
```


PHASE PORTRAIT
10.0
5.0
0.0
-5.0
-10.0
-10.0
-5.0
0.0
5.0
10.0
i(L2)
v(C1)

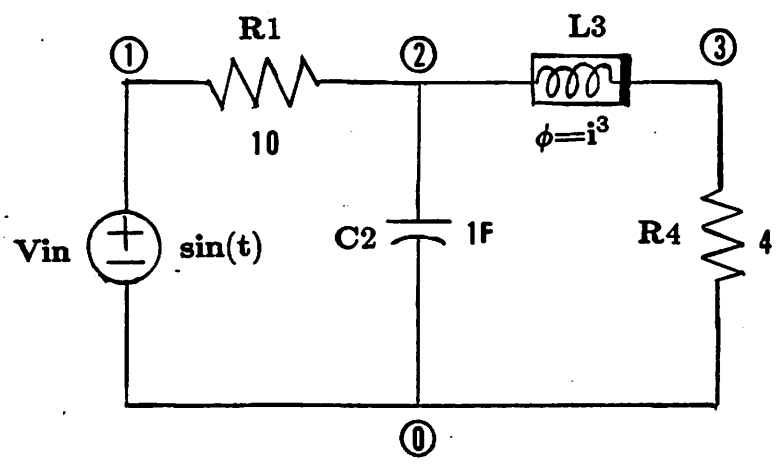


Fig.1

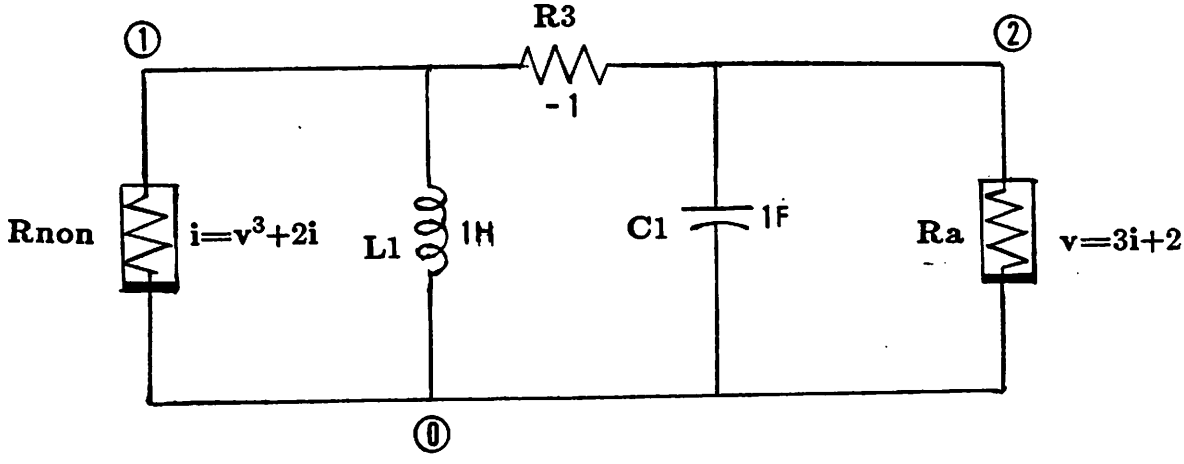


Fig.2

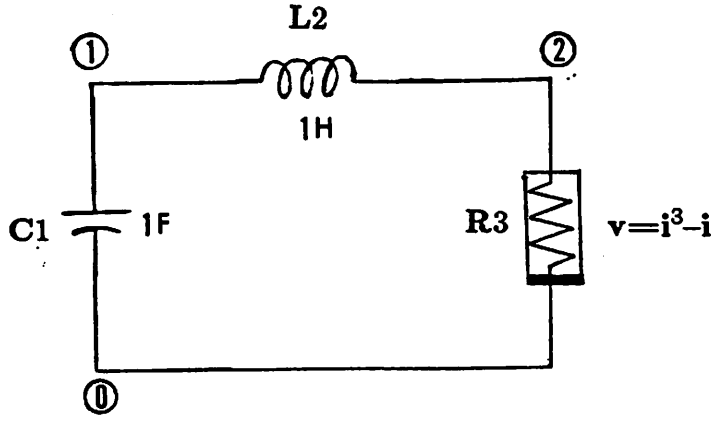


Fig.3

APPENDIX

Source Code Listings

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

int n,k=3,ngrid,reverse;
double h,hi,*x,*t,t0,*c,*z,*y,ft;
double *r,*s,*rx,*sx,*d,*e;
double xmin,xmax,ymin,ymax,*wi;
int *iz,xv,yv;
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];
    int i,j,k,ii,jj;
    FILE *fopen(),*fp;
    GRAF *gp;
    double *xi;

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

    /* get the circuit dimension */
    get_n(&xi);

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

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

    /* choose the phase portrait parameters */
    get_tbl(fp);
    draw_graf(&gp);

    /* the xv-th and yv-th components of x-vector are dynamic elements */
    iz[xv]=1;
    iz[yv]=1;

    for (ii=1;ii<ngrid;ii++)
        for (jj=1;jj<ngrid;jj++)
        {
            /* choose a grid point as the initial condition */
            wi[xv]=xmin+ii*(xmax-xmin)/ngrid;
            wi[yv]=ymin+jj*(ymax-ymin)/ngrid;

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

            /* save the starting point */
```

```
            for (i=0;i<n;i++)
                xi[i]=x[i];

            /* phase portrait */
            for (i=0;i<2;i++)
            {
                /* forward (resp.; backward) transient analysis */
                /* if i=0 (resp.; i=1)                          */
                if (i == 0)
                    reverse = 1;
                else
                    reverse = -1;

                /* reset the initial stepsize and starting condition */
                h=hi;
                t[0]=t0;
                for (k=0;k<n;k++)
                    x[k]=xi[k];

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

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

    graf_close(gp);
    mgideagp();
}

/*****************************************************************************/
/* Get the circuit dimension n.                                              */
/*****************************************************************************/

get_n(xi)
double **xi;
{
    /* dummy variables required in calling var_alloc() */
    double *z2,*z3,*z4,*z5;

    /* call var_alloc in equation file to get the circuit dimension */
    var_alloc(&n,xi,&z2,&z3,&z4,&z5);
    cfree(z2);
    cfree(z3);
    cfree(z4);
    cfree(z5);
}

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

read_bdf()
{
    printf("enter the initial time\n");
```

```
    scanf("%lf",&t0);
    callocd(n*(k+2),&x,"x");
    printf("enter the final time\n");          /* read ft */
    scanf("%lf",&ft);
    printf("enter the initial stepsize\n");    /* read h */
    scanf("%lf",&hi);
    printf("enter the number of grid points in x and y axis\n");
    scanf("%d",&ngrid);
    ngrid++;
}

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

pre_alloc()
{
    callocd(8,&t,"t");
    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] = -reverse*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 = -reverse*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=reverse*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]=reverse*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]=reverse*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 (i>=ix && 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,reverse;
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*reverse*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] = -reverse*x[i]/h;
        z[i]=x[i]+reverse*h;
    }
    ha=reverse/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] = -reverse*(2*x[i+n]-x[i]/2)/h;
        z[i]=2*x[i+n]-x[i];
    }
    ha=1.5*reverse/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] = -reverse*(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*reverse/(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] = -reverse*(-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*reverse/(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] = -reverse*(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*reverse/(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] = -reverse*(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*reverse/(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,reverse;
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;
extern double xmin,xmax,ymin,ymax;
extern int xv,yv;
extern double t0;

/*****************************************************************************/
/* 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(gp)
GRAF *gp;
{
    int ii=0,cg;
    double ratio,ha,pow(),fabs();

    t[k+1]=t[k]+reverse*h;
    if (reverse == 1)
        mgihue(3);
    else
        mgihue(2);
    graf_move(x[xv],x[yv],gp);
    graf_draw(x[xv+n*(k-1)],x[yv+n*(k-1)],gp);

    /* 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 (fabs(t[k+1])<ft && xmin<x[xv+(k-1)*n] && x[xv+(k-1)*n]<xmax
           && ymin<x[yv+(k-1)*n] && x[yv+(k-1)*n]<ymax)
    {
        /* 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) && ii<3)
```

```
        {
            reduce_h(cg,ratio);
            if (cg != -1 && ratio <1.1)
                ii++;
        }
        else
        {
            ii=0;
            /* accept the computed solution and renew the */
            /* timing vector and the solution vector      */
            renew();

            draw_phase(gp);

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

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

        /* renew the coefficients in BDF formula */
        re_form();
    }
}

/*****************************************************************************/
/* 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 */
    phase_plot(x_name,y_name);

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

    strcpy(title,"PHASE PORTRAIT");     /* graphic title */

    mgiclearpln(0,-1,0);
    mgihue(3);
```

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

}

/*****************************************************************************/
/* 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;

    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("y_variable=x[?]\n");
    scanf("%d",&yv);                    /* yv : variable index for y axis */
    strcpy(x_name,var_name[xv]);
    strcpy(y_name,var_name[yv]);
}

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

draw_phase(gp)
GRAF *gp;
{
    graf_move(x[xv+n*(k-1)],x[yv+n*(k-1)],gp);
    graf_draw(z[xv],z[yv],gp);
}
```

```
#include "stdio.h"
#define OVFL 1e6
#define CNVG 1e-7

extern int n;
extern double *x,*t,t0;
extern *iz;
char *var_name[10]; /* string representation for each variable */
int lx=1,outfile=0;
double *f,*jf,*dummy1,*dummy2;
extern double *wi;

/*****************************************************************************/
/* 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,fp)
int argc;
char *argv[];
FILE *fp;
{
    /* perform dc analysis */
    dc_simu(fp);
}

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

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

dc_simu(fp)
FILE *fp;
{
    int iter=1;
    double *dummy3;

    /* allocate spaces for the variables used in */
    /* equation and Jacobian matrix routines     */
    var_alloc(&n,&f,&dummy3,&dummy1,&dummy2,&jf);
    cfree(dummy3);

    /* 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                */

    while (iter==1)       /* while iteration has't converged */
    {
        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("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);
    }
}

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

dc_pt(iter)
int *iter;
{
    int i;

    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];

    *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')
    else*iter=0;
        get_ini_gs();
```

```
}

/*****************************************************************************/
/* 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<CNVG)   /* 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)) > OVFL && 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 l-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=0,k;
    char line[30],*calloc();

    calloci(n,&iz,"iz");
    callocd(n,&wi,"wi");
    while (fgets(line,30,fp)!=NULL)
    {
        k=find_index("=",line);
        var_name[i]=calloc(20,sizeof(char));
        strcpy(var_name[i],line+k+1);
        strdel(var_name[i],strlen(var_name[i])-1,1);
        i++;
    }
}

/*************************************************************************/
/* 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>

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

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);
}

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

```
#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);
    mgicleanpln(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);
}
```